



THESIS

2



This is to certify that the

dissertation entitled

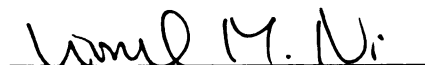
Scalable Data Redistribution Services for
Distributed Memory Machines

presented by

Edgar T. Kalns

has been accepted towards fulfillment
of the requirements for

Doctoral degree in Computer Science


Major professor

Date May 19, 1995

**LIBRARY
Michigan State
University**

PLACE IN RETURN BOX to remove this checkout from your record.
TO AVOID FINES return on or before date due.

DATE DUE	DATE DUE	DATE DUE
OCT 13 1999		

MSU is An Affirmative Action/Equal Opportunity Institution

c:\circ\data\due.pm3-p.1

SCALABLE DATA REDISTRIBUTION
SERVICES FOR DISTRIBUTED-MEMORY
MACHINES

By

Edgar T. Kalns

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1995

ABSTRACT

SCALABLE DATA REDISTRIBUTION SERVICES FOR
DISTRIBUTED-MEMORY MACHINES

By

Edgar T. Kalns

Run-time data redistribution can effect higher algorithm performance in distributed-memory machines. Redistribution of data can be performed between algorithm phases when a different data decomposition is expected to deliver increased performance for a subsequent phase of computation. Data-parallel Fortran languages, *e.g.*, HPF (High Performance Fortran), support run-time data redistribution. Data redistribution can be invoked explicitly with primitives or may occur implicitly. One type of implicit data redistribution occurs when the distribution of actual parameters to a subprogram do not match the distribution of the dummy arguments in the subprogram interface. Typically, with distributed-memory machines, redistribution causes data exchange among processor memories resulting in interprocessor communication overhead. Consequently, there is a performance tradeoff between the higher efficiency of a new data distribution for subsequent computation and the time required to establish it.

This dissertation investigates the pertinent issues that affect the performance of data redistribution on distributed-memory machines, focusing on four primary areas. First, we address the partitioning (or mapping) of data onto processor memories. A technique that facilitates the minimal amount of data exchange among processor memories during redistribution between HPF's regular patterns is proposed. Second, we present the design of a portable and communication-efficient data redistribution library whose implementation is portable among a large class of distributed-memory machines. Portability is enhanced through MPI (Message Passing Interface) communication primitives. Third, we develop a framework for quantifying the scalability of parallel algorithms together with the machines upon which they execute. Fourth, we apply the framework to quantify the scalability of the data redistribution library for a large range of processor configurations and data set sizes on selected distributed-memory machines.

© Copyright 1995 by Edgar T. Kalns
All Rights Reserved

To my parents

ACKNOWLEDGMENTS

My deepest gratitude to my parents whose continual love and encouragement helped me achieve my academic goals. For as long as I can remember, they have told me, and shown me by example, that life's most gratifying accomplishments require industriousness, perseverance, tenacity, and an ability to find enjoyment not only in the achievement itself, but in the progression toward the goal.

To my dissertation advisor, Professor Lionel Ni, I extend my warmest thanks for his guidance, inspiration, and patience. His door was always open to me, his consultation forthcoming whenever I asked for it. With his help, I honed my abilities to think critically as a researcher, and I heightened my satisfaction of pursuing solutions to difficult problems in Computer Science. I will always be indebted to him for his contribution to my professional development. Additionally, I would like to thank the remaining members of my Ph.D. guidance committee: Professors Li, McKinley, Mutka, and Rover for their numerous helpful comments and suggestions.

I would like to thank a number of fellow graduate students with whom I have developed close friendships. I am particularly grateful to Barbara Birchler, who with her love, kindness, and encouragement, helped me immensely. She critiqued my ideas

and proofread my technical reports and dissertation chapters. With her ever-present smile and cheerful nature, she helped me through the inevitable ups and downs. To my “lunch crowd” friends: Reid Baldwin, Barb Birchler, Kevin Bradtke, Eddie Burris, Dave Chesney, Marie-Pierre Dubuisson, Jon Engelsma, Maureen Galsterer, Dan Meyers, Michele Morin, Tim Newman, Ron Sass, Steve Turner, Christian Trefftz, and Steve Walsh I extend my thanks for many fond, and oftentimes boisterous, noon-time get-togethers. A special thanks to Ron Sass, for his careful preparation of the document style used to format this dissertation in compliance with the Graduate School’s regulations.

To Frank Northrup and to all the past and present Computer Science systems administrators with whom I have worked, I extend my gratitude for the camaraderie and team spirit we shared in managing one of the largest computing facilities in the country. I would especially like to thank Frank for his understanding and friendship.

Lastly, I extend my gratitude to the Michigan State University as a whole for providing me the highest quality graduate education. As a Michigan native, I have been extremely fortunate that this institution, attended by students the world over, exists in “my backyard.”

TABLE OF CONTENTS

LIST OF TABLES	x
LIST OF FIGURES	xi
1 Introduction	1
1.1 HPF Example	4
1.2 Motivation	6
1.3 Organization of Dissertation	9
2 Data Decomposition	11
2.1 Static: Align and Distribute	12
2.2 Dynamic: Realign and Redistribute	16
2.3 Example of Redistribution in HPF	18
2.4 Rudimentary Operation of Data Redistribution	20
2.4.1 Determination of Processor Send and Receive Sets	21
2.4.2 Data Exchange with Message-Passing	23
2.5 Implicit Redistribution	25
2.6 Redistribution Optimizations	28
3 Data Mapping Optimization	32
3.1 One-dimensional Logical Processor Mapping	33
3.1.1 Processor Mapping Technique	35
3.1.2 Mapping Function	38
3.1.3 Optimality of Mapping Function	39
3.2 Multidimensional Logical Processor Mapping	46
3.2.1 m -dimensional Redistribution	46
3.2.2 Two-dimensional to One-dimensional Redistribution	48
3.2.3 One-dimensional to Two-dimensional Redistribution	53
4 Data Redistribution Library	59
4.1 Library Design Issues	60
4.1.1 Advantages of Software Libraries	60
4.1.2 Role of HPF Compiler	61
4.1.3 Use of MPI for Libraries	63
4.1.4 Scalability	64
4.2 DaReL Design Overview	65
4.2.1 DaReL Interface	65
4.2.2 Compute Data Exchange Sets (CDES)	69

4.2.3	Partitioning CDES Functionality	72
4.2.4	Exchange Data (ED)	73
4.2.5	Standard and Derived Datatypes	76
4.2.6	Scalability	80
5	Quantifying Scalability	82
5.1	Examples of Ambiguity in Scalability	84
5.2	Goal-directed Scalability Metrics	87
5.3	Metrics for Quantifying Scalability	89
5.3.1	Taxonomy of Scalability Measures	92
5.4	Mathematical Framework	94
5.4.1	Algorithm and Machine Parameterization	94
5.4.2	Fixed-work Problems	95
5.4.3	Scaled-work Problems	98
5.4.4	Deriving AM Pair Parameters	99
5.5	A Three-dimensional Illustration of AM Pair Scalability	100
5.6	Case Study: Matrix Multiplication	104
5.6.1	Algorithm-Machine Pair a_1M	105
5.6.2	Algorithm-Machine Pair a_2M	106
5.6.3	Algorithm-Machine Pair a_3M	108
5.6.4	Computing AM Pair Parameters	108
5.6.5	Scalability Quantification of a_1M , a_2M , and a_3M	113
5.7	Inclusion of Other Scalability Models	115
6	Performance and Scalability	120
6.1	Processor Mapping Technique Analysis	122
6.1.1	Effect of Optimal Mapping on the Programmer	122
6.1.2	Effect of Optimal Mapping on the Compiler	125
6.1.3	Performance Comparison with Traditional Technique	126
6.2	DaReL Performance and Scalability	127
6.2.1	CDES and ED Execution-time Performance	129
6.2.2	Algorithm Choices for ED	134
6.2.3	Scalability	141
7	Conclusion and Future Work	147
7.1	Research Contribution	147
7.2	Future Work	150
	BIBLIOGRAPHY	152

LIST OF TABLES

4.1	MPI Communication Primitive Choices	77
5.1	AM parameters for a_1M , a_2M , and a_3M	112
5.2	$S(\lambda_D)$ measure for a_1M , a_2M , and a_3M	113
6.1	Redistribution test cases for DaReL	129

LIST OF FIGURES

1.1	HPF Linear System Solver	5
2.1	Example of HPF alignment	13
2.2	One- and two-dimensional HPF data distributions.	14
2.3	Example of <code>!HPF\$ REDISTRIBUTE</code>	19
2.4	Redistribution from <code>(*,BLOCK,CYCLIC)</code> to <code>(CYCLIC,CYCLIC,BLOCK)</code>	19
2.5	Collective communication patterns	22
2.6	Realignment of x causes redistribution.	26
2.7	Redistribution at subprogram interface	27
3.1	Data Distribution and Redistribution onto physical nodes.	34
3.2	Logical processor to data mapping alternatives.	35
3.3	Mapping $lpids$ to place holders.	39
3.4	$lpid_i$ is mapped to first element of its data block.	42
3.5	$z \leq p$	43
3.6	$z > p$	45
3.7	<code>(BLOCK₃,BLOCK₄)</code> to <code>(CYCLIC₃(3),CYCLIC₄(2))</code> redistribution.	49
3.8	<code>(BLOCK₃,BLOCK₂)</code> to <code>(CYCLIC₆(2),*)</code> redistribution.	53
3.9	<code>(BLOCK₄,BLOCK₃)</code> to <code>(CYCLIC₁₂(1),*)</code> redistribution.	54
3.10	<code>(BLOCK₆,*)</code> to <code>(CYCLIC₃(2),BLOCK₂)</code> redistribution.	57
3.11	<code>(BLOCK₁₂,*)</code> to <code>(CYCLIC₄,BLOCK₃)</code> redistribution.	58
4.1	Distinct scatter-gather sets.	62
4.2	C-based interface for DaReL	66
4.3	Parameters for the invocation of DaReL	67
4.4	Different shaped process configurations.	68
4.5	CDES example for process $(1,1)$	71
4.6	Partitioning CDES	73
4.7	ED using MPI point-to-point communication	75
4.8	ED using MPI collective communication	76
4.9	Limitation of derived types and collective communication	79
5.1	Comparison of three AM pairs	85
5.2	Scaled work <i>vs.</i> execution time of two AM pairs	87
5.3	Quality of Solution for $\sin(x)$	91
5.4	Execution time <i>vs.</i> number of processors tradeoff	93
5.5	Data distribution of a_1	106

5.6	Algorithm a_1	106
5.7	Data distribution and operation of a_2	107
5.8	Communication pattern of a_2	108
5.9	Algorithm a_2	109
5.10	Data distribution and operation of a_3	109
5.11	Algorithm a_3	110
5.12	Tree-based MPI Reduction used in a_3M	112
5.13	A generic AM pair 3D surface illustration	116
5.14	Scalability metrics illustrated on AM surface	116
5.15	$S(\lambda_D)$ comparison of a_2M and a_3M where $\lambda_D = 20$	117
5.16	$S(\lambda_D)$ comparison of a_1M and a_2M where $\lambda_D = 5$	118
5.17	Incorporating speed metric	119
5.18	Incorporating efficiency metric	119
6.1	Mappings when physical node mapping is <i>static</i>	123
6.2	Mappings when physical node mapping is <i>dynamic</i>	124
6.3	(BLOCK,*) to (CYCLIC(c),*) on 8×1 processors	128
6.4	(BLOCK,BLOCK) to (CYCLIC(c),CYCLIC(c)) on 4×3 processors	128
6.5	(BLOCK,BLOCK) to (CYCLIC(c),CYCLIC(c)) on 6×4 processors	128
6.6	CDES performance on 20 processor configurations	130
6.7	ED performance on 20 processor configurations	131
6.8	Local data set for (BLOCK,BLOCK) to (CYCLIC,CYCLIC)	133
6.9	Local data set for (BLOCK,CYCLIC) to (CYCLIC,BLOCK)	134
6.10	CDES performance on 100 processor configurations	135
6.11	ED performance on 100 processor configurations	135
6.12	(BLOCK,BLOCK) to (CYCLIC,CYCLIC) redistributions	137
6.13	Communication paradigm using Scatter (top) and Gather (bottom)	141
6.14	1000×1000 redistribution with	144
6.15	DaReL scalability measured by $NegSlope(N, N')$	146

Chapter 1

Introduction

Only in the past several years, Scalable Parallel Computers (SPCs) have come to be viewed not only as indispensable for large-scale engineering and scientific applications, but also as viable platforms for commercial and financial applications. Many SPCs, such as the IBM Scalable POWERparallel (SP-1 and SP-2), Intel Paragon, Cray T3D, nCUBE-2/3, and Networks of Workstations (NoW), have demonstrated *scalable* performance in these domains. Distributed-memory machines comprise an ensemble of nodes where each node consists of a processor, local memory, and other supporting devices. The nodes of the machine are interconnected by a point-to-point (direct) or switch-based (indirect) network. These distributed-memory systems offer proportional performance increases as the processor configuration size, network and I/O bandwidth, and memory capacity and bandwidth are increased without changing the basic machine architecture.

In order to fully utilize SPCs, the application software must also be *scalable* to exploit increased computing capacity as it becomes available with larger machine

configurations. Programming SPCs has been a major challenge impeding the greater success of such systems. Unfortunately, the traditional message-passing programming paradigm for these machines based on separate name spaces is tedious, time consuming, and error-prone for programmers. In contrast, the data-parallel or SPMD programming model provides an easier and more familiar programming style for users. The SPMD model is based on a single name space and loosely synchronous parallel computation with a distinct data set for each processor. In order to provide high-level language support for data-parallel programming, several data-parallel Fortran languages have been proposed, such as Fortran D [1] and Vienna Fortran [2]. The High Performance Fortran Forum, composed of over forty academic, industrial, and governmental agencies, has developed HPF (High Performance Fortran) [3] in an effort to standardize data-parallel Fortran programming for distributed-memory machines. Many of the concepts originally proposed in Fortran D, Vienna Fortran, and other data-parallel Fortran languages have been incorporated in HPF.¹ Dataparallel C [4], and pC++ [5] represent language design efforts focusing on C-based data-parallel extensions.

An essential part of HPF is the specification of the decomposition of data arrays through compiler directives. Due to the non-uniform memory access times characteristic of distributed-memory machines, determining an appropriate data decomposition is critical to the performance of data-parallel programs on these machines. The data decomposition problem involves *data distribution*, which deals with how data arrays

¹The research contained in this dissertation is presented in the context of HPF due to its emerging acceptance as a standard. With minor syntactic changes, the concepts proposed herein could be applied to other data-parallel Fortran languages as well.

should be distributed among processor memories, and *data alignment*, which specifies the collocation of data arrays. The goal of data decomposition is to maximize system performance by balancing the computational load among the processors and by minimizing remote memory accesses (or communication messages). To facilitate data distribution onto an application-dependent, rather than a machine-dependent, topology, an HPF programmer declares a multi-dimensional mesh of *logical* (or virtual) processors. The mapping of the logical processors to the physical machine is determined by the compiler or run-time system.

HPF provides flexible primitives for specifying data decompositions; however, it does not provide the programmer any guidance in selecting a suitable data decomposition. Given the large number of distribution and alignment possibilities, it is inherently difficult for programmers to select an appropriate decomposition which maximizes program performance. Mace [6] proved that determining an optimal data distribution, even for one- and two-dimensional arrays is NP-complete. Li and Chen [7] proved that finding a set of alignments for the indices of multiple program arrays that minimizes data movement among the processors is also NP-complete. A number of heuristics for determining suitable distributions and alignments have been proposed in the literature.

Assuming a viable technique for selecting an appropriate data decomposition, the best possible performance may not be achieved with only a single data distribution or alignment for the entire program. Depending upon the algorithm and its data-parallel implementation, a particular data decomposition that is well-suited for one phase of an algorithm may not be good, in terms of performance, for a subsequent

phase. Therefore, many data-parallel languages define mechanisms for explicit run-time data *redistribution* and *realignment*.

1.1 HPF Example

We illustrate the above data decomposition constructs in a small HPF code example for solving a linear system of equations. Figure 1.1 illustrates an HPF program for solving the system $Ax = b$ using Gaussian elimination and subsequent backward substitution [8]. The basic idea of the algorithm is to reduce the matrix A to upper triangular form in the first phase and then to use backward substitution to diagonalize the matrix. In the program, A is declared as an $n \times (n + 1)$ array, where the $(n + 1)$ st column of A stores the constant vector b . Following the array declarations are a set of compiler directives, including alignment and distribution primitives described earlier. Statements s7-s8 demonstrate the alignment of two different data vectors to the rows of A , while statement s9 shows the alignment of another array to the $(n + 1)$ st column of A . Statement s10 declares a vector of n logical processors. The matrix, along with its aligned data, is distributed by contiguous rows in statement s11 onto the logical processor array. Statement s27 specifies a run-time *redistribution* of data between the Gaussian elimination phase (s13-s26) and the backward substitution phase (s28-s33). Note that this code segment merely illustrates HPF constructs and is by no means the most efficient algorithm.

```

s1:    PROGRAM LINSYS(A,x,n)
s2:    REAL INTENT (IN) :: A(:, :)
s3:    REAL INTENT (OUT) :: x(:)
s4:    INTEGER INTENT (IN) :: n
s5:    INTEGER xindx(n)
s6:    REAL intrm(n), temp(n)
s7:    !HPF$ ALIGN x(:) WITH A(*, :)
s8:    !HPF$ ALIGN xindx(:) WITH A(*, :)
s9:    !HPF$ ALIGN intrm(:) WITH A(:, n+1)
s10:   !HPF$ PROCESSORS P(n)
s11:   !HPF$ DYNAMIC, DISTRIBUTE A(BLOCK, *) ONTO P
s12:   FORALL (i = 1:n) xindx(i) = i
s13:   DO i = 1, n
s14:       maxloc = MAXLOC(A(i, i:n))
s15:       maxval = A(i, maxloc)
s16:       temp = A(:, maxloc)
s17:       A(:, maxloc) = A(:, i)
s18:       A(:, i) = temp
s19:       tempx = xindx(maxloc)
s20:       xindx(maxloc) = xindx(i)
s21:       xindx(i) = tempx
s22:       A(i, i:n+1) = A(i, i:n+1) / maxval
s23:   !HPF$ INDEPENDENT (j, k)
s24:       FORALL (j = i+1:n, k = i+1:n+1)
s25:           A(j, k) = A(j, k) - A(j, i) * A(i, k)
s26:   END DO
s27:   !HPF$ REDISTRIBUTE A(CYCLIC, *)
s28:   intrm(n) = 0
s29:   DO i=n, 1, -1
s30:       x(xindx(i)) = (A(i, n+1) - intrm(i)) / A(i, i)
s31:       FORALL (j = i-1:1:-1)
s32:           intrm(j) = intrm(j) + (A(j, i) * x(xindx(i)))
s33:   END DO

```

Figure 1.1: HPF Linear System Solver

1.2 Motivation

Two-dimensional and three-dimensional FFT (Fast Fourier Transform) [9] and ADI (Alternating Direction Implicit method) [10] are frequently cited examples for which efficient data redistribution between algorithm phases results in increased performance. Use of redistribution mechanisms, however, involves communicating program data arrays among the nodes of the machine resulting in interprocessor communication overhead. Data redistribution may require a temporary interruption of application-useful computation while processors await their new data. Consequently, there is a performance tradeoff between the higher efficiency of a new data decomposition for a subsequent algorithm phase and the time required to establish it.

Many researchers have espoused the necessity of incorporating a redistribution capability into Fortran- and C-based data-parallel languages. The Kali language [11] was one of the first to incorporate run-time data redistribution mechanisms. DINO [12] addresses the implicit redistribution of data at procedure boundaries. The Hypertasking compiler [13] for data-parallel C programs incorporated a run-time redistribution facility. Both Vienna Fortran [14] and Fortran D [15] specify data redistribution primitives as well.

While data-parallel languages provide run-time primitives that facilitate an *explicit* change in data decomposition, *implicit* data redistribution is possible as well. Array assignment statements, subprogram boundaries, and array realignment are implicit sources of redistribution in data-parallel programs. For instance, consider two arrays: **A** and **B** whose distributions are different. The assignment statement **A=B**

causes an implicit redistribution of array **B** to match the distribution of array **A**. At subprogram boundaries, data redistribution occurs when the distribution of actual parameters to a subprogram do not match the distribution of the dummy arguments in the subprogram interface. Finally, realignment of data can require the redistribution of data to satisfy the chosen new alignment.

Many data-parallel languages define dozens of language intrinsics that could be implemented as software libraries, significantly simplifying the role of the language compiler. Data redistribution is one type of operation whose implementation can be provided as a software library. Providing a rich set of libraries to data-parallel programmers is becoming increasingly popular because libraries offer some unique advantages [8]. For instance,

- A uniform interface provided by the library enhances the portability of programs among various distributed-memory architectures.
- Although a library provides a uniform interface to programs, the design and implementation of the library can explicitly exploit machine specific features to provide better performance.
- The program development cycle for the consumer of the library can be shortened as the correctness of libraries can be independently verified.
- The existence of libraries can simplify the tasks that the data-parallel compilers would otherwise have to perform.

As there are multiple sources of redistribution in a data-parallel program, even when it is not explicitly invoked by the programmer, we contend that research in the area of data redistribution in distributed-memory machines has particular merit. Changing array data decompositions within a program may yield substantial performance gains, but only if the overhead of such operations is mitigated. Therefore, the

efficient and *scalable* implementation of data redistribution mechanisms is important to the overall performance of data-parallel programs on SPC architectures.

The term *scalability* has been used extensively in the parallel processing community to characterize the ability of parallel architectures and algorithms to exhibit greater *performance* as more processors are employed to solve a problem. However, the term “performance” is equivocal as it may be used to identify widely varying algorithmic or architectural properties: speedup, execution time, processor speed or efficiency, or the quality (accuracy) of a solution. Many of these properties have been used either separately or in conjunction to describe the scalability of parallel algorithms and architectures. The ambiguity regarding scalability leads us to ask several questions: How are the relative scalabilities of parallel algorithms and machines related? Can a universal definition of scalability be applicable to all scenarios? How can scalability be *quantified*? If scalability can be quantified, which property best describes it, or is scalability a combination of factors? Answers to these questions are relevant to providing scalable redistribution methods.

Efficient and scalable data redistribution necessitates consideration of many issues. First, redistribution is a communication-dominant task, thus, the efficiency of the communication mechanisms used is of great importance. For instance, the relative merits of point-to-point or collective communication for redistribution must be considered along with the tradeoffs of using blocking or non-blocking message-passing. Second, the amount of data communicated between processors during redistribution is a significant factor in the total execution time of the operation. Consequently, methods for reducing the amount of data that must be exchanged via message-passing

may benefit redistribution performance. Third, to be viable to potential data-parallel programmers and compilers alike, efficient redistribution must consider the obtained performance as data and processor configuration sizes as scaled. *Quantification* of redistribution scalability using relevant performance metrics facilitates comparison of different techniques. Fourth, to the extent possible, efficient redistribution mechanisms should be portable among distributed-memory machines to enhance their utility in today’s heterogeneous high-performance computing environments.

Problem Statement

The efficient operation of implicit and explicit data redistribution on distributed-memory architectures can greatly impact the overall performance of data-parallel programs. This dissertation investigates language, data mapping, message-passing, and scalability issues, all of which affect the performance of data redistribution. This dissertation is distinguished as the earliest known research to propose a portable and scalable redistribution library, processor-data mapping techniques for optimizing data exchange, and a framework for quantifying the scalability of data redistribution on SPC platforms.

1.3 Organization of Dissertation

Chapter 2 discusses static (distribute and align) and dynamic (redistribute and realign) data decomposition. Salient data redistribution issues are discussed and related work is summarized. Chapter 3 presents a technique for partitioning (or mapping)

data onto processor memories that facilitates the *minimal* amount of data exchange among processor memories during redistribution between a large class of regular HPF distribution patterns. The technique is independent of the underlying machine architecture, and thus it is portable among distributed-memory platforms. We prove the optimality of the technique with respect to minimizing the amount of data exchanged. Chapter 4 proposes a data redistribution library, DaReL, for distributed-memory machines that utilizes the emerging message-passing standard, MPI [16]. The library supports multi-dimensional data redistribution among arbitrary regular HPF distribution patterns. A uniform interface to DaReL is defined, facilitating its potential incorporation into a data-parallel compiler. Chapter 5 proposes a framework for *quantifying* scalability that incorporates end-user requirements to enable quantification of the scalability of parallel algorithm-machine combinations, or pairs. We illustrate the framework with a matrix multiplication case study to assess the relative performance of several different algorithm-machine pairs. Chapter 6 presents performance and scalability results. We demonstrate redistribution performance improvements of up to 40% using the processor mapping technique (Chapter 3) on a distributed-memory machine, an IBM SP-*x*.² We address the impacts of the mapping technique on the data-parallel programmer and compiler, respectively. We extend the scalability framework (Chapter 5) to quantify DaReL's performance as data and processor configuration sizes are scaled. Chapter 7 concludes the dissertation with a summary of the major contributions and suggests potential future research directions.

²SP-*x* denotes an IBM SP multicomputer consisting of SP-1 processors and an SP-2 network switch.

Chapter 2

Data Decomposition

Due to their non-uniform memory access times, determining an appropriate data decomposition among different memories is critical to the performance of data-parallel programs on distributed-memory machines. The goal of data decomposition is to maximize system performance by balancing the computational load among the processors and by minimizing remote memory accesses. Data-parallel languages, *e.g.*, HPF, support compiler directives that statically specify the decomposition of data to processor memories prior to program execution. Additionally, these languages support modification of data decompositions during run-time in favor of a different one that, presumably, is better suited to the ensuing computation. Existing data-parallel Fortran languages, however, provide no inherent capability for determining static data placement nor run-time modification of data decomposition. This chapter surveys research in the area of data decomposition with particular emphasis on data redistribution issues.

2.1 Static: Align and Distribute

In HPF, data arrays are *aligned* relative to one another and then, as a group, they are *distributed* onto an abstract, or *logical*, processor configuration¹. This process of static data decomposition is accomplished with the `!HPF$ ALIGN`, `!HPF$ DISTRIBUTE`, and `!HPF$ PROCESSORS` compiler directives. These directives do not affect the result of the program, rather they suggest an implementation strategy to the compiler. Therefore, these directives may only appear in the declaration part of an HPF program.

The `!HPF$ ALIGN` directive specifies the collocation of data array elements from distinct arrays. Figure 2.1 illustrates a possible alignment of an array, \mathbf{x} to the m -th column of an $n \times m$ matrix \mathbf{A} . For aligning multiple arrays to one another, it is sometimes convenient to define an HPF *template*. The `!HPF$ PROCESSORS` directive defines a processor configuration as a multi-dimensional mesh of processors. Data arrays, or templates, are mapped onto a processor configuration with the `!HPF$ DISTRIBUTE` directive. The programmer applies an HPF distribution *pattern* to each dimension of a data array specifying the mapping of the data dimension onto a dimension of the processor configuration. The number of data dimensions must equal the number of processor dimensions. `BLOCK`, `CYCLIC`, `BLOCK(b)`, and `CYCLIC(c)` comprise the set of regular data distribution patterns in HPF; `*` denotes the absence of a distribution pattern for a dimension, *i.e.*, the entire dimension is allocated to the processor. [17] discusses possible HPF language extensions to permit user-defined array distribu-

¹The mapping of logical processors onto the physical nodes of the machine is not within the scope of HPF. Henceforth, we assume *logical* processor when we write processor, unless otherwise stated.

tion patterns.² With these mechanisms, the program data is aligned and distributed among the processors. By HPF's *owner-computes* rule, the processor that stores the data element on the left-hand side of a program statement is responsible for computing its update. Thus, the distribution of computation is determined by the decomposition of data.

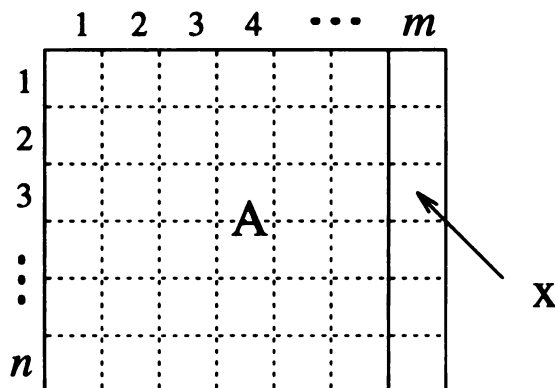


Figure 2.1: Example of HPF alignment

With the **BLOCK** specification, contiguous blocks of an array are distributed to each processor. If n is the data size along a given dimension and p denotes the number of processors, the block size is $\lceil \frac{n}{p} \rceil$ for the first $p - 1$ processors, the last processor receiving the residual if it does not receive a full block. With the **CYCLIC** distribution, elements of a dimension of an array are assigned to each processor in a round-robin fashion. An extension of **BLOCK** and **CYCLIC** are the **BLOCK**(b) and **CYCLIC**(c) distribution patterns. With **BLOCK**(b), each processor is assigned a data segment of size b , with the restriction that $n \leq p \times b$, while **CYCLIC**(c) distributes round-robin contiguous blocks of size c to each processor. By definition, **CYCLIC** is equivalent to **CYCLIC**(1). **BLOCK**(b) and **CYCLIC**(c) imply the same distribution when

²This dissertation focuses on regular distribution patterns as defined in the original language specification.

$b = c$ and $n \leq p \times b$; however, **BLOCK**(b) additionally asserts that there is no wrap-around. The potential round-robin effect, when $n > p \times c$, of **CYCLIC**(c) may be difficult to discern at compile-time if c is an expression.

Figure 2.2 illustrates various data distribution examples: (a) and (b) show an eight-element vector distributed **BLOCK** and **CYCLIC**, respectively, on four processors; (c) illustrates **CYCLIC**(c) where $c = 2$ on two processors; (d), (e), and (f) show various possible distributions for an 8×8 data matrix. The distributions for (d)-(f) are shown as an ordered pair: the first pattern applies to the row dimension while the second pattern applies to the column dimension. The processor IDs, indicating data ownership, are shown in italics: in (a)-(c) they are superimposed over the data, while in (d)-(f) they are shown as Cartesian coordinates. Processor IDs are numbered beginning with 0. Recall that, the * denotes the absence of a distribution pattern for a dimension. The row dimension is not distributed in (f).

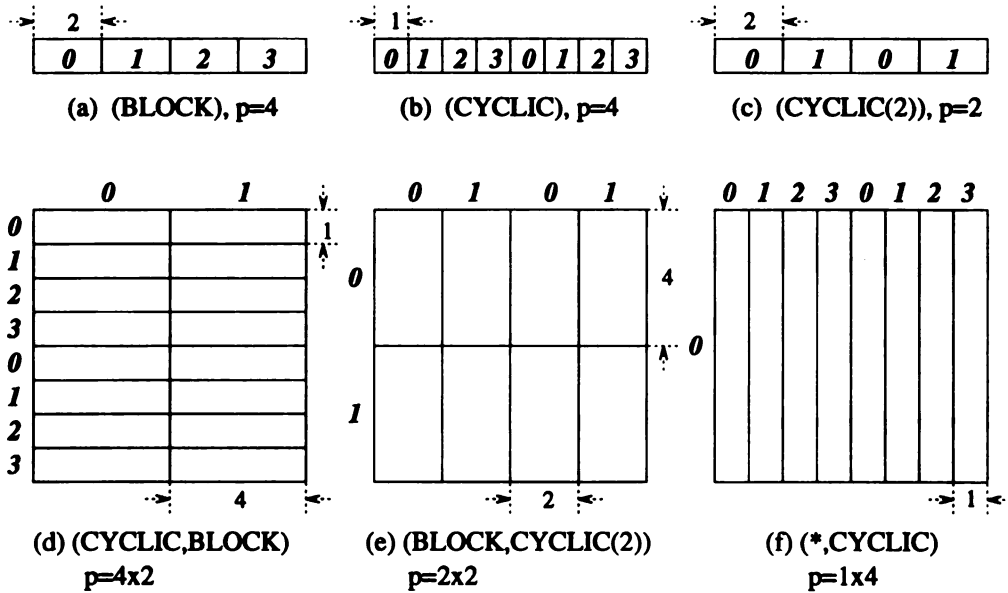


Figure 2.2: One- and two-dimensional HPF data distributions.

Figure 2.2 shows only a few of the many possible distributions for the one- and two-dimensional data. Recall that depending upon the application algorithm, the choice of data distribution will affect the computational load balance and the size and number of remote memory accesses when implemented on a distributed-memory machine. Selecting the optimal distribution is NP-complete as stated in Chapter 1. The situation is further complicated when one considers the possible data array alignments. Automatically determining array alignments is an active area of research [18]. Several heuristics have been proposed to automatically determine an appropriate data decomposition based on the application code. A survey of some of these heuristics and their limitations and drawbacks can be found in [19]. Anderson and Lam [20] present a mathematical framework for systematically determining data decompositions. A first-order model for reducing the search space of possible distributions to a select few based on architecture-specific communication and computation parameters can be found in [21]. Xu and Ni [22] propose low-complexity algorithms for determining array alignments in an effort to maximize processor load balance while minimizing interprocessor communication. Ponnusamy, *et al.* [23] focus on automatic selection of data distributions for irregular problems, *i.e.*, programs for which the data access patterns are input-dependent. Lee and Tsai [24] propose a dynamic programming technique for determining data distributions.

The HPF programmer specifies an initial distribution of data onto a set of logical processors as discussed previously. The data mapping, from the programmer's perspective, is independent of the underlying machine topology and physical processor IDs. The mapping of the logical processor IDs to the physical nodes of the machine

is relegated to the compiler. To facilitate efficient program execution, the logical to physical processor mapping must consider the machine topology, communication subsystem, memory capacity, and number of nodes [25]. Chittor and Enbody [26, 27] investigate logical to physical processor mapping for wormhole-routed architectures.

2.2 Dynamic: Realign and Redistribute

Run-time data redistribution has been shown to improve the performance of FFT and ADI. HPF defines run-time primitives for changing data alignment (`!HPF$ REALIGN`) and distribution (`!HPF$ REDISTRIBUTE`). These constructs may be inserted anywhere in the executable portion of a program to change the alignment or distribution of the global data arrays or templates. Henceforth, we shall refer to the aggregate program data as the *global* data and the data owned by each processor, from the perspective of the processor, as its *local* data.

To determine the appropriateness of utilizing these run-time primitives, an HPF programmer must consider the tradeoff between the anticipated increased performance of a new distribution or alignment and the cost of changing the data decomposition which typically involves interprocessor communication. Given the difficulty of estimating the execution time of a phase of an algorithm with a particular decomposition, the effective *use* of realignment and redistribution is an important research issue. Chatterjee, *et al.* [28] propose a model for assessing realignment cost, and they formulate the alignment problem as an optimization of realignment cost. Algorithms for automatically determining good *mobile* alignments, *i.e.*, alignments which are a

function of the loop induction variable, are presented. Kunchithapadam and Miller [29] present a graph-coloring technique for recording data movement among nodes during program execution. They propose a simple algorithm to optimize the graph coloring to describe new distributions which would result in reduced communication. Thus, the technique identifies places within a program's execution where redistribution can effect greater program performance.

In contrast to HPF's approach to *user*-specified static and dynamic data decompositions, techniques for automatic data decomposition are proposed in [20]. The compiler, rather than the programmer, determines the decomposition of data for each program loop. Redistribution is viewed as an implicit activity to be undertaken when the decomposition of an array in a loop differs from the decomposition of the same array in another (subsequent) loop. They propose a graph model, where nodes represent loops and edges represent redistribution communication costs between loops. A greedy algorithm for eliminating edges with large weights, thereby eliminating costly redistributions, is presented. The elimination of the redistributions deemed to be too costly, however, is achieved at the cost of reduced parallelism in the resulting code. Chase and Reeves [30] propose a similar approach for automatically performing array redistribution. Arrays are grouped into *compatibility classes*, *i.e.*, aligned together and distributed. The compiler inserts assertions about the classes into the code, and these assertions are checked during run-time to discern whether a redistribution is necessary. Should a compatibility relationship change, redistribution is performed. Unfortunately, the cost/benefit tradeoff of redistribution is not addressed.

Many researchers have espoused the necessity of incorporating a redistribution

capability into data-parallel languages. The Kali language [11] was one of the first to incorporate run-time data redistribution mechanisms. DINO [12] addresses the implicit redistribution of data at procedure boundaries. Hall *et al.* [15] discuss global optimizations that can be employed to a set of redistribution calls in a Fortran D program. The Hypertasking compiler [13] for data-parallel C programs incorporated a run-time redistribution facility. To avoid packing and unpacking of data and ownership calculation in some instances, the Hypertasking compiler performs data redistribution by moving an entire data set around to all processors, each processor removing its portion. Chapman *et al.* [14] introduce Vienna Fortran's dynamic data distribution capability and discuss the high-level implementation of it in the Vienna Fortran Engine (VFE).

2.3 Example of Redistribution in HPF

Figure 2.3 presents a segment of HPF code illustrating the use of data redistribution in HPF. A $20 \times 12 \times 10$ array of real numbers, **A**, is initially distributed onto a three-dimensional processor configuration, **P**, with distribution patterns *****, **BLOCK**, and **CYCLIC** applied to the three dimensions, respectively. **P** contains forty processors and each processor owns $\frac{20 \times 12 \times 10}{40} = 60$ data elements. Following some amount of computation, **A** is redistributed with a new set of patterns, **CYCLIC**, **CYCLIC**, **BLOCK**, onto a different shape logical processor configuration, **Q**. Figure 2.4 illustrates the initial distribution of **A** (left) and its subsequent redistribution (right). The shaded portions of the figure depict the data elements of **A** owned by processor $(0,0,0)$.

Whereas $(0,0,0)$ owns data in globally contiguous locations initially, it owns data in globally *non*-contiguous locations following redistribution. Note that processor configuration Q also consists of forty processors; however, P and Q vary in shape.

```

REAL, DIMENSION(20,12,10) :: A
!HPF$ PROCESSORS P(4,1,10)
!HPF$ PROCESSORS Q(5,4,2)
!HPF$ DISTRIBUTE A (BLOCK,*,CYCLIC) ONTO P
.
.      (computation)
.
!HPF$ REDISTRIBUTE A (CYCLIC,CYCLIC,BLOCK) ONTO Q
.
.      (computation)
.

```

Figure 2.3: Example of !HPF\$ REDISTRIBUTE.

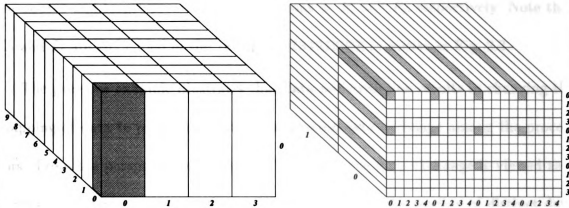


Figure 2.4: Redistribution from $(*,\text{BLOCK},\text{CYCLIC})$ to $(\text{CYCLIC},\text{CYCLIC},\text{BLOCK})$

The example illustrates one redistribution possibility; many more distinct combinations of source and destination distribution patterns are possible. The HPF programmer has great flexibility in declaring data and processor configurations of arbitrary dimension. Recall, however, that HPF restricts the programmer to have the same dimensionality for the data, processor, and distribution pattern declara-

tions, *e.g.*, in Fig. 2.3, the data, processor, and pattern declarations are all three-dimensional. HPF does not require that the source and target processor configurations have equal size.

2.4 Rudimentary Operation of Data Redistribution

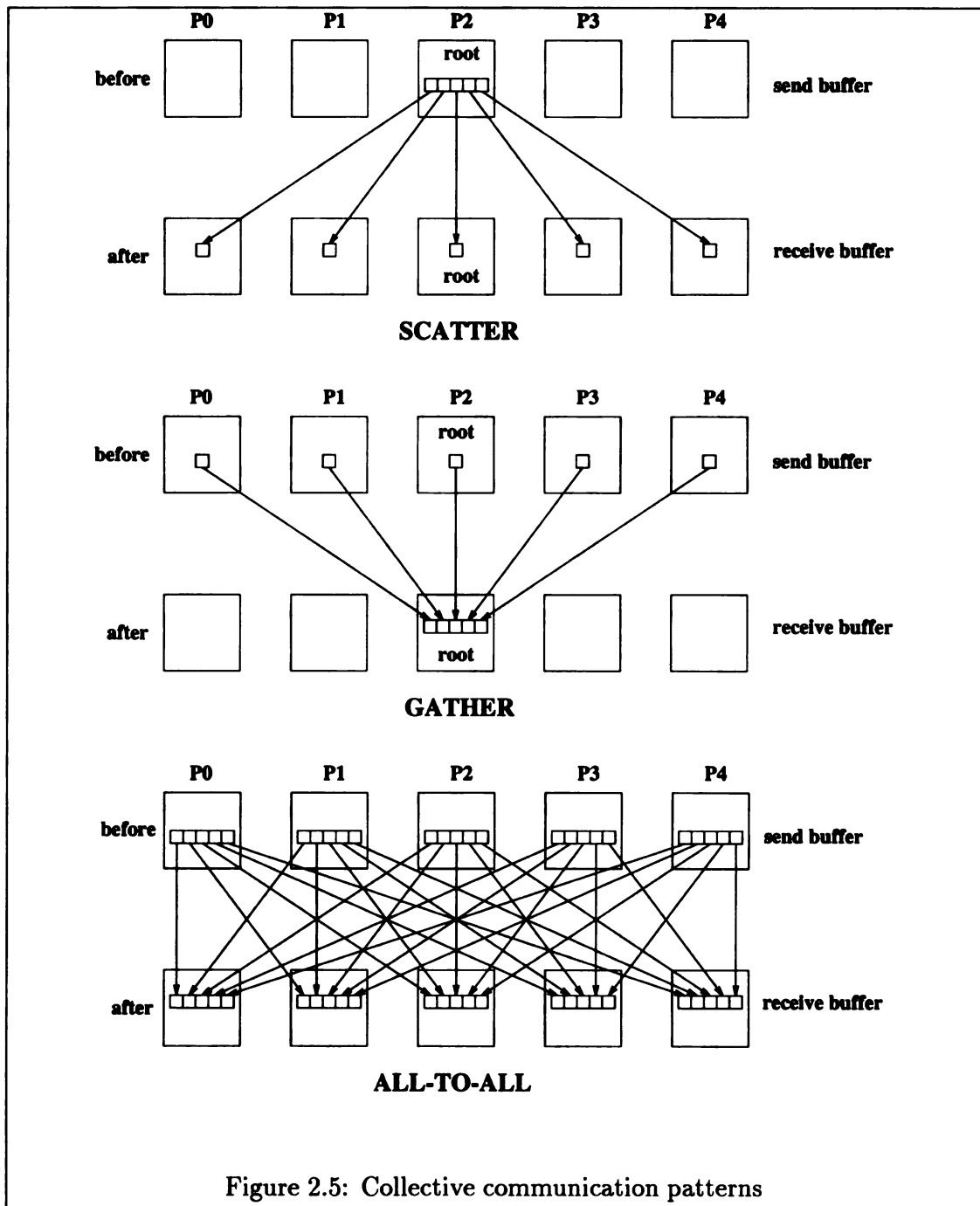
Henceforth, we denote the initial (source) distribution pattern(s) of a program as D_s , and the destination (target) distribution pattern(s) as D_t . Similarly, we shall refer to the source and target processor configurations as P_s and P_t , respectively. Note that D_t and P_t are embedded in the `!HPF$ REDISTRIBUTE` primitive in Fig. 2.3.

The previous example illustrates that data redistribution results in changing the mapping of data to processors, thus necessitating data exchange among the processors. From the perspective of a sending processor, the data exchange is viewed as a *scattering* of data elements to the processors in P_t , *i.e.*, the sending of distinct data elements to each processor in the target processor configuration. This paradigm is illustrated in the top portion of Fig. 2.5. From the perspective of a receiving processor, the data exchange is viewed as a *gathering* of data elements from the processors in P_s , *i.e.*, the receiving of distinct data elements from each processor in the source processor configuration. This paradigm is illustrated in the middle portion of Fig. 2.5. When $P_s = P_t$, data exchange is viewed as an *all-to-all personalized* communication as illustrated in the bottom portion of Fig. 2.5.

In order to perform data redistribution between D_s and D_t , a processor must determine the identity of the processors from which it is to receive data as well as the identity of processors to which it must send data. Once these sets are computed, processors exchange their data. Recall that we have defined redistribution between logical processors; therefore, the logical to physical processor mapping determines the actual data movement among the memories of the machine. For example, logical processors mapped to the same physical node would obviously not require message-passing to exchange data.

2.4.1 Determination of Processor Send and Receive Sets

Several research efforts have focused on efficient methods for determining the send and receive processor sets for redistribution. Gupta *et al.* [31] derive closed form expressions for these communication sets based on Fortran D's (BLOCK), (CYCLIC), and (BLOCK-CYCLIC) distribution patterns. In this approach, global array (or template) indices combined with knowledge of D_s and D_t are used to compute the send and receive sets. An alternative approach [32] is for each node to scan its local data array once, determine the destination processor for each element and place the element in a message packet bound for that processor. In [32], one-dimensional data redistribution is performed by distinct algorithms for different combinations of distribution patterns for D_s and D_t . Distinct algorithms are proposed in order to introduce optimizations that apply only to the given case. Multi-dimensional redistributions are implemented as a series of (sequential) one-dimensional redistributions. This approach to multidimensional



mensional redistributions is unnecessarily costly and does not scale well. Stichnoth *et al.* [33] propose methods for computing ownership sets for array assignment statements. Due to the similarity of determining send and receive sets, they advocate computing these together on the sending processor and communicating the information together with the data to a receiver. While this approach is chiefly intended for communicating right-hand side operands, it can be incorporated into data redistribution. Ramaswamy and Banerjee [34] propose a mathematical representation for regular distributions called PITFALLS, which facilitates determining the processor sets for data redistribution. PITFALLS robustly handles arbitrary source and target processor configurations and arbitrary number of data array dimensions in a scalable manner. PITFALLS is being developed for inclusion in the PARADIGM [35] compiler project at the University of Illinois. The research presented in [31, 32, 33, 34] focus on the efficiency of computing send/receive processor sets, excluding the actual data exchange portion of redistribution which can be several orders of magnitude more costly, in terms of execution time, than send and receive set determination; see Chapter 6.

2.4.2 Data Exchange with Message-Passing

Data exchange on a distributed-memory machine is performed with either point-to-point or collective communication message-passing primitives. In an effort to standardize message-passing primitives on distributed-memory platforms, the MPFI (Message Passing Interface Forum) [16] was founded. MPI defines a host of point-

to-point and collective primitives. Collective communication primitives such as `MPI_Scatter`, `MPI_Gather`, or `MPI_Alltoall` may achieve lower communication latencies and less network contention than straightforward point-to-point routines if implemented to effect parallelization in communication. Techniques for efficient multiple scatter and gather operations are presented in [36]. To achieve a higher performance, the scatter and gather operations may be optimized for the specific architecture. Portability of a redistribution implementation that utilizes MPI primitives would be ensured among platforms which conform to the MPI standard. A data redistribution library utilizing MPI is described in Chapter 4. McKinley *et al.* [37] survey issues related to efficient collective communication in wormhole-routed machines. Of particular relevance to data redistribution, they review techniques for all-to-all personalized communication.

The efficiency of the simultaneous redistribution of data among physical processors is affected by the topology, routing, and switching mechanisms of the underlying machine. The routing mechanism, such as dimension-order routing [38, 39] for hypercube and mesh-connected machines, affects communication latency. A technique for communication-efficient data redistribution which addresses message contention for certain topologies is presented in [40]. The authors propose a data redistribution communication cost model which parameterizes the number of messages and their sizes. Network contention is modeled by expressing the communication as a sequence of permutations which may be executed in a fixed number of (contention-free) steps. *Multi-phase* redistribution is defined as redistributing data to intermediate distribution patterns, eventually arriving at the destination distribution. The model is

used in conjunction with multi-phase redistribution to show that lower overall cost can be achieved as compared to single-phase redistribution. Redistributing perfect power-of-two sized arrays on hypercubes is discussed in [41].

As static distribution of a data array or template causes the distribution of all aligned data; redistribution of those arrays or templates causes the redistribution of aligned data as well. Depending upon the data alignment(s), there may be more data distributed to some processors than others. Thus, during redistribution, the amount of data to be exchanged among processor memories could vary significantly, resulting in an unbalanced communication load. This situation could increase the time of the redistribution operation for some physical nodes.

2.5 Implicit Redistribution

While the example in Figs. 2.3 and 2.4 illustrates explicit run-time redistribution of data arrays, other program statements or HPF constructs can cause the *implicit* redistribution of data. For instance, array assignment statements, `!HPF$ REALIGN`, and subprogram interfaces may cause implicit data redistribution in data-parallel programs.

Consider two arrays **A** and **B** that are distributed **BLOCK** and **CYCLIC**, respectively. The assignment statement **A=B** causes an implicit redistribution of array **B** to the distribution of array **A**. While such an assignment is not the same as a redistribution of **B**, the former operation must perform the same functions as the latter, *e.g.*, computing processor sets and exchange of data. This type of redistribution may occur frequently

in data-parallel programs with many arrays that have dissimilar distributions.

Figure 2.6 illustrates how realignment can cause implicit data redistribution. Initially, x is aligned to the last column of A ; then it is shown realigned to the first row of A . With the (BLOCK,*) distribution, this would cause the implicit redistribution of a portion of x from processors 1, 2, and 3 to processor 0. Processor 0 must remap its own local data set to conform to the new alignment. Realignment, like static alignment, may cause an unbalanced communication load since processors may not own equal amounts of data.

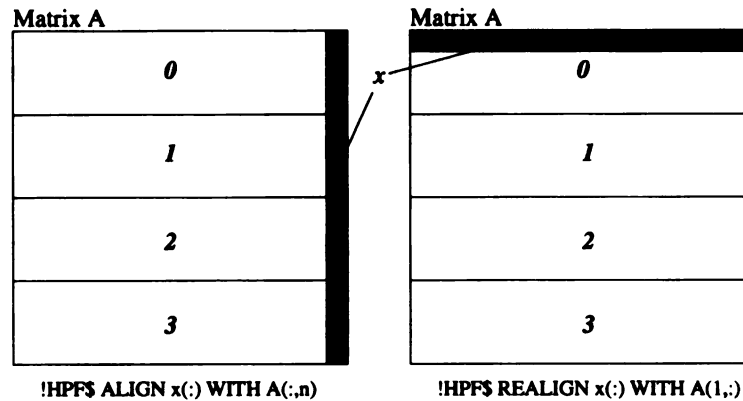


Figure 2.6: Realignment of x causes redistribution.

Implicit redistribution can occur at subprogram (subroutine) boundaries. If a distribute or align directive is applied to a dummy argument specified in a subprogram interface, then HPF requires that the corresponding actual parameter to the subprogram conform with the dummy argument's decomposition. If the dummy argument specifies a decomposition that differs from the actual parameter, implicit data redistribution and/or realignment is necessary. Additionally, upon return to the main program, the original decomposition of the parameter data arrays must be re-established. Thus, there are two redistributions and/or realignments necessary

```

      REAL A(20,20)
      !HPF$ DISTRIBUTE A(*,BLOCK)
      ...
      CALL SUB(A)

      SUBROUTINE SUB(Z)
      REAL Z(20,20)
      !HPF$ TEMPLATE T(20,20)
      !HPF$ DISTRIBUTE T(BLOCK, BLOCK)
      !HPF$ ALIGN WITH T::Z

```

Figure 2.7: Redistribution at subprogram interface

for each parameter. Figure 2.7 demonstrates this type of implicit redistribution. In this case, the actual parameter, *A*, originally distributed *(*,BLOCK)* is redistributed *(BLOCK,BLOCK)* to conform with the distribution of the dummy parameter, *Z*.

Programmers will attempt to optimize the subprogram with a particular data decomposition [20], so long as the change in decomposition does not result in higher overhead than the performance benefit of a new decomposition. Remapping of the actual parameter to the dummy argument can be overridden with HPF's *INHERIT* attribute. This attribute specifies that the dummy argument takes on the data decomposition of the actual parameter, whatever it may be. Providing for *any* legal distribution in a library, however, may be complicated and may not be cost effective [8].

Johnsson discusses redistribution at subroutine calls in the context of designing subroutine libraries [42]. He cites the ability to communicate data decomposition information to the subroutine, and subsequent determination of whether to use redistribution or realignment, as salient issues. These issues are explored in the design of

FFT and matrix-vector multiplication subroutines. Hall *et al.* [15] propose insertion of redistribution primitives in the caller, rather than the callee. This approach is shown to lead to compiler optimizations, for instance, the elimination of unnecessary redistributions.

2.6 Redistribution Optimizations

Due to the overhead cost that data redistribution represents, it can be beneficial to explore optimizations that can reduce the execution time of data redistribution. For instance, data that is to be redistributed at a certain point in program execution may not be referenced in subsequent computation. Certainly, for correct program execution, such data need not be redistributed. For example, suppose the redistribution of a template T with aligned arrays A and B is called for, but only B is used in subsequent computation; redistribution of A would be unnecessary for correct program semantics. If the compiler is able to recognize such a situation, then the data could be left in place, reducing the redistribution cost. If such data represented a significant portion of the whole and could be recognized by the compiler as unused in later computation, then this optimization may prove useful. Less straightforward situations, however, may prove more difficult to optimize. For instance, say that A is a large matrix which is to be redistributed, but only a fraction of its elements will be accessed subsequently. A desirable optimization would be to only redistribute data elements which will be referenced later.

Pipelined redistribution of data is another possible optimization considered by

several researchers. Strip Mining Redistribution [43] is a technique whereby the data to be communicated in a redistribution is temporally overlapped with the subsequent computation phase. Rather than redistribute all the data prior to resuming algorithmic computation, data is redistributed in a pipelined manner. A key issue in this technique is the ability to identify the computation pattern under the destination distribution pattern and to arrange the communication schedule to enable significant overlapping with computation. The technique is shown to have speedups of up to 1.7 over “massive redistribution”, *i.e.*, non-pipelined, when the data sizes are relatively large compared to message start-up latencies. Strip Mining Redistribution is limited to two-to-one and one-to-two dimension redistributions so that the size of the individual messages fed into the pipe are large enough to amortize message startup cost. Explicit Data Placement (XDP) [44, 45] is similar to Strip Mining Redistribution in that it facilitates pipelined data redistribution. A novel feature of this approach is the unified treatment of data and *ownership* transfer³ to allow for compiler-recognizable optimizations. Using a 3D FFT application as an example, XDP is shown to facilitate overlapped data redistribution with algorithm computation. Unfortunately, no performance data illustrating the performance of the proposed method are given.

Various message-level optimization techniques, such as *message coalescing*, *message aggregation*, and *message pipelining* [25] may prove useful in data redistribution. Message coalescing involves the combining of separate references to the same array. Message aggregation combines messages from distinct arrays to the same processor. Message pipelining attempts to hide communication latency through the separation

³In contrast to HPF’s owner-computes rule.

of *send* and *receive* primitives. Message aggregation, for example, could be utilized to combine aligned data in a common message buffer for redistribution to another processor. Techniques for applying these optimizations to the data movement problem, when the owner-computes rule is relaxed, are presented in [46].

Prior to data redistribution, whether using `!HPF$ REDISTRIBUTE` or at a subprogram interface, the logical to physical processor mapping is already determined for the initial data distribution, *i.e.*, D_s ; however, it is possible to manipulate the data element/logical processor mapping in the target, *i.e.*, the mapping of P_t onto D_t , so long as the semantics of the target distribution pattern are not violated. For instance, in Fig. 2.2 (b), the mapping of processors to data elements is in increasing processor number order, *i.e.*, $0, 1, 2, 3$. This mapping, however, need not be in increasing order. For instance, the mapping order could be permuted to be $0, 2, 1, 3$. The alternative mapping does not violate HPF semantics and can lead to possible run-time optimizations. In Chapter 3, we present a theory for permuting the processor-to-data mapping for HPF patterns.

Wakatani and Wolfe [47] propose a technique for mapping logical to physical processors in a data redistribution library that can reduce communication overhead. This technique assumes an underlying torus topology and maps the data in such a way that communicating processors are partitioned into non-overlapping sets, *i.e.*, there is no channel contention between sets. Thus, by keeping communication local to a group, message contention on the physical network, presumably, is reduced. The drawback of this approach is that the logical to physical data mapping is based on “local” information. In other words, the technique imparts a mapping based solely

on what is best to optimize redistribution. However, since the logical to physical data mapping must remain consistent throughout the execution of a data-parallel program, the chosen mapping may increase communication cost elsewhere. We assert that the compiler, privy to global information, *i.e.*, the entire data-parallel program, ought to determine the logical to physical processor mapping.

Chapter 3

Data Mapping Optimization

Since interprocessor communication is the predominant source of redistribution execution time, minimizing the total amount of data movement among processor memories may increase redistribution performance. This chapter presents a technique, based on logical processor to data element mapping, that *minimizes* the total amount of data movement among processor memories for **BLOCK** to **CYCLIC**(c), and vice-versa, redistributions. We present mapping functions for one-to-one, m -to- m , one-to-two, and two-to-one dimension data redistributions, and we prove their optimality. The proposed methodology is architecture-independent facilitating its potential integration into distinct redistribution implementations for different distributed-memory architectures.

In Chapter 6, we discuss the possible impacts on the programmer and compiler of using the optimal mapping technique. We show that the technique offers the programmer extra flexibility in determining data placement in some instances. Additionally, the technique could be used in a straightforward manner by a compiler. We demon-

strate redistribution performance improvements over the traditional data-processor mapping of up to 40% using the optimal technique on an IBM SP- x .

3.1 One-dimensional Logical Processor Mapping

We begin by illustrating the utility of the mapping technique for one-dimensional data. Figure 3.1 illustrates an initial **BLOCK** distribution of a sixteen element data array, **A**, onto eight processors and a subsequent redistribution of the array using the **CYCLIC** pattern. The mapping of the initial distribution of data onto the physical nodes of the machine is established at program initialization; see arrow labeled (1). Subsequent redistribution among processors must retain a consistent logical to physical processor mapping; see arrow labeled (2). Logical processor IDs (*lpids*) are in bold italics and are superimposed over the data. Each processor owns two contiguous elements under **BLOCK**, but two non-contiguous elements, with a stride of p , the number of processors, under **CYCLIC**. The *lpids* are mapped in increasing numerical order as specified in [3]. We claim that this is an unnecessary restriction as *any* permutation of the *lpids*, $0..7$, can conform to the semantics of the **CYCLIC** pattern since data distribution to logical processors in HPF has no concept of the “ordering” of processors. **CYCLIC** requires only that the global data elements owned by a processor have global indices that are separated by a stride of p . For clearer presentation, we view the data as being *static*, and we manipulate the processor mapping to the data.

Figure 3.2 illustrates the benefit of permuting the *lpids* when mapping them to data elements. Using the same sixteen element array of Fig. 3.1, we show two alter-

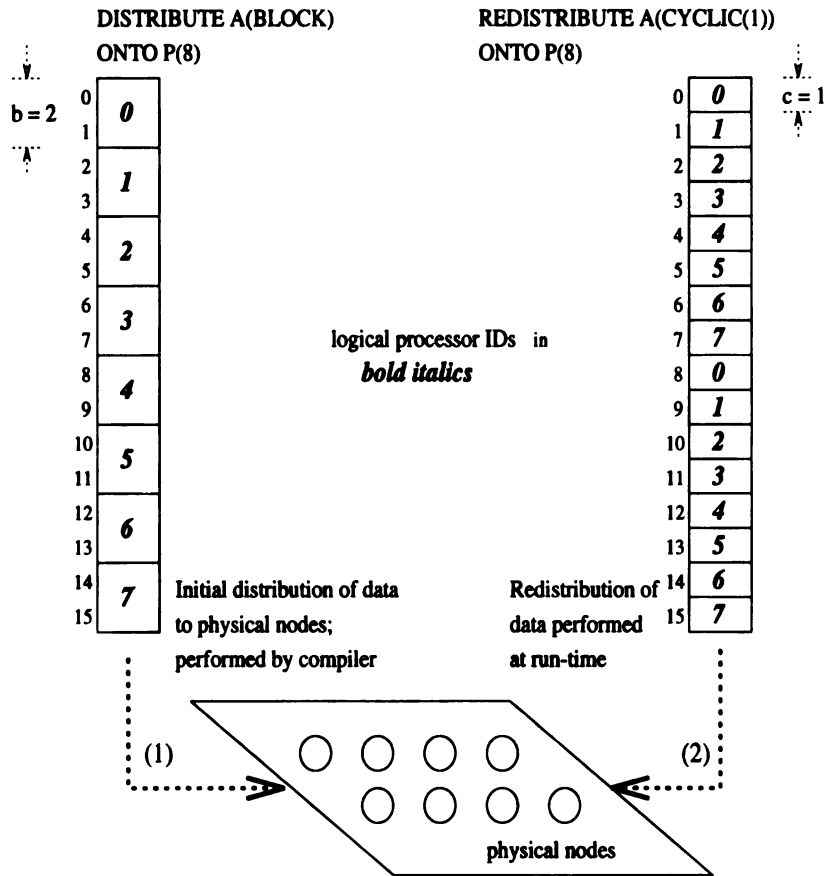


Figure 3.1: Data Distribution and Redistribution onto physical nodes.

natives for redistributing the array cyclically. Choice 1 shows the conventional cyclic mapping of *lpids* to data. This results in only two of sixteen data elements (marked with filled rectangles) remaining on the same processor following redistribution. We call this the number of *data hits* among all processors. Choice 2 shows an alternative cyclic mapping with the *lpids* permuted, *0,4,1,5,2,6,3,7*, which results in a total of eight data elements, one per processor, remaining on their original memories. This eliminates the exchange of six data elements among processors. Another advantage is that processors *1,...,6* must send data to two processors when using Choice 1, but only one processor each when using Choice 2, thus reducing the number of destinations. Each of these factors reduces interprocessor communication overhead. The reduction

in redistribution cost for the example in Fig. 3.2 is small given the size of the example.

If we extend **A** to be sixteen million data elements with the same permutation of *lpids*, we eliminate the exchange of six million data elements across processors.¹

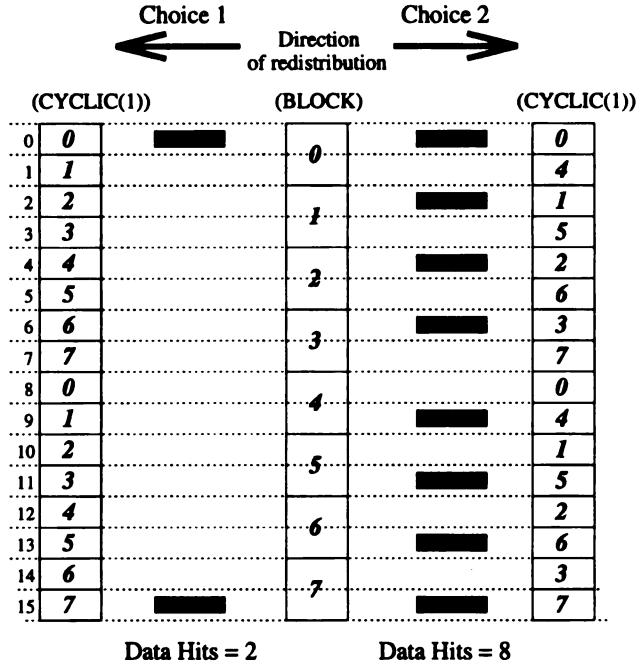


Figure 3.2: Logical processor to data mapping alternatives.

3.1.1 Processor Mapping Technique

To consistently minimize the size of data transfer for arbitrary data block and processor set sizes, we develop a systematic method for determining a permutation of *lpids* to map to the data. The technique ensures that each processor retains the maximum amount of data possible while conforming to the semantics of the source and target distribution patterns, D_s and D_t .

We establish the upper bound on the ratio of the amount of data that can be

¹ Assuming the number of processors is kept the same, the block size in **BLOCK** is two million, and the block size in **CYCLIC(c)** is one million.

retained on p processors to the total number of data elements, n . We present a function for determining an *lpid* to data mapping for redistribution from **BLOCK** to **CYCLIC**(c)² that achieves the upper bound. We make the following assumptions:

1. Let p be the number of processors numbered $0..p-1$, and n be the total number of data elements, numbered $0..n-1$, distributed over p . We assume each processor owns b elements, thus $b = n/p$.
2. We consider redistribution between **BLOCK** and **CYCLIC**(c) patterns; the **BLOCK**(b) pattern, where b is a variable, is not considered; $b = n/p$ by assumption 1. For **CYCLIC**(c), we assume that c divides b , i.e., $b = cz$ for an integer z .
3. If the data is initially distributed among p processors, then we assume that the data is redistributed among p processors.

Extensions of the above assumptions to the general case will be discussed in Chapter 7.

Definition 1 *Define r as the number of data elements of the global array that remain (data hits) on their original processors following redistribution between **BLOCK** and **CYCLIC**(c). Define HitRatio as $r : n$ and MaxHitRatio as the upper bound on HitRatio.*

²Redistribution is symmetric in terms the amount of data movement among processors, i.e., redistribution from **BLOCK** to **CYCLIC**(c) or redistribution from **CYCLIC**(c) to **BLOCK** results in an equal amount of data movement.

Lemma 1 *Let b and c be the block sizes in the BLOCK and CYCLIC(c) patterns, respectively. If $n = bp$ and $z = b/c$, where z is an integer, then $\text{MaxHitRatio} = \lceil \frac{b}{cp} \rceil cp : n$ and $\text{HitRatio} \leq \text{MaxHitRatio}$.*

Proof: *Case 1:* $z = ip$, for all $i \geq 0$, i is an integer. For every cycle of cp data elements, each $lpid_j$, $0 \leq j \leq p - 1$, maps to c contiguous data elements with CYCLIC(c). Since $b = icp$, there are i complete cycles of $lpids$ that map to one complete data block owned by $lpid_j$ under the BLOCK pattern. Processor $lpid_j$ will map to exactly ic of the elements, i.e., there are ic data hits. For p processors, the number of data hits is $icp = \lceil \frac{ip}{p} \rceil cp = \lceil \frac{b}{cp} \rceil cp$. HitRatio is $\lceil \frac{b}{cp} \rceil cp : n$.

Case 2: $(i - 1)p < z < ip$, for all $i > 0$, i is an integer. Since $b < icp$, there cannot be i complete cycles of $lpids$ mapped to $lpid_j$'s original block of data under BLOCK. Thus, the number of hits can be no greater than i for each $lpid_j$; consequently, the number of hits across all processors can be no greater than icp . The remainder of the proof follows as in *Case 1*. \square

MaxHitRatio is achieved with *any* permutation of $lpids$ when z is an integer multiple of the number of processors, i.e., when $z = ip$. However, our goal is to achieve the upper bound for *all* values z where $(i - 1)p < z < ip$. In order to satisfy this aim, we must consider different permutations of the $lpids$ to maximize the number of data hits. Figure 3.2 demonstrates that not all permutations of $lpids$ yield MaxHitRatio. Next, using the semantics of D_t , we define a function for determining a permutation of $lpids$ to map to the data array.

3.1.2 Mapping Function

Define a p -tuple, $(q_0, q_1, q_2, \dots, q_{p-1})$, as p place holders. Let $f : i \rightarrow j$ be a function that maps $lpid_i$ to place holder q_j for $0 \leq i, j \leq p - 1$. An assignment of each $lpid$ to a place holder specifies a permutation of the $lpids$ and represents a mapping of $lpids$ to data elements for the **CYCLIC**(c) distribution pattern. There are $p!$ possible permutations for p processors and it may be the case that many of the permutations yield a ratio of **MaxHitRatio**. However, exhaustively testing each permutation to determine whether it produces the ratio would be impractical since this would require an exponential amount of computation for general p . Therefore, we present a function for determining a permutation that achieves **MaxHitRatio** for b, p and c . Each $lpid_i$, $0 \leq i \leq p - 1$, maps to a unique $q_{f(i)}$. Equation (3.1) specifies the function that maps $lpid_i$ to $q_{f(i)}$.

$$f(i) = (iz) \bmod p, \quad 0 \leq i \leq p - 1, \quad z = b/c. \quad (3.1)$$

The intuition behind the mapping function is to first view the place holders, q_j , as a circular list. The function maps $lpid_0$ to place holder q_0 , maps $lpid_1$ z places from $lpid_0$, maps $lpid_2$ z places from $lpid_1$, and so on. In general, we map $lpid_{i+1}$, $(z \bmod p)$ places from $lpid_i$. Figure 3.3 illustrates the behavior of f applied to different values of z and p . For simplicity, we choose $c = 1$ which reduces D_t to **CYCLIC**. Part (a) shows an example for $p = 7$ and $z = 4$, while part (b) illustrates the case for $p = 6$ and $z = 4$. The mapping is broken into rows to better illustrate the distance $z = b/1$ between consecutive $lpids$. The distinction between the two examples is that $f : i \rightarrow j$

is one-to-one in part (a), but f is not one-to-one in part (b); that is, in part (b), more than one $lpid$ maps to some locations, while no $lpids$ map to other place holders. More formally, depending on the values of z and p , it is possible that $f(i) = f(j)$, $i \neq j$. While f yields one permutation for part (a), it produces six possible permutations³ for part (b): since $lpid_0$ and $lpid_3$ map to q_0 , it turns out that we can arbitrarily map the two $lpids$ to place holders q_0 and q_1 . The same holds true for $lpid_2$ and $lpid_5$ to place holders q_2 and q_3 and for $lpid_1$ and $lpid_4$ to place holders q_4 and q_5 . We shall prove this result in a later lemma.

q_0	0
q_1	2
q_2	4
q_3	6
q_4	1
q_5	3
q_6	5

(a) $p = 7, z = 4$

q_0	0	3
q_1		
q_2	2	5
q_3		
q_4	1	4
q_5		

(b) $p = 6, z = 4$

Figure 3.3: Mapping $lpids$ to place holders.

3.1.3 Optimality of Mapping Function

Given arbitrary z and p , the $\gcd(z, p)$ determines whether f is one-to-one or not. Lemmata 2 and 3 establish this. Lemmata 4 and 5 establish that f achieves MaxHitRatio whether or not it is one-to-one.

Lemma 2 *Let p and z be natural numbers and $z = b/c$. Let $\gcd(z, p)$ be the greatest common divisor of z and p . If $\gcd(z, p) = 1$, then $f : i \rightarrow j$ establishes a one-to-one*

³all optimal in terms of MaxHitRatio

mapping between $lpid_i$, $0 \leq i \leq p-1$, and place holder q_j , $0 \leq j \leq p-1$. In other words, if $\gcd(z, p) = 1$, then $f(j) \neq f(k)$ for all j, k and $0 \leq j, k \leq p-1$, $j \neq k$.

Proof: Proof by contradiction. Assume $\gcd(z, p) = 1$, $f(j) = f(k) = n$ for arbitrary j, k and choose $k > j$. Recall that the mapping function, f , maps each $lpid_{i+1}$, $(z \bmod p)$ places from $lpid_i$. Let $r = k - j$, then $lpid_k$ is mapped a distance of $r(z \bmod p)$ place holders from $lpid_j$. Since $f(j) = f(k) = n$, then $lpid_j$ and $lpid_k$ map to the same place holder, so their distance, $\bmod p$, is zero, i.e., $r(z \bmod p) = 0$. Since $r > 0$, it must be that $z \bmod p = 0$. This implies that the $\gcd(z, p) > 1$, a contradiction. Thus, our assumption that $f(j) = f(k) = n$ when $\gcd(z, p) = 1$ is false. \square

Lemma 3 Let p and z be natural numbers and $z = b/c$. If $\gcd(z, p) = k$, then f maps $lpid_{i+j(p/k)}$ to $q_{f(i)}$ for $0 \leq i \leq p/k - 1$ and $0 \leq j \leq k - 1$.

Proof: Show that $f(i) = f(i + j(p/k))$ for $0 \leq i \leq p/k - 1$, $0 \leq j \leq k - 1$.

$$f(i) = f(i + j\frac{p}{k})$$

$$(iz) \bmod p = (i + j\frac{p}{k})z \bmod p$$

$$(iz) \bmod p = (iz + jz\frac{p}{k}) \bmod p$$

$$(iz) \bmod p = (iz) \bmod p + (jz\frac{p}{k}) \bmod p$$

Since k divides z , $jz_k^p = mp$ for some integer m . $mp \bmod p = 0$ for arbitrary m .

Thus, the second term of the sum, $(jz_k^p) \bmod p = 0$, so we are left with

$$(iz) \bmod p = (iz) \bmod p$$

□

We have established two lemmas that capture the behavior of f in Equation (3.1) for natural numbers $z = b/c$ and p and shown the relationship of $\gcd(z, p)$ to the function f . In Fig. 3.3 (a), $\gcd(z, p) = 1$ and thus f is one-to-one, while in part (b), $\gcd(z, p) = 2$ and thus f maps two $lpids$ to place holders q_0, q_2, q_4 . Next we establish two lemmas that show f will produce permutations that always yield MaxHitRatio.

Lemma 4 *Let p and z be natural numbers and $z = b/c$. If $\gcd(z, p) = 1$, then f determines a single permutation of $lpids$ that achieves MaxHitRatio.*

Proof: Since f maps $lpid_{i+1}$ z places from $lpid_i$, each $lpid$ will map, under CYCLIC, to the *first* data element of its data block under BLOCK. Figure 3.4 illustrates the situation for arbitrary $lpid_i$. Thus, there is always at least one data hit per $lpid$.

Case 1: If $z \leq p$, then there are exactly c data hits per $lpid$. The mapping cycle begins with $lpid_i$ ensuring c data hits. Since $z \leq p$, which is equivalent to $b \leq cp$, $lpid_i$ cannot map to another c data elements of the b -sized data block. If it did, this would violate the semantics of a cyclic mapping. Thus, there are exactly c hits per processor, cp hits over all processors. It follows that $cp : n = \lceil \frac{b}{cp} \rceil cp : n$ since $b \leq cp$.

Case 2: If $z > p$, then there are at least c data hits per $lpid$ as established above. Since $lpid_i$ maps to the first element of the data block, it will also map to the $(cp + 1)$ th element as well since $z > p$, see Fig. 3.4. Let $j = z/p = b/cp$ (integer division), then $lpid_i$ will map to elements numbered $kcp, kcp + 1, \dots, kcp + c - 1$ for $0 \leq k \leq j - 1$; a total of cj elements⁴. Thus, there are $jc = \lceil \frac{b}{cp} \rceil c$ data hits per $lpid$ and $\lceil \frac{b}{cp} \rceil cp$ data hits over all $lpids$. Again, $\lceil \frac{b}{cp} \rceil cp : n$ is achieved. \square

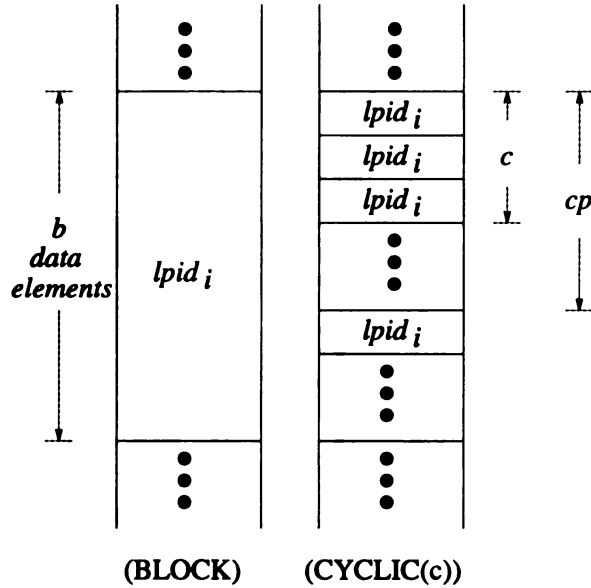


Figure 3.4: $lpid_i$ is mapped to first element of its data block.

Lemma 5 *Let p and z be natural numbers and $z = b/c$. If $k = \gcd(z, p) > 1$, then the k $lpids$ that map to q_{ik} under f can be remapped, in any of $k!$ ways, to the place holders $q_{ik}, q_{ik+1}, q_{ik+2}, \dots, q_{(i+1)k-1}$, for $0 \leq i \leq p/k - 1$. All $(p/k)k!$ permutations yield MaxHitRatio.*

⁴Data elements are numbered $0..b - 1$ for the data block owned by $lpid_i$.

Proof: In Lemma 3, we established that if $k > 1$, then k *lpids*, namely $lpid_{i+j(p/k)}$ for $0 \leq j \leq k - 1$ map to the same place holder, $q_{f(i)}$. Consequently, k *lpids* map to the first data element of $lpid'_i$'s data block under BLOCK.

Case 1: If $z \leq p$, then $b \leq cp$, and there are exactly c data hits per *lpid*. Figure 3.5 illustrates the situation. If $z \leq p$, then $k \leq z$. Thus, $ck \leq cz$ and $cz = b$. Furthermore, the k *lpids* with c data elements each can “fit” into $lpid'_i$'s b -sized data block. Each *lpid* that mapped to place holder $q_{f(i)}$ can be remapped to one of the first ck data elements regardless of the permutation of the k *lpids*. Since $lpid_i$ is in this group, there are c data hits for it. $lpid_i$ cannot appear again within the data block since $b \leq cp$. Therefore, there are exactly c data hits for $lpid_i$, $cp = \lceil \frac{b}{c} \rceil cp$ data hits total among all p processors.

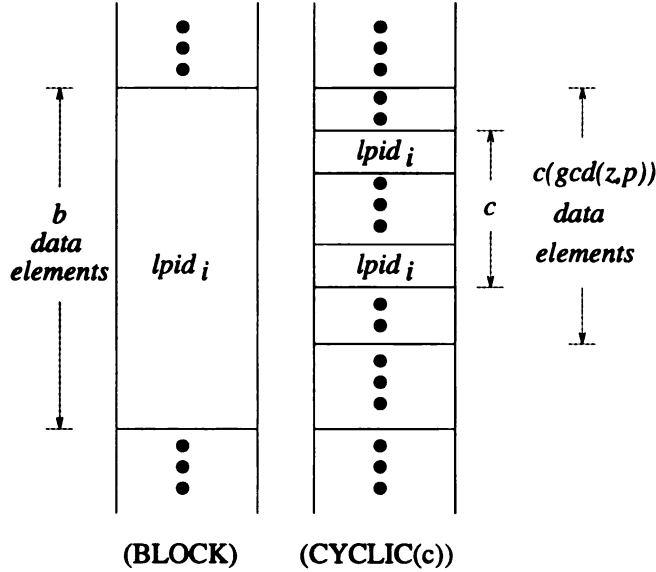


Figure 3.5: $z \leq p$.

Case 2: If $z > p$, then $b > cp$ and there are at least c data hits per *lpid* as established above. For some integer m , if $mcp < b < (m + 1)cp$, then there are mc

data hits for $lpid_i$; since all p $lpids$ map at least m cycles to a b -sized data block. We claim that there are $(m+1)c$ data hits, even when $b < (m+1)cp$. Figure 3.6 illustrates the situation. There are m groups of p processors mapping c -sized blocks to $lpid'_i$'s data block under BLOCK. We must show that $lpid_i$ maps to one of the last $b - mcp$ elements of the data block. In other words, show $ck \leq b - mcp$. Proof by contradiction. Assume $ck > b - mcp$. There exist two integers, r, s , such that $z = rk$ and $p = sk$. If $z > p$, then $k \leq p$.

$$ck > b - mcp$$

$$k > \frac{b}{c} - mp$$

$$k > z - mp \geq rk - mk$$

$$k > rk - mk$$

$$k > k(r - m)$$

$$m > r$$

$$mp > rp$$

$$z > mp > \frac{z}{k}p$$

$$z > \frac{p}{k}z$$

Since $p/k \geq 1$, we have a contradiction. Thus, $c(\gcd(z, p)) \leq b - mcp$. Given this result, $lpid'_i$'s c -sized block must appear within the last $b - mcp$ elements of the data block regardless of permutation order. Therefore we have $(m + 1)c$ data hits per $lpid_i$; $(m + 1)cp$ for p processors. \square

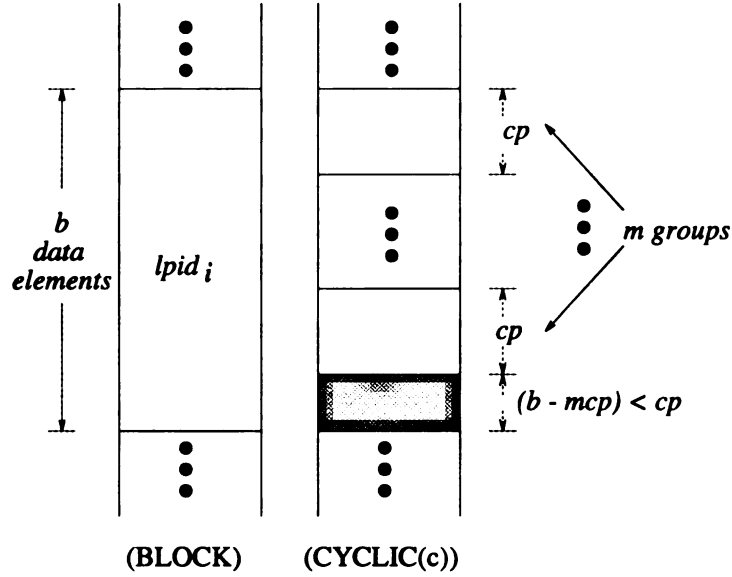


Figure 3.6: $z > p$.

Lemmata 1 through 5 prove the following result.

Theorem 1 For redistribution from **BLOCK** to **CYCLIC(c)**, where b and c are the respective block sizes, $z = b/c$ for an integer z , p is the number of processors, and $n = bp$ is the number of global data array elements, the logical processor mapping function in Equation (3.1) achieves $\text{MaxHitRatio} = \lceil \frac{b}{cp} \rceil cp : n$.

3.2 Multidimensional Logical Processor Mapping

This section extends the logical processor mapping technique presented in Section 3.1 to m -dimensional data arrays. Specifically, we extend the technique to optimizing the logical processor mapping when redistributing from $(\text{BLOCK}, \text{BLOCK}, \dots, \text{BLOCK})$ to $(\text{CYCLIC}(c_0), \text{CYCLIC}(c_1), \dots, \text{CYCLIC}(c_{m-1}))$. Additionally, we demonstrate the approach for redistribution of two-dimensional data that is (1) initially distributed in both dimensions and subsequently redistributed in only one dimension, *e.g.*, $(\text{BLOCK}, \text{BLOCK})$ to $(\text{CYCLIC}, *)$, and (2) initially distributed in one dimension and then redistributed in both dimensions, *e.g.*, $(\text{BLOCK}, *)$ to $(\text{CYCLIC}(c_1), \text{CYCLIC}(c_2))$.

3.2.1 m -dimensional Redistribution

We extend the technique by applying the one-dimensional *lpid* mapping to each dimension of the global data array. Let A be a m -dimensional data array with $n_0 \times n_1 \times n_2 \times \dots \times n_{m-1}$ data elements. The **PROCESSORS** directive in HPF declares a processor arrangement specifying the name, rank, and extent in each dimension. Let $P = (p_0, p_1, \dots, p_{m-1})$ be the processor arrangement over which A is distributed. Let $B = (b_0, b_1, \dots, b_{m-1})$ and $C = (c_0, c_1, \dots, c_{m-1})$ be the set of block sizes and cyclic block sizes respectively for each dimension of A . We extend Equation (3.1) to Equation (3.2) to map the rectilinear set of *lpids* p_j to the n_j data elements in the j -th dimension for $0 \leq j \leq m - 1$.

$$g(i) = (iz_j) \bmod p_j, z_j = b_j/c_j, 0 \leq i \leq p_j - 1, 0 \leq j \leq m - 1. \quad (3.2)$$

For m -dimensional data redistribution, **MaxHitRatio** is very similar to the ratio presented in Section 3.1; it is the product of the **MaxHitRatios** for each of the m dimensions. We present a lemma to substantiate this result.

Lemma 6 ***MaxHitRatio** for a m -dimensional global array A as defined above is $(\lceil \frac{b_0}{c_0 p_0} \rceil c_0 p_0 \times \dots \times \lceil \frac{b_{m-1}}{c_{m-1} p_{m-1}} \rceil c_{m-1} p_{m-1}) : (n_0 \times \dots \times n_{m-1})$ and the mapping function in Equation (3.2) achieves the upper bound.*

Proof: Lemmata 1 through 5 established that $\lceil \frac{b}{cp} \rceil cp : n$ is the **MaxHitRatio** for one-dimensional data and that the mapping function, f , achieved the upper bound. For m -dimensional data, the product of the upper bounds in each dimension yields the maximum data hit ratio for the m -dimensional array. The function g is a generalization of f . By applying g respectively to each dimension, the upper bounds are achieved. □

Figure 3.7 illustrates g applied to an 18×16 data matrix distributed across a 3×4 processor grid; *lpids* are in bold italics⁵. We redistribute the data matrix from (BLOCK,BLOCK) to (CYCLIC(3),CYCLIC(2)). The *lpid* to data mappings for D_s in both dimensions is marked **Block**. The traditional cyclic mapping of *lpids* to data is indicated with **Trad**, while the optimized technique using g is shown with **Opt**. The key to the right of the figure indicates data hits following redistribution for the

⁵Subscripts in the figure label denote the number of processors in the given dimension.

traditional and optimized mappings. The traditional mapping yields $\text{HitRatio} = 1 : 12$. The mapping using g results in $\text{HitRatio} \lceil \frac{6}{(3)(3)} \rceil (3)(3) \times \lceil \frac{4}{(2)(4)} \rceil (2)(4) : (18)(16)$ which reduces to $1 : 4$, a three-fold increase in the number of data hits over the traditional mapping. Theorem 1 and Lemma 6 prove the following result.

Theorem 2 *For redistribution from $(\text{BLOCK}, \text{BLOCK}, \dots, \text{BLOCK})$ to $(\text{CYCLIC}(c_0), \text{CYCLIC}(c_1), \dots, \text{CYCLIC}(c_{m-1}))$, where b_i and c_i are the respective block sizes, $z_i = b_i/c_i$ for an integer z_i , p_i is the number of processors, and $n_i = b_i p_i$ is the number of data elements in dimension i , $0 \leq i \leq m-1$, the logical processor mapping function in Equation (3.2) achieves $\text{MaxHitRatio} = (\lceil \frac{b_0}{c_0 p_0} \rceil c_0 p_0 \times \dots \times \lceil \frac{b_{m-1}}{c_{m-1} p_{m-1}} \rceil c_{m-1} p_{m-1}) : (n_0 \times \dots \times n_{m-1})$.*

3.2.2 Two-dimensional to One-dimensional Redistribution

Redistribution of an m -dimensional array need not necessarily involve redistribution in all m dimensions. For instance, a data matrix may initially be distributed $(\text{BLOCK}, \text{BLOCK})$ and redistributed to $(\text{CYCLIC}, *)$ in which only the row dimension is distributed. We extend the mapping technique to these cases. We define a vector of $p = p_0 p_1$ place holders, q_0, q_1, \dots, q_{p-1} . Equation (3.3) defines a new function that maps a Cartesian *lpid* to a placeholder.

$$h(r, s) = (rz) \bmod (p_0 \times p_1), \quad z = b_0/c_0. \quad (3.3)$$

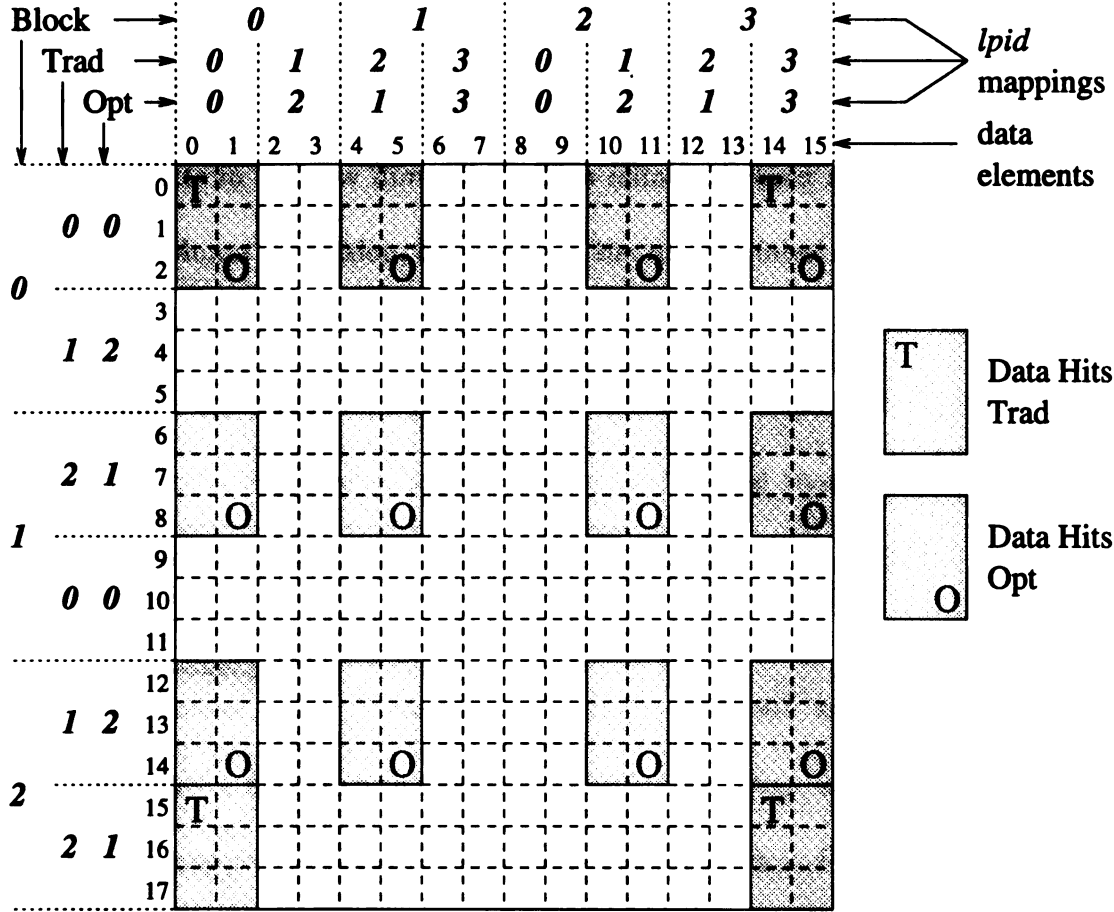


Figure 3.7: $(\text{BLOCK}_3, \text{BLOCK}_4)$ to $(\text{CYCLIC}_3(3), \text{CYCLIC}_4(2))$ redistribution.

Lemma 7 establishes MaxHitRatio for two-dimensional to one-dimensional redistribution. Based on the value of $\gcd(z, p_0)$, Lemmata 8 and 9 establish that the mapping function h achieves MaxHitRatio.

Lemma 7 For redistribution between $(\text{BLOCK}, \#)^6$ and $(\text{CYCLIC}(c_0), *)$, let b_0 be the block size in the row dimension of D_s and c_0 be the cyclic block size in D_t . Max-HitRatio = $(\lceil \frac{b_0}{c_0 p_0 p_1} \rceil c_0 p_0 p_1 b_1) : (n_0 \times n_1)$ for an $n_0 \times n_1$ data matrix.

Proof: The proof is quite similar to the proof of Lemma 1. Here, the two-dimensional grid of processors is remapped into a vector of place holders since

⁶ # denotes any HPF distribution pattern

D_t is distributed in only one dimension. We substitute $p_0 p_1$ for p in Lemma 1 and the remainder of the proof follows from that point. Thus $\lceil \frac{b_0}{c_0 p_0 p_1} \rceil c_0 p_0 p_1$ counts the number of hits in the row dimension. This number is then multiplied by the block size in the column dimension, b_1 , to obtain the total number of data hits. \square

We prove that the mapping function, h , yields MaxHitRatio by first establishing the correspondence between *lpids* and place holders and then showing the mapping yields the given ratio. Since the column coordinate, s , of an *lpid* is not relevant to the value computed by h , all *lpids* with the same row coordinate, r , map to the same place holder, $q_{h(r)}$. Since $0 \leq s \leq p_1 - 1$, there are exactly p_1 *lpids* for each r , $0 \leq r \leq p_0 - 1$ that map to the same place holder. Thus, no greater than p_0 place holders are mapped to under h . Lemma 8 establishes that h achieves MaxHitRatio when only *lpids* with the same row coordinate r map to the same place holder while Lemma 9 establishes the result when $k p_1$ *lpids* map to the same place holder.

Lemma 8 *Let z be a natural number such that $z = b_0/c_0$. If $\gcd(z, p_0) = 1$, then exactly p_0 place holders are mapped by $h(r, s)$. The p_1 *lpids* that map to place holder $q_{h(r, s)}$ can be remapped in any of $p_1!$ ways to place holders $q_{h(r, s)}, q_{h(r, s)+1}, q_{h(r, s)+2}, \dots, q_{h(r, s)+p_1-1}$. All $(p_0)p_1!$ permutations yield MaxHitRatio $= \lceil \frac{b_0}{c_0 p_0 p_1} \rceil c_0 p_0 p_1 b_1$.*

Proof: The first conjecture of the lemma follows directly from Lemma 2. The proof of the second part of the lemma is similar to the proof of Lemma 5. Instead of $k = \gcd(z, p)$ *lpids* mapping to the same place holder as in Lemma 5, we have p_1

lpids mapping to the same place holder as established previously. The remainder of the proof follows after making this substitution. MaxHitRatio is the same result as in Lemma 5 except that b_1 must be a factor since the number of data elements in a row contribute to the number of data hits. \square

Lemma 9 *Let z be a natural number such that $z = b_0/c_0$. If $\gcd(z, p_0) = k$, then p_0/k place holders are mapped by $h(r, s)$. $h(r, s)$ maps $lpid_{r+j(p_0/k), s}$ to place holder $q_{h(r, s)}$ for $0 \leq r \leq (p_0/k) - 1$ and $0 \leq j \leq k - 1$. The $p_1 k$ *lpids* that map to place holder $q_{h(r, s)}$ can be remapped in any of $(p_1 k)!$ ways to place holders $q_{h(r, s)}, q_{h(r, s)+1}, q_{h(r, s)+2}, \dots, q_{h(r, s)+p_1 k-1}$. All $(p_0)(p_1 k)!$ permutations yield MaxHitRatio $= (\lceil \frac{b_0}{c_0 p_0 p_1} \rceil c_0 p_0 p_1 b_1) : (n_0 \times n_1)$.*

Proof: The first portion of the lemma follows directly from Lemma 3. The proof of the second part of the lemma is similar to the proofs of Lemmata 5 and 8. In the current situation, we have $p_1 k$ *lpids* that map to the same place holder. The remainder of the proof follows after making this substitution. MaxHitRatio is the same as in Lemma 8. \square

Figure 3.8 demonstrates the mapping function, h , applied to a 24×16 matrix. The matrix is originally distributed (BLOCK, BLOCK) on a 3×2 processor grid and is redistributed (CYCLIC(2), *) on a six processor vector. *lpids* are in bold italics as before. $lpid_{r, s}$ identifies a processor in the Cartesian system; $0 \leq r \leq p_0 - 1$ and $0 \leq s \leq p_1 - 1$. The block size in the row dimension of (BLOCK, BLOCK) is eight

$(b_0 = 8)$; the cyclic block size in $(\text{CYCLIC}(2), *)$ is two.

In Fig. 3.8, we do not show a “traditional” mapping since mapping from a set of Cartesian $lpids$ to a vector of $lpids$ is undefined in HPF. Using h to map the $lpids$ produces a HitRatio of $96 : 384 = 1 : 4$. The distribution for columns of D_s is inconsequential to the number of data hits that can be achieved since only rows are distributed in D_t . In other words, redistribution from $(\text{BLOCK}, \text{CYCLIC}(c))$ to $(\text{CYCLIC}, *)$ would result in the same number of data hits as the redistribution of $(\text{BLOCK}, \text{BLOCK})$ to $(\text{CYCLIC}, *)$. Additionally, any of the $lpids$ with the same row index, r , could be permuted and the same data hit ratio is achieved; *e.g.*, $lpid_{0,0}$ and $lpid_{0,1}$. Figure 3.9 shows an example of when $\gcd(z, p_0) > 1$. Processors $lpid_{0,s}$ and $lpid_{2,s}$ for $0 \leq s \leq 2$ could be permuted in any of the $6!$ possible ways and the optimal data hit ratio would be obtained.

Lemmata 7 through 9 prove the following result.

Theorem 3 *For redistribution from $(\text{BLOCK}, *)$ to $(\text{CYCLIC}(c_0), *)$, where b_0 and c_0 are the respective block sizes in the row dimension, $z = b_0/c_0$ for an integer z , b_1 is the block size in the column dimension, $p_0 \times p_1$ defines the grid of processors for D_s , $n_0 \times n_1$ is the number of global data array elements where $n_i = b_i p_i$ $0 \leq i \leq 1$, the logical processor mapping function in Equation (3.3) achieves $\text{MaxHitRatio} = \lceil \frac{b_0}{c_0 p_0 p_1} \rceil c_0 p_0 p_1 b_1$.*

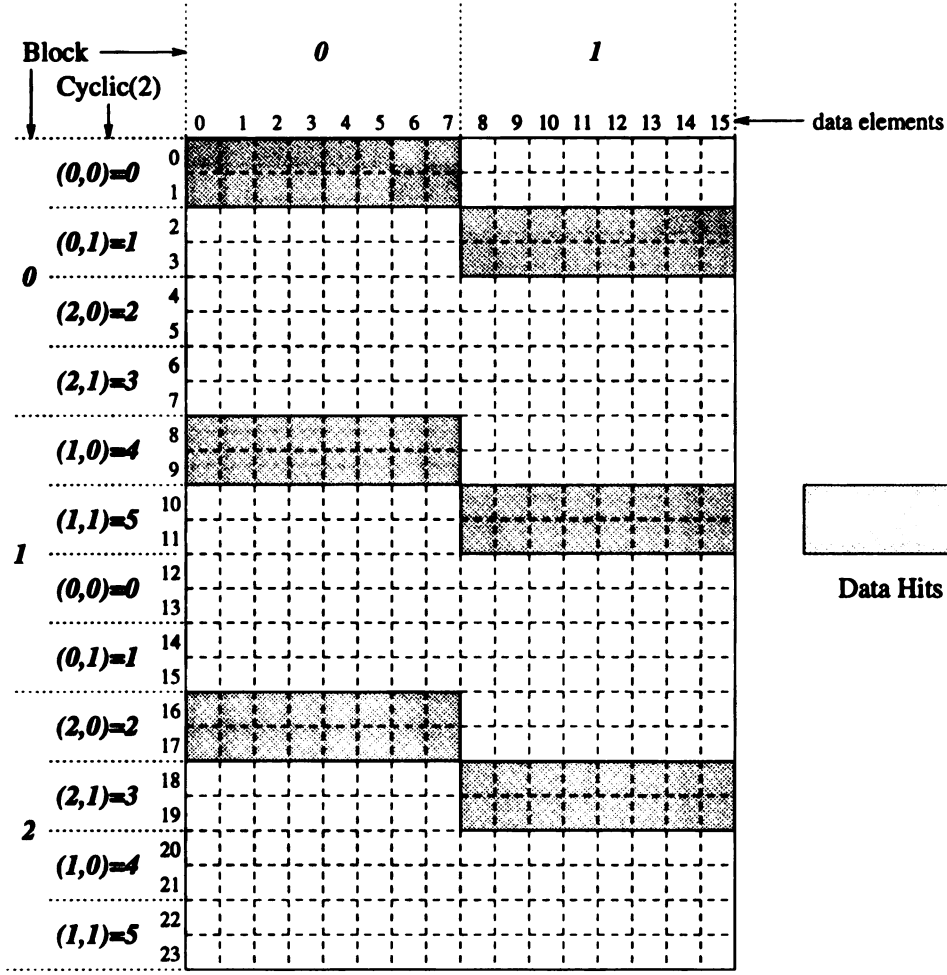


Figure 3.8: $(\text{BLOCK}_3, \text{BLOCK}_2)$ to $(\text{CYCLIC}_6(2), *)$ redistribution.

3.2.3 One-dimensional to Two-dimensional Redistribution

Another possibility is to have a data matrix initially distributed in one dimension and then redistributed in two dimensions. We derive a new mapping function, Equation (3.4), for redistribution from $(\text{BLOCK}, *)$ to $(\text{CYCLIC}(c_0), \#)$. Since D_t specifies a two-dimensional data distribution, we declare a two-dimensional grid of place holders, $q_{[r,s]}$, $0 \leq r \leq p_0 - 1$, $0 \leq s \leq p_1 - 1$. The function, ℓ , maps a processor $lpid_i$, $0 \leq i \leq p - 1$, to a place holder $q_{[r,s]}$; thus, $\ell(i)$ computes an ordered pair. Lemma 10 establishes MaxHitRatio for one-dimensional to two-dimensional redistribi-

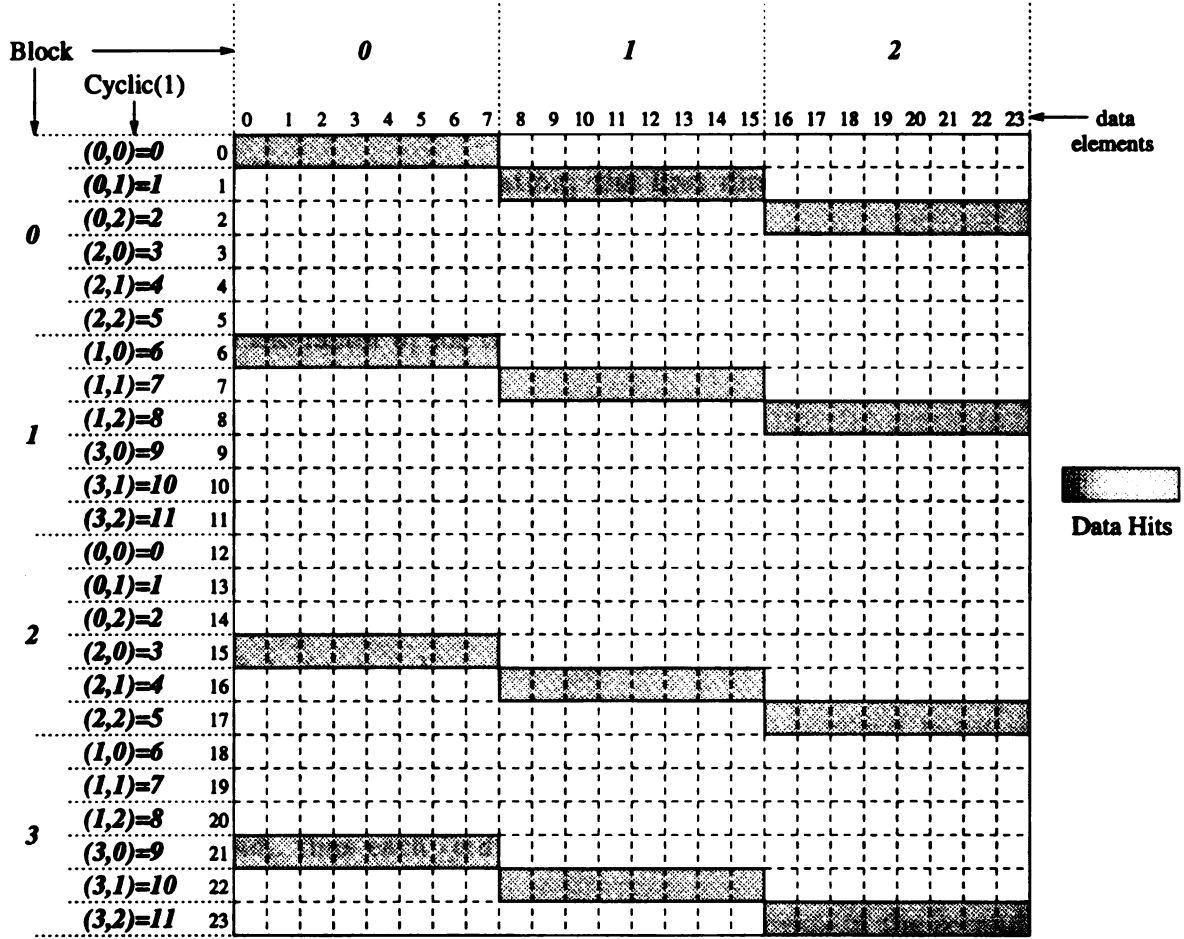


Figure 3.9: $(\text{BLOCK}_4, \text{BLOCK}_3)$ to $(\text{CYCLIC}_{12}(1), *)$ redistribution.

bution. Based on the value of $\gcd(z, p_0)$, Lemmata 11 and 12 prove that ℓ achieves MaxHitRatio.

$$\ell(i) = [(iz) \bmod p_0, i/p_0], \quad 0 \leq i \leq p-1, \quad z = b_0/c_0. \quad (3.4)$$

Lemma 10 For redistribution between $(\text{BLOCK}, *)$ and $(\text{CYCLIC}(c_0), \#)$, let b_0 be the block size in the row dimension of D_s and c_0 be the cyclic block size in D_t . MaxHitRatio = $(\lceil \frac{b_0}{c_0 p_0 p_1} \rceil c_0 p_0 p_1 b_1) : (n_0 \times n_1)$ for an $n_0 \times n_1$ data matrix.

Proof: The proof is quite similar to the proofs of Lemmata 1 and 7. In the single dimension case of Lemma 1, the redistribution between **BLOCK** and **CYCLIC**(c) is considered. In the current situation, the first dimension of both D_s and D_t is **BLOCK** and **CYCLIC**(c); thus, the proof of Lemma 1 can be applied. As with Lemma 7, we must include the factor b_1 for the second dimension. \square

Lemma 11 *Let z be a natural number such that $z = b_0/c_0$. If $\gcd(z, p_0) = 1$, then all p place holders are mapped by $\ell(i)$ and the mapping yields $\text{MaxHitRatio} = (\lceil \frac{b_0}{c_0 p_0 p_1} \rceil c_0 p_0 p_1 b_1) : (n_0 \times n_1)$.*

Proof: The proof is similar to the proof of Lemma 4 since $\ell(i)$ maps $lpid_{i+1}$ z places from $lpid_i$, thus each $lpid$ will be mapped to the first row of its data block under the (**BLOCK**, $*$) pattern. Since $lpid_i$ owns b_1 elements in the second dimension, this factor contributes to MaxHitRatio . \square

Lemma 12 *Let z be a natural number such that $z = b_0/c_0$ and $p = p_0 p_1$. If $\gcd(z, p_0) = k$, then $(p_0/k)p_1$ place holders are mapped by $\ell(i)$. All place holders in the p_1 dimension are mapped, while only p_0/k place holders in the p_0 dimension are mapped. Exactly k $lpids$ map to each place holder. The k $lpids$ that map to place holder $q_{[r,s]}$ can be remapped to place holders $q_{[r,s]}, q_{[r+1,s]}, \dots, q_{[r+k-1,s]}$ in any of $k!$ ways and achieve $\text{MaxHitRatio} = (\lceil \frac{b_0}{c_0 p_0 p_1} \rceil c_0 p_0 p_1 b_1) : (n_0 \times n_1)$.*

Proof: The proof is similar to the proof of Lemma 5 and Lemma 9. In the current situation, there is a second, p_1 , processor dimension and all place holders

in this dimension are mapped. The p_0 dimension in this case is the same as the one-dimensional mapping in Lemma 5. Since $lpid_i$ owns b_1 elements in the second dimension, this factor contributes to MaxHitRatio. \square

Figure 3.10 illustrates the *lpid* mapping function, ℓ , applied to a 24×24 matrix. The matrix is initially distributed (BLOCK,*) across six processors, and redistributed (CYCLIC(2), BLOCK) across a 3×2 processor grid. *lpids* (in bold italics) are superimposed on the matrix. The data hit ratio, using the mapping function is $\lceil \frac{4}{(2)(3)(2)} \rceil (2)(3)(2)(12) : 24 \times 24 = 144 : 576 = 1 : 4$. Figure 3.10 is an example of when $\gcd(z, p_0) = 1$ while Fig. 3.11 illustrates a situation where $\gcd(z, p_0) = 2$. The latter figure demonstrates the flexibility of permuting *lpids*. For instance, processors $lpid_0$ and $lpid_2$ could be permuted and the same data hit ratio would be achieved.

Lemmata 10 through 12 prove the following result.

Theorem 4 *For redistribution from (BLOCK,*) to (CYCLIC(c_0),#), where b_0 and c_0 are the respective block sizes in the row dimension, $z = b_0/c_0$ for an integer z , b_1 is the block size in the column dimension, $p_0 \times p_1$ defines the grid of processors for D_s , $n_0 \times n_1$ is the number of global data array elements where $n_i = b_i p_i$ $0 \leq i \leq 1$, the logical processor mapping function in Equation (3.4) achieves $\text{MaxHitRatio} = \lceil \frac{b_0}{c_0 p_0 p_1} \rceil c_0 p_0 p_1 b_1$.*

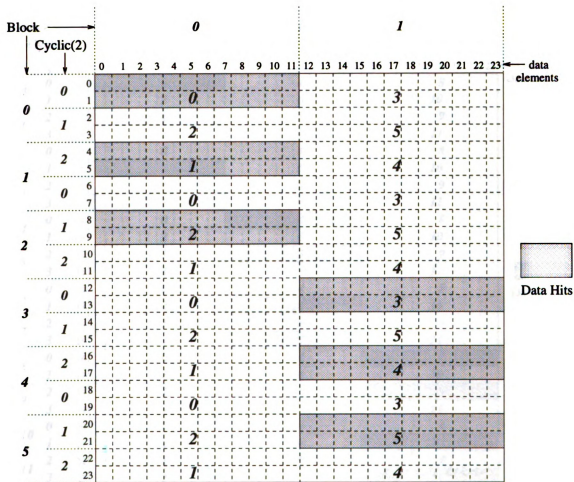


Figure 3.10: $(\text{BLOCK}_6, *)$ to $(\text{CYCLIC}_3(2), \text{BLOCK}_2)$ redistribution.

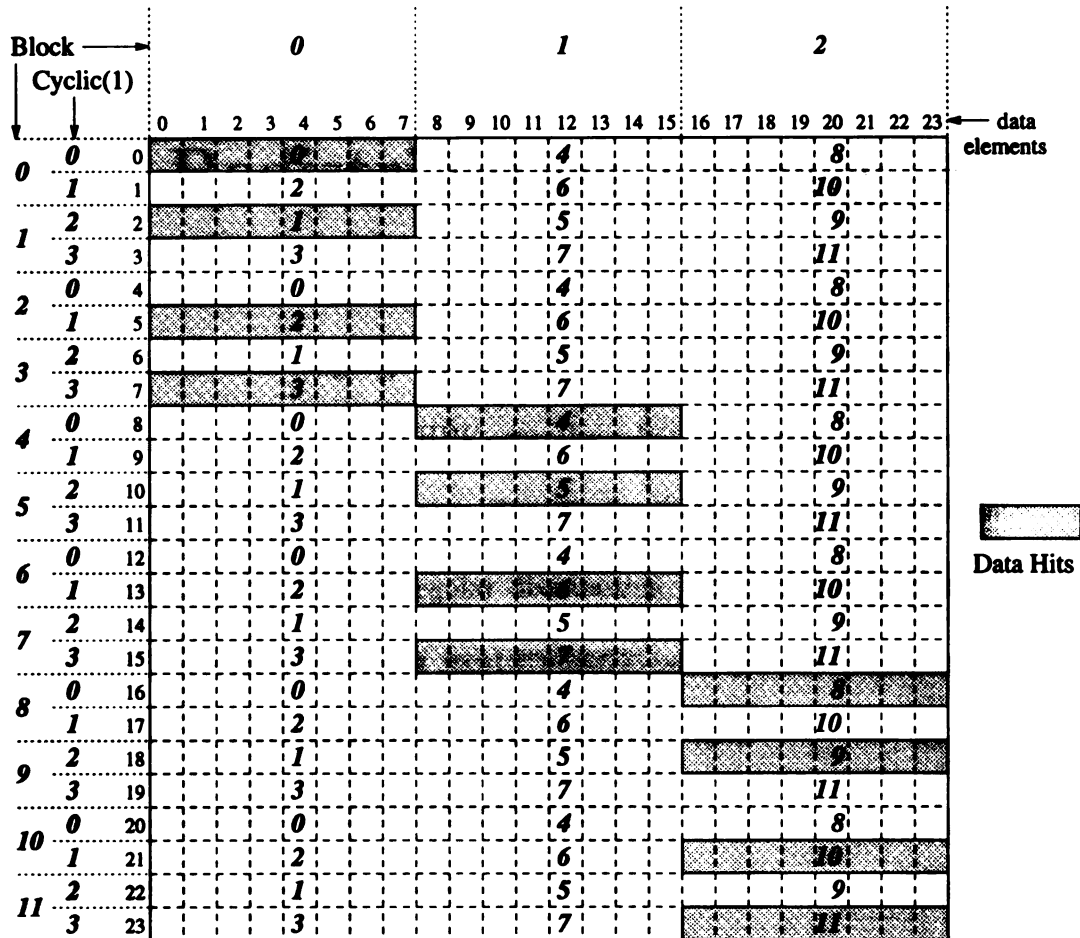


Figure 3.11: $(\text{BLOCK}_{12}, *)$ to $(\text{CYCLIC}_4, \text{BLOCK}_3)$ redistribution.

Chapter 4

Data Redistribution Library

HPF defines dozens of language intrinsics that could be implemented as libraries, significantly simplifying the role of HPF compilers. In this chapter, we propose a library, DaReL, for HPF-style redistribution. Specifically, DaReL supports multi-dimensional data redistribution for HPF’s regular distribution patterns, **BLOCK**, **CYCLIC**, and *****. The library is designed for `!HPF$ REDISTRIBUTE`; however, we envision its applicability to *implicit* redistributions that can occur within HPF programs; see Section 2.5. We discuss a number of salient issues affecting the design of a redistribution library for HPF including scalability and the respective roles of the compiler and the library.

We present a detailed overview of DaReL’s design and motivate the critical design choices. In contrast to other approaches, *e.g.*, [32], DaReL decouples processor send/receive set calculation from data exchange. We assert that this decoupling simplifies library design and facilitates multiple data exchange algorithm options. Data exchange is performed with MPI primitives, enhancing DaReL’s portability among distributed-memory platforms that utilize the emerging message-passing standard.

We discuss the advantages of using MPI for data redistribution, and we give a detailed description of DaReL's use of MPI point-to-point and collective communication, process topology, and derived datatype constructs. Performance and scalability analysis of DaReL is presented in Chapter 6.

4.1 Library Design Issues

There are a number of issues to consider in the design of a redistribution library for distributed-memory machines. For example, what are the advantages of implementing data redistribution as a library? What is the role of the HPF compiler vis-à-vis DaReL? How do we provide a flexible yet concise interface between the compiler and DaReL? What is the appropriate choice of message-passing mechanisms for implementing data movement among processors? What are the advantages of using MPI? How can we ensure DaReL's scalability? These issues are examined next.

4.1.1 Advantages of Software Libraries

The functionality of `!HPF$ REDISTRIBUTE` may be provided as a library that could be utilized by an eventual HPF compiler or programmer. Providing a rich set of libraries to data-parallel programmers is becoming increasingly popular because libraries offer some unique advantages [8]. For instance,

- A uniform interface provided by the library enhances the portability of programs among various distributed-memory architectures.

- Although a library provides a uniform interface to programs, the design and implementation of the library can explicitly exploit machine specific features to provide better performance.
- The program development cycle for the consumer of the library can be shortened as the correctness of libraries can be independently verified.
- The existence of libraries can simplify the tasks that the data-parallel compilers would otherwise have to perform.

4.1.2 Role of HPF Compiler

The compiler usually translates data-parallel Fortran code into a set of message-passing SPMD node programs [25]. Correspondingly, the implementation of a redistribution library is message-passing code to be linked with the node programs generated by the compiler. The message-passing programs are in turn compiled by the native machine compiler.¹ Given the number of possible data distributions, it would be impractical to provide different redistribution code for every possible HPF source/destination distribution scenario. A redistribution library ought to provide a *generic* redistribution capability for an arbitrary number of data and processor configurations and HPF distribution pattern possibilities. This is the approach taken in the design of DaReL.

Since redistribution does not contribute to the result of the computation being performed by the HPF program, its execution time must be minimized. If some of the functionality of `!HPF$ REDISTRIBUTE` can be performed at compile- rather than run-time, then the execution time of `!HPF$ REDISTRIBUTE` can be reduced, albeit at a cost of increased compile-time. This, however, is typically preferable for data-parallel

¹It is possible that HPF compilers may generate object code directly from HPF source code.

applications.

In order to redistribute data as called for by a specific invocation of **!HPF\$ REDISTRIBUTE**, a node program must know the identity of the processors from which it is to receive data (gather set) as well as the identity of processors to which it must send data (scatter set) as explained in Section 2.4. It is possible for these sets to be distinct. Figure 4.1 shows 15 data elements distributed across 5 processors under the **BLOCK** and **CYCLIC** patterns, respectively. Processor 0 owns data elements 0,1,2 under **BLOCK**, but it owns elements 0,5,10 under **CYCLIC**. Redistribution from **BLOCK** to **CYCLIC** causes processor 0 to scatter to processors 0,1,2 while it gathers from 0,1,3; the scatter and gather sets are distinct. In addition to calculating the scatter/gather sets, the data elements which are to be sent, kept, and replaced must be identified.

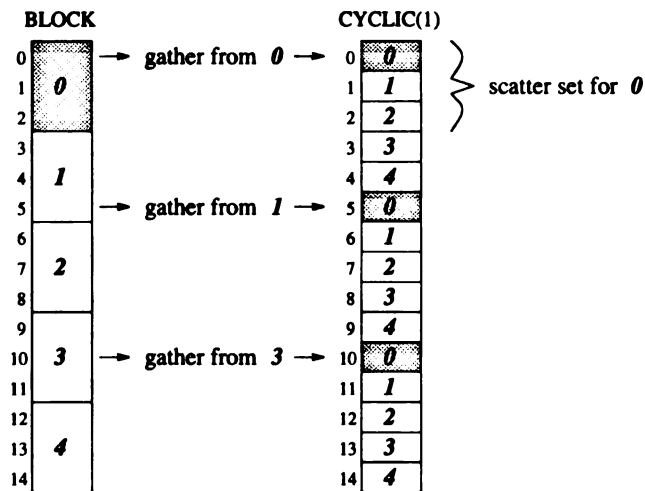


Figure 4.1: Distinct scatter-gather sets.

Following processor and data set identification, the data must be exchanged among processors utilizing message-passing primitives. The node programs use the communication services provided by the target machine to perform the data transfer. One could use unicast (point-to-point) or collective communication primitives depending

upon the services offered. The relative performance and scalability of the routines will be critical factors in deciding which set of primitives is most appropriate. We discuss communication primitive selection in greater detail in Section 4.2.

4.1.3 Use of MPI for Libraries

Data redistribution may utilize MPI to perform the necessary data exchanges. The primary advantages to using MPI are its portability, ease-of-use, and in particular, its support for parallel library development [48]. MPI uses communication contexts, first proposed in Zipcode [49], to provide safe communication spaces for processes. Communication contexts shield processes from unrelated, ongoing communication which otherwise may be interpreted, erroneously, as related to the current communication due to the asynchronous behavior of SPMD programs. This is particularly important in the context of libraries as different processes may enter a library asynchronously [50].

MPI also facilitates the specification of arbitrary logical process topologies with automatic support for Cartesian topologies. This mechanism frees the MPI programmer from defining communication in terms of hardware-dependent processor IDs. Process topologies are particularly important in the context of `!HPF$ REDISTRIBUTE` as the programmer has similar flexibility in specifying logical Cartesian processor configurations² in HPF with the `!HPF$ PROCESSORS` directive.

MPI specifies a number of point-to-point and collective communication primitives

²We use HPF's "processor" and MPI "process" interchangeably since they are functionally equivalent.

that can operate in conjunction with communication contexts and process topologies. The collective communication primitives that are of particular relevance to DaReL are the `MPI_Scatter`, `MPI_Gather`, and the `MPI_Alltoall` primitives.

4.1.4 Scalability

To be viable for SPC architectures, scalability is an important design criterion for parallel software libraries. A redistribution library ought to achieve good performance within a reasonably large range of physical processor configuration sizes and for various sizes of data that are to be redistributed. The scalability of data redistribution will in large part be dictated by the scalability of the MPI communication primitives used; see Section 6.2.3. The scalability of MPI communication routines may vary across different distributed-memory platforms as vendors are free to implement MPI routines in the most efficient manner possible and thus will attempt to optimize the primitives for their system. One can track the most efficient MPI routines as they are developed and incorporate them in the library. Additionally, factors such as the distribution patterns, size of data, and number of processors used are the primary factors affecting redistribution performance.

Another factor which may influence the scalability of a redistribution library is the size of data to be redistributed. Since the library must determine and store data ownership, the size of data can become an important issue if the overhead for performing this function grows as a multiple of data size.

4.2 DaReL Design Overview

DaReL is proposed as a scalable and portable MPI-based library to perform data redistribution. DaReL comprises two major modules: Compute Data Exchange Sets (CDES) and Exchange Data (ED). CDES determines the identity of the processes to (from) which a process sends (receives) data and which data elements are involved in the exchange. Using the ownership information computed by CDES, ED performs the actual data exchange. This section presents the design of these modules and DaReL's interface. We present a detailed explanation of DaReL's use of MPI process topologies, derived datatypes, and point-to-point and collective communication primitives.

4.2.1 DaReL Interface

Figure 4.2 presents the C-based interface between DaReL's calling environment, (*i.e.*, the SPMD application message-passing program created from HPF source by the compiler), and the library. All parameters, except the data to be redistributed, are unchanged by the library. **dim_size** is the number of data and process dimensions. **src_pconf** and **dest_pconf** are the source and destination MPI process configurations (P_s and P_t), respectively. **src_ptrns** and **dest_ptrns** describe the source and destination patterns (D_s and D_t) chosen from: **BLOCK**(b), **CYCLIC**(c), or *****. **src_global** and **dest_global** specify the global indices owned by a process under D_s and D_t . **local_data** is the process' application data which is to be redistributed. This parameter is modified by the library and returned to the calling environment.

```

redistribute (
    short          dim_size,          {IN}
    MPI_Comm       src_pconf,         {IN}
    MPI_Comm       dest_pconf,        {IN}
    struct distribution src_ptrns,     {IN}
    struct distribution dest_ptrns,    {IN}
    struct global_indices src_global,  {IN}
    struct global_indices dest_global, {IN}
    datatype       local_data         {IN-OUT}
)

```

Figure 4.2: C-based interface for DaReL

Figure 4.3 gives an explanation of each parameter in the interface. Each parameter is marked either IN or IN-OUT, the former indicates a value that the calling environment provides to the library, the latter represents a value that is provided by the calling environment and is modified by the library upon return to the calling program. Though many of the parameters are passed as pointers, we use the IN, OUT, IN-OUT formalism of Fortran 90 [51] to indicate which variables are changed by DaReL's operation and which are not.

Figure 4.4 illustrates a Cartesian to process rank mapping. The source topology, P_s , is a (3×2) grid and P_t is a (2×3) grid. The figure demonstrates the association of a process in the source, $s(2,1)$, and destination, $d(1,2)$, grids and its rank within the process group to which it is mapped. The process' rank is then mapped to a physical processor ID of the machine. Since MPI message-passing routines use rank to identify the processes, the association of process rank to physical processor ID is *not* relevant to DaReL, *i.e.*, DaReL performs its function in MPI process topology space. The two Cartesian topologies in Fig. 4.4 have different shapes but the same

dim_siz	The number of data and process dimensions; these must be the same to avoid ambiguity. Although, it is legal to have data dimensions which remain <i>undistributed</i> ; these dimensions must be denoted by the special symbol <i>*</i> in the corresponding placeholder.
src_pconf	an MPI communicator which contains information regarding the source distribution process configuration. It stores the number of process dimensions, the number of processes in each dimension, <i>i.e.</i> , <i>P</i> in the HPF example in Fig. 2.3, and the identity of the local process.
dest_pconf	an MPI communicator which contains information regarding the destination distribution process configuration. It stores the number of process dimensions, the number of processes in each dimension, <i>i.e.</i> , <i>Q</i> in the HPF example in Fig. 2.3, and the identity of the local process.
src_ptrns	array[0.. dim_siz -1] of distribution structures; stores the pattern and block size information for the source distribution; each array element corresponds to the distribution for that dimension.
dest_ptrns	array[0.. dim_siz -1] of distribution structures; stores the pattern and block size information for the destination distribution; each array element corresponds to the distribution for that dimension.
src_global	array[0.. dim_siz -1] of global_indices structures; stores the global indices corresponding to the local process' data under the source pattern.
dest_global	array[0.. dim_siz -1] of global_indices structures; stores the global indices corresponding to the local process' data under the destination pattern.
local_data	array[0.. dim_siz -1] of type datatype . the actual data to be redistributed. The type of data, <i>e.g.</i> , integer or real, can be interpreted as a byte stream. Note: the data prior to redistribution could be passed as an IN and the redistributed data returned by another parameter as an OUT. We choose to provide one parameter and make it IN-OUT.

Figure 4.3: Parameters for the invocation of DaReL

size. It is possible in HPF to have different sized configurations for the source and destination. With different sized configurations, a process may be a member of only one of the two configurations. The association between Cartesian coordinates and process rank must be known within DaReL for both configurations. For instance, if a process has a rank defined in the source topology, but not in the destination topology, then it will scatter data, but *not* gather data.

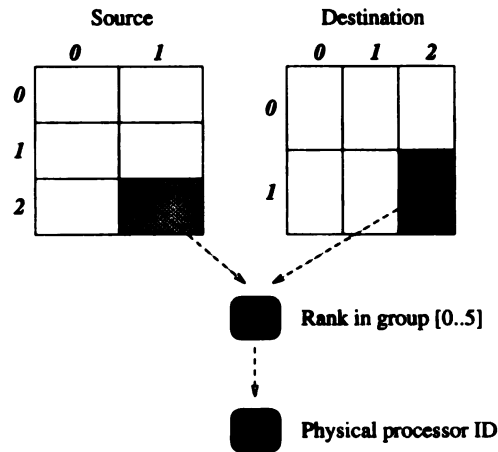


Figure 4.4: Different shaped process configurations.

To create a Cartesian topology, MPI provides `MPI_Make_cart`. In functionality, it is very similar to HPF’s `PROCESSORS` construct since it defines a logical topology in terms of the number of dimensions and the extent in each dimension. Since the compiler has a “global view” of the entire application and DaReL does not, it is reasonable for the compiler to determine global performance optimizations such as the logical to physical processor mapping as discussed in Section 2.6.³ DaReL inherits the processor mapping as determined by the calling environment. Thus, it follows that the creation of the Cartesian topologies, to be used within DaReL, occur within the

³Alternatively, command-line arguments or the run-time system may determine the best mapping [3].

calling environment as well. There are two reasons for this. First, **MPI_Make_cart** facilitates choosing a specific logical to physical mapping with the **reorder** parameter; and second, the Cartesian topologies are most likely used in the calling environment already for message-passing that occurs prior to, and after, the invocation of **!HPF\$ REDISTRIBUTE**. DaReL assumes that **src_pconf** and **dest_pconf** have been created with **MPI_Make_cart** in the calling environment.

4.2.2 Compute Data Exchange Sets (CDES)

CDES utilizes the MPI Cartesian topology routines to manipulate the process sets since these routines naturally correspond to logical processor topologies as defined in HPF. Similar to [31], each process computes the process sets for data exchange based on the global data elements it “owns”⁴ under each of D_s and D_t , respectively. DaReL relies on the calling environment to furnish global index information which could be provided either by the compiler or the run-time system.

To determine the scatter set, a process must identify the ownership of its “current” data under the destination distribution. In other words, by mapping D_t over the initial global indices, **src_global**, the scatter set is computed. To determine the gather set, the process identifies the ownership of its “future” data under the source distribution. In other words, by mapping D_s over the destination global indices, **dest_global**, the gather set is computed. Given the global indices owned by a process under each of D_s and D_t , CDES performs these computations for each data dimension as follows.

⁴HPF’s owner computes rule.

Consider one dimension of data. Given n data elements distributed across p processes, Equations 4.1 and 4.2 compute the process ID, p_i , that owns data element with global index j for **BLOCK**(b) and **CYCLIC**(c) respectively, where $0 \leq i \leq p - 1$, and $0 \leq j \leq n - 1$. For *****, a process owns all data in the corresponding dimension.

$$p_i = j/b \quad (4.1)$$

$$p_i = (j/c) \bmod p \quad (4.2)$$

Since each dimension of data must have a corresponding distribution pattern and number of processes onto which the data is mapped, Equations 4.1 and 4.2 are applied to each dimension *independently*.

Note that a process need not compute ownership for each data element it owns, rather process ownership along each dimension suffices. Thus, for multidimensional data, a Cartesian process ID is obtained. The Cartesian process ID is subsequently used in an MPI message-passing primitive in ED. The identity of source and destination distribution patterns is inconsequential to the operation of ED. Our approach alleviates the need for numerous closed form expressions and algorithms [31, 32] for different source/destination pattern combinations. The computational complexity of CDES is $O(n_0 + n_1 + \dots + n_{m-1})$ when n_0, n_1, \dots, n_{m-1} are the data extents in each dimension of an m -dimensional redistribution. If CDES were to compute data ownership for each data element, rather than by dimension, the complexity would be $O(n_0 \times n_1 \times \dots \times n_{m-1})$.

Figure 4.5 illustrates the use of CDES by process $(1,1)$ in a **(BLOCK,BLOCK)** to **(CYCLIC,CYCLIC)** redistribution to determine the set of processes to which it must scatter data. The source and destination process configurations are the same, 4×2 . Under $D_s = \text{(BLOCK,BLOCK)}$, process $(1,1)$ owns global indices j_r , where $4 \leq j_r \leq 7$ in the row dimension, and global indices j_c , where $0 \leq j_c \leq 1$ in the column dimension. Since D_t is **CYCLIC** in each dimension, Equation (4.2) is applied to the row dimension, *i.e.*, $p_{i_r} = (j_r/1) \bmod 4$ where $4 \leq j_r \leq 7$ and in the column dimension, *i.e.*, $p_{i_c} = (j_c/1) \bmod 2$ where $0 \leq j_c \leq 1$, respectively. The ownership for scattering data in each dimension is shown in the right portion of the figure in bold-italic.

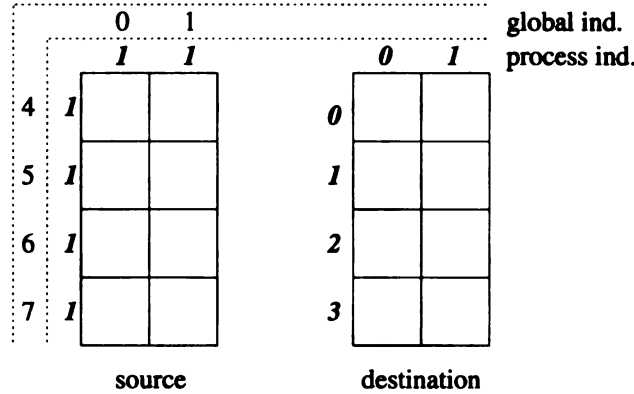


Figure 4.5: CDES example for process $(1,1)$

Storing ownership information for each local data item is unnecessarily expensive in terms of memory cost. Thus, CDES builds an ownership record for each recipient of local data, and it builds a record for each sender of data for which the local process is a receiver. The description of ownership is done with *displacement*, *block size* and *stride* arguments. The displacement specifies the location of the first element, the block size indicates the number of contiguous elements, and the stride specifies the stride between blocks. This ownership information is stored on a per-dimension basis,

e.g., for a two-dimensional array, there are ownership records for the row and column dimensions, respectively. This method of storing ownership information is minimal in terms of storage cost and maps well to MPI derived datatypes which will be explained in detail in Section 4.2.5.

4.2.3 Partitioning CDES Functionality

Figure 4.6 illustrates two high-level approaches to implementing CDES that differ, not in their functionality, but in their partitioning of tasks between compile- and run-time. In Fig. 4.6(a) all of CDES is performed at run-time while in Fig. 4.6(b), CDES is partitioned between compile- and run-time. The motivation for “elevating” part of CDES to compile-time is to reduce program execution time at the expense of increased compile-time. The compile-time portion of CDES involves partially computing the data exchange sets. Equations 4.1 and 4.2 could be performed at compile time if all pertinent values are known at compile-time. The run-time portion of CDES completes the computation, using a process’ own ID. The CDES compile-time service would be provided as a library to the HPF compiler, thus designated `hpf.a`. All run-time library services are found in `node.a`, so designated since they are linked by the node programs. One would expect the option in Fig. 4.6(b) to be preferable in all situations, since it decreases program execution time. Unfortunately, the CDES compile-time module can only be performed if all relevant arguments, *e.g.*, data size and number of processes, are known at compile-time as they are in the example code segment in Fig. 2.3. Since HPF allows these arguments to be variables whose values remain

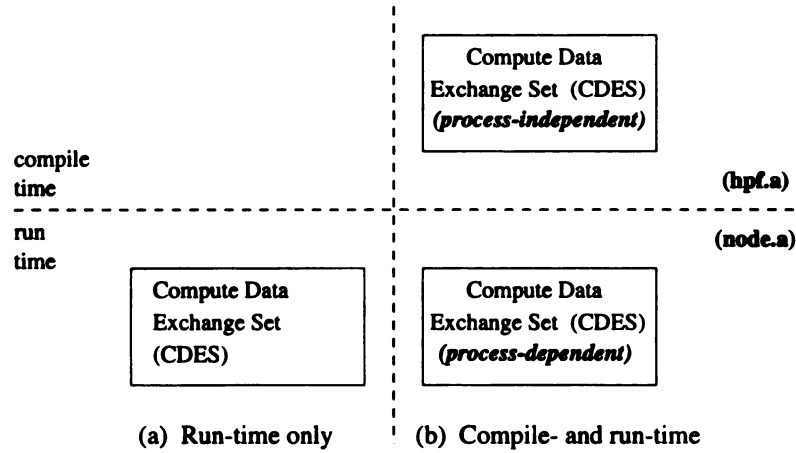


Figure 4.6: Partitioning CDES

unknown until run-time, the CDES variant in Fig. 4.6(a) is necessary. The extra run-time computation needed is of little consequence as CDES execution time is small relative to ED execution time, see Chapter 6. DaReL is currently implemented with the CDES version of Fig. 4.6(a).

4.2.4 Exchange Data (ED)

ED performs the data exchange among the set of processes computed in CDES. ED utilizes MPI point-to-point or collective communication primitives depending upon the services offered on a particular system and their relative scalabilities. Additionally, ED uses MPI derived datatype facilities to avoid the overhead of packing data into a message buffer at the sender and unpacking data at the receiver.

This section presents a more detailed explanation of the use of MPI in the ED module. Given the communication pattern required by data redistribution, it is natural to consider the MPI collective communication services: **MPI_Scatter**, **MPI_Gather** and **MPI_Alltoall** in ED. Additionally, MPI point-to-point routines are viable choices as

well. Broadcast, *i.e.*, **MPI_Bcast**, is not a valid choice for redistribution since it sends the *same* data to each process. In some cases, redistribution performance using point-to-point message-passing may be better than that of redistribution using collective routines if the latter are not optimized for the target architecture. See Chapter 6 for more detail regarding performance.

Figure 4.7 illustrates the algorithm used by ED to perform data exchange using MPI point-to-point message-passing. In the point-to-point algorithm, each process sends *Scount* data elements of type *Stype* from an offset of *Sdispls* in *Sbuf* to all other processes. *Stype* is a derived datatype; see Section 4.2.5 for a discussion of DaReL’s use of MPI derived datatypes. Each process receives data from all other processes where the placement of the data in *Rbuf* is described by *Rcount*, *Rtype*, and *Rdispls*. The boolean arrays *send* and *recv* indicate whether the indicated process is a recipient or a sender of data, respectively.⁵ Recall in Fig. 4.1 that the scatter and gather sets may be distinct.

Figure 4.8 illustrates the algorithm when one of the collective communication paradigms is used to exchange data. If **MPI_Scatter** or **MPI_Gather** are used, then a loop over all group members where each group member is the *root* of one operation is needed. For **MPI_Alltoall**, a single invocation is needed.

Note that each of the collective communication routines in Fig. 4.8 uses variants of the base operation with “v” appended, *e.g.*, **MPI_Gatherv**. The “v” extends the primitive to allow for specification of displacements from the starting address of a buffer, and it facilitates different sized data buffers to be specified for the distinct processes

⁵From the local process’ perspective depending upon the data computed by CDES.

```

Input:  Sbuf
Output: Rbuf

For root = 0 To group_size-1
  If (root == my_rank)
    For rank = 0 To group_size-1
      If ((rank != my_rank) AND (send[rank] == TRUE))
        MPI_Send( Sbuf, Sdispls[rank], Scount[rank], Stype[rank] )
      End If
    End For
    If (send[root] == TRUE)
      Copy relevant portion of Sbuf to Rbuf
    End If
  Else If (recv[root] == TRUE)
    MPI_Recv( Rbuf, Rdispls[root], Rcount[root], Rtype[root] )
  End If
End For

```

Figure 4.7: ED using MPI point-to-point communication

in the collective group. This flexibility is necessary for generic data redistribution as provided by DaReL.

Each of the MPI point-to-point and collective routines takes a single communicator as an argument (not shown) and all participants in the collective operation must specify consistent arguments, *e.g.*, all members must specify the same value for *root*. When the communicators (process topologies) corresponding to P_s and P_t in a data redistribution are equivalent, then any of the MPI routines discussed above may be used. If P_s and P_t are the same size, but have different shapes, one communicator suffices so long as an association between each process in the source and each process in the destination communicators can be made when the topologies are created. When the source and destination topologies have different sizes, however, there is a problem

```

Input:  Sbuf
Output: Rbuf

Switch (collective_type)
  Case Scatter:
    For root = 0 To group_size-1
      MPI_Scatterv( Sbuf, Sdispls, Scount, Stype, Rbuf, Rdispls, Rcount, Rtype, root )
    End For
  Case Gather:
    For root = 0 To group_size-1
      MPI_Gatherv( Sbuf, Sdispls, Scount, Stype, Rbuf, Rdispls, Rcount, Rtype, root )
    End For
  Case All2all:
    MPI_Alltoallv( Sbuf, Sdispls, Scount, Stype, Rbuf, Rdispls, Rcount, Rtype, root )
End Switch

```

Figure 4.8: ED using MPI collective communication

using **MPI_Alltoall**, since at least one process is in either the source or destination, but not in both. **MPI_Scatter** or **MPI_Gather** may be used, but only if a new group communicator is created for each scatter or gather. Creating groups would contribute to the overhead of the entire operation and does not scale well as the group size increases. MPIX [52] proposes extensions to MPI which would overcome the above limitation of collective communication. Table 4.1 summarizes whether or not an MPI routine can be used, based on the relationship between source and destination process topologies.

4.2.5 Standard and Derived Datatypes

To avoid costly packing of data at the sender and unpacking of data at the receiver, DaReL utilizes MPI's derived datatype facility. MPI has great flexibility in specify

Table 4.1: MPI Communication Primitive Choices

Primitive	$ P_s = P_t $	$ P_s \neq P_t $	$P_s \neq P_t$
Shape	same	distinct	distinct
Pt-to-pt	Yes	Yes	Yes
Scatter	Yes	Yes	Yes
Gather	Yes	Yes	Yes
All-to-all	Yes	Yes	No
Broadcast	No	No	No

datatypes for message passing. MPI manages the system resources for buffering messages and storing their internal representations. The buffers are not directly user-accessible, thus, MPI provides “handles” in user space to access system objects. This design hides the internal representations of MPI data structures allowing for similar calls between dissimilar languages (C and Fortran) and also avoids typing rule conflicts between these languages. MPI defines a number of standard datatypes, *e.g.*, `MPI_INT`, `MPI_FLOAT`, `MPI_CHAR`, `MPI_BYTE`.

To specify the complex data mappings necessary for efficient redistribution, DaReL utilizes MPI’s facility for *derived* datatypes. MPI standard datatypes facilitate specifying the data content of a message when the data is contiguous and of the same datatype. Derived types extend MPI datatype specification by allowing for non-contiguous and mixed-type data in a message buffer. DaReL makes use of the former feature of derived types.

In the redistribution example of Fig. 2.3, redistribution requires processes to send multiple data items to a particular destination from locally *non*-contiguous locations. For large data sizes, it is impractical to send data items individually due to sending

latency overhead costs, thus DaReL's approach is to aggregate data destined for a specific process. Data can be aggregated into the sending buffer, and retrieved into non-contiguous data locations at the receiver, by means of the **MPI_Pack** and **MPI_Unpack** facility, respectively. This approach, however, introduces extra overhead at both the sender and receiver. At the sender, data is copied from the original non-contiguous locations in user-space to the buffer for sending, also in user-space; the inverse copying is performed at the receiver. The flexibility of MPI derived datatypes alleviates this costly packing/unpacking overhead.

DaReL uses the **MPI_Type_vector** and **MPI_Type_struct** primitives to specify the message buffers for the exchange of data. The former primitive facilitates replication of a datatype mapping onto non-contiguous memory locations. The ownership data computed by CDES is used to form the **MPI_Type_vector** argument for each dimension of data. There is a limitation of **MPI_Type_vector**, however, when used in conjunction with collective communication primitives, necessitating the use of MPI's most general derived type constructor, **MPI_Type_struct**. We illustrate this limitation with an example involving the scatter primitive.

Figure 4.9 shows a **BLOCK** to **CYCLIC** redistribution in one dimension among three processes. We focus on the operation to be performed by process *1* which scatters its local data set to the other two processes (shaded) and retains one element of data itself. Using **MPI_Type_vector**, process *1* would create datatypes for each of processes *2* and *0* specifying a block size of 1, a stride of 3, and a total count of 2. The displacement from the beginning of the local data is not an argument to **MPI_Type_vector**, but it is an argument to **MPI_Scatterv**. Thus, process *1* would

specify displacements of 0 and 1 for processes 2 and 0, respectively. The difficulty arises in that **MPI_Scatterv** interprets the displacement as follows,

$$\text{send_buf} + \text{displs}[i] \times \text{extent}(\text{sendtype}[i])$$

where `send_buf` is the sending buffer, `displs[i]` is the displacement for process *i*, and `extent(sendtype[i])` represents the size of the datatype, `sendtype[i]`. The extent of a datatype generated by **MPI_Type_vector** is calculated to be `count × block size × stride`. In the current example, the extent would be 6. Thus, the actual displacements as used by **MPI_Scatterv** would be 0 for process 2 and 6 for process 0. This, of course, is erroneous since process 0 ought to start receiving data from displacement 1 as explained earlier. This problem occurs regardless of the choice of collective communication primitive. One alternative is to use the pack/unpack facilities as described earlier. Of course, we wish to avoid the overhead that this alternative implies.

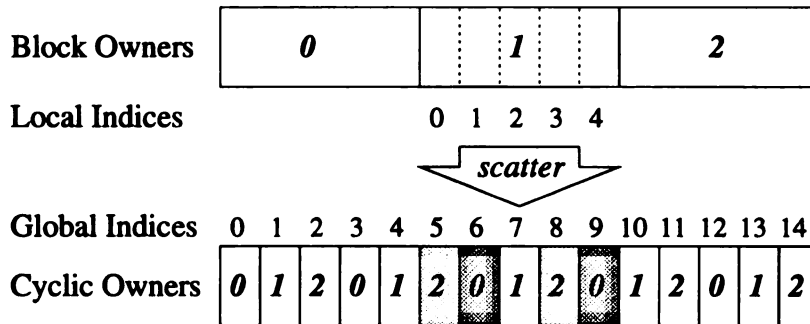


Figure 4.9: Limitation of derived types and collective communication

MPI provides a facility that allows the programmer to override the default datatype extent calculation. **MPI_Type_struct** together with **MPI_UB** provide the nec-

essary functionality. `MPI_UB` is a “pseudo-datatype” used to mark the upper bound of a datatype. We use this facility to specify the extent of a datatype to be correct for use in a collective communication operation. `MPI_Type_struct` allows the most general datatype construction for this purpose. For the above example, we would define the upper bound of the datatype, `send.type` to be 1. Thus, the displacements for processes 2 and 0 would be 0 and 1, respectively, as is needed for correct operation.

4.2.6 Scalability

Beyond the basic redistribution operation described in this chapter, DaReL ought to provide *scalable* performance over large range of data and process configuration sizes. As alluded to earlier in this chapter, the performance of the MPI point-to-point and collective message-passing primitives may vary for different sizes of data and numbers of processes. Additionally, the choice of distribution patterns for D_s and D_t may affect redistribution performance. We assert that DaReL must provide the best performance possible for any given situation.

Therefore, DaReL shall provide the ability of selecting, at run-time, which of the primitives from Table 4.1 is best suited to the given situation. While DaReL may not be privy to the mapping of the process topologies onto the physical processors,⁶ it can determine the *number* of physical processors and the *shape* of the configuration onto which the logical topology is mapped. HPF supports two intrinsics `NUMBER_OF_PROCESSORS` and `PROCESSOR_SHAPE` [3], for inquiring about the physical

⁶The calling environment may provide this information.

nodes. If the logical topology is larger than the number of physical processors in the underlying hardware, then more than one logical processors will be mapped to some or all of the physical nodes. If the aforementioned HPF intrinsics are supported in the environment that DaReL is operating within, then they may provide useful performance information for selecting the appropriate primitive.

Chapter 5

Quantifying Scalability

The term *scalability* has been used extensively in the parallel processing community to characterize the ability of parallel architectures and algorithms to exhibit greater *performance* as more processors are employed to solve a problem. However, the term “performance” is equivocal as it may be used to identify widely varying algorithmic or architectural properties: speedup, execution time, processor speed or efficiency, or the quality (accuracy) of a solution. Many of these properties have been used either separately or in conjunction to describe the scalability of parallel algorithms and architectures. The ambiguity regarding scalability leads us to ask several questions: What is a scalable algorithm and a scalable architecture, and how are their relative scalabilities related? Can a universal definition of scalability be applicable to all scenarios? How can scalability be quantified? If scalability can be quantified, which property best describes it, or is scalability a combination of factors? This chapter shall attempt to address these questions.

A *scalable architecture* is one which is designed to offer a proportional performance

increase with a balanced increase in the number of processors, memory capacity and bandwidth, and network and I/O bandwidth of the machine without changing the basic underlying design. Typically, architectures based on a distributed-memory model have been characterized as *scalable* due to their ability to provide such balanced increases. While the above definition captures the essence of scalability as it relates to parallel architectures, it does not provide guidance for assessing *relative* machine scalabilities. It does not suffice to simply compare the peak performance of machines since *sustained* performance typically provides a more accurate measure of the performance seen by the end-user. Sustained performance, however, is a function of the parallel algorithm used. In fact, not only the algorithm, but its implementation affects sustained performance. The implementation of an algorithm includes its data decomposition, message-passing paradigm, degree of parallelism, and processor synchronization, all of which affect performance [8]. Henceforth, when we use the term *algorithm*, we mean parallel algorithm and its implementation detail.

A *scalable algorithm* can utilize additional machine resources, *e.g.*, additional number of processors, to increase algorithm performance. Increased algorithm performance may be measured as producing a more accurate result or producing an equivalent result in less time. As with the definition for scalable architecture, this definition describes the property of scalable algorithms but does not assist us in comparing relative scalabilities of parallel algorithms. The performance of an algorithm will be greatly influenced by the underlying machine. For instance, if an algorithm frequently performs barrier synchronization, the efficiency of the barrier implementation on the underlying machine will greatly affect the execution time of the algorithm. Similarly,

the memory capacity of the machine will influence the size of problem, or accuracy, that can be solved with an algorithm.

We conclude that scalability quantification necessarily involves both the parallel algorithm and the parallel target machine. Thus, we shall examine the scalability of *algorithm-machine* (AM) pairs. A number of researchers have formulated scalability models based on algorithm-machine pairings, some of which we shall survey in Section 5.3.

5.1 Examples of Ambiguity in Scalability

We demonstrate the equivocal nature of the term *scalability* and demonstrate the difficulty in quantifying the scalability of AM pairs with two examples. Our goal in the following examples is to assess which among several AM pairs is “the most scalable” for the application.

In the first example, suppose we have two algorithms A and B and two machines m_1 and m_2 from which to choose to solve a hypothetical problem. We define an *end-user* to be a consumer of the result(s) generated by an algorithm on a particular machine. We compare the performance of algorithms A and B executing a constant amount of work on m_1 and the relative performance of machines m_1 and m_2 with the same algorithm, A. Figure 5.1 shows the execution times of pairs Am_1 , Bm_1 , and Am_2 as a function of different sized processor configurations. On m_1 , B is superior to A up to approximately 35 processors, while A achieves lower execution times between

35 and 70 processors.¹ The execution times of Bm_1 begin to increase after only 60 processors. Executing algorithm A, m_2 is superior to m_1 up to about 55 processors; m_1 is preferable between 55 and 70 processors. The best execution time is achieved on 70 processors with Am_1 .

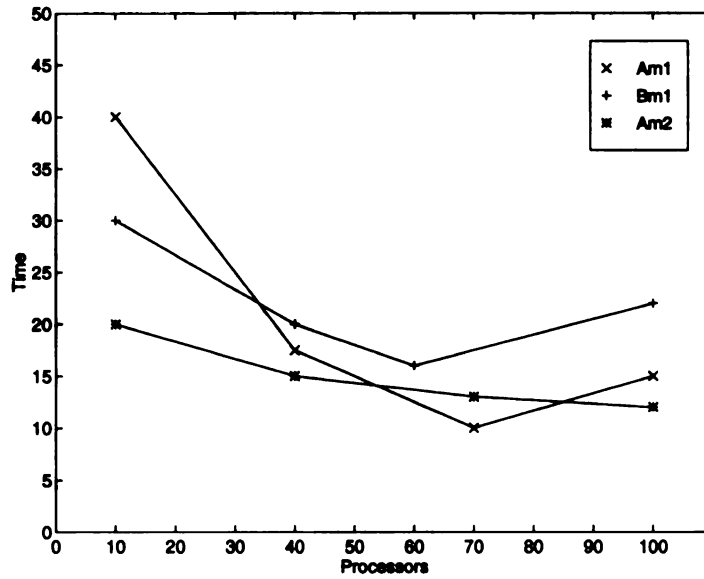


Figure 5.1: Comparison of three AM pairs

The example clearly demonstrates the importance of considering the performance of an algorithm together with the parallel machine upon which it is executed. Even in the simple example, however, it is unclear which AM pair is the most scalable; to a large extent it depends upon the goals and constraints of the end-user. To determine which AM pair is the “most scalable,” we need to answer the following questions. Does the end-user have access to as many processors of m_1 or m_2 as desired? Is the end-user’s goal to achieve minimum execution time for constant work as depicted in Fig. 5.1? Or, is the user concerned with scaling work to achieve better solution

¹We are not interested in execution times for beyond 70 processors since they begin to increase. We shall elaborate on this phenomenon in Section 5.5.

accuracy? The answers to these questions will affect the relative “scalabilities” of the AM pairs.

In the second example, we compare two real matrix multiplication algorithms, a_1 and a_2 , executing on the same machine M , an SP- x . Let the base amount of work be to compute the product of two 100×100 matrices of single-precision (4-byte) real numbers. We assume two hypothetical end-users, User₁ and User₂, each of whom wants to choose “the most scalable” matrix multiplication algorithm among a_1 and a_2 for his/her application. The difficulty in choosing “the most scalable” AM pair arises when the goals of each end-user differ. User₁’s application requires computing the product of a set of 100×100 matrices and thus wants to choose the fastest AM pair to perform these multiplications. User₂ wants to increase (scale) the base work performed to achieve a more accurate result for his/her application and thus requires matrix multiplications up to a maximum size of 800×800 . Thus, User₂ is less concerned with execution time and more concerned with the quality of the solution. Figure 5.2 plots the execution time (vertical axis) as the matrix size is increased (horizontal axis) for the two algorithms a_1 and a_2 using 16-processor configurations. a_1 is faster than a_2 on a 100×100 matrix (dashed line). Even as matrix size is increased, a_1 has consistently lower execution times than a_2 . This is due to the implementation detail of a_1 : one of the two matrix operands is replicated among all processors thus avoiding interprocessor communication. Such replication, however, limits the size of the matrix that can be computed as the aggregate memory requirements grow quickly with increasing matrix size. Algorithm a_1 ’s large memory requirements limit its ability to scale the problem size beyond a 540×540 matrix computation. Algorithm a_2 ,

while slower, can scale up to an 800×800 matrix product. In a_2 , the matrix operands are distributed, thus reducing the per-node memory requirement but incurring inter-processor communication. Clearly, a_1 is unable to satisfy the needs of User₂; thus, a_2 would be the most scalable choice. Algorithm a_1 , however, best suits the needs of User₁ since it can execute sets of 100×100 products in less time.

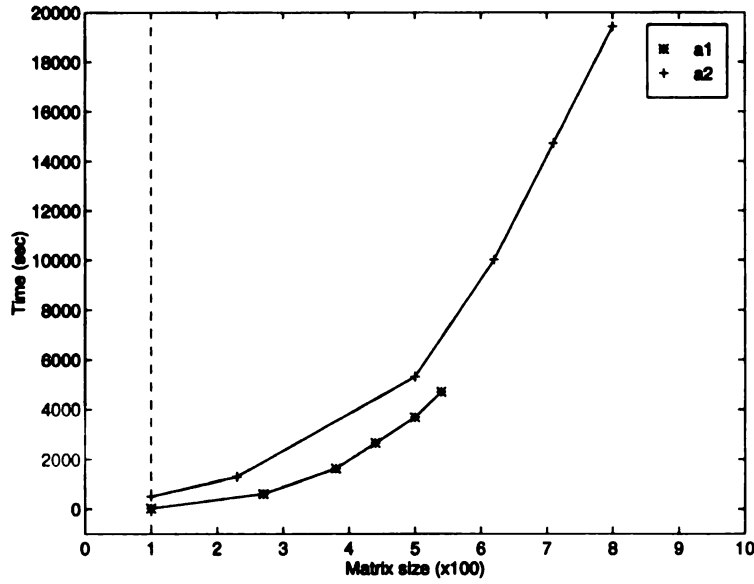


Figure 5.2: Scaled work *vs.* execution time of two AM pairs

5.2 Goal-directed Scalability Metrics

The above example illustrates the difficulty in quantifying scalability due primarily to the inability of a single metric to adequately capture variations in end-user requirements. We assert that a framework for quantifying scalability must account for end-user constraints and requirements. Rather than achieving higher speedups or processor efficiencies, our thesis is that the end-user is chiefly concerned with achieving the following criteria with an AM pair:

- By scaling the amount of work, increase the quality (accuracy) of the generated solution given more time and/or machine resources. The amount of acceptable execution time may be constrained by the end-user. The extent to which quality can be increased is a measure of the scalability of the AM pair.
- For a fixed solution quality, provide a large range of execution times and processor configuration sizes (number of processors) that can achieve the indicated solution quality. Large ranges will facilitate greater choices for the end-user. The size of these ranges is a measure of the AM pair's scalability.

This chapter presents a simple end-user-oriented framework for quantifying the scalability of AM pairs together with a facility to aid in visualizing AM pair behavior. Specifically, we propose several metrics for quantifying the scalability of AM pairs based on the needs of the end-user as captured by the above criteria. We develop a mathematical framework to characterize the properties of AM pairs, *e.g.*, communication overhead, and we propose a framework for visually illustrating an AM pair's performance over ranges of number of processors and scaled work. The benefit of a three-dimensional illustration is that a number of critical parameters, *e.g.*, memory capacity and communication overhead, can be viewed together in a unified manner. Additionally, the end-user's chosen scalability metric(s) can be visualized as distinct points or lines along the generated surface. We present a detailed case study that illustrates the utility of our framework in comparing the relative scalabilities of distinct AM pairs for performing matrix multiplication. In Chapter 6, we examine extensions to this framework geared specifically to quantifying the scalability of data

redistribution.

5.3 Metrics for Quantifying Scalability

Perhaps the most widely accepted metric for measuring the performance increase gained by parallel systems is *speedup*. Speedup is usually computed as the ratio of execution time on a single processor to execution time on a parallel machine. There are different categories of speedup: fixed-size, fixed-time, and memory-bounded speedup. Using *fixed-size speedup*, where the amount of work remains constant, Ware [53] defined *Amdahl's Law* based on Amdahl's earlier research [54]. Gustafson *et al.* [55] introduce the concept of *scaled speedup* [55] where the amount of work is increased, as the number of processors increases, to achieve a more accurate result. In this context, Gustafson defined *fixed-time speedup* [56] where execution time is fixed and the ability to scale work is measured. While Gustafson's scalability metric constrains execution time, the amount of aggregate memory may also impose a limitation on the ability of scaling work. *Memory-bounded speedup* [57] measures the ability to scale work as the memory capacity increases with the number of processors.

A number of models for quantifying the scalability of algorithm-machine pairs have been proposed. *Isoefficiency* [58] measures scalability by assessing an AM pair's ability to maintain a constant processor efficiency as the problem size and number of processors are increased. In this framework, efficiency is defined to be the ratio of speedup to the number of processors. Extensions to isoefficiency to account for the relation between the amount of memory used by the algorithm and size of available

machine memory is proposed in [59]. As a problem is scaled, however, it can be problematic to obtain execution time on a single processor due to lack of memory capacity. Obtaining single processor execution time is a necessary component for measuring speedup.² The *isospeed* metric [57] removes the dependence on speedup by measuring an AM pair’s ability to maintain average speed as the problem size and number of processors are increased. In this context, speed is defined to be the ratio of work to execution time. A comprehensive study of various scalability measures is presented in [60].

While measuring scalability in terms of efficiency or speed may be a viable approach, we believe that the primary focus ought to be the quality of the computed solution. Other researchers have developed benchmarks which measure a system’s ability to generate quality solutions for a given fixed time (SLALOM [61]) or as a function of increasing time (HINT [62]). We strive to apply this concept of solution quality to measuring the scalability of an AM pair.

Definition 2 *The Quality of Solution (QoS) is a measure of the accuracy or precision of a computed result. It is a function of the amount of work performed by an algorithm-machine pair.*

We illustrate QoS with a simple example. Let the objective be to compute $\sin(x)$ to a specified accuracy. Let the range of needed accuracies be $[5, 35]$ significant digits. The higher the accuracy, the larger the discretization of the $\sin(x)$ curve. Thus, obtaining a higher accuracy may require more computation time and memory

²Systems that support virtual memory may obtain superlinear speedups.

resources. For $\sin(x)$, we measure QoS by the number of digits of accuracy that are computable by an AM pair. Thus, $QoS_{min} = 5$ and $QoS_{max} = 35$. For four hypothetical AM pairs, Fig. 5.3 illustrates the maximum achievable QoS of each pair: 7.5, 20, 40, and 47.5, respectively. The dashed lines denote QoS_{min} and QoS_{max} , respectively. All AM pairs achieve QoS_{min} while only pairs 3 and 4 achieve the maximum desired accuracy. If the end-user is concerned primarily with the solution

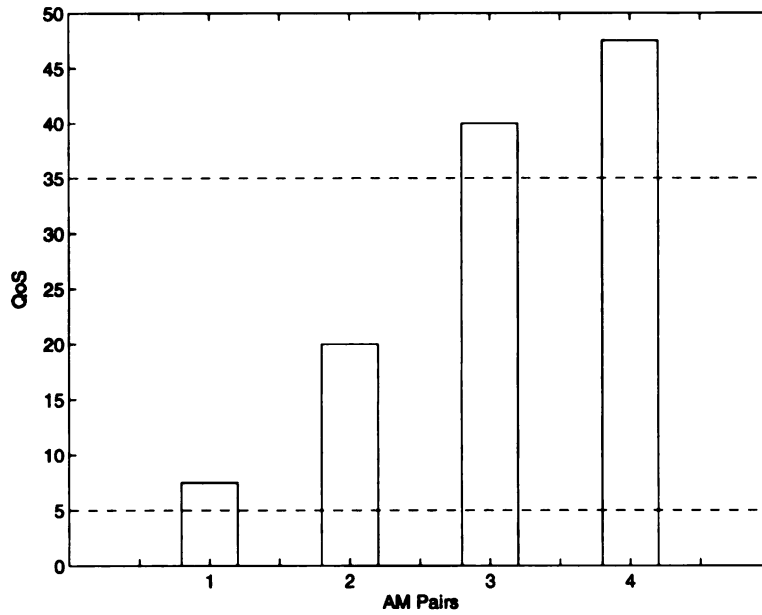


Figure 5.3: Quality of Solution for $\sin(x)$.

accuracy, then processor speedup or efficiency may be of little use since there is no guarantee of a correlation between those metrics and solution quality. However, these metrics may provide a secondary measure for assessing which of pairs 3 or 4 ought to be chosen given their ability to provide the highest accuracies desired. Later in this section, we propose a metric to aid an end-user in choosing between these pairs.

We contend that end-users of scalable systems are primarily concerned with so-

lution quality. How QoS is measured is application-dependent; however, as QoS is a function of work, we can use the amount of *scaled work* to determine an AM pair's scalability. Additionally, we can rank different AM pairs based on the magnitude of the achievable scaling.

5.3.1 Taxonomy of Scalability Measures

To quantify the absolute scalability of an AM pair, or compare the relative scalabilities of several AM pairs, we derive three scalability metrics. One or more metrics may apply to an end-user's situation, and the relative importance among different metrics are problem dependent. The first two scalability metrics quantify, respectively, the maximum work computable and the maximum work achievable with time constraints. The third metric is for work fixed at a level desired by the end-user. We observe the range of *number of processors* and *execution times* that can compute the fixed level of work. Let W represent the base amount of work.

λ_∞ is the largest factor by which W can be scaled, *i.e.*, $\lambda_\infty W$ represents the maximum amount of scaled-up work achievable on the AM pair. This value is bounded by either the aggregate memory capacity of the machine or QoS_{max} , whichever is smaller.

λ_T is the largest factor by which W can be scaled on the AM pair when the execution time, T , is fixed by the end-user. Implicitly, $\lambda_T \leq \lambda_\infty$.

$S(\lambda_D)$ is the set of ordered pairs (N_j, T_j) that can compute $\lambda_D W$, where the scale factor, λ_D , is set by the end-user. Each pair (N_j, T_j) is the number of processors

and the corresponding amount of execution time needed to compute $\lambda_D W$ on the AM pair. Figure 5.4 is a generic illustration of the quantifier for an arbitrary amount of fixed work. Typically, as the processor configuration size increases, execution time decreases. However, the cost of additional communication may outweigh the benefit of splitting the work among more processors resulting in *increasing* execution time. Thus, in Fig. 5.4, each point on the curve to the left of the dashed line represents one (N_j, T_j) pair in the set $S(\lambda_D)$. The number of pairs in the set is determined, in part, by the granularity by which processor configuration size can be increased, *e.g.*, for hypercubes the increment is a power-of-2. Implicitly, $\lambda_D \leq \lambda_\infty$.

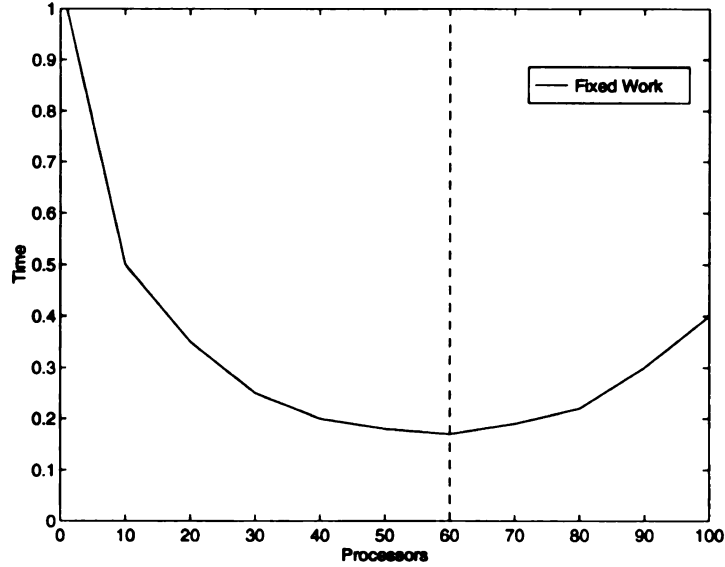


Figure 5.4: Execution time vs. number of processors tradeoff

For the first two metrics, the larger the value of the metric, the more scalable the AM pair. Determining the maximum work for a fixed time (λ_T) is similar to the SLALOM [61] benchmark which fixes computation time at one minute and determines how

accurately an answer could be generated in that time. For the third metric, $S(\lambda_D)$, scalability is measured by the number of (N_j, T_j) pairs and the magnitude of the respective N_j and T_j ranges. One could use $S(\lambda_D)$ to determine the most scalable AM pair in the $\sin(x)$ example given earlier where QoS_{max} is a function of λ_D . In contrast to the $\sin(x)$ example, many applications particularly in image and signal processing have physical limitations on data set sizes and, therefore, the work does not scale [63]. The $S(\lambda_D)$ metric could be utilized by end-users of such applications where $\lambda_D = 1$.

5.4 Mathematical Framework

To quantify AM pair scalability with the metrics introduced in Section 5.3, the various properties that characterize algorithms and machines must be parameterized. We introduce a simple mathematical framework, similar to the model used in Amdahl's Law [54, 53], for this purpose. This model is intended to facilitate AM pair scalability quantification and not as an absolute predictor of performance. A number of factors inhibit a precise prediction of algorithm performance on a parallel machine, *e.g.*, overlap of computation and communication, cache effects, network contentions, *etc.*

5.4.1 Algorithm and Machine Parameterization

Let $W = W_1 + W_\infty$ represent the amount of work to be performed for an implementation of an algorithm, where W_1 is the amount of work that can only be performed sequentially and W_∞ is the amount of work that can be fully parallelized, i.e., W_∞ can

take advantage of whatever number of processors are available. This model obviously provides the most optimistic case of an algorithm in terms of the degree of parallelism. For many applications, the amount of work can be characterized in terms of the data size of an application. For example, the amount of work for solving a system of n equations with n unknowns using Gaussian elimination is a function of n , i.e., $\frac{2}{3}n^3$.

To characterize the machine upon which an algorithm executes, we assume a distributed-memory architecture composed of N homogeneous *nodes*. Each node consists minimally of a processor with computation capacity Δ , local memory of size m , and a communication router. The aggregate computation capacity and memory of the machine are ΔN and mN , respectively. Henceforth, we shall use the terms *node* and *processor* interchangeably.

We define $M(W, N)$ as the amount of memory capacity required to execute W work on N processors. Depending upon the algorithm, $M(W, N)$ is a function of W or N or both. For instance, if the data size requirements of an algorithm increase with added computation, then the memory capacity requirement is a function of W . An algorithm where data is replicated among processors would have a memory capacity that is a function of N .

5.4.2 Fixed-work Problems

With the above parameters, we can formulate execution time in terms of computed work. Let $t_1 = \frac{W_1}{\Delta}$ and $t_\infty = \frac{W_\infty}{\Delta}$ be the respective execution times of W_1 and W_∞ on one processor. Equation (5.1) defines the execution time of W work on N processors;

it is the theoretical lower bound on execution time since it ignores the communication overhead and does not account for the memory constraints of the machine.

$$T_N(W) = \frac{W_1}{\Delta} + \frac{W_\infty}{\Delta N} = t_1 + \frac{t_\infty}{N} \quad (5.1)$$

When a parallel algorithm is executed on a distributed-memory machine and the number of nodes is greater than one, then typically there is communication among the nodes of the machine for non-local memory accesses. We define the *communication overhead* in terms of execution time as a function of work and number of processors, $h(W, N)$. Note that presently W is fixed, but N is variable as we may change the number of processors involved to compute the given work. Equation (5.2) expresses execution time that incorporates the amount of time attributable to interprocessor communication.

$$T_N(W) = t_1 + \frac{t_\infty}{N} + h(W, N) \quad (5.2)$$

Similar to the discussion on memory capacity in Section 5.4.1, communication overhead $h(W, N)$, may be a function of either the amount of work or the number of processors or both. In a five-point stencil application [64], nodes communicate with a fixed number of other nodes; thus, total communication overhead is not a function of N , but may be a function of W , *e.g.*, it may depend on the number of loop iterations. In contrast, assume a particular matrix multiplication algorithm where the columns of one of the matrix operands are distributed evenly to the nodes of the machine. In the process of computing the matrix product, columns are shifted around the nodes

of the machine until each column has been to each node. Such a communication paradigm is dependent on W (or matrix size in this case) and on N . Furthermore, the implementation of communication primitives used in an algorithm may also affect communication overhead. For example, a naive implementation of broadcast, where the source node uses separate messages sent sequentially to inform all destinations, *i.e.*, *separate addressing*, is an example of communication cost that is a linear function of N . In contrast, a tree-based implementation of broadcast would have logarithmic complexity, *i.e.*, $\log_2(N)$ [39].

In general, modeling communication overhead can be complicated since there are a number of issues to consider. Sending, receiving, and network latencies impact message transmission time. Second, different messages can contend for the same physical channels in a network causing increased network latency as message sizes increase [39]. Third, communication may overlap with algorithm computation. Message contention and communication/computation overlap can be very difficult to model precisely. We assert that modeling communication-overhead requires thorough analysis of the algorithm and the implementation of its message-passing primitives on the chosen machine. We elaborate on this process in Section 5.4.4.

In the computation of $T_N(W)$, there is an implicit assumption that $M(W, N)$ is not greater than the aggregate memory of the machine. If the memory requirement were greater than the aggregate memory capacity of the machine, then $T_N(W)$ would increase substantially due to the overhead of paging, if it is supported. Equation (5.3), therefore, defines the *memory bound* of an AM pair. Therefore, $T_N(W)$ is defined for

an AM pair if and only if Equation (5.3) holds.

$$M(W, N) \leq mN \quad (5.3)$$

5.4.3 Scaled-work Problems

Scaling up the initial work can contribute to a higher QoS as discussed in Section 5.3. A higher QoS may include an increase in computation, *e.g.*, increased number of iterations, and/or an increase in memory requirements. We define *scaled work* as $W^{(k)} = W_1 + kW_\infty$ if the fully parallelizable portion of the algorithm, W_∞ , can be scaled up by a factor of k and the sequential portion of the algorithm remains constant.³ Obviously, expecting the sequential portion of work to remain constant as work is scaled up is an optimistic assumption. The model could be extended to account for a corresponding increase in the sequential portion of the algorithm. We omit such an extension at this time. The execution time of scaled work is given in Equation (5.4).

$$T_N(W^{(k)}) = t_1 + \frac{kt_\infty}{N} + h(W^{(k)}, N) \quad (5.4)$$

Consistent with the formula for scaled work, Equation (5.4) multiplies the scaling factor, k , by the parallel portion of the algorithm. Communication overhead is extended similarly to account for scaled work. As discussed previously, the derivation of communication overhead, $h(W^{(k)}, N)$ when $k > 1$, is necessarily algorithm-specific.

Scaling the work may also increase the amount of memory required. Equation (5.5)

³We place k in parenthesis to clarify that we are *not* raising W to the power of k .

specifies the increased memory as a function of k and the original work where a is a real number.

$$M(W^{(k)}, N) = k^a [M(W, N)] \quad (5.5)$$

When $a = 0$, scaling the work does not increase the memory required to perform the work. For instance, in a particular molecular dynamics application [65], force calculations are performed for a fixed amount of time. Increasing the number of iterations requires additional work (computation) but not additional memory. In general, if $a = 1$, the memory increase is linear, if $a = 2$, the increase is quadratic, *etc.* In Section 5.6, we show an example where $a = \frac{2}{3}$. Equation (5.6) specifies the *scaled work memory bound*; it is a generalization of the inequality in Equation (5.3), *i.e.*, in that case $k = 1$.

$$M(W^{(k)}, N) \leq mN \quad (5.6)$$

AM pair a_2M in Fig.5.2 illustrates the potentially significant impact of the scaled work memory bound. We shall explore more examples in greater detail in Section 5.6.

5.4.4 Deriving AM Pair Parameters

As our goal is to *quantify* scalability of an AM pair, we must not only determine the appropriate formulas that characterize the AM pair but also determine the appropriate values for the chosen parameters. We assert that the determination of these values can be largely performed by algorithm analysis and empirical testing. We illustrate the determination of these parameters for several AM pairs in Section 5.6.

The work parameters, W_1 and W_∞ , can typically be determined by evaluating the

major loop bounds of an algorithm together with its data decomposition among the nodes of the machine. The sizes of these bounds may vary with program data size. Modeling communication can be difficult as discussed earlier. However, it is typically straightforward to determine the number of communication invocations, the size(s) of messages, and the number of processors involved. Parameterizing the complexity of the implementation of a communication primitive, *i.e.*, a linear or logarithmic function of N , may be obtained given access to vendor source code or by performing measurements.

5.5 A Three-dimensional Illustration of AM Pair Scalability

Typically, the work of a parallel application cannot be scaled without bound. As work is scaled, the memory size requirements of the algorithm may exceed the aggregate capacity of the machine as expressed in Equation (5.6). Additionally, communication overhead may grow with an increase in the number of processors as shown in Fig. 5.4. Given that we can characterize an AM pair with the mathematical framework of Section 5.4, we assert that it may be difficult to fully visualize the scalability of an AM pair or the relative scalabilities of a number of AM pairs. AM pairs may be characterized by a number of independent and complex formulas. A three-dimensional view of an AM pair, based on its mathematical characterization, may facilitate scalability analysis.

To illustrate our AM visualization, let us suppose a hypothetical AM pair whose scaled work execution time and memory capacity requirements are given with Equations (5.7) and (5.8), respectively. These formulas were arbitrarily chosen for the purpose of illustration. Let $n = 10$ represent the data size of the problem to be solved by the AM pair. Furthermore, let the parallelizable work be a function of the data size and let the sequential component be constant, *i.e.*, $W = W_1 + W_\infty = c_0 + n^2$. Scaled work is expressed as $W^{(k)} = c_0 + kn^2$. Communication overhead, which is a function of k and n , is given by the last operand of Equation (5.7). c_0 , c_1 and c_2 are small constants.

$$T_N(W^{(k)}) = c_0 + k \left(\frac{n^2}{\Delta N} \right) + (N)(c_1 + c_2(k)(n^2)) \quad (5.7)$$

$$M(W^{(k)}, N) = k(n^2)(\sqrt{N}) \quad (5.8)$$

Equations (5.7) and (5.8) are fairly simple, yet it is not straightforward to visualize their behavior or interrelationship as the number of processors or work are scaled. Additionally, suppose Δ were doubled by virtue of new processor hardware: how could this change affect scalability and how can it be visualized?

Figure 5.5 depicts the above hypothetical AM pair's scalability as a three-dimensional surface.⁴ The vertical axis plots execution time, $T_N(W^{(k)})$, as a function of the base axes: processor configuration size, N , and scaled work, $W^{(k)}$. Recall from Section 5.4 that $T_N(W^{(k)})$, is defined only when the memory requirement is

⁴Figure 5.5 and all subsequent color figures are found at the end of this chapter.

within the aggregate memory capacity of the machine configuration, *i.e.*, the relation in Equation 5.6 must hold. For the present example, $T_N(W^{(k)})$ is defined when $k(n^2)(\sqrt{N}) \leq mN$. The BLUE and WHITE regions in Fig. 5.5 denote, respectively, points which satisfy the relation and the points which do not. The bounds for an AM pair base axes are described next; in {}, we give these values for the current example.

- W - Base work, *i.e.*, $W^{(k)}$, where $k = 1$; {1}.
- $\lambda_\infty W$ - Upper bound on scaled work, *i.e.*, the maximum $W^{(k)}$, for which Equation 5.6 holds. If there is no sequential component, W_1 , then the maximum k is equal to λ_∞ ; {10}.
- N_{lb} - Lower bound on the number of processors on which W can be performed; {4}.
- N_{ub} - Total number of processors on the machine; {128}.

Due to the scaled work memory bound, only the blue region of Fig. 5.13 is of further interest. Figure 5.14 shows the same AM pair with the white region occluded and the vertical axis reduced in scale to “zoom-in” on the blue region. Figure 5.14 is partitioned by color with the following meanings.

- GREEN The execution time of the base work, W , over $[N_{lb}, N_{ub}]$ is denoted with this line. As the size of the processor configuration grows, execution time initially decreases, but may eventually increase, if the cost of greater communication outweighs the benefit of splitting the work among more processors.

Fixed-work problems (Section 5.4.2) can be modeled using this line exclusively.

Thus, they represent a subset of our framework.

- **BLUE** The values in this region denote execution times for scaled work, *i.e.*, $k > 1$. Lightest to darkest shadings represent execution time intervals: $(0,2]$, $(2,3]$, $(3,4]$, $(4,6)$. The shadings facilitate mapping the maximum work that can be accomplished in time T , *i.e.*, the λ_T measure. For example, suppose we want to find λ_T when $T = 3$. The border between the $(2,3]$ and $(3,4]$ shadings depicts this fixed time value for the entire range of processors. The **YELLOW** point at the “top” of the $(2,3]$ shading depicts the point at which maximum work is achieved when $T = 3$, occurring on 64 processors. Here, $\lambda_T = 4.8$.
- **MAGENTA** The upper bound on scaled work occurs with N_{ub} processors. The maximum work is $\lambda_\infty W$. This point is labeled “Inf.”
- **RED** This line plots execution time over a range of processors for fixed work, “D”, and thus illustrates the $S(\lambda_D)$ measure. In Fig. 5.14, the amount of work performed is $\lambda_D W$ where $\lambda_D = 3$. Points along the solid portion show the range of decreasing execution times on increasingly larger processor configurations, *i.e.*, the set (N_j, T_j) that achieve the desired level of work. The dashed segment illustrates increasing execution time due to the communication overhead of additional processors. The end-user would not be interested in the dashed segment since equivalent execution times are obtainable on smaller processor configurations. Alternatively, the entire red line segment may denote QoS_{max} , the upper bound on the end-user’s range of desired solution quality. Under this

end-user constraint, the upper bound on the scaled work axis would be replaced by 3.

Figures 5.13 and 5.14 are generic illustrations of an AM pair's performance. Different AM pair surfaces may vary greatly depending upon their $T_N(W^{(k)})$ and $M(W^{(k)}, N)$ functions and the range of available processors. It is precisely for this reason that we assert that presenting an AM pair in this manner is of particular importance. In the next section, we demonstrate the utility of our approach in comparing the relative scalabilities of three different AM pairs for performing matrix multiplication.

5.6 Case Study: Matrix Multiplication

We compare the relative scalabilities of three AM pairs for performing matrix multiplication. We choose this application for simplicity of illustration, *i.e.*, the straightforward algorithms allow us to focus our attention on presenting the utility of our scalability quantification framework. We plot the three-dimensional surfaces of the AM pairs and compare their respective $S(\lambda_D)$ metrics for fixed values of λ_D .

We select three parallel algorithms a_1 , a_2 , and a_3 , executing on the same machine, M , an IBM SP- x at Argonne National Laboratory [66]. The allowable processor configuration sizes are $[N_{lb}, N_{ub}] = [1, 128]$. Each AM pair computes C , the product of two $n \times n$ matrices A and B . The algorithms differ in basic operation and distribution of matrices across the nodes of the machine. The major distinctions of a_1 , a_2 , and a_3 are summarized next.

a_1 Matrices A and C are distributed by rows among nodes, and B is replicated on all nodes. The replication of B allows each processor to compute its portion of C without accessing off-processor data items. Thus, there is no interprocessor communication needed. The memory cost, however, grows with increasing processor configuration size due to the replication of B .

a_2 Matrices A and C are distributed by rows among nodes, and B is distributed by columns. The total memory capacity requirement is reduced in comparison to a_1 as it is not a function of the number of processors; however, point-to-point communication is used to shift columns of B among the processors. The total number of messages exchanged is a function of N , and the message size is based on the ratio of n to N .

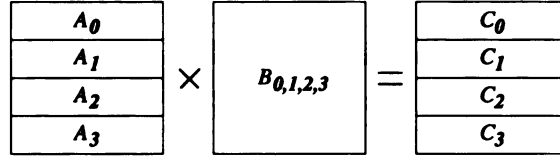
a_2 Matrices B and C are distributed by rows among nodes, and A is distributed by columns. Total memory cost is the same as for a_2 . Data exchange is performed with a collective sum reduction operation for every data element of C .

Next, we present the pseudo-code of each algorithm and describe their operation in more detail.

5.6.1 Algorithm-Machine Pair a_1M

Figure 5.5 depicts the distribution of the three matrices in a_1 for four processors numbered $0,1,2,3$. Subscripts denote the processor number to which the data is mapped.⁵ Figure 5.6 shows the operation of a_1 from the perspective of each processor.

⁵The distributions of A and C follow the (BLOCK,*) pattern of HPF [3].

Figure 5.5: Data distribution of a_1

Note that the bounds on the indices in statements S1-S3 are relative to the node's local data set. a_1 adheres to the *owner-computes rule* [3] that stipulates that the processor that stores the data element on the left-hand side of a program statement is responsible for computing its update. Thus, each node is responsible for computing its respective portion of C , an $\frac{n}{N} \times n$ data block. The outermost loop, S1, iterates over the node's local block C owned by the node. The degree of parallelism of a_1 is $N \leq N_{ub}$.

```

S1: For rows = 0 To  $\frac{n}{N} - 1$ 
S2:   For cols = 0 To  $n - 1$ 
S3:     For j = 0 To  $n - 1$ 
S4:       C[rows,cols] = C[rows,cols] + A[rows,j] × B[j,cols]
S5:     End For
S6:   End For
S7: End For

```

Figure 5.6: Algorithm a_1

5.6.2 Algorithm-Machine Pair a_2M

In algorithm a_2 , contiguous rows of A and C are partitioned among processors, just as in a_1 , but a_2 partitions contiguous columns of B among nodes. The operation of a_2 is described for four processors as follows. Computation begins with the diagonal

data blocks of C as depicted by the darkest shaded boxes of Fig. 5.7. The diagonal $\frac{n}{N} \times \frac{n}{N}$ -sized blocks of C are owned by different processors and thus can be computed in parallel. Next, each node proceeds with an off-diagonal block as illustrated with progressively lighter shading in Fig. 5.7. Each shading shows the block set that can be computed in parallel. Computing off-diagonal blocks, however, requires access to off-processor data. For example, to compute block C_{01} , processor 0 requires row A_0 (which is local) and column B_1 which is owned by processor 1. In general, to perform the latter steps, the columns of B are circulated one step to the left around the processors arranged in a ring as shown in Fig. 5.8 using the MPI [67] *send-and-receive* primitive which combines the two message-passing operations in one call.

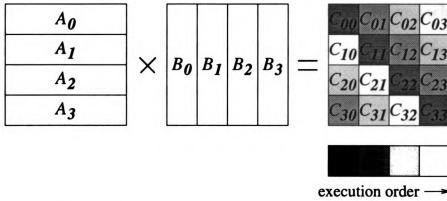


Figure 5.7: Data distribution and operation of a_2

Figure 5.9 presents the a_2 algorithm. The number of steps in the outermost loop (S1) is determined by the number of processors since each processor shifts its locally-owned column to all other processors. All processors perform a barrier synchronization at statement S4.

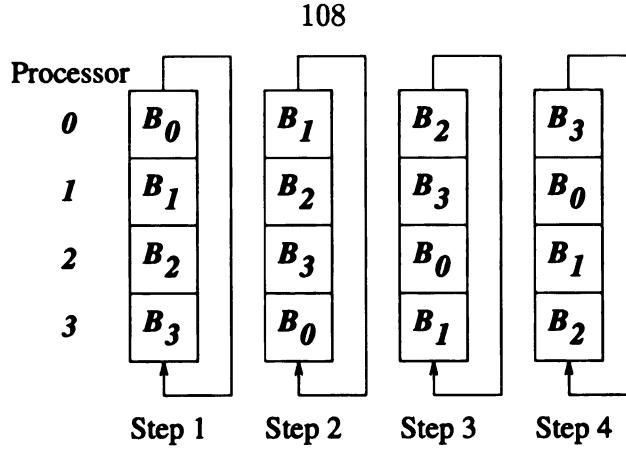


Figure 5.8: Communication pattern of a_2

5.6.3 Algorithm-Machine Pair a_3M

In algorithm a_3 , A is distributed by contiguous columns, and B and C are distributed by contiguous rows as shown in Fig. 5.10 for four processors. In contrast to the previous two algorithms, a_3 uses all processors in the computation of each data element of C . Figure 5.10 illustrates the computation of element C_{43} where each processor computes a partial sum of products of row 4 of A and column 3 of B . Processor 2, to which C_{43} is distributed, performs a *sum reduction* operation to obtain the final result. Figure 5.11 illustrates the general operation of a_3 . Statements S1 and S2 iterate over all row and column indices bounded by n . In statement S3, processors compute their respective partial sums. The operation of S6 is implemented as a tree-based reduction in MPICH [68].

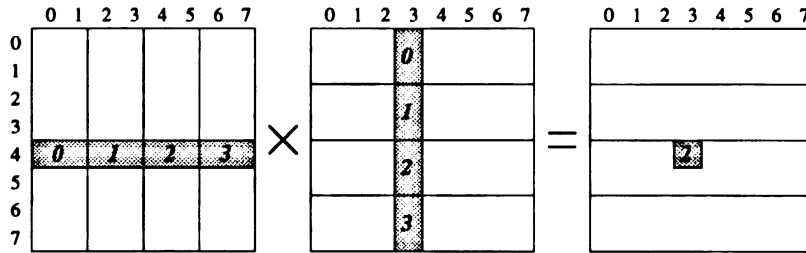
5.6.4 Computing AM Pair Parameters

Scalability quantification begins with the determination of the values of the parameters that characterize AM pair behavior, *e.g.*, W_∞ and Δ . The derivation of the parameters is based on the matrix sizes and distributions, and measured execution

```

S1: For step = 0 To  $N - 1$ 
S2:   ofst = [(my_rank + step) mod  $N$ ]  $\times \frac{n}{N}$ 
S3:   If (step > 0)
S4:     MPI_Sendrecv(B_col)
S5:   End If
S6:   For rows = 0 To  $\frac{n}{N} - 1$ 
S7:     For cols = 0 To  $\frac{n}{N} - 1$ 
S8:       For j = 0 To  $n - 1$ 
S9:          $C[\text{rows}, \text{cols} + \text{ofst}] = C[\text{rows}, \text{cols} + \text{ofst}] + A[\text{rows}, j] \times B_{\text{col}}[j, \text{cols}]$ 
S10:      End For
S11:    End For
S12:  End For
S13: End For

```

Figure 5.9: Algorithm a_2 Figure 5.10: Data distribution and operation of a_3

times on machine M .

Clearly, the amount of work in matrix multiplication is a function of data size, or n . We fix the base size to be $n = 1000$. To determine the amount of work (computation), we examine the number of floating-point operations (FLOPs) in the three algorithms. As presented, the algorithms perform n multiplications and n additions to obtain each element of C . Since there are n^2 elements of C , a rough estimate of the base work is $W = 2n^3 = 2$ GFLOPs, for $n = 1000$. As W can be fully parallelized (limited by

```

S1: For rows = 0 To  $n - 1$ 
S2:   For cols = 0 To  $n - 1$ 
S3:     For  $j = 0$  To  $\frac{n}{N} - 1$ 
S4:       partial = partial + A[rows,j]  $\times$  B[j,cols]
S5:     End For
S6:     MPI_Reduce(partial,sum,root)
S7:     If (root)
S8:       C[(rows mod  $\frac{n}{N}$ ),cols] = sum
S9:     End If
S10:   End For
S11: End For

```

Figure 5.11: Algorithm a_3

N_{ub}), $W = W_\infty$. The sequential component W_1 is never exactly zero for any algorithm due to such factors as process initialization. However, in a_1 and a_2 , the sequential component is of minimal consequence and can be omitted in the present analysis. The computation is scaled by increasing the matrix size by a real factor i , i.e., we solve $(in) \times (in)$ matrices, where $i \geq 1$. The amount of increased work required to solve a factor i larger problem is given by $k = 2i^3$. Henceforth, we use i rather than k as the former is more intuitive.

Memory capacity requirements for each AM pair is based on the sizes and distributions of the matrices. For a_1 , each node stores $(in) \times \frac{(in)}{N}$ blocks of elements of A and C and all of B . Thus, the memory requirement per node is $(in)^2(1 + \frac{2}{N})$. The aggregate memory requirement, $M(W^{(i)}, N)$, is $(in)^2(N + 2)$. Due to the replication of B , each additional processor added to the computation causes the aggregate memory requirement to grow by n^2 . For a_2 and a_3 , $M(W^{(i)}, N) = 3(in)^2$. Each

single-precision real data element displaces four bytes on machine M .

The derivation of several AM pair parameters is gathered by empirical testing. We execute a_1M , a_2M , and a_3M to obtain values of $T_N(W^{(i)})$ for various i and N . Using Equation (5.9) and execution times achieved with a_1M , we solve for Δ . Equation (5.9) has no h component since there is no communication in algorithm a_1 . To obtain a stable value for Δ , we perform a number of tests with varying i and N and then take the mean.

$$T_N(W^{(i)}) = \frac{2(in)^3}{\Delta N} \quad (5.9)$$

Using this approach, we obtain a per-node sustained computing capacity estimation on the SP- x of $\Delta = 56$ MFLOPS. Algorithm a_2 contains a communication component which comprises a sending and receiving latency, c_0 , and network latency as a function of the message size, $c_1(\frac{(in)^2}{N})$. The number of communication steps in a_2 is $N - 1$. The total execution time of a_2 is given in Equation (5.10).

$$T_N(W^{(i)}) = \frac{2(in)^3}{\Delta N} + \left[c_0 + c_1 \left(\frac{(in)^2}{N} \right) \right] (N - 1) \quad (5.10)$$

Using a system of equations based on measured execution time of a_2 , we determine the values of $c_0 = 0.075$ and $c_1 = 7.94^{-5}$. In a_3 , the communication component is based on its use of the MPI_Reduce operation implemented with a tree-based paradigm [69]. Figure 5.12 illustrates the message-passing sequence for eight nodes. Based on the (binary) rank of the node, starting from the right least significant bit, if the bit is 1, send to the node with that bit 0 and exit; if the bit is 0, receive from the node

with that bit set and combine with the addition operator. `MPI_Reduce` combines summation and communication, however, we shall not separate this primitive into distinct work and communication components. We model the execution time of the operation with $c_2 \lceil \log(N) \rceil$, where c_2 is some constant. The message size of each constituent message sent in the collective reduction is one data element, regardless of matrix size. Reduction is invoked for each data element in C ; thus, the total cost of reduction is $(in)^2(c_2 \lceil \log(N) \rceil)$. Equation (5.11) captures the total execution time for

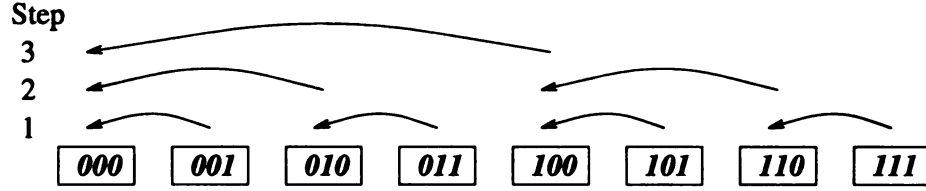


Figure 5.12: Tree-based MPI Reduction used in a_3M

a_3M . We determine the value of c_2 to be $1.33 \cdot 10^{-5}$.

$$T_N(W^{(i)}) = \frac{2(in)^3}{\Delta N} + (in)^2(c_2 \lceil \log(N) \rceil) \quad (5.11)$$

Table 5.1 summarizes the parameter values for the three AM pairs. Next, we quantify the relative scalabilities of a_1M , a_2M , and a_3M based on these parameter values.

AM pair	$W^{(i)}$	$M(W^{(i)}, N)$	Δ	h
a_1M	$2(in)^3$	$(in)^2(N + 2)$	56MFLOPS	n/a
a_2M	$2(in)^3$	$3(in)^2$	56MFLOPS	$[c_0 + c_1(\frac{(in)^2}{N})](N - 1)$
a_3M	$2(in)^3$	$3(in)^2$	56MFLOPS	$(in)^2(c_2 \lceil \log(N) \rceil)$

Table 5.1: AM parameters for a_1M , a_2M , and a_3M

5.6.5 Scalability Quantification of a_1M , a_2M , and a_3M

Let us assume two end-users for matrix multiplication, User₁ and User₂, who are interested in determining the most scalable AM pair based on the $S(\lambda_D)$ measure for $\lambda_D = 5$ and $\lambda_D = 20$, respectively. Due to its memory constraint (see Table 5.1), a_1M can only scale i from 1 to 5.6. In contrast, a_2M and a_3M can obtain matrix scaling on the range, $1 \leq i \leq 36.6$. The measure λ_D represents a *fixed* value of i . Thus, for $\lambda_D = 20$, we can only compare the latter two AM pairs.

Figure 5.15 illustrates the respective behaviors of a_2M (left) and a_3M (right) for $1 \leq i \leq 36.6$ and $1 \leq N \leq 128$. Base work is shown in green as explained in Section 5.5. The red lines in Fig. 5.15 illustrate $\lambda_D = 20$ for a_2M and a_3M , respectively. The sets (N_j, T_j) for a_2M and a_3M are given in Table 5.2. Note that the processor ranges, N_j , are given from smallest to largest and the corresponding execution times range from largest to smallest. As explained in Section 5.5, execution time values which do not satisfy Equation (5.6) are obscured.

λ_D	a_1M		a_2M		a_3M	
	N_j	T_j	N_j	T_j	N_j	T_j
5	[9,128]	[464,32]	[5,128]	[2363,1935]	n/a	n/a
20	n/a	n/a	[45,128]	[41576,37329]	[45,128]	[189094,184334]

Table 5.2: $S(\lambda_D)$ measure for a_1M , a_2M , and a_3M

When $\lambda_D = 20$, a_2M and a_3M have equivalent processor ranges, but a_2M achieves much faster execution times. The larger execution times for a_3M are due to the number of communication reduction operations required. Each reduction has a small message size as only one data element is contributed by each node. Each participant,

however, must incur a costly sending latency for each invocation. Due to the large number of reductions, sending latencies result in a large aggregate overhead as data sizes grow. The “flatness” of the surfaces with respect to increasing N in Fig. 5.15, pictorially illustrates that increasing the number of processors has a minimal impact on execution time. This phenomenon is explained by the communication overhead components of Equations (5.10) and (5.11). In each equation, $(in)^2$ contributes far more to the overall cost than the respective N and $\log(N)$ factors.

The surfaces in Fig. 5.16 illustrate the respective behaviors of a_1M (left) and a_2M (right) over $1 \leq i \leq 5.6$ and $1 \leq N \leq 128$. Again, base work is shown in green.⁶ Figure 5.16 (left) illustrates the full range of matrix size scale-up for a_1M . Figure 5.16 (right) illustrates a_2M ’s matrix scale range up to only 5.6 to facilitate comparison with a_1M . We omit a_3M from the $\lambda_D = 5$ comparison since Fig. 5.15 clearly establishes that a_2M achieves better relative performance over the entire range of i from 1 to 36.6. The red lines in Fig. 5.16 illustrate performance for the fixed level of work corresponding to $\lambda_D = 5$ for a_1M and a_2M , respectively. Again, the precise values for (N_j, T_j) are summarized in Table 5.2.

We conclude that a_1M is the most scalable AM pair for User₁ who restricts λ_D to 5. This conclusion is based on a_1M ’s relatively lower execution times; however, it can only perform the required work on no fewer than 9 processors, compared to 5 processors for a_2M . Presumably, this limitation would be of minimal importance to User₁. For $\lambda_D = 20$, both a_2M and a_3M require the same number of (N_j, T_j) pairs,

⁶Note that the higher resolution for a_2M reveals that base work execution time *increases* as the processor configuration size grows.

but a_2M achieves significantly better execution times over the entire range. Thus, a_2M is the most scalable AM pair for User₂.

5.7 Inclusion of Other Scalability Models

We favor a scalability framework that is primarily concerned with an AM pair's ability to increase solution quality through increased work over the approach taken by other models which measure scalability mainly in terms of processor speedup, efficiency, or speed. However, our model does not preclude using these latter metrics as additional parameters. For instance in Fig. 5.17, the shading within the blue region is based on the ability to maintain *average speed*, the quotient of work over the product of number of processors and execution time [57]. Speed values are normalized, *i.e.*, the base work on a single processor has average speed equal to 1.0. The borders between shadings from darkest to lightest are given by average speed values of [1.0,0.9,0.8,0.7,0.6,0.2]. Similarly, *efficiency* [58], measured as the quotient of speedup over number of processors, may be incorporated in an AM surface. Since speedup is required to measure efficiency, we are limited to scaling the work up to the memory bound for a single processor. Figure 5.18 illustrates efficiencies using the same hypothetical AM pair and the same values for shadings as given above for average speed. Note that the upper bound on the axis for scaled work on the efficiency plot is 10. Thus, the efficiency surface represents a subset of the average speed surface (Fig. 5.17).

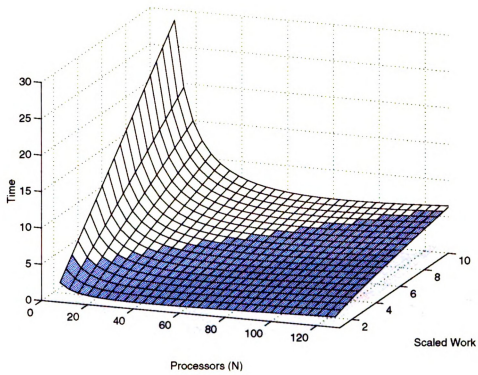


Figure 5.13: A generic AM pair 3D surface illustration

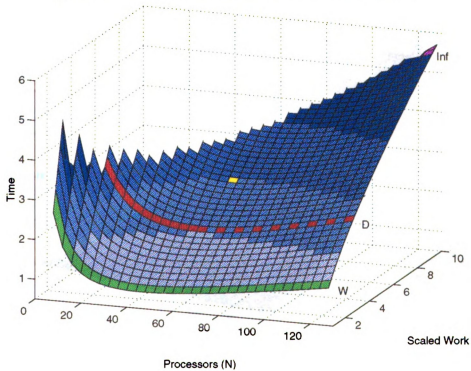


Figure 5.14: Scalability metrics illustrated on AM surface

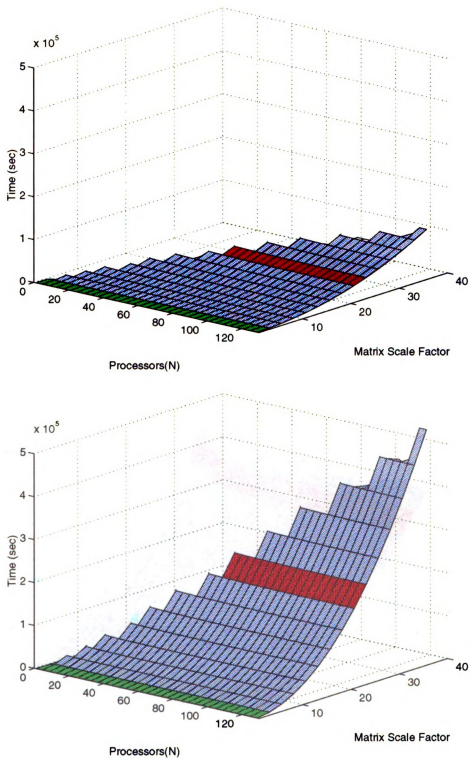


Figure 5.15: $S(\lambda_D)$ comparison of a_2M and a_3M where $\lambda_D = 20$

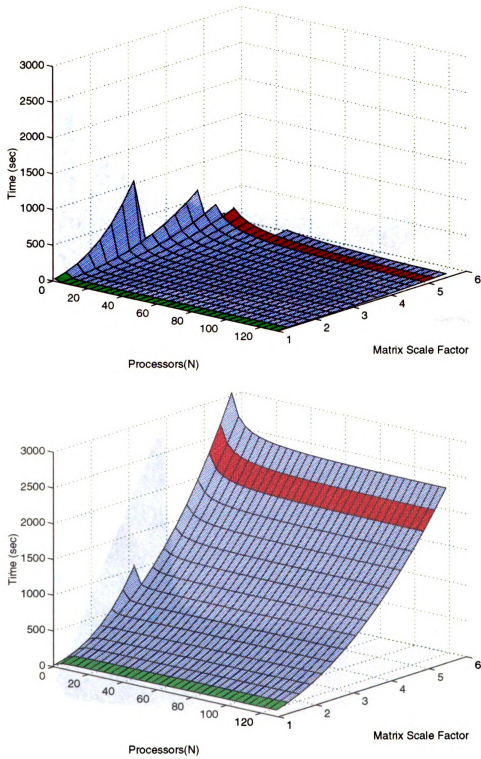


Figure 5.16: $S(\lambda_D)$ comparison of a_1M and a_2M where $\lambda_D = 5$

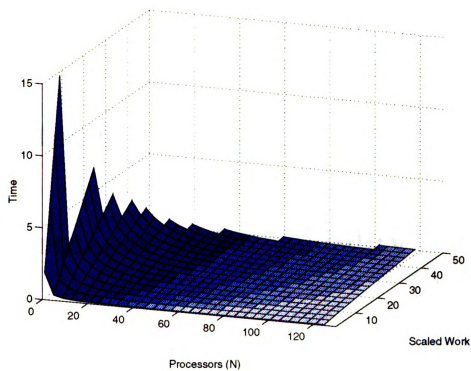


Figure 5.17: Incorporating speed metric

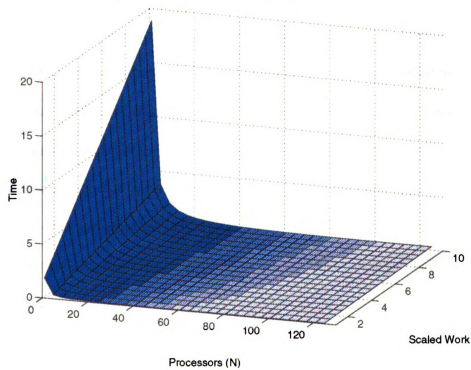


Figure 5.18: Incorporating efficiency metric

Chapter 6

Performance and Scalability

The goal of data redistribution is to enhance data-parallel program performance by providing a capability to rearrange data among the processor memories of a machine that better suits subsequent computation. Data redistribution does not contribute to the data-parallel program's computed result; thus, the ability to scale work or scale the quality of the solution are not relevant factors for analyzing redistribution. We assert that the viability of the optimal processor mapping technique (Chapter 3) and redistribution library, DaReL, (Chapter 4) ought to be measured in terms of execution time performance as that is the most relevant criterion for measuring redistribution performance. For any particular program data size, selected distribution patterns, and processor configuration, the time required to perform data redistribution should be minimized.

Beyond providing adequate performance for a particular redistribution scenario, we emphasize the importance of providing long-term, *i.e.*, *scalable*, redistribution performance as data and processor configuration sizes increase. Chapter 5 presents a

model for quantifying algorithm-machine (AM) pair scalability using end-user-defined criteria. Central to the quantification framework is measuring an AM pair’s ability to scale the amount of work performed. Since data redistribution scalability assessment focuses on the efficiency of data exchange, and not computation, we shall address the question: by what criteria is data redistribution scalability to be quantified? We assert that the trend in the *slope* of the redistribution execution time curve over a range of processors is a viable means for quantifying redistribution scalability. The concept of slope over a range of processors is similar to our earlier work where we coined *range of scalability* [8] as a means for assessing scalable library design.

In this chapter, we present data redistribution performance using DaReL on an IBM SP-*x* at Argonne National Laboratory [66] and demonstrate that DaReL is highly communication-dominant. The performance of redistribution using the optimal processor mapping technique is compared with redistribution performance using the traditional mapping technique. We review a number of MPI point-to-point and collective communication algorithm options to perform data exchange, and we determine the best, in terms of performance, implementations as a function of data size and number of processors. Lastly, using the scalability quantification framework as a basis, we propose extensions for quantifying the scalability of DaReL, and we present results based on these extensions.

6.1 Processor Mapping Technique Analysis

In this section, we discuss the possible impacts on the data-parallel programmer and the compiler, respectively, when using the optimal processor mapping technique. We present the execution time benefit of the technique compared to the traditional mapping technique for a number of data redistribution cases when performed on an IBM SP- x .

6.1.1 Effect of Optimal Mapping on the Programmer

Permuting *lpids* with the optimal mapping functions in Sections 3.1 and 3.2 may incur undesired side effects and thus may not be suitable in all redistribution instances. For instance, the programmer may lose “neighboring data” relationships. In Fig. 3.2, under the traditional cyclic mapping, *lpids* 0 and 1 are neighbors¹ while under the optimized mapping, *lpids* 0 and 4 are neighbors. Suppose a programmer imparts a particular logical to physical processor mapping for a given program and utilizes this information for maintaining neighboring data relationships for *physical* processors. In order to preserve these relationships, a programmer may favor the traditional processor-data mappings over the optimal mapping in a call to data redistribution. Figure 6.1 illustrates *static*, *i.e.*, not varying from one program execution to another, logical to physical processor mappings: logical processor i always maps to physical node p_i . In (a), the traditional mapping, *i.e.*, 0,1,2,3, preserves the neighboring processor relationships while in (b) neighboring data relationships on the physical

¹A processor is a neighbor if it owns data elements that are adjacent from the perspective of the global data.

nodes is corrupted when using the optimal mapping, *i.e.*, 0,2,1,3 makes p0 and p2 become neighbors where previously p0 and p1 were neighbors.

<u>Logical</u>	<u>Physical</u>	<u>Logical</u>	<u>Physical</u>
0	-----> p0	0	-----> p0
1	-----> p1	2	-----> p2
2	-----> p2	1	-----> p1
3	-----> p3	3	-----> p3
(a) Traditional		(b) Optimal	

Figure 6.1: Mappings when physical node mapping is *static*

When a compiler or run-time system uniquely determines the logical to physical processor mappings, however, the programmer is unable to maintain neighboring data relationships on the physical nodes. Indeed, indirect-network multicomputers, *e.g.*, IBM SP-*x*, or most workstation clusters have no concept of “neighboring nodes.” Furthermore, the allocation of parallel jobs, and thus data, may vary for distinct executions of the program since different physical processors may be allocated to each job. Figure 6.2 illustrates a number of program executions where the allocation of logical processors to physical nodes is different for each invocation. The top part of Fig. 6.2 illustrates the traditional processor-data mapping for three distinct program executions; the bottom portion of Fig. 6.2 illustrates the optimal processor-data mapping for three distinct program executions. From the perspective of the physical nodes of the machine, the permutation of the *lpids*, whether traditional or optimal, is transparent to the programmer. We argue that in these situations, the use of the optimal mapping is always justified since neighboring data relationships on the physical nodes cannot be maintained, or exploited, by the programmer.

Permuting *lpids* can also facilitate greater flexibility: the programmer may want to

Traditional

<u>Logical</u>	<u>Physical</u>	<u>Logical</u>	<u>Physical</u>	<u>Logical</u>	<u>Physical</u>
0	-----> p0	0	-----> p2	0	-----> p1
1	-----> p1	1	-----> p0	1	-----> p2
2	-----> p2	2	-----> p1	2	-----> p3
3	-----> p3	3	-----> p3	3	-----> p0
(a) First execution		(b) Second execution		(c) Third execution	

Optimal

<u>Logical</u>	<u>Physical</u>	<u>Logical</u>	<u>Physical</u>	<u>Logical</u>	<u>Physical</u>
0	-----> p0	0	-----> p2	0	-----> p1
2	-----> p1	2	-----> p0	2	-----> p2
1	-----> p2	1	-----> p1	1	-----> p3
3	-----> p3	3	-----> p3	3	-----> p0
(a) First execution		(b) Second execution		(c) Third execution	

Figure 6.2: Mappings when physical node mapping is *dynamic*

influence which data elements are redistributed to other processors and which remain on-processor. Under the traditional mapping technique, the programmer has but one choice, *i.e.*, *lpids* are mapped in increasing numerical order. With the optimized mapping technique, there are often several options since a number of *lpids* may map to the same place holder; see Fig. 3.3(b). If the compiler supports programmer-specification of *lpid* permutations, then the user has inherently greater control over the data to processor mapping. Such flexibility may become even more significant when data alignments are introduced. In such a situation, a number of data elements may map to some local indices while fewer data elements map to other indices. Therefore, being able to influence which indices are redistributed and which indices remain on-processor could enhance overall performance.

6.1.2 Effect of Optimal Mapping on the Compiler

The loss of neighboring data relationships may complicate the role of the compiler in generating SPMD node programs from the HPF source code. Let us examine a simple example. Let A be a 16-element array that is initially distributed with **BLOCK**, and then redistributed **CYCLIC**; see Fig. 3.2. Assume that the following reference pattern appears in the compiler-generated SPMD code following the call redistributing A to **CYCLIC**.

$$A(i) = A(i - 1) + A(i + 1)$$

Using the traditional mapping technique, the compiler generates the following communication paradigm to obtain off-processor elements: Excluding the boundary processors, each $lpid_i$ communicates with $lpid_{i-1}$ and $lpid_{i+1}$. Under the optimal mapping technique, neighboring processor relationships are not maintained, thus the above communication paradigm cannot be used. For example, $lpid_3$ communicates with $lpid_6$ and $lpid_7$, while $lpid_4$ communicates with $lpid_0$ and $lpid_1$. This problem can be easily overcome, however, if the compiler utilizes the place holder mapping information determined by the optimal mapping function. Let $q = 0, 4, 1, 5, 2, 6, 3, 7$ be the array of place holders of the $lpids$ as generated by Equation (3.1). Under the optimal mapping, the compiler would specify neighboring communication for non-boundary processors as follows: for $lpid_i$ in place holder q_j , communicate with $lpids$ in q_{j-1} and q_{j+1} . Essentially, the compiler performs a table lookup to determine neighboring $lpid$ relationships. The extension to boundary processors is straightforward and is excluded from the present discussion.

6.1.3 Performance Comparison with Traditional Technique

The optimal mapping technique has been integrated into DaReL. Either the traditional or optimal mapping technique can be selected as a run-time option without need for recompilation. To assess the run-time performance of the optimal mapping technique, we compare data redistribution execution times using the optimal technique with redistribution execution times using the traditional mapping. An experimental research version of MPI, MPI-F [70, 71], was used to obtain the performance results on the Argonne IBM SP- x .

Figures 6.3 through 6.5 illustrate various performance comparisons of the two mapping techniques. Figure 6.3 shows (**BLOCK**,*) to (**CYCLIC**(c),*) redistributions on 8 nodes over a range of matrix sizes: 32-thousand to 34-million 4-byte floating point elements. The block size of the **CYCLIC**(c) pattern (row dimension) was maintained at one-half the block size, b , of the **BLOCK** distribution. The optimal mapping technique, applied to only the row dimension of the matrix, demonstrates significantly lower execution times over the traditional mapping. The optimal technique achieves the same redistribution result in roughly 60% of the time needed for redistribution using the traditional technique. The larger the value of c with respect to b the greater the effect of the optimized mapping technique on overall execution time because significantly more data hits occur relative to the traditional mapping. With smaller values of c relative to b , the benefit of the optimal technique is lessened since the relative difference in the number of data hits is reduced. For all redistribution instances, regardless of block sizes (b and c) or global data size, we find the optimal technique outperforms

or equals the traditional mapping.

Figures 6.4 and 6.5 demonstrate $(\text{BLOCK}, \text{BLOCK})$ to $(\text{CYCLIC}(c), \text{CYCLIC}(c))$ redistributions on 12 (logically 4×3) and 24 (logically 6×4) processor configurations, respectively. Matrix sizes ranged up to 91-million floating point numbers. In these performance plots, the mapping technique is applied to both dimensions of the matrices. As before, the optimal mapping redistributions outperform the traditional mapping in all cases. The block size c was maintained at one-sixth and one-eighth of the respective block sizes in the row and column dimensions of the matrices. The execution time advantage of the optimal mapping technique remains consistent for larger (> 24) processor configurations as well.

The computation of the send and receive processor sets, CDES, is included in all of the execution time plots shown. Execution time attributable to these calculations represented a small fraction of the overall total, *i.e.*, on the order of hundreds of micro-seconds for small data sizes and on the order of tens of milliseconds for the largest data sizes plotted. See Section 6.2.1 for more details.

6.2 DaReL Performance and Scalability

In this section, we examine in detail DaReL's performance for several types of data redistribution scenarios covering a large range of data shapes and sizes and processor configurations. We assert that scalability in the case of DaReL ought to be defined by its ability, for a fixed data size, to improve performance with additional processors. To this end, we propose a simple metric for quantifying DaReL scalability. The

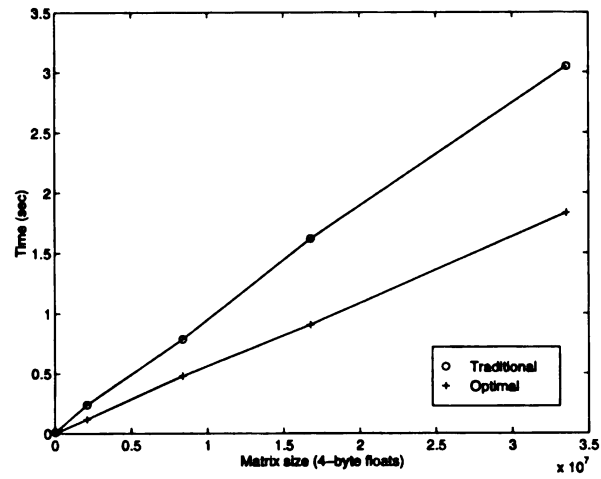


Figure 6.3: (BLOCK,*) to (CYCLIC(c),*) on 8×1 processors

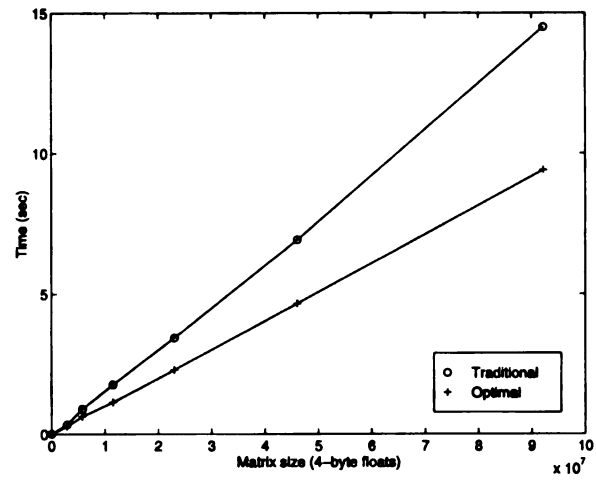


Figure 6.4: (BLOCK,BLOCK) to (CYCLIC(c),CYCLIC(c)) on 4×3 processors

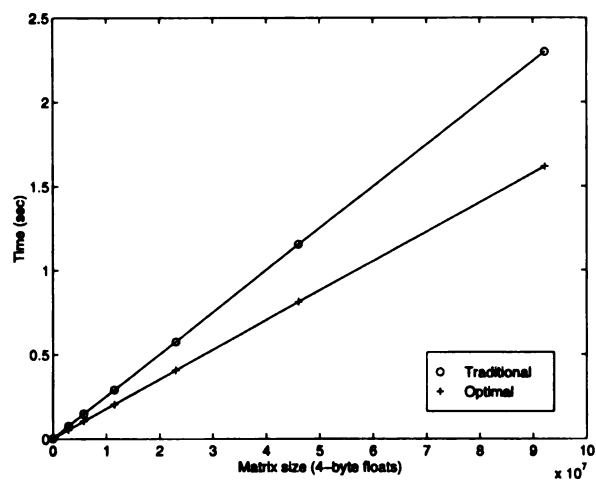


Figure 6.5: (BLOCK,BLOCK) to (CYCLIC(c),CYCLIC(c)) on 6×4 processors

metric is based on the slope of the execution time curve with respect to the number of processors used.

6.2.1 CDES and ED Execution-time Performance

The performance of DaReL's main components, CDES and ED, for a number of distinct redistribution cases are presented. The results were obtained using the public domain MPI, MPICH [68], on the Argonne IBM SP- x . Eight different redistribution cases were performed, see Table 6.1. CASES 1-4 represent different kinds of redistributions in terms of data shape and selected patterns executed on 20-processor configurations of the SP- x ; CASES 5-8 are the same as CASES 1-4 except that they performed on 100-processor configurations. The individual cases within the groups 1-4 and 5-8 are dissimilar to demonstrate DaReL's flexibility in dealing with arbitrary patterns and data shapes and sizes. We explore the execution time effects of these factors. All data points were obtained using the traditional processor mapping technique.

Table 6.1: Redistribution test cases for DaReL

CASE	D_s	D_t	$P_s = P_t$	Data configuration
1	(BLOCK,BLOCK)	(CYCLIC,CYCLIC)	5×4	$1000 \times 1000 - 4000 \times 4000$
2	(BLOCK,CYCLIC)	(CYCLIC,BLOCK)	5×4	$1000 \times 1000 - 4000 \times 4000$
3	(BLOCK,*)	(CYCLIC,*)	20×1	$400 \times (2500 - 40000)$
4	(*,CYCLIC)	(*,BLOCK)	1×20	$(1250 - 20000) \times 800$
5	(BLOCK,BLOCK)	(CYCLIC,CYCLIC)	10×10	$1000 \times 1000 - 4000 \times 4000$
6	(BLOCK,CYCLIC)	(CYCLIC,BLOCK)	10×10	$1000 \times 1000 - 4000 \times 4000$
7	(BLOCK,*)	(CYCLIC,*)	100×1	$400 \times (2500 - 40000)$
8	(*,CYCLIC)	(*,BLOCK)	1×100	$(1250 - 20000) \times 800$

Figures 6.6 and 6.7 show the execution time of the CDES and ED modules, respectively, on 20-processor configurations, CASES 1-4. The range of data sizes is shown on the horizontal axis corresponding to the last column of Table 6.1. The vertical axis shows the execution time of the respective module. For all four cases, the execution time of CDES is approximately two orders of magnitude smaller than the execution time of ED. The execution time data is obtained with the point-to-point algorithm shown in Fig. 4.7. Later in the chapter, we discuss the significance of the selected ED algorithm on the overall performance of redistribution.

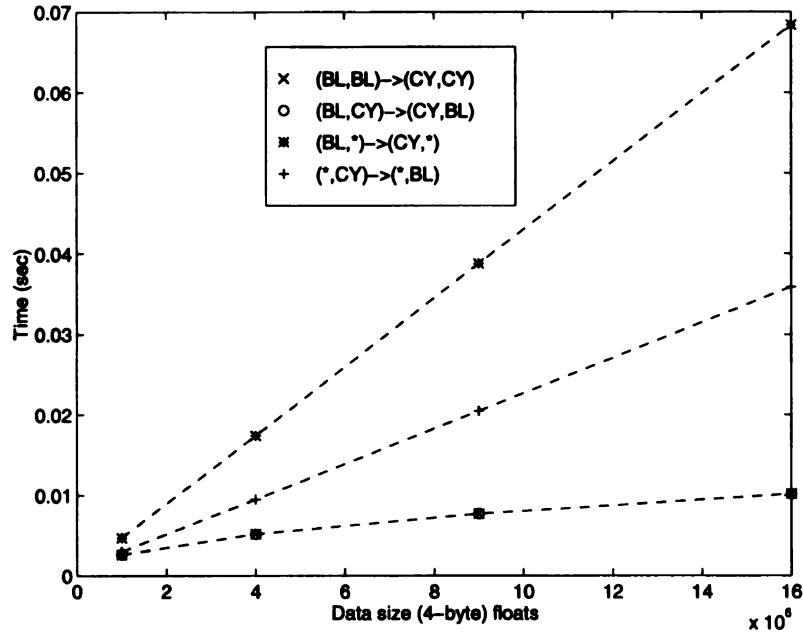


Figure 6.6: CDES performance on 20 processor configurations

CDES execution time depends upon the size and the shape of the data to be redistributed. Recall, from Chapter 4, that the complexity of CDES is $O(n_0 + n_1 + \dots + n_{m-1})$ where m is the number of dimensions and n_i is the number of elements in dimension i . For CASES 1-4, CDES complexity is $O(n_0 + n_1)$. Thus, CASES 3-4

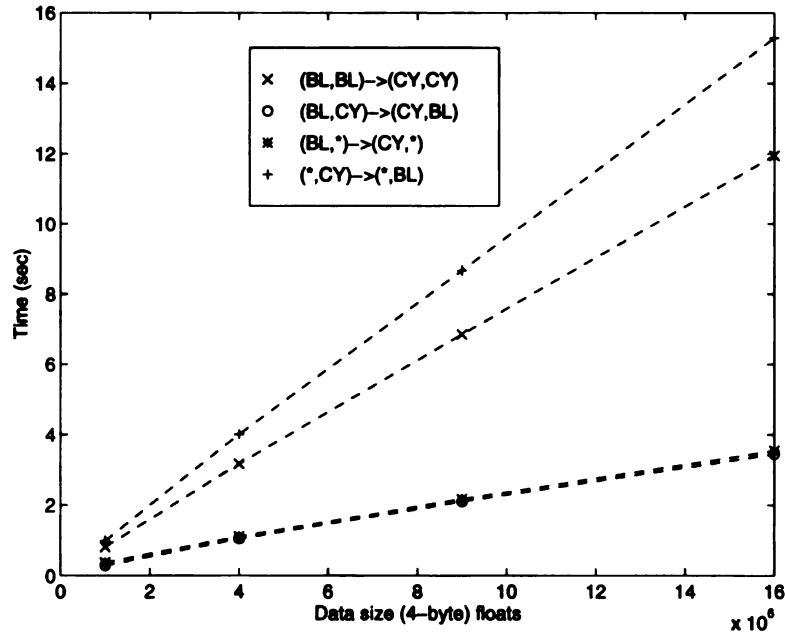


Figure 6.7: ED performance on 20 processor configurations

whose data matrices are not square in shape, exhibit higher complexities than CASES 1-2 whose shapes are perfect squares.² The measured execution time of the four cases is consistent with the CDES complexity estimations.

Since ED constitutes the majority of redistribution execution time, we concentrate on assessing its performance among CASES 1-4 in greater detail. Redistribution CASES 1 and 4 exhibit much higher execution times, up to a factor of 5 on 20 processors, than the remaining two redistributions cases. CASES 1 and 4 do not distinguish themselves by exchanging more data, indeed, the amount of data exchanged in all four cases is exactly the same. The discrepancy in execution time is explained by whether the data to be sent to other processors is taken from contiguous or non-contiguous locations in the local processors' memory.

²See last column of Table 6.1.

We compare CASES 1 and 2, which appear quite similar, but whose measured execution times on 20 (logically 5×4) processors vary significantly. We discuss their execution for the largest data size redistributions, *i.e.*, sixteen million elements. In both cases, each processor stores a 800×1000 block of 4-byte floating-point numbers stored in row-major order. Figures 6.8 and 6.9 illustrate the local data sets for processor $(0,0)$ for CASE 1 and CASE 2, respectively. The 0,1,2,3 numbering applied to the columns and the 0,1,2,3,4 pattern applied to the rows of Fig. 6.8 illustrate the ownership of $(0,0)$'s elements under the D_t pattern, *i.e.*, (CYCLIC,CYCLIC). The shaded blocks indicate the local elements that are sent to processor $(0,1)$; there are forty-thousand such elements. Each of these elements must be obtained from non-contiguous memory locations. MPI's derived datatype facility, discussed in Section 4.2.5, provides for the collection of non-contiguous data to be done by the underlying MPI implementation rather than by the user. Collecting data from non-contiguous data locations, however, impacts the overall execution time of a message-passing operation [72]. For CASE 1, all processors send forty-thousand element messages to the nineteen remaining processors where the elements are taken from non-contiguous locations.

Contrast the situation of CASE 1 to that of CASE 2 shown in Fig. 6.9. The same 0,1,2,3 and 0,1,2,3,4 patterns are applied to the columns and rows, respectively, as in CASE 1. However, due to the semantics of D_t in the column dimension, contiguous data blocks for this dimension are destined for individual processors. The semantics of D_t in the row dimension precludes the entire data block (shown shaded) from being obtained from fully contiguous locations. Nevertheless, the data destined to $(0,1)$ is obtained from 160 non-contiguous locations of 250-element blocks. We assert that the

lower overhead of collecting data for CASE 2 accounts for its better execution time relative to CASE 1. Similar distinctions can be shown when comparing redistribution CASES 3 and 4. For CASE 3, processors send data from largely contiguous locations while for CASE 4, all data elements are taken from non-contiguous locations.

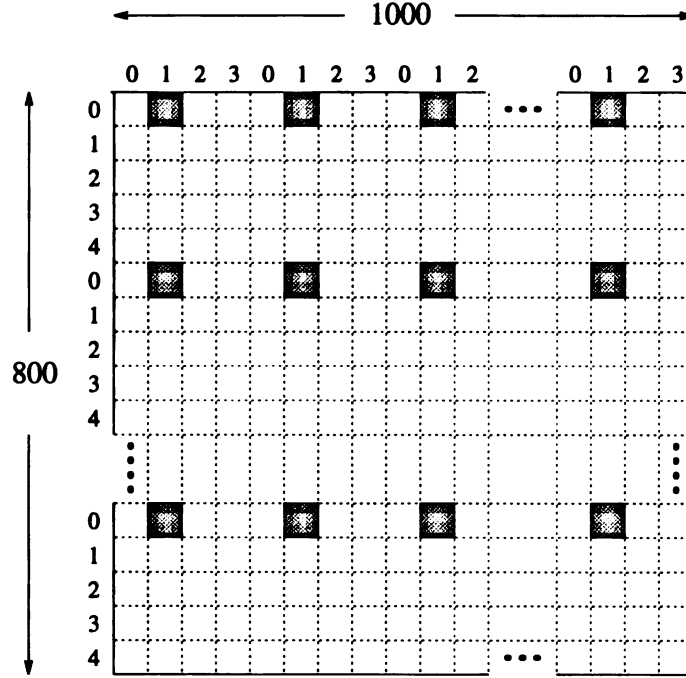


Figure 6.8: Local data set for (BLOCK,BLOCK) to (CYCLIC,CYCLIC)

We turn our attention to the 100-processor redistributions, CASES 5-8 in Table 6.1. The performance of CDES, shown in Fig. 6.10 for each case corresponds almost exactly to the performance exhibited by the equivalent type of redistribution in the 20-processor case, *e.g.*, CDES performance of CASES 4 and 8 are almost exactly equal. Thus, we conclude that the number of processors is of minimal consequence to CDES performance, and data size is the primary performance factor. This conclusion is consistent with our earlier CDES complexity analysis. ED performance for CASES 7 and 8 are significantly lower relative to their counterparts, CASES 3 and 4,

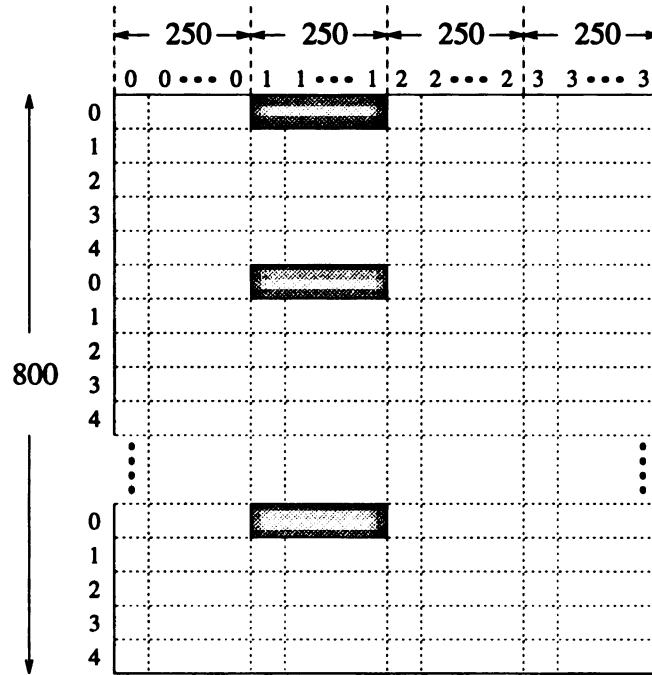


Figure 6.9: Local data set for (BLOCK,CYCLIC) to (CYCLIC,BLOCK)

respectively. This phenomenon is attributable to the small number of destinations that a processor sends to in the former cases as compared to the latter cases. For instance, each processor in CASE 7 sends to only three other destinations. In CASES 7 and 8, the data shapes are highly non-square. If the data shapes were more square in shape, each processor would need to send to a greater number of processors, most likely resulting in increased execution time. CASES 5 and 6 are similar to the execution times of CASES 1 and 2, respectively.

6.2.2 Algorithm Choices for ED

Due to the predominance of ED execution time, in this section, we examine the different MPI-based implementations of the ED module discussed in Chapter 4; see Figs. 4.7 and 4.8. We define four algorithm-machine (AM) pairs (see Chapter 5)

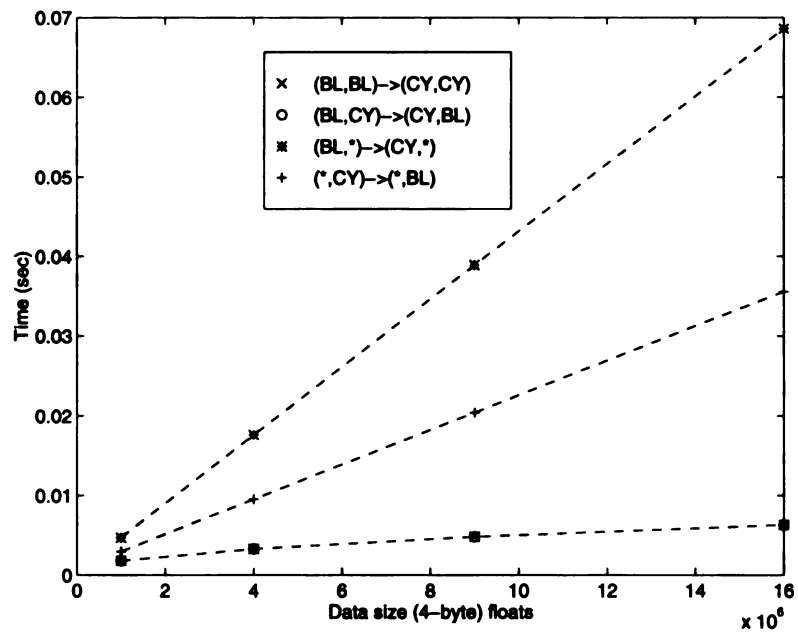


Figure 6.10: CDES performance on 100 processor configurations

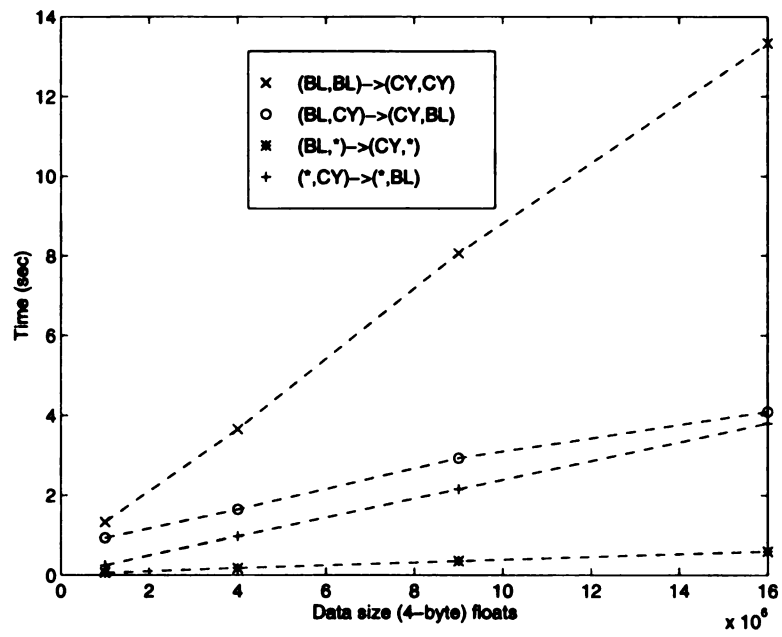


Figure 6.11: ED performance on 100 processor configurations

for performing ED. The algorithms are distinguished by their use of MPI primitives.

Below, N represents the number of participating processors:

1. **Point-to-point** Uses $N - 1$ calls to `MPI_Send` and $N - 1$ calls to `MPI_Recv`.
2. **Gather** Uses N calls to `MPI_Gather`.
3. **Scatter** Uses N calls to `MPI_Scatter`.
4. **All-to-all** Uses one call to `MPI_Alltoall`.

Figure 6.12 shows the execution time performance of the four AM pairs over a range of processor configuration sizes, $[4, 100]$, for (BLOCK,BLOCK) to (CYCLIC,CYCLIC) redistributions. This type of redistribution was selected since it exhibited the worst case redistribution performance on the 100-processor configuration and the second worst performance on the 20-processor configuration. We assert that the scalability analysis applied to this case can be extrapolated to the other redistribution cases as well. The four plots in Fig. 6.12 range from a data size of 1000×1000 (4-byte) floating point numbers (upper left) to 4000×4000 floating point numbers (lower right).

The four plots in Fig. 6.12 show that execution time increases with increasing data size. Consistent throughout the four plots is the relative performance of the four AM pairs. In all plots, the Scatter algorithm represents the worst performance followed by the Point-to-point algorithm. Significantly better performance is seen with the Gather and All-to-all algorithms. To analyze the performance data, we examine the implementation details of the above collective communication primitives

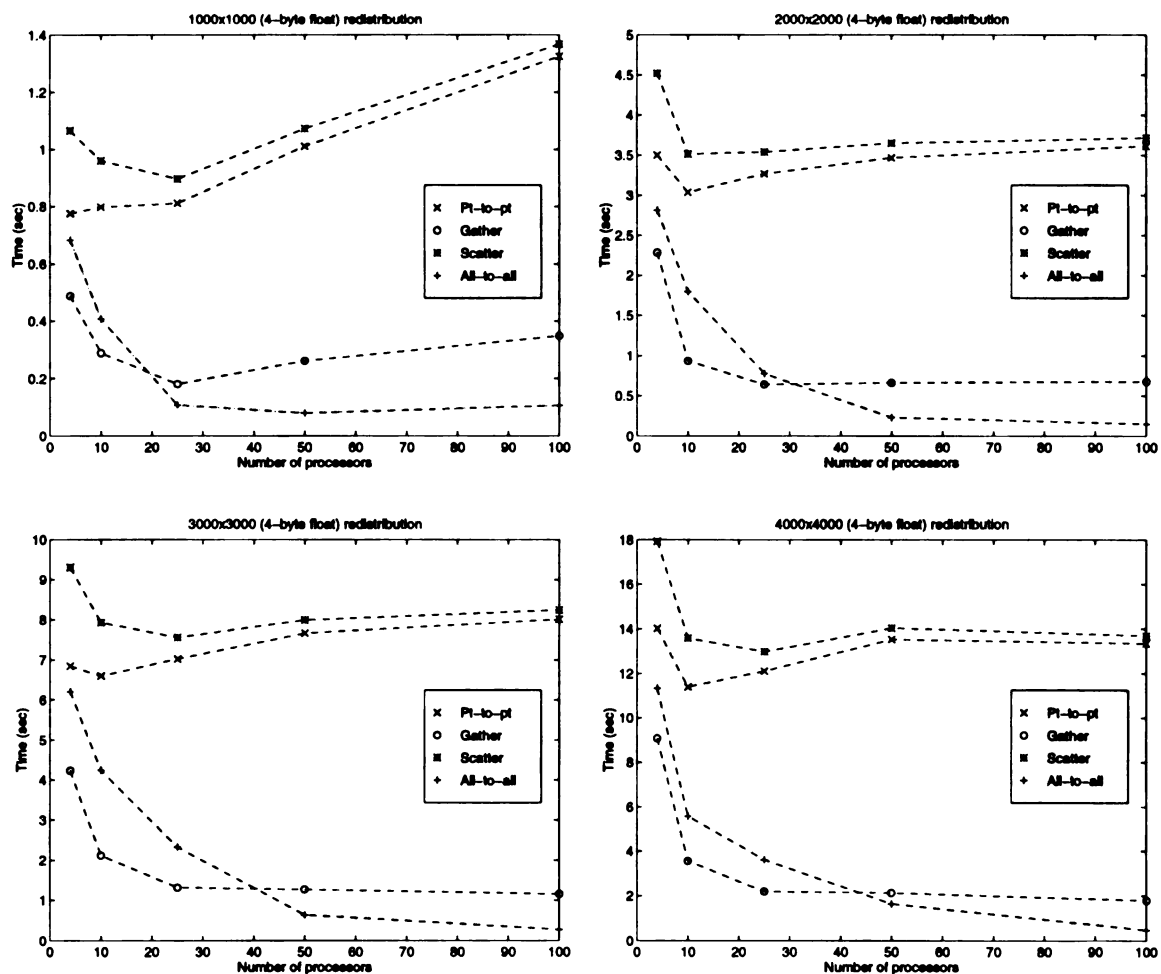


Figure 6.12: (BLOCK,BLOCK) to (CYCLIC,CYCLIC) redistributions

in MPICH.³ Each of the collective operations is implemented by MPI blocking and non-blocking send and receive primitives. **MPI_Alltoall** is implemented exclusively with non-blocking sends and receives. This approach is particularly advantageous when the number of processors is large as messages can be received from processors in the order of arrival. Using blocking receives imparts a static order on the reception of messages in user space from distinct senders. The imposed order may not mirror the actual order of message arrival leading to inefficiencies.

Perhaps the most surprising performance result is the clear advantage of the Gather over the Scatter algorithm. Intuitively, both ought to exhibit comparable performance since the former is a sequence of N many-to-one operations while the latter is a sequence of N one-to-many operations. An **MPI_Gatherv** is implemented as a sequence of blocking **MPI_Recv** calls for the root process and a single blocking **MPI_Send** call for all non-root processes. To gather data from its send buffer into its receive buffer, the root process performs a non-blocking send to itself that is eventually received with a blocking receive. An **MPI_Scatterv** is implemented as a sequence of blocking **MPI_Send** calls at the root process and a single blocking **MPI_Recv** call for all non-root processes. The root process performs a blocking **MPI_Sendrecv** to scatter data from its send buffer into its receive buffer.

In the context of MPI, a *blocking* send is defined as a process being blocked until its communication buffer can be reused. Thus, returning from a call to **MPI_Send** does not imply that the intended recipient has received the message. On the other hand, returning from a blocking **MPI_Recv** means that the message has been successfully

³MPICH is public domain software, thus, we can analyze the source code.

received into the user's communication buffer. To explain the performance difference of the Gather and Scatter algorithms, we examine the interaction of communicating processes using the two algorithms combined with message-passing benchmark data for the SP- x . The amount of time consumed by a point-to-point message is commonly subdivided into sending, network, and receiving latencies. The sending latency, t_{send} , is the time required to inject a message into the network; the network latency, t_{net} , is the time consumed traversing the network, and the receiving latency, t_{recv} , is the time needed to retrieve the message at the recipient. Nupairoj and Ni [73] find the following values for these parameters on the SP- x where m represents the message size in bytes:

$$t_{send} = 20 + (0.02)m \text{ } \mu\text{-seconds}$$

$$t_{net} = (0.03)m \text{ } \mu\text{-seconds}$$

$$t_{recv} = 35 + (0.02)m \text{ } \mu\text{-seconds}$$

We examine a specific case when the redistribution data size is 1000×1000 on four processors. For these data points, the relative proportions of $t_{send} : t_{net} : t_{recv}$ are approximately 1.0 : 1.5 : 1.0. Using these relative proportions, Fig. 6.13 illustrates the communication paradigm of the Scatter (top) and Gather (bottom) algorithms for performing the ED function with the processors numbered 0,1,2,3. For both algorithms, processor 0 assumes the role of root for the first **MPI_Scatter** (**MPI_Gather**) operation. Subsequently, the other processors in increasing numeric order become

the roots of sequential scatters (gathers). Data is communicated using the blocking **MPI_Send** and **MPI_Recv** primitives. The line segments labelled **S** and **R** represent sending and receiving latencies, respectively. Diagonal lines indicate network latency. Note that in both portions of the figure the first **MPI_Scatter** and first **MPI_Gather** complete in the same number of time steps, 5.5. The Gather algorithm, however, has already initiated all four **MPI_Gather** operations and completes them in 16 time steps.⁴ In contrast, Scatter completes the same number of **MPI_Scatter** operations in 19 time steps. The Gather algorithm has the advantage that more sending, network, and receive latencies are completely overlapped, *i.e.*, they occur in parallel. The current analysis does not entirely account for the difference between Gather and Scatter execution times; for instance, such factors as message packetization and message contention are not considered. The analysis, however, provides some insight into the non-intuitive advantage of Gather over Scatter for performing ED.

The advantage of the Point-to-point algorithm over the Scatter algorithm is explained by the implementation of the Scatter algorithm. As described earlier, the MPICH implementation of Scatter is a succession of calls to the blocking point-to-point routines. In other words, the Scatter algorithm looks very much like the Point-to-point algorithm in Fig. 4.7. The distinction is that Scatter uses a **MPI_Send-recv** operation to scatter data from its sending buffer to the receiving buffer. In the Point-to-point algorithm used in ED, **MPI_Pack** and **MPI_Unpack** are used for the reshuffling of data that stays local to a process. This results in better execution time performance

⁴The diagram assumes that all four processes initiate the respective collective communication calls simultaneously. In SPMD, this may not always be the case. We assume simultaneous behavior for the purpose of this analysis.

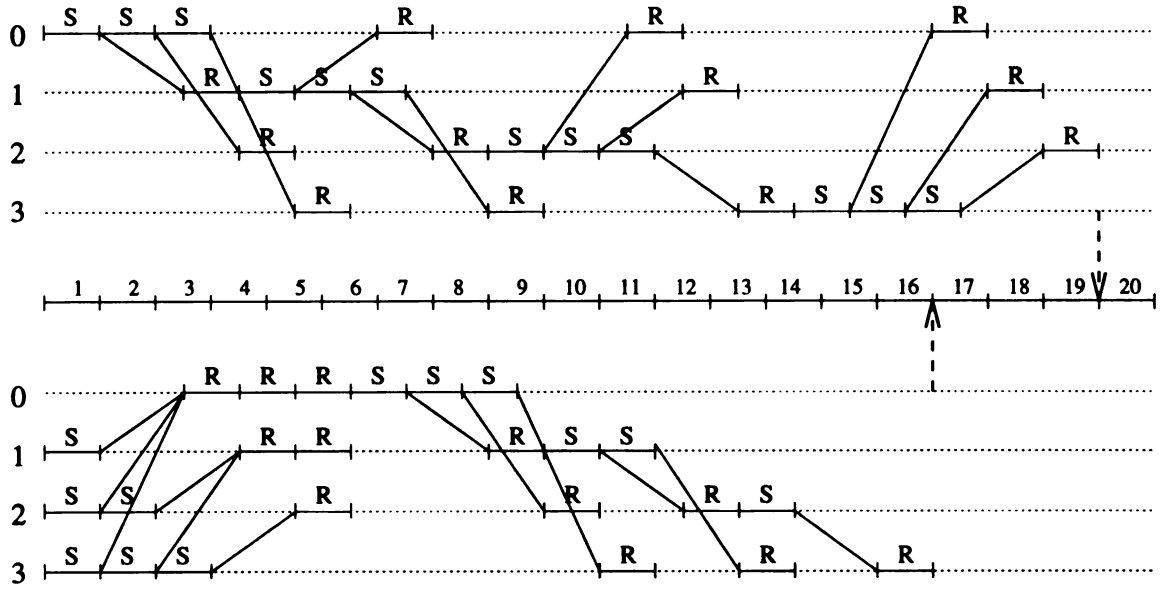


Figure 6.13: Communication paradigm using Scatter (top) and Gather (bottom)

as shown in the four plots of Fig. 6.12.

6.2.3 Scalability

In Sections 6.2.1 and 6.2.2, we discussed DaReL's performance for various redistribution cases using different algorithmic approaches to perform data exchange. In this section, we address DaReL's scalability. Specifically, we quantify DaReL's scalability as the number of processors participating in redistribution is increased. Scalability, in the context of data redistribution, measures the long-term trend in execution time as the number of processors used increases. To quantify scalability, we shall observe the *slope* of the execution time over a range of processors $[N, N']$, where $N' > N$.

In Chapter 5, we declared that an AM pair whose execution time increased with additional processors performing the same amount of work, was inherently *unscalable*. We use this idea to define scalability in terms of slope: as we increase from a pro-

cessor configuration of size N to one of size N' , we expect increased performance. In the context of redistribution, the increased performance is measured by reduced execution time. To measure increased performance, it suffices to observe the slope of the execution time curve over the range $[N, N']$. We specify slope formally in Equation (6.1).

$$\text{Slope}(N, N') = \frac{T_{N'} - T_N}{N' - N} \quad (6.1)$$

We define *negative slope*, $\text{NegSlope}(N, N')$ as the negation of $\text{Slope}(N, N')$ in Equation (6.2).

$$\text{NegSlope}(N, N') = -\text{Slope}(N, N') \quad (6.2)$$

When $\text{NegSlope}(N, N') < 0$, execution time is increasing with additional processors; when $\text{NegSlope}(N, N') > 0$, execution time is decreasing with additional processors. We define the scalability/unscalability of a data redistribution AM pair based on whether $\text{NegSlope}(N, N')$ is positive or negative. If $\text{NegSlope}(N, N')$ is positive, its magnitude is used to quantify scalability.

Definition 3 *An AM pair is unscalable on $[N, N']$ when $\text{NegSlope}(N, N') < 0$.*

Definition 4 *An AM pair is scalable on $[N, N']$ when $\text{NegSlope}(N, N') > 0$. The larger the magnitude of a negative slope, the more scalable the AM pair is.*

We examine the use of the quantifier $\text{NegSlope}(N, N')$ by calculating the slopes of the four AM pairs in the 1000×1000 plot of Fig. 6.12. Figure 6.14 repeats the plot with the computed values for $\text{NegSlope}(N, N')$ shown directly below it. The line segments illustrate $\text{NegSlope}(N, N')$ values on a specific range of processors for a

given AM pair. Note that the All-to-all AM pair consistently achieves the highest $NegSlope(N, N')$ values. Certainly, the trend in slope can be seen in either the top or bottom portion of Fig. 6.14. The additional information contributed by the bottom plot is the relative *magnitude* of the scalability quantifier $NegSlope(N, N')$ for the distinct AM pairs. For instance, Scatter's $NegSlope(4, 10)$ is less than All-to-all's $NegSlope(10, 25)$. Additionally, one can clearly see for which processor ranges the $NegSlope(N, N')$ values drop below zero and by what magnitude. Note that none of the AM pairs succeeds in having a positive $NegSlope(50, 100)$. This result is attributable to the ratio of data size to the number of processors participating in the redistribution. The data set size is a constant 1000×1000 floating-point numbers. Beyond 50 processors, the size of the local data set, and thus the message size, is too small to amortize the cost of sending and receiving latencies.

As shown in Section 6.2.2, DaReL has a number of algorithms at its disposal for performing the exchange of data. Given the gathered performance data of Fig. 6.12, we are able to fine-tune ED to select the highest performer among the AM pairs for a particular data size and processor configuration. The selection can be done in ED by means of a switch/case-type statement based on run-time information provided to the library; see Section 4.2.6. For example, in the case of the 1000×1000 redistribution (Fig. 6.14) the Gather AM pair is used on the range $[4, 20]$ and the All-to-all AM pair is used on the range $[21, 100]$. The Point-to-point and Scatter AM pairs are not considered since their execution times are not competitive with Gather and Alltoall. Thus, DaReL's performance for any given data point of number of processors and data size, is determined by the best performing AM pair. The performance plots are

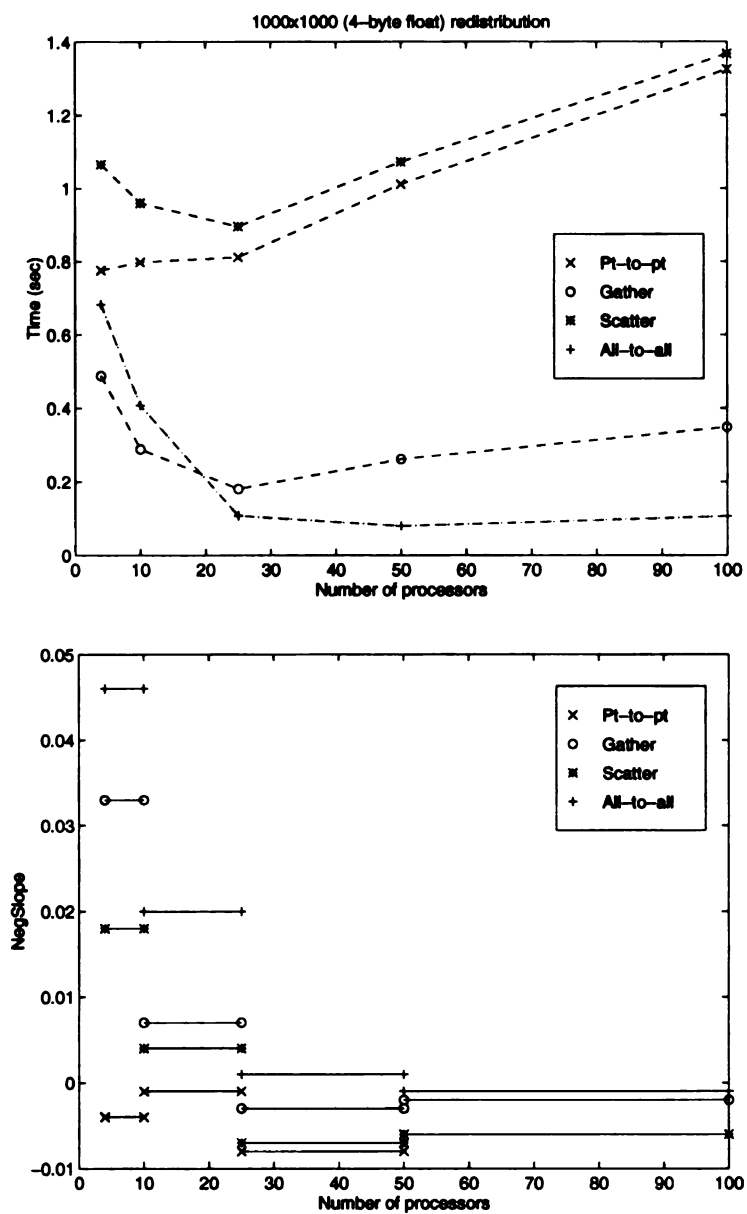


Figure 6.14: 1000 × 1000 redistribution with

coarse, providing information for only specific processor and data sizes. To perform a fine-tuning of DaReL many more performance results for fine-grain size increments would be needed. For the obtained data, we select the best-performing AM pair and quantify DaReL's scalability in terms of the $NegSlope(N, N')$ measure for the selected AM pair. We examine the change in this measure as the number of processors and data size increase.

We compute $NegSlope(N, N')$ for DaReL over the full range of processors under consideration, $[4, 100]$. This is done for data sizes in the range 1000×1000 up to 4000×4000 . The result of this analysis is illustrated in Fig. 6.15. Each of the horizontal line segments represents $NegSlope(N, N')$ values for a specific redistribution data size and range of processors. The vertical line segments attach the horizontal segments for clearer presentation. As data size increases, the values for $NegSlope(N, N')$ increase as well. As processor configuration size increases, $NegSlope(N, N')$ decreases. This result is acceptable, so long as $NegSlope(N, N')$ does not become negative. With the exception of $NegSlope(50, 100)$ for the 1000×1000 case (see dashed line segment in Fig. 6.15), DaReL achieves $NegSlope(N, N')$ for all other execution time data.

The plotted values of Fig. 6.15 are coarse since the processor increments are relatively large: chosen to establish long-term trends. With smaller processor value increments, the plotted lines would be smoother. The “bump” in each line is due to the switch from the Gather AM pair to All-to-all AM pair that yields a higher slope value.

In the computation of $NegSlope(N, N')$, the units for T_N values are in seconds. Note that this affects the magnitude of the slope computation. Thus, the *relative*

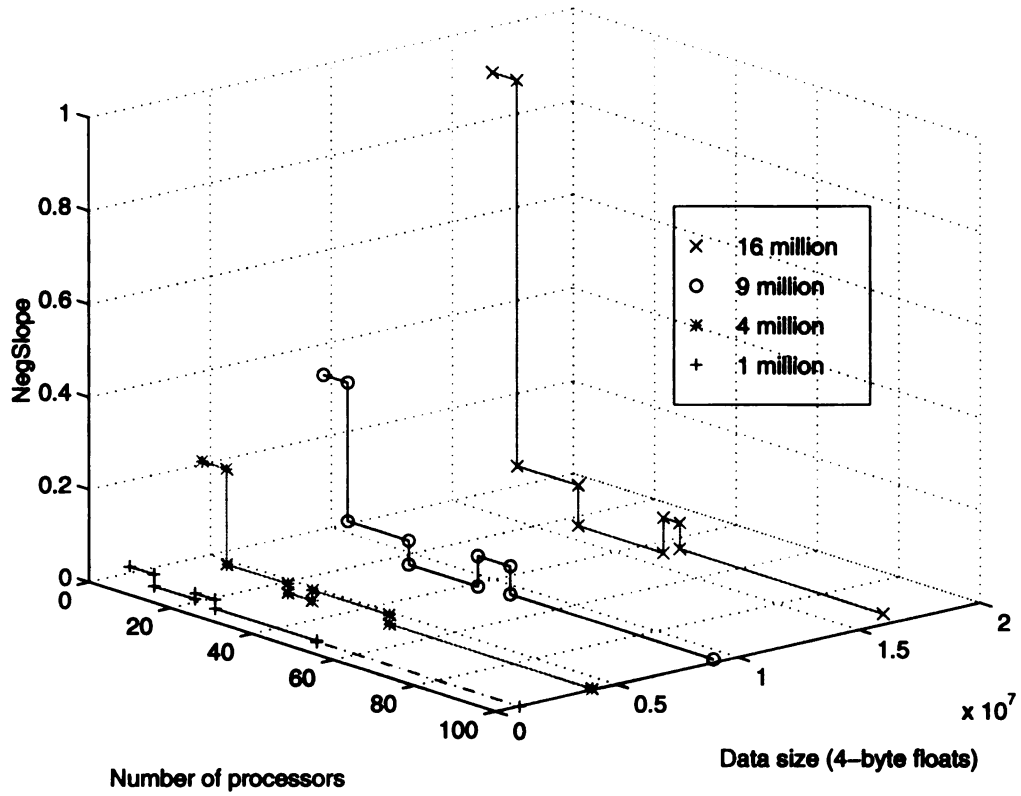


Figure 6.15: DaReL scalability measured by $NegSlope(N, N')$

values for $NegSlope(N, N')$ rather than *absolute* values are important for scalability quantification. If time units were measured in nanoseconds, then the vertical axis in Fig. 6.15 would change, but the relative distinctions between the plotted lines would not.

Chapter 7

Conclusion and Future Work

Changing array distributions within a data-parallel program can substantially affect overall program performance. Explicit redistribution is intended to optimize subsequent computation while implicit redistribution can be a common occurrence in a program. Implicit redistribution can occur with array assignment statements where the distribution of the operands are not equivalent, through the use of data realignment, or through the use of subroutine libraries whose operands require specific distributions of data. This dissertation has examined the critical issues for performing efficient and scalable data redistribution in distributed-memory machines, and it has proposed methods to mitigate the cost of redistribution in terms of execution time.

7.1 Research Contribution

Since interprocessor communication is the predominant source of redistribution execution time, minimizing the total amount of data movement among processor memories

may increase redistribution performance. This dissertation presented a technique, based on logical processor to data element mapping, that *minimizes* the total amount of data movement among processor memories for **BLOCK** to **CYCLIC**(*c*), and vice-versa, redistributions. Mapping functions for one-to-one, *m*-to-*m*, one-to-two, and two-to-one dimension data redistributions were presented and their optimality was proven. The proposed methodology is architecture-independent, facilitating its potential integration into distinct redistribution implementations for different distributed-memory architectures. We discussed the possible impacts on the programmer and compiler of using the mapping technique. We showed that the technique offers the programmer extra flexibility in determining data placement in some instances. Additionally, the use of the technique in a straightforward manner by the compiler is discussed. We demonstrated redistribution execution time improvements over the traditional data-processor mapping of up to 40% using the optimal mapping technique on an IBM SP-*x*. Portions of this work have appeared in [74] and shall appear in [75].

We presented a simple end-user-oriented framework for quantifying the scalability of algorithm-machine (AM) combinations, or pairs, together with a facility to aid in visualizing their scalability in three dimensions. We developed a mathematical framework to characterize the properties of AM pairs, *e.g.*, communication overhead, and we proposed a model for visually illustrating an AM pair's execution-time performance over user-defined ranges of processor configurations and scaled work. The benefit of a three-dimensional illustration is that a number of critical parameters, *e.g.*, memory capacity and communication overhead, can be viewed together in a unified manner, together with the end-user's chosen scalability metric(s). We presented a de-

tailed case study that illustrates the utility of our framework in comparing the relative scalabilities of distinct AM pairs for performing matrix multiplication. We demonstrated the flexibility of the framework in its ability to incorporate other scalability metrics, such as isospeed [57] and isoefficiency [58]. The scalability quantification framework can be found in [76] which has been submitted for publication.

We proposed the design of a portable, MPI-based library, DaReL, for explicit data redistributions, and we motivated the critical design choices. DaReL supports multi-dimensional data redistribution for HPF’s regular distribution patterns, **BLOCK**, **CYCLIC**, and *****. It is designed for `!HPF$ REDISTRIBUTE`; however, we envision its applicability to *implicit* redistributions that can occur within HPF programs. We discuss a number of salient issues affecting the design of a redistribution library for HPF including scalability and the respective roles of the compiler and the library. In contrast to other approaches, *e.g.*, [32], DaReL decouples processor send/receive set calculation from data exchange. This decoupling simplifies library design and facilitates multiple data exchange algorithm options. Data exchange is performed with MPI primitives, enhancing DaReL’s portability among distributed-memory platforms that utilize the emerging message-passing standard. We discussed the advantages of using MPI for data redistribution, and we detailed DaReL’s use of MPI point-to-point, collective communication, process topology, and derived datatype constructs.

We demonstrated the execution time performance of DaReL’s implementation on an SP-*x* and focused our attention on optimizing the data exchange portion of DaReL due to its execution-time predominance. We examined a number of MPI point-to-point and collective (scatter, gather, and all-to-all) communication algorithms for

performing data exchange, selecting the best performer for a given data size and number of processors. The clear advantages of using the collective gather and all-to-all primitives for data redistribution were demonstrated. Portions of this work have appeared in [77].

We extended the framework for quantifying scalability specifically for data redistribution algorithm-machine pairs. We proposed a metric based on *slope*, which is measured as the ratio of change in execution time to the change in the number of processors. We quantified the scalability of DaReL using this metric and demonstrated its scalability for a large range of data and processor configuration sizes.

7.2 Future Work

There are a number of areas pertaining to efficient data redistribution services that could be pursued in the future. The impact of data alignment and realignment on redistribution represents one such area. Data alignments may cause an unbalanced amount of data exchange between processor memories during data redistribution. Such imbalances may lead to increased redistribution execution time. Another area of future investigation would be a general theory for minimizing data exchange among processors using HPF regular distribution patterns. For instance, optimal processor-data mappings for $\text{CYCLIC}(c_1)$ to $\text{CYCLIC}(c_2)$ redistributions where $c_1 \neq c_2$ have not been addressed. Furthermore, the impact of data alignments may also influence a general theory. Further investigation of DaReL's scalability and performance on other distributed-memory platforms supporting MPI would be desirable. First, it would

validate the claim of portability; and second, it would be beneficial to investigate whether the relative merits of the different message-passing algorithms are consistent on other hardware platforms. Perhaps, other, more efficient, algorithms can be found. Further study on different hardware platforms may suggest new scalability criteria for data redistribution as well. Finally, we anticipate that DaReL can be used not only for explicit redistribution, but also in instances of implicit redistribution. The fundamental operation of DaReL would remain unchanged; however, the interface to the compiler may change. Also, optimization of certain components of the library unneeded for implicit redistribution may be found.

BIBLIOGRAPHY

Bibliography

- [1] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu, "Fortran D language specification," Tech. Rep. COMP TR90-141, Rice University, Department of Computer Science, Dec. 1990.
- [2] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald, *Vienna Fortran: A Language Specification (Version 1.1)*, 1991.
- [3] High Performance Fortran Forum, "High Performance Fortran Language Specification (version 1.0, draft)," May 1993.
- [4] P. J. Hatcher and M. J. Quinn, *Data-Parallel Programming On MIMD Computers*. Cambridge, Massachusetts: MIT Press, 1991.
- [5] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr, "Implementing a parallel C++ runtime system for scalable parallel systems," in *Proceedings of Supercomputing'93*, pp. 588–597, Nov. 1993.
- [6] M. Mace, *Memory Storage Patterns in Parallel Processing*. Kluwer Academic, 1987.
- [7] J. Li and M. Chen, "The data alignment phase in compiling programs for distributed-memory machines," *Journal of Parallel and Distributed Computing*, vol. 13, pp. 213–221, Oct. 1991.
- [8] L. M. Ni, H. Xu, and E. T. Kalns, "Issues in scalable library design for massively parallel computers," in *Supercomputing'93*, pp. 181–190, Nov. 1993.
- [9] U. Kremer, "Automatic data layout for distributed-memory machines," Tech. Rep. CRPC-TR93299-S, Rice University, 1993.
- [10] W. Press, *Numerical Recipes in C*. Cambridge, 1988.
- [11] P. Mehrotra and J. V. Rosendale, *Programming Distributed Memory Architectures Using Kali*, ch. 19, pp. 364–384. MIT Press, 1991.
- [12] M. Rosing, R. B. Schnabel, and R. P. Weaver, "The DINO parallel programming language," *Journal of Parallel and Distributed Computing*, vol. 13, pp. 30–42, Sept. 1991.

- [13] M. Baber, "Hypertasking support for dynamically redistributable and resizable arrays on the iPSC," in *Proceedings of the Sixth Distributed Memory Computing Conference*, pp. 59–66, Apr. 1991.
- [14] B. Chapman, P. Mehrotra, H. Moritsch, and H. Zima, "Dynamic data distributions in Vienna Fortran," in *Proceedings of Supercomputing'93*, pp. 284–293, Nov. 1993.
- [15] M. W. Hall, S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Interprocedural compilation of Fortran D for MIMD machines," in *Proceedings of Supercomputing'92*, pp. 522–534, Nov. 1992.
- [16] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," tech. rep., University of Tennessee, Mar. 1994.
- [17] High Performance Fortran Forum, "HPF-2 scope of activities and motivating applications," Nov. 1994.
- [18] M. Ikei and M. Wolfe, "Automatic array alignment for distributed memory multicomputers," in *Proceedings of the Hawaii International Conference on Systems Sciences*, Jan. 1994.
- [19] M. Gupta and P. Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 179–193, Mar. 1992.
- [20] J. M. Anderson and M. S. Lam, "Global optimizations for parallelism and locality on scalable parallel machines," in *ACM SIGPLAN Notices, Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, vol. 28, pp. 112–125, June 1993.
- [21] E. T. Kalns, H. Xu, and L. M. Ni, "Evaluation of data distribution patterns in distributed-memory machines," in *Proceedings of the 1993 International Conference on Parallel Processing*, vol. II, pp. 175–183, Aug. 1993.
- [22] H. Xu and L. M. Ni, "Optimizing data decomposition for data parallel programs," in *Proceedings of the 1994 International Conference on Parallel Processing*, vol. II, pp. 225–232, Aug. 1994.
- [23] R. Ponnusamy, J. Saltz, R. Das, C. Koelbel, and A. Choudhary, "Embedding data mappers with distributed memory machine compilers," in *ACM SIGPLAN Notices, Workshop on Languages, Compilers, and Run-Time Environments For Distributed Memory Multiprocessors*, vol. 28, pp. 52–55, Jan. 1993.
- [24] P. Z. Lee and T.-B. Tsai, "Compiling efficient programs for tightly-coupled distributed memory computers," in *Proceedings of the 1993 International Conference on Parallel Processing*, vol. 2, pp. 161–165, Aug. 1993.

- [25] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiling Fortran D for MIMD distributed-memory machines," *Communications of the ACM*, vol. 35, pp. 66–80, Aug. 1992.
- [26] S. Chittor and R. Enbody, "Performance evaluation of mesh-connected wormhole-routed networks for interprocessor communication in multicomputers," in *Proceedings of the 1990 Supercomputing Conference*, pp. 647–656, Nov. 1990.
- [27] S. Chittor and R. Enbody, "Predicting the effect of mapping on the communication performance of large multicomputers," in *Proceedings of the 1991 International Conference on Parallel Processing*, vol. 2, pp. 1–4, Aug. 1991.
- [28] S. Chatterjee, J. R. Gilbert, and R. Schreiber, "Mobile and replicated alignment of arrays in data-parallel programs," in *Proceedings of Supercomputing'93*, Nov. 1993.
- [29] K. Kunchithapadam and B. P. Miller, "Optimizing array distributions in data-parallel programs." 7th Annual Workshop on Languages and Compilers for Parallel Computing, August 1994.
- [30] C. M. Chase and A. P. Reeves, "Data remapping for distributed-memory multicomputers," in *Proceedings of the 1992 Scalable High Performance Computing Conference*, pp. 137–144, Apr. 1992.
- [31] S. Gupta, S. Kaushik, S. Mufti, S. Sharma, C.-H. Huang, and P. Sadayappan, "On the generation of efficient data communications for distributed-memory machines," in *Proceedings of the 1992 International Computer Symposium*, (Taichung, Taiwan), pp. 504–513, Dec. 1992.
- [32] R. Thakur, A. Choudhary, and G. Fox, "Runtime array redistribution in HPF programs," in *Proceedings of the 1994 Scalable High Performance Computing Conference*, pp. 309–316, May 1994.
- [33] J. Stichnoth, D. O'Hallaron, and T. Gross, "Generating communication for array statements: Design implementation, and evaluation," *Journal of Parallel and Distributed Computing*, vol. 21, no. 1, pp. 150–159, 1994.
- [34] S. Ramaswamy and P. Banerjee, "Automatic generation of efficient array redistribution routines for distributed memory multicomputers," Tech. Rep. CRHC-94-09, Center for Reliable and High Performance Computing, Computer Systems and Research Laboratory, University of Illinois, 1994.
- [35] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su, "The PARADIGM compiler for distributed-memory message passing," in *First International Workshop on Parallel Processing*, (Bangalore, India), Dec. 1994.

- [36] J. Bruck, R. Cypher, C.-T. Ho, and S. Kipnis, "Efficient algorithms for the index operation in message-passing systems," Tech. Rep. RJ 9300 (82230), IBM Research Division, Almaden Research Center, San Jose, CA and T.J. Watson Research Center, Yorktown Heights, NY, Apr. 1993.
- [37] P. K. McKinley, Y.-J. Tsai, and D. F. Robinson, "A survey of collective communication in wormhole-routed massively parallel computers," Tech. Rep. MSU-CPS-94-35, Department of Computer Science, Michigan State University, 1994.
- [38] C.-T. Ho and S. L. Johnsson, "Distributed routing algorithms for broadcasting and personalized communication in hypercubes," in *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 640–648, Aug. 1986.
- [39] P. K. McKinley, H. Xu, A.-H. Esfahanian, and L. M. Ni, "Unicast-based multicast communication in wormhole-routed networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 1252–1265, Dec. 1994.
- [40] S. D. Kaushik, C.-H. Huang, R. W. Johnson, and P. Sadayappan, "An approach to communication-efficient data redistribution," in *Proceedings of the 1994 International Conference on Supercomputing*, pp. 364–373, July 1994.
- [41] S. L. Johnsson and C.-T. Ho, "The complexity of reshaping arrays on boolean cubes," in *Proceedings of the Fifth Distributed Memory Computing Conference*, vol. 1, pp. 370–377, Apr. 1990.
- [42] S. L. Johnsson, "Language and compiler issues in scalable high performance scientific libraries," 1992. (Tutorial Notes at Frontiers'92).
- [43] A. Wakatani and M. Wolfe, "A new approach to array redistribution: Strip mining redistribution," in *PARLE '94 Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece*, pp. 323–335, July 1994.
- [44] V. Bala and J. Ferrante, "Explicit data placement (XDP): A methodology for explicit compile-time representation and optimization of data movement, (extended abstract)," in *ACM SIGPLAN Notices, Workshop on Languages, Compilers, and Run-Time Environments For Distributed Memory Multiprocessors*, vol. 28, pp. 36–39, Jan. 1993.
- [45] V. Bala, J. Ferrante, and L. Carter, "Explicit data placement (XDP): A methodology for explicit compile-time representation and optimization of data movement," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* *ACM SIGPLAN Notices*, vol. 28, pp. 139–148, July 1993.
- [46] S. Amarasinghe and M. Lam, "Communication optimization and code generation for distributed memory machines," in *ACM SIGPLAN Notices, Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, vol. 28, pp. 126–138, June 1993.

- [47] A. Wakatani and M. Wolfe, "Optimization of the redistribution of arrays for distributed memory multicomputers," tech. rep., Matsushita Electric Industrial Co., Ltd., Osaka, Japan and Dept. of Computer Science and Engineering, Oregon Graduate Institute, Portland, OR, Dec. 1993.
- [48] A. Skjellum, N. E. Doss, and P. V. Bangalore, "Writing libraries in MPI," in *Proceedings of the 1993 Scalable Parallel Libraries Conference*, pp. 166–173, Oct. 1993.
- [49] A. Skjellum and A. P. Leung, "Zipcode: A portable multicomputer communication library atop the reactive kernel," in *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, pp. 767–776, Apr. 1990.
- [50] D. W. Walker, "The design of a standard message passing interface for distributed memory concurrent computers," Tech. Rep. ORNL/TM-12512, Oak Ridge National Laboratory, Oak Ridge, TN, Oct. 1993.
- [51] J.C.Adams, W.S.Brainerd, J.T.Martin, B.T.Smith, and J.L.Wagener, *Fortran 90 Handbook*. 1221 Avenue of the Americans, New York, NY 10020: Intertext Publications, 1992.
- [52] A. Skjellum, N. E. Doss, and K. Viswanathan, "Inter-communicator extensions to MPI in the MPIX (MPI eXtension) library," tech. rep., Department of Computer Science and NSF Engineering Research Center for Computational Field Simulation, Mississippi State University, Mississippi State, MS, Aug. 1994.
- [53] W. Ware, "The ultimate computer," *IEEE Spectrum*, vol. 9, pp. 84–91, 1972.
- [54] G. Amdahl, "Validity of the single-processor approach to achieving large scale computing capabilities," in *Proceedings AFIPS Conference*, pp. 483–485, 1967.
- [55] J. L. Gustafson, G. R. Montry, and R. E. Benner, "Development of Parallel Methods for a 1024-Processor Hypercube," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, pp. 609–638, July 1988.
- [56] J. L. Gustafson, "Reevaluating Amdahl's Law," *CACM*, vol. 31, pp. 532–533, May 1988.
- [57] X.-H. Sun and D. T. Rover, "Scalability of parallel algorithm-machine combinations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 599–613, June 1994.
- [58] A. Grama, A. Gupta, and V. Kumar, "Isoefficiency: Measuring the scalability of parallel algorithms and architectures," *IEEE Parallel and Distributed Technology*, vol. 1, pp. 12–21, Aug. 1993.
- [59] C.-C. Lin, "Isoefficiency analysis of parallel systems with limited memory," in *Contributed Papers of the IPPS 93 Workshop on Analyzing Scalability of Parallel Algorithms and Architectures*, pp. 11–15, Apr. 1993.

- [60] V. Kumar and A. Gupta, "Analyzing scalability of parallel algorithms and architectures," *Journal of Parallel and Distributed Computing - Special Issue on Scalability of Parallel Algorithms and Architectures*, vol. 22, pp. 379–391, Sept. 1994.
- [61] J. L. Gustafson, "The design of a scalable, fixed-time computer benchmark," *Journal of Parallel and Distributed Computing*, vol. 12, pp. 388–401, 1991.
- [62] J. L. Gustafson and Q. O. Snell, "HINT: A new way to measure computer performance," Tech. Rep. IS-5109, UC-405, Ames Laboratory, Ames, Iowa, 50011-3020, 1994.
- [63] P. Dinda, T. Gross, D. O'Hallaron, E. Segall, J. Stichnoth, J. Subhlok, J. Webb, and B. Yang, "The CMU task parallel program suite," Tech. Rep. CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1994.
- [64] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors: Volume I: General Techniques and Regular Problems, Chapter 7*. Prentice Hall, 1988.
- [65] S. Sharma and High Performance Fortran Forum, "HPF-2 scope of activities and motivating applications," Nov. 1994. (Molecular Dynamics application, Section 4.3).
- [66] W. Gropp and E. Lusk, "User's guide for the ANL IBM SPx system." URL <http://www.mcs.anl.gov/Projects/sp1/guide-r2.html>, 1995.
- [67] Message Passing Interface Forum (MPIF), "MPI: A Message-Passing Interface Standard," tech. rep., University of Tennessee, May 1994.
- [68] P. Bridges, N. Doss, W. Gropp, E. Karrels, E. Lusk, and A. Skjellum, "Users' guide to MPICH, a portable implementation of MPI," tech. rep., Argonne National Laboratory and Mississippi State University, 1994.
- [69] Argonne National Laboratory and Mississippi State University, "Comments from C-based source code of MPICH reduction primitive," 1994.
- [70] H. Franke, P. Hochschild, P. Pattnaik, and M. Snir, "MPI-F: An efficient implementation of MPI on IBM SP-1," in *Proc. of the 1994 International Conference on Parallel Processing*, vol. 3, pp. 197–201, Aug. 1994.
- [71] H. Franke, P. Hochschild, P. Pattnaik, J.-P. Prost, and M. Snir, "MPI on IBM SP1/SP2: Current status and future directions," in *Proceedings of the 1994 Scalable Parallel Libraries Conference*, Oct. 1994.
- [72] H. Franke, "MPI-F an implementation for IBM SP-1/SP-2," Tech. Rep. Version 1.39, IBM T. J. Watson Research Center, Yorktown Heights, NY, Feb. 1995.

- [73] N. Nupairoj and L. Ni, "Benchmarking of multicast communication services," Tech. Rep. MSU-CPS-ACS-103, Department of Computer Science, Michigan State University, Apr. 1995.
- [74] E. T. Kalns and L. M. Ni, "Processor mapping techniques toward efficient data redistribution," in *Proceedings of the 8th International Parallel Processing Symposium*, pp. 469–476, Apr. 1994.
- [75] E. T. Kalns and L. M. Ni, "Processor mapping techniques toward efficient data redistribution," *IEEE Transactions on Parallel and Distributed Systems*. (accepted to appear).
- [76] E. T. Kalns and L. M. Ni, "A framework for quantifying scalability," Tech. Rep. MSU-CPS-ACS-100, Department of Computer Science, Michigan State University, Mar. 1995. submitted to Supercomputing '95.
- [77] E. T. Kalns and L. M. Ni, "DaReL: A portable data redistribution library for distributed-memory machines," in *Proceedings of the 1994 Scalable Parallel Libraries Conference II*, Oct. 1994.

MICHIGAN STATE UNIV. LIBRARIES



31293013995588