



3 1293 01020 1519

This is to certify that the

dissertation entitled

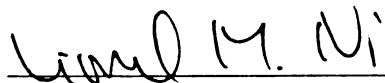
Compilation-Time Data Decomposition
Optimization for Data Parallel Programs

presented by

Hong XU

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science


Major professor

Date November 9, 1994

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
FEB 02 1998		

**COMPILATION-TIME DATA
DECOMPOSITION OPTIMIZATION FOR
DATA PARALLEL PROGRAMS**

By

Hong Xu

A DISSERTATION

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

DOCTOR OF PHILOSOPHY

Department of Computer Science

1994

ABSTRACT

**COMPILATION-TIME DATA
DECOMPOSITION OPTIMIZATION FOR
DATA PARALLEL PROGRAMS**

By

Hong Xu

Data decomposition is critical to the performance of data parallel programs on scalable parallel computers. A data decomposition model can be viewed as a two-level mapping of array elements to abstract processors. Data alignment determines what array elements are aligned relative to one another, and data distribution resolves how the group of aligned arrays is distributed onto the processors. Depending on the alignment relationship as within dimension or across dimension, alignment can be classified into base alignment and offset alignment.

The purpose of base alignment is to reduce the amount of unstructured communication. In this research, we use the data reference graph model to describe the various reference patterns associated with each array and to resolve the conflict of the compatible alignment requirements. An efficient spanning tree algorithm addresses the fundamental issues in base alignment. Base alignment is further studied with the consideration of the optimal expression evaluation and dataflow analysis. Efficient base

alignment algorithms are proposed to reduce the redundant communication and optimize the RHS expression evaluation. These contributions make this research unique from related research.

The purpose of offset alignment is to reduce the amount of data shift movement. This thesis successfully models the cost of data shift movement using the piecewise linear function. This cost model solves the accuracy problem in measuring the quantity of data shift movement, an unresolved problem left by other work in this area. Based on this cost model, the optimal post-alignment algorithm is first proposed to exceed the limitation of the owner-computes rule and minimize the amount of data shift movement after offset alignment is determined. The data reference graph model is used to address the problem of offset alignment and develop efficient spanning tree algorithms. The RHS expression evaluation and dataflow optimizations are incorporated with the proposed offset alignment algorithms.

The purpose of data distribution is to reduce the impact of data shift movement and increase processor workload balance. Segment distribution is proposed to resolve the conflict between reducing data shift movement and increasing processor workload balance with regard to a particular dimension of the template array. An optimal processor allocation algorithm is introduced to minimize the overall cost of data shift communication across multiple dimensions of the template array. The segment distribution and optimal processor allocation proposed in this thesis provide the best data distribution support for most data parallel programs.

To my parents

ACKNOWLEDGMENTS

I wish to thank my thesis advisor, Prof. Lionel Ni, for his considerable help and guidance with my thesis and my doctoral program. His patience in listening to initial versions of many of these results is especially appreciated. His suggestions and questions have greatly improved both the accuracy and the presentation of this thesis. Without his enthusiasm and encouragement, the completion of this thesis work would be impossible.

I would like to thank Prof. Tien Yien Li, Prof. Abdol Esfahanian, Prof. Philip McKinley, and Prof. Diane Rover, the members of my Ph.D guidance committee for their guidance and support during my thesis work. I would particularly like to express my appreciation to Prof. Diane Rover for her careful reading of this dissertation and her many useful comments.

Special thanks go to my parents for constant encouragement and support for everything I do.

Lastly and most importantly, I would like to thank my wife, Yiru, who makes everything I do worthwhile.

TABLE OF CONTENTS

LIST OF TABLES	ix
LIST OF FIGURES	x
1 Introduction	1
1.1 Scalable Parallel Computer Architectures	2
1.1.1 The NUMA Multiprocessors	3
1.1.2 The Message-Passing Based Multiprocessors	3
1.1.3 Workstation Clusters	4
1.2 Data Parallel Programming Model	5
1.3 Motivation and Problem Definition	6
1.4 Objectives and Scope of Research	9
1.5 Thesis Outline	11
2 Data Decomposition Overview	13
2.1 Loop Characteristics	14
2.1.1 Parallelizable Loop	14
2.1.2 Perfectly Nested Loops	15
2.1.3 Special-Purpose Loops	17
2.2 Interprocessor Communication	17
2.3 Processor Workload Balance	18
2.4 The Model for Data Decomposition	19
2.4.1 Formulation for Data Alignment and Distribution	20
2.4.2 Layered Structure for Data Decomposition	21
2.4.3 The Base Alignment Phase	22
2.4.4 The Offset Alignment Phase	28
2.4.5 The Data Distribution Phase	30
2.5 Data Re-distribution and Data Re-Alignment	34
2.6 Data Flow Analysis	36
2.7 Related Work	37

2.7.1	Data Parallel Languages	39
2.7.2	Preference Graph Model	40
2.7.3	Using Communication Cost Estimation	41
2.7.4	Dynamic Programming Methods	42
2.7.5	Linear Algebra Methods	42
2.7.6	Parallelizing Loops with Data Dependence	43
3	Base Alignment	45
3.1	Terminology	45
3.2	Base Alignment for Single Reference	48
3.2.1	Base Alignment Equation	48
3.2.2	A Legitimate Solution of Alignment Matrix	51
3.2.3	Solving Base Alignment Equation	56
3.3	The Cost of Reorganization Communication	58
3.3.1	Reorganization Communication	58
3.3.2	The Weighted Cost	63
3.4	Spanning-Tree Base Alignment Algorithm	64
3.4.1	Data Reference Graph	64
3.4.2	Spanning Tree Base Alignment Algorithms	68
3.4.3	Experimental Results	73
3.5	Optimizing RHS Expression Evaluation	74
3.5.1	RHS Expression Evaluation Optimization	74
3.5.2	Alignment Graph	76
3.5.3	AG Base Alignment Algorithm	77
3.5.4	Experimental Results	80
3.6	Avoiding Redundant Communication	81
3.6.1	Redundant Communication	81
3.6.2	Enhanced Alignment Graph	83
3.6.3	EAG Base Alignment Algorithm	84
4	Offset Alignment	88
4.1	Offset Alignment for Single Reference	88
4.1.1	Offset Alignment Equation	89
4.1.2	Multiple Aligned Base Groups	91
4.1.3	Calculating Alignment Offset	94
4.2	The Cost of Neighboring Communication	96
4.2.1	The Basic Cost	96
4.2.2	The Weight of the Basic Cost	98

4.3	The Impact of Access Offset	102
4.3.1	Piecewise Linear Cost Function	102
4.3.2	Properties of Piecewise Linear Cost Function	105
4.4	Spanning-Tree Offset Alignment Algorithm	112
4.4.1	Offset Reference Graph	112
4.4.2	Spanning Tree Offset Alignment Algorithms	115
4.5	Optimizing RHS Expression Evaluation	120
4.5.1	RHS Expression Evaluation Optimization	120
4.5.2	Post-Alignment Optimization	122
4.5.3	Alignment Graph	124
4.5.4	AG-Based Offset Alignment Algorithm	125
4.5.5	Performance Comparison	128
4.6	Avoiding Redundant Communication	129
4.6.1	Redundant Communication	129
4.6.2	Enhanced Alignment Graph	132
4.6.3	EAG-Based Offset Alignment Algorithm	132
5	Data Distribution	136
5.1	Segment Distribution	136
5.1.1	The Limitation of Existing Distribution Types	136
5.1.2	Segment Distribution	140
5.1.3	Multiple-Variable Density Function	146
5.1.4	Experimental Results	150
5.2	Virtual Processor Allocation	151
5.2.1	Reducing the Overall Neighboring Communication	151
5.2.2	Optimal Processor Allocation	153
5.2.3	Performance Results	157
6	Conclusions and Future Research	160
6.1	Research Contributions	160
6.2	Directions of Future Work	163
	BIBLIOGRAPHY	165

LIST OF TABLES

2.1	Related work in base alignment analysis	38
2.2	Related work in offset alignment analysis	39
2.3	Related work in data distribution analysis	39
3.1	Values of T , Q_1 , and Q_2 in executing the MICS algorithm for Example 7	72
3.2	Resolving base alignment for Example 9	79
3.3	Resolving base alignment for Example 10	87
4.1	Values of T , Q_1 , and Q_2 in executing the MICS algorithm for Example 13	119
4.2	Resolving offset alignment for Example 13 by using the AGOA algorithm	127
4.3	Comparison of the MWST, STOA, and AGOA algorithms using Ex- ample 13	129

LIST OF FIGURES

2.1	Pipeline computation and communication	16
2.2	The model for data decomposition	19
2.3	Layered structure for data decomposition process	22
2.4	Example 1: A NAS benchmark loop	23
2.5	Base alignment in Example 1	24
2.6	Example 2: A Purdue benchmark loop	26
2.7	Base alignment in Example 2	26
2.8	Example 3: A NAS benchmark loop	28
2.9	Offset alignment in Example 3	29
2.10	Data distribution in Example 3	31
2.11	Example 4: A Heatwave benchmark loop	32
2.12	Processor allocation in Example 4	33
2.13	A dynamic programming algorithm for data re-alignment and data re-distribution	35
3.1	Example 1 in single occurrence statements	46
3.2	The compactness of the image on T	52
3.3	Example 5: A Whetstone benchmark loop	53
3.4	The alignment for A and B in Example 5	55
3.5	Example 6: Inner product benchmark loop	56
3.6	Reorganization communication for $D_X = (1, -1)$ and $D_Y = (1, 1)$ in Example 1	61
3.7	Example 7: A Lapack benchmark loop	65
3.8	The DRG and base alignment for Example 7	66
3.9	The minimum-weight induced communication algorithm	70
3.10	Use the MICS algorithm to resolve base alignment for Example 7 . .	71
3.11	Comparison of MWST algorithm and MICS algorithm on 16-node nCUBE-2	73
3.12	Example 8: A Splash benchmark loop	74
3.13	Optimal evaluation trees for Example 8	75

3.14	Example 9: An Electric benchmark loop	77
3.15	Alignment graph for Example 9	77
3.16	The alignment graph base alignment algorithm	78
3.17	Comparison of the MWST, MICS, and AGBA algorithms on 16-node nCUBE-2	80
3.18	Example 10: An Oceanwater benchmark loop	82
3.19	Enhanced alignment graph for Example 10	84
3.20	The enhanced alignment graph base alignment algorithm	85
4.1	Example 11: An Eispack benchmark loop	91
4.2	Example 12: A Weather-Climate benchmark loop	95
4.3	Neighboring communication in Example 3	100
4.4	The sum of $c_{Y,1}$ and $c_{Y,3}$	106
4.5	The sum of $c_{B,j}$ and $c_{B,k}$	108
4.6	The minimal-cost pairwise offset alignment algorithm	110
4.7	Example 13: A Livermore benchmark loop	113
4.8	The offset reference graph for Example 13	114
4.9	The offset reference graph for Example 13	116
4.10	The spanning-tree offset alignment algorithm	117
4.11	Resolving offset alignment for Example 13	118
4.12	Example 14: A Livermore Kernel 7 loop	120
4.13	The optimal evaluation trees for Example 14	121
4.14	An example of post-alignment optimization	124
4.15	The AG for Example 13	125
4.16	The alignment graph offset alignment algorithm	126
4.17	Example 15: A Dhrystone benchmark loop	130
4.18	Enhanced alignment graph for Example 15	132
4.19	The enhanced alignment graph offset alignment algorithm	133
5.1	Example 16: An Electromagnetic benchmark loop	137
5.2	Different types of data distribution for Example 16	138
5.3	Linpack TQL2 benchmark loop	141
5.4	Some common loop patterns	143
5.5	An example of no optimal distribution pattern	149
5.6	Comparison of processor workload balance between block distribution and segment distribution	150
5.7	Two different processor allocation patterns for Example 4	152
5.8	A livermore kernel 18 benchmark loop	158

5.9	Comparison of different processor allocation strategies on a 64-node	
	nCUBE-2	158

CHAPTER 1

Introduction

Sequential computers are approaching a fundamental physical limit, the speed of light, on their potential computational power. Such a limit cannot satisfy the computation power requirement of the so-called grand-challenge problems including three-dimensional fluid flow calculations, real-time simulations of complex systems, and intelligent robots which require over 1,000 times the computing power of the maximum-power uniprocessors. It has been widely recognized that parallel computing represents the only plausible way to continue to increase the computational power available to such grand-challenge applications. Such parallel computers, also known as *scalable parallel computers* (SPCs), are designed to offer corresponding increases in the processing capability of the system, memory bandwidth, and total interprocessor communication bandwidth as the number of processors and the number of memory modules are increased.

However, programming on SPCs is much more complicated than programming on uniprocessor computers. The naive approach of exposing system architecture features directly to programmers has been proved to be time-consuming, tedious, and error-prone due to the difficulty in compromising various conflicting requirements arising from concurrent processing and distributed data allocation. As a result, an easy-to-use and highly efficient programming model is critical to the success of SPCs.

The data-parallel programming model has been accepted as one of the most efficient programming models provided for SPCs. The basic idea of a data-parallel program is to decompose the global data objects across the processors each of which executes the same program structure on the data objects it owns. Based on this type of *single-program-multiple-data* (SPMD) mode of computation, data decomposition determines both the process scheduling and interprocessor communication. Selecting an efficient data decomposition is one of the most important intellectual steps in developing quality data-parallel programs.

The purpose of this dissertation research is to make an in-depth study of optimizing data decomposition for data-parallel programs. The proposed algorithmic results provide a framework for data decomposition optimization used by parallelizing compilers in optimizing user-written data-parallel programs at compilation-time. The theoretical results obtained in this thesis are derived from the fundamental architecture features among various existing SPC platforms. It is not the intention of this thesis to address any specific issues involved in a particular machine architecture. Instead, our proposed framework has chosen the machine-independent approach as the focus of our study.

In this introductory chapter, we provide a brief review of the fundamental features of SPC architectures, present an overview of data-parallel programming model and its implementations, and describe the objectives and scope of this research.

1.1 Scalable Parallel Computer Architectures

There are two major classes of parallel computers: *shared-memory multiprocessors* and *message-passing multiprocessors*. Differing in how the memory is shared and distributed among the processors, shared-memory multiprocessors can be further classified into *uniform-memory-access* (UMA) multiprocessors and *non-uniform-memory-*

access (NUMA) multiprocessors. In UMA multiprocessors, all processors have equal access time to all memory words. This feature prevents maximizing the performance from the impact of data locality with regard to data allocation in the memory. However, limited by the current technique and cost, the delayed memory access time due to the added interconnection network significantly degrades the overall performance as the number of the processors and memory modules increases. For this reason, it is believed that UMA multiprocessors shall not be a good candidate for SPCs.

1.1.1 The NUMA Multiprocessors

Typically in NUMA multiprocessors, each processor has a *local memory*. The collection of all local memories forms a global address space accessible by all processors. However, the memory access time varies with the location of the memory word. It is quicker to access a local memory with a local processor. The access of remote memory attached to other processors takes longer due to the added delay through the interconnection network. BBN TC-2000 [1] is an example of the commercial machines using NUMA architecture. The TC-2000 can be configured to have up to 512 M88100 processors interconnected by a multistage cube network. In the TC-2000, the remote memory access time is about five times as expensive as the local memory access time. Besides the TC-2000, the Cray T3D [2] and Convex Exemplar [3] are two other important commercial machines using NUMA architecture.

1.1.2 The Message-Passing Based Multiprocessors

Distributed-memory multiprocessors are organized as ensembles of nodes, each of which is a programmable computer with its own processor, local memory, and other supporting devices. As the number of nodes in the system increases, the total communication bandwidth, memory bandwidth, and processing capability of the system

also increase. Nodes communicate each other by passing messages through the interconnection network. For this reason, distributed-memory multiprocessors are also known as the *message-passing* based multiprocessors. A novel switching technique, known as *wormhole routing* [4], uses a cut-through approach to reduce the impact of the physical distance between two communicating nodes. The commercial machines characterized by the message-passing based multiprocessors include the Intel Paragon (successor to Touchstone DELTA [5]), Ncube nCUBE-2 [6], Meiko CS-2 [7], and TMC CM-5 [8]. However, in those systems, the communication latency is orders of magnitude as expensive as the local memory access.

1.1.3 Workstation Clusters

Recently, the evolution of fast LAN-connected workstation cluster has the trend in creating yet another kind of SPCs. The high-performance workstation clusters interconnected through high-speed switches have been advocated in the place of special-purpose multicomputers. The IBM SP-1 [9] development has already moved in this direction. In the SP-1 configuration, a collection of IBM RS6000s are interconnected through the IBM high-performance Vulcan switch [10]. Though great efforts are being made to reduce the communication overhead involved in operating systems and communication protocols [11, 12], message transmission is still much more expensive than local memory access in workstation clusters.

Overall, the hierarchy of the local memory and remote memory with significant access latency difference characterizes the fundamental feature of the current existing SPCs. Exploring data locality is critical to the performance of message-passing based multicomputers, NUMA multicomputers, and workstation clusters. The data objects referenced by each processor should be arranged to reside in the local memory with that processor in order to avoid the communication overhead added by remote data access. The importance of data locality motivates the research of this thesis. It should

be emphasized that it is not the intent of this thesis to consider the detailed implementation of remote data access, such as remote memory reference through the crossbar in NUMA multicomputers or message transmission through the interconnection network in distributed-memory multiprocessors. Our research is based on the abstract memory hierarchy model which characterizes the existing SPC architectures.

1.2 Data Parallel Programming Model

The data parallel programming model has been recognized as one of the most successful programming models for the SPCs. Data objects are pre-distributed across the local processor memories before a data parallel program is executed. During the execution, each processor runs an identical copy of the original program but only writes to the data objects owned by that processor. As a result, instruction partition in data parallel programs is fully determined by data partition. With the similar concept of lockstep operations in the SIMD programming model, the data parallel programs are easier to write, easy to port, and much more scalable. However, unlike the SIMD programming model, the data parallel programs emphasize medium-grain parallelism and synchronization at the subprogram level rather than at the instruction level.

It has been commonly accepted that the programming languages supporting global name space are much easier to use than the programming languages supporting separate name space, in particular for those scientific application programmers who wish to write the programs in some dialect of Fortran when using the SPCs. One of the greatest advantages that the data parallel programming model has is the nature of supporting global name space at the language level. Many successful data parallel languages, including Fortran 90 [13] and High Performance Fortran (HPF) [14], support global name space.

Though programmers may give some hint about how to decompose the data ob-

jects using specific language extensions, compiler is responsible for arranging and optimizing data allocation across the processor local memories. The type of memory space addressing is hidden from the viewpoint of programmers. If the underlying memory architecture serves the global address space, the reference to a data object owned by a remote processor is simply achieved by a remote memory access. If the underlying system architecture only supports separate address space, the reference to a data object owned by a remote processor is implemented by a proper message passing which is inserted at compilation time. Such a data parallel programming model maximizes the programmability.

1.3 Motivation and Problem Definition

There are two levels of data parallel activities in data-parallel application programs. First, there is the question of how arrays should be *aligned* with respect to one another, both within and across array dimensions. This is called as the *problem mapping* [15, 16] induced by the structure of the underlying computation. It represents the minimal requirements for reducing data movement for the program, and largely independent of any machine considerations. The alignment of arrays in the program depends on the natural fine-grain parallelism defined by individual members of data arrays.

Second, there is the question of how arrays should be *distributed* onto the SPCs. This is called as the *machine mapping* caused by translating the computation structure onto the finite resources of the machine. Data distribution provides opportunities to reduce data movement, but must also maintain load balance. The distribution of arrays in the program depends on the coarse-grain parallelism defined by the SPCs.

The objective of data decomposition is to minimize data movement across distinct processor local memories and maximize the processor workload balance among all processors. The first effort of reducing data movement is to optimize data alignment.

There are two-level of alignment: base alignment and offset alignment. Base alignment determines the alignment across dimensions of various arrays. Offset alignment specifies the alignment within each dimension of various arrays.

Traditionally, data arrays are decomposed based on each dimension and base alignment is dimension-based alignment. However, the limitation of such a dimension-based alignment prevents the optimizer from exploring the inherit parallelism available in many programming structures. Recently, this limitation has been removed by partitioning a data array in a family of parallel hyperplanes in the linear-tangle space defined by the data array [17, 18]. Array elements residing on the same hyperplane are allocated to the same processor local memory such that any unstructured data reference among elements on the same hyperplane is free of interprocessor communication. Based on this new partitioning strategy, this thesis proposes efficient base alignment algorithms to resolve the conflicted communication-free partitions imposed by different computation structures.

The goal of offset alignment is to reduce data shift movement. The previous work [19] on the offset alignment problem focuses on the SIMD programming model on the SIMD multiprocessors, where each processor is assigned a single element in each array operand. In the SPMD programming model on the SPCs, however, each processor can be assigned a collection of elements in the same data array. As a result, the amount of shifted data with respect to each local processor is the accumulation of all shifted data requested by defining each element owned by the local processor. Since each processor owns a collection of data elements in SPMD programs, the shifted data requested by defining one local element has great chance to overlap another local element on the same processor. However, this fact has been ignored by other research works which have been done so far in the area of data decomposition. This thesis establishes a mathematical framework to model the problem of offset alignment for data parallel programs. The theoretical results built upon this framework provide

efficient solutions in optimizing offset alignment.

This thesis emphasizes the impact of optimal expression evaluation to the data alignment analysis [20, 21]. The traditional implementation of data parallel programs follows the *owner-computes* rule in which all the *right-hand side* (RHS) operands must be first transmitted to the local processor and then the evaluation of the RHS is taken place on that local processor. Data movement may be minimized by evaluating an intermediate result on a remote processor rather than the local processor which owns the *left-hand side* (LHS) operand. Of course, this approach of optimal expression evaluation is based on the presumption that the associate and commutative properties of the RHS are not violated. The issues of optimizing expression evaluation for SIMD mode of programming have been addressed in [19]. However, the algorithm in [19] is given only for a single assignment statement with the assumption that the alignment of each array operand is given. This thesis pioneerly studies optimal expression evaluation with regard to multiple assignment statements. The algorithmic results given by this thesis take advantage of such optimal expression evaluation in resolving both base alignment and offset alignment.

In traditional parallelizing compilers, interprocessor communication optimization including redundant message avoidance, message vectorization, and overlapping communication with computation is performed after data decomposition analysis. However, it is not true that the profitability of every communication optimization technique is only passively determined by the result of data decomposition. In this thesis, the effect of redundant communication avoidance is considered in the first place during the decision making for data alignment. More efficient solution of data alignment can be obtained by the alignment algorithms proposed in this thesis because the dataflow analysis eliminates the impact of redundant communication in cost estimation and assists the data alignment analyzer to make a more accurate decision. Message vectorization usually reduces the software latency per message, provided that the software

latency in a SPC is significant as against the network latency [22]. Since the emphasis of this thesis is on the issues of machine independent data decomposition, we do not make any assumption about the detailed machine parameters including the ratio between the software latency and network latency. Therefore, the message vectorization technique is not discussed in this thesis. For the same reason, we do not make any assumption about the computation speed over the communication speed. Hence, the technique of overlapping communication with computation is left to the back-end compilation optimizer after the data decomposition is performed.

Data distribution is responsible for reducing data shift communication and increasing processor workload balance. The amount of data shift communication can be greatly reduced if elements are distributed to processors in *block* fashion. On the other hand, however, limited by the owner-computes rule, the requirement of processor workload balance favors *cyclic* distribution when the workload is not uniformly distributed among all the LHS elements. The conflict between block and cyclic distribution has become an open issue in the research of data distribution. In this thesis, we propose a *segment* distribution which minimizes the impact of data shift movement by allocating elements consecutively to processors and balances the processor workload by varying the size of the segments assigned to different processors. An optimal processor allocation algorithm is given to minimize the overall cost of data shift communication across multiple dimensions of the template array. The segment distribution and optimal processor allocation proposed in this thesis provide the best data distribution support for most data parallel programs.

1.4 Objectives and Scope of Research

The results of our data decomposition analysis are presented based on data parallel Fortran languages which are written in the global name space. The reason we choose

data parallel Fortran languages is that scientists wish to use the SPCs in their familiar dialect of sequential Fortran. The parallelizable loops are the major resources of parallelism available in data parallel Fortran programs. Many sophisticated loop transformation techniques [23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34] have been developed to transform various sequential loops into parallelizable loops in order to explore the inherent parallelism. In this thesis, we assume that the data decomposition analysis takes place after the loop transformation has been performed. Given an application program, our data decomposition analysis neglects those non-parallelizable subprogram structures and only focuses on the parallelizable subprogram structures which can be represented by the HPF **FORALL** structures [14]. We believe that this approach is reasonable and practical due to the following reasons:

- **DOACROSS** loops can be successfully parallelized by pipelining both computation and communication. The corresponding decomposition for arrays referenced in the **DOACROSS** loops is also straightforward: *block* distribution on the partitioned dimension.
- Most of non-parallelizable subprogram structures in data parallel Fortran programs can be formalized into a few intrinsic functions whose efficient implementation is machine dependent and may be resorted to special system software or even hardware support.

We do not address the data decomposition issue involved in the procedure calling because the techniques available in dealing with the interprocedural data dependence test are still at the preliminary stage. Without the support of a mature interprocedural data dependence testing technique, we feel it extremely difficult to achieve quality data decomposition results.

Our data decomposition framework is based on the abstract model of the SPCs. The basic assumption is that interprocessor communication is much more expensive

than local memory access and thus the performance of data parallel programs is dictated by the data locality. Only the number of elements involved in remote data reference is measured as the cost of interprocessor communication. Detailed machine-dependent parameters, such as the network latency, software latency, and switching techniques, have been ignored. In the data distribution analysis, we simply assume that data elements are distributed to the virtual processors which are interconnected through a fully connected network. Though the actual architecture of a SPC is not likely to afford the fully connected network topology, the scalable communication library [35, 36, 37, 38] usually can offset the adverse impact of network topology and achieve efficient data communication. In addition, other machine-dependent optimization issues [39, 40], including run-time support [41], will not be addressed in this thesis.

It should be emphasized that it is not the intention of this thesis to address the compilation techniques [41] involved in generating efficient parallel programs for the SPCs after data decomposition is well-defined. The data partition, instruction partition, and communication generation have been actually performed in order to obtain the performance results of benchmark subprogram structures. However, they are just treated as a part of implementation and will not be further addressed in the rest of this thesis.

1.5 Thesis Outline

The rest of this thesis is organized as follows. In Chapter 2, an overview of data decomposition for data parallel programs is presented. Different subprogram structures are clarified and different types of interprocessor communication are examined. A layer structured data decomposition model is used to illustrate the tasks accomplished in each major phase. Strategies and difficulties are discussed for data re-distribution

and data re-alignment. Major previous work is classified based on the proposed layer structure model. Chapter 3 addresses the base alignment phase. Efficient base alignment algorithms are proposed with regard to various concerns including the optimal expression evaluation and redundant communication avoidance. Chapter 4 addresses the offset alignment phase. Efficient offset alignment algorithms are proposed by considering both the optimal expression evaluation and redundant communication avoidance. Chapter 5 addresses the data distribution phase. An optimal virtual processor allocation strategy is proposed and the closed form of segment distribution is given to maximize the processor workload balance and minimize the neighboring communication. Experimental results of benchmark program structures are shown in each of Chapters 3, 4, and 5. Finally, Chapter 6 summarizes the major contribution of this research and provides directions for future research.

CHAPTER 2

Data Decomposition Overview

This chapter gives an overview of the data decomposition problem. Different subprogram structures are clarified and different types of interprocessor communication are examined. A layer structured data decomposition model is proposed to illustrate each major phase in data decomposition. Strategies and difficulties are discussed for data re-distribution and data re-alignment. Influential previous work has been classified based on the proposed layer structure model.

In this thesis, we follow Fortran 90 array specification. In a one-dimensional array, an array subscript is declared as $0 : n - 1 : s$, where n is the total number of elements and s is the value of stride. An array subscript always starts with zero. The stride option can be omitted if its value is one. For example, $A(0 : n - 1)$ represents $A(0 : n - 1 : 1)$. Consecutive array elements $A(\ell)$, $A(\ell + 1)$, \dots , $A(r)$ ($\ell < r$) can be represented by array segment $A(\ell : r)$. In a multi-dimensional array, the array subscript declared for each dimension is separated by a comma. For example, a 8×10 two-dimensional array A can be represented by $A(0 : 7, 0 : 9)$. $A(1, 2)$ is an element in $A(0 : 7, 0 : 9)$. The notation for array subscript is also applicable to the loop statement declaration.

2.1 Loop Characteristics

Loop is the major resource of parallelism in data parallel Fortran programs [42]. What type of parallelism a loop can explore is determined by the pattern of data dependence among different iterations of the loop.

2.1.1 Parallelizable Loop

A loop is a *parallelizable loop* if a data object defined in an iteration is not used or re-defined in other iterations in the loop [24, 43]. A general form of parallelizable loops can be modeled by the **FORALL** loop in HPF [14]. A **FORALL** loop may consist of one or more **FORALL** assignment statements, or **FORALL assignments** for short. A **FORALL** assignment is implemented by first executing the evaluation of the RHS of the assignment statement for all combination of loop index subscripts, and then assigning to the corresponding elements of the array referenced at the LHS of the assignment statement. A **FORALL** assignment is free of loop-carried data dependence. Semantically, in a **FORALL** loop consisting of several **FORALL** assignments, the execution of the next **FORALL** assignment will not take place until the execution of the previous **FORALL** assignment is fully completed.

In a **FORALL** assignment, the access to a RHS data element owned by a remote processor is achieved by passing a message to the local processor. Since the **FORALL** assignment is free of loop-carried data dependence, messages sent to the local processor can be combined and transferred prior to the loop execution. As a result, in the execution of a **FORALL** loop consisting of several **FORALL** assignments, computation and communication are alternated statement-by-statement. Since it is fully parallelizable, the performance of a **FORALL** loop is only determined by processor workload balance and interprocessor communication overhead. Therefore, the **FORALL** loop becomes one of the most important program structures on which data decomposition

issues are studied. The survey of previous work with regard to the **FORALL** loop can be found in Chapter 2.7.

2.1.2 Perfectly Nested Loops

In many perfectly nested loops with constant distance vectors, the outmost loop cannot be parallelized by any legal loop transformation [29]. This feature makes the coarse-grain parallelism difficult to explore and thus degrades the overall performance on the SPCs due to the large overhead of barrier synchronization at the end of each iteration of the outmost loop. A closer inspection reveals that computation and communication can be pipelined in a perfectly nested loop with constant distance vectors. For example, consider the following four point difference operation.

```

DO  $i_1 = 2, n - 1$ 
  DO  $i_2 = 2, n - 1$ 
 $s_1$ :    $X(i_1, i_2) = (X(i_1 - 1, i_2) + X(i_1 + 1, i_2) + X(i_1, i_2 - 1) + X(i_1, i_2 + 1))/4$ 
      END DO
  END DO

```

Figure 2.1(a) shows the original iteration space in which an arrow represents the direction of data dependence. Figure 2.1(b) illustrates how this loop can be executed in pipeline. The iterations are assigned to different processors in shaded blocks, so are columns of array X . The row segment assigned to each processor is labeled by the lock step in pipeline. As long as the computation on the first row is finished, all processors can start executing in parallel. This technique is also known as *tiling* [44]. The tiled code the corresponds to Figure 2.1(b) is as follows.

Here b is the length of the row segment assigned to each processor. The original i_2 loop is split into two dimensions: the outer i_4 loop and inner i_3 loop. Iterations of


```

DO  $i_4 = 2, n - 1, b$ 
  DO  $i_2 = 2, n - 1$ 
    DO  $i_3 = i_4, \min(n - 1, i_4 + b - 1)$ 
 $s_1$ :       $X(i_1, i_3) = (X(i_1 - 1, i_3) + X(i_1 + 1, i_3) + X(i_1, i_3 - 1) + X(i_1, i_3 + 1))/4$ 
    END DO
  END DO
END DO

```

the i_4 loop are spread across processors, while iterations of the i_2 and i_3 loops reside on the same processor.

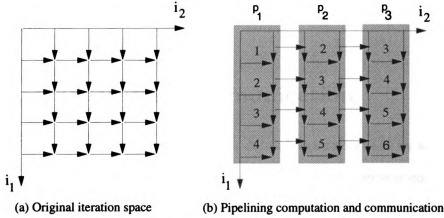


Figure 2.1. Pipeline computation and communication

An important feature of pipelining is that computation and communication can be overlapped [17]. As shown in Figure 2.1(b), processor p_1 can send the result of $X(i_1, b)$ to processor p_2 , while processor p_2 is still working on element $X(i_1 - 1, i_3)$ where $b \leq i_3 \leq 2b$. The pipelining technique can also be applied to the perfectly nested loops with irregular and complex dependence constraints using the dependence uniformization methods [45].

2.1.3 Special-Purpose Loops

Special parallel algorithms may be required to parallelize those special-purpose operations, such as prefix, suffix, gather, scatter, and other combining operations. Generally speaking, these algorithms are machine-dependent or architecture-dependent. As a result, they are typically supported as library routines, such as *intrinsic* procedures in HPF, and usually not further optimized at compilation time. An important issue which the compilation optimizer should deal with is how to efficiently utilize these library routines, in particular, when library routines are designed for various data decomposition patterns. Issues in designing scalable library routines for MPCs can be found in [46].

2.2 Interprocessor Communication

Interprocessor communication occurs when a data object referenced by a local processor resides in a remote memory. As mentioned in [47], interprocessor communication generated in executing data parallel Fortran programs can be classified into *intrinsic* communication and *residual* communication. Intrinsic communication arises from those special-purpose computational operations such as scatter and gather that require data motion as an integral part of the operation. As addressed in the previous section, minimizing intrinsic communication is a major task of efficient intrinsic library implementation [48, 35, 36, 37, 49, 38, 50, 51] and thus is beyond the scope of compiler optimization.

Residual communication arises from nonlocal data references required in a computation whose operands are not aligned with respect to each other. Typically, residual communication can be further separated into *neighboring* communication and *reorganization* communication. Neighboring communication refers to the nearest-neighbor shifts of data. Reorganization communication is due to the mismatch in data de-

composition, which requires reorganizing the entire data structure. Not all residual communication patterns are equally expensive in SPCs. Neighboring communication can be significantly reduced by distributing data arrays in blocks. On the other hand, however, reorganization communication is often much more expensive because the data movement pattern is unstructured, such as transpose, change of stride, and vector-valued subscripts.

Reducing interprocessor communication by properly allocating data objects into processors is a key issue in the data decomposition analysis.

2.3 Processor Workload Balance

The main reason to employ SPCs is to reduce the overall execution time. In most data parallel scientific programs, parallelizable loops are the major resources of the parallelism. For concurrent execution of a parallelizable loop, the iterations have to be assigned to processors. This is also known as workload distribution. Ideally, given the fixed amount of the overall workload, the workload should be evenly distributed to each processor. Therefore, the overall execution time regarding to the whole system can be minimized. Otherwise, the overall execution time would be longer if there is a processor idle when other processors are still working. Note that a barrier synchronization is typically required between two adjacent subprogram phases. As a result, if it finishes its own work earlier than others, a processor has to be idle rather than start working on the next subprogram phase.

The previous study in the processor workload balance has been focused on the run-time scheduling on shared-memory multiprocessor systems [52, 53, 54, 55, 56, 57, 58, 59]. In data parallel programs, the workload is distributed in the way that a data object is written only by the local processor which owns it. Therefore, the workload distribution depends on the pattern of data decomposition which is determined at

compilation-time. In this thesis, we study the static processor workload balance, another key issue in the data decomposition analysis. It has been shown [60] that static processor workload balance is critical to the overall performance even with the support of the runtime scheduling.

2.4 The Model for Data Decomposition

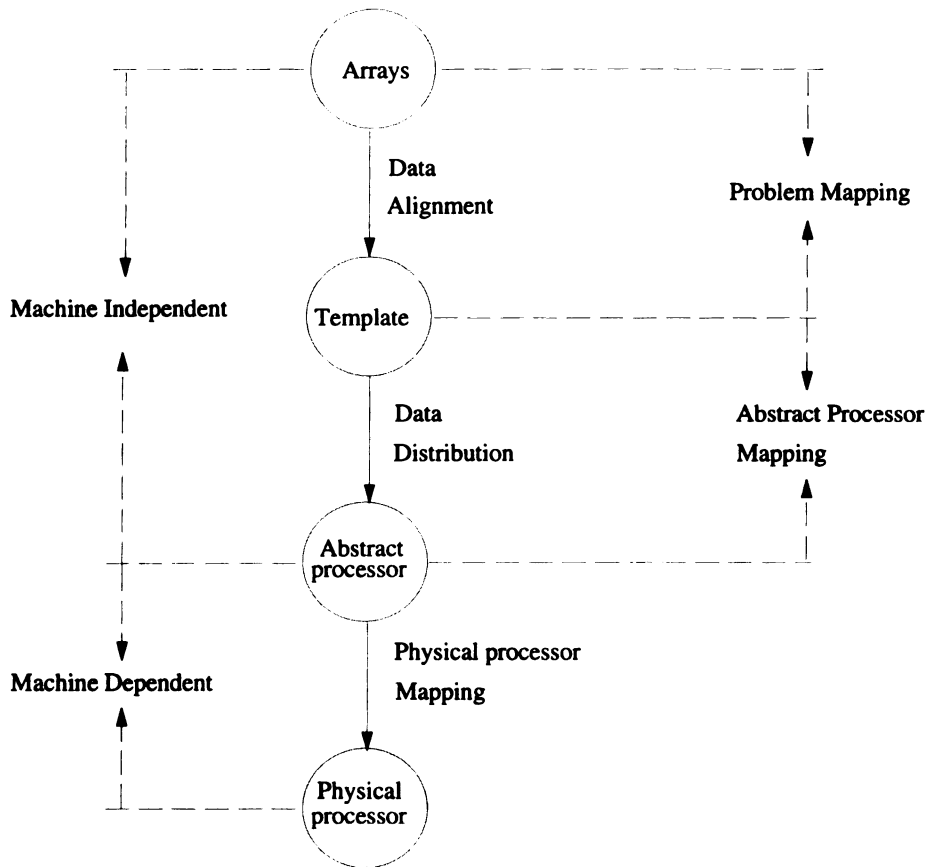


Figure 2.2. The model for data decomposition

As illustrated in Figure 2.2, the data decomposition model can be viewed as a two-level mapping of array elements to abstract processors. Array elements are first aligned

relative to one another, known as *data alignment*; the group of aligned arrays is then distributed onto a set of virtual processors, known as *data distribution*. Data alignment, the problem mapping, represents the requirements of reducing data movement induced by the structure of the underlying computation. Data alignment is achieved by aligning array elements to an abstract index space, known as *template* in HPF [14], which is typically represented by a rectilinear space. Data distribution, the virtual-machine mapping, represents the requirement of efficiently allocating computation structures to finite machine resources. Data distribution is attained by allocating template elements onto the rectilinear arrangement of virtual processors. As a result, all array elements which are aligned to the same template element are allocated to the processor to which that particular template element is allocated. However, the data alignment and distribution phases are machine-independent. The final step in which the rectilinear arrangement of virtual processors are mapped to the physical processors is machine-dependent. The issues in physical processor mapping are beyond the scope of this thesis. For simplicity, in the rest of this thesis a processor refers to a virtual processor rather than a physical processor.

2.4.1 Formulation for Data Alignment and Distribution

A mathematical model is presented for data alignment and data distribution. An array of dimension m defines an array space \mathcal{A} , an m -dimensional rectangle. Each element in the array is accessed by an integer vector $\vec{a} = (a_1, a_2, \dots, a_m)$. For the same reason, a template of dimension h defines an array space \mathcal{T} , an h -dimensional rectangle. Each element in the template is accessed by an integer vector $\vec{t} = (t_1, t_2, \dots, t_h)$.

Definition 2.1 *For each index \vec{a} of an m -dimensional array, the data alignment of the array onto an h -dimensional template is a function $\delta_A(\vec{a}) : \mathcal{A} \rightarrow \mathcal{T}$, where*

$$\delta_A(\vec{a}) = D_A \vec{a} + \vec{d}_A$$

D_A is an $h \times m$ linear transformation matrix and \vec{d}_A is a constant vector.

Definition 2.2 D_A is called alignment matrix of array A . \vec{d}_A is called alignment offset of array A .

The rectilinear arrangement of abstract processors is represented by a processor array. The dimension of the processor should be the same as the dimension of the template. The processor array defines an array space \mathcal{P} , an h -dimensional rectangle. Each element in the processor array is accessed by an integer vector $\vec{p} = (p_1, p_2, \dots, p_h)$.

Definition 2.3 For each index \vec{t} of an h -dimensional template, the data distribution of the array onto an h -dimensional processor array is a function $\gamma_T(\vec{t}) : \mathcal{T} \rightarrow \mathcal{P}$.

The function γ_T may not necessarily be an affine function and varies with different applications.

2.4.2 Layered Structure for Data Decomposition

A typical data decomposition process can be described by a three-phase layered structure shown in Figure 2.3. The *base alignment* phase determines alignment matrix D_A in alignment function δ_A for each array A . The *offset alignment* phase specifies alignment offset \vec{d}_A in alignment function δ_A for each array A . The base alignment and offset alignment phases accomplish the task of data alignment by mapping array elements to template elements. The *distribution* phase decides in what fashion the template elements are distributed to the processors. All array elements which are aligned to the same template element are allocated to the same processor to which that particular template element is allocated. As a result, taking the bridge of the template, the distribution phase fulfills the requirement of data distribution.

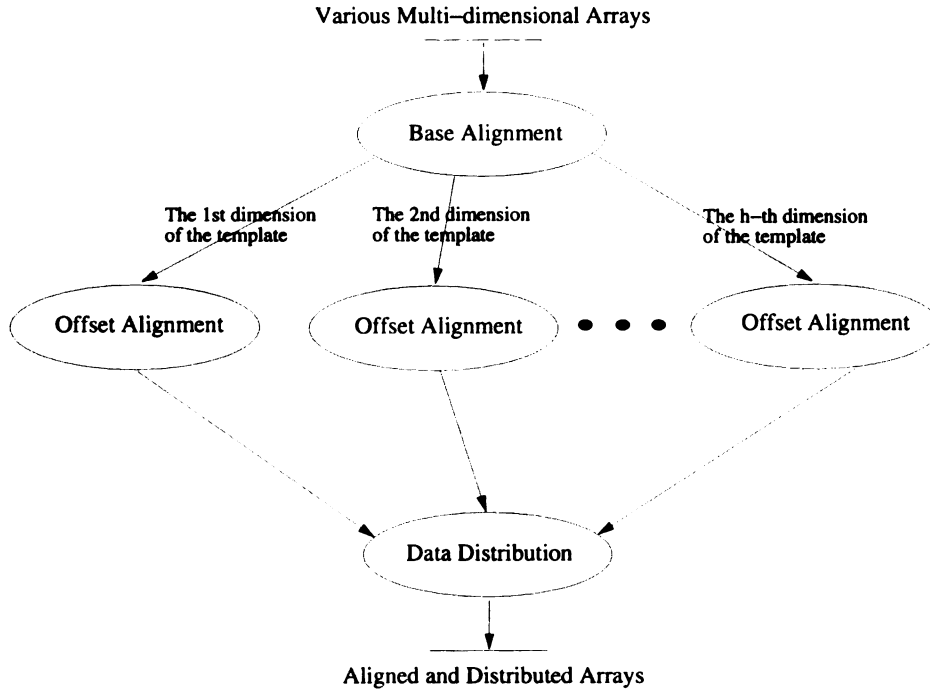


Figure 2.3. Layered structure for data decomposition process

2.4.3 The Base Alignment Phase

A multi-dimensional array can be treated as a multi-dimensional bounded linear space, called *data space*, in which each integer grid represents an array element. A multi-dimensional array can be partitioned in a family of parallel hyperplanes such that array elements on the same hyperplanes are always allocated to the same processor. As a result, the potential overhead of interprocessor communication involved in the mutual references between the elements in the same hyperplane can be fully avoided.

Consider the NAS benchmark loop in Example 1 (Figure 2.4). Array elements $Y(i_1, i_2)$ and $Y(i_2, i_1)$ referenced in **FORALL** assignment s_1 are along the diagonal. Therefore, **FORALL** assignment s_1 is free of interprocessor communication if array Y is partitioned in a family of off-diagonals, as shown in Figure 2.5(b). In Figure 2.5(b),


```

FORALL( $i_1 = 0 : n - 1, i_2 = 0 : n - 1 - i_1$ )
s1:    $Y(i_1, i_2) = Y(i_2, i_1)$ 
s2:    $X(i_1, i_1 + i_2) = Y(i_1, i_2)$ 
END FORALL

```

Figure 2.4. Example 1: A NAS benchmark loop

y_1 is the index of the column dimension and y_2 is the index of the row dimension of array Y . T represents the one-dimensional template array. The size of array Y is declared as 8×8 and the size of T is 8. Each array element is represented by a circle. Elements on each off-diagonal (represented by a solid line) are collapsed and mapped (represented by each dash line) to the same element in the template. Some array elements are not covered by any solid line since they are out of the loop boundary.

Vector $(1, -1)$, the direction vector of an off-diagonal, is called the *collapse base* of array Y . Two Y elements must be collapsed and mapped to the same element in the template if the vector starting with one element and ending with another is parallel to collapse base $(1, -1)$. Vector $(1, 1)$, which is orthogonal to the collapse base $(1, -1)$, is called the *distribution base* of Y . Two Y elements must be mapped to the different elements in the template and thus may be distributed to different processors if the inner-product of the distribution base and the vector constructed by these two Y elements is non-zero. Alignment matrix is constructed by distribution bases. In this example, since there is only one distribution base $(1, 1)$, $D_Y = (1, 1)$. Thus, alignment function δ_Y can be specified as follows.

$$t = \delta_Y \left(\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \right) = D_Y \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = (1, 1) \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = y_1 + y_2$$

where $Y(y_1, y_2)$ is a Y element and $T(t)$ is a template element. Base alignment only

determines the value of alignment matrices. The value of alignment offsets is decided in the offset alignment phase. In the above δ_X and δ_Y , we simply assume that $d_X = 0$ and $d_Y = 0$.

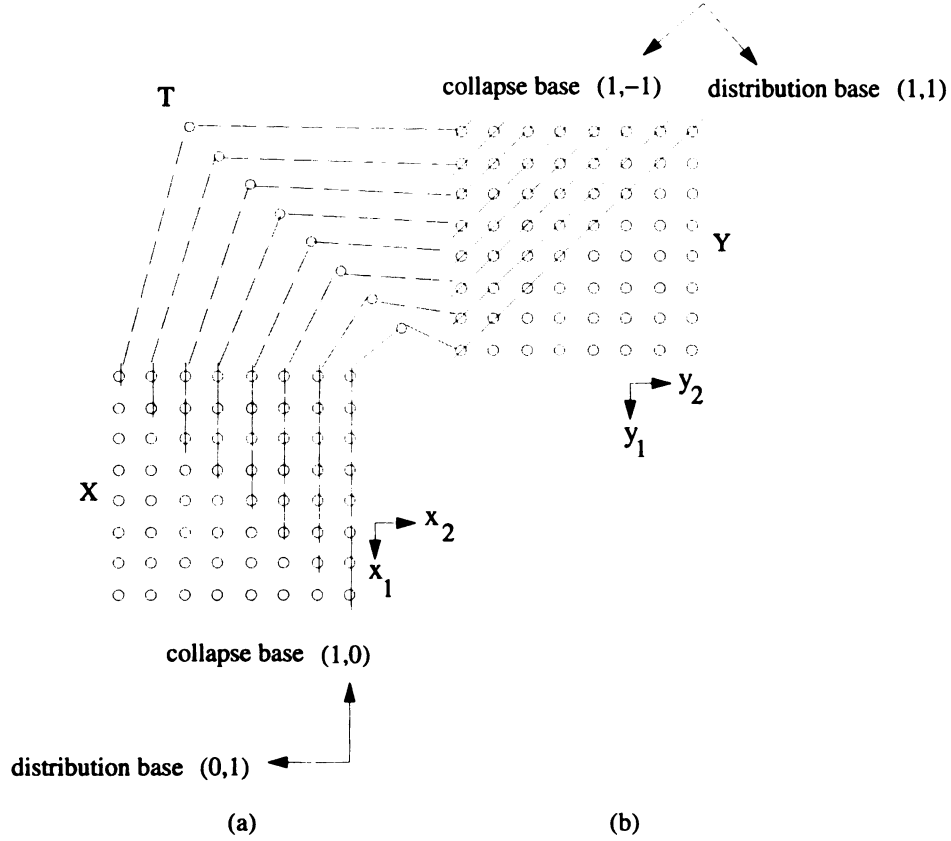


Figure 2.5. Base alignment in Example 1

Generally speaking, the vector represented by each row in an alignment matrix indicates a distribution base. As a result, the rows in the same alignment matrix must be mutually linear independent. On the other hand, the vector which is orthogonal to all rows in a alignment matrix implies that that vector is a collapse base. The distinct collapse bases must also be mutually linear independent. All collapse bases and distribution bases should span the entire data space with regard to each array.

Consider the array subscript structure in **FORALL** assignment s_2 (Figure 2.4). Since Y elements are collapsed along the off-diagonal, X elements have to be collapsed in columns so as to make **FORALL** assignment s_2 free of interprocessor communication. Therefore, the collapse base of array X is equal to $(1,0)$, the direction vector of a column. The distribution base of array X is equal to $(0,1)$, which is orthogonal to collapse base $(1,0)$. Therefore, we have $D_X = (0,1)$. Thus, alignment function δ_X can be specified as follows.

$$t = \delta_X \left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \right) = D_X \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = (0,1) \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = x_2$$

where $X(x_1, x_2)$ is an X element and $T(t)$ is a template element. Figure 2.5(a) shows the alignment of array X with regards to the template. In Figure 2.5(a), x_1 is the index of the column dimension and x_2 is the index of the row dimension of array X . The size of array X is declared as 8×8 . X elements in the same column (represented by each solid line) are collapsed and mapped (represented by each dash line) to the same element in the template. Some elements are not covered by any solid line since they are out of loop boundary.

Note that the X partition in columns and Y partition in off-diagonals are not randomized. They are in fact inducted from the requirement of minimizing interprocessor communication based on the given array subscript structures in Example 1. Such a relationship between the alignment of X and the alignment of Y is called *base alignment*. Base alignment represents the alignment relationship between various distribution bases of various arrays and the alignment relationship between various collapse bases of various arrays.

Depending on the pattern of how array elements are mapped to the template, an array may have more than one distribution base and/or more than one collapse base. Consider the Purdue benchmark loop in Example 2 (Figure 2.6). Unlike Example


```

FORALL( $i_1 = 0 : n_1 - 1, i_2 = 0 : n_2 - 1$ )
 $s_1:$     $W(i_1, i_2) = Z(i_2, i_1)$ 
END FORALL

```

Figure 2.6. Example 2: A Purdue benchmark loop

1, in Example 2 there is no self-reference regarding to arrays W and Z . Both W and Z can be partitioned in both row and column dimensions. Thus, the collapse base of both W and Z is degenerated to $(0,0)$. Both W and Z have distribution bases $(1,0)$ and $(0,1)$, which span the entire data space. This implies that there is no requirement that two particular elements in the same array should be mapped to the same template element.

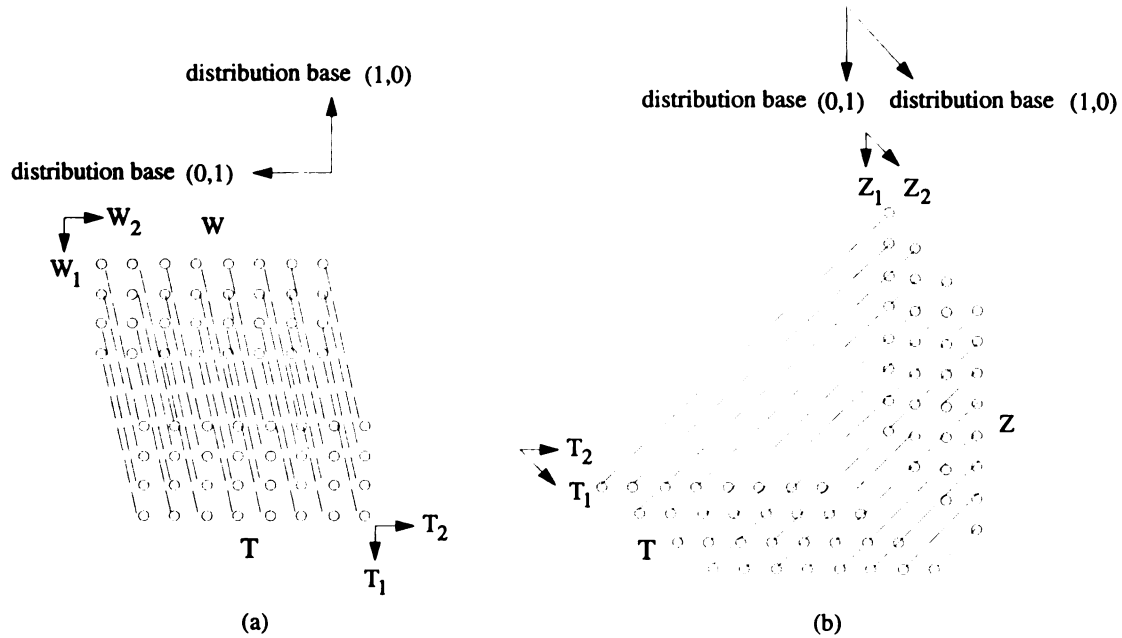


Figure 2.7. Base alignment in Example 2

Figure 2.7 shows the alignment of W and Z . In Figure 2.7, the size of W is declared as 4×8 and the size of Z is declared as 8×4 . Template T is declared as a two-dimensional array which size is 4×8 . w_1 , z_1 , and t_1 represent the column dimension of arrays W , Z , and T , respectively. w_2 , z_2 , and t_2 represent the row dimension of arrays W , Z , and T , respectively. Figure 2.7(a) shows the alignment of array W : dimension w_1 is aligned with dimension t_1 and dimension w_2 is aligned with dimension t_2 . Therefore, $D_W = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ and alignment function δ_W is specified as follows.

$$\begin{pmatrix} t_1 \\ t_2 \end{pmatrix} = \delta_W \left(\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} \right) = D_W \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$$

where $W(w_1, w_2)$ is a W element and $T(t_1, t_2)$ is a template element. Figure 2.7(b) shows the alignment of array Z : dimension z_2 is aligned with dimension t_1 and dimension z_1 is aligned with dimension t_2 . Therefore, $D_Z = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ and alignment function δ_Z is specified as follows.

$$\begin{pmatrix} t_1 \\ t_2 \end{pmatrix} = \delta_Z \left(\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} \right) = D_Z \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} z_2 \\ z_1 \end{pmatrix}$$

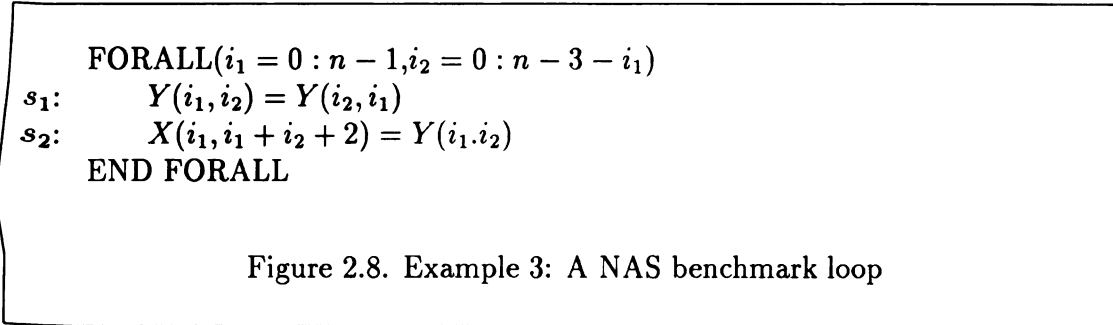
where $Z(z_1, z_2)$ is a Z element and $T(t_1, t_2)$ is a template element.

For arrays each of which has multiple (linear independent) distribution bases, it is crucial that which distribution base of one array should be aligned with which distribution base of the other. In Example 2 (Figure 2.6), in order to make **FORALL** assignment s_1 free of interprocessor communication, distribution base $(1, 0)$ (the column dimension) of W is aligned with distribution base $(0, 1)$ (the row dimension) of Z , and distribution base $(0, 1)$ (the row dimension) of W is aligned with distribution

base $(1, 0)$ (the column dimension) of Z . Aligned distribution bases in various arrays comprise an *aligned-base group*. Example 2 has two distinct aligned-base groups. One consists of distribution base $(0, 1)$ of W and distribution base $(1, 0)$ of Z . The other consists of distribution base $(1, 0)$ of W and distribution base $(0, 1)$ of Z . By the definition of the alignment function, each aligned-base group can be uniquely identified by a dimension in the template array. Different distribution bases of the same array can never be aligned with each other and never be included in the same aligned-base group.

2.4.4 The Offset Alignment Phase

As described in the previous section, an array can be partitioned in a family of parallel hyperlanes in each of which the array elements are collapsed and mapped to the same element in the template. However, base alignment only determines the constituent base(s), known as distribution base(s) and collapse base(s), for such a family of parallel hyperlanes. Offset alignment is responsible for the displacement of each hyperplane with regard to the template. In other words, while base alignment determines D_A , offset alignment determines \vec{d}_A in alignment function δ_A for each array A .



Consider the NAS benchmark loop in Example 3 (Figure 2.8). Except the access

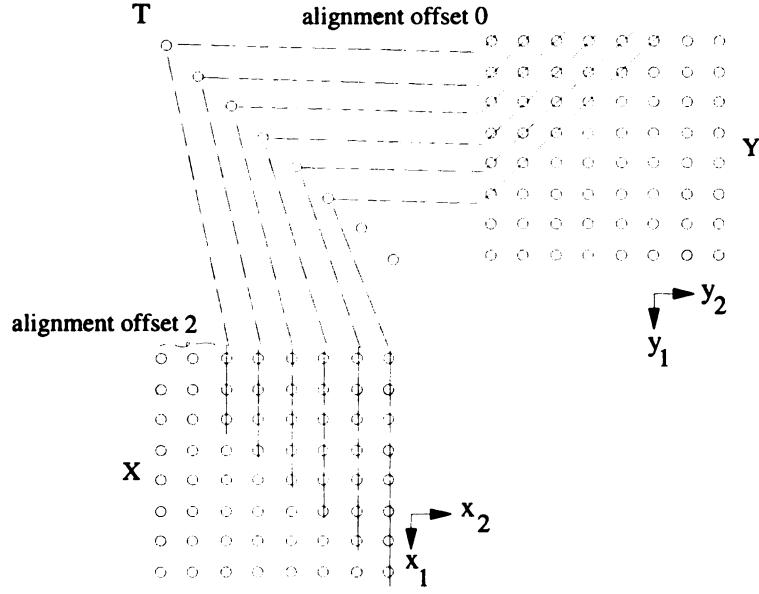


Figure 2.9. Offset alignment in Example 3

offset 2 in the array subscript $X(i_1, i_1 + i_2 + 2)$ (Figure 2.8), Example 3 is similar to Example 1. For this reason, the base alignment analysis in Example 3 is identical to that in Example 1. $D_X = (0, 1)$ and $D_Y = (1, 1)$. Figure 2.9 shows the alignment for arrays X and Y in Example 3. In Figure 2.9, Y is partitioned in off-diagonals and X is partitioned in columns. Unlike Example 1 (Figure 2.5), however, in Example 3 (Figure 2.8), the $(k + 2)$ -th column in X should be aligned with the k -th off-diagonal in Y , in order to make **FORALL** assignment s_2 free of interprocessor communication. Such an alignment relationship between the column displacement in X and the off-diagonal displacement in Y is called *offset alignment*. In Example 3, we have $d_X = 2$ and $d_Y = 0$. Alignment function δ_X can be written as follows:

$$t = \delta_X \left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \right) = D_X \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + d_X = x_2 + 2$$

where $X(x_1, x_2)$ is an X element and $T(t)$ is a template element. Alignment function

δ_Y can be written as follows:

$$t = \delta_Y \left(\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \right) = D_Y \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} + d_Y = y_1 + y_2$$

where $Y(y_1, y_2)$ is a Y element and $T(t)$ is a template element.

Generally speaking, the value of alignment offset is represented by a constant vector. Offset alignment should determine each component in such an alignment offset vector. In Chapter 4, we will show that the components in the alignment offset vector are independent with one another. Each component can be decided based on a distinct aligned-base group.

2.4.5 The Data Distribution Phase

The data distribution phase determines how to map the template elements to virtual processors. Since both the template and the virtual processor arrangement are represented by multi-dimensional arrays, there are two major decisions to make regarding to each dimension in the template: what is the distribution type and how many virtual processors should be allocated.

We first consider the case in which both the template array and the processor array are one-dimensional. There are numerous ways to distribute template elements across processors. Among them the *cyclic* distribution and *block* distribution are most popular. In cyclic distribution, template elements are assigned to processors in the round-robin fashion. In block distribution, template elements are contiguously allocated to each processor. Figure 2.10 shows the data distribution phase of Example 3. In Figure 2.10, arrays X and Y are declared as 8×8 . There are total two processors available, denoted as $P(0)$ and $P(1)$. The template array is distributed in cyclic. As a result, processor $P(0)$ owns the even-numbered X columns and the even-numbered

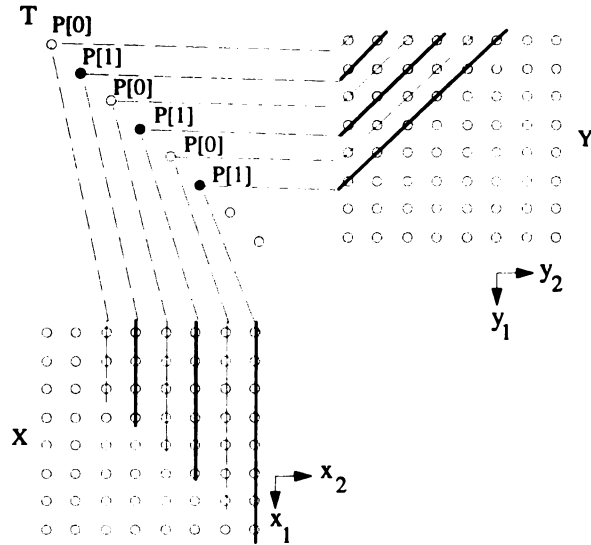


Figure 2.10. Data distribution in Example 3

Y off-diagonals. Processor $P(1)$ owns the odd-numbered X columns and the odd-numbered Y off-diagonals. The data distribution function, γ_T , can be written as follows:

$$p = \gamma_T(t) = t \bmod 2$$

where $P(p)$ is an element in the processor array and $T(t)$ is an element in the template array. Note that the number of the LHS array elements mapped to each template element is varied. For example, two Y elements are mapped to $T(1)$ but five Y elements are mapped to $T(4)$. Since workload assigned to each processor depends on the number of the LHS elements owned by that processor, the workload imposed by each template element can be varied. For this reason, the cyclic distribution increases the processor workload balance. Another reason of using cyclic distribution is that interprocessor communication has been fully eliminated by the alignment function. Otherwise, significant communication cost could occur if cyclic distribution is employed.

The template array and processor array can be multi-dimensional. The task of processor allocation is to determine the layout of the multi-dimensional processor array.

```

FORALL( $i_1 = 1 : n_1 - 3, i_2 = 1 : n_2 - 3$ )
 $s_1:$     $W(i_1, i_2) = Z(i_1, i_2) + Z(i_1 + 1, i_2) + Z(i_1 - 1, i_2)$ 
         $+ Z(i_1, i_2 + 1) + Z(i_1, i_2 - 1)$ 
 $s_2:$     $Z(i_1, i_2) = W(i_1, i_2)$ 
END FORALL

```

Figure 2.11. Example 4: A Heatwave benchmark loop

Consider the Heatwave benchmark loop in Example 4 (Figure 2.6). By the construction of array subscripts in **FORALL** assignment s_2 , there is no interprocessor communication as long as $W(i_1, i_2)$ and $Z(i_1, i_2)$ are mapped to the same template element. On the other hand, however, since both $Z(i_1 + 1, i_2)$ and $Z(i_1, i_2)$ are referenced, assignment s_1 is free of communication only if Z elements are collapsed in the row dimension. Similarly, since both $Z(i_1, i_2 + 1)$ and $Z(i_1, i_2)$ are referenced, assignment s_1 is free of communication only if Z elements are collapsed in the column dimension. This conflict implies that interprocessor communication in executing assignment s_1 cannot be avoided no matter how W and Z are aligned. For this reason, alignment functions δ_W and δ_Z are simply defined as follows:

$$\begin{pmatrix} t_1 \\ t_2 \end{pmatrix} = \delta_W \left(\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} \right) = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$$

and

$$\begin{pmatrix} t_1 \\ t_2 \end{pmatrix} = \delta_Z \left(\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} \right) = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix}$$

where $W(w_1, w_2)$ is a W element, $Z(z_1, z_2)$ is a Z element, and $T(t_1, t_2)$ is a template element.

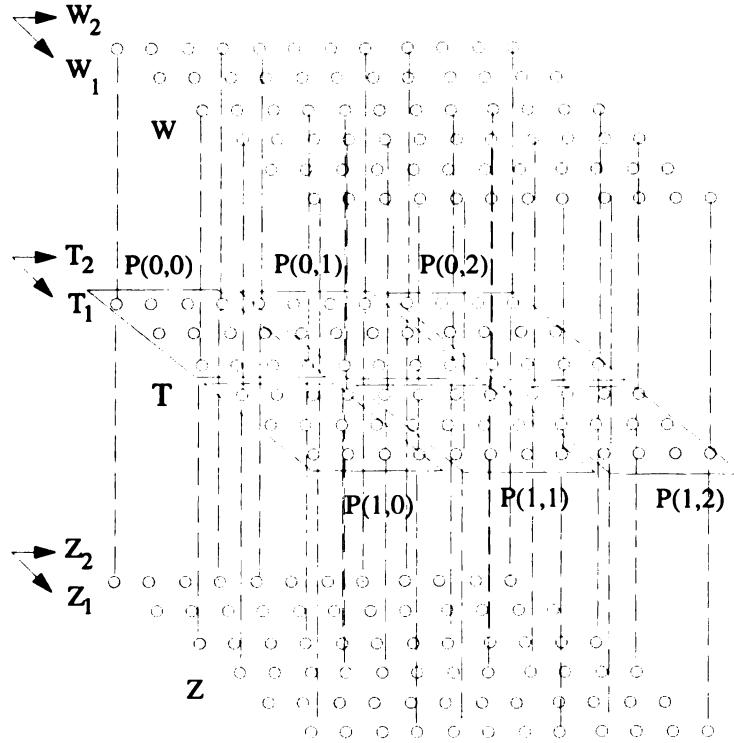


Figure 2.12. Processor allocation in Example 4

Figure 2.12 shows a pattern of processor allocation in Example 4. In Figure 2.12, arrays W , Z , and T (the template array) are declared as 6×12 . There are total 6 processors available. The processor array is specified as 2×3 , denoted as $P(0 : 1, 0 : 2)$. $P(0 : 1, 0 : 2)$ indicates there are 2 processors assigned to row dimension and 3 processors assigned to column dimension. Template elements are distributed to processors in *block* fashion. The mapped data blocks in arrays W and Z are indicated

by the dash lines. The data distribution function, γ_T , can be written as follows:

$$\begin{pmatrix} p_1 \\ p_2 \end{pmatrix} = \gamma_T\left(\begin{pmatrix} t_1 \\ t_2 \end{pmatrix}\right) = \begin{cases} \lfloor \frac{t_1}{2} \rfloor \\ \lfloor \frac{t_2}{3} \rfloor \end{cases}$$

where $P(p_1, p_2)$ is an element in the processor array and $T(t_1, t_2)$ is an element in the template array. Note that the 2×3 processor allocation minimizes the overall cost of interprocessor communication involved in executing assignment s_1 . The theory of the processor allocation optimization will be addressed in Chapter 5.

2.5 Data Re-distribution and Data Re-Alignment

It is not necessary to keep the same layout of data decomposition in the life time of a data parallel program. Data objects can be re-aligned and re-distributed from one subprogram phase to the other in order to meet the requirement of minimizing interprocessor communication and maximizing processor workload balance imposed by different computing structures. However, few researchers have addressed these issues due to the lack of good mathematical model(s) to represent data re-alignment and data re-distribution. The major difficulty that data re-distribution and re-alignment are encountering is the lack of the knowledge of the subprogram breaking point. A straight forward approach is to treat each array assignment statement as a building block on which data re-distribution and data re-alignment is considered. A dynamic programming algorithm based on this approach is proposed [61] to resolve the data re-distribution and data re-alignment. We briefly review its basic idea as follows.

Let s_1, s_2, \dots, s_k be k **FORALL** assignments in sequence in the program. Let $M_{i,j}$ be the cost of computing in sequence of loops s_i, s_{i+1}, \dots , and s_{i+j-1} using the component-alignment algorithm [62], and $D_{i,j}$ be the distribution scheme, for $1 \leq i \leq k$ and $1 \leq j \leq k - i + 1$. Define $C_{i,j}$ to be the minimum cost of interprocessor

communication involved in computing the sequence of loops s_i, s_{i+1}, \dots , and s_{i+j-1} with the restriction that the final data distribution scheme after computing $C_{i,j}$ is $D_{i,j}$.

A dynamic programming algorithm for computing the minimum cost of data distribution schema of executing a sequence of k **FORALL** assignments is formalized in Figure 2.13. In Figure 2.13, $cost(D_{i-\ell,\ell}, D_{i,j})$ returns the communication cost of changing data distribution layouts from $D_{i-\ell,\ell}$ to $D_{i,j}$.

Input: $M_{i,j}$ and $D_{i,j}$, where $1 \leq i \leq k$ and $1 \leq j \leq k - i + 1$
Output: The minimum-cost data distribution of executing k **FORALL** assignments

```

(1) for  $i = 2$  to  $k$  do
(2)   for  $j = 1$  to  $s - i + 1$  do
(3)      $C_{i,j} = \min_{1 \leq \ell \leq k} \{C_{i-\ell,\ell} + M_{i,j} + cost(D_{i-\ell,\ell}, D_{i,j})\}$ 
(4)   endfor
(5) endfor
(6)  $MinimumCost = \min_{1 \leq \ell \leq k} \{C_{k-\ell+1,\ell}\}$ 

```

Figure 2.13. A dynamic programming algorithm for data re-alignment and data re-distribution

However, the above dynamic programming algorithm has the following drawbacks:

- 1 The component-alignment algorithm cannot give the accurate cost of interprocessor communication, in particular, the cost of data shift movement generated by mismatched alignment offset.
- 2 In the component-alignment algorithm, only a dimension can be chosen as a distribution base. In other words, the partition of an array is dimension-based.
- 3 The dynamic programming algorithm only emphasizes data re-distribution. In fact, in many data parallel programs, the distribution layout of the template

may not be changed, while the alignment function is likely to change during the different subprogram phases.

- 4 The assumption of arranging k **FORALL** assignments in the sequential order is too limited. Depending on the program control flow, the relationship among those **FORALL** assignments can form much more complicated graph, such as a directed acyclic graph (DAG).
- 5 The algorithm is time-consuming by taking each **FORALL** assignment as the building block of the dynamic programming when the program has tens of thousands lines.

The first two issues have been resolved by the results obtained in this thesis. In addition, this thesis has done an in-depth study of the data alignment analysis and provided the accurate cost of interprocessor communication imposed by mismatched alignment. This basically answers the questions raised in the third issue. The forth issue can be resolved by extending the dynamic programming algorithm to the situation where the control dependence of adjacent subprograms constitutes a DAG. The fifth issue is still an open problem. However, the concept of single assignment block proposed in this thesis has made a great effort in finding the bigger building block for the dynamic programming algorithm. For the above reasons, we do not address the issues in data re-distribution and data re-alignment in the rest of this thesis.

2.6 Data Flow Analysis

Communication overhead can significantly affect the performance of executing data parallel programs on the SPCs. The dataflow analysis has been proposed to assist communication optimizations [63, 64, 65]. This section briefly reviews the basic ideas used in the dataflow analysis for optimizing communication [64].

For each remote reference requiring communication, it is fundamental to determine all points in the program at which the communication can be performed. This information can be computed by identifying the communication associated with each statement and propagating it in the backward direction along all execution paths in the program. An array reference nested within a loop may reference different remote array elements during each loop iteration. The group of remote elements residing on the same remote processor are represented as a single entity. Groups of array elements that must be communicated are propagated. If definitions of array elements are encountered, the propagation of those elements is discontinued. A reference to a group of remote elements may have to be split when a definition which only defines a subset of elements in the group is encountered. Moreover, repeated communication of the same remote elements may be encountered during the back paths. In this case, the repeated references can be combined and the redundant communication can be avoided.

The extra cost of analyzing dataflow information for array reference is the set representation and set operation for array elements, in particular, the set union and the set intersection. For regular data distribution pattern, this extra cost is reasonable. The detailed framework of such dataflow analysis can be found in [64]. This thesis will use the results of existing array dataflow analysis to assist data alignment but not address the issues in dataflow analysis.

2.7 Related Work

This section briefly summarizes influential previous work in the research area of data decomposition. Table 2.1 compares the existing work in the area of base alignment. The RHS expression evaluation optimization indicates that part or whole RHS expression evaluation can be executed on the remote processor which does not own the LHS

operand. Table 2.2 compares the existing work in the area of offset alignment. The piecewise linear function accurately models the cost of data shift movement regarding to multiple instances of the same array variable referenced in the same statement. Table 2.3 compares the existing work in the area of data distribution. The summary of each major related work is given in the rest of this section. Note that the scope of most existing data decomposition work, including this thesis, is limited to the scope of **FORALL** structure.

Table 2.1. Related work in base alignment analysis

Methods	Base Alignment Analysis		
	Dimension partition	Hyperlane partition	RHS expression evaluation optimization
HPF <i>et al.</i>	No	No	No
Li and Chen	Yes	No	No
Knob <i>et al.</i>	Yes	No	No
Gupta <i>et al.</i>	Yes	No	No
Chatterjee <i>et al.</i>	Yes	No	No
Anderson and Lam	Yes	Yes	No
This thesis	Yes	Yes	Yes

Table 2.2. Related work in offset alignment analysis

Methods	Offset Alignment Analysis		
	Using access offset	Using piecewise linear function	RHS expression evaluation optimization
HPF <i>et al.</i>	No	No	No
Li and Chen	No	No	No
Knob <i>et al.</i>	No	No	No
Gupta <i>et al.</i>	No	No	No
Anderson and Lam	No	No	No
Chatterjee <i>et al.</i>	Yes	No	No
This thesis	Yes	Yes	Yes

Table 2.3. Related work in data distribution analysis

Methods	Data Distribution Analysis		
	Reducing communication	Increasing load balance	Optimizing processor allocation
HPF <i>et al.</i>	No	No	No
Li and Chen	No	No	No
Knob <i>et al.</i>	No	No	No
Anderson and Lam	No	No	No
Chatterjee <i>et al.</i>	No	No	No
Gupta <i>et al.</i>	Yes	No	No
This thesis	Yes	Yes	Yes

2.7.1 Data Parallel Languages

In order to provide high-level language support for data-parallel programming, several data-parallel Fortran extensions have been proposed, such as Fortran D [66] and Vienna Fortran [67]. In an effort to standardize data parallel Fortran program-

ming, HPF (High Performance Fortran) is being proposed as a standard by the High Performance Fortran Forum led by Rice University [14] for distributed-memory machines. An essential part of these data parallel Fortran extensions is the specification, through compiler directives, of the distribution and alignment of data arrays. The languages, however, do not provide the programmer any guidance in selecting the data decomposition. Programmers are fully responsible for choosing an efficient data decomposition.

2.7.2 Preference Graph Model

One representation that has been frequently used in alignment analysis is the *preference* graph. The preference graph is a undirected, weighted graph. The nodes are constructed by dimensions of various arrays. The edges are constructed from data references in the source program. Edges represent the alignment preferences among various array dimensions. There are two principal variants of this general framework.

Knob, Lukas, and Steel [68] use the preference graph model to address alignment issues in the SIMD (Single Instruction Multiple Data) mode of computation. In their approach, the concept of virtual processor space is adopted in which each array element is mapped to a distinct virtual processor grid. The mapping function can be represented by $ai + b$ where *axis alignment* determines the dimension i , *stride alignment* determines the value of a , and *offset alignment* determines the value of b . Note that axis alignment and stride alignment are two special cases of base alignment. In their preference graph model, the weight on each edge is determined by the loop nesting depth of the reference the edge represents. Alignment is determined in the way that a preference edge with higher weight is honored by constructing a maximum-weight spanning tree.

Li and Chen [62] use the preference graph model, known as *component affinity* graph, to address the axis alignment problem. In their approach, the weight on each

edge is represented by either ϵ , or 1, or ∞ , which depends on the type of the reference the edge represents. The purpose of the component affinity algorithm is to partition the nodes into different components such that the total weight of edges, which incident nodes are in different components, is minimized. They have proved such a component affinity problem is NP-complete.

The limitation of the preference graph model is that arrays can only be partitioned based on each dimension. For instance, a 2D array can be partitioned only in row or column, but not diagonal. This limitation prevents the compilation optimizer from exploiting inherent data locality in the source program.

2.7.3 Using Communication Cost Estimation

The preference graph model has been extended by other researchers [69]. Among them, Gupta and Banerjee [70] use the cost of underlying communication primitives to estimate the penalty if an alignment preference is not honored. This approach is known as the *constraint-based approach*. Li and Chen [71] first present the idea of using a set of low level communication primitives to best match the communication requirements generated in data parallel programs. The cost of such low level communication primitives can also be evaluated by given a particular system architecture. In the constraint-based approach, edge weight is assigned by the communication penalty cost when such an alignment preference is not honored.

However, the penalty cost is a function of alignment. Different values of penalty cost can be obtained by different alignment results. The constraint-based approach has difficulties in representing and using such dynamic cost evaluation.

2.7.4 Dynamic Programming Methods

Chatterjee, Gilbert, Schreiber, and Teng [19, 72, 47, 73, 74] propose a dynamic programming method to solve axis alignment, stride alignment, and offset alignment for a basic block. A DAG representation is used to model the basic block. Their approach first fixes the alignment position of each input array represented by a leaf node and then uses dynamic programming method to find the optimal alignment for each intermediate array represented by an internal node. A set of distance functions is incorporated to characterize different architecture topologies. Their approach does find an optimal alignment when the common subexpression is not contained in a DAG, in other words, the DAG is a tree. Otherwise, the cost function defined in their approach is only an approximation to the real communication cost. They have proved that finding a minimum-cost offset alignment using their definition of cost function is NP-complete.

One major limitation in their approach is that the computation complexity of the dynamic algorithm would be unacceptably high when the alignment position of each input array is free.

2.7.5 Linear Algebra Methods

In stead of using the graph model, Ramanujam and Sadayappan [75] use a matrix notation to describe array access functions. Their approach focuses on the search of the existence of a hyperplane such that communication is free if the array is partitioned in a family of such parallel hyperplanes. The hyperplane partition determines both base alignment and offset alignment. However, their approach does not address the issues of how to minimize the interprocessor communication overhead when there does not exist a communication-free hyperplane partition.

Recently, Anderson and Lam [17, 76] use linear space properties to find a hyper-

lane such that *reorganization communication* is free if arrays referenced in a loop are partitioned in a family of such parallel hyperplanes. Reorganization communication refers to communication due to mismatches in base alignment which requires moving the entire data structure. Communication due to mismatches in offset alignment is named *neighboring communication*. Their approach first searches for a linear subspace such that reorganization communication is free if arrays are partitioned in a family of such parallel subspaces. If such reorganization communication-free subspace partition does not exist, their approach uses a data flow iterative algorithm to find an efficient partition which can reduce reorganization communication. However, their approach does not address the offset alignment problem and cannot minimize neighboring communication.

2.7.6 Parallelizing Loops with Data Dependence

All techniques summarized above are focusing on the study of Fortran 90 [13] like array languages. In other words, there is no loop-carried data dependence existing during an array operation. The analysis is much more complicated and difficult when loop-carried data dependence is considered. A simple approach taken by Li and Chen [62] in the component affinity graph model is to assign the weight ∞ to an edge which models a reference involved in a loop-carried data dependence.

Anderson and Lam [17] use tiling method [18, 77] to pipeline communication and computation in a perfectly nested loop with loop-carried data dependence. In [17], a relationship between iteration space and data space is studied and arrays are distributed with regards to the iteration space partition. When the data array size is much larger than the number of available processors, the linear speedup can be almost achieved. In order to guarantee the legality of tiling, their approach first transforms a nested loop into a fully permutable loop [29]. This transformation requires the knowledge of all distance vectors which may not be always practical for any type of

nested loops. In fact, the tiling method can be further extended by using convex hull method proposed by Tzen and Ni [45]. In [45], Tzen and Ni use convex hull to search for the maximum and minimum values of dependence slopes by given any type of dependence functions. A legal tiling can be easily implemented by given the values of the maximum and minimum dependence slopes [18]. Therefore, in theory, the pipeline technique can be used to overlap communication and computation given any perfectly nested loop.

CHAPTER 3

Base Alignment

Base alignment determines the alignment matrix for each array in order to reduce the cost of interprocessor communication. In this chapter, we present a mathematical model to represent base alignment, analyze the communication cost imposed by mismatched alignment, and propose efficient base alignment algorithms.

3.1 Terminology

A loop nest of depth ℓ , with loop bounds that are affine functions of the loop indices, defines an iteration space \mathcal{I} , a polytope in ℓ -dimensional space. Each iteration of the loop nest corresponds to an integer point in the polytope and is identified by its index vector $\vec{i} = (i_1, i_2, \dots, i_\ell)$. An array of dimension m defines an array space \mathcal{A} , an m -dimensional rectangle. Each element in the array can be accessed by an integer vector $\vec{a} = (a_1, a_2, \dots, a_m)$. Therefore, an affine array subscript can be written as $F\vec{i} + \vec{f}$, where F is a linear transformation and \vec{f} is a constant vector. F is called *access matrix* and \vec{f} is called *access offset*. The subscript $F\vec{i} + \vec{f}$ is called *access function*.

If an instance $A(F\vec{i} + \vec{f})$ is referenced in a statement s_k , we want to label the subscript $F\vec{i} + \vec{f}$ by the array variable and the statement number. However, multiple

instances of the same array may be referenced in the same **FORALL** assignment. For instance, $Y(i_1, i_2)$ and $Y(i_2, i_1)$ in statement s_1 of Example 1 (Figure 2.4). The concept of *single-occurrence statement* is used to simplify the symbolic notation used in the thesis.

Definition 3.1 *A single-occurrence statement is the statement in which any referenced array variable can not have more than one type of distinct instances.*

In Example 1, **FORALL** assignment s_2 is a single-occurrence statement, but **FORALL** assignment s_1 is not because two distinct types of instances $Y(i_1, i_2)$ and $Y(i_2, i_1)$ appear. Any assignment statement can be transformed to the equivalent single-occurrence statements by using extra temporary array variables. Example 1 can be re-written as shown in Figure 3.1.

```

FORALL( $i_1 = 0 : n - 1, i_2 = 0 : n - 1 - i_1$ )
 $s_3:$      $TT(i_2, i_1) = Y(i_2, i_1)$ 
 $s_4:$      $Y(i_1, i_2) = TT(i_2, i_1)$ 
 $s_2:$      $X(i_1, i_1 + i_2) = Y(i_1, i_2)$ 
END FORALL

```

Figure 3.1. Example 1 in single occurrence statements

The statement s_2 in the original loop (Figure 2.4) is equivalent to single occurrence statements s_3 and s_4 in the transformed loop (Figure 3.1) by using the temporary array variable TT . The temporary variables used for such single occurrence transformation are only necessary for the sake of alignment analysis and will be ignored in the code generation.

In a **FORALL** structure consisting of only single occurrence statements, index matrices of different instances can be uniquely distinguished by the pair $\langle A, k \rangle$ where

A is an array variable, and k is the statement number. Let $F_{A,k}$ be the access matrix for the instance of array A referenced in **FORALL** assignment s_k . In Example 1 (Figure 3.1), we have $F_{Y,3} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, $F_{Y,4} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $F_{Y,2} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, and $F_{X,2} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$.

In order to make the base alignment results transparent from the single occurrence transformation, the access function of a temporary variable is forced to be same as that of the data array instance which the temporary variable replaces. In Example 1 (Figure 3.1), $F_{TT,4} = F_{TT,3} = F_{Y,3} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. Moreover, the alignment function of the temporary variable is also forced to be same as that of the data array instance which the temporary variable replaces. For temporary variable TT in Example 1 (Figure 3.1), δ_{TT} is always equal to δ_Y . It is assumed that any program structure used in this chapter has been pre-processed by the necessary single occurrence transformation and all base alignment theorems and algorithms are stated based on the denotation of single occurrence statements.

Definition 3.2 *If array A is referenced on the LHS and array B is referenced on the RHS in statement s_k , the read/write relationship between arrays A and B is called a reference and is represented by the symbolic form “ $A \leftarrow B@s_k$ ”.*

In Example 1 (Figure 3.1), we have references “ $TT \leftarrow Y@s_3$ ”, “ $Y \leftarrow TT@s_4$ ”, and “ $X \leftarrow Y@s_2$ ”. The notation “ $A \leftarrow B@s_k$ ” is not ambiguous because A and B have only one type of instance referenced in statement s_k by the assumption of the single occurrence statement.

3.2 Base Alignment for Single Reference

In this section, we use the linear space theory to model the inter-relationship between base alignment and access function regarding to a single reference.

3.2.1 Base Alignment Equation

Consider reference “ $X \leftarrow Y@s_2$ ” in Example 1 (Figure 3.1). In iteration $\begin{pmatrix} i_1 \\ i_2 \end{pmatrix}$, assume that $X(x_1, x_2)$ is referenced on the LHS and $Y(y_1, y_2)$ is referenced on the RHS. Using the access function specification, we have

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = F_{X,2} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} \quad (3.1)$$

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = F_{Y,2} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} \quad (3.2)$$

Multiplying both sides of Equation 3.1 by D_X and both sides of Equation 3.2 by D_Y , we have

$$D_X \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = D_X F_{X,2} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} \quad (3.3)$$

$$D_Y \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = D_Y F_{Y,2} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} \quad (3.4)$$

The purpose of data alignment is to eliminate interprocessor communication. In this case, interprocessor communication guarantees to be avoided if both $X(x_1, x_2)$

and $Y(y_1, y_2)$ can be aligned to the same template element. In other words,

$$\delta_X \left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \right) = \delta_Y \left(\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \right)$$

Therefore, we search for the solution of D_X and D_Y such that

$$D_X \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = D_Y \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

Substituting $D_X \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ by Equation 3.3 and $D_Y \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ by Equation 3.4, we get

$$D_X F_{X,2} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} = D_Y F_{Y,2} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} \quad (3.5)$$

Equation 3.5 can be re-written as follows:

$$(D_X F_{X,2} - D_Y F_{Y,2}) \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} = 0$$

Since $\begin{pmatrix} i_1 \\ i_2 \end{pmatrix}$ represents any iteration in the iteration space, Equation 3.5 holds if and only if

$$D_X F_{X,2} - D_Y F_{Y,2} = 0 \quad (3.6)$$

Equation 3.6 can be easily extended to the following property for any multi-dimensional arrays A and B .

Proposition 1 *Reference “ $A \leftarrow B @_{s_k}$ ” is free of interprocessor communication if*

the following equation holds.

$$D_A F_{A,k} - D_B F_{B,k} = 0$$

or

$$D_A F_{A,k} = D_B F_{B,k} \quad (3.7)$$

Alignment matrices D_A and D_B are *compatible with respect to statement s_k* if Equation 3.7 holds. Otherwise, D_A and D_B are called *incompatible* in respect to that statement.

Consider reference “ $Y \leftarrow TT@s_4$ ” in Example 1 (Figure 3.1). Using Proposition 1, we get

$$D_Y F_{Y,4} = D_{TT} F_{TT,4} \quad (3.8)$$

Since TT is the temporary variable used to replace $Y(i_2, i_1)$ in statement s_3 , it is always true that $D_{TT} = D_Y$ and $F_{TT,4} = F_{Y,3}$. Substituting the values of D_{TT} and $F_{TT,4}$ into Equation 3.8, we get

$$D_Y F_{Y,4} = D_Y F_{Y,3}$$

This can be re-written as

$$D_Y (F_{Y,4} - F_{Y,3}) = 0$$

On Substitution of $F_{Y,4}$ and $F_{Y,3}$, we have

$$D_Y \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \right) = 0$$

$$D_Y \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} = 0 \quad (3.9)$$

which conducts $D_Y = (1, 1)$. Substituting the result of D_Y into alignment function δ_Y , we have

$$\delta_Y \left(\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \right) = D_Y \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = y_1 + y_2$$

Alignment function δ_Y implies that all elements in an off-diagonal must be collapsed to the same element in the template. This guarantees that $Y(i_1, i_2)$ and $Y(i_2, i_1)$ must be assigned to the same processor and thus assignment s_1 is free of reorganization communication.

Substituting values of D_Y , $F_{Y,2}$, and $F_{X,2}$ into Equation 3.6, we have $D_X = (0, 1)$ and the alignment function of X can be specified as follows.

$$\delta_X \left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \right) = D_X \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = x_2$$

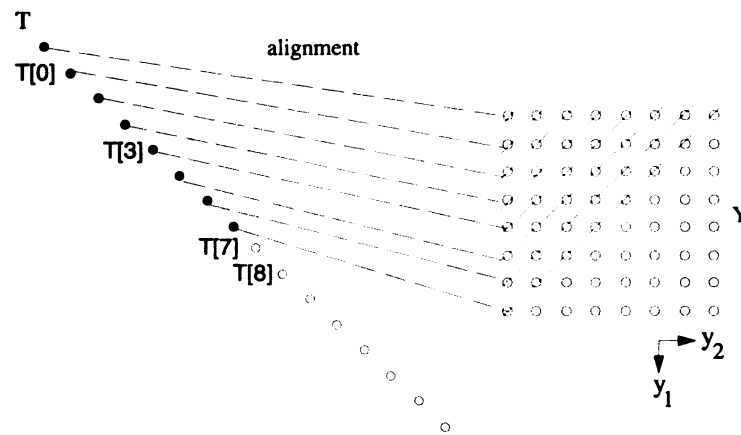
This implies that, in order to make assignment s_2 free of reorganization communication, X must be partitioned in columns if Y is partitioned in off-diagonals.

3.2.2 A Legitimate Solution of Alignment Matrix

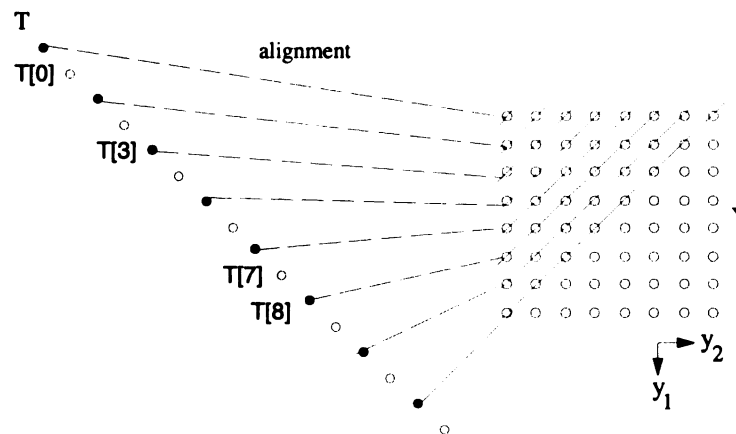
Note that there are infinite solutions of D_Y which can satisfy Equation 3.9. The reason we chose $D_Y = (1, 1)$ is that the image projected by δ_Y from \mathcal{Y} , data space of Y , to \mathcal{T} , data space of T , should be *compact*.

Definition 3.3 *A subspace is compact if and only if the subspace includes every integer grid on any line which two ends are included in the subspace.*

An image on the template space \mathcal{T} is compact if and only if the image includes every integer grid on any line which two ends are included in the image. The concept of the compactness can be explained in Figure 3.2. In Figure 3.2, the template elements included in the image projected by the alignment function are in dark. Figure 3.2(a) shows the compact image projected by δ_Y defined by $D_Y = (1, 1)$. Figure 3.2(b) shows the non-compact image projected by δ_Y defined by $D_Y = (2, 2)$. In Figure 3.2(b), $T(3)$ is the integer grid on the line incident with $T(0)$ and $T(7)$. However, $T(3)$ is not included in the image, while $T(0)$ and $T(7)$ are.



(a) $D_y = (1, 1)$ and the image on T is compact



(b) $D_y = (2, 2)$ and the image on T is not compact

Figure 3.2. The compactness of the image on T

Given a $n \times m$ matrix A , let A_r be a square submatrix of A such that its determinant $|A_r| \neq 0$ and r is the rank of the original matrix A . The value of $|A_r|$ is defined to be the determinant of the rectangle $n \times m$ matrix A , denoted as $|A|$. Array elements which are defined or used in a **FORALL** structure are called *effective elements*. The *effective domain* of an array is composed of all effective elements in the array. Generally speaking, the solution of an alignment matrix D_A is legitimate if the effective domain of A is compact and $|D_A| = 1$. In this case, the image on the template space projected by δ_A will be guaranteed compact. In Example 1 (Figure 2.5), the effective domain of array Y consists of the whole upper triangle and thus is compact. Therefore, $D_Y = (1, 1)$ is legitimate since $|D_Y| = 1$.

However, the determinant of legitimate D_A can be any rational number if the effective domain of array A is no longer compact. Under such circumstance, the stride in A 's effective domain has to be considered in order to find a correct solution of D_A . This can be illustrated by using the Whetstone benchmark loop shown in Example 5 (Figure 3.3).

```

FORALL( $i_1=0:n/2-1, i_2=0:n/3-1$ )
 $s_1:$      $B(2i_1, 3i_2) = A(i_1, 2i_2)$ 
 $s_2:$      $A(i_1, 2i_2) = B(i_1, 3i_2)$ 
END FORALL

```

Figure 3.3. Example 5: A Whetstone benchmark loop

For Example 5, by Proposition 1, we get

$$D_B F_{B,1} = D_A F_{A,1}$$

$$D_A F_{A,2} = D_B F_{B,2}$$

where $F_{B,1} = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$, $F_{B,2} = \begin{pmatrix} 1 & 0 \\ 0 & 3 \end{pmatrix}$, and $F_{A,1} = F_{A,2} = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$. Since $F_{A,1}$ and $F_{A,2}$ are invertible, the above equations can be re-written as follows:

$$D_B F_{B,1} F_{A,1}^{-1} = D_A \quad (3.10)$$

$$D_A = D_B F_{B,2} F_{A,2}^{-1} \quad (3.11)$$

By substitution, we have

$$D_B F_{B,1} F_{A,1}^{-1} = D_B F_{B,2} F_{A,2}^{-1}$$

which can be re-written as follows:

$$\begin{aligned} D_B(F_{B,1}F_{A,1}^{-1} - F_{B,2}F_{A,2}^{-1}) &= 0 \\ D_B\left(\begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & \frac{1}{2} \end{pmatrix} - \begin{pmatrix} 1 & 0 \\ 0 & 3 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & \frac{1}{2} \end{pmatrix}\right) &= 0 \\ D_B \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} &= 0 \end{aligned} \quad (3.12)$$

Any vector $(0, h)$ could be a solution of Equation 3.12. However, there is one legitimate solution of D_B which makes the compact image on the template space projected by alignment function δ_B . Since $B(2i_1, 3i_2)$ and $B(i_1, 3i_2)$ are referenced in Example 5 (Figure 3.3), the stride in B 's effective domain regarding to the row dimension is 3. Therefore, we choose

$$D_B = (0, \frac{1}{3})$$

Substituting the value of D_B into Equation 3.11, we have

$$D_A = (0, \frac{1}{2})$$

Therefore, the alignment functions of A and B can be specified as follows.

$$\delta_A \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \frac{1}{2}a_2$$

$$\delta_B \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \frac{1}{3}b_2$$

where $A(a_1, a_2)$ is an A element and $B(b_1, b_2)$ is a B element.

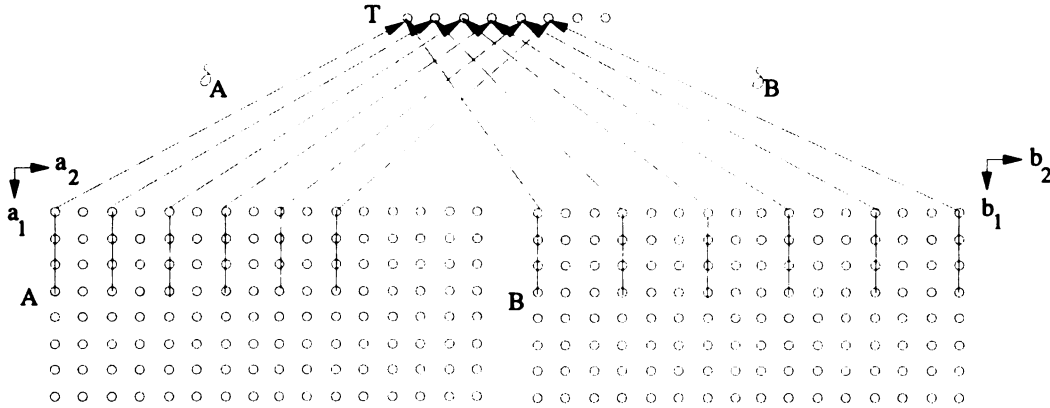


Figure 3.4. The alignment for A and B in Example 5

Figure 3.4 shows the alignment of A and B in Example 5. In Figure 3.4, effective elements in A and B are covered by solid lines. The effective domain of A only contains elements (a_1, a_2) such that a_2 must be a multiplier of 2. The effective domain of B only contains elements (b_1, b_2) such that b_2 must be a multiplier of 3. Though neither

A 's effective domain nor B 's effective domain is compact, the image on the template space projected by δ_A and δ_B is compact.

3.2.3 Solving Base Alignment Equation

If access matrix $F_{A,k}$ is non-singular, Equation 3.7 can be easily transformed into

$$D_A = D_B F_{B,k} F_{A,k}^{-1}$$

and the relationship of D_A and D_B is straightforward. However, solving Equation 3.7 may not be simple when both $F_{A,k}$ and $F_{B,k}$ are singular. For singular matrices $F_{A,k}$ and $F_{B,k}$, let $F_{A,k}^R$ be the right inverse matrix of $F_{A,k}$ and $F_{B,k}^R$ be the right inverse matrix of $F_{B,k}$. Therefore, Equation 3.7 is equivalent to the following two equations

$$D_A I_{r_A} = D_B F_{B,k} F_{A,k}^R$$

$$D_A F_{A,k} F_{B,k}^R = D_B I_{r_B}$$

where r_A is the rank of D_A and r_B is the rank of D_B . The method to find a right inverse matrix of a given matrix can be found in [78]. Example 6 (Figure 3.5) is used to illustrate the basic idea.

```

      FORALL( $i_1 = 0 : n - 1, i_2 = 0 : n - 1$ )
 $s_1$ :       $ZZ(i_1, i_2) = 0$ 
          DO  $i_3 = 0 : n - 1$ 
 $s_2$ :       $ZZ(i_1, i_2) = ZZ(i_1, i_2) + XX(i_1, i_3) \times YY(i_3, i_2)$ 
          END DO
      END FORALL

```

Figure 3.5. Example 6: Inner product benchmark loop

In Example 6, the inner DO loop is sequential and thus will not be distributed across the processors. $F_{ZZ,2} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$, $F_{XX,2} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$, and $F_{YY,2} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$. Consider the equation

$$D_{ZZ}F_{ZZ,2} = D_{XX}F_{XX,2}$$

for reference “ $ZZ \leftarrow XX@s_2$ ”. Since $F_{ZZ,2}$ and $F_{XX,2}$ are singular matrices, the above equation is equivalent to

$$D_{ZZ}I_{r_{ZZ}} = D_{XX}F_{XX,2}F_{ZZ,2}^R$$

$$D_{ZZ}F_{ZZ,2}F_{XX,2}^R = D_{XX}I_{r_{XX}}$$

where $r_{ZZ} = r_{XX} = 2$, $F_{XX,2}^R = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}$ and $F_{ZZ,2}^R = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$. On substitution of those values, we have

$$D_{ZZ} = D_{XX} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

$$D_{ZZ} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} = D_{XX}$$

Therefore, the solution of both D_{ZZ} and D_{XX} have to be in the format of $(h, 0)$.

The similar approach is used in solving the equation

$$D_{ZZ}F_{ZZ,2} = D_{YY}F_{YY,2}$$

for reference “ $ZZ \leftarrow YY@s_2$ ” since $F_{ZZ,2}$ and $F_{YY,2}$ are singular matrices. The solution specified by this equation requires both D_{ZZ} and D_{YY} to be in the format of $(0, g)$. The requirement of D_{ZZ} imposed by two equations conflicts with each other, which implies that interprocessor communication cannot be avoided.

3.3 The Cost of Reorganization Communication

Reorganization communication occurs if Proposition 1 does not hold for a given reference. This section studies the cost of reorganization communication.

3.3.1 Reorganization Communication

Consider reference “ $X \leftarrow Y@s_2$ ” in Example 1 (Figure 2.4). In iteration $\begin{pmatrix} i_1 \\ i_2 \end{pmatrix}$, assume that $X(x_1, x_2)$ is referenced on the LHS and $Y(y_1, y_2)$ is referenced on the RHS. Using the access function specification, we have

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = F_{X,2} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix}$$

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = F_{Y,2} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix}$$

Since $F_{X,2}$ and $F_{Y,2}$ are invertible, the above equations can be rewritten as

$$\begin{pmatrix} i_1 \\ i_2 \end{pmatrix} = F_{X,2}^{-1} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

$$\begin{pmatrix} i_1 \\ i_2 \end{pmatrix} = F_{Y,2}^{-1} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

where $F_{X,2}^{-1}$ is the inverted matrix of $F_{X,2}$ and $F_{Y,2}^{-1}$ is the inverted matrix of $F_{Y,2}$. By substitution, we get

$$F_{X,2}^{-1} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = F_{Y,2}^{-1} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

It can be rewritten as

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = F_{Y,2} F_{X,2}^{-1} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

Multiplying both sides of the above equation by D_Y , we get

$$D_Y \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = D_Y F_{Y,2} F_{X,2}^{-1} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad (3.13)$$

Assume that given values of D_Y and D_X , Proposition 1 does not hold for reference “ $X \leftarrow Y@s_2$ ”. In other words,

$$D_X F_{X,2} \neq D_Y F_{Y,2}$$

Since $F_{X,2}$ is invertible,

$$D_X \neq D_Y F_{Y,2} F_{X,2}^{-1} \quad (3.14)$$

On substitution of $D_Y F_{Y,2} F_{X,2}^{-1}$ from Inequality 3.14, Equation 3.13 can be rewritten as

$$D_Y \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \neq D_X \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

In other words,

$$\delta_Y\left(\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}\right) \neq \delta_X\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right)$$

This means that array elements $\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ and $\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ are never aligned to the same element in the template. Moreover, by Equation 3.13, we have

$$\begin{aligned} \delta_Y\left(\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}\right) - \delta_X\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) &= D_Y\left(\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}\right) - D_X\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) \\ &= D_Y F_{Y,2} F_{X,2}^{-1} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - D_X \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = (D_Y F_{Y,2} F_{X,2}^{-1} - D_X) \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \end{aligned}$$

By Equation 3.14, the difference between $D_Y\left(\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}\right)$ and $D_X\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right)$ is an affine function of $\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$. However, if template elements are distributed in regular patterns,

such as block or cyclic, $\delta_Y\left(\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}\right)$ and $\delta_X\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right)$ are unlikely to be mapped onto the same processor. This implies that to write an X element, the local processor almost always requires a remote access to the RHS Y element. Figure 3.6 is used to illustrate the idea.

In Figure 3.6, arrays X and Y (in Example 1) are declared as 8×8 . The template array is one-dimensional and has eight elements. There are four processors, denoted as $P(0 : 3)$. The template elements are distributed onto these four processors in block. Since $D_X = (1, -1)$ and $D_Y = (1, 1)$, X is distributed along diagonal and Y is distributed along anti-diagonal. Equation 3.7 does not hold for reference “ $X \leftarrow$

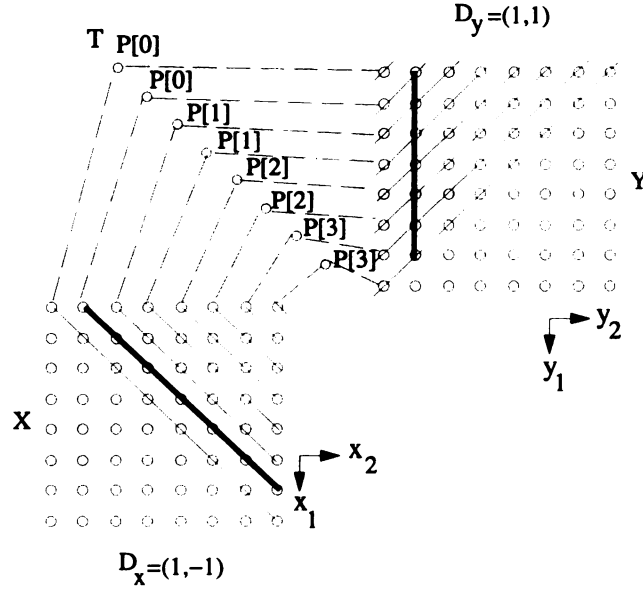


Figure 3.6. Reorganization communication for $D_X = (1, -1)$ and $D_Y = (1, 1)$ in Example 1

$Y@s_2$ ". To write an X element on the line $x_1 - x_2 = 1$ (highlighted in dark), a distinct RHS Y element on the line $y_2 = 1$ (highlighted in dark) is accessed. However, all X elements on the line $x_1 - x_2 = 1$ are owned by processor $P(0)$, while six out of seven elements on the line $y_2 = 1$ are owned by other three processors.

Based on the above observation, we have the following proposition.

Proposition 2 Assume that array elements are evenly distributed across the processors. For reference " $A \leftarrow B@s_k$ ", the cost of reorganization communication is $\frac{n}{p}$ if

$$D_A F_{A,k} \neq D_B F_{B,k}$$

where n is the total number of elements involved in reference " $A \leftarrow B@s_k$ " and p is the total number of available processors.

Since p is the total number of available processors and is a fixed constant to the

base decomposition analysis, n , the total number of elements involved in reference “ $A \leftarrow B@s_k$ ”, determines the cost of reorganization communication. In the rest of this chapter, the cost of reorganization communication is simply measured in n .

Proposition 2 justifies the correctness of the single occurrence transformation with regarding to base alignment. In the following loop, the original **FORALL** assignment

```

FORALL( $i_1 = 0 : n - 1, i_2 = 0 : n - 1 - i_1$ )
 $s_1$ :    $X(i_1, i_1 + i_2) = Y(i_1, i_2) + Y(i_1, i_1 + i_2)$ 
END FORALL

```

s_1 references two distinct instances of array Y : $Y(i_1, i_2)$ and $Y(i_1, i_1 + i_2)$. In order to distinguish the access matrices of these two instances, let $F_{Y,1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ and $F'_{Y,1} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$. Assume that $D_X = (1, -1)$ and $D_Y = (1, 1)$. Since $D_X F_{X,1} \neq D_Y F_{Y,1}$, by Proposition 2, the number of remote Y required to be accessed is $\frac{n}{p}$ regarding to instance $Y(i_1, i_2)$. Similar, since $D_X F_{X,1} \neq D_Y F'_{Y,1}$, by Proposition 2, the number of remote Y required to be accessed is $\frac{n}{p}$ regarding to instance $Y(i_1, i_1 + i_2)$. Moreover, since $D_Y F_{Y,1} \neq D_Y F'_{Y,1}$, corresponding elements $Y(i_1, i_2)$ and $Y(i_1, i_1 + i_2)$ are allocated to different remote processors. Therefore, the total cost of reorganization communication is $\frac{2n}{p}$. This cost estimation is consistent with the result obtained from the following transformed loop, which has the property of single occurrence. For this

```

FORALL( $i_1 = 0 : n - 1, i_2 = 0 : n - 1 - i_1$ )
 $s_2$ :    $TT(i_1, i_2) = Y(i_1, i_2)$ 
 $s_3$ :    $X(i_1, i_1 + i_2) = TT(i_1, i_2) + Y(i_1, i_1 + i_2)$ 
END FORALL

```


reason, we claim that single occurrence transformation does not bring any side effect in the base alignment analysis.

3.3.2 The Weighted Cost

In Proposition 2, the total number of elements involved in reference “ $A \leftarrow B@s_k$ ” may not be equal to the number of all elements in array A or B . Typically there are three different cases resulting in different costs of reorganization communication. First, the number of elements involved in reference “ $A \leftarrow B@s_k$ ” are limited to the effective domain imposed by the loop boundary. In Example 1 (Figure 2.4), suppose that the size of arrays X and Y is $n_0 \times n_1$. The number of effective elements limited by the loop boundary is only equal to $\frac{1}{2}n^2$.

Second, the number of effective elements involved in reference “ $A \leftarrow B@s_k$ ” can be limited by the probability of the **WHERE** clause used in a **FORALL** assignment. For example, in the following code, the total number of elements in array X is 10,000.

```

FORALL( $i_1 = 0 : 100 - 1, i_2 = 0 : 100 - 1$ )
  WHERE( $X(i_1, i_2).NE.0$ )
 $s_1:$      $X(i_1, i_2) = X(i_1, i_2) * 2$ 
  END WHERE
END FORALL

```

However, if the possibility for the condition in **WHERE** clause to be true is only 90%, the cost of reorganization would be only 9,000 provided that the elements which values are zero are evenly distributed among processors.

Third, the number of elements involved in reference “ $A \leftarrow B@s_k$ ” can be equal to the number of elements in B where the size of B is much larger than that of A . For example, in the following code, the inner loop s_2 is a sequential loop. Therefore, the


```

      FORALL( $i_1 = 0 : 100 - 1, i_2 = 0 : 100 - 1$ )
 $s_1$ :       $X(i_1, i_2) = Y(i_1, i_2)$ 
          For( $i_3 = 0 : 100 - 1$ )
 $s_2$ :       $X(i_1, i_2) = X(i_1, i_2) + Y(i_1, i_2, i_3)$ 
          END FOR
      END FORALL

```

cost of reorganization communication is 1,000,000, which equals to the number of elements in array Y , if D_X and D_Y are incompatible with respect to statement s_2 .

3.4 Spanning-Tree Base Alignment Algorithm

3.4.1 Data Reference Graph

The problem of base alignment can be simply modeled by *data reference graph* (DRG). Given a program structure, a DRG $G = (V, E)$ is constructed as follows. An array is represented by a node in V . For an array which has multiple instances referenced in one or more **FORALL** assignments, there is only one corresponding node in the DRG. There is a distinct edge in E connecting two nodes for each reference between the corresponding two arrays. A DRG is undirected.

Figure 3.8(a) shows the DRG for Example 7 (Figure 3.7). In Figure 3.8(a), each array variable is represented by a node labeled by the variable name. Edges are constructed based on the references generated by each **FORALL** assignment in Example 7. For example, edge (A, X) connects nodes A and X due to reference " $A \leftarrow X@s_2$ ". There is no edge between nodes X and Y because these two arrays are not involved in the same reference. Since there is one-to-one correspondence between an array and a node, terms "array" and "node" are used alternatively in the rest of this chapter. Similarly, since there is one-to-one correspondence between an edge and reference, terms "edge" and "reference" are also used alternatively in the rest of this chapter.

In Figure 3.8 (a), each edge is weighted by the cost of reorganization communication if alignment matrices of two arrays connected by the edge are incompatible with respect to the reference represented by the edge. The weight on each edge is decided based on the probability of true condition in the **WHERE** clause.

```

      FORALL( $i_1 = 0 : 10, i_2 = 0 : 10$ )
s1:       $Y(i_1, i_2) = A(2i_1 + i_2, i_1 + i_2)$ 

      WHERE( $A(i_1, i_2).NE.0$ )
!HPF the probability of  $A(i_1, i_2) \neq 0$  is 83%
s2:       $A(i_1, i_2) = W(i_1, i_2) + X(i_1 + i_2, i_1 + 2i_2)$ 
      END WHERE

      WHERE( $Z(i_1, i_2).NE.0$ )
!HPF the probability of  $Z(i_1, i_2) \neq 0$  is 83%
s3:       $Z(i_1, i_2) = B(i_1, i_2) * Y(i_1, i_2)$ 
      END WHERE

s4:       $W(i_1, i_2) = Z(i_1 + i_2, i_1 + 2i_2)$ 

      WHERE( $B(i_1, i_2).NE.0$ )
!HPF the probability of  $B(i_1, i_2) \neq 0$  is 53%
s5:       $B(i_1, i_2) = A(2i_1 + i_2, i_1 + i_2)$ 
      END WHERE

      WHERE( $X(i_1, i_2).NE.0$ )
!HPF the probability of  $X(i_1, i_2) \neq 0$  is 53%
s6:       $X(i_1, i_2) = Z(i_1, i_2)$ 
      END WHERE
END FORALL

```

Figure 3.7. Example 7: A Lapack benchmark loop

Generally speaking, there may not exist a solution of the alignment matrices such that every reference can be free of reorganization communication. This is because the

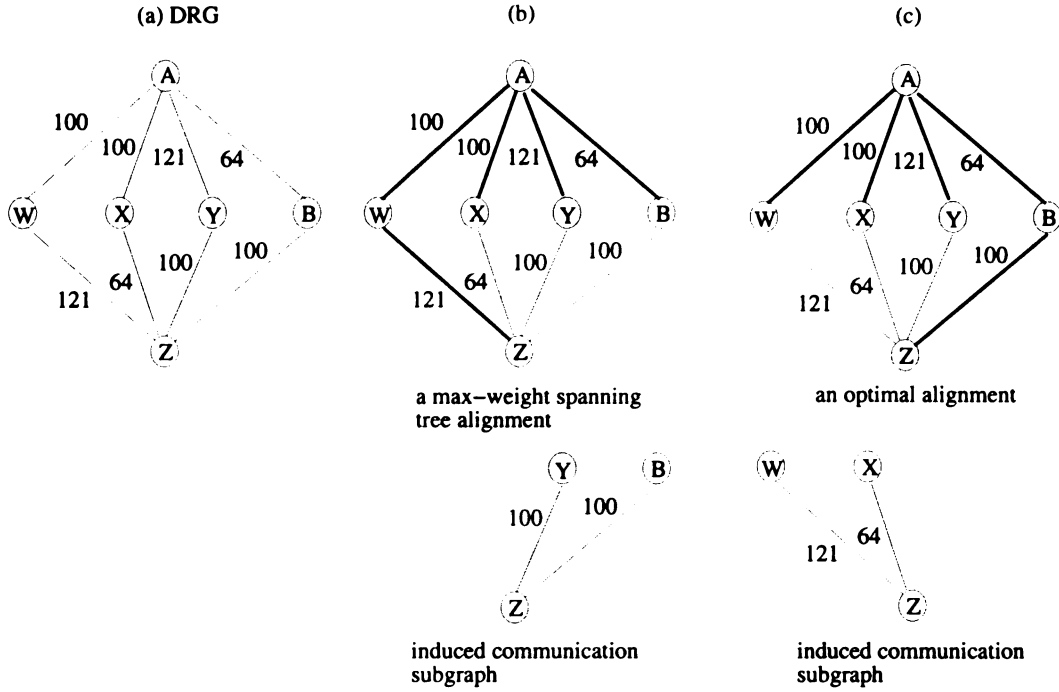


Figure 3.8. The DRG and base alignment for Example 7

compatibility requirement imposed by one reference may conflict with that imposed by another reference, in particular, when the two edges representing these two references are involved in a cycle. For example, Figure 3.8(a) consists of cycle $A \rightarrow W \rightarrow Z \rightarrow Y \rightarrow A$. By Proposition 3.7, the four references represented by four edges in the cycle are free of reorganization communication if and only the following equations have a non-trivial solution of alignment matrices D_A , D_W , D_Z , and D_Y .

$$\begin{cases} D_A F_{A,2} = D_W F_{W,2} \\ D_W F_{W,4} = D_Z F_{Z,4} \\ D_Z F_{Z,3} = D_Y F_{Y,3} \\ D_Y F_{Y,1} = D_A F_{A,1} \end{cases}$$

where $F_{A,2} = F_{W,4} = F_{Z,3} = F_{Y,1} = F_{Y,3} = F_{W,2} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $F_{Z,4} = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}$, and $F_{A,1} = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}$. Since all index matrices are invertible, the above equations can be re-written as follows.

$$\begin{cases} D_A = D_W F_{W,2} F_{A,2}^{-1} \\ D_W = D_Z F_{Z,4} F_{W,4}^{-1} \\ D_Z = D_Y F_{Y,3} F_{Z,3}^{-1} \\ D_Y = D_A F_{A,1} F_{Y,1}^{-1} \end{cases}$$

By substitution, we get

$$D_A = D_A F_{A,1} F_{Y,1}^{-1} F_{Y,3} F_{Z,3}^{-1} F_{Z,4} F_{W,4}^{-1} F_{W,2} F_{A,2}^{-1}$$

Using the value of each alignment matrix, the above equation can be written as

$$\begin{aligned} D_A &= D_A \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \\ D_A \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \begin{pmatrix} 3 & 4 \\ 2 & 3 \end{pmatrix} \right) &= 0 \\ D_A \begin{pmatrix} -2 & -2 \\ -3 & -2 \end{pmatrix} &= 0 \end{aligned}$$

In order to satisfy the above equation, D_A has to be $(0,0)$. This implies that there does not exist a solution of alignment matrices such that all the four references in the cycle are free of reorganization communication. In other words, there exists at least one reference with respect to which any given solution of alignment matrices is incompatible.

3.4.2 Spanning Tree Base Alignment Algorithms

If all index matrices are invertible, the conflict of compatibility requirement can only occur within a cycle of a DRG. Intuitively, such conflict can be resolved by a spanning tree. Base alignment between two arrays are specified by the tree edge connecting these two arrays using Equation 3.7. As a result, references represented by tree edges are always free of reorganization communication. Each non-tree edge determines a unique fundamental cycle of the DRG with respect to the spanning tree. Reorganization communication may not be avoided for each non-tree edge. An edge is weighted. Given a choice between two edges, the tree edge would be chosen as the one with higher weight. As a result, the spanning tree for base alignment would be chosen as a maximum-weight spanning tree.

For example, Figure 3.8(b) shows a maximum-weight spanning tree for base alignment in Example 7. In Figure 3.8(b), $D_W = D_A$, $D_X = D_A \begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix}$, $D_Y = D_A \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}$, $D_B = D_A \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}$, and $D_Z = D_W \begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix} = D_A \begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix}$. Though (Z, X) is a non-tree edge, D_X and D_Z are compatible with respect to edge (Z, X) because D_X and D_Z satisfy the equation

$$D_X F_{X,6} = D_Z F_{Z,6}$$

Therefore, non-tree edge (Z, X) is also free of reorganization communication. Given $D_W = D_A$ and $D_X = D_A \begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix}$, edges (W, Z) and (X, Z) are called *homogeneous* since the base alignment equations imposed by two edges are equivalent.

$$D_X F_{X,6} = D_Z F_{Z,6}$$

$$D_W F_{W,4} = D_Z F_{Z,4}$$

Edges (W, Z) and (X, Z) comprise a *homogeneous edge set*.

In Figure 3.8(b), reorganization communication can not be avoided on non-tree edges (Y, Z) and (B, Z) since D_X and D_Z are not compatible with respect to edge (X, Z) and D_X and D_Y are not compatible with respect to edge (Y, Z) . Fixing base alignment in a DRG G , an *induced communication subgraph* (ICS) $G^I = (V^I, E^I)$ is a subgraph of G such that alignment matrices are compatible with respect to each edge in $E - E^I$ but incompatible with respect to each edge in E^I . Therefore, the total cost of reorganization communication in G is equal to that in G^I . The ICS for the maximum-weight spanning tree alignment is shown in Figure 3.8(b). The total cost of reorganization communication in Figure 3.8(b) is equal to 200. The problem of optimizing base alignment is to find a base alignment such that the cost of reorganization communication in the ICS is minimal.

Does the maximum-weight spanning tree (MWST) algorithm always minimize the cost of neighboring communication? The answer is no. Figure 3.7(c) shows yet another spanning tree alignment. The cost of reorganization communication in Figure 3.7(c) is only 185 because D_Z and D_Y are compatible with respect to non-tree edge (X, Y) . The failure of the MWST algorithm is due to the assumption that only tree edge is free of reorganization communication. However, in fact not every non-tree edge is necessary included in the ICS. A non-tree edge can also be free of reorganization communication as long as the alignment matrices of two incident arrays are compatible. Next, we introduce a new spanning tree base alignment algorithm which aims to minimize the sum of weights on those non-tree edges included in the ICS. We name it as the minimum-weight ICS (MICS) algorithm.

Definition 3.4 *Given a DRG $G = (V, E)$, let U be an arbitrary subset of V . Node A has the single-degree connectivity with U if and only if A is not in U and there is only one node B in U such that edge (A, B) is in E .*

Definition 3.5 Given a DRG $G = (V, E)$, let U be an arbitrary subset of V . Node A has the multi-degree connectivity with U if and only if A is not in U and there exist at least two distinct nodes X and Y in U such that edges (A, X) and (A, Y) are in E .

Definition 3.6 Given a DRG $G = (V, E)$, let U be an arbitrary subset of V . Node A is a single-degree neighbor of U if A has the single-degree connectivity with U . Node B is a multi-degree neighbor of U if B has the multi-degree connectivity with U .

For example, in Figure 3.8(a), let $U = \{A, X, Y\}$. Thus, W and B are single-degree neighbors of U . Z is a multi-degree neighbor of U since it is incident with both X and Y .

Given a DRG $G = (V, E)$, the MICS algorithm can be formalized as follows:

```

(1)  $T = \phi$ 
(2) while  $T \neq V$  do
(3)   Let  $Q_1$  be the set of single-degree neighbors of  $T$ 
(4)   if  $Q_1 \neq \phi$  then
(5)     Find a node  $A$  in  $Q_1$  such that edge  $(A, B)$  has the maximum weight
        among all edges incident with one node in  $Q_1$  and another in  $T$ 
(6)     Define  $D_A$  such that  $D_A F_{A,k} = D_B F_{B,k}$  where  $(A, B)$  represents
        reference " $A \leftarrow B@s_k$ "
(7)      $T = T \cup \{A\}$ 
(8)   else
(9)     Let  $Q_2 = V - T$ 
(10)    Find a node  $A$  in  $Q_2$  such that the homogeneous edge set, each edge in
        which is incident with  $A$  and a node in  $T$ , has the maximum
        accumulated weight
(11)    Define  $D_A$  such that  $D_A F_{A,k} = D_B F_{B,k}$  where reference " $A \leftarrow B@s_k$ "
        is represented by an edge  $(A, B)$  in the above homogeneous edge set
(12)     $T = T \cup \{A\}$ 
(13)   end if
(14) end while

```

Figure 3.9. The minimum-weight induced communication algorithm

Figure 3.10 illustrates how base alignment is resolved using the MICS algorithm (Figure 3.9). Table 3.1 shows the contents of T , Q_1 , and Q_2 at each step of outmost while loop (lines (2)-(14)). Initially, T is empty and any node in V is assumed to be a single-degree neighbor of an empty set. In Figure 3.10, the nodes included in T at each step are highlighted. The algorithm starts with X which is arbitrarily selected and $T = \{X\}$. In step (b), since the weight of edge (X, A) is greater than that of (X, Z) , A is chosen to be a new member of T (line (5)). Arrays X and A are aligned by $D_A F_{A,2} = D_X F_{X,2}$ (line (6)). For the same reason, in step (c), node Y is included in T because edge (A, Y) has the maximum weight among edges incident with nodes W, Z, Y , and B , all single-degree neighbors of $\{A, X\}$ (line (5)). Arrays A and Y are aligned by $D_Y F_{Y,1} = D_A F_{A,1}$ (line (6)). The same algorithm repeats in steps (d) and (e). Nodes W and B are included in T , respectively. Eventually, node Z becomes a multi-degree neighbor of T .

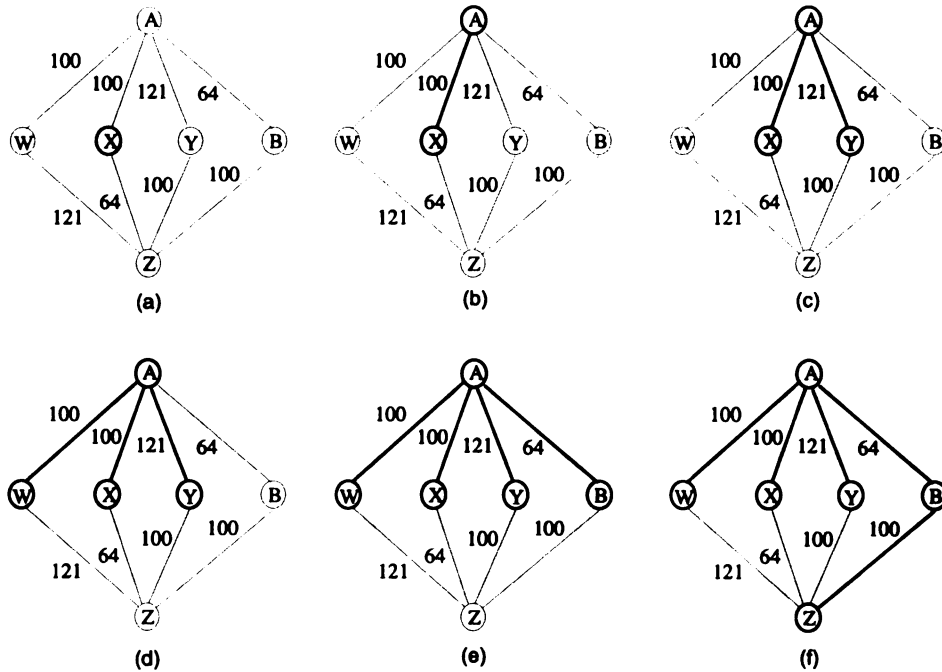


Figure 3.10. Use the MICS algorithm to resolve base alignment for Example 7

Table 3.1. Values of T , Q_1 , and Q_2 in executing the MICS algorithm for Example 7

steps	T	Q_1	Q_2	base alignment
Initial	ϕ	$\{A, W, X, Y, B, Z\}$	ϕ	
(a)	$\{X\}$	$\{A, Z\}$	ϕ	
(b)	$\{X, A\}$	$\{W, Y, B, Z\}$	ϕ	$D_A = D_X F_{X,2}$
(c)	$\{X, A, Y\}$	$\{W, B\}$	$\{Z\}$	$D_Y = D_A F_{A,1}$
(d)	$\{X, A, Y, W\}$	$\{B\}$	$\{Z\}$	$D_W = D_A$
(e)	$\{X, A, Y, W, B\}$	ϕ	$\{Z\}$	$D_B = D_A F_{A,4}$
(f)	$\{X, A, Y, W, B, Z\}$	ϕ	ϕ	$D_Z = D_Y$

In step (f), $Q_1 = \phi$. By line (9), $Q_2 = V - T = \{Z\}$. Node Z is incident with two homogeneous edge sets. One set includes edges (W, Z) and (X, Z) . The other includes edges (Y, Z) and (B, Z) . The accumulated weight in the set $\{(W, Z), (X, Z)\}$ is equal to 185, while the accumulated weight in the set $\{(Y, Z), (B, Z)\}$ is equal to 200. Since nodes W , X , Y , and B are all included in T , by line (11), D_Z is chosen such that $D_Z F_{Z,3} = D_Y F_{Y,3}$ because edge (Z, Y) is in the homogeneous edge set with the larger weight.

The MICS algorithm (Figure 3.9) is an improvement of the MWST algorithm. If Q_1 is not empty, like the MWST algorithm, the tree edge is selected as an edge with the largest weight among all the edges which are incident with one node in Q_1 and another node in T (lines (4)-(7)). If Q_1 is empty but Q_2 is not, alignment matrix is determined such that every edge in a homogeneous edge set with the maximum accumulated-weight is free of reorganization communication. By its construction, the MICS algorithm is superior to the MWST algorithm in general. If the heap-sort algorithm [79] is used in lines (5) and (10), the time complexity of finding the maximum-weight edge or homogeneous edge set can be reduced to $O(\log|E|)$ where $|E|$ is the number of edges in DRG $G = (V, E)$. As a result, the time complexity of the MICS algorithm is $O(|E|\log|E|)$.

3.4.3 Experimental Results

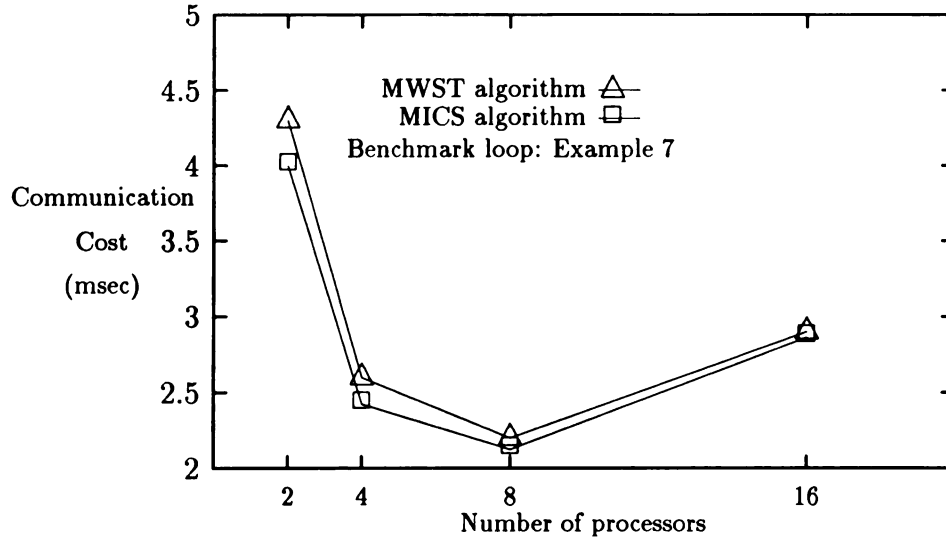


Figure 3.11. Comparison of MWST algorithm and MICS algorithm on 16-node nCUBE-2

Figure 3.11 shows the comparison of communication cost between the MWST algorithm and MICS algorithm on 16-node Purdue nCUBE-2. Example 7 (Figure 3.7) is used as the benchmark loop in our experiment. We increase the iteration space in Example 7 by allocating the loop boundary as $(i_1 = 0 : 34, i_2 = 0 : 34)$. In Figure 3.11, when the number of processors is 16, the size of the messages sent out from each processor reaches the minimum and the number of messages reaches the maximum. Since the startup software latency is much expensive than the network latency in message transmission on Purdue nCUBE-2, the startup latency dominates the overall communication latency when the message size is relatively small. This explains why the overall communication overhead increases when the number of processors becomes 16.

3.5 Optimizing RHS Expression Evaluation

3.5.1 RHS Expression Evaluation Optimization

In the MICS algorithm, it is assumed that all the operations in a **FORALL** assignment are performed in the local processor which owns the LHS operand. If a RHS operand is owned by a remote processor, message passing is invoked in order to transfer the operand to the local processor first. However, this limitation, known as *owner-computes rule*, can be exceeded by evaluating different parts of the RHS expression in a **FORALL** assignment on different processors. Given a **FORALL** assignment which consists of one kind of associative and commutative operations, data movement can be minimized by an optimal evaluation tree in which an intermediate result may be evaluated by a remote processor rather than the one which owns the LHS operand.

```

FORALL( $i_1 = 0 : 9, i_2 = 0 : 9$ )
 $s_1:$      $A(i_1, i_2) = B(i_1, i_2) * X(i_1, i_2) * Y(i_1, i_2) * Z(i_1, i_2)$ 
 $s_2:$      $B(i_1, i_2) = A(2i_1 + i_2, i_1 + i_2) + X(2i_1 + i_2, i_1 + i_2) + Y(2i_1 + i_2, i_1 + i_2)$ 
            $+ Z(i_1, i_2)$ 
END FORALL

```

Figure 3.12. Example 8: A Splash benchmark loop

In Example 8 (Figure 3.12), it is assumed that base alignment is pre-determined such that $D_A F_{A,1} = D_B F_{B,1} = D_X F_{X,1} = D_Y F_{Y,1} = D_Z F_{Z,1}$. Therefore, **FORALL** assignment s_1 is free of reorganization communication. Nevertheless, reorganization communication cannot be eliminated in **FORALL** assignment s_2 because $D_B F_{B,2} \neq D_A F_{A,2}$, $D_B F_{B,2} \neq D_X F_{X,2}$, and $D_B F_{B,2} \neq D_Y F_{Y,2}$. If the owner-computes rule is followed, remote elements $A(2i_1 + i_2, i_1 + i_2)$, $X(2i_1 + i_2, i_1 + i_2)$, and $Y(2i_1 + i_2, i_1 + i_2)$

have to be transferred to the local processor which owns the LHS $B(i_1, i_2)$. As a result, the cost of reorganization communication is equal to 300.

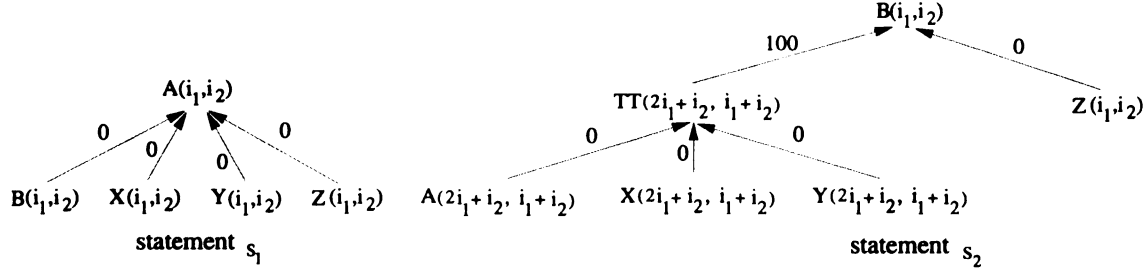


Figure 3.13. Optimal evaluation trees for Example 8

However, closer inspection reveals that $D_A F_{A,2} = D_X F_{X,2} = D_Y F_{Y,2}$. This implies that the sum of $A(2i_1 + i_2, i_1 + i_2)$, $X(2i_1 + i_2, i_1 + i_2)$, and $Y(2i_1 + i_2, i_1 + i_2)$ can be first calculated on a remote processor without any data movement. Figure 3.13 shows such an optimal evaluation tree for Example 8. As shown in Figure 3.13, $TT(2i_1 + i_2, i_1 + i_2)$, a temporary variable which stores the sum, is then added with $Z(i_1, i_2)$ by passing a message to the local processor which owns $B(i_1, i_2)$. In Figure 3.13, each arc represents a reference relationship from one array to the other. Each arc is weighted by the cost of reorganization communication if the alignment matrices of two corresponding arrays are incompatible with respect to the reference represented by the arc. The optimal evaluation of Example 8 can be rewritten as follows:

```

FORALL( $i_1 = 0 : 9, i_2 = 0 : 9$ )
 $s_1:$        $A(i_1, i_2) = B(i_1, i_2) * X(i_1, i_2) * Y(i_1, i_2) * Z(i_1, i_2)$ 
 $s_{2.1}:$    $TT(2i_1 + i_2, i_1 + i_2) = A(2i_1 + i_2, i_1 + i_2) + X(2i_1 + i_2, i_1 + i_2)$ 
            $+ Y(2i_1 + i_2, i_1 + i_2)$ 
 $s_{2.2}:$    $B(i_1, i_2) = TT(i_1, i_2) + Z(i_1, i_2)$ 
END FORALL

```


The original assignment s_2 is transformed into two assignments $s_{2.1}$ and $s_{2.2}$. The alignment matrix of TT is determined as $D_{TT} = D_A$. Statement $s_{2.1}$ is free of reorganization communication. Reorganization communication only occurs in assignments $s_{2.2}$. The total cost of reorganization communication is reduced to 100.

Generally speaking, given a **FORALL** assignment, the RHS expression evaluation can be optimized using the following rule.

Proposition 3 *Assume that in a given **FORALL** assignment, the RHS expression is operated by one kind of associate and commutative operations. Each group of the RHS array operands which alignment matrices are compatible with each other should be evaluated together. Only the intermediate results needs to be transmitted to the local processor which owns the LHS operand if the intermediate results are generated on a remote processor.*

In order to take advantage of optimal expression evaluation, we assume that the original program has been pre-processed by transforming each original **FORALL** assignment into an equivalent set of **FORALL** assignments, each of which has the RHS expression operated by one kind of associate and commutative operations.

3.5.2 Alignment Graph

An *alignment graph* (AG) is used to model the alignment problem. An AG is a collection of arrays and statements which can be represented as a bipartite graph, $G = (V_a, V_s, E)$. An array is represented by a node in V_a . A statement is represented by a node in V_s . A undirected edge in E connects an array and a statement if the array is referenced in the statement.

Figure 3.15 shows the AG for Example 9 (Figure 3.14).


```

FORALL( $i_1 = 0 : 9, i_2 = 0 : 9$ )
 $s_1$ :    $A(i_1, i_2) = B(i_1, i_2) * X(i_1, i_2) * Y(i_1, i_2)$ 
 $s_2$ :    $B(i_1, i_2) = A(i_1 + i_2, i_1) + X(i_1 + i_2, i_1) + Y(i_1 + i_2, i_1)$ 
 $s_3$ :    $X(i_1, i_2) = B(i_1, i_2) * A(i_1 + i_2, i_1)$ 
END FORALL

```

Figure 3.14. Example 9: An Electric benchmark loop

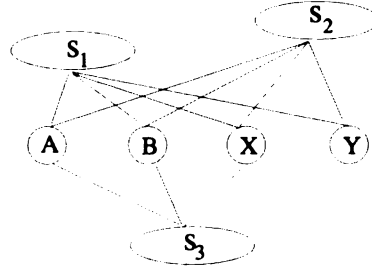


Figure 3.15. Alignment graph for Example 9

3.5.3 AG Base Alignment Algorithm

The AG-based base alignment (AGBA) algorithm is shown in Figure 3.16. In an AG $G = (V_a, V_s, E)$, there is a set, denoted as Q_k , associated with each node s_k in V_s . Initially, each set Q_k is empty. During the execution, each Q_k will contain elements of type $D_A F_{A,k}$. Each element of Q_k , $D_A F_{A,k}$, is further weighted by the number of effective elements in array A referenced in assignment s_k .

By Proposition 3, a group of the RHS array operands with compatible alignment matrices can be evaluated together. Thus, the cost of reorganization communication only depends on the effective elements in the intermediate results, rather than the accumulation of effective elements among all the RHS array operands. For this reason, with respect to a single FORALL assignment, the fewer the number of the distinct groups of compatible alignment matrices among all the RHS array operands, the less


```

(1) for each node  $A$  in  $V_a$ 
(2)    $T = \phi$ 
(3)   for each neighboring node  $s_k$  (in  $V_s$ ) of  $A$ 
(4)     for each element  $D_B F_{B,k}$  in  $Q_k$ 
(5)        $T = T \cup D_B F_{B,k} F_{A,k}^{-1}$ 
(6)     endfor
(7)   endfor
(8)   Choose  $D_A = D_B F_{B,k} F_{A,k}^{-1}$  such that  $D_B F_{B,k} F_{A,k}^{-1}$  is the member
      in  $T$  which has the maximum accumulated-weight
(9)   for each neighboring node  $s_k$  (in  $V_s$ ) of  $A$ 
(10)    if  $D_A F_{A,k}$  is not in  $Q_k$  then
(11)       $Q_k = Q_k \cup \{D_A F_{A,k}\}$ 
(12)    endif
(13)  endfor
(14) endfor

```

Figure 3.16. The alignment graph base alignment algorithm

the cost of reorganization communication. In Figure 3.16, Q_k records the distinct groups of compatible alignment matrices among all operands in statement s_k (lines (10)-(12)). When array A is referenced in statement s_k , we hope that $D_A F_{A,k}$ can be chosen in the way that it matches to another element $D_B F_{B,k}$ existing in Q_k and the size of Q_k is not increased. In other words, D_A should be compatible with D_B . Therefore, no extra reorganization communication occurs since A and B can be evaluated together and the communication cost of moving intermediate results is equal to that of moving the operand B . On the other hand, however, array A may be referenced in multiple statements and the compatibility requirements imposed by different **FORALL** assignment may conflict with one another. T collects all the types of different $D_B F_{B,k} F_{A,k}^{-1}$ if both array A and array B are referenced in **FORALL** assignment s_k . Note that T may contain multiple identical elements each of which comes from a distinct **FORALL** assignment. As a result, D_A is chosen to be the member in T which

has the maximum accumulated-weight (line (8)). The accumulated weight refers to the weight accumulated among all identical elements. Thus, the maximum amount of reorganization communication has been eliminated.

Table 3.2 illustrates how the AGBA algorithm works using Example 9. Table 3.2 shows the content of each Q_k at each step of the outmost loop between lines (1)-(14) in Figure 3.16. The algorithm begins with A . After executing lines (9)-(13), Q_1 , Q_2 , and Q_3 become $\{D_A F_{A,1}\}$, $\{D_A F_{A,2}\}$, and $\{D_A F_{A,3}\}$, respectively. $F_{A,1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ and $F_{A,2} = F_{A,3} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. Next, B is selected. After executing lines (3)-(7), $T = \{D_A F_{A,1} F_{B,1}^{-1}, D_A F_{A,2} F_{B,2}^{-1}, D_A F_{A,3} F_{B,3}^{-1}\}$. Since $F_{B,1} = F_{B,2} = F_{B,3} = I$, $T = \{D_A, D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\}$. Consequently, $D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ is the member in T which has the larger accumulated-weight. Thus, $D_B = D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ (line(8)). Continuing in the algorithm, D_X and D_Y are likewise obtained.

Table 3.2. Resolving base alignment for Example 9

	alignment matrices	Q_1	Q_2	Q_3
A		D_A	$D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$	$D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
B	$D_B = D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$	$D_A, D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$	$D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$	$D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
X	$D_X = D_A$	$D_A, D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$	$D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$	$D_A, D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
Y	$D_Y = D_A$	$D_A, D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$	$D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$	$D_A, D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$

In Figure 3.16, the order in which V_a nodes are visited (line (1)) is important in minimizing reorganization communication. An efficient heuristic approach can be

addressed as follows. Let U be a subset of V_a . A node in V_a is a *neighboring node* of U if this node is connected to a node in U . Initially, U is empty. The algorithm begins with a node in V_a , say A , which has the maximum connectivity degree. $U = U \cup \{A\}$. Then, in each of the rest steps, a node in $V_a - U$, say B , is selected such that B is connected to a neighboring node of U , say s_k , where s_k has the minimum connectivity degree among those neighboring nodes of U . $U = U \cup \{B\}$. This procedure repeats until U contains every node in V_a .

If the heap-sort algorithm [78] is used in line (6), the time complexity of finding the element with the maximum accumulated-weight would be reduced to $O(\log|E|)$ where $|E|$ is the number of edges in AG $G = (V, E)$. As a result, the time complexity of the AGBA algorithm is $O(|E|\log|E|)$.

3.5.4 Experimental Results

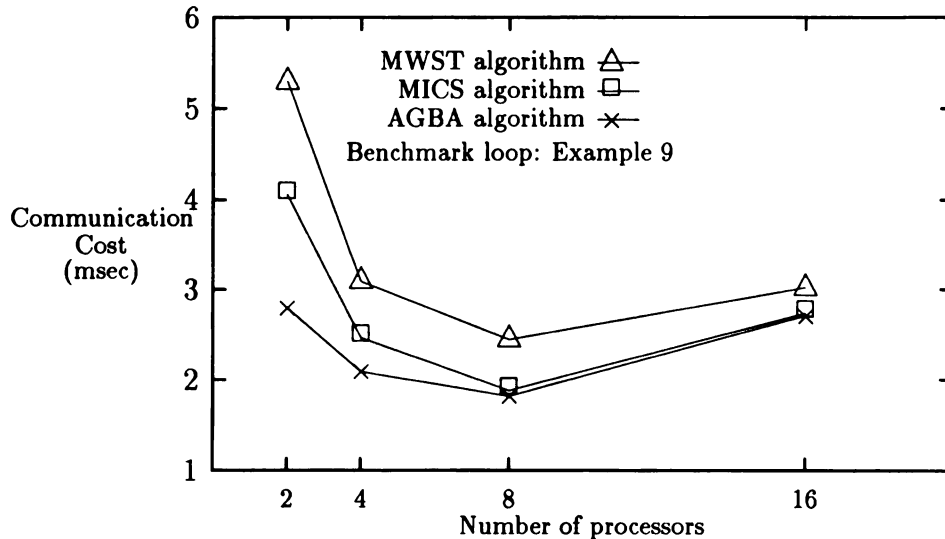


Figure 3.17. Comparison of the MWST, MICS, and AGBA algorithms on 16-node nCUBE-2

Figure 3.17 shows the comparison of communication cost among the MWST, MICS, and AGBA algorithms on 16-node nCUBE-2. Example 9 (Figure 3.14) is used as the benchmark loop in our experiment. We increase the iteration space in Example 9 by allocating the loop boundary as $(i_1 = 0 : 24, i_2 = 0 : 24)$. The AGBA algorithm outperforms other two algorithms. In Figure 3.11, when the number of processors increases, the number of the messages sent out from each processor increases, while the size of each message decreases. The startup software latency is much expensive than the network latency in message transmission on Purdue nCUBE-2. Therefore, the startup latency dominates the overall communication latency when the message size is relatively small. Since the amount of communication generated by the proposed MICS and AGBA algorithms is small comparing with the MWST algorithm, both the performance of MICS algorithm and the performance of AGBA algorithm are dominated by the startup latency and getting close each other after the number of processors exceeds 8.

3.6 Avoiding Redundant Communication

3.6.1 Redundant Communication

The same context of array elements may be referenced in more than one **FORALL** assignment. If these elements reside on remote processors, a local processor should receive a single copy of these elements rather than multiple identical copies. This technique is known as *redundant communication avoidance*. The identification of redundant communication can not only reduce communication overhead following up the data decomposition phase, but also have great impact in the alignment analysis. In this section, we focus on the issues of avoiding redundant reorganization communication.

In Example 10 (Figure 3.18), assume that $D_B = D_Z = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} D_A$. As a result, $D_B F_{B,2} \neq D_A F_{A,2}$ and $D_Z F_{Z,3} \neq D_A F_{A,3}$. In other words, remote element $A(i_1+i_2, i_1)$ is referenced in both statements s_2 and s_3 . However, $D_B = D_Z$ implies that the LHS element $B(i_1, i_2)$ in statement s_2 and the LHS element $Z(i_1, i_2)$ in statement s_3 reside on the same local processor. $A(i_1 + i_2, i_1)$ is required to be sent from the same remote processor to the same local processor in executing both statements s_2 and s_3 . However, since A is not written between statement s_2 and s_3 , the message transmission of remote element $A(i_1 + i_2, i_1)$ in statement s_2 is *redundant* with that of remote element $A(i_1 + i_2, i_1)$ in statement s_3 . These two messages transmitting the same $A(i_1 + i_2, i_1)$ comprise *redundant communication*.

```

FORALL( $i_1 = 0 : 10, i_2 = 0 : 10$ )
  WHERE( $X(i_1, i_2).NE.0$ )
    !HPF the probability is 83%
 $s_1:$        $X(i_1, i_2) = A(i_1, i_2)$ 
 $s_2:$        $B(i_1, i_2) = A(i_1 + i_2, i_1)$ 
 $s_3:$        $Z(i_1, i_2) = A(i_1 + i_2, i_1) * B(i_1, i_2)$ 
  END WHERE
 $s_4:$        $A(i_1, i_2) = Z(i_1, i_2)$ 
 $s_5:$        $B(i_1, i_2) = X(i_1, i_2) + A(i_1, i_2)$ 
END FORALL

```

Figure 3.18. Example 10: An Oceanwater benchmark loop

The identification of redundant communication is resorted to both the location and the context. The location requires that the senders of redundant communication must be the same, so do the receivers. The location requirement can be judged by alignment matrices. In Example 10, since $D_B F_{B,2} = D_Z F_{Z,2}$, elements $B(i_1, i_2)$ and $Z(i_1, i_2)$ are

always owned by the same local processor. The context requires that the context of every redundant message must be identical. In Example 10, array A is not defined between references “ $B(i_1, i_2) \leftarrow A(i_1 + i_2, i_1)@s_2$ ” and “ $Z(i_1, i_2) \leftarrow A(i_1 + i_2, i_1)@s_3$ ”. Therefore, two messages transmitting the same context of $A(i_1 + i_2, i_1)$ are redundant. Consider references “ $X(i_1, i_2) \leftarrow A(i_1, i_2)@s_1$ ” and “ $B(i_1, i_2) \leftarrow A(i_1, i_2)@s_5$ ”. The messages transmitting remote element $A(i_1, i_2)$ are not redundant because array A is re-defined in statement s_4 and the context of $A(i_1, i_2)$ referenced in statement s_2 is different from the context of $A(i_1, i_2)$ referenced in statement s_5 .

The concept of *single assignment block* is used to identify the distinct contexts with regard to the same array variable.

Definition 3.7 *Given a program structure, a single assignment block of array A , denoted as S_A , contains a block of statements such that A is only defined in the first statement in S_A and A is used in the other statements in S_A .*

In Example 10, there are two single assignment blocks with regard to $A : \{s_1, s_2, s_3\}$ and $\{s_4, s_5\}$. B has two single assignment blocks: $\{s_2, s_3\}$ and $\{s_5\}$. Z has one single assignment block: $\{s_3, s_4\}$. The single assignment block is based on the *def/use* flow for array variables. Therefore, the identification of a single assignment block can be easily obtained using data flow analysis algorithms [63]. The single assignment block has the following important property:

Proposition 4 *The context of an array element is not changed within the same single assignment block. The contexts of the same array element in different single assignment blocks are different.*

3.6.2 Enhanced Alignment Graph

An alignment graph can be enhanced to model the *use/def* flow for array variables by using the concept of single assignment block. The definition of an enhanced

alignment graph (EAG) $G = (V_a, V_s, E, A_s)$ is similar to that of an alignment graph $G = (V_a, V_s, E)$ except that an EAG has an extra arc set A_s . The definition of V_a , V_s , and E can be found in Section 3.5. There is an arc in set A_s from node s_ℓ to node s_k if an array defined in statement s_ℓ is used in statement s_k . Figure 3.19 shows the EAG for Example 10 (Figure 3.18).

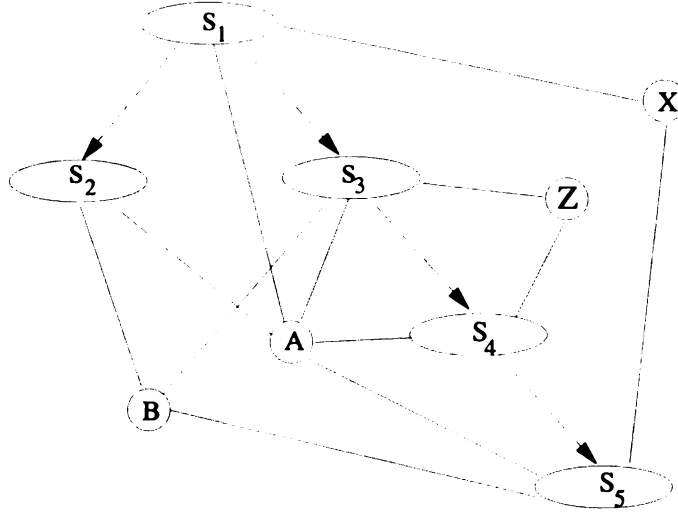


Figure 3.19. Enhanced alignment graph for Example 10

3.6.3 EAG Base Alignment Algorithm

The EAG-based base alignment (EAGBA) algorithm is shown in Figure 3.20. In an EAG $G = (V_a, V_s, E, A_s)$, there is a set, denoted as Q_k , associated with each node s_k in V_s . Initially, each set Q_k is empty. During the execution, each Q_k will contain elements of type $D_A F_{A,k}$. Each element of Q_k , $D_A F_{A,k}$, is further weighted by the number of effective elements in array A referenced in assignment s_k .

In Figure 3.20, lines (3)-(13) indicate that the same element $D_B F_{B,k} F_{A,k}$ is only included once in T for each single assignment block regarding to B . Therefore, redun-


```

(1) for each node  $A$  in  $V_a$ 
(2)    $T = \phi$ 
(3)   for each neighboring node  $s_k$  (in  $V_s$ ) of  $A$ 
(4)     for each  $D_B F_{B,k}$  in  $Q_k$ 
(5)       if  $D_B F_{B,k} F_{A,k}^{-1}$  is not in  $T$  then
(6)          $T = T \cup D_B F_{B,k} F_{A,k}^{-1}$ 
(7)       else if  $s_k$  is in a new single assignment block for  $B$  then
(8)          $T = T \cup D_B F_{B,k} F_{A,k}^{-1}$ 
(9)       else if there exists an element  $q$  in  $Q_k$  such that
            $q = D_B F_{B,k} F_{A,k}^{-1}$  but some effective elements of  $B$ 
           are not included in  $q$ 's effective domain then
(10)        Merge those new effective elements of  $B$  to  $q$ 's effective domain
(11)      endif
(12)    endfor
(13)  endfor
(14)  Choose  $D_A = D_B F_{B,k} F_{A,k}^{-1}$  such that  $D_B F_{B,k} F_{A,k}^{-1}$  is the member
       in  $T$  which has the maximum accumulated-weight
(15)  for each neighboring node  $s_k$  (in  $V_s$ ) of  $A$ 
(16)    if  $D_A F_{A,k}$  is not in  $Q_k$  then
(17)       $Q_k = Q_k \cup \{D_A F_{A,k}\}$ 
(18)    endif
(19)  endfor
(20) endfor

```

Figure 3.20. The enhanced alignment graph base alignment algorithm

dant communication will not be counted in line (14) and the selection of alignment matrix D_A will be immuned from the adverse impact of redundant communication. This feature makes the EAGBA algorithm superior to the AGBA algorithm. Note that the same array variable may be referenced in different **FORALL** assignments within the same single assignment block regarding to that particular array variable. The effective domains imposed by those different **FORALL** assignment loop boundaries can be different. For this reason, different effective domains for the same type of the element in \mathbb{T} need to be combined (lines (9)-(10)). The array expression operation proposed in [64] can be used to combine different effective domains. The single assignment block can be found by using data flow analysis regarding to array variables [80, 81].

Example 10 is used to illustrate the basic idea. In the EAG for Example 10 (Figure 3.19), A has the maximum degree of the connectivity. Therefore, A is selected first in line (1) of the EAG-based algorithm (Figure 3.20). Executing lines (13)-(17), we have $Q_1 = \{D_A\}$, $Q_2 = \{D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\}$, $Q_3 = \{D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\}$, $Q_4 = \{D_A\}$, and $Q_5 = \{D_A\}$. Assume that B is selected next. Since B is only referenced in statements s_2 , s_3 , and s_4 , only Q_2 , Q_3 , and Q_4 are considered in determining \mathbb{T} in lines (3)-(11). Since statements s_2 and s_3 are in the same single assignment block regarding to A , $D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ is only included once in \mathbb{T} . Therefore,

$\mathbb{T} = \{D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, D_A\}$. The weight associated with D_A is 121 (in statement s_4),

and the weight associated with $D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ is 100 (in statement s_2). Therefore, by line (12), D_B is chosen such that $D_B = D_A$. In contrast, if the AGBA algorithm is used, $\mathbb{T} = \{D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, D_A\}$. Since the weight associated with

each $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} D_A$ is 100, the accumulated-weight over these two identical elements $D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ in T is 200, which is larger than 121, the weight of D_A . As a result, D_B would be chosen as $D_B = D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ instead of D_A . The AGBA algorithm replicates the cost of redundant reorganization communication and makes the wrong decision in base alignment. As shown in Table 3.3, continuing in the algorithm, D_X and D_Z are likewise obtained.

Table 3.3. Resolving base alignment for Example 10

result	A	B	X	Z
		$D_B = D_A$	$D_X = D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$	$D_Z = D_A$
Q ₁	D_A	D_A	$D_A, D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$	$D_A, D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
Q ₂	$D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$	$D_A, D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$	$D_A, D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$	$D_A, D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
Q ₃	$D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$	$D_A, D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$	$D_A, D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$	$D_A, D_A \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
Q ₄	D_A	D_A	D_A	D_A
Q ₄	D_A	D_A	D_A	D_A

The time complexity of the EAGBA algorithm is the same as that of the AGBA algorithm. In general, the size of T should be smaller in the EAGBA algorithm than that in the AGBA algorithm. Therefore, less searching time would be taken in line (14) of the EAGBA algorithm.

CHAPTER 4

Offset Alignment

In the base alignment phase, an array can be partitioned in a family of parallel hyperplanes. Offset alignment determines the displacement of each hyperlane with respect to the template. More precisely, the alignment offset \vec{d}_A for each array A is resolved in the offset alignment phase. This chapter provides a mathematical framework to address communication cost in offset alignment. The impact of access offset is accurately represented by piecewise linear cost function. Efficient offset alignment algorithms are proposed to be incorporated with the RHS expression evaluation optimization and redundant communication avoidance.

4.1 Offset Alignment for Single Reference

Similar to labeling access matrix, if an instance $A(F\vec{i} + \vec{f})$ is referenced in a statement s_k , we want to label the access offset \vec{f} by the array variable A and the statement number k . However, multiple instances of the same array may be referenced in the same **FORALL** assignment. Unlike the base alignment analysis, the single occurrence transformation will change the meaning of access offset in the transformed program (details discussed in Section 4.3). For this reason, this section only considers the case where each array has no more than one type of instance referenced in each **FORALL**

assignment. Let $\vec{f}_{A,k}$ be the access offset for the instance of array A referenced in **FORALL** assingment s_k .

4.1.1 Offset Alignment Equation

Consider reference “ $X \leftarrow Y@s_2$ ” in Example 3 (Figure 2.8). Assume that in iteration $\begin{pmatrix} i_1 \\ i_2 \end{pmatrix}$, $X(x_1, x_2)$ is referenced on the LHS and $Y(y_1, y_2)$ is referenced on the RHS.

We want to map $\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ and $\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ to the same template element in order to avoid interprocessor communication. In other words,

$$\delta_X\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \delta_Y\left(\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}\right)$$

By the definition of alignment function, the above equation can be re-written as follows:

$$D_X\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) + \vec{d}_X = D_Y\left(\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}\right) + \vec{d}_Y \quad (4.1)$$

On the other hand, by the definition of access function, we have

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = F_{X,2}\left(\begin{pmatrix} i_1 \\ i_2 \end{pmatrix}\right) + \vec{f}_{X,2} \quad (4.2)$$

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = F_{Y,2}\left(\begin{pmatrix} i_1 \\ i_2 \end{pmatrix}\right) + \vec{f}_{Y,2} \quad (4.3)$$

where $F_{X,2} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$, $F_{Y,2} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $\vec{f}_{X,2} = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$, and $\vec{f}_{Y,2} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$. Substituting Equation 4.2 and Equation 4.3 into Equation 4.1, we get

$$D_X F_{X,2} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} + D_X \vec{f}_{X,2} + \vec{d}_X = D_Y F_{Y,2} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} + D_Y \vec{f}_{Y,2} + \vec{d}_Y$$

To satisfy the above equation, the following two equations must hold.

$$D_X F_{X,2} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} = D_Y F_{Y,2} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} \quad (4.4)$$

$$D_X \vec{f}_{X,2} + \vec{d}_X = D_Y \vec{f}_{Y,2} + \vec{d}_Y \quad (4.5)$$

Equation 4.4 implies that D_X and D_Y must be compatible with respect to reference “ $X \leftarrow Y@s_2$ ”. Equation 4.1 cannot hold if Equation 4.4 is not satisfied. This implies that offset alignment should be studied only if D_X and D_Y are compatible with respect to reference “ $X \leftarrow Y@s_2$ ”. In general, offset alignment is only studied among those references which are free of reorganization communication. Equation 4.5 specifies the relationship between alignment offsets \vec{d}_X and \vec{d}_Y in order to make reference “ $X \leftarrow Y@s_2$ ” free of interprocessor communication.

As specified in section 2.4, let $D_X = (0,1)$ and $D_Y = (1,1)$. This solution of D_X and D_Y satisfies Equation 4.4. Substituting the values of D_X and D_Y into Equation 4.5, we get

$$(0,1) \begin{pmatrix} 0 \\ 2 \end{pmatrix} + \vec{d}_X = (1,1) \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \vec{d}_Y$$

In other words, $\vec{d}_Y - \vec{d}_X = (2)$. For simplicity, we choose $\vec{d}_Y = (2)$ and $\vec{d}_X = (0)$. Therefore, reference “ $X \leftarrow Y@s_2$ ” is free of interprocessor communication.

Equation 4.5 can be easily extended to the following property for any multi-dimensional arrays.

Proposition 5 *In reference “ $A(F_{A,k}\vec{i} + \vec{f}_{A,k}) \leftarrow B(F_{B,k}\vec{i} + \vec{f}_{B,k})@s_k$ ”, suppose that alignment matrices D_A and D_B are given such that $D_A F_{A,k} = D_B F_{B,k}$. The reference is free of interprocessor communication if and only the following equation holds.*

$$D_A \vec{f}_{A,k} + \vec{d}_A = D_B \vec{f}_{B,k} + \vec{d}_B \quad (4.6)$$

Given reference “ $A \leftarrow B@s_k$ ”, if the values of \vec{d}_A and \vec{d}_B cannot satisfy Equation 4.6, \vec{d}_A and \vec{d}_B are called to be *mismatched* each other regarding to that reference.

4.1.2 Multiple Aligned Base Groups

In Example 3 (Figure 2.8), there is only one aligned-base group. In this section, we study how to resolve offset alignment for multiple aligned-base groups.

Consider Example 11 in Figure 4.1.

```

FORALL( $i_1 = 2 : n_1 - 2, i_2 = 0 : n_2 - 1$ )
 $s_1:$     $W(i_1 + 1, i_2) = Z(i_2 - 2, i_1)$ 
END FORALL

```

Figure 4.1. Example 11: An Eispack benchmark loop

Suppose that base alignment in Example 11 is pre-determined as follows. Both arrays W and Z are distributed in row dimension and column dimension. The row dimension of W is aligned with the column dimension of Z . The column dimension

of W is aligned with the row dimension of Z . Therefore, $D_W = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ and $D_Z = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. Since $F_{W,1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ and $F_{Z,1} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, it is easy to verify that Equation 3.7 holds for such D_W and D_Z . In other words,

$$D_W F_{W,1} = D_Z F_{Z,1}$$

Therefore, by Proposition 5, reference “ $W(i_1 + 1, i_2) \leftarrow Z(i_2 - 2, i_1) @ s_1$ ” is free of interprocessor communication, if the following equation holds

$$D_W \vec{f}_{W,1} + \vec{d}_W = D_Z \vec{f}_{Z,1} + \vec{d}_Z \quad (4.7)$$

where $\vec{f}_{W,1} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\vec{f}_{Z,1} = \begin{pmatrix} -2 \\ 0 \end{pmatrix}$.

Alignment matrices D_W and D_Z can be re-written as $\begin{pmatrix} D_{1,W} \\ D_{2,W} \end{pmatrix}$ and $\begin{pmatrix} D_{1,Z} \\ D_{2,Z} \end{pmatrix}$ where $D_{1,W} = (1, 0)$, $D_{2,W} = (0, 1)$, $D_{1,Z} = (0, 1)$, and $D_{2,Z} = (1, 0)$. Note that $D_{1,W}$ and $D_{2,W}$ are two distribution bases of W . $D_{1,Z}$ and $D_{2,Z}$ are two distribution bases of Z . Let $\vec{d}_W = \begin{pmatrix} d_{1,W} \\ d_{2,W} \end{pmatrix}$ and $\vec{d}_Z = \begin{pmatrix} d_{1,Z} \\ d_{2,Z} \end{pmatrix}$. Let $\vec{f}_{W,1} = \begin{pmatrix} f_{1,W,1} \\ f_{2,W,1} \end{pmatrix}$ and $\vec{f}_{Z,1} = \begin{pmatrix} f_{1,Z,1} \\ f_{2,Z,1} \end{pmatrix}$. Equation 4.7 can be re-written as follows.

$$\begin{pmatrix} D_{1,W} \\ D_{2,W} \end{pmatrix} \begin{pmatrix} f_{1,W,1} \\ f_{2,W,1} \end{pmatrix} + \begin{pmatrix} d_{1,W} \\ d_{2,W} \end{pmatrix} = \begin{pmatrix} D_{1,Z} \\ D_{2,Z} \end{pmatrix} \begin{pmatrix} f_{1,Z,1} \\ f_{2,Z,1} \end{pmatrix} + \begin{pmatrix} d_{1,Z} \\ d_{2,Z} \end{pmatrix}$$

The above equation holds if and only if the following two equations hold.

$$D_{1,W} \begin{pmatrix} f_{1,W,1} \\ f_{2,W,1} \end{pmatrix} + d_{1,W} = D_{1,Z} \begin{pmatrix} f_{1,Z,1} \\ f_{2,Z,1} \end{pmatrix} + d_{1,Z} \quad (4.8)$$

$$D_{2,W} \begin{pmatrix} f_{1,W,1} \\ f_{2,W,1} \end{pmatrix} + d_{2,W} = D_{2,Z} \begin{pmatrix} f_{1,Z,1} \\ f_{2,Z,1} \end{pmatrix} + d_{2,Z} \quad (4.9)$$

Equations 4.8 and 4.9 reveal two important facts. First, Equation 4.8 indicates that distribution base $(1, 0)$ of W (represented by $D_{1,W}$) and distribution base $(0, 1)$ of Z (represented by $D_{1,Z}$) comprise an aligned-base group. Equation 4.9 indicates that distribution base $(0, 1)$ of W (represented by $D_{2,W}$) and distribution base $(1, 0)$ of Z (represented by $D_{2,Z}$) comprise another aligned-base group. In general, if two distribution bases of two various arrays are aligned, these two distribution bases must satisfy Equation 3.7.

Second, the solution of $d_{1,W}$ and $d_{1,Z}$ in Equation 4.8 is completely independent to the solution of $d_{2,W}$ and $d_{2,Z}$ in Equation 4.9. The values of $d_{1,W}$ and $d_{1,Z}$ are determined with regard to the first aligned-base group. The values of $d_{2,W}$ and $d_{2,Z}$ are determined with regard to the second aligned-base group.

On substitution of $D_{1,W}$, $D_{1,Z}$, $D_{2,W}$, $D_{2,Z}$, $\begin{pmatrix} f_{1,W,1} \\ f_{2,W,1} \end{pmatrix}$, $\begin{pmatrix} f_{1,Z,1} \\ f_{2,Z,1} \end{pmatrix}$, $\begin{pmatrix} f_{1,W,1} \\ f_{2,W,1} \end{pmatrix}$, and $\begin{pmatrix} f_{1,Z,1} \\ f_{2,Z,1} \end{pmatrix}$ by their values, Equations 4.8 and 4.9 can be re-written as follows.

$$d_{1,W} + 1 = d_{1,Z} - 2$$

$$d_{2,W} = d_{2,Z}$$

Therefore, reference “ $W(i_1+1, i_2) \leftarrow Z(i_2-2, i_1)@s_1$ ” is free of interprocessor commu-

nication if $\vec{d}_W = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$ and $\vec{d}_Z = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$. Substituting the values of two alignment offsets in the alignment functions, we get

$$\begin{pmatrix} t_1 \\ t_2 \end{pmatrix} = \delta_W \left(\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} \right) = D_W \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} + \vec{d}_W = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \end{pmatrix} = \begin{pmatrix} w_1 - 1 \\ w_2 \end{pmatrix} \quad (4.10)$$

$$\begin{pmatrix} t_1 \\ t_2 \end{pmatrix} = \delta_Z \left(\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} \right) = D_Z \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} + \vec{d}_Z = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} + \begin{pmatrix} 2 \\ 0 \end{pmatrix} = \begin{pmatrix} z_2 - 2 \\ z_1 \end{pmatrix} \quad (4.11)$$

where $T(t_1, t_2)$ is a template element. Alignment functions 4.10 and 4.11 indicate that the $(k+1)$ -th row of W is aligned with the $(k-2)$ -th column of Z , and the ℓ -th column of W is aligned with the ℓ -th row of Z .

Since offset alignment can be analyzed independently with each aligned-base group, the examples used in the rest of this chapter will contain only one aligned-base group. Moreover, the template is only a one-dimensional array for a data decomposition pattern which employs a single aligned-base group. Therefore, the vector of alignment offset, \vec{d} , will contain only one element. For this reason, \vec{d}_A will be simplified as integer d_A in the rest of this chapter.

4.1.3 Calculating Alignment Offset

The value of alignment offset can be decimal. Consider Example 12 (Figure 4.2). The base alignment analysis in Example 12 is identical to that in Example 5. Therefore,

Example 12 also has

$$D_A = (0, \frac{1}{2})$$

$$D_B = (0, \frac{1}{3})$$

```

FORALL( $i_1=0:n/2-1, i_2=1:n/3-2$ )
 $s_1:$      $B(2i_1, 3i_2 - 1) = A(i_1, 2i_2 + 1)$ 
 $s_2:$      $A(i_1, 2i_2) = B(i_1, 3i_2)$ 
END FORALL

```

Figure 4.2. Example 12: A Weather-Climate benchmark loop

By Proposition 5, **FORALL** assignment s_1 is free of neighboring communication if

$$D_A \vec{f}_{A,1} + d_A = D_B \vec{f}_{B,1} + d_B$$

where $\vec{f}_{A,1} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ and $\vec{f}_{B,1} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$. The above equation can be re-written as

$$\frac{1}{2} + d_A = -\frac{1}{3} + d_B$$

Therefore, we choose $d_A = -\frac{1}{2}$ and $d_B = \frac{1}{3}$ to avoid interprocessor communication.

The alignment functions of A and B can be specified as follows.

$$\delta_A\left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}\right) = \frac{1}{2}a_2 - \frac{1}{2}$$

$$\delta_B\left(\begin{pmatrix} b_1 \\ b_2 \end{pmatrix}\right) = \frac{1}{3}b_2 + \frac{1}{3}$$

4.2 The Cost of Neighboring Communication

4.2.1 The Basic Cost

Consider reference “ $X(i_1, i_1 + i_2 + 2) \leftarrow Y(i_1, i_2)$ ” in Example 3 (Figure 2.8). Assume that in iteration $\begin{pmatrix} i_1 \\ i_2 \end{pmatrix}$, $X(x_1, x_2)$ is referenced on the LHS and $Y(y_1, y_2)$ is referenced on the RHS. Therefore, we have

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = F_{X,2} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} + \vec{f}_{X,2} \quad (4.12)$$

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = F_{Y,2} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} + \vec{f}_{Y,2} \quad (4.13)$$

We assume that element $X(x_1, x_2)$ is mapped to the template element $T(t_X)$ and element $Y(y_1, y_2)$ is mapped to the template element $T(t_Y)$. Therefore, alignment functions for X and Y can be written as follows:

$$t_X = D_X \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + d_X$$

$$t_Y = D_Y \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} + d_Y$$

Hence, if the communication-free requirement cannot be honored, the misalignment between X and Y can be evaluated based on the difference between t_X and t_Y . On

substitution of $\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ from Equation 4.12 and $\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ from Equation 4.13, we get

$$t_X - t_Y = (D_X F_{X,2} - D_Y F_{Y,2}) \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} + (D_X \vec{f}_{X,2} + d_X) - (D_Y \vec{f}_{Y,2} + d_Y)$$

Since offset alignment is only studied for references which are free of reorganization communication, we only consider those values of D_X and D_Y such that $D_X F_{X,2} = D_Y F_{Y,2}$. For example, $D_X = (0, 1)$ and $D_Y = (1, 1)$. On substitution of $D_X = (0, 1)$ and $D_Y = (1, 1)$, the above equation can be re-written as follows:

$$t_X - t_Y = 2 + d_X - d_Y \quad (4.14)$$

Note that the value of $t_X - t_Y$ is independent from $\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ or $\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$.

In this thesis, a segment distribution is used to distribute the template elements across the processors. Like block distribution, the basic idea of segment distribution is to assign template elements consecutively to processors. However, unlike block distribution, the length of segment owned by each processor can be varied. The detailed description of segment distribution will be found in Chapter 5.

Assume that template is declared as $T(0 : n - 1)$ and the number of available processors is q . $T(0 : n - 1)$ is partitioned into q segments $T(m_i : m_{i+1} - 1)$ where $0 = m_0 < m_1 < m_2 < \dots < m_{q-1} < m_q = n - 1$. Segment $T(m_i : m_{i+1} - 1)$ is owned by processor $P(i)$. Therefore, by the owner-computes rule, processor $P(i)$ will work on the X elements which are mapped to $T(m_i : m_{i+1} - 1)$. By equation 4.14, we know that the Y elements accessed by processor $P(i)$ are mapped to $T(m_i - 2 - d_X + d_Y : m_{i+1} - 2 - d_X + d_Y - 1)$. As a result, those Y elements which are mapped to the portion

$T(m_{i+1} - 2 - d_X + d_Y : m_{i+1})$ *are remote to processor $P(i)$. For grand-challenge applications, the size of the template is usually large. Therefore, the remote portion $T(m_{i+1} - 2 - d_X + d_Y : m_{i+1})$ should be owned by processor $P(i + 1)$. As a result, this type of data shift communication from processor $P(i + 1)$ to processor $P(i)$ is classified as neighboring communication. For this reason, neighboring communication occurs if alignment offsets are mismatched.

Definition 4.1 *Suppose that alignment function of A and alignment function of B are pre-determined. Given reference “ $A \leftarrow B@s_k$ ”, the basic cost of neighboring communication, basic cost for short, is defined to be the number of remote template elements to which remote B elements are mapped in order to define A .*

In this above example, the basic cost is equal to $2 + d_X - d_Y$. This property can be formalized as follows:

Proposition 6 *In reference “ $A(F_{A,k}\vec{i} + \vec{f}_{A,k}) \leftarrow B(F_{B,k}\vec{i} + \vec{f}_{B,k})@s_k$ ”, suppose that alignment matrices D_A and D_B are given such that $D_A F_{A,k} = D_B F_{B,k}$. The basic cost of neighboring communication is equal to $|(D_A \vec{f}_{A,k} - D_B \vec{f}_{B,k}) + (d_A - d_B)|$.*

4.2.2 The Weight of the Basic Cost

The basic cost depends on the number of template elements to which the remote effective data array elements mapped. The basic cost is equal to the actual cost of neighboring communication if there is only one effective element of each data array mapped to each template element. However, in many cases, the map between array elements and template elements is many-to-one. If the number of effective elements mapped to a template element is called the *density* of the template element, the actual amount of neighboring communication should be the product of the density and the basic cost.

*Assume $2 + d_X - d_Y > 0$. For other cases, see Section 3.3.

A closer inspection of many benchmark programs reveals that the density of different template elements can be different. The density function, $\omega_{A,k}$, is defined such that the value of $\omega_{A,k}(\vec{t})$ is the number of effective elements in array A which are mapped to template element $T(\vec{t})$ with respect to **FORALL** assignment s_k . Consider **Example 3** (Figure 2.8). Suppose alignment functions of X and Y are defined as follows:

$$t = \delta_X \left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \right) = x_2 + 1$$

$$t = \delta_Y \left(\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \right) = y_1 + y_2$$

where $T(t)$ is a template element. Thus, X is partitioned in columns and Y is partitioned along diagonal. Alignment offsets $d_X = 1$ and $d_Y = 0$ imply that the $(k + 1)$ -th column in X is aligned with the k -th off-diagonal. By the definity of density function, we have

$$\omega_{X,2}(t) = \max\{0, t - 1\}$$

$$\omega_{Y,2}(t) = t$$

Since the density function is not uniform, the amount of neighboring communication between different pairs of neighboring processors is different. In this thesis, neighboring processors refer to processors $P(i)$ and $P(i + 1)$.

Figure 4.3 shows the amount of neighboring communication between each pair of neighboring processors. In Figure 4.3, arrays X and Y are declared as 8×8 and the number of available processors is 3. Each array element is represented by a circle. In each array, elements mapped to the same element in the template are connected by the same solid line. Only effective elements are covered by solid lines. By Proposition 6,

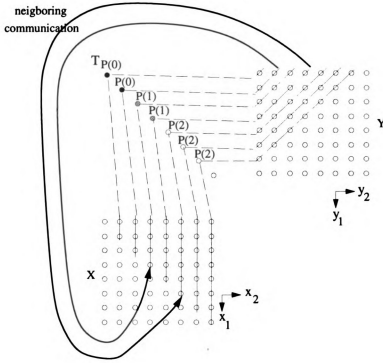


Figure 4.3. Neighboring communication in Example 3

the basic cost of neighboring communication is equal to 1. Figure 4.3 gives the real meaning which the value 1 stands for. In order to define the LHS X elements on a column, each processor requires to access remote Y elements which form one off-diagonal line. For example, in order to write elements $X(x_1, x_2)$ on line $x_2 = 5$, processor $P(2)$ accesses elements $Y(y_1, y_2)$ on line $y_1 + y_2 = 3$ which are owned by remote processor $P(1)$. Therefore, the amount of neighboring communication between processors $P(1)$ and $P(2)$ is equal to the number of elements on line $y_1 + y_2 = 3$, which is 4. Similarly, the amount of neighboring communication between processors $P(0)$ and $P(1)$ is equal to 2. Note that the amount of neighboring communication between $P(0)$ and $P(1)$ is less than that between $P(1)$ and $P(2)$. Since the messages for neighboring communication can be transmitted simultaneously among pairs of neighboring processors, the cost of neighboring communication among all processors should be equal to the maximum amount of neighboring communication per pair of

processors. Thus, the actual cost of neighboring communication is 4 in Figure 4.3. This justifies the reason that the basic cost can be used to measure the actual cost of neighboring communication.

Like the cost of reorganization communication, the cost of neighboring communication can be weighed by various array sizes. For example, in the following code segment, loop s_2 is a sequential loop and can not be parallelized among the processors.

```

ALIGN  $Y(j)$  WITH  $X(j)$ 
ALIGN  $Z(j)$  WITH  $X(j)$ 

FORALL( $i_1 = 0 : 10$ )
 $s_1$ :    $X(i_1) = Y(i_1 + 1)$ 
      FOR( $i_2 = 0 : 10$ )
 $s_2$ :    $X(i_1) = X(i_1) + Z(i_1 + 1, i_2)$ 
      END FOR
END FORALL

```

$Y(j)$ is aligned with $X(i)$, so is $Z(i)$. By Proposition 5, the basic cost of neighboring communication involved in reading $Y(i_1 + 1)$ is 1, and the basic cost of neighboring communication involved in reading $Z(i_1 + 1, i_2)$ is also 1. However, in the former case, the actual cost of neighboring communication is 1. In the later case, the actual cost of neighboring communication is 10 since the whole column in Z is mapped to the same template element.

In the rest of this Chapter, we use the weight of the basic cost to formalize the impact of the density function and different array sizes on the actual amount of neighboring communication imposed by a given offset alignment. The actual amount of neighboring communication will be represented by the so-called weighted basic cost.

4.3 The Impact of Access Offset

4.3.1 Piecewise Linear Cost Function

The following code pattern is very common in scientific application programs. Multiple instances of array Y referenced on the RHS have the same access matrix

```

FORALL( $i_1 = 0 : 10$ )
 $s_1$ :    $X(i_1) = Y(i_1) + Y(i_1 + 1) + Y(i_1 + 2)$ 
END FORALL

```

but different access offsets. The access offset of instance $X(i_1)$ is 0, the access offset of instance $X(i_1 + 1)$ is 1, and the access offset of instance $X(i_1 + 2)$ is 2. Using the notation from the previous section, assume that there are totally q processors and template elements $T(m_i + 1 : m_{i+1} - 1)$ are owned by the local processor $P(i)$ where $0 = m_0 < m_1 < m_2 < \dots < m_{q-1} < m_q = n - 1$. Since $t = \delta_X(x) = x + d_X$, the X elements owned by processor $P(i)$ are $X(m_i - d_X : m_{i+1} - 1 - d_X)$. Consequently, by the construction of statement s_1 , processor $P(i)$ requires to access elements $Y(m_i - d_X : m_{i+1} - 1 - d_X)$ with respect to instance $Y(i_1)$, $Y(m_i - d_X + 1 : m_{i+1} - d_X)$ with respect to instance $Y(i_1 + 1)$, and $Y(m_i - d_X + 2 : m_{i+1} - d_X + 1)$ with respect to instance $Y(i_1 + 2)$. On the other hand, since $t = \delta_Y(y) = y + d_Y$, the Y elements owned by processor $P(i)$ are $Y(m_i - d_Y : m_{i+1} - 1 - d_Y)$. Consider the remote Y elements accessed by processor $P(i)$.

If $d_X < d_Y$, remote elements $Y(m_{i+1} - d_Y : m_{i+1} - d_X)$ are accessed with respect to instance $Y(i_1)$. Remote elements $Y(m_{i+1} - d_Y : m_{i+1} - d_X + 1)$ are accessed with respect to instance $Y(i_1 + 1)$. Remote elements $Y(m_{i+1} - d_Y : m_{i+1} - d_X + 2)$ are accessed with respect to instance $Y(i_1 + 2)$. Elements in these three segments overlap.

The union of these three segments is $Y(m_{i+1} - d_Y : m_{i+1} - d_X + 2)$. Therefore, the cost of neighboring communication is equal to $d_Y - d_X + 2$ if $d_X < d_Y$.

If $d_X > d_Y + 2$, remote elements $Y(m_i - d_X : m_i - 1 - d_Y)$ are accessed with respect to instance $Y(i_1)$. Remote elements $Y(m_i - d_X + 1 : m_i - 1 - d_Y)$ are accessed with respect to instance $Y(i_1 + 1)$. Remote elements $Y(m_i - d_X + 2 : m_i - 1 - d_Y)$ are accessed with respect to instance $Y(i_1 + 2)$. The union of these three segments is $Y(m_i - d_X : m_i - 1 - d_Y)$. Therefore, the cost of neighboring communication is equal to $d_X - d_Y$ if $d_X > d_Y + 2$.

In particular, if $d_X = d_Y$, remote elements $Y(m_i - d_Y)$ and $Y(m_i - d_Y + 1)$ are accessed with respect to instances $Y(i_1 + 1)$ and $Y(i_1 + 2)$, respectively. If $d_Y + 1 = d_X$, remote elements $Y(m_i - d_X - 1)$ and $Y(m_i - d_Y)$ are accessed with respect to instances $Y(i_1)$ and $Y(i_1 + 2)$, respectively. If $d_Y + 2 = d_X$, remote elements $Y(m_i - d_X - 2)$ and $Y(m_i - d_X - 1)$ and $Y(m_i - d_Y)$ are accessed with respect to instances $Y(i_1)$ and $Y(i_1 + 1)$, respectively. Overall, under these three special conditions, the cost of neighboring communication is equal to 2.

Let $c_{B,k}$ be the cost of neighboring communication involved in accessing all instances of array B referenced on the RHS of statement s_k . In the above example, we have the following result.

$$c_{Y,1} = \begin{cases} 2 - (d_X - d_Y) & \text{when } (d_X - d_Y) < 0 \\ 2 & \text{when } 0 \leq (d_X - d_Y) \leq 2 \\ (d_X - d_Y) & \text{when } 2 < (d_X - d_Y) \end{cases} \quad (4.15)$$

Unlike the cost of reorganization communication, Equation 4.15 shows that the cost of neighboring communication cannot be treated independently for each distinct instance when there are multiple instances for the same array referenced in the same statement. In the above example, if the cost of neighboring communication regarding to instances $Y(i_1)$, $Y(i_1 + 1)$, and $Y(i_1 + 2)$ were estimated independently by

Proposition 5, the overall cost would be

$$|d_X - d_Y| + |d_X - d_Y + 1| + |d_X - d_Y + 2|$$

The value of the above equation would be three times as much as $c_{Y,1}$. For this reason, the single occurrence transformation is not used in the offset alignment analysis since it can change the cost estimation and thus affect the result of offset alignment. For ease of explanation, in the rest of this chapter, we use the symbol “ $A \leftarrow B@s_k$ ” to represent the references of all distinct instances of array B in order to write the LHS A elements in statement s_k . For the sake of consistency, the symbol “ $A \leftarrow B@s_k$ ” is still called a *reference*.

Equation 4.15 can be extended as follows:

Proposition 7 *Assume that in a FORALL assignment s_k , all instances of array B are referenced on the RHS. These instances are represented by $B(i_1 + r_1)$, $B(i_1 + r_2)$, ..., and $B(i_1 + r_\ell)$ where $r_1 < r_2 < \dots < r_\ell$. The access offset tuple is denoted as $u_{B,k} = \{r_1, r_2, \dots, r_\ell\}$, which is distinguished by the array variable B and the statement number k . $c_{B,k}$, the cost of neighboring communication generated by accessing all B instances in executing statement s_k , can be formalized by the following piecewise linear equation.*

$$c_{B,k} = \begin{cases} r_\ell - (d_A - d_B) & \text{when } (d_A - d_B) < r_1 \\ r_\ell - r_1 & \text{when } r_1 \leq (d_A - d_B) \leq r_\ell \\ (d_A - d_B) - r_1 & \text{when } r_\ell < (d_A - d_B) \end{cases} \quad (4.16)$$

where array A is referenced on the LHS. For simplicity, $c_{B,k}$ is called the cost function.

Proposition 7 can be easily extended to the case where the array referenced on the LHS has different instance(s) referenced on the RHS in the same statement. Assume that $A(i_1 + r_j)$ is referenced on the LHS and other instances of A referenced on the RHS are $A(i_1 + r_1)$, ..., $A(i_1 + r_{j-1})$, $A(i_1 + r_{j+1})$, ..., and $A(i_1 + r_\ell)$ where $r_1 <$

$\dots < r_{j-1} < r_j < r_{j+1} < \dots < r_\ell$. By using the similar analysis as in Equation 4.16, the cost of neighboring communication involved in accessing A elements is equal to $r_\ell - r_1$ no matter whatever d_A is. In other words, the cost is a constant which is independent with the alignment of array A .

4.3.2 Properties of Piecewise Linear Cost Function

In the rest of this section, we explore more properties of the piecewise linear cost function. Consider the following program structure. Assignment s_2 is free of neigh-

```

FORALL( $i_1 = 0 : 10$ )
 $s_1$ :    $X(i_1) = Y(i_1) + Y(i_1 + 1) + Y(i_1 + 2)$ 
 $s_2$ :    $Z(i_1) = X(i_1) * 2$ 
 $s_3$ :    $X(i_1) = X(i_1) + Y(i_1 + 4) + Y(i_1 + 5) + Y(i_1 + 7)$ 
      END IF
END FORALL

```

boring communication as long as $d_Z = d_X$. Consider the relationship of d_Y and d_X .

Since $u_{Y,1} = \{0, 1, 2\}$, we have

$$c_{Y,1} = \begin{cases} 2 - (d_X - d_Y) & \text{when } (d_X - d_Y) < 0 \\ 2 & \text{when } 0 \leq (d_X - d_Y) \leq 2 \\ (d_X - d_Y) & \text{when } 2 < (d_X - d_Y) \end{cases}$$

Since $u_{Y,3} = \{4, 5, 7\}$, we have

$$c_{Y,3} = \begin{cases} 7 - (d_X - d_Y) & \text{when } (d_X - d_Y) < 4 \\ 3 & \text{when } 4 \leq (d_X - d_Y) \leq 7 \\ (d_X - d_Y) - 4 & \text{when } 7 < (d_X - d_Y) \end{cases}$$

The task of offset alignment is to find the optimal value of $d_X - d_Y$ such that the sum of $c_{Y,1}$ and $c_{Y,3}$ is minimal. Figure 4.4 shows the sum of $c_{Y,1}$ and $c_{Y,3}$ in a coordinate system.

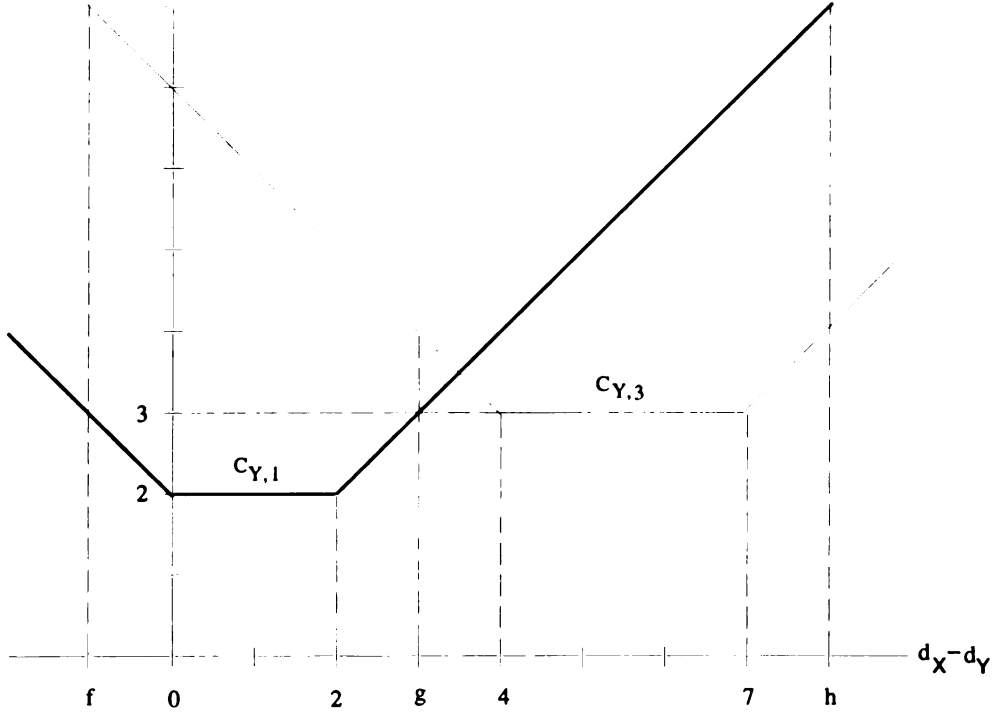


Figure 4.4. The sum of $c_{Y,1}$ and $c_{Y,3}$

Studying Figure 4.4, we obtain the following results. If the position of $d_X - d_Y$ is chosen between 2 and 4, such as the position g , the value of $c_{Y,1}$ is $(g - 2) + 2$ and the value of $c_{Y,3}$ is $(4 - g) + 3$. As a result, the sum of $c_{Y,1}$ and $c_{Y,3}$ is equal to $(4 - g) + (g - 2) + 2 + 3$, or $(4 - 2) + 2 + 3$. Note that $(4 - 2)$ is the distance between the maximum element in tuple $u_{Y,1}$ and the minimum element in tuple $u_{Y,3}$. If the position of $d_X - d_Y$ is chosen to the left of 2, such as the position f , the value of $c_{Y,1}$ is $(2 - f) + 2$ and the value of $c_{Y,3}$ is $(4 - f) + 3$. Since f extends left to 2, $(4 - f)$, the distance between f and 4, is greater than 2, the distance between 2 and 4. Therefore,

the sum of $(2 - f) + 2$ and $(4 - f) + 3$ is greater than $(4 - 2) + 2 + 3$. On the other hand, if the position of $d_X - d_Y$ is chosen to the right of 4, such as the position h , the value of $c_{Y,1}$ is $(h - 2) + 2$ and the value of $c_{Y,3}$ is $(h - 7) + 3$. Since h extends right to 4, $(h - 2)$, the distance between 2 and h , is greater than 2, the distance between 2 and 4. Overall, we conclude that the minimal value of $c_{Y,1} + c_{Y,3}$ is 7 when $d_X - d_Y$ is chosen as any integer between (including) 2 and (including) 4.

We further notice that the middle elements, 1 in $u_{Y,1}$ and 5 in $u_{Y,3}$, do not affect the shape of the piecewise linear function and thus are not relevant to the decision making for the minimal sum of the cost. For this reason, the minimum and maximum elements in an access offset tuple are sufficient to identify a piecewise linear cost function. For this reason, the cost function $c_{B,k}$ is denoted as $c_{B,k} = \langle l_{B,k}, r_{B,k} \rangle$ where $l_{B,k}$ is the minimum element and $r_{B,k}$ is the maximum element in $u_{B,k}$. Element $l_{B,k}$ is called the *left end point* and element $u_{B,k}$ is called then *right end point*. Both $l_{B,k}$ and $u_{B,k}$ are called *end points* for simplicity.

Using the similar analysis as in Figure 4.4, we can obtain the general results as follows.

Proposition 8 *Assume that $c_{B,j}$ is the cost function for accessing remote B elements in executing reference “ $A \leftarrow B@s_j$ ” and $c_{B,k}$ is the cost function for accessing remote B elements in executing reference “ $A \leftarrow B@s_k$ ”. Let $c_{B,j} = \langle l_{B,j}, r_{B,j} \rangle$ and $c_{B,k} = \langle l_{B,k}, r_{B,k} \rangle$. If $l_{B,j} < r_{B,k}$, the sum of $c_{B,j}$ and $c_{B,k}$ is minimized when $d_A - d_B$ is chosen as any integer between $r_{B,j}$ and $l_{B,k}$. The minimum value of the sum of $c_{B,j}$ and $c_{B,k}$ is equal to $r_{B,j} - l_{B,k}$.*

Figure 4.4 only shows the case where $r_{B,j} < l_{B,k}$. Proposition 8 also holds for both case $r_{B,j} < l_{B,k}$ and case $r_{B,j} > l_{B,k}$, as shown in Figure 4.5(a) and (b).

A close inspection of Figure 4.5 reveals that the sum of $c_{B,j}$ and $c_{B,k}$ can be represented by the piecewise linear function when $d_A - d_B$ varies between $l_{B,j}$ and

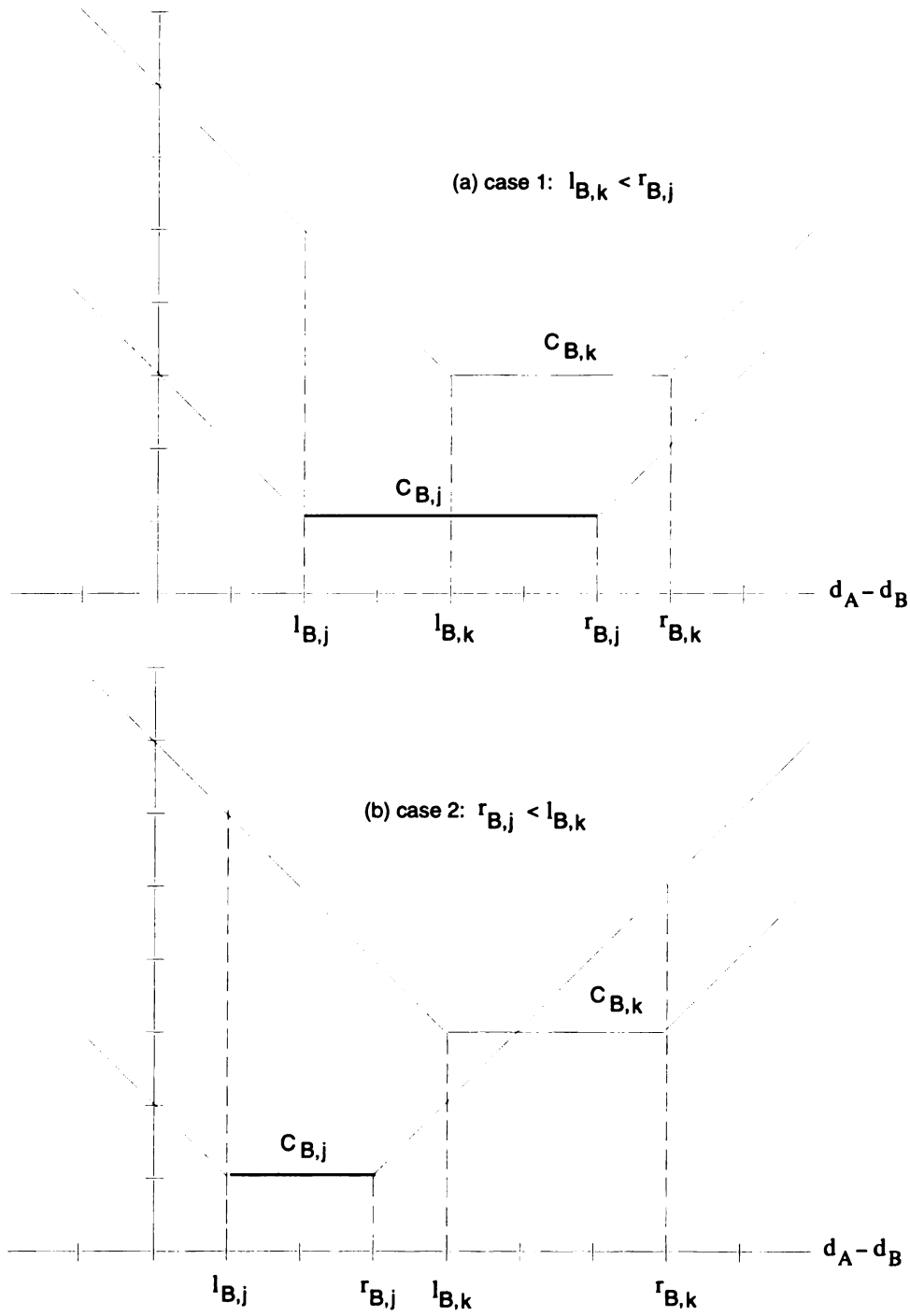


Figure 4.5. The sum of $c_{B,j}$ and $c_{B,k}$

f

7

E

F

fo

le

d

ar

er

po

$r_{B,k}$. This can be formalized as follows.

Proposition 9 Assume that $c_{B,j}$ is the cost function for accessing remote B elements in executing reference “ $A \leftarrow B@s_j$ ” and $c_{B,k}$ is the cost function for accessing remote B elements in executing reference “ $A \leftarrow B@s_k$ ”. Let $c_{B,j} = \langle l_{B,j}, r_{B,j} \rangle$ and $c_{B,k} = \langle l_{B,k}, r_{B,k} \rangle$. If $l_{B,j} < r_{B,k}$, the sum of $c_{B,j}$ and $c_{B,k}$ have the following property.

$$c_{B,j} + c_{B,k} = \begin{cases} (d_A - d_B) - l_{B,j} + (r_{B,k} - l_{B,k}) & \max\{r_{B,j}, l_{B,k}\} < (d_A - d_B) \leq r_{B,k} \\ r_{B,j} - l_{B,k} & \min\{r_{B,j}, l_{B,k}\} \leq (d_A - d_B) \leq \max\{r_{B,j}, l_{B,k}\} \\ r_{B,k} - (d_A - d_B) + (r_{B,j} - l_{B,j}) & l_{B,j} \leq (d_A - d_B) < \min\{r_{B,j}, l_{B,k}\} \end{cases}$$

For example, in Figure 4.4, the sum of $c_{Y,1}$ and $c_{Y,3}$ can be represented as the following piecewise linear function where the value of $d_X - d_Y$ varies between 0 and 7.

$$c_{Y,1} + c_{Y,3} = \begin{cases} 7 - (d_X - d_Y) + 2 & \text{when } 0 \leq (d_X - d_Y) < 2 \\ 7 & \text{when } 2 \leq (d_X - d_Y) \leq 4 \\ (d_X - d_Y) + 3 & \text{when } 4 < (d_X - d_Y) \leq 7 \end{cases}$$

Proposition 9 is important in determining offset alignment between arrays A and B such that the sum of the cost of neighboring communication with regard to multiple **FORALL** assignments is minimal. More precisely, the problem can be formalized as follows. Assume that there are m distinct references “ $A \leftarrow B@s_j$ ” where $c_{B,j} = \langle l_{B,j}, r_{B,j} \rangle$ for $1 \leq j \leq m$. The following algorithm can be used to find the value of $d_A - d_B$ such that $\sum_{j=1}^m c_{B,j}$ can be minimized.

The minimal-cost pairwise offset alignment (MPOA) algorithm is shown in Figure 4.6. In Figure 4.6, piecewise linear cost function $c_{B,\ell}$ is monotonously increasing when $d_A - d_B$ extends right to $r_{B,\ell}$ or left to $l_{B,\ell}$. Therefore, if $l_{B,j}$ is the minimal end point and $r_{B,k}$ is the maximum end point among all access offset tuples in T , each


```

(1)  $T = \bigcup_{j=1}^m \{ \langle l_{B,j}, r_{B,j} \rangle \}$ 
(2) while  $T$  contains more than one tuple do
(3)   Find  $j$  such that  $l_{B,j}$  has the minimal value among all end points in  $T$ 
(4)   Find  $k$  such that  $r_{B,k}$  has the maximum value among all end points in  $T$ 
(5)   if  $k = j$  then
(6)      $T = T - \{ \langle l_{B,j}, r_{B,j} \rangle \}$ 
(7)   else
(8)      $T = T - \{ \langle l_{B,j}, r_{B,j} \rangle, \langle l_{B,k}, r_{B,k} \rangle \}$ 
(9)      $T = T \cup \{ \langle \min\{r_{B,j}, l_{B,k}\}, \max\{r_{B,j}, l_{B,k}\} \rangle \}$ 
(10)  end if
(11) end while
(12)  $d_A - d_B$  can be chosen as any integer number between (including)  $l_{B,\ell}$ 
      and (including)  $r_{B,\ell}$  where  $\{l_{B,\ell}, r_{B,\ell}\}$  is the only tuple left in  $T$ 

```

Figure 4.6. The minimal-cost pairwise offset alignment algorithm

piecewise linear function $c_{B,j}$ ($1 \leq j \leq m$) is monotonously increasing when $d_A - d_B$ extends left to $l_{B,\ell}$ or right to $r_{B,k}$ (lines (3)-(4)). This implies that the value of $d_A - d_B$ must be between $l_{B,\ell}$ and $r_{B,k}$ such that $\sum_{j=1}^m c_{B,j}$ can be minimized. If $j = k$ (line (5)), $l_{B,j}$ and $r_{B,k}$ are two end points in the same tuple $u_{B,k}$. By Equation 4.16, the cost function $c_{B,k}$ is a constant when $d_A - d_B$ is between $l_{B,\ell}$ and $r_{B,k}$. Therefore, this tuple is no longer considered in the rest steps (line (6)). If $j \neq k$ (lines (7)-(9)), the sum of $c_{B,j}$ and $c_{B,k}$ is considered. By Proposition 9, the sum function can be represented by the piecewise linear function $\langle \min\{r_{B,j}, l_{B,k}\}, \max\{r_{B,j}, l_{B,k}\} \rangle$ when $d_A - d_B$ is between $l_{B,j}$ and $r_{B,k}$. Finally, when there is one tuple left in step (12), the optimal value of $d_A - d_B$ can be chosen using Equation 4.16. The MPOA algorithm terminates in $m - 1$ steps of the **while** loop because one tuple is deleted from T at each step (lines (6) and (8)). By its construction, the MPOA algorithm does find the optimal $d_A - d_B$. If the heap-sort algorithm is used to search the maximum and minimum end points (lines (3) and (4)), the time complexity of the MPOA algorithm is $O(m \log(m))$.

No new end point can be generated in the MPOA algorithm, though new tuples may be generated (line (9)). This important feature implies that given a set of $c_{B,j} = \langle l_{B,j}, r_{B,j} \rangle$ for $1 \leq j \leq m$, there exists an end point to which $d_A - d_B$ can be assigned such that $\sum_{j=1}^m c_{B,j}$ is minimized. This fact results in a straight forward algorithm to find the minimal value of $\sum_{j=1}^m c_{B,j}$: assign each end point to $d_A - d_B$ and find the one which generates the minimal-cost.

Proposition 10 *Given a set of $c_{B,j} = \langle l_{B,j}, r_{B,j} \rangle$ for $1 \leq j \leq m$, there exists an end point to which $d_A - d_B$ can be assigned such that $\sum_{j=1}^m c_{B,j}$ is minimized.*

As mentioned in the previous section, the access offset can have different weights due to various array sizes. For example, in the following program structure, the cost

```

FORALL( $i_1 = 0 : 10$ )
 $s_1$ :    $X(i_1) = Y(i_1, 0) + Y(i_1 + 1, 0) + Y(i_1 + 2, 0)$ 
      FOR( $i_2 = 0 : 10$ )
 $s_2$ :    $X(i_1) = X(i_1) + Y(i_1 + 4, i_2) + Y(i_1 + 5, i_2) + Y(i_1 + 7, i_2)$ 
      END FOR
END FORALL

```

of neighboring communication is $c_{Y,1} + 10 \times c_{Y,2}$ where

$$c_{Y,1} = \begin{cases} 2 - (d_X - d_Y) & \text{when } (d_X - d_Y) < 0 \\ 2 & \text{when } 0 \leq (d_X - d_Y) \leq 2 \\ (d_X - d_Y) & \text{when } 2 < (d_X - d_Y) \end{cases}$$

$$c_{Y,2} = \begin{cases} (7 - (d_X - d_Y)) & \text{when } (d_X - d_Y) < 4 \\ 3 & \text{when } 4 \leq (d_X - d_Y) \leq 7 \\ ((d_X - d_Y) - 4) & \text{when } 7 < (d_X - d_Y) \end{cases}$$

The cost $c_{Y,2}$ is weighted by 10 since a column of remote Y elements are accessed

in the innerloop s_2 . Therefore, the problem of finding the optimal $d_A - d_B$ in the presence of weighted cost situation can be formalized as follows: Assume that there are m distinct references “ $A \leftarrow B@s_j$ ” where $c_{B,j} = \langle l_{B,j}, r_{B,j} \rangle$ for $1 \leq j \leq m$. Each $c_{B,j}$ is weighted by w_j . The optimal $d_A - d_B$ should minimize the value of $\sum_{j=1}^m w_j \times c_{B,j}$.

One approach to resolve this alignment problem is to use the MPOA algorithm. In line (1), let T consist of w_j identical copies of tuple $\langle l_{B,j}, r_{B,j} \rangle$ for $1 \leq j \leq m$. Though it may take excessive time if w_j is large, the MPOA algorithm guarantees that Proposition 10 still holds regarding to the weighted cost of neighboring communication. Therefore, we can assign $d_A - d_B$ to be each distinct end point in all access offset tuples and find the one with the minimal-cost.

4.4 Spanning-Tree Offset Alignment Algorithm

4.4.1 Offset Reference Graph

The problem of offset alignment can be modeled by *offset reference graph* (ORG). Given a program structure in which the alignment matrix of each array is pre-determined, an ORG $G = (V, E)$ is constructed as follows. An array is represented by a node in V . For an array which has multiple instances referenced in one or more **FORALL** assignments, there is only one corresponding node in the ORG. A reference which is free of reorganization communication is presented by an edge in E . The edge connects two nodes which represent two arrays specified by the corresponding reference. There is only one edge connecting the LHS array A and the RHS array B for all B instances which have the same access matrix and are referenced in the same statement. A ORG is undirected. Since offset alignment is studied independently for each aligned-base group, different ORGs are used to model different aligned-base groups. For simplicity, the examples used in the rest of this chapter only contain a

single aligned-base group and there is only one ORG for each program structure.

```

    FORALL( $i_1 = 0 : 9$ )
      FOR( $i_1 = 0 : 1$ )
 $s_1$ :       $A(i_1) = X(i_1 + 2, i_2) + A(i_1 + 4)$ 
      END FOR

      FOR( $i_2 = 0 : 9$ )
 $s_2$ :       $A(i_1) = A(i_1) + Z(i_1 + 3, i_2) + Z(i_1 + 5, i_2)$ 
      END FOR

 $s_3$ :       $Y(i_1) = A(i_1 - 3) + A(i_1) + A(i_1 + 1) + X(i_1 - 2, 0)$ 

      FOR( $i_2 = 0 : 1$ )
 $s_4$ :       $Y(i_1) = Y(i_1) + Z(i_1 + 2, i_2) + Z(i_1 + 4, i_2)$ 
      END FOR
    END FORALL

```

Figure 4.7. Example 13: A Livermore benchmark loop

Figure 4.8 shows the ORG for Example 13 (Figure 4.7). In Figure 4.8, each array variable is represented by a node labeled by the variable name. Edges are constructed based on the references generated by each **FORALL** assignment in Example 13. For example, edge (A, X) connects nodes A and X due to reference “ $A \leftarrow X@s_1$ ”. Edge (A, Z) connects nodes A and Z because of reference “ $A \leftarrow Z@s_2$ ”. Note that reference “ $A \leftarrow Z@s_2$ ” represents the access to all distinct instances of array Z in statement s_2 . Therefore, there is only one edge incident with A and Z . Since there is one-to-one correspondence between an array and a node, terms “array” and “node” are used alternatively in the rest of this chapter. Similarly, since there is one-to-one correspondence between an edge and a reference, terms “edge” and “reference” are also used alternatively in the rest of this chapter.

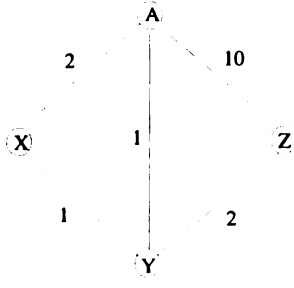


Figure 4.8. The offset reference graph for Example 13

Consider reference “ $A \leftarrow B@s_k$ ” with access offset tuple $u_{B,k}$. By Proposition 4.16, the minimum cost of neighboring communication is zero if $u_{B,k}$ only contains one element. The minimum cost of neighboring communication is equal to $r_{B,k} - l_{B,k}$ if $u_{B,k}$ contains more than one element and $l_{B,k}$ is the left end point and $r_{B,k}$ is the right end point. However, considering multiple references, there may not exist a solution of the alignment offsets such that the minimum cost of neighboring communication can be achieved for every reference. The reason is that the minimum-cost requirement imposed by one reference may conflict with that imposed by another reference, in particular, when the two edges representing these two references are involved in a cycle. For example, Figure 4.8 has a cycle $A \rightarrow X \rightarrow Y \rightarrow A$. By Equation 4.16, since $u_{X,1} = \{2\}$, the minimum cost for reference “ $A \leftarrow X@s_1$ ” is zero if the following equation is satisfied.

$$d_A - d_X = 2 \quad (4.17)$$

Since $u_{Y,3} = \{-2\}$, the minimum cost for reference “ $Y \leftarrow X@s_3$ ” is zero if the following equation is satisfied.

$$d_Y - d_X = -2 \quad (4.18)$$

Since $u_{A,3} = \{-3, -1, 1\}$, the minimum cost for reference “ $Y \leftarrow A@s_3$ ” is 4 if the

following inequality is satisfied.

$$-3 \leq d_Y - d_A \leq 1 \quad (4.19)$$

However, subtracting Equation 4.18 from Equation 4.17, we have

$$d_A - d_Y = 4$$

which does not satisfy Inequality 4.19. This implies that the minimum cost of neighboring communication cannot be attained for at least one reference among the above three.

4.4.2 Spanning Tree Offset Alignment Algorithms

The conflict of minimum-cost requirement can only occur if an ORG consists of cycles. Intuitively, such conflict can be resolved by a spanning tree: offset alignment between two arrays is specified by the tree edge connecting these two arrays such that the minimum cost can be achieved on the reference represented by this tree edge. Each non-tree edge determines a unique fundamental cycle of the ORG with respect to the spanning tree. The minimum cost of neighboring communication may not be achieved for each non-tree edge. Each edge is weighted. Since loop s_2 in Example 13 is a sequential loop, the cost of neighboring communication for reference “ $A \leftarrow Z$ ” is weighted by 10, the number of elements in a row of two-dimensional array Z . Consequently, edge (A, Z) is weighted by 10. For simplicity, the weight on edge (A, B) is denoted as $w_{A,B}$. In Figure 4.8, $w_{A,Z} = 10$, $w_{A,X} = w_{Y,Z} = 2$, and $w_{X,Y} = w_{A,Y} = 1$. Given a choice between two edges, the tree edge would be chosen as the one with higher weight. As a result, the spanning tree for offset alignment would be chosen as a maximum-weight spanning tree (MWST).

Similar to the analysis in section 3.4 for the MICS algorithm, the MWST algorithm may not generate offset alignment which offers the minimum overall cost of neighboring communication regarding to all references in a program structure. For Example 13, the maximum-weight spanning tree is shown in Figure 4.9(a). Alignment offsets are selected such that $d_A - d_Z = 3$, $d_Y - d_Z = 2$, and $d_A - d_X = 2$. Thus, the minimum cost requirements of neighboring communication imposed by edges (A, Z) , (Y, Z) , and (A, X) are satisfied, respectively. Without loss of generality, let $d_A = 0$. We have $d_Z = -3$, $d_Y = 2$, and $d_X = -2$. Therefore, by Proposition 7, the overall cost of neighboring communication is 42 regarding to all five references in Example 13. However, the spanning tree alignment in Figure 4.9(b) generates lower overall cost of neighboring communication. In Figure 4.9(b), alignment offsets are selected such that $d_A - d_Z = 3$, $d_Y - d_A = 0$, and $d_A - d_X = 2$. Thus, the minimum cost requirements of neighboring communication imposed by edges (A, Z) , (Y, A) , and (A, X) are satisfied, respectively. Without loss of generality, let $d_A = 0$. We have $d_Z = -3$, $d_Y = 0$, and $d_X = -2$. Therefore, by Proposition 7, the overall cost of neighboring communication is only 40.

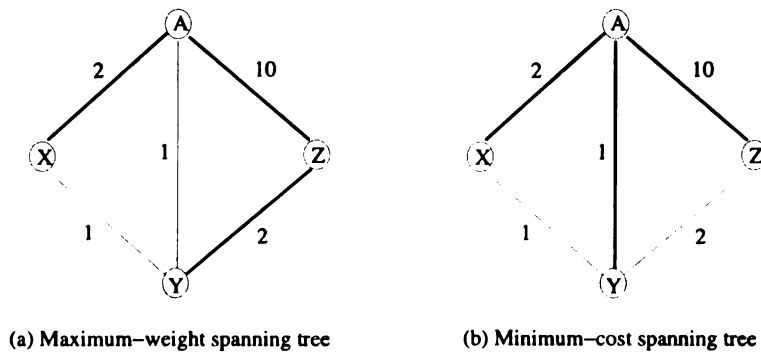


Figure 4.9. The offset reference graph for Example 13

In this section, we propose an efficient spanning-tree offset alignment (STOA)

algorithm which is an improvement of the MWST algorithm. Given an ORG $G = (V, E)$, the STOA algorithm can be formalized in Figure 4.10.

```

(1) Choose an arbitrary array variable  $A$ 
    Let  $T = \{A\}$ 
(2) while  $T \neq V$  do
(3)   Let  $Q_1$  be the set of single-degree neighbors of  $T$ 
(4)   if  $Q_1 \neq \emptyset$  then
(5)     Find a node  $B$  in  $Q_1$  such that there exists an edge  $(B, H)$ 
        where  $H \in T$  and  $w_{B,H} = \min_{\{X \in Q_1, Y \in T\}} w_{X,Y}$ 
(6)     Let  $d_B - d_H = l_{H,k}$  where edge  $(B, H)$  represents reference
        " $B \leftarrow H@s_k$ " and  $l_{H,k}$  is the left end point in  $u_{H,k}$ 
(7)      $T = T \cup \{B\}$ 
(8)   else
(9)     Let  $Q_2 = V - T$ 
(10)    for each node  $X$  in  $Q_2$  do
(11)      Let  $w_X = \sum_{\{Y \in T\}} w_{X,Y}$ 
(12)    end for
(13)    Find a node  $B$  in  $Q_2$  such that  $w_B = \max_{\{X \in Q_2\}} w_X$ 
(14)    Use Proposition 10 to determine the value of  $d_B$  which achieves
         $\min_{\{d_B - d_A\}} \sum_{\{(B,H) \in E, H \in T, \text{ and } (H,B) \text{ represents } "H \leftarrow B@s_j" \}} c_{B,j}$ 
(15)     $T = T \cup \{B\}$ 
(16)  end if
(17) end while

```

Figure 4.10. The spanning-tree offset alignment algorithm

In Figure 4.10, the single-degree neighbor and the multi-degree neighbor is defined as in section 3.4. If Q_1 is not empty, like the MWST algorithm, the tree edge is selected as an edge with the largest weight among all the edges which are incident with one node in Q_1 and another node in T (lines (4)-(7)). If Q_1 is empty but Q_2 is not, we find the next tree edge as follows. First, w_X , the sum of the weight over all edges connecting X and another node in T , is computed for each node X in Q_2 (lines

(10)-(12)). Then the node B in Q_2 with the maximum sum of the weight is selected to be the next tree node (line (15)). The value of alignment offset d_B is determined by using Proposition 10. By this construction, the STOA algorithm is superior to the MWST algorithm.

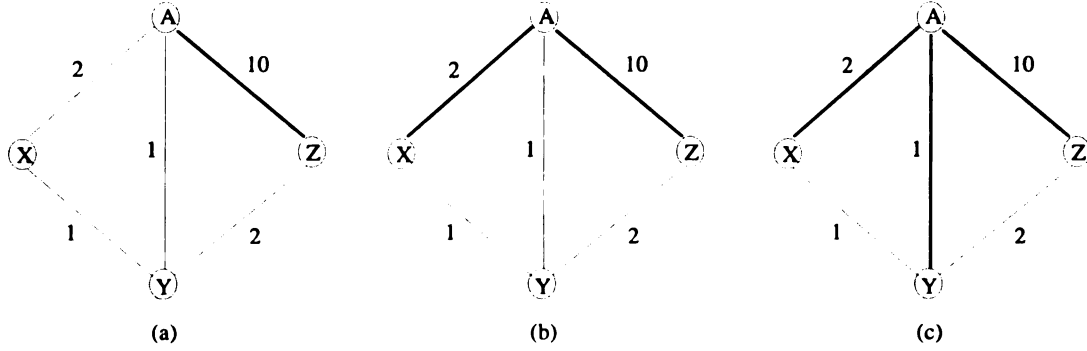


Figure 4.11. Resolving offset alignment for Example 13

Figure 4.11 shows how the STOA algorithm works to resolve offset alignment for Example 13. In Figure 4.11, array A is selected as the template (line(1)). Note that the selection is arbitrary. The values of other alignment offsets will be in the form of $\frac{+}{-} d_A + k$ where k can be any integer. $T = \{A\}$. The first phase is shown in Figure 4.11(a). Since all X , Y , and Z are single-degree neighbors of T , $Q_1 = \{X, Y, Z\}$. Since it has the maximum weight among edges (A, Y) and (A, Z) , edge (A, Z) is selected as the tree edge (line (5)). Since edge (A, Z) represents reference " $A \leftarrow Z@s_2$ " where $u_{Z,2} = \{3, 5\}$, $d_A - d_Z = 3$ (line (6)). Note that line (6) only shows one case. In another case, edge (B, H) may represent reference " $H \leftarrow B@s_k$ ". As a result, d_B should be determined as $d_H - d_B = l_{B,k}$.

The second phase is shown in Figure 4.11(b). In the second phase, $T = \{A, Z\}$. Therefore, node X becomes the only single-degree neighbor of T . Similar to the first phase, we have $d_A - d_X = 2$.

The last phase is shown in Figure 4.11(c). In the last phase, $T = \{A, X, Z\}$. Node Y is multi-degree neighbor of T . The cost functions for references “ $Y \leftarrow X@s_3$ ”, “ $Y \leftarrow A@s_3$ ”, and “ $Y \leftarrow Z@s_4$ ” can be represented by piecewise linear functions $c_{X,3} = \langle -2 \rangle$, $c_{A,3} = \langle -3, 1 \rangle$, and $c_{Z,4} = \langle 2, 4 \rangle$, respectively. Note that $c_{X,3} = \langle -2 \rangle$ is a function of $d_Y - d_X$. Since $d_X = d_A - 2$ (step (b)), $c_{X,3}$ can be rewritten as $\langle 0 \rangle$ with respect to $d_Y - d_A$. $c_{Z,4} = \langle 2, 4 \rangle$ is a function of $d_Y - d_Z$. Since $d_A - d_Z = 3$ (step (a)), $c_{Z,4}$ can be rewritten as $\langle -1, 1 \rangle$ with respect to $d_Y - d_A$. Therefore, using Proposition 10, we conclude that the minimal value of the sum $c_{Z,4} + c_{A,3} + c_{X,3}$ can be achieved when $d_Y - d_A = 0$ (lines (14)). The values of T , Q_1 , and Q_2 in each phase are illustrated in Table 4.1.

Table 4.1. Values of T , Q_1 , and Q_2 in executing the MICS algorithm for Example 13

phases	T	Q_1	Q_2	offst alignment
(a)	$\{A\}$	$\{X, Y, Z\}$	ϕ	$d_A - d_Z = 3$
(b)	$\{A, X\}$	$\{Z\}$	$\{Y\}$	$d_X = d_A - 2$
(c)	$\{A, X, Y\}$	ϕ	$\{Y\}$	$d_Y - d_A = 0$

In the STOA algorithm, each edge in the ORG G is referenced only once. If the heap-sort algorithm [79] is used in lines (5) and (13), the time complexity of finding the maximum-weight edge can be reduced to $O(\log|E|)$ where $|E|$ is the number of edges in DRG $G = (V, E)$. As a result, the time complexity of the STOA algorithm is $O(|E|\log|E|)$.

4.5 Optimizing RHS Expression Evaluation

4.5.1 RHS Expression Evaluation Optimization

Similar to the base alignment analysis, neighboring communication can be minimized by an optimal evaluation tree in which an intermediate result may be evaluated on a remote processor instead of the local processor which owns the LHS operand. The limitation of the owner-computes rule can be exceeded by executing different parts of a **FORALL** assignment in distinct processors. We assume the original program has already been pre-processed by transforming each original **FORALL** assignment into an equivalent set of **FORALL** assignments, each of which has the RHS expression operated by one kind of associate and commutative operations.

```

FORALL( $i_1 = 0 : 9$ )
 $s_1$ :    $A(i_1) = B(i_1) * X(i_1) * Y(i_1)$ 
 $s_2$ :    $B(i_1) = A(i_1 + 1) + X(i_1 + 3) + Y(i_1 + 4)$ 
END FORALL

```

Figure 4.12. Example 14: A Livermore Kernel 7 loop

Consider Example 14 in Figure 4.12. In Example 14, offset alignment are pre-determined such that $d_A = d_B = d_X = d_Y$. By Equation 4.16, the cost of neighboring communication for all references in statement s_1 is 0. Therefore, statement s_1 is free of neighboring communication. However, such an offset alignment decision makes every reference in statement s_2 not free of neighboring communication. Consider how to minimize the cost of neighboring communication in statement s_2 . Since the intermediate result can be generated by a remote processor rather than the one which owns the LHS operand, the key issue is to find what operands should be operated together

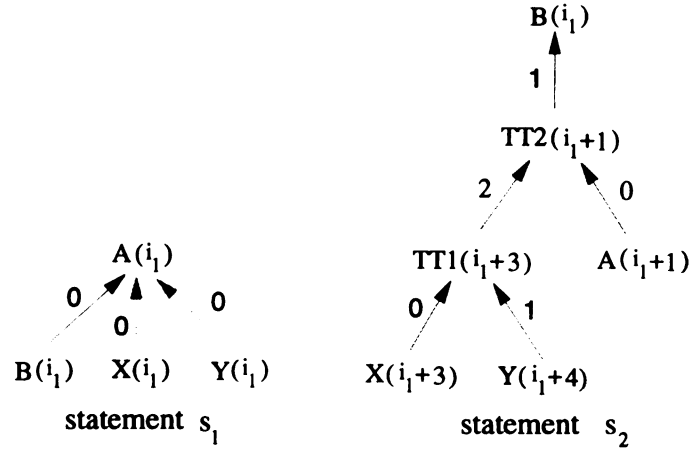


Figure 4.13. The optimal evaluation trees for Example 14

on what processor? Figure 4.13 shows an optimal evaluation tree for statement s_2 . In Figure 4.13, each reference is represented by an arc. The cost of neighboring communication required for each reference is represented by the weight on the corresponding arc. $X(i_1 + 3)$ and $Y(i_1 + 4)$ are first added and the result is saved to a temporary variable $TT1(i_1 + 3)$ at the remote processor which owns $X(i_1 + 3)$. $d_{TT1} = d_X$. The cost of neighboring communication for reference “ $TT1 \leftarrow Y$ ” is 1. $TT1(i_1 + 3)$ and $A(i_1 + 1)$ are then added and the result is saved to another temporary variable $TT2(i_1 + 1)$ at the remote processor which owns $A(i_1 + 1)$. $d_{TT2} = d_A$. The cost of neighboring communication for reference “ $TT2 \leftarrow TT1$ ” is 2. Finally, $TT2(i_1 + 1)$ is assigned to $B(i_1)$. The cost of neighboring communication for reference “ $B \leftarrow TT2$ ” is 1. Therefore, the optimal evaluation for Example 14 can be written as follows:

```

FORALL( $i_1 = 0 : 9$ )
 $s_1$ :    $A(i_1) = B(i_1) * X(i_1) * Y(i_1)$ 
 $s_{2.1}$ :  $TT1(i_1 + 3) = X(i_1 + 3) + Y(i_1 + 4)$ 
 $s_{2.2}$ :  $TT2(i_1 + 1) = TT1(i_1 + 3) + A(i_1 + 1)$ 
 $s_{2.3}$ :  $B(i_1) = TT2(i_1 + 1)$ 
END FORALL

```


4.5.2 Post-Alignment Optimization

The study of Example 14 reveals two important phases in determining the optimal expression evaluation with regard to multiple **FORALL** assignments. One phase is the offset alignment phase. By Proposition 6, no cost of neighboring communication is involved in a RHS expression evaluation if an appropriate offset alignment is used. For example, in Example 14, assignment s_1 is free of neighboring communication when $d_A = d_B = d_X = d_Y$. However, for multiple assignments, neighboring communication may not be fully avoided no matter how good offset alignment is. Therefore, the other phase is to minimize the cost of neighboring communication in each statement after alignment offset for each array is determined. This phase is known as *post-alignment optimization*. In Example 14, the post-alignment optimization for assignment s_2 is shown in Figure 4.13. These two phases are not isolated. By the definition, the post-alignment optimization is carried out after the offset alignment analysis is done. On the other hand, however, the decision of offset alignment depends on the accurate cost estimation provided by post-alignment optimization techniques. We study the post-alignment optimization technique as follows.

Proposition 11 *Assume that in a **FORALL** assignment, the RHS expression is operated by one kind of associate and commutative operations. $A_0(i_1 + r_0)$ is referenced on the LHS. All instances referenced on the RHS are $A_1(i_1 + r_1)$, $A_2(i_1 + r_2)$, \dots , and $A_m(i_1 + r_m)$. Assume that d_{A_0} , d_{A_1} , d_{A_2} , \dots , and d_{A_m} are pre-determined. Assume that the sequence of $r_0 + d_{A_0}, r_1 + d_{A_1}, r_2 + d_{A_2}, \dots, r_m + d_{A_m}$ is in monotonous (either increase or decrease) order. Therefore, the optimal evaluation of the assignment is as follows:*

1. *Begin with $A_m(i_1 + r_m)$. Operate $A_m(i_1 + r_m)$ with $A_{m-1}(i_1 + r_{m-1})$ and save the intermediate result in the temporary variable $TT_{m-1}(i_1 + r_{m-1})$ at the processor which owns $A_{m-1}(i_1 + r_{m-1})$.*

2. Operate $TT_j(i_1 + r_j)$ with $A_{j-1}(i_1 + r_{j-1})$ and save the intermediate result in the temporary variable $TT_{j-1}(i_1 + r_{j-1})$ at the processor which owns $A_{j-1}(i_1 + r_{j-1})$ for $2 \leq j \leq m - 1$.

3. Assign $TT_1(i_1 + r_1)$ to $A_0(i_1 + r_0)$.

It is easy to prove that Proposition 11 is true. Consider the cost of neighboring communication involved in reading $A_m(i_1 + r_m)$. Since $(r_m + d_{A_m}) - (r_0 + d_{A_0}) > r_j + d_{A_j} - (r_0 + d_{A_0})$ for $(1 \leq j \leq m - 1)$, the distance between $A_m(i_1 + r_m)$ and the LHS operand is the longest among that between any other RHS operand and the LHS operand. Therefore, we hope that $A_m(i_1 + r_m)$ can be operated with another operand which is closer to the LHS operand. On the other hand, the cost of moving $A_m(i_1 + r_m)$ to such an operand should be as small as possible. Since the closest operand to $A_m(i_1 + r_m)$ is $A_{m-1}(i_1 + r_{m-1})$, $A_{m-1}(i_1 + r_{m-1})$ becomes the best candidate. Repeating the similar approach, the optimism in Proposition 11 can be proved.

In Proposition 11, the symbols A_0, A_1, A_2, \dots , and A_m are not required to be pairwise distinct. Proposition 11 still holds if an arbitrary subset of them represents the same array. For example, in the following **FORALL** assignment, assume $d_X = d_Y = 0$.

```

FORALL( $i_1 = 0 : 9$ )
 $s_1:$     $X(i_1) = Y(i_1 - 3) + Y(i_1 - 1) + Y(i_1 + 1) + X(i_1 - 2) + X(i_1 + 2)$ 
         $+ X(i_1 + 6)$ 
END FORALL

```

Figure 4.14 shows the result of the post-alignment optimization for the above **FORALL** assignment. In Figure 4.14, each reference is represented by an arc. The cost of neighboring communication required for each reference is represented by the

weight on the corresponding arc. Note that the sequence of instances with respect to the monotonous order of access offset is $\{Y(i_1 - 3), X(i_1 - 2), Y(i_1 - 1), X(i_1), Y(i_1 + 1), X(i_1 + 2), X(i_1 + 6)\}$. Since the LHS instance $X(i_1)$ resides in the middle, the sequence is separated into two subsequences: $\{Y(i_1 - 3), X(i_1 - 2), Y(i_1 - 1), X(i_1)\}$ and $\{X(i_1), Y(i_1 + 1), X(i_1 + 2), X(i_1 + 6)\}$. Proposition 11 works on each of subsequences as shown in Figure 4.14. Note that the unit cost of neighboring communication must have the same weight with regard to all arrays referenced in the same **FORALL** assignment. Therefore, Proposition 11 holds with the consideration of the weighted cost.

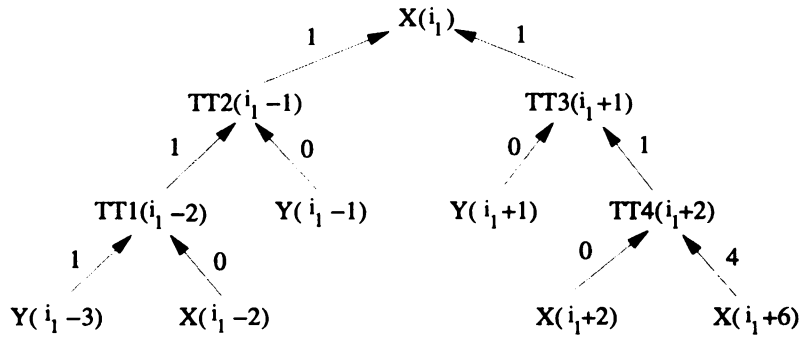


Figure 4.14. An example of post-alignment optimization

4.5.3 Alignment Graph

Like base alignment, *alignment graph* (AG) is used to model the offset alignment problem. An AG is a collection of arrays and statements which can be represented as a bipartite graph, $G = (V_a, V_s, E)$. An array is represented by a node in V_a . A statement is represented by a node in V_s . A undirected edge in E connects an array A and a statement s_k if A is the LHS array. A undirected edge in E connects an array B and a statement s_k if $D_B F_{B,k} = D_A F_{A,k}$ where A is referenced on the LHS and B

is referenced on the RHS in statement s_k . There is only one edge connecting the LHS array A and the RHS array B for all B instances which have the same access matrix and are referenced in the same statement. Figure 4.15 shows the AG for Example 13 (Figure 4.7).

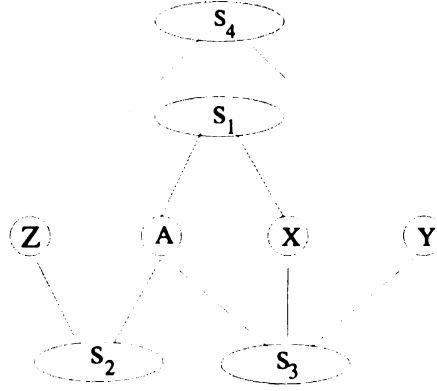


Figure 4.15. The AG for Example 13

4.5.4 AG-Based Offset Alignment Algorithm

The AG-based offset alignment (AGOA) algorithm is shown in Figure 4.16. In an AG $G = (V_a, V_s, E)$, there is a set, denoted as Q_k , associated with each node s_k in V_s . Initially, each set Q_k is empty. During the execution, each Q_k will contain elements in access offset tuple for each array which is referenced in statement s_k and whose alignment offset has been determined.

In Figure 4.16, $g_{\ell,A,k}$ represents an element in the access offset tuple $u_{A,k} = \{g_{1,A,k}, g_{1,A,k}, \dots, g_{h,A,k}\}$ (lines (5) and (15)). The first array selected in line (1), say A , serves as the template. The value of other alignment offset is in the form of $\pm d_A + k$ where k is any integer. Set T contains all choices of $\pm d_A + k$ for every other array to select as alignment offset. Given an array B , d_B is chosen among all

elements in T such that the sum of the cost of neighboring communication over each **FORALL** assignment where B is referenced can be minimized (line (13)). The cost of neighboring communication in executing each **FORALL** assignment is estimated using Proposition 11. If the heapsort algorithm [79] is used in finding the minimum value, the time complexity in line (13) is

$$O(\log \sum_{(B,s_k) \in E} |u_{B,k}|)$$

```

(1) for each node  $B$  in  $V_a$  do
(2)    $T = \phi$ 
(3)   for each neighboring node  $s_k$  (in  $V_s$ ) of  $B$  do
(5)     for each element  $g_{\ell,B,k}$  in  $u_{B,k}$  do
(6)       for each element  $q$  in  $Q_k$  do
(7)         if  $q - g_{\ell,B,k}$  is not in  $T$  then
(8)            $T = T \cup \{q - g_{\ell,B,k}\}$ 
(9)         end if
(10)      end for
(11)    end for
(12)  end for
(13)  Assign  $d_B$  to be the element in  $T$  such that the sum of the
      cost of neighboring communication over each FORALL assignment
      where  $B$  is referenced is the minimum
(14)  for each neighboring node  $s_k$  (in  $V_s$ ) of  $B$  do
(15)    for each  $g_{\ell,B,k}$  in  $u_{B,k}$  do
(16)      if  $d_B + g_{\ell,B,k}$  is not in  $Q_k$  then
(17)         $Q_k = Q_k \cup \{d_B + g_{\ell,B,k}\}$ 
(18)      end if
(19)    end for
(20)  end for
(21) end for

```

Figure 4.16. The alignment graph offset alignment algorithm

where $|u_{B,k}|$ is the number of distinct elements in $u_{B,k}$. Thus, the time complexity of the AGOA algorithm is

$$O\left(\sum_{(B,s_k)} |u_{B,k}| \log \sum_{(B,s_k)} |u_{B,k}|\right)$$

Table 4.2. Resolving offset alignment for Example 13 by using the AGOA algorithm

	Z	Y	X
Q_1	$\{d_A, d_A + 4\}$	$\{d_A, d_A + 4\}$	$\{d_A, d_A + 4\}$
Q_2	$\{d_A\}$	$\{d_A, d_A + 2\}$	$\{d_A, d_A + 2\}$
Q_3	$\{d_A - 3, d_A, d_A + 1\}$	$\{d_A - 3, d_A, d_A + 1\}$	$\{d_A - 3, d_A, d_A + 1\}$
Q_4	$\{\}$	$\{d_A - 1, d_A + 1\}$	$\{d_A - 1, d_A + 1\}$
result	$d_Z = d_A - 3$	$d_Y = d_A + 1$	$d_X = d_A + 2$

Table 4.2 illustrates how offset alignment in Example 13 is resolved by using the AGOA algorithm (Figure 4.16). The algorithm begins with A since A has the maximum degree of connectivity. Thus, A serves as the template. Arrays Z , Y , and X are chosen in sequential (line (1)). Table 4.2 shows the content of Q_1 , Q_2 , Q_3 , and Q_4 at the end of each iteration in the outmost **for** loop, where each of arrays A , Z , Y , and X is selected. When A is first select, sets Q_1 , Q_2 , Q_3 , and Q_4 are all empty. Therefore, no operation is done in line (13). Since A is referenced in statements s_1 , s_2 , and s_3 , $Q_1 = \{d_A, d_A + 4\}$, $Q_2 = \{d_A\}$, and $Q_3 = \{d_A - 3, d_A, d_A + 1\}$ by lines (14)-(20).

Next, Z is selected. Since Z is referenced in statement s_2 and $u_{Z,2} = \langle +3, +5 \rangle$, $T = \{d_A - 3, d_A - 5\}$. Note that though Z is referenced in statement s_4 , Q_4 is still empty. Therefore, Q_4 makes no contribution to T . By Proposition 11 (line (13)), either $d_A - 3$ or $d_A - 5$ can be the value of d_Z . Thus, we choose $d_Z = d_A - 3$. Executing lines (14)-(20), $Q_2 = \{d_A, d_A + 2\}$ and $Q_4 = \{d_A - 1, d_A + 1\}$ since Z is referenced in both statements s_2 and s_4 .

Next, Y is selected. Since Y is referenced in statements s_3 and s_4 , for executing lines (2)-(12), $T = \{d_A - 3, d_A - 1, d_A, d_A + 1\}$. Consider $c_{A,3}$, the cost of neighboring communication in accessing the RHS elements $A(i_1 - 3)$, $A(i_1)$, and $A(i_1 + 1)$ in statement s_3 . By Proposition 11 (line (13)), $c_{A,3} = 4$ if d_Y is chosen as any element in T . Consider $c_{Z,3}$, the cost of neighboring communication in accessing the RHS elements $Z(i_1 + 2)$ and $Z(i_1 + 4)$ in statement s_4 . By Proposition 11 (line (13)), $c_{Y,4} = 8$ if d_Y is selected as $d_A - 3$. $c_{Y,4} = 4$ if d_Y is selected as either $d_A - 1$, or d_A , or $d_A + 1$. Thus, we choose $d_Y = d_A + 1$. Executing lines (14)-(20), Q_3 and Q_4 are not changed.

Last, X is selected. Since X is referenced in statements s_1 and s_3 , by executing lines (2)-(12), $T = \{d_A - 2, d_A - 1, d_A + 2, d_A + 3\}$. Consider $c_{X,1}$, the cost of neighboring communication in accessing the RHS elements $X(i_1 + 2)$ in statement s_1 . No extra cost of neighboring communication is paid if d_X is chosen as either $d_A - 2$ or $d_A + 2$. If $d_X = d_A - 2$, elements $A(i_1)$ and $X(i_1 + 2)$ are owned by the same processor. If $d_X = d_A + 2$, elements $A(i_1 + 4)$ and $X(i_1 + 2)$ are owned by the same processor and the RHS expression can be evaluated on the processor which owns $A(i_1 + 4)$. Consider $c_{X,3}$, the cost of neighboring communication in accessing the RHS elements $X(i_1 - 2)$ in statement s_3 . Similarly, no extra cost of neighboring communication is paid if d_X is chosen as either $d_A - 1$ or $d_A + 2$. Therefore, $d_A + 2$ is the only solution of d_X (line (13)).

4.5.5 Performance Comparison

Table 4.3 shows the comparison of the overall cost of neighboring communication generated by the MWST, STOA, and AGOA algorithms using Example 13. The AGOA algorithm outperforms the other two algorithms. In the AGOA algorithm, each element in access offset tuple is taken count into in performing the best post-alignment optimization. For example, when d_X is chosen as $d_A + 2$, in statement

Table 4.3. Comparison of the MWST, STOA, and AGOA algorithms using Example 13

Algorithms	Overall cost of neighboring communication
MWST	42
STOA	40
AGOA	36

s_3 elements $X(i_1 - 2, 0)$ and $A(i_1)$ are owned by the same processor. However, this information is not utilized by the STOA algorithm since the STOA algorithm only considers the two end points in $u_{A,3} = \langle -3, 0, 1 \rangle$ and ignores the middle element 0.

4.6 Avoiding Redundant Communication

When the same context of remote elements are referenced in more than one **FORALL** assignment, the local processor should receive a single copy of these elements instead of multiple identical copies. The importance of such redundant communication avoidance has been shown in Section 3.6 regarding to the base alignment analysis. In this section, we concentrate on issues of avoiding redundant communication regarding to the offset alignment analysis.

4.6.1 Redundant Communication

In Example 15 (Figure 4.17), array A is two-dimensional and other arrays are one-dimensional. A is partitioned in columns such that all elements in the same column are collapsed to the same processor. Only the outmost loop indexed by i_1 is distributed across the processors. Assume that offset alignment is pre-determined such that $d_A = d_B = d_Z = 0$. Since $d_B = d_Z$, the LHS element $Z(i_1)$ in statement s_3 and the LHS element $B(i_1)$ in statement s_4 are located to the same local processor. Therefore,

the two copies of $A(i_1+3, 0)$ referenced in both statements are identical. The messages transmitting remote element $A(i_1+3, 0)$ in statement s_3 and s_4 are redundant. Assume that processor $P(i)$ owns elements $A(m_i : m_{i+1} - 1, 0)$. The remote A elements referenced by $P(i)$ in executing statement s_3 and s_4 are $A(m_{i+1} : m_{i+1}+2, 0)$. Consider statement s_2 . Since $d_A = d_B$, the remote A elements referenced by $P(i)$ in executing statement s_2 are $A(m_{i+1} : m_{i+1}+4, 0)$. Therefore, the messages transmitting remote A elements in statements s_3 and s_4 are further redundant with the messages transmitting remote A elements in statements s_2 . Combining the remote A elements referenced by local processor $P(i)$ in executing statements s_2 , s_3 , and s_4 , we conclude that $A(m_{i+1} : m_{i+1} + 4, 0)$ are only remote A elements required to be transmitted from the remote processor to the local processor.

```

      FORALL( $i_1 = 0 : 9$ )
 $s_1$ :       $A(i_1, 0) = X(i_1)$ 
 $s_2$ :       $B(i_1) = A(i_1 + 4, 0) + A(i_1 + 5, 0)$ 
 $s_3$ :       $Z(i_1) = A(i_1 + 3, 0) * B(i_1)$ 
 $s_4$ :       $B(i_1) = X(i_1) + A(i_1 + 3, 0)$ 
          FORALL( $i_2 = 0 : 9$ )
 $s_5$ :       $A(i_1, i_2) = A(i_1, i_2 - 1) + i_2$ 
          END FORALL
          FOR( $i_2 = 0 : 1$ )
 $s_6$ :       $B(i_1) = B(i_1) + A(i_1 + 1, i_2)$ 
          END FOR
      END FORALL

```

Figure 4.17. Example 15: A Dhrystone benchmark loop

As addressed in section 3.6, the identification of redundant neighboring communication depends on both location and the context. The location requires that the senders of redundant communication must be the same, so do the receivers. The

context requirement implies that the context of the redundant remote elements must be the same. As introduced in Section 3.6, the concept of single assignment block can be used to identify the distinct contexts with regard to a particular array variable. In Example 15, there are two single assignment blocks with regard to A : $\{s_1, s_2, s_3, s_4\}$ and $\{s_5, s_6\}$. Therefore, the message transmission of $A(m_{i+1}, 0)$ in executing statement s_6 is not redundant with the message transmission of $A(m_{i+1} : m_{i+1} + 4, 0)$ in executing statements s_2 , s_3 , and s_4 because the context of array A is re-defined in statement s_5 .

More accurately, redundant neighboring communication can be classified into two types: *fully redundant messages* and *partially redundant messages*. In respect to fully redundant messages, the messages carrying remote elements for executing two distinct **FORALL** assignments are identical, such as the messages transmitted in executing statements s_3 and s_4 (Example 15). Only one copy of the fully redundant messages needs to be transmitted and the cost of neighboring communication is equal to the size of the single message. In respect to partially redundant messages, the messages carrying remote elements for executing two distinct **FORALL** assignments overlap in a subset of common elements. However, the two messages are not identical, such as the messages transmitted in executing statements s_2 and s_3 (Example 15). In this case, the redundant elements should be only transmitted once. In Example 15, the result of combining remote A elements accessed in executing statements s_2 and s_3 can be treated as if instances $A(i_1+3, 0)$, $A(i_1+4, 0)$, and $A(i_1+5, 0)$ are accessed in the same statement and thus the access offset tuple becomes $\langle 3, 4, 5 \rangle$. The understanding of avoiding partially redundant messages is important in correctly estimating the cost of necessary neighboring communication.

4.6.2 Enhanced Alignment Graph

As addressed in Section 4.6, the enhanced alignment graph (EAG) is used to identify single assignment block. The definition of an enhanced alignment graph (EAG) $G = (V_a, V_s, E, A_s)$ is similar to that of an alignment graph $G = (V_a, V_s, E)$ except that an EAG has an extra arc set A_s . The definition of V_a , V_s , and E can be found in Section 4.5. There is an arc in A_s from node s_ℓ to node s_k if an array defined in statement s_ℓ is used in statement s_k . A distinct EAG is used for the offset alignment analysis in each aligned-base group. Figure 4.18 shows the EAG for Example 15 (Figure 4.17). In Figure 4.18, edges are weighted 1 if their weights are not labeled.

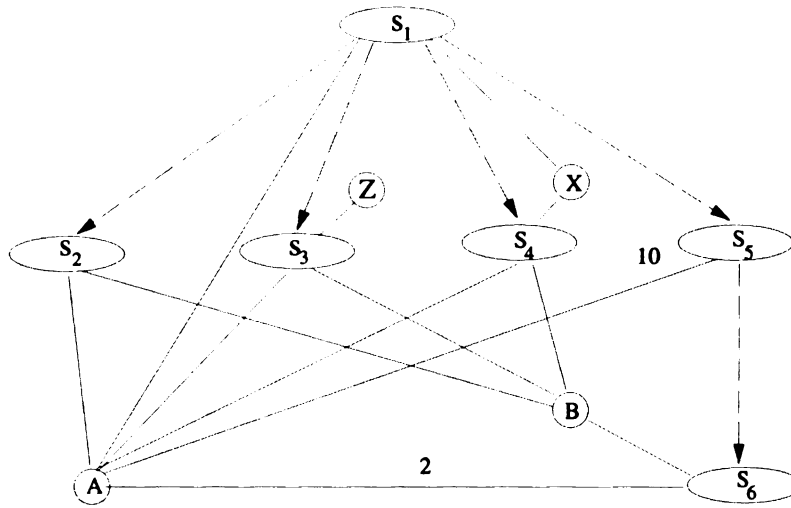


Figure 4.18. Enhanced alignment graph for Example 15

4.6.3 EAG-Based Offset Alignment Algorithm

The EAG-based offset alignment (EAGOA) algorithm is shown in Figure 4.19. The EAGOA algorithm is an invariant of AGOA algorithm introduced in Section 4.5. Like the AGOA algorithm, in Figure 4.19, set Q_k is associated with each node s_k in V_s .

Initially, each set Q_k is empty. During the execution, each Q_k will contain elements in access offset tuple for each array which alignment offset has been determined and is referenced in statement s_k . $g_{\ell,A,k}$ represents an element in the access offset tuple $u_{A,k} = \{g_{1,A,k}, g_{1,A,k}, \dots, g_{h,A,k}\}$ (lines (5) and (15)). The first array selected in line (1), say A , serves as the template. The values of other alignment offsets are in the form of $-d_A + k$ where k is any integer.

```

(1) for each node  $B$  in  $V_a$  do
(2)    $T = \phi$ 
(3)   for each neighboring node  $s_k$  (in  $V_s$ ) of  $B$  do
(5)     for each element  $g_{\ell,B,k}$  in  $u_{B,k}$  do
(6)       for each element  $q$  in  $Q_k$  do
(7)         if  $q - g_{\ell,B,k}$  is not in  $T$  then
(8)            $T = T \cup \{q - g_{\ell,B,k}\}$ 
(9)         end if
(10)      end for
(11)    end for
(12)  end for
(13)  Assign  $d_B$  to be the element in  $T$  such that the sum of the
      cost of neighboring communication over each single assignment
      block with respect to  $B$  is the minimum
(14)  for each neighboring node  $s_k$  (in  $V_s$ ) of  $B$  do
(15)    for each  $g_{\ell,B,k}$  in  $u_{B,k}$  do
(16)      if  $d_B + g_{\ell,B,k}$  is not in  $Q_k$  then
(17)         $Q_k = Q_k \cup \{d_B + g_{\ell,B,k}\}$ 
(18)      end if
(19)    end for
(20)  end for
(21) end for

```

Figure 4.19. The enhanced alignment graph offset alignment algorithm

The major difference between the EAGOA and AGOA algorithms is how to es-

timate the sum of the cost of neighboring communication (line (13)). In the AGOA algorithm, the sum of the cost is estimated based on each **FORALL** assignment where a particular array is referenced. However, in the EAGO algorithm, the sum of the cost is estimated based on each single assignment block with respect to that particular array. Therefore, the EAGO algorithm prevents the selection of alignment offset from the adverse impact of redundant neighboring communication. This feature makes the EAGO algorithm superior to the AGOA algorithm. Example 15 can be used to illustrate the idea.

In Figure 4.18, A has the maximum connectivity degree. Therefore, A is selected first. Executing lines (14)-(20) in the EAGO algorithm, we have $Q_1 = \{d_A\}$, $Q_2 = \{d_A+4, d_A+5\}$, $Q_3 = \{d_A+3\}$, $Q_4 = \{d_A+3\}$, $Q_5 = \{d_A\}$, and $Q_6 = \{d_A+1\}$. Assume that B is selected next. Since B is referenced in statements s_2 , s_3 , s_4 , and s_6 , T is only constructed based on Q_2 , Q_3 , Q_4 , and Q_6 . By lines (3)-(12), we have $T = \{d_A+1, d_A+3, d_A+4, d_A+5\}$. Statements s_2 , s_3 , and s_4 comprise the single assignment statement with regard to arrays A , X , and Z . When redundant neighboring communication is removed, the sum of the cost of neighboring communication in statements s_2 , s_3 , and s_5 can be modeled by the piecewise linear function $\langle 3, 5 \rangle$ with respect to $d_B - d_A$. On the other hand, the cost of neighboring communication generated by statement s_6 is modeled by piecewise linear function $\langle 1 \rangle$ with respect to $d_B - d_A$. Since statement s_6 is not in the same single assignment block in which statements s_2 , s_3 , and s_4 are (regarding to array A), the sum of the cost in line (13) is equal to $\langle 3, 5 \rangle + \langle 1 \rangle + \langle 1 \rangle$. Note that the cost of neighboring communication is weighted by 2 for statement s_6 . Therefore, the minimal value of $\langle 3, 5 \rangle + \langle 1 \rangle + \langle 1 \rangle$ can be achieved when $d_B = d_A + 1$. In contrast, if the AGOA algorithm is used, the sum of the cost of neighboring communication is estimated based on each **FORALL** assignment. In other words, the cost sum would be $c_{B,2} + c_{B,3} + c_{B,4} + c_{B,6}$, where $c_{B,2} = \langle 4, 5 \rangle$, $c_{B,3} = \langle 3 \rangle$, $c_{B,4} = \langle 3 \rangle$, and $c_{B,6} = \langle 1 \rangle + \langle 1 \rangle$ regarding

to $d_B - d_A$. This wrong cost estimation leads to the solution $d_B = d_A + 3$ which minimizes the value of $\langle 4, 5 \rangle + \langle 3 \rangle + \langle 3 \rangle + \langle 1 \rangle + \langle 1 \rangle$.

The extra time complexity in the EAGOA algorithm is spent in identifying single assignment blocks for each array variable. Using the dataflow analysis technique proposed in [64], the extra time complexity is $O(|V_a||V_s|)$. Thus, the overall time complexity is

$$O(|V_a||V_s| + \sum_{(B,s_k)} |u_{B,k}| \log \sum_{(B,s_k)} |u_{B,k}|)$$

CHAPTER 5

Data Distribution

The task of data distribution is to determine the mapping of the template elements onto the processors. Data elements are assigned to the processor to which the mapped template element is assigned. The goal of data distribution is to both reduce neighboring communication and increase processor workload balance. In this chapter, segment distribution is introduced as the best distribution pattern for data distribution within a single dimension. An optimal processor allocation algorithm is proposed to further minimize the overall cost of neighboring communication across multiple dimensions of the template array.

5.1 Segment Distribution

In this section, we study data distribution with regard to a single dimension in the template array. The issues involved in data distribution across multiple dimensions in the template array will be addressed in the next section.

5.1.1 The Limitation of Existing Distribution Types

We illustrate the limitation of existing distribution types by using Example 16 (Figure 5.1).


```

FORALL( $i_1 = 0 : n - 1, i_2 = i_1 : n - 2$ )
 $s_1:$     $B(i_1, i_2 + 1) = B(n - 1 + i_1 - i_2, i_2) * * 2$ 
END FORALL

```

Figure 5.1. Example 16: An Electromagnetic benchmark loop

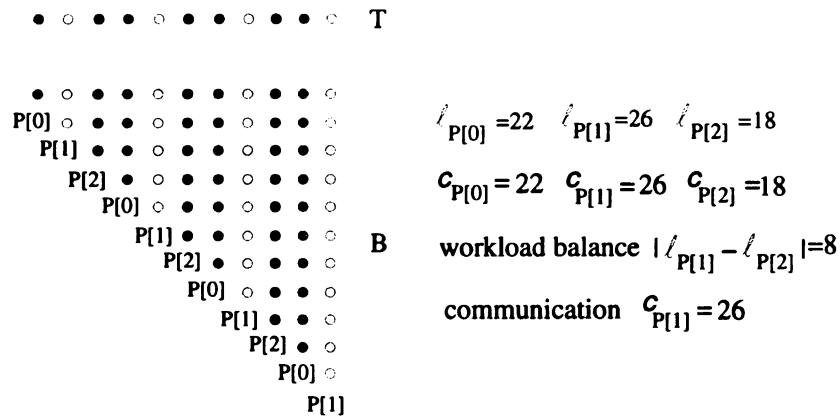
In Example 16, array B is distributed in columns. In other words, the distribution function δ_B is defined as follows:

$$t = \delta_B\left(\begin{pmatrix} b_1 \\ b_2 \end{pmatrix}\right) = b_2$$

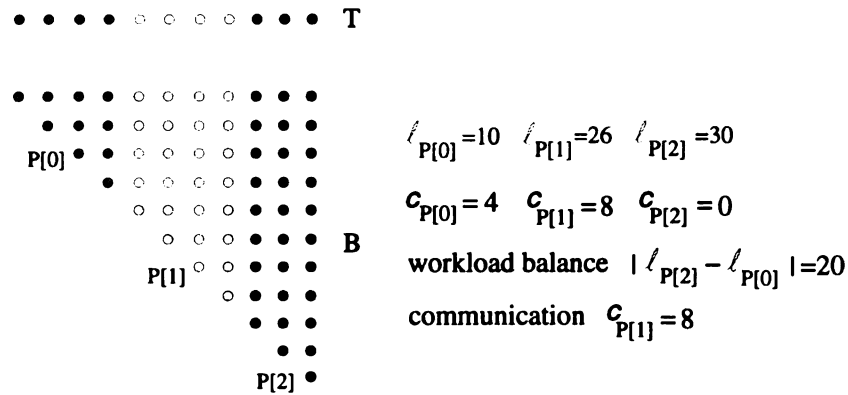
where $T(t)$ is a template element and $B(b_1, b_2)$ is a B element. By Proposition 1, assignment s_1 is free of reorganization communication. However, neighboring communication cannot be avoided. By Proposition 5, the basic cost of neighboring communication generated by assignment s_1 is 1. An effective element is a data element that is used or defined in assignment s_1 . Limited by the loop boundary, the effective elements form the upper triangle in the two-dimensional data space. Thus, density function ω_B can be defined as follows:

$$\omega_B(t) = t \quad \text{where } T(t) \text{ is a template element}$$

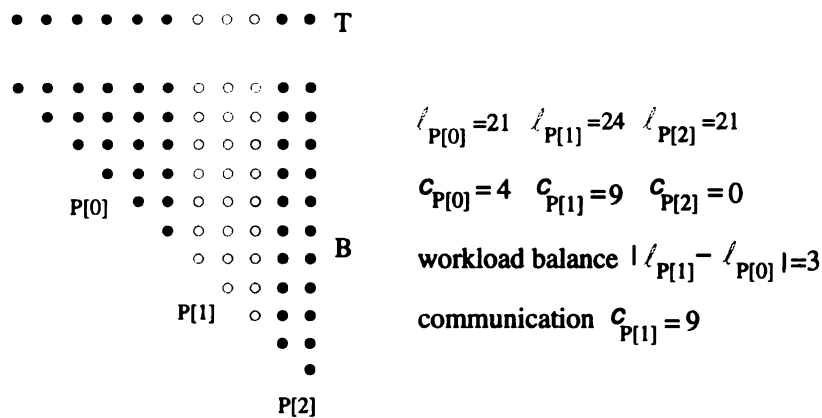
Figure 5.2 shows three different types of data distribution for Example 16: cyclic distribution, block distribution, and segment distribution. In Figure 5.2, array B is declared as 11×11 . Template T is one-dimensional array with 11 elements. There are three processors available, denoted as $P(0)$, $P(1)$, and $P(2)$. Figure 5.2 shows the distribution patterns of both the template elements and the effective elements in array B .



(a) cyclic distribution



(b) block distribution



(c) segment distribution

Figure 5.2. Different types of data distribution for Example 16

Since the RHS computation is uniform among all the LHS elements, workload on a processor is proportional to the number of the LHS elements that the processor owns. For simplicity, $\ell_{P(i)}$, the workload on processor $P(i)$, is represented by the number of the LHS elements that processor $P(i)$ owns. The quality of processor workload balance imposed by a particular distribution pattern can be evaluated by the variance of workload,

$$\max_{0 \leq i \neq j \leq 2} \{|\ell_{P(i)} - \ell_{P(j)}|\}$$

The cost of neighboring communication generated on processor $P(i)$ is denoted as $c_{P(i)}$. The value of $c_{P(i)}$ is determined by the number of remote elements that processor $P(i)$ accesses in executing assignment s_1 . Since the basic cost of neighboring communication in Example 16 is 1, each processor will access one solid line owned by a remote processor. However, since the length of solid lines are varied, the value of $c_{P(i)}$ for different processor $P(i)$ is different. Therefore, the cost of neighboring communication is measured by

$$\max_{0 \leq i \leq 2} \{c_{P(i)}\}$$

Figure 5.2(a) shows cyclic distribution in which the template elements are distributed to three processors in the round-robin fashion. Though it attains a good processor workload balance ($|\ell_{P(1)} - \ell_{P(2)}| = 8$), cyclic distribution generates an extremely high cost of neighboring communication ($c_{P(0)} = 22$). Figure 5.2(b) shows block distribution in which the template elements are distributed to three processors in the block fashion. The number of template elements each processor owns is equal. Though it attains a low cost of neighboring communication ($c_{P(1)} = 8$), the poor quality of processor workload balance ($|\ell_{P(2)} - \ell_{P(1)}| = 20$) is hard to accept. A new distribution pattern, segment distribution, is employed in Figure 5.2(c). In segment distribution, the template elements are consecutively assigned to the processors. However, the number of template elements owned by different processors can be

varied. In Figure 5.2(c), the number of template elements owned by each processor is nicely arranged such that processor workload is balanced ($|\ell_{P(2)} - \ell_{P(1)}| = 3$) and the cost of neighboring communication is small ($c_{P(1)} = 9$).

The study of Example 16 reveals two important facts. First, as shown in Figure 5.2(a), with cyclic distribution, a remote element is read for writing each local element in Example 1. This implies that the cost of neighboring communication generated by an inappropriate distribution pattern can be as much as the cost of reorganization communication generated by mismatched alignment bases. Second, the impact of neighboring communication can be significantly reduced by consecutively assigning template elements to processors. As shown in Figure 5.2(b) and (c), template elements are consecutively assigned to processors in both block distribution and segment distribution. As a result, the cost of neighboring communication is small, $c_{P(1)} = 8$ in Figure 5.2(b) and $c_{P(1)} = 9$ in Figure 5.2(c). The relative difference of the cost of neighboring communication between block distribution and segment distribution will be smaller when the size of arrays increases.

5.1.2 Segment Distribution

As discussed in the previous section, in segment distribution, template elements are consecutively assigned to processors in order to minimize the impact of neighboring communication. The length of the segment assigned to each processor may be varied. This feature gives segment distribution a big flexibility to exploit the maximum processor workload balance.

In data parallel programs, data elements are typically written by the processor which owns them. As a result, the workload assigned to a processor is determined by the LHS elements owned by that processor. Therefore, the distribution strategy for the LHS array elements on each assignment statement determines processor workload balance.

Given total q processors, template array $T(0 : n - 1)$ is partitioned into q pieces such that each piece $T(m_i : m_{i+1} - 1)$ is owned by processor $P(i)$ where $0 \leq i \leq q - 1$, $m_0 = 0$, and $m_{q-1} = n$. Our goal is to find the value of those break points m_i ($0 \leq i \leq q - 1$) such that the workload imposed by elements in each segment can be equal. This can be formalized as follows:

$$\sum_{t=m_0}^{m_1-1} \omega_A(t) = \sum_{j=m_1}^{m_2-1} \omega_A(t) = \dots = \sum_{j=m_{q-2}}^{m_{q-1}-1} \omega_A(t) \quad (5.1)$$

where A serves as the LHS array in a **FORALL** assignment and $\omega_A(t)$ is the density function of A .

In Equation 5.1, the density $\omega_A(t)$ only shows the number of LHS A elements mapped to the template element $T(t)$. Strictly speaking, the number of the LHS elements may not be proportional to the workload imposed by the LHS elements. For example, in the Linpack TQL2 loop (Figure 5.3), array Z is referenced on the LHS. Since Z is the one-dimensional array, there is only one Z element mapped to each template element. However, based on the inner loop indexed by i_2 , the workload imposed by different Z elements is obviously different: total nn inner loop iterations are executed for element $Z(1)$, while only one inner loop iteration is executed for element $Z(nn)$.

```

      FORALL( $i_1 = 2 : nn$ )
 $s_1:$        $Z(i_1) = Z(i_1 + 1)$ 
          DO ( $i_2 = i_1, nn$ )
 $s_2:$        $Z(i_1) = Z(i_1) + S * (H(i_1, i_2) + F * H(i_1 - 1, i_2))$ 
          END DO
      END FORALL

```

Figure 5.3. Linpack TQL2 benchmark loop

In order to precisely model the workload, we modify the definition of density function ω_A as follows:

$$\omega_A(t) = \sum_{\delta_A(a)=t} \pi_A(a)$$

where $T(t)$ is a template element, $A(a)$ is a data element, and $\pi_A(a)$ represents the computation estimate to define the LHS element $A(a)$. The computation can be estimated based on the number of integer or floating-point operations required by the RHS expression evaluation. For instance, in assignment s_2 of Example 16,

$$\pi_Z(z) = (nn - z + 1)(2t_* + 2t_+ + 2t_l + t_s)$$

where t_* represents a floating-point multiplication, t_+ represents a floating-point addition, t_m represents a memory load operation, and t_s represents a memory store operation. The amount of computation can be further normalized by converting various operations to the equivalent number of clock cycles.

In most scientific applications, the density function of a LHS array is typically a constant or an affine function with a single variable. Figure 5.4 shows a few common loop patterns extracted from scientific application programs. In Figure 5.4, A is a two-dimensional array and B is a one-dimensional array. The template array is one-dimensional and A is distributed in columns. In Figure 5.4(d), (e), (f), and (h), the inner loop is sequential. In Figure 5.4(a), $\omega_A(t) = t$ for element $T(t)$. In Figure 5.4(b), $\omega_A(t) = n - 1 - t$ for element $T(t)$. In Figure 5.4(c), $\omega_A = \min\{t, n - 1 - t\}$ for element $T(t)$. In Figure 5.4(d), $\omega_B(t) = t$ for element $T(t)$. In Figure 5.4(e), $\omega_B(t) = n - 1 - t$ for element $T(t)$. In Figure 5.4(f), $\omega_B = \min\{t, n - 1 - t\}$ for element $T(t)$. In Figure 5.4(g), $\omega_B(t) = 1$ for element $T(t)$. In Figure 5.4(h), $\omega_B(t) = n$ for element $T(t)$. In Figure 5.4(i), $\omega_A(t) = n$ for element $T(t)$.

For this reason, in this thesis we assume that $\omega_A = aj + b$ for each template element $T(j)$. Here, both a and b are fixed integers. Suppose there are total q

processors. Template elements $T(0 : n - 1)$ are partitioned into q segments such that each segment $T(m_i : m_{i+1} - 1)$ is owned by processor $P(i)$ where $0 \leq i \leq q - 1$, $m_0 = 0$, and $m_{q-1} = n$. Break points m_i ($0 \leq i \leq q - 1$) are optimized such that the workload imposed by each segment is equal. In other words,

$$\sum_{j=m_0}^{m_1-1} aj + b = \sum_{j=m_1}^{m_2-1} aj + b = \dots = \sum_{j=m_{q-2}}^{m_{q-1}-1} aj + b$$

<pre>FORALL(i₁=0:n-1) FORALL(i₂=0:i₁) A(i₁, i₂) = ... END FORALL END FORALL</pre> <p>(a)</p>	<pre>FORALL(i₁=0:n-1) FORALL(i₂=i₁:n-2) A(i₁, i₂) = ... END FORALL END FORALL</pre> <p>(b)</p>	<pre>FORALL(i₁=0:n-1) FORALL(i₂=i₁:n-1-i₁) A(i₁, i₂) = ... END FORALL END FORALL</pre> <p>(c)</p>
<pre>FORALL(i₁=0:n-1) FOR(i₂=0:i₁) B(i₁) = ... END FOR END FORALL</pre> <p>(d)</p>	<pre>FORALL(i₁=0:n-1) FOR(i₂=i₁:n-2) B(i₁) = ... END FOR END FORALL</pre> <p>(e)</p>	<pre>FORALL(i₁=0:n-1) FORALL(i₂=i₁:n-1-i₁) B(i₁) = ... END FORALL END FORALL</pre> <p>(f)</p>
<pre>FORALL(i₁=0:n-1) B(i₁) = ... END FORALL</pre> <p>(g)</p>	<pre>FORALL(i₁=0:n-1) FOR(i₂=0:n-2) B(i₁) = ... END FOR END FORALL</pre> <p>(h)</p>	<pre>FORALL(i₁=0:n-1) FORALL(i₂=0:n-1) A(i₁, i₂) = ... END FORALL END FORALL</pre> <p>(i)</p>

Figure 5.4. Some common loop patterns

We find these optimal break points by extending density function ω_A to all real numbers within the range $[0, n - 1]$. The density function is extended to $\omega_A = av + b$ for any real v in $[0, n - 1]$. Segment $[0, n - 1]$ is partitioned into q pieces of $[v_i, v_{i+1}]$

where $0 \leq i \leq q-2$, $v_0 = 0$, and $v_{q-1} = n-1$. We need to find break points v_i ($0 \leq i \leq q-1$) such that

$$\int_{v_0}^{v_1} av + b = \int_{v_1}^{v_2} av + b = \dots = \int_{v_{q-2}}^{v_{q-1}} av + b$$

This implies

$$\int_{v_0}^{v_i} av + b = \frac{i}{q} \int_{v_0}^{v_{q-1}} av + b$$

or

$$\int_0^{v_i} av + b = \frac{i}{q} \int_0^{n-1} av + b$$

Solving the integration, we get

$$\begin{aligned} \frac{1}{2}av_i^2 + bv_i &= \frac{i}{q} \left(\frac{1}{2}a(n-1)^2 + b(n-1) \right) \\ v_i^2 + \frac{2b}{a}v_i &= \frac{i}{q} \left((n-1)^2 + \frac{2b}{a}(n-1) \right) \end{aligned}$$

Solving the above equation, we get the solution of v_i as

$$\begin{aligned} v_i &= \frac{-\frac{2b}{a} + \sqrt{\frac{4b^2}{a^2} + 4\frac{i}{q}((n-1)^2 + \frac{2b}{a}(n-1))}}{2} \\ v_i &= \sqrt{\frac{b^2}{a^2} + \frac{i}{q}((n-1)^2 + \frac{2b}{a}(n-1))} - \frac{b}{a} \end{aligned} \quad (5.2)$$

Since v_i may not be an integer grid, m_i is chosen as an integer approximation of v_i by

$$m_i = \lceil v_i \rceil = \lceil \sqrt{\frac{b^2}{a^2} + \frac{i}{q}((n-1)^2 + \frac{2b}{a}(n-1))} - \frac{b}{a} \rceil$$

On the other hand, we should know what processor a particular template element $T(j)$ is assigned to. Suppose that $T(j)$ is owned by processor $P(i)$. Therefore, we

should have $v_i \leq j \leq v_{i+1}$. Thus, by Equation 5.2, we have

$$\begin{aligned} & \sqrt{\frac{b^2}{a^2} + \frac{i}{q}((n-1)^2 + \frac{2b}{a}(n-1))} - \frac{b}{a} \leq j \\ j & \leq \sqrt{\frac{b^2}{a^2} + \frac{i+1}{q}((n-1)^2 + \frac{2b}{a}(n-1))} - \frac{b}{a} \end{aligned}$$

This can be re-written as

$$\begin{aligned} i & \leq \frac{q(\frac{2b}{a}j + j^2) - \frac{2b}{a}(n-1)}{(n-1)^2} \\ \frac{q(\frac{2b}{a}j + j^2) - \frac{2b}{a}(n-1)}{(n-1)^2} - 1 & \leq i \end{aligned}$$

In other words, we have

$$\frac{q(\frac{2b}{a}j + j^2) - \frac{2b}{a}(n-1)}{(n-1)^2} - 1 \leq i \leq \frac{q(\frac{2b}{a}j + j^2) - \frac{2b}{a}(n-1)}{(n-1)^2}$$

Since it should be an integer, i can be selected as

$$i = \lfloor \frac{q(\frac{2b}{a}j + j^2) - \frac{2b}{a}(n-1)}{(n-1)^2} \rfloor$$

Overall, we summerize the above results as follows:

Proposition 12 *Assume that $\omega_A(j) = aj + b$ is the density function conducted by the mapping from a data array A to a one-dimensional template array $T(0 : n - 1)$. Given the q processors, segment distribution of T can be specified as follows:*

1. *Processor $P(i)$ ($0 \leq i \leq q - 1$) owns the consecutive elements $T(m_i : m_{i+1} - 1)$ where*

$$\begin{aligned} m_i & = \lceil \sqrt{\frac{b^2}{a^2} + \frac{i}{q}((n-1)^2 + \frac{2b}{a}(n-1))} - \frac{b}{a} \rceil \\ m_{i+1} & = \lceil \sqrt{\frac{b^2}{a^2} + \frac{i+1}{q}((n-1)^2 + \frac{2b}{a}(n-1))} - \frac{b}{a} \rceil \end{aligned}$$

2. *Element $T(j)$ is owned by processor $P(i)$ where*

$$i = \lfloor \frac{q(\frac{2b}{a}j + j^2) - \frac{2b}{a}(n-1)}{(n-1)^2} \rfloor$$

The specification of segment distribution in Figure 5.2(c) can be obtained using Proposition 12. In Figure 5.2(c), the template has 11 elements $T(0 : 11 - 1)$ and thus $n = 11$. $\omega_A(j) = j$ for each element $T(j)$ and thus $a = 1$ and $b = 0$. Using Proposition 12, we obtain that

$$\begin{aligned} m_0 &= 0 \\ m_1 &= \lceil 10\sqrt{\frac{1}{3}} \rceil = 6 \\ m_2 &= \lceil 10\sqrt{\frac{2}{3}} \rceil = 9 \\ m_3 &= 11 - 1 \end{aligned}$$

Therefore, $T(0 : 6 - 1)$ is distributed to processor $P(0)$, $T(6 : 9 - 1)$ is distributed to processor $P(1)$, and $T(9 : 11 - 1)$ is distributed to processor $P(2)$. Given element $T(j)$, it is owned by processor $P(i)$ where

$$i = \lfloor 3(\frac{j}{10})^2 \rfloor$$

5.1.3 Multiple-Variable Density Function

If the template is a multi-dimensional array, a density function can be a function of multiple variables each of which represents a distinct dimension in the template array. In this case, we cannot simply formulate the requirement of processor workload balance by using Equation 5.1. In this section, we study data distribution regarding to multiple-variable density function.

Assume that the template is represented by a m -dimensional array $T(0 : n_1 - 1, 0 : n_2 - 1, \dots, 0 : n_{m-1} - 1)$. Processors are represented by a m -dimensional array $P(0 : q_1 - 1, 0 : q_2 - 1, \dots, 0 : q_{m-1} - 1)$. Each element $P(k_0, k_1, \dots, k_{m-1})$ represents a particular processor. Each dimension of the template array is distributed to processors in segment fashion. Assume that the j -th dimension of the template is partitioned into q_j blocks: $T(\dots, n_{j,0} : n_{j,1} - 1, \dots)$, $T(\dots, n_{j,1} : n_{j,2} - 1, \dots)$, \dots , and $T(\dots, n_{j,q_j-2} : n_{j,q_j-1} - 1, \dots)$ where $n_{j,0} = 0$ and $n_{j,q_j-1} = n_j$. Therefore, the block of the template elements which processor $P(k_0, k_1, \dots, k_{m-1})$ owns is $T(n_{0,k_0} : n_{0,k_0+1} - 1, n_{1,k_1} : n_{1,k_1+1} - 1, \dots, n_{1,k_{m-1}} : n_{1,k_{m-1}+1} - 1)$ where $0 \leq k_0 \leq q_0 - 1$, $0 \leq k_1 \leq q_1 - 1$, \dots , and $0 \leq k_{m-1} \leq q_{m-1} - 1$.

Assume that A is the LHS array referenced in **FORALL** assignment s_h . Let ω_A be the density function with respect to A . Therefore, $\ell_{P(k_0, k_1, \dots, k_{m-1})}$, the workload on processor $P(k_0, k_1, \dots, k_{m-1})$, can be formalized as follows:

$$\ell_{P(k_0, k_1, \dots, k_{m-1})} = \sum_{j_0=n_{0,k_0}}^{n_{0,k_0+1}-1} \sum_{j_1=n_{1,k_1}}^{n_{1,k_1+1}-1} \dots \sum_{j_{m-1}=n_{1,k_{m-1}}}^{n_{1,k_{m-1}+1}-1} \omega_A(j_0, j_k, \dots, j_{m-1}) \quad (5.3)$$

In Equation 5.3, template element $T(j_0, j_1, \dots, j_{m-1})$ is owned by processor $P(k_0, k_1, \dots, k_{m-1})$. $\omega_A(j_0, j_k, \dots, j_{m-1})$, the density of template element $T(j_0, j_1, \dots, j_{m-1})$, can be estimated as follows:

$$\omega_A(j_0, j_k, \dots, j_{m-1}) = \sum_{\delta_A(a_0, a_1, \dots, a_h) = T(j_0, j_k, \dots, j_{m-1})} \pi_A(a_0, a_1, \dots, a_h)$$

where $\pi_A(a_0, a_1, \dots, a_h)$ is the estimate of the RHS expression evaluation in order to define the LHS element $A(a_0, a_1, \dots, a_h)$.

Similar to Equation 5.3, ℓ_{s_h} , the overall workload over all LHS elements with

respect to statement s_h , can be obtained as follows:

$$\ell_{s_h} = \sum_{j_0=0}^{n_0-1} \sum_{j_1=1}^{n_1-1} \dots \sum_{j_{m-1}=0}^{n_{m-1}-1} \omega_A(j_0, j_1, \dots, j_{m-1}) \quad (5.4)$$

Therefore, the problem of finding the optimal segment size can be formalized as follows.

Proposition 13 *If the size of segment $T(n_{0,k_0} : n_{0,k_0+1} - 1, n_{1,k_1} : n_{1,k_1+1} - 1, \dots, n_{1,k_{m-1}} : n_{1,k_{m-1}+1} - 1)$ owned by processor $P(k_0, k_1, \dots, k_{m-1})$ is optimal, then the following condition must be satisfied.*

$$\ell_{(p_{k_0}, p_{k_1}, \dots, k_{m-1})} = \frac{\ell_{s_h}}{q}$$

where $q = q_0 q_1 \dots q_{m-1}$ is the number of total processors. A distribution pattern is optimal if the size of the segment assigned to each processor is optimal.

By Equation 5.4, the workload distribution depends on the property of density function ω_A . For certain density function ω_A , there may not exist an optimal distribution pattern which can satisfy Proposition 13. For example, consider two-dimensional template $T(0 : n_0 - 1, 0 : n_1 - 1)$ with density function

$$\omega_A(j_0, j_1) = \begin{cases} 1 & \text{where } 0 \leq j_0 = j_1 \leq \min\{n_0, n_1\} \\ 0 & \text{otherwise} \end{cases}$$

Therefore, $\min\{n_0, n_1\}$ is the overall workload. However, as shown in Figure 5.5(a), there always exists some data block on which workload is zero as long as there are two or more processors assigned to each dimension of the template T . Therefore, the optimal distribution pattern does not exist if T is partitioned along both row and column dimensions.

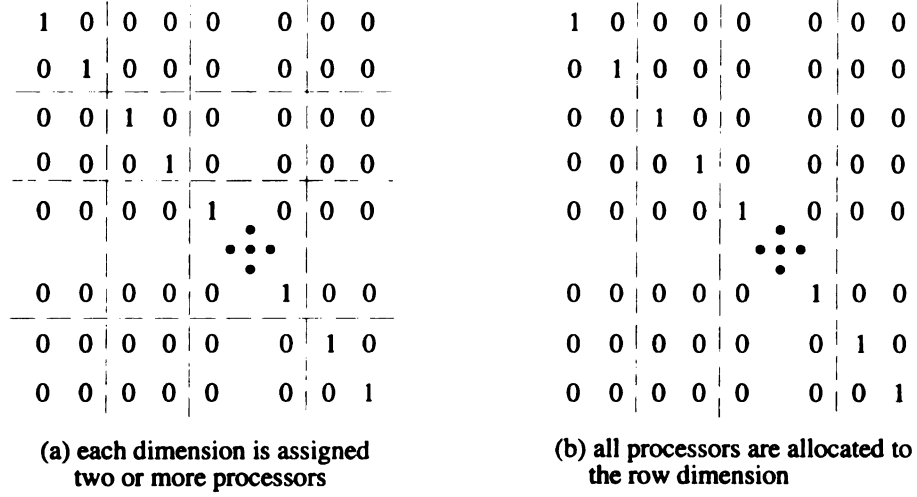


Figure 5.5. An example of no optimal distribution pattern

To resolve this problem, we convert the multiple-variable density function into a new single-variable density function and then use the proposed segment distribution to assign the template elements along that dimension which index is the variable in the new density function. This idea can be formalized as follows. Assume that template $T(0 : n_1 - 1, 0 : n_2 - 1, \dots, 0 : n_{m-1} - 1)$ is m -dimensional array. A density function ω_A is a function of dimensions i_1, i_2, \dots , and i_h where $2 \leq h \leq m$. No processor is allocated to dimension i_2, i_3, \dots , and i_h . The template elements on the i_1 -th dimension is distributed in segment distribution by the new density function ω'_A :

$$\omega'_A(i_1) = \sum_{i_2=0}^{n_{i_2}-1} \sum_{i_3=0}^{n_{i_3}-1} \dots \sum_{i_h=0}^{n_{i_h}-1} \omega_A(i_1, i_2, i_3, \dots, i_h)$$

For the example shown in Figure 5.5, since ω_A is a function of variables j_0 and j_1 , no processor is allocated to dimension j_1 . As shown in Figure 5.5(b), template elements are only distributed to processors along the j_0 -th dimension with the new density function

$$\omega'_A(j_0) = \sum_{j_1=1}^{n_1} \omega_A(j_0, j_1) = 1$$

Since ω'_A has uniform density, the number of columns allocated to each processor should be equal.

5.1.4 Experimental Results

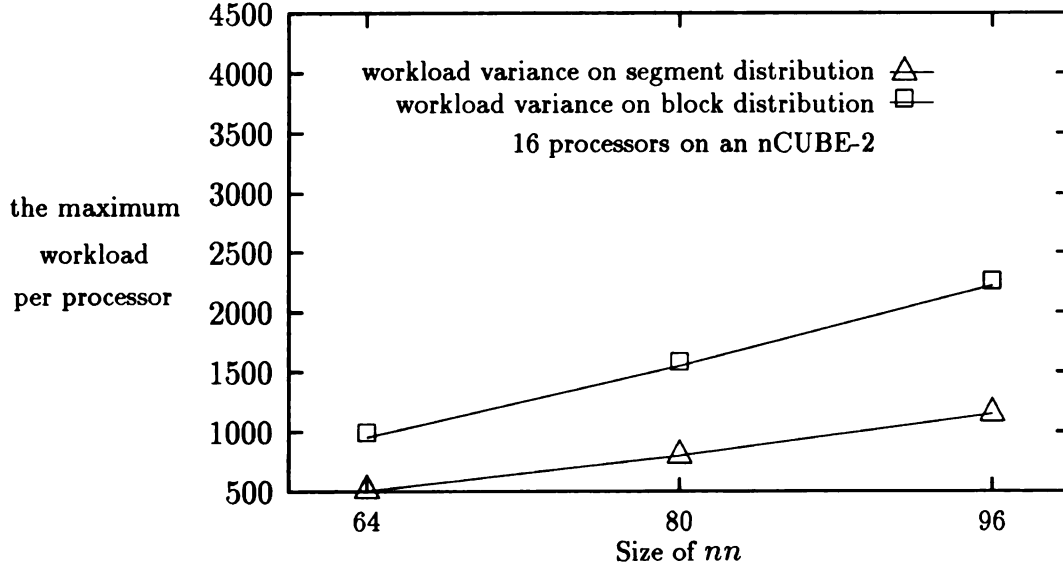


Figure 5.6. Comparison of processor workload balance between block distribution and segment distribution

Figure 5.6 shows the comparison of processor workload variance between block distribution and segment distribution by running the Linpack TQL2 loop (Figure 5.3) on a 16-node nCUBE-2. In our experiment, the number of processors is fixed as 16 and the problem size is varied. Figure 5.6 only shows the workload variance of each distribution pattern. The cost of neighboring communication involved in accessing remote elements is not included. Segment distribution outperforms block distribution in all cases.

5.2 Virtual Processor Allocation

If the data alignment analysis generates multiple aligned-base groups, the template must be a multi-dimensional array. Segment distribution is used to specify the distribution pattern of template elements regarding to each dimension of the template array. Segment distribution maximizes processor workload balance regardless of the number of assigned processors. Therefore, as long as template elements are distributed in segment fashion along each dimension of the template array, processor workload will be well-balanced. However, though segment distribution minimizes neighboring communication regarding to each dimension of the template array, the sum of the cost of neighboring communication across all dimensions depends on the number of processors assigned to each dimension of the template array.

5.2.1 Reducing the Overall Neighboring Communication

Figure 5.7 shows the impact of processor allocation on the overall cost of neighboring communication in Example 4 (Figure 2.11), in which the template is a two-dimensional array.

In Figure 5.7, arrays W , Z , and T (the template array) are declared as 6×12 . Alignment functions δ_W and δ_Z are specified as follows:

$$\begin{pmatrix} t_1 \\ t_2 \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$$

$$\begin{pmatrix} t_1 \\ t_2 \end{pmatrix} = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix}$$

where $W(w_1, w_2)$ is a W element, $Z(z_1, z_2)$ is a Z element, and $T(t_1, t_2)$ is a template element. Thus, there are two aligned-base groups: the row dimension of W is aligned with the row dimension of Z ; the column dimension of W is aligned with the column

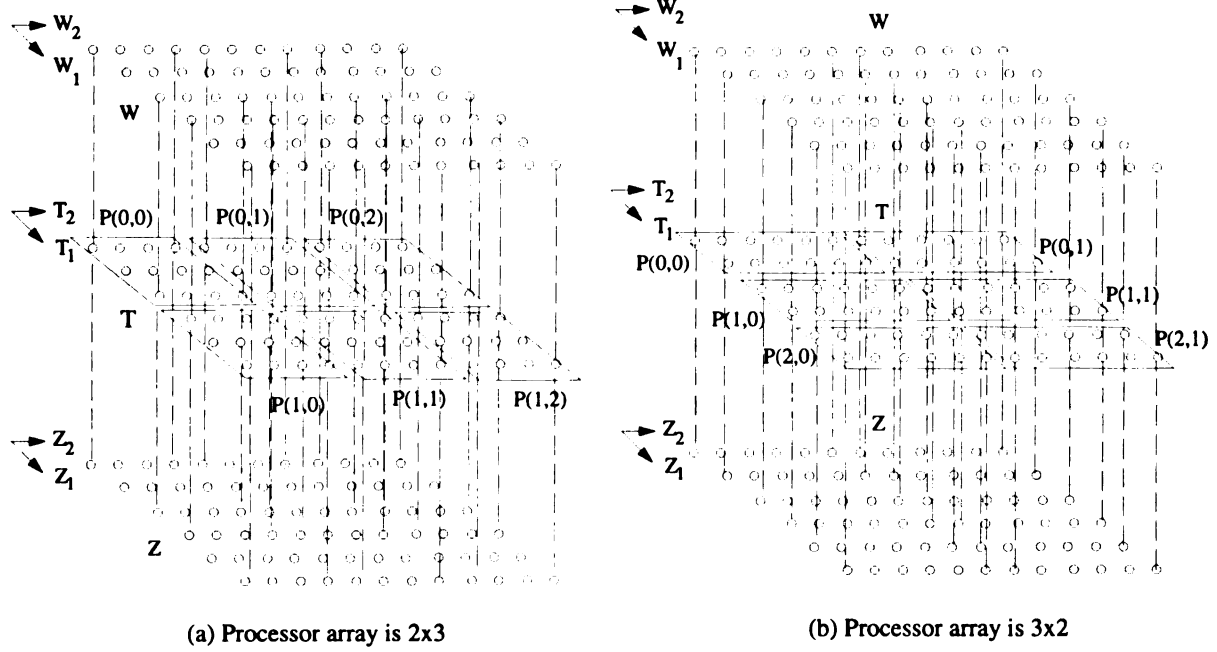


Figure 5.7. Two different processor allocation patterns for Example 4

dimension of Z . There are total 6 processors available. As shown in Figure 5.7, template T is distributed to processors in both row and column dimensions. Each dimension is distributed in segment fashion. Since each template element is mapped by one W element and one Z element, the density function of both W and Z is uniform. Therefore, each dimension in fact is distributed in block fashion. Figure 5.7(a) and (b) show two different processor allocation patterns.

By the construction of array subscripts in assignment s_1 (Example 4), one remote Z element must be referenced to define $W(i_1, i_2)$ if $W(i_1, i_2)$ is a boundary element in the data block owned by the local processor. For example, in Figure 5.7(a), in order to write the LHS element $W(1, 4)$ owned by processor $P(0, 1)$, the remote RHS element $Z(1, 3)$ owned by processor $P(0, 0)$ is read. Moreover, two remote Z elements will be accessed to define $W(i_1, i_2)$ if $W(i_1, i_2)$ resides on the corner of the data block owned by the local processor. For example, in Figure 5.7(b), in order to write the LHS element $W(4, 5)$ owned by processor $P(2, 0)$, the remote RHS elements $Z(4, 6)$

owned by processor $P(2,1)$ and $Z(3,5)$ owned by processor $P(1,0)$ are read. This implies that the basic cost of neighboring communication in either aligned-base group is 1. The cost of neighboring communication regarding to the aligned-base group represented by the row dimension of the template is the number of the elements on the vertical boundary. Similarly, the cost of neighboring communication regarding to the aligned-base group represented by the column dimension of the template is the number of the elements on the horizontal boundary. Thus, the overall cost of neighboring communication is equal to the number of elements on the circumference.

Since each block is two-dimensional, given the total number of processors, the length of the circumference in each block depends on the shape of template array and the shape of processor array. In Figure 5.7(a), the processor array is specified as 2×3 , denoted as $P(0 : 1, 0 : 2)$. Processors $P(0,1)$ and $P(1,1)$ access 10 remote elements each. The other four processors access 7 remote elements each. In Figure 5.7(b), the processor array is specified as 3×2 , denoted as $P(0 : 2, 0 : 1)$. Processors $P(1,0)$ and $P(1,1)$ access 14 remote elements each. The other four processors access 8 remote elements each. Therefore, the amount of neighboring communication in processor allocation 2×3 is much less than that in processor allocation 3×2 . Given the total number of processors, the shape of processor array makes difference in reducing the overall neighboring communication.

5.2.2 Optimal Processor Allocation

In this section, we propose the optimal processor allocation approach such that overall neighboring communication can be minimized by given a multi-dimensional template array and a total number of processors. The problem of processor allocation can be formalized as follows. Let $T(0 : n_1 - 1, 0 : n_2 - 1, \dots, 0 : n_{m-1} - 1)$ be the m -dimensional template array. Assume that c_k is the basic cost of neighboring communication regarding to the aligned-base group represented by the k -th dimension of

the template ($0 \leq k \leq m-1$). There are totally q number of available processors.

Proposition 14 *If there exists a k ($0 \leq k \leq m-1$) such that $c_k = 0$, assign all processors to the k -th dimension of the template.*

If there does not exist a k ($0 \leq k \leq m-1$) such that $c_k = 0$, find the positive integer numbers q_1, q_2, \dots, q_m such that the total cost of neighboring communication

$$\begin{aligned} & c_0 \frac{n_1}{q_1} \frac{n_2}{q_2} \dots \frac{n_{m-1}}{q_{m-1}} + c_1 \frac{n_0}{q_0} \frac{n_2}{q_2} \dots \frac{n_{m-1}}{q_{m-1}} + \dots + c_k \frac{n_0}{q_0} \dots \frac{n_{k-1}}{q_{k-1}} \frac{n_{k+1}}{q_{k+1}} \dots \frac{n_{m-1}}{q_{m-1}} + \dots \\ & + c_{m-1} \frac{n_0}{q_0} \frac{n_1}{q_1} \dots \frac{n_{m-2}}{q_{m-2}} \end{aligned} \quad (5.5)$$

can be minimized where q_0, q_1, \dots, q_{m-1} are subject to

$$q_0 q_1 \dots q_{m-1} = q \quad (5.6)$$

In Equation 5.5, the value

$$\frac{n_0}{q_0} \dots \frac{n_{k-1}}{q_{k-1}} \frac{n_{k+1}}{q_{k+1}} \dots \frac{n_{m-1}}{q_{m-1}}$$

is the number of total elements on the k -th dimension boundary in the block owned by processor $P(k)$. The rest of this section shows how to find the solution of minimizing Equation 5.5.

Let n be the total number of the elements in the template array. Thus, we have $n = n_0 n_1 \dots n_{m-1}$. Substituting the value of q_k from Equation 5.6 into Equation 5.5, Formula 5.5 can be re-written as follows:

$$c_0 \frac{n}{n_0} \frac{q}{q_0} + c_1 \frac{n}{n_1} \frac{q}{q_1} + \dots + c_k \frac{n}{n_k} \frac{q}{q_k} + \dots + c_{m-1} \frac{n}{n_{m-1}} \frac{q}{q_{m-1}}$$

Since $c_k \frac{nq}{n_k}$ for $0 \leq k \leq m-1$ is constant, it is denoted as $y_k = c_k \frac{nq}{n_k}$. Furthermore, we define

$$x_k = \frac{1}{q_k} \quad \text{where } 0 \leq k \leq m-1$$

and

$$x = \frac{1}{q}$$

Therefore, the problem of finding the minimal value of Equation 5.5 is equivalent to that of finding the minimal value in the following equation

$$\begin{aligned} & y_0 x_0 + y_1 x_1 + \dots + y_{m-1} x_{m-1} \\ & \text{where } x_0 x_1 \dots x_{m-1} = x \end{aligned} \tag{5.7}$$

Note that

$$\sqrt[m]{y_0 x_0 y_1 x_1 \dots y_{m-1} x_{m-1}} \leq \frac{y_0 x_0 + y_1 x_1 + \dots + y_{m-1} x_{m-1}}{m}$$

The equality holds if and only if

$$y_0 x_0 = y_1 x_1 = \dots = y_{m-1} x_{m-1} \tag{5.8}$$

Therefore, the minimum value is achieved when Equation 5.8 holds. From Equation 5.8, we have

$$\begin{cases} x_1 = \frac{y_0}{y_1} x_0 \\ \dots \\ x_k = \frac{y_0}{y_k} x_0 \\ \dots \\ x_{m-1} = \frac{y_0}{y_{m-1}} x_0 \end{cases} \quad (5.9)$$

Substituting values of x_1, x_2, \dots , and x_{m-1} from Equation 5.9 into Equation 5.7, we have

$$x_0 = \sqrt[m]{\frac{x y_1 y_2 \dots y_{m-1}}{y_0^{m-1}}}$$

Substituting the value of x_0 in Equation 5.9, we get

$$\begin{cases} x_1 = \sqrt[m]{\frac{x y_0 y_2 \dots y_{m-1}}{y_1^{m-1}}} \\ \dots \\ x_k = \sqrt[m]{\frac{x y_0 \dots y_{k-1} y_{k+1} \dots y_{m-1}}{y_k^{m-1}}} \\ \dots \\ x_{m-1} = \sqrt[m]{\frac{x y_0 y_1 \dots y_{m-2}}{y_{m-1}^{m-1}}} \end{cases}$$

Therefore, by substitution of the real values of x_k, y_k , and x ($0 \leq k \leq m-1$), we have the following conclusion.

Proposition 15 *If there does not exist a k ($0 \leq k \leq m-1$) such that $c_k = 0$, then the total cost of neighboring communication*

$$\begin{aligned} & c_0 \frac{n_1}{q_1} \frac{n_2}{q_2} \dots \frac{n_{m-1}}{q_{m-1}} + c_1 \frac{n_0}{q_0} \frac{n_2}{q_2} \dots \frac{n_{m-1}}{q_{m-1}} + \dots + c_k \frac{n_0}{q_0} \dots \frac{n_{k-1}}{q_{k-1}} \frac{n_{k+1}}{q_{k+1}} \dots \frac{n_{m-1}}{q_{m-1}} + \dots \\ & + c_{m-1} \frac{n_0}{q_0} \frac{n_1}{q_1} \dots \frac{n_{m-2}}{q_{m-2}} \end{aligned}$$

can be minimized when

$$q_k = \sqrt[m]{\frac{qn_0 \dots n_{k-1}nn_{k+1} \dots n_{m-1}c_k^{m-1}}{n_k^{m-1}c_0 \dots c_{k-1}c_{k+1} \dots c_{m-1}}} \quad \text{where } 0 \leq k \leq m-1$$

The optimal processor allocation strategy proposed by Proposition 15 is machine independent and architecture independent. We assume that all processors are interconnected with a fully-connected network topology. The mapping from such a virtual model of fully-connected network to a particular machine architecture is beyond the scope of this thesis.

5.2.3 Performance Results

Figure 5.8 shows a livermore kernel 18 benchmark loop. In Figure 5.8, the size of all arrays is 256×256 . Array ZA serves the template. There are two aligned-base groups. The $(j-1)$ -th rows of ZU , ZP , ZQ , ZR , and ZM are aligned with the j -th row of ZA . The k -th rows of ZU , ZP , ZQ , ZR , and ZM are aligned with the k -th row of ZA . Let c_0 be the basic cost of neighboring communication with regard to the aligned-based group represented by the row dimension of ZA . Therefore, $c_0 = 1$ due to the access to remote element $ZR(j, k)$. Let c_1 be the basic cost of neighboring communication with regard to the aligned-based group represented by the column dimension of ZA . Therefore, $c_1 = 4$ due to the accesses to remote elements $ZU(j-1, k+1)$, $ZP(j-1, k+1)$, $ZQ(j-1, k+1)$, and $M(j-1, k+1)$.

Figure 5.9 compares the overall cost of neighboring communication in three different approaches on a 64-node nCUBE-2. Three approaches are the optimal allocation proposed in this thesis, the square allocation, and the row allocation. By Proposi-


```

FORALL( $j = 2 : 255, k = 2 : 255$ )
 $s_1:$     $ZA(j, k) = ZU(j - 1, k + 1) + ZU(j - 1, k) + (ZP(j - 1, k + 1) +$ 
         $ZQ(j - 1, k + 1) - ZP(j - 1, k) - ZQ(j - 1, k)) * (ZR(j, k) +$ 
         $ZR(j - 1, k)) / (ZM(j - 1, k) + ZM(j - 1, k + 1))$ 
END FORALL

```

Figure 5.8. A livermore kernel 18 benchmark loop

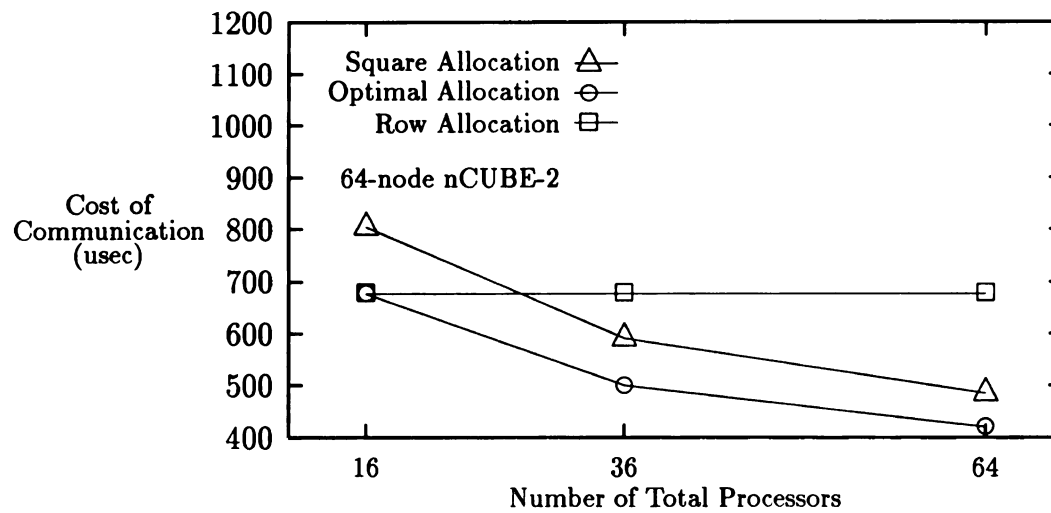


Figure 5.9. Comparison of different processor allocation strategies on a 64-node nCUBE-2

tion 15, the optimal processor allocation can be specified as

$$\begin{cases} q_0 = \sqrt{\frac{q}{4}} \\ q_1 = 2\sqrt{q} \end{cases}$$

where q_0 is the number of the processors assigned to the row dimension of ZA , q_1 is the number of the processors assigned to the column dimension of ZA , and q is the total number of processors. In the square allocation, the same number of processors is allocated to each dimension. In the row allocation, all processors are allocated to the row dimension since the basic cost on that dimension is smaller. The experimental results show that the optimal allocation outperforms the other two allocation strategies.

CHAPTER 6

Conclusions and Future Research

Data decomposition is critical to the performance of data parallel programs on scalable parallel computers. This thesis studies each fundamental phase in data decomposition and develops important theoretical results and practical algorithmic results to determine efficient data decomposition. In this chapter, we summarize the salient contributions made by this research and present interesting avenues for possible future research.

6.1 Research Contributions

The data decomposition model can be viewed as a two-level mapping of array elements to abstract processors. Data alignment determines what array elements are aligned relative to one another, and data distribution resolves how the group of aligned arrays is distributed onto the processors. The template array performs as an abstract index space, each grid in which represents a group of aligned array elements. The concept of the template makes the specification of the alignment and distribution clear. Depending on the alignment relationship as within dimension or across dimension, alignment can be classified into base alignment and offset alignment. If two arrays are mismatched in base alignment with respect to a reference, the whole data structure

of the array to be used is required to be reorganized and almost every array element will be involved in the data movement across the processors. As a result, an efficient base alignment has the first priority in the data decomposition analysis. After base alignment is determined, offset alignment specifies the offset between aligned array elements of various arrays with respect to the abstract index space. Since the alignment offset is a constant, the penalty of mismatched offset alignment with respect to a reference will be the data shift operation. The communication cost of the data shift operation can be greatly reduced if elements (in the template) are consecutively assigned to the processors. This requires that the pattern of data distribution be of the block distribution type. On the other hand, however, limited by the owner-computes rule in code generation, the requirement of processor workload balance favors the cyclic distribution type when the RHS computation is not uniform among all the LHS elements or when the number of the LHS elements mapped to different template elements is varied. This conflict between increasing workload balance and reducing data shift communication has become an open issue in the research of data distribution.

Using the affine alignment function [17], we have extended and developed the mathematical framework to model the relationship between data reference and base alignment. The cost estimate of reorganization communication has been studied based on different types of data reference. Data reference graph model is used to describe the impact of multiple data references and resolve the conflict of compatible alignment requirement. An efficient spanning tree algorithm addresses the fundamental issues in base alignment. Efficient base alignment algorithms are proposed to be incorporated with the RHS expression evaluation and dataflow optimizations. These contributions make this research unique from related research.

This thesis has made a significant contribution in the research area of offset alignment. The mathematical framework has been constructed to model the inter-

relationship between offset alignment and data reference. The cost of data shift communication has been estimated based on different types of data reference. In particular, the piecewise linear function is introduced to represent the cost of data shift movement with regard to multiple distinct instances of the same array which are accessed in the same statement. This cost model solved the accuracy problem in measuring the quantity of data shift movement, an unresolved problem left by other work in this area. Based on this cost model, the optimal post-alignment algorithm has been first proposed to exceed the limitation of the owner-computes rule and minimize the amount of data shift movement in each **FORALL** assignment after offset alignment is determined. Data reference graph model has been proposed to model the problem of offset alignment and develop efficient spanning tree algorithms. Like the base alignment analysis, the RHS expression evaluation and dataflow optimizations have been incorporated with the proposed offset alignment algorithms.

The thesis has done extensive research in the area of data distribution. The open problem of the efficient distribution type has been resolved to a great extent by this research. Segment distribution has been proposed to resolve the conflict between reducing data shift movement and increasing processor workload balance. Regarding to a particular dimension in the template array, segment distribution minimizes the impact of data shift movement by allocating elements consecutively to processors and balances the processor workload by varying the size of the segment assigned to different processors. The concept of the density function is introduced to estimate the computation load at compilation time. An optimal processor allocation algorithm has been proposed to minimize the overall cost of data shift communication across multiple dimensions of the template array. The segment distribution and optimal processor allocation proposed in this thesis provide the best data distribution support for most data parallel programs.

We have demonstrated the effectiveness of the proposed algorithms on the nCUBE-

2 commercial multiprocessors. The performance results have shown that the proposed algorithms are superior to the approaches presented by other research. We believe that our framework of the data decomposition analysis will serve to optimize data decomposition for increasingly popular data parallel programs with greater fidelity than exists in the current state of the art.

6.2 Directions of Future Work

The data decomposition framework presented in this thesis establishes a foundation for future study but needs to be extended in several ways.

The algorithmic results proposed in this research are considered for parallelizable loops. However, a real application program is a mixture of different types of subprogram structures including parallelizable loops, DOACROSS loops, and intrinsic loops. Some preliminary approaches have been proposed [71] to estimate the amount of communication for intrinsic loops with respect to different types of data alignment and data distribution. An integration of data decomposition information for different subprogram structures is highly demanded.

A good framework for data redistribution and data realignment is still an open issue in the research area of the data decomposition analysis. Data arrays are required to be re-aligned for different computing structures in different subprogram phases. Moreover, based on the data distribution analysis proposed in this research, the template may be re-distributed not for the requirement of data re-alignment but for the purpose of balancing processor workload instead. This condition brings additional complexity into the process of finding a good framework for data redistribution and data realignment.

Finally, the data decomposition framework presented in this research is proposed for compilation-time optimization. However, certain programming characteristics are

unknown until run-time, in particular, the information for processor workload balance [60]. In this case, the framework design for static data decomposition should be incorporated with the run-time support.

BIBLIOGRAPHY

BIBLIOGRAPHY

- [1] BBN Advanced Computers Inc., *Inside the TC2000 Computer*, 1990.
- [2] Cray Research, Inc., Mendota Heights, MN, *CRAY T3D System Architecture Overview*, Sept. 1993.
- [3] Convex Computer Corporation, Richardson, Texas, *CONVEX Exemplar Programming Guide*, Mar. 1994.
- [4] W. J. Dally and C. L. Seitz, "The torus routing chip," *Journal of Distributed Computing*, vol. 1, no. 3, pp. 187–196, 1986.
- [5] Intel Corporation, *A Touchstone DELTA System Description*, 1991.
- [6] NCUBE Company, *NCUBE 6400 Processor Manual*, 1990.
- [7] Meiko Limited, Waltham, MA., *Computing Surface: CS-2 Communications Networks*, 1993.
- [8] C. E. Leiserson *et al.*, "The network architecture of the Connection Machine CM-5," in *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, (San Diego, CA.), pp. 272–285, Association for Computing Machinery, 1992.
- [9] W. Gropp, E. Lusk, and S. Pieper, "Users Guide for the ANL IBM SP-1 DRAFT," Tech. Rep. ANL/MCS-TM-00, Argonne National Laboratory, Feb. 1994.
- [10] C. B. Stunkel *et al.*, "Architecture and implementation of vulcan," in *8th International Parallel Processing Symposium*, (Cancun Mexico), pp. 268–274, IEEE, 1994.
- [11] D. Patterson, "A case for now (networks-of-workstations)," in *Hot Interconnects II*, (Stanford), IEEE, Aug. 1994.

- [12] M. Blumrich and et al, "Two virtual memory mapped network interface designs," in *Hot Interconnects II*, (Stanford), IEEE, Aug. 1994.
- [13] 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.
- [14] High Performance Fortran Forum, "High Performance Fortran Language Specification (version 1.0)," May 1993.
- [15] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiler optimizations for Fortran D on MIMD distributed-memory machines," in *Proceedings of Supercomputing'91*, pp. 86–100, Nov. 1991.
- [16] 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.
- [17] J. M. Anderson and M. S. Lam, "Global optimizations for parallelism and locality on scalable parallel machines," in *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pp. 112–125, June 1993.
- [18] J. Ramanujam and P. Sadayappan, "Tiling multidimensional iteration spaces for nonshared memory machines," in *Proceedings of Supercomputing'91*, pp. 111–120, Nov. 1991.
- [19] J. R. Gilbert and R. Schreiber, "Optimal expression evaluation for data parallel architectures," *Journal of Parallel and Distributed Computing*, vol. 13, pp. 58–64, Sept. 1991.
- [20] H. Xu and L. M. Ni, "Optimizing data alignment in data parallel programs," in *Proceedings of the IEEE/CS 14th International Conference on Distributed Computing Systems*, pp. 336–344, June 1994.
- [21] H. Xu and L. M. Ni, "Optimizing data decomposition for data parallel programs," in *Proceedings of the 1994 International Conference on Parallel Processing*, vol. 2, pp. 225–232, Aug. 1994.
- [22] L. M. Ni and P. K. McKinley, "A survey of wormhole routing techniques in direct networks," *IEEE Computer*, vol. 26, pp. 62 – 76, Feb. 1993.

- [23] W. Pugh, "A practical algorithm for exact array dependence analysis," *Communications of the ACM*, pp. 102–114, Aug. 1992.
- [24] U. Banerjee, *Dependence Analysis for Supercomputing*. 101 Philip Drive, Assinippi Park, Norwell, Massachusetts 02061: Kluwer Academic, 1988.
- [25] M. W. Hall, K. Kennedy, and K. McKinley, "Interprocedural transformations for parallel code generation," Tech. Rep. CRPC-TR91149, Rice University, Center for Research on Parallel Computation, Apr. 1991.
- [26] J. Allen and K. Kennedy, "PFC: A program to convert Fortran to parallel form," Tech. Rep. MASC-TR82-6, Rice University, Mar. 1982.
- [27] R. Allen and K. Kennedy, "Automatic loop interchange," in *Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction*, pp. 233–246, June 1984.
- [28] V. Balasundaram and K. Kennedy, "A technique for summarizing data access and its use in parallelism enhancing transformations," in *Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction*, June 1989.
- [29] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 452–471, Oct. 1991.
- [30] M. Wolfe, "Loop skewing: The wavefront method revisited," *International Journal of Parallel Programming*, vol. 15, pp. 279–293, Aug. 1986.
- [31] U. Banerjee, "Unimodular transformations of double loops," *The Third Workshop on Programming Language and Compilers for Parallel Computing*, Aug. 1990.
- [32] M. Wolfe and C.-W. Tseng, "The power test for data dependence analysis," Tech. Rep. TR CS/E 90-015, Oregon Graduate Institute, Aug. 1990.
- [33] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, vol. 29, pp. 1184–1201, Dec. 1986.
- [34] D. J. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proceedings of the 8-th ACM Symposium on Principles of Programming Languages*, pp. 207–218, Jan. 1981.

- [35] H. Xu, E. T. Kalns, P. K. McKinley, and L. M. Ni, "ComPaSS: A Communication Package for Scalable Software Design," *Journal of Parallel and Distributed Computing*, vol. 22, pp. 449–461, Sept. 1994.
- [36] Message Passing Interface Forum, "Document for a standard Message-Passing Interface," Tech. Rep. CS-93-214, University of Tennessee, Nov. 1993.
- [37] A. Bar-Noy, J. Bruck, C.-T. Ho, S. Kipnis, and B. Schieber, "Computing global combine operations in the multi-port postal model," in *Proceedings of the fifth IEEE symposium on parallel and distributed processing*, pp. 336–343, Dec. 1993.
- [38] C.-T. Ho and M.-Y. Kao, "Optimal broadcast on hypercubes with wormhole and E-cube routings," in *Proceedings of the 1993 International Conference on Parallel and Distributed Systems*, pp. 694–697, 1992.
- [39] E. 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.
- [40] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "A static performance estimator to guide data partitioning decisions," in *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Apr. 1991.
- [41] J. H. Saltz and C.-W. Tseng, *Compilation and Runtime Support for Massively Parallel Processors*. Supercomputing'93 Tutorial, 1993.
- [42] Z. Shen, Z. Li, and P.-C. Yew, "An empirical study of Fortran programs for parallelizing compilers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 356–364, July 1990.
- [43] M. Wolfe, *Optimizing Supercompilers for Supercomputers*. Cambridge, Massachusetts: The MIT Press, 1989.
- [44] W. Shang and J. Fortes, "Tiling of iteration spaces for multicomputers," in *Proceedings of the 1990 International Conference on Parallel Processing*, vol. 2, pp. 179–186, Aug. 1990.
- [45] T. H. Tzen and L. M. Ni, "Dependence uniformization: A loop parallelization technique," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 547–558, May 1993.

- [46] L. M. Ni, H. Xu, and E. T. Kalns, "Issues in Scalable Library Design for Massively Parallel Computers," in *Proceedings of Supercomputing'93*, pp. 181–190, Nov. 1993.
- [47] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S. H. Teng, "Automatic array alignment in data-parallel programs," in *the Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pp. 16–28, Jan. 1993.
- [48] H. Xu, P. K. McKinley, and L. M. Ni, "Efficient implementation of barrier synchronization in wormhole-routed hypercube multicomputers," *Journal of Parallel and Distributed Computing*, vol. 16, pp. 172–184, 1992.
- [49] D. F. Robinson, D. Judd, P. K. McKinley, and B. H. C. Cheng, "Efficient collective data distribution in all-port wormhole-routed hypercubes," in *Proceedings of Supercomputing'93*, pp. 792–801, Nov. 1993.
- [50] H. Xu, Y.-D. Gui, and L. M. Ni, "Optimal software mulitcast in wormhole-routed multistage networks," in *Proceedings of Supercomputing'94*, Nov. 1994.
- [51] H. Xu, P. K. McKinley, and L. M. Ni, "A Scalable Multicast Service in 2d Mesh Networks," in *Frontiers'92: The 4th Symposium on the Frontiers of Massively Parallel Computation*, pp. 156–163, Oct. 1992.
- [52] Z. Fang, P.-C. Yew, , P. Tang, and C. Q. Zhu, "Dynamic processor self-scheduling for general parallel nested loops," *IEEE Transactions on Computers*, vol. 39, pp. 919–929, July 1990.
- [53] C. Polychronopoulos and D. Kuck, "Guided self-scheduling: A practical self-scheduling scheme for parallel supercomputers," *IEEE Transactions on Computers*, vol. C-36, pp. 1425–1439, Dec. 1987.
- [54] P. Tang and P.-C. Yew, "Processor self-scheduling for multiple-nested parallel loops," in *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 528–535, Aug. 1986.
- [55] M. Weiss, Z. Fang, C. R. Morgan, and P. Belmont, "Effective dynamic scheduling and memory management on parallel processing systems," in *Proceedings of the 1989 COMPSAC*, pp. 122–129, Sept. 1989.
- [56] T. H. Tzen and L. M. Ni, "Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 87–98, Jan. 1993.

- [57] H.-M. Su and P.-C. Yew, "Efficient DOACROSS execution on distributed shared-memory multiprocessors," in *Proceedings of Supercomputing'91*, pp. 842–853, Nov. 1991.
- [58] J.-H. Chow and W. L. Harrison III, "Switch-stacks: A scheme for microtasking nested parallel loops," in *Proceedings of Supercomputing'90*, pp. 190–199, Nov. 1990.
- [59] L. M. Ni and C.-F. E. Wu, "Design tradeoffs for processor scheduling in shared-memory multiprocessor systems," *IEEE Transactions on Software Engineering*, vol. 15, pp. 327–334, Mar. 1989. also in Proc. of the 1985 Int. Conf. on Parallel Processing, pp. 63–70.
- [60] J. Liu and V. A. Salefore, "Self-scheduling on distributed-memory machines," in *Proceedings of Supercomputing'93*, Nov. 1993.
- [61] P. Lee and T. Tsai, "Compiling efficient programs for tightly-coupled distributed memory computers," in *Proceedings of the 1993 International Conference on Parallel Processing*, vol. II, pp. 161–165, Aug. 1993.
- [62] 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.
- [63] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.
- [64] C. Gong, R. Gupta, and R. Melhem, "Compilation techniques for optimizing communication on distributed-memory systems," in *Proceedings of the 1993 International Conference on Parallel Processing*, vol. II, pp. 39–46, Aug. 1993.
- [65] E. Duesterwald, R. Gupta, and M. L. Soffa, "A practical data flow framework for array reference analysis and its use in optimizations," in *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pp. 68–77, June 1993.
- [66] 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.
- [67] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald, *Vienna Fortran: A Language Specification (Version 1.1)*, 1991.

- [68] K. Knob, J. D. Lukas, and G. L. Steele, "Data optimization: allocation of arrays to reduce communication on SIMD Machines," *Journal of Parallel and Distributed Computing*, vol. 2, pp. 102–118, Feb. 1990.
- [69] B. Cukic and F. B. Bastani, "Automatic array alignment as a step in hierarchical program transformation," in *8th International Parallel Processing Symposium*, (Cancun Mexico), pp. 578–582, IEEE, 1994.
- [70] 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.
- [71] J. Li and M. Chen, "Compiling communication-efficient programs for massively parallel machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 361–376, July 1991.
- [72] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S. H. Teng, "Optimal evaluation for array expressions on massively parallel machines," in *the Second Workshop on Languages, Compilers, and Runtime Environments for Distributed Memory Multiprocessors*, Oct. 1992.
- [73] S. Chatterjee, J. R. Gilbert, and R. Schreiber, "The alignment-distribution graph," in *the Sixth Annual Workshop on Languages and Compilers for Parallelism*, Aug. 1993.
- [74] J. R. Gilbert, S. Chatterjee, and R. Schreiber, "Mobil and replicated alignment of arrays in data-parallel programs," in *Proceedings of Supercomputing'93*, Nov. 1993.
- [75] J. Ramanujam and P. Sadayappan, "Compile-time techniques for data distribution in distributed memory machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 472–482, Oct. 1991.
- [76] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and A. W. Lim, "An overview of a compiler for scalable parallel machines," in *the Sixth Annual Workshop on Languages and Compilers for Parallelism*, Aug. 1993.
- [77] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 30–44, June 1991.
- [78] R. A. Usmani, *Applied Linear Algebra*. Marcel Dekker INC., 1987.

- [79] A. V. Aho, J. E. Hopcraft, and J. D. Ullman, *Data Structures and Algorithms*. Addison-Wesley, 1983.
- [80] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, Jan. 1991.
- [81] P. Havlak and K. Kennedy, "An implementation of interprocedural bounded regular section analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, July 1991.

MICHIGAN STATE UNIV. LIBRARIES



31293010201519