

PERFORMANCE CHARACTERIZATION FOR GENERATING OPTIMUM PARALLEL
CODES IN HETEROGENEOUS ENVIRONMENTS

by

Duygu BİRECİKLİ

Submitted to the Institute of Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science
in
Computer Engineering

Yeditepe University

2007

PERFORMANCE CHARACTERIZATION FOR GENERATING OPTIMUM PARALLEL
CODES IN HETEROGENEOUS ENVIRONMENTS

APPROVED BY:

Assist. Prof. Dr. Esin Onbaşıoğlu
(Thesis Supervisor)

Assist. Prof. Dr. Ender Özcan

Prof. Dr. Melek Başak

DATE OF APPROVAL:

ACKNOWLEDGEMENTS

I would like to express my thanks to the valuable people whose support allowed me to carry this project to completion.

I am especially grateful to my supervisor Assist. Prof. Dr. Esin Onbaşıoğlu, who offered me a challenging research area. I would especially like to thank her for guiding me throughout this thesis, for her helpful comments, suggestions, encouragement, support and always welcoming me with a warm smile.

I would also like to thank to Assist. Prof. Dr. Ender Özcan, for sharing GALib Tool and his experiences on Genetic Algorithms.

I especially thank to my family for their constant support throughout my education. I owe special thanks to my parents for providing me great opportunities to improve myself, for their love, patience and understanding. I also thank to my sister for being cheerful all the time and for her patience.

My deepest appreciation goes to my friend Rıfat Benveniste, for his encouragement, support and understanding. I thank him for helping me with his suggestions, whenever I had a problem. I am grateful for his patience during my long hours in the laboratories. It is very nice to have such a friend!

ABSTRACT

PERFORMANCE CHARACTERIZATION FOR GENERATING OPTIMUM PARALLEL CODES IN HETEROGENEOUS ENVIRONMENTS

A sequential program can be parallelized such that each level executes on a different number of processors. During the execution of the parallel program, the processors may need to transfer data before the next computation level begins. In a parallel program, an array may be used in more than one level, with different decomposition types. Various conversion functions are proposed, here, to transfer data between all processor pairs. These functions are used between levels to convert the decomposition of arrays from one type to another. They are also used as benchmarks to obtain the performance characteristics of the heterogeneous environment. These characteristics are used to calculate the communication time of a parallel program. A tool is proposed for generating optimum parallel codes. It consists of User Program Parser, Conversion Functions Library, Parallel Program Optimizer, Array Information Table Generator and Parallelizer. The User Program Parser parses the sequential program and generates the possible decompositions for each level, according to the alignment of the arrays. Using Memetic Algorithms, the Parallel Program Optimizer finds an optimum solution, which maps each level to a data decomposition, a processor group and a number of processors in that group. The Array Information Table Generator stores the decomposition of arrays at the previous levels for calculating the communication time and preparing the conversion function calls. The Parallelizer is a text processor. It inserts conversion function calls in the program and generates the parallel program which is ready to compile and execute on the heterogeneous environment with an optimum execution scheme.

ÖZET

HETEROJEN ORTAMLARDA EN İYİ PARALEL KODLARI OLUŞTURMAK İÇİN PERFORMANS KARAKTERİSTİKLERİNİN BULUNMASI

Bir ardışık program, her seviyesi farklı sayıda işlemci tarafından çalıştırılacak şekilde paralelleştirilebilir. Paralel programın çalışması esnasında, işlemciler bir sonraki hesaplama seviyesi başlamadan önce veri aktarımına ihtiyaç duyabilirler. Bir paralel programda, bir dizilim farklı ayrışımalarla birden çok seviyede kullanılmış olabilir. Burada, bütün işlemci çiftleri arasında veri aktarmak için, çeşitli dönüşüm fonksiyonları önerilmiştir. Bu fonksiyonlar seviyeler arasında kullanılır ve dizilimlerin ayrışımını bir türden diğerine çevirirler. Ayrıca bunlar heterojen ortamın performans karakteristiklerinin elde edilmesinde denektaş testi olarak kullanılırlar. Bu karakteristikler bir paralel programın iletişim süresinin hesaplanmasında kullanılırlar. En iyi paralel kodları oluşturmak için, Kullanıcı Programı Ayrıştırıcı, Dönüşüm Fonksiyonları Program Kitaplığı, Paralel Program Eniyileştirici, Dizilim Bilgisi Tablosu Oluşturucu ve Paralelleştirici içeren bir araç önerilmiştir. Kullanıcı Programı Ayrıştırıcı ardışık programı ayrıştırır ve dizilimlerin uyuşmasına göre, her bir seviye için mümkün olan ayrışimleri üretir. Paralel Program Eniyileştirici, Memetik Algoritmaları kullanarak, her seviyeyi bir veri ayrışımı, bir işlemci grubu ve o gruptan belli sayıda işlemci ile eşleyen en iyi çözümü bulur. Dizilim Bilgisi Tablosu Oluşturucu, iletişim süresini hesaplamak ve dönüşüm fonksiyonu çağrılarını hazırlamak için, dizilimlerin önceki seviyelerdeki ayrışımını saklar. Paralelleştirici bir metin işlemcidir. Programa dönüşüm fonksiyonu çağrılarını ekler ve en iyi çalışma tasarısı ile heterojen ortam üzerinde derlenip çalıştırılmaya hazır olan paralel kodu oluşturur.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iii
ABSTRACT	iv
ÖZET	v
LIST OF FIGURES	x
LIST OF TABLES	xvii
LIST OF SYMBOLS/ABBREVIATIONS	xix
1. INTRODUCTION	1
1.1. Data Decomposition Types	2
1.2. Overview of the Thesis	4
2. PREVIOUS WORK	7
2.1. Previous Work About Processor Allocation in Heterogeneous Environments	7
2.2. Previous Work About Parallelizers and Parallel Program Optimization	12
3. USER PROGRAM PARSER	16
4. CONVERSION FUNCTIONS	21
4.1. The Butterfly Algorithm	22
4.1.1. The Original Butterfly Algorithm	22
4.1.2. The Modified Butterfly Algorithm	23
4.1.3. The Reason for Using Modified Butterfly Algorithm	26
4.2. Initialization, Conversion and Gather Functions	29
4.3. Matrix Partitioning	30
4.4. Conversion Functions for Matrices	33
4.4.1. Horizontal to Horizontal Conversion for Matrices (mH2H)	33
4.4.2. Horizontal to Vertical Conversion for Matrices (mH2V)	34
4.4.3. Horizontal to Rectangular Conversion for Matrices (mH2R)	35
4.4.4. Horizontal to Horizontal Block Conversion for Matrices (mH2HB)	35
4.4.5. Horizontal to Vertical Block Conversion for Matrices (mH2VB)	36
4.4.6. Horizontal to Replicated Conversion for Matrices (mH2REP)	37
4.4.7. Vertical to Horizontal Conversion for Matrices (mV2H)	38
4.4.8. Vertical to Vertical Conversion for Matrices (mV2V)	39

4.4.9. Vertical to Rectangular Conversion for Matrices (mV2R)	39
4.4.10. Vertical to Horizontal Block Conversion for Matrices (mV2HB) . .	40
4.4.11. Vertical to Vertical Block Conversion for Matrices (mV2VB)	41
4.4.12. Vertical to Replicated Conversion for Matrices (mV2REP)	42
4.4.13. Rectangular to Horizontal Conversion for Matrices (mR2H)	43
4.4.14. Rectangular to Vertical Conversion for Matrices (mR2V)	44
4.4.15. Rectangular to Rectangular Conversion for Matrices (mR2R)	45
4.4.16. Rectangular to Horizontal Block Conversion for Matrices (mR2HB)	45
4.4.17. Rectangular to Vertical Block Conversion for Matrices (mR2VB) .	46
4.4.18. Rectangular to Replicated Conversion for Matrices (mR2REP) . . .	47
4.4.19. Horizontal Block to Horizontal Conversion for Matrices (mHB2H) .	48
4.4.20. Horizontal Block to Vertical Conversion for Matrices (mHB2V) . .	49
4.4.21. Horizontal Block to Rectangular Conversion for Matrices (mHB2R)	50
4.4.22. Horizontal Block to Horizontal Block Conversion for Matrices (mHB2HB)	50
4.4.23. Horizontal Block to Vertical Block Conversion for Matrices (mHB2VB)	51
4.4.24. Horizontal Block to Replicated Conversion for Matrices (mHB2REP)	52
4.4.25. Vertical Block to Horizontal Conversion for Matrices (mVB2H) . .	53
4.4.26. Vertical Block to Vertical Conversion for Matrices (mVB2V)	54
4.4.27. Vertical Block to Rectangular Conversion for Matrices (mVB2R) .	55
4.4.28. Vertical Block to Horizontal Block Conversion for Matrices (mVB2HB)	56
4.4.29. Vertical Block to Vertical Block Conversion for Matrices (mVB2VB)	57
4.4.30. Vertical Block to Replicated Conversion for Matrices (mVB2REP) .	58
4.4.31. Replicated to Horizontal Conversion for Matrices (mREP2H) . . .	58
4.4.32. Replicated to Vertical Conversion for Matrices (mREP2V)	59
4.4.33. Replicated to Rectangular Conversion for Matrices (mREP2R) . . .	60
4.4.34. Replicated to Horizontal Block Conversion for Matrices (mREP2HB)	61
4.4.35. Replicated to Vertical Block Conversion for Matrices (mREP2VB) .	62
4.4.36. Replicated to Replicated Conversion for Matrices (mREP2REP) . .	63
4.5. Vector Partitioning	64
4.6. Conversion Functions for Vectors	66
5. PARALLEL PROGRAM OPTIMIZER	67
6. ARRAY INFORMATION TABLE GENERATOR	69

6.1. Array Information Tables	69
6.1.1. The R/L position	70
6.2. Initializing the Tables	72
6.3. Preparing the Tables for the New Individual	72
6.4. Updating the Tables	73
6.5. Reading the Tables to Calculate the Communication Time	73
6.6. Calculation of the Communication Time	74
6.7. Reading the Tables to Write the Conversion Function Call Lines for the Best Chromosome	74
7. PARALLELIZER	77
7.1. The Parallelization Process	77
7.2. The "include" keyword	79
7.3. The "main" keyword	79
7.4. The "Level" keyword	80
7.5. The "for" keyword	84
7.6. The "return" keyword	86
7.7. Copying the User's Statements	86
7.8. The "printf" keyword	87
8. EXPERIMENTS	88
8.1. Measurement Experiments	88
8.1.1. Heterogeneous Test Environment	89
8.1.2. Performance Measurement of the Environment	92
8.1.3. Time Prediction of Parallel Programs	94
8.2. Optimization Experiments	97
8.3. Verification of the Results	105
9. CONCLUSION	108
APPENDIX A:	112
A.1. Horizontal Initialization Function for Matrices	112
A.2. Horizontal to Horizontal Conversion Function for Matrices	113
A.3. Horizontal to Vertical Conversion Function for Matrices	115
A.4. Horizontal Gather Function for Matrices	119
APPENDIX B:	121

B.1. The Slopes and Intercepts	121
REFERENCES	172



LIST OF FIGURES

Figure 1.1.	The heterogeneous network	1
Figure 1.2.	Partitioning data on 4 processors is shown for all Data Decomposition Types for a matrix.	2
Figure 1.3.	Partitioning data on 4 processors is shown for all Data Decomposition Types for a vector.	3
Figure 1.4.	The overview of the thesis	4
Figure 3.1.	An example for a computation level	16
Figure 4.1.	The original Butterfly Algorithm	22
Figure 4.2.	The Modified Butterfly Algorithm	24
Figure 4.3.	The Modified Butterfly Algorithm, where $p = 4$ and $q = 8$	26
Figure 4.4.	The Modified Butterfly Algorithm, where $p = 8$ and $q = 4$	27
Figure 4.5.	Horizontal (H) decomposition of matrices for 4, 8 and 16 processors .	31
Figure 4.6.	Vertical (V) decomposition of matrices for 4, 8 and 16 processors . . .	31
Figure 4.7.	Rectangular (R) decomposition of matrices for 4, 8 and 16 processors	31
Figure 4.8.	Horizontal Block (HB) decomposition of matrices for 4, 8 and 16 processors	32

Figure 4.9.	Vertical Block (VB) decomposition of matrices for 4, 8 and 16 processors	32
Figure 4.10.	Replicated (REP) decomposition of matrices for 4, 8 and 16 processors	32
Figure 4.11.	Conversion from 4H to 8H	33
Figure 4.12.	Conversion from 8H to 4H	33
Figure 4.13.	Conversion from 4H to 8V	34
Figure 4.14.	Conversion from 8H to 4V	34
Figure 4.15.	Conversion from 4H to 8R	35
Figure 4.16.	Conversion from 8H to 4R	35
Figure 4.17.	Conversion from 4H to 8HB	36
Figure 4.18.	Conversion from 8H to 4HB	36
Figure 4.19.	Conversion from 4H to 8VB	36
Figure 4.20.	Conversion from 8H to 4VB	37
Figure 4.21.	Conversion from 4H to 8REP	37
Figure 4.22.	Conversion from 8H to 4REP	38
Figure 4.23.	Conversion from 4V to 8H	38
Figure 4.24.	Conversion from 8V to 4H	38

Figure 4.25.	Conversion from 4V to 8V	39
Figure 4.26.	Conversion from 8V to 4V	39
Figure 4.27.	Conversion from 4V to 8R	40
Figure 4.28.	Conversion from 8V to 4R	40
Figure 4.29.	Conversion from 4V to 8HB	40
Figure 4.30.	Conversion from 8V to 4HB	41
Figure 4.31.	Conversion from 4V to 8VB	41
Figure 4.32.	Conversion from 8V to 4VB	42
Figure 4.33.	Conversion from 4V to 8REP	42
Figure 4.34.	Conversion from 8V to 4REP	43
Figure 4.35.	Conversion from 4R to 8H	43
Figure 4.36.	Conversion from 8R to 4H	43
Figure 4.37.	Conversion from 4R to 8V	44
Figure 4.38.	Conversion from 8R to 4V	44
Figure 4.39.	Conversion from 4R to 8R	45
Figure 4.40.	Conversion from 8R to 4R	45

Figure 4.41.	Conversion from 4R to 8HB	46
Figure 4.42.	Conversion from 8R to 4HB	46
Figure 4.43.	Conversion from 4R to 8VB	46
Figure 4.44.	Conversion from 8R to 4VB	47
Figure 4.45.	Conversion from 4R to 8REP	47
Figure 4.46.	Conversion from 8R to 4REP	48
Figure 4.47.	Conversion from 4HB to 8H	48
Figure 4.48.	Conversion from 8HB to 4H	48
Figure 4.49.	Conversion from 4HB to 8V	49
Figure 4.50.	Conversion from 8HB to 4V	49
Figure 4.51.	Conversion from 4HB to 8R	50
Figure 4.52.	Conversion from 8HB to 4R	50
Figure 4.53.	Conversion from 4HB to 8HB	51
Figure 4.54.	Conversion from 8HB to 4HB	51
Figure 4.55.	Conversion from 4HB to 8VB	51
Figure 4.56.	Conversion from 8HB to 4VB	52

Figure 4.57.	Conversion from 4HB to 8REP	52
Figure 4.58.	Conversion from 8HB to 4REP	53
Figure 4.59.	Conversion from 4VB to 8H	53
Figure 4.60.	Conversion from 8VB to 4H	54
Figure 4.61.	Conversion from 4VB to 8V	54
Figure 4.62.	Conversion from 8VB to 4V	55
Figure 4.63.	Conversion from 4VB to 8R	55
Figure 4.64.	Conversion from 8VB to 4R	55
Figure 4.65.	Conversion from 4VB to 8HB	56
Figure 4.66.	Conversion from 8VB to 4HB	56
Figure 4.67.	Conversion from 4VB to 8VB	57
Figure 4.68.	Conversion from 8VB to 4VB	57
Figure 4.69.	Conversion from 4VB to 8REP	58
Figure 4.70.	Conversion from 8VB to 4REP	58
Figure 4.71.	Conversion from 4REP to 8H	59
Figure 4.72.	Conversion from 8REP to 4H	59

Figure 4.73.	Conversion from 4REP to 8V	60
Figure 4.74.	Conversion from 8REP to 4V	60
Figure 4.75.	Conversion from 4REP to 8R	60
Figure 4.76.	Conversion from 8REP to 4R	61
Figure 4.77.	Conversion from 4REP to 8HB	61
Figure 4.78.	Conversion from 8REP to 4HB	62
Figure 4.79.	Conversion from 4REP to 8VB	62
Figure 4.80.	Conversion from 8REP to 4VB	63
Figure 4.81.	Conversion from 4REP to 8REP	63
Figure 4.82.	Conversion from 8REP to 4REP	64
Figure 4.83.	Horizontal (H) decomposition of vectors for 4, 8 and 16 processors . .	64
Figure 4.84.	Horizontal Block (HB) decomposition of vectors for 4, 8 and 16 pro- cessors	65
Figure 4.85.	Replicated (REP) decomposition of vectors for 4, 8 and 16 processors	65
Figure 5.1.	Pseudocode of a Memetic Algorithm	68
Figure 7.1.	The modification of the <i>for loop limit</i> according to the decomposition type	83

Figure 7.2.	The <i>for loop limit</i> is a string.	85
Figure 7.3.	The <i>for loop limit</i> is an integer.	85
Figure 8.1.	The benchmark which measures the computation time	89
Figure 8.2.	The pingpong benchmark	91
Figure 8.3.	The measurement of matrix conversion duration	93
Figure 8.4.	The measurement of vector conversion duration	93
Figure 8.5.	Computation level example	95
Figure 8.6.	Decomposed computation level example	95
Figure 8.7.	Hessenberg Reduction	98
Figure 8.8.	Dongarra's benchmark	99
Figure 8.9.	The possible decompositions for Hessenberg Reduction	100
Figure 8.10.	The possible decompositions for Dongarra's benchmark	101

LIST OF TABLES

Table 4.1.	The original Butterfly Algorithm among eight processors	23
Table 4.2.	The Modified Butterfly Algorithm, where $p = 8$ and $q = 8$	24
Table 4.3.	The Modified Butterfly Algorithm, where $p = 4$ and $q = 8$	27
Table 4.4.	The Modified Butterfly Algorithm, where $p = 8$ and $q = 4$	28
Table 8.1.	The average computation time of each processor	90
Table 8.2.	The average communication time (in msec) between the processors of the groups	91
Table 8.3.	The best chromosome for Hessenberg Reduction, where the groups have 16, 4 and 4 processors, $N = 2048$. The fitness is 250438.946887 msec.	102
Table 8.4.	The best chromosome for Dongarra's benchmark, where the groups have 16, 4 and 4 processors, $N = 2048$. The fitness is 30191.427704 msec.	103
Table 8.5.	The best chromosome for Hessenberg Reduction, where the groups have 4, 4 and 4 processors, $N = 2048$. The fitness is 734366.768356 msec.	105
Table 8.6.	The best chromosome for Dongarra's benchmark, where the groups have 4, 4 and 4 processors, $N = 2048$. The fitness is 45893.620401 msec.	106

B.1	The slopes and intercepts of total array size (in bytes) for all possibilities (all previous and next groups, p and q values, conversion functions) . .	121
-----	--	-----



LIST OF SYMBOLS/ABBREVIATIONS

H	Horizontal Data Decomposition Type
HB	Horizontal Block Data Decomposition Type
h_q	The number of horizontal pieces, <i>i.e.</i> rows. (Used in partitioning the array)
ID	Identification number
LAN	Local Area Network
LHS	Left-hand side (of equal sign)
m	Matrix (Conversion Function for matrices)
MPI	Message Passing Interface
msec	millisecond
NONE	No Data Decomposition Type is used.
p	The previous <i>number of processors</i>
q	The next <i>number of processors</i>
R	Rectangular Data Decomposition Type
REP	Replicated Data Decomposition Type
RHS	Right-hand side (of equal sign)
v	Vector (Conversion Function for vectors)
V	Vertical Data Decomposition Type
VB	Vertical Block Data Decomposition Type
v_q	The number of vertical pieces, <i>i.e.</i> columns. (Used in partitioning the array)

1. INTRODUCTION

In this thesis, a tool is implemented, which makes "parallelizing a sequential program" an easy task. The aim of this project is to parallelize a sequential program automatically. The parallelized program can be compiled and run on a heterogeneous environment with an optimum execution scheme.

The heterogeneous environment in this thesis is defined as follows: There are some processor groups. The number of processors in each group may be different. The processors of the same group are identical, i.e. the groups are homogeneous. However, the processors of different groups may have different properties such as memory, processor speed, operating system, etc. All groups are connected to a Local Area Network (LAN). The communication speed among the processors of the same group are equal. The communication among different groups may have different speeds.

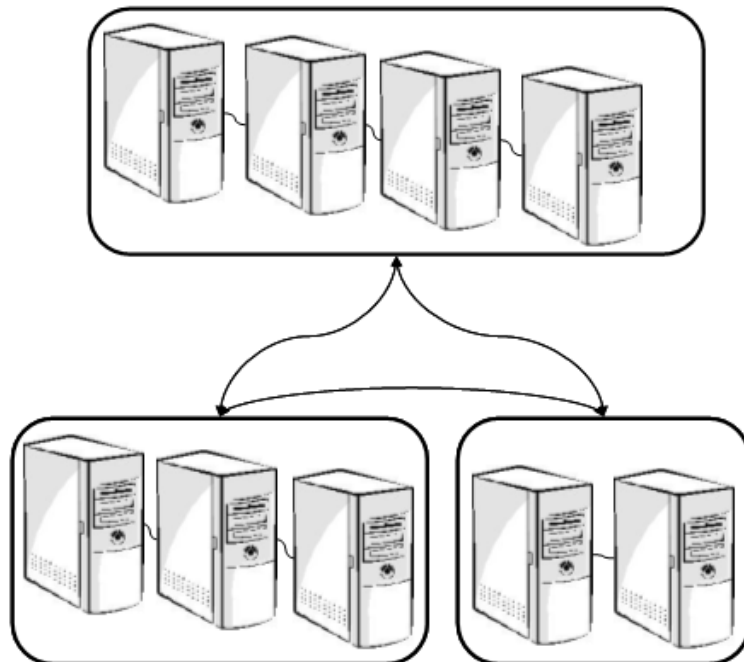


Figure 1.1. The heterogeneous network

1.1. Data Decomposition Types

It is assumed that, a user wants to run a program, which includes some levels. In each level, some calculations are done on matrices and vectors with large sizes. To run such a program on one processor takes too much time. However, if the program was executed on many processors, the duration would absolutely be shorter.

The method to parallelize a large-sized array is to decompose (partition) it into smaller pieces and send each piece to a processor. An array can be decomposed in many ways. The data decomposition types for a matrix are Horizontal, Vertical, Rectangular, Horizontal Block, Vertical Block and Replicated (Figure 1.2). For a vector, Horizontal, Horizontal Block and Replicated decomposition types can be used (Figure 1.3).

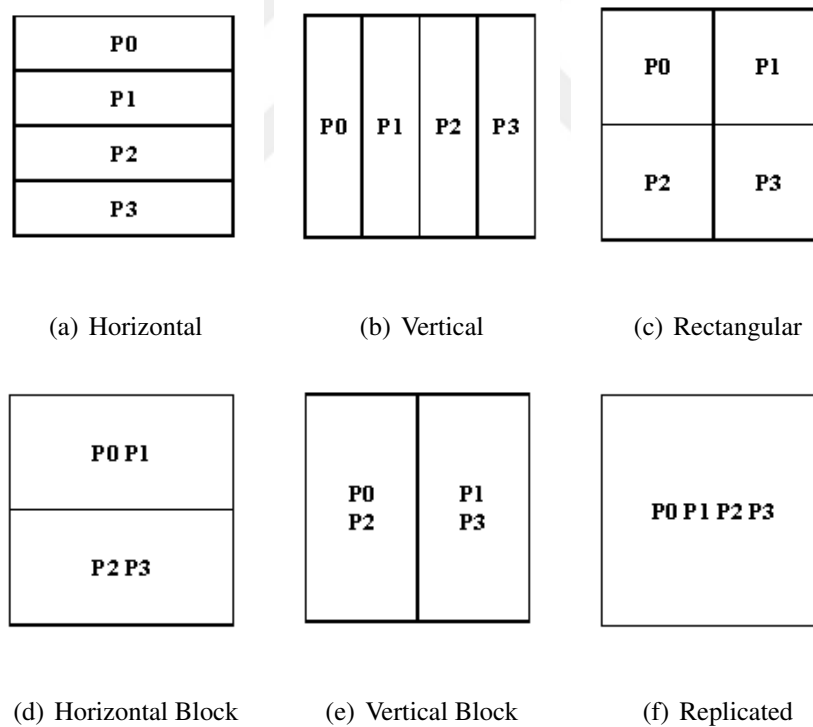


Figure 1.2. Partitioning data on 4 processors is shown for all Data Decomposition Types for a matrix.

As mentioned before, the user's sequential program is composed of some levels. There are 3 types of levels:

1. *Initialization level (Level 0)*: The first level of the user's sequential program, where the

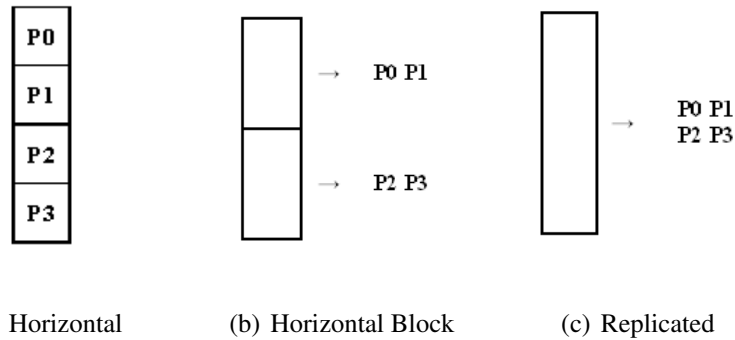


Figure 1.3. Partitioning data on 4 processors is shown for all Data Decomposition Types for a vector.

arrays take the initial values.

2. *Computation level (Level 1 to <number of levels>):* The levels where some calculations on the arrays are done.
3. *Gather-fprintf level (Level 999):* The user may want to gather and print the new elements of some LHS arrays to observe the results of the calculations. Before each gather-fprintf level, an array is gathered, and inside the level, the elements are printed into a file.

In each level, some matrices and/or vectors are used in calculations. According to the execution scheme determined by Parallel Program Optimizer, each level of the user's parallelized program may run on a different processor group, on different number of processors. An array may have different decomposition types at different levels.

In this project, the Conversion Functions are introduced to transfer the data of an array from one decomposition type to another. These Conversion Functions are used in the user's parallel program between the levels, to transfer arrays from the processors used in the previous level to the processors used in the next level. The total number of Conversion Functions is 65. For matrices, 49 functions are written (6 initialization/scatter, 37 conversion, 6 gather functions). For vectors, 16 functions are written (3 initialization/scatter, 10 conversion, 3 gather functions).

In the Conversion Functions and in the parallelized program, the Message Passing

Interface (MPI) is used. MPI is a specification for a set of functions which manage the movement of data among sets of communicating processes. These functions are defined for point-to-point communication between two processes, for collective operations among processes, for parallel I/O, and for process management [1]. Dongarra and Lastovetsky compared PVM (Parallel Virtual Machine) and MPI. They claimed that MPI provided high-performance communication [1].

1.2. Overview of the Thesis

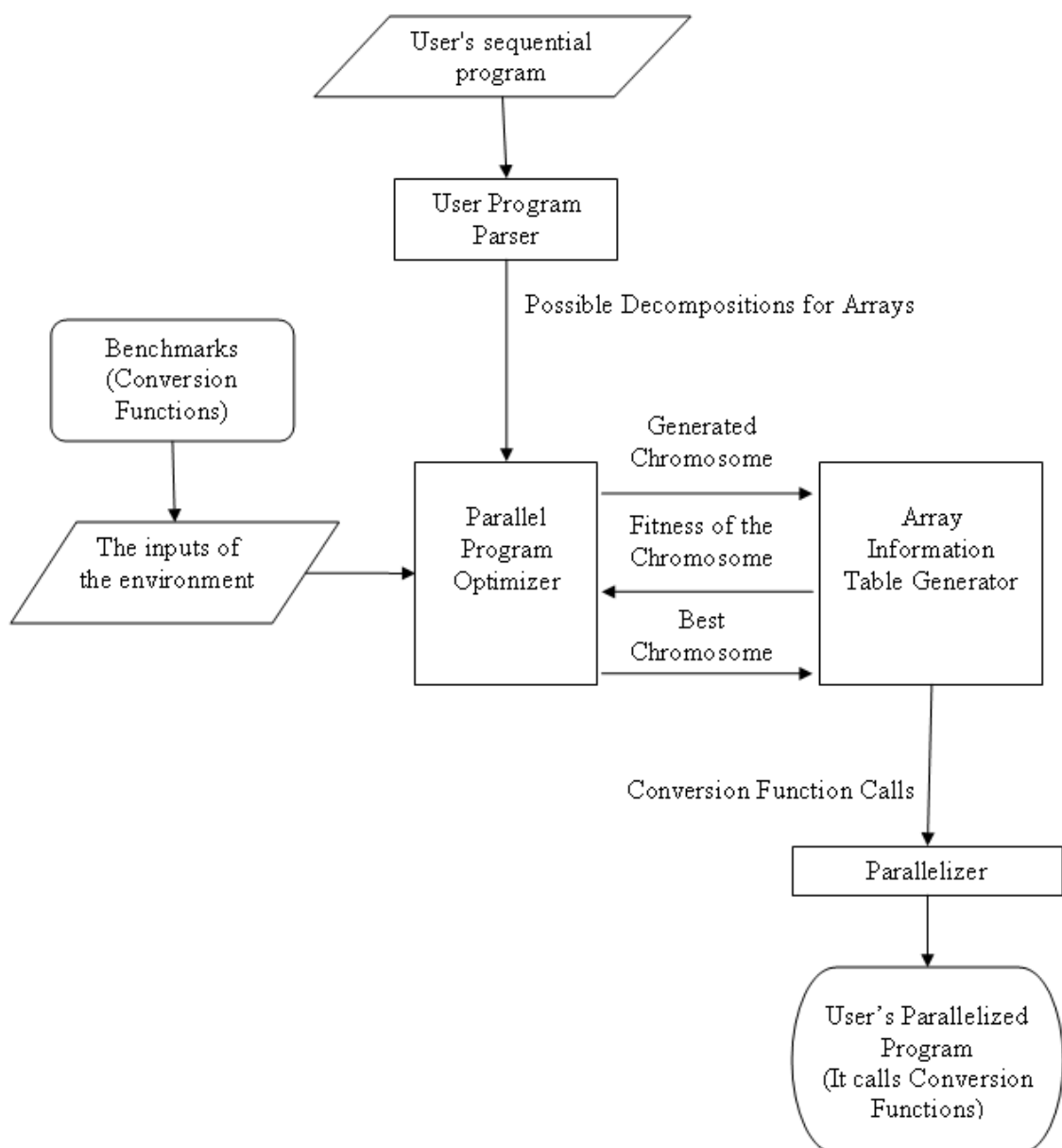


Figure 1.4. The overview of the thesis

In Figure 1.4, an overview of the thesis is given. Inputs to the system are the user's sequential program and the inputs of the environment such as the number of processor groups to be used, the number of processors in each group and the results of the heterogeneous environment parameters. These results are determined by executing some benchmarks with the Conversion Functions. The duration of each Conversion Function is measured for different cases. For different sizes of arrays, from p number of processors to q number of processors the functions are executed in each group and between the groups. Using the average durations, the slopes and intercepts of the total array size (in bytes) are calculated. Then, the results are used in communication time calculations as constant values. In the user's program, the size of the arrays and some *for loop limits* are decreased. The program is executed sequentially on one processor from each processor group. The duration of each computation level is measured. The average durations are used in computation time calculations.

The User Program Parser parses the user's sequential program. It gathers information about the levels and arrays, such as how many levels there are, how many *for loops* there are in each level, the statements of each level, the name, dimension, size, alignment of each array, etc. The parser also generates possible decompositions for each level, according to the alignment of the arrays at that level.

The Parallel Program Optimizer uses Memetic Algorithm to find an optimum solution, which minimizes the total execution time for the parallel program to be generated. The Memetic Algorithm generates some chromosomes, where each gene stores the necessary data for a level. The generated chromosomes carry the following data for each level in their genes: the ID of the processor group where the level will execute on, the number of processors in the chosen group and the selected decomposition's ID. One-point crossover, traditional mutation and some hill climbing methods are applied on the chromosomes.

Each chromosome is sent to Array Information Table Generator. The Array Information Table Generator writes each array's level, decomposition type, processor group ID, number of processors, its position (according to the equal sign) to a table. By analyzing the table and using the results of the heterogeneous environment parameters that were determined by benchmarks (average durations of computation levels, slopes and intercepts), it calculates

the fitness (total execution time, which is the sum of communication and computation times of all levels) of the chromosome and returns the fitness to Parallel Program Optimizer.

When the Parallel Program Optimizer finds the best chromosome (the one with minimum fitness), it sends the chromosome to Array Information Table Generator. Using the data in the best chromosome, the generator writes the necessary data to the table. This time, by analyzing the table, the generator writes the necessary lines to call the Conversion Functions (from parallelized program) into a file.

Parallelizer reads the user's sequential program line by line. In each line, it searches for some keywords and calls the related function according to the keyword. While processing and copying the lines, the functions insert MPI function calls and Conversion Function calls. At the end, the sequential program is parallelized. The parallelized program can be compiled and run on the heterogeneous environment.

2. PREVIOUS WORK

This thesis consists of two parts: Performance measurement and characterization of heterogeneous computing environments, optimum processor allocation using Genetic Algorithm and generation of the parallel program. The optimum processor allocation using Genetic Algorithm includes the mapping of levels (tasks) to a group of processors. Therefore, the literature survey covered processor allocation in heterogeneous environments. Moreover, the studies on parallelizers are investigated. The survey on the previous works is presented under two sections.

2.1. Previous Work About Processor Allocation in Heterogeneous Environments

Wang *et al.* [2] worked on the matching and concurrent scheduling problem in heterogeneous computing systems. They considered that an application task can be decomposed into computationally homogeneous subtasks. The subtasks might have different machine architectural requirements and might have data dependencies among them. They made two decisions: how to assign subtasks to machines (defined as *matching*) and how to order subtask execution for each machine and intermachine data transfers (defined as *scheduling*). Their goal was to minimize the overall execution time of the application task in the machine suite. They proposed a genetic-algorithm-based approach. They compared it with LMT (leveled min-time), baseline nonevolutionary heuristics and random search. They concluded that their GA-based approach found optimal solutions for small-scale scenarios. For larger scenarios, it outperformed the others.

Plata *et al.* [3] studied on a static processor allocation scheme for arbitrarily nested loops on distributed memory hypercubes. They proposed a loop partitioning scheme which produces a local code by reducing the value of the upper bound of all or some of the loops. This scheme preserves the loop structure. They used HYPAL algorithm in order to obtain optimal assignments of dimensions in the sense that minimizes execution time of the loop. They used matrix algebra and signal processing programs to test their loop partitioning scheme. They claimed that the scheme simplified the selection of a suitable data distribution and the

generation of the communication instructions.

Hamidzadeh *et al.* [4] mentioned homogeneous and heterogeneous systems, static and dynamic scheduling algorithms. They proposed a general model for describing and evaluating heterogeneous systems. As a measure of the heterogeneity in the system, they considered the degree of uniformity in the processing elements and the communication channels. They expressed that using the estimates for the execution times of the tasks, static scheduling can theoretically precompute an ideal schedule which produces the shortest possible execution time. However, using imprecise estimates might cause large computational load imbalances at run-time and produce significantly longer execution times. They proposed a class of dynamic scheduling algorithms for a heterogeneous computing system. Their optimization technique was based on the potentially non-uniform computation and communication costs on the processors. They claimed that their algorithms could easily adapt to different task granularities, to dynamically varying processor and system loads, and to systems with varying degrees of heterogeneity. They concluded performing a sophisticated scheduling technique on a dedicated processor could produce improvements in total execution times. For a completely homogeneous system, Hamidzadeh *et al.* [4] indicated that the computation and communication costs are uniform across the system. In other words, these costs are independent of any specific task-to-processor assignments provided by the scheduling operation. The cost of executing a single task is the sum of communication cost and computation cost incurred in the system and this sum is the same no matter on what processor that task is executed. When a parallel task is assigned to any of the processors, the total execution time of that task will not be affected by adding extraneous computation or communication time.

Braun *et al.* [5] defined the matching of tasks to machines and scheduling the execution order of them as *mapping*. They stated that the problem of optimally mapping tasks to machines in a heterogeneous computing suite had been shown to be NP-complete. The heuristics developed for this problem have different underlying assumptions. Therefore, they selected, adapted, implemented, and compared 11 heuristics from the literature under one set of common assumptions. They defined a metatask as a collection of independent tasks with no intertask data dependencies. They performed the mapping statically and their goal was to minimize the total execution time of the metatask. They assumed that a machine

executes a single task at a time, according to the order in which the tasks were assigned. The number of tasks to be executed and the number of machines were known beforehand. The heuristics they compared are: Opportunistic Load Balancing (OLB), Minimum Execution Time (MET), Minimum Completion Time (MCT), Min-min heuristic, Max-min heuristic, Duplex heuristic (combination of the Min-min and Max-min heuristics), Genetic Algorithms (GAs), Simulated Annealing (SA), Genetic Simulated Annealing (GSA), Tabu search and A^* heuristic. They concluded that for the situations, implementations and parameter values they used, GA consistently gave the best results.

Ali *et al.* [6] also stated that, in general, finding a mapping for tasks that optimizes some performance measure is an NP-complete problem. Therefore, instead of using an exhaustive search, it is necessary to have heuristics to find near-optimal mappings. They mention that in *static* mapping heuristics, the mapping decisions are made before the execution of the meta-task and a static mapper considers the entire meta-task to be executed when making decisions. They advise to use a static mapping heuristic if (1) the tasks that comprise the meta-task are known *a priori*, (2) predictions about the available heterogeneous computing resources are likely to be accurate, and (3) the estimated expected execution time of each task on each machine in the suite is known reasonably accurately.

Yang and Chang [7] mentioned that in order to solve parallel loop scheduling problems on heterogeneous cluster environments, self-scheduling schemes were used. They claimed that such schemes worked well in moderate heterogeneous cluster environment. However, in extremely heterogeneous environment, where the performance difference between the fastest computer and the slowest computer was large, these schemes could not achieve good load balancing. Since the loops comprised a large portion of a program's parallelism, they concentrated on the parallelism available in the loops. They proposed an approach to partition loop iterations, if all the iterations of the loop could be executed in parallel. They assigned 80% workload corresponding to the CPU clock, and 20% workload using traditional self-scheduling and achieved good load balancing in extremely heterogeneous environment.

Lilja [8] declared that if the machines in a heterogeneous system have radically different architectures, it was much more difficult to determine which machine would produce

the minimum execution time for a given program task. Because, the relative performance differences of the machines were dependent on the algorithm used in the application and on the problem size. In his paper, the task partitioning model was based on a *remote procedure call* (RPC) model. His strategy assigned the tasks partitioned from a program to one of two heterogeneous machines interconnected with a communications link. In other words, the problem was to determine which tasks would execute on the local machine, and which would execute on the remote machine. While making a decision, his methodology considered the communication cost and the relative performance of the two machines for each task in the program. His goal was to reduce the total execution time of the entire program compared to the execution time of running the program on either machine separately. However, actually his strategy does not execute the tasks concurrently, i.e. in parallel. He declared that when the main program was initiated by the local machine, the arguments for the remote task were transferred to the remote machine across an interconnection network. The remote machine executed the task and returned the results. During this period, the local machine temporarily suspended its execution.

Liu *et al.* [9] mentioned *high performance parallelism*. To achieve this, the computation was divided into manageable subtasks. These subtasks were distributed to the processors within the cluster. They stated that the Message Passing Interface (MPI) library evolved a standard for writing message-passing parallel code. Using a standardized high-level programming interface, the programmer could write a highly portable parallel code, where he could exchange information among processes, and run it on any parallel machine (including network of workstation) that has MPI implementation. They pointed out that any workstation cluster needed an efficient collective communication and gave the examples of barrier, scatter and gather operations. They claimed that the efficiency of these collective communications would affect the overall performance, sometimes dramatically. If a system consisted of a set of n processors, during the reduction process $n-1$ messages would be sent by $n-1$ different processors. They suggested a reduction schedule where the slowest processor did not send any message -instead it only received message and computed the final reduction result. They used earliest possible scheduling and slowest-node-first methods.

Kim *et al.* [10] defined a distributed heterogeneous computing system as follows:

the resources had different capabilities and tasks had different requirements. They defined *matching* as assigning resources to tasks, *scheduling* as ordering the execution of tasks on each resource in a manner that exploited the heterogeneity of the resources and tasks. The *mapping* is a combination of these two operations. They declared that mapping the tasks onto machines with varied computational capabilities had been shown, in general, to be an NP-complete problem and decided to use heuristic techniques to find a near-optimal solution. In their heterogeneous environment, tasks arrived randomly, tasks were independent (i.e., no communication among tasks), and tasks had priorities and multiple deadlines. They proposed, evaluated, and compared eight dynamic heuristics: Max-Max, Max-Min, Min-Min, Queueing Table, Relative Cost, Slack Sufferage, Switching Algorithm, and Percent Best. They compared these heuristics according to the cases of loose deadlines, tight deadlines, high heterogeneity cases and low heterogeneity cases and declared the best in each case.

Wu and Shu [11] proposed a new mapping algorithm *Relative Cost (RC)* for heterogeneous computing systems where a new indicator the *relative cost* was utilized. They claimed that their new algorithm retained the advantages of the Min-min algorithm, combined the advantages of Max-min and Min-min algorithms, balanced the load very well and it generated good mapping in various heterogeneous environments. They observed that the Min-min algorithm mapped small tasks first, so it was unable to balance the load well. Their method, *RC*, separated the relative execution cost from the execution time of a task. Thus, large tasks with low costs could be assigned early for a good load balance. They concluded that their mapping algorithm outperformed the Genetic Algorithm that delivered the best performance among 11 existing algorithms, and ran much faster than GA.

Kwok *et al.* [12] considered the problems where an iterative application was to be mapped onto a dedicated heterogeneous parallel hardware platform and the execution of each iteration could be represented by a directed acyclic graph (DAG) of subtasks. Since the characteristics of the application subtasks were unknown *a priori* and would change from iteration to iteration during execution-time, it was not be feasible to use the same off-line derived mapping throughout the whole execution of the application. They offered to employ a *semi-static* methodology, which started with an initial mapping but dynamically decided whether to perform a remapping between iterations of the application. From one iteration

to another, the effects of the changing characteristics of the application's input data, called *dynamic parameters*, were observed on the application's execution time. In their study, the operating system would be able to make a heuristically-determined decision during the execution of the application whether to perform a remapping. If a remapping was needed, the operating system would select a pre-computed and stored mapping that was appropriate for the given state of the application. During the off-line phase, they used genetic algorithm to generate high quality mappings for a range of values for the dynamic parameters. During the on-line phase, the actual dynamic parameters were observed and the off-line derived mapping table was referenced to choose the most suitable mapping. To analyze the effectiveness of the proposed scheme, they compared it with a completely dynamic approach using a fast mapping heuristic and an ideal approach that uses a genetic algorithm on-line but ignores the time for remapping. The semi-static approach consistently outperformed a fast dynamic mapping heuristic, and it gave reasonable performance compared with directly using the genetic algorithm on-line for a wide range of task graph structures. They declared a limitation of their semi-static approach as the additional off-line execution time needed to build the mapping table.

2.2. Previous Work About Parallelizers and Parallel Program Optimization

Mazzariol [13] used the Computer-Aided Parallelization (CAP) framework developed at Ecole Polytechnique Fédérale de Lausanne (EPFL) to facilitate the design and development of pipelined parallel applications. In his thesis, he declared that the CAP tool simplified the creation of pipelined parallel distributed memory applications and he showed how parallel applications could be developed by using CAP, for example parallel linear algebra algorithms such as matrix multiplication and LU decomposition, parallel image filtering, parallel cellular automata and parallel discrete optimization algorithm (e.g. Branch and Bound) and Radiocontrol application aiming at computing in parallel the listening quotes of radio stations. He mentioned the CAP language did not explicitly provide data transfer mechanisms such as a send or receive primitive. The communications were managed automatically without programmer intervention. Since the data transfers occurred only at the end of the execution of operations, the output token of an operation was redirected to the next operation as input in the macro-dataflow. Thus, the task of writing parallel programs was simplified.

To execute tasks in parallel, CAP introduced split and merge operations. A split routine took the data of its previous operation, split it into sub-tokens and sent in a pipelined parallel manner to the next operations. A merge routine collected all sub-results and passed them to the next operation. In the thesis, he presented several parallel applications developed within the CAP framework and demonstrated the ability of CAP to produce efficient parallel solutions.

Vikram *et al.* [14] mentioned some techniques used to parallelize programs for running on a grid and their tool, OlyMPIx, which was built using these techniques. The tool was based on SUIF compiler infrastructure. In their approach, the master was responsible for distributing and redistributing the data required by the slaves. The data sent to each slave was decided by the computation requirements of that slave. The master sent only the variables which would be used during the computation at the slave threads. For simplicity, they assumed that the value of a variable could be changed only in statements in the code. Therefore, the master distributed only the right-hand side variables in all the assignment statements. After finishing the tasks, the slaves transferred all the variables on the left hand side of the assignment statements to the master. They parallelized the sequential program using olyMPIx. The parallel program was run on a grid and the number of available processors was varied from 2 to 6. When the number of threads was equal to 2, the number of threads doing the actual computation was actually 1, since one of the threads was the master and it did not share parallel iterations with the slaves. The time taken by the parallel program with $n = 2$ was slightly more than that required by the sequential version, because the program was actually not parallelized. Additionally there was an extra overhead of moving data between master and the slave which made it costlier than the sequential version. However, when the number of threads was greater than 2, the duration of the program decreased.

Rugina and Rinard [15] mentioned the difficulty of parallelizing divide and conquer algorithms, since they include recursive procedures and often use pointers into dynamically allocated memory blocks and pointer arithmetic. They designed and implemented a compiler, using the SUIF compiler infrastructure, to parallelize divide and conquer algorithms whose subproblems accessed disjoint regions of dynamically allocated arrays. They implemented both pointer analysis and a set of new symbolic analysis algorithms. Using these algorithms, the compiler found the regions of memory accessed in procedures that used

pointers and pointer arithmetic. With the region access information, the compiler detected independent calls to these procedures and generated code, in the Cilk parallel programming language, that could execute the independent calls in parallel. They experimentally showed that the compiler was capable of automatically extracting concurrency from optimized implementations of divide and conquer algorithms.

Kung and Subhlok [16] described a new approach for generating efficient parallel programs from sequential blocked linear algebra programs, automatically. In their implementation, they executed block routines on the iWarp processor array. They distributed the matrices over the array such that they could replace the sequential block routine calls (in the original program) with corresponding parallel block routines automatically. They stated that using this data distribution scheme, the parallelization process was simplified and they could generate the parallel versions of programs automatically. They used a fixed set of block routines in the LAPACK library, which was written using block routines for linear algebra computations. In their approach, the matrices were equally divided into two dimensional block sub-matrices and each block was distributed uniformly over the processor array. Before initiating and during the computation, the sub-blocks in each processor were shifted by a fixed number of steps and the operands and results were left in place. The authors claimed that on iWarp, the computation and communication could be overlapped. They implemented the LU decomposition. As a conclusion, they declared that to improve their implementation, they had to use faster mathematical operations and an efficient communication library to reduce the communication overhead by eliminating unnecessary communication.

Özcan and Onbaşıoğlu [17, 18] assumed that a parallel algorithm was made up of consecutive stages (i.e. levels). These stages might be a single loop or a group of nested loops. The levels could be executed on different number of processors according to their computation requirements. Since the data were distributed among the processors, communication might be required before the execution of the levels. They stated that, the performance of a serial algorithm could be expressed in terms of the problem size. However, the performance of a parallel algorithm would depend on not only the problem size but also the number of available processors, the distribution of the data among the processors, and the methods used for transferring data among the processors. The aim of their studies was to minimize the

total execution time of the program by choosing the best configuration for both degree of parallelism and the data distribution. They used a parallel algorithm for reducing matrices to Hessenberg form, which had five levels. They also used Dongarra's Benchmark, which had 11 loops.

In [17], they proposed a Genetic Algorithm for parallel algorithms to find the best number of processors and the best data distribution method. They concluded that steady state genetic algorithm was better than transgenerational genetic algorithm. As a future work, they offered to use different hill climbing techniques.

In [18], they proposed a memetic algorithm (MA). They compared steady state memetic algorithm with transgenerational memetic algorithm using different crossover operators and hill-climbing methods. They implemented a self-adaptive MA and four different hill-climbing methods. They used computationally intensive, communication intensive, and mixed problem instances. They concluded that memetic algorithms and multimeme strategies could successfully solve the problem of optimizing the performance of parallel codes. They stated that the results of the MA Solver could either be used at program development stage or by an automatic parallelizing compiler.

In this thesis, the levels of the user's sequential program are matched with some or all of the processors in one of the processor groups. The levels execute in their original order, according to the user's program. If the sequential execution time of a level is long enough, the level is parallelized and executed on more than one machine, in order to decrease the computation time. Between levels, the processors may need to exchange data. Since machines communicate via a high-speed link, the communication time will not be too long. Therefore, it is expected that the total execution time of the parallelized program will be less than that of the sequential program.

3. USER PROGRAM PARSER

A parser, which parses and gathers information about an input program, was implemented by Assist. Prof. Dr. Esin Onbaşıoğlu at Yeditepe University. (In this chapter, "the original parser" refers to this parser.) A modified version of this parser is used as the User Program Parser. The User Program Parser parses the user's sequential program, written in a C like language, and gathers information about levels and arrays such as:

- the number of levels, matrices and vectors,
- the name, dimension and size of the arrays,
- the statements in the levels (expression or loop statements),
- the loop counters (e.g. i, j, k , etc.) of the loop statements,
- which arrays are used in expression statements,
- the position of the arrays according to the equal sign (e.g. RHS, LHS) and
- the indexes of the arrays.

```
for(i = 0; i < SIZE; i++)
{
    for(j = 0; j < SIZE; j++)
    {
        a[i][j] = b[i][j] + c[i][j];
    }
}
```

Figure 3.1. An example for a computation level

In order to determine the possible decomposition types for the arrays of each computation level, the parser compares the indexes of the arrays and for-loop counter. In Figure 3.1, an example level is given. The counter of the outer loop, i , is used to access the rows of the matrices. The counter of the inner loop, j , is used to access to columns of the matrices. In the parallel program, if the *for loop limit* ($SIZE$) of the outer loop is divided, then the decomposition type of the matrices will be Horizontal or Horizontal Block (according to the divisor). If the *for loop limit* of the inner loop is divided, then the decomposition type of the matrices will be Vertical or Vertical Block. If both of the *for loop limits* are divided, the decompo-

sition type will be Rectangular. For Replicated and None decomposition types, none of the *for loop limits* are divided. Özcan and Onbaşıoğlu [17, 18] explained data decomposition and alignment in details. After parsing the sequential program and determining the possible decomposition types for the arrays, the necessary part of the gathered information and the decompositions are written to the output files.

The original parser classifies the arrays in a level according to their position relative to the equal sign. There are three data structures, which store LHS-arrays, RHS-arrays and BOTH-SIDE-arrays separately. (In this chapter, "array" means the matrices and vectors in the user's program. "Data structure" means the arrays which store information about the matrices and vectors of the user's program.) Inside a level, there may be more than one expression statement and an array may be used more than once. In such a case, the original parser stores the name of the array once and determines possible decompositions for one of these repetitions.

However, the Array Information Table Generator needs to know more details about the sequential program. If an array repeats in a level (i.e. it is used more than once in a level), the number of times it repeats and its position in each repetition must be known. Therefore, in a new function, the expression statements are re-examined and the arrays in those statements are re-classified. This time, each repetition of an array is stored according to its position. Moreover, by making a modification, if an array repeats itself in a level, different decomposition types may be proposed for each repetition. Similar to the original parser, three data structures are declared for LHS-arrays, RHS-arrays and BOTH-SIDE-arrays, named as LHS-array-structure, RHS-array-structure and BOTH-SIDE-array-structure. These data structures store the name, position (RHS or LHS of equal sign) and possible decompositions of the arrays. Here the difference is, an array is stored as many times as it is used in the level.

An array which is on both sides of equal sign may have the same decomposition type on both sides, or different decomposition types. Moreover, a BOTH-SIDE-array may repeat itself on RHS more than once. For example, it may be used once on LHS and twice on RHS. All repetitions on RHS may not have the same decomposition type with the one on LHS. For this reason, the BOTH-SIDE-arrays are classified again in detail.

The position and decomposition of an array (at each usage in a level) are important for Array Information Table Generator. For example, in a sequential program, the user used an array in the first, second and fourth levels. For the usage in the fourth level, Array Information Table Generator has to determine from which previous level the data pieces of the array can be taken, in the parallel program. Here, the position of the array according to the equal sign is important, because if the array is on LHS in the second level, then the contents of the array are re-calculated. The previous decomposition of this array in the first level has old, and wrong, data. Therefore, the array must be transferred from the second level to the fourth level. The decompositions of the array in these levels are also important. The array must be converted from its previous decomposition type to the next decomposition type. In this chapter, the modifications for adapting the original parser program to parallelization project are explained.

There are two kinds of statements in the user's program: *expression statements* and *loop statements*. In all computation levels, the loop statements are counted to find the number of *for loops* in the level. For all expression statements in the level, all matrices and vectors are classified as RHS, LHS or BOTH-SIDE array as follows:

- If the array is a BOTH-SIDE-array of the level, its name, RHS/LHS value, number of possible decompositions and all possible decompositions are copied to BOTH-SIDE-array-structure. The BOTH-SIDE-array may repeat itself on right-hand-side for many times, in the statements of a level. Here, the User Program Parser is interested in the one on the left-hand-side of the first statement in this level. If this BOTH-SIDE-array is on the LHS of the equal sign in the first expression statement of this level, then its name, number of indexes, name of each index, (e.g. i, j), number of possible decompositions, the possible decompositions are written to an output file.
- If this array is on the left-hand-side of equal sign, its name, RHS/LHS value, number of possible decompositions and all possible decompositions are copied to LHS-array-structure. Since the User Program Parser is interested in the left-hand-side array of the first statement in the level, it checks for it. If this LHS-array is in the first expression statement of this level, then its name, number of indexes, name of each index (e.g. i, j), number of possible decompositions, the possible decompositions are written to the

output file.

- If this array is on the right-hand-side of equal sign, its name, RHS/LHS value, number of possible decompositions and all possible decompositions are copied to RHS-array-structure.

After classifying the arrays of expression statements in this level as RHS, LHS or BOTH-SIDE array, the information about the possible decompositions are written to another output file as follows: For all possible decompositions of this level, the arrays only on the right-hand-side of equal sign (not both sides) are written first. For all RHS-arrays, the name, decomposition type and "1" are written to the file. ("1" means the array is on the RHS only.)

Then, the matrices on both sides of equal sign are written to the file. As mentioned above, the original parser generates a data structure, where a BOTH-SIDE-array is stored only once. Here, another data structure is used where each repetition of a BOTH-SIDE-array is stored. For example, if array A is used 3 times in this level (on LHS for once and on RHS for twice), there is only 1 item for array A in the old version and there are 3 items for array A in the new version of the program. For each BOTH-SIDE-array, the answers of the following questions searched:

- How many times the array on the left-hand-side repeats on the right-hand-side of equal sign?
- How many of the repeated right-hand-side arrays have got the SAME decomposition type with the one on the left-hand-side of equal sign?
- How many of the repeated right-hand-side arrays have got DIFFERENT decomposition type with the one on the left-hand-side of equal sign?

If there is a BOTH-SIDE-array, whose decomposition types on RHS and LHS are the same, the name, decomposition type and "2" are written to the file. ("2" means the array is on the both sides of equal sign with the same decomposition type.) Otherwise (the LHS and RHS decompositions are different), the name, decomposition type of the one on LHS and "3" are written to the file. (Here, "3" means the array is on the both sides of equal sign with different decomposition types and this is the decomposition type of the one on the LHS.)

For the RHS repetitions of the BOTH-SIDE-array, the name, decomposition type of the ones on RHS and "0" are written to the file. ("0" means the array is on the both sides of equal sign with different decomposition types and this is the decomposition type of the one on the RHS.)

Finally, the arrays only on the left-hand-side of equal sign (not both sides) are written. For all LHS arrays, the name, decomposition type and "3" are written to the file. (Here, "3" means the array is on the LHS only.) The 0, 1, 2 and 3 values will be explained in Section 6.1.1 in detail.



4. CONVERSION FUNCTIONS

Before each computation level begins, the arrays to be used in the computations must be transferred to the processors which will execute the level. In [17, 18], some communication structures, such as Shift and Multiphase, are mentioned. When the Shift structure is employed, beginning from the sender, each processor transfers the data to its neighbor, until the data reach to the receiver processor. The Multiphase structure includes a gather and a scatter operation. If this structure is employed in the communication, all processors have to send their data to a root processor at the end of the level. The root processor collects and re-distributes the data to be used in the next level. If the root processor itself or its communication speed is slow, then using this communication structure will cause a bottleneck. Instead of using the known communication structures, a new library, named as Conversion Functions Library, is written.

The Conversion Functions are proposed to transfer data directly from the sender to the receiver. The idea behind the Conversion Functions is to avoid the unnecessary communications of the processors, in order to reduce the communication time of the parallel program. All processors are included in the same MPI communication channel, but they may be grouped according to their properties. The Conversion Functions allow the processors to send the data directly to the receiver, even if the receiver belongs to another group.

Additionally, to reduce the communications among the processors, the processors are not allowed to send data to themselves. Instead, they copy the data to be sent. In the Conversion Function algorithms, in all cases, where *copying the data* is possible, the data are not sent or received. In this chapter, the Conversion Functions will be explained in detail.

Dongarra and Lastovetsky [1] declared that a good parallel application for a homogeneous distributed memory multiprocessor system would try to evenly distribute computations over the available processors. They claimed that this very distribution would ensure the maximal speedup on the system which consists of identical processors. Since the processors of a group are identical (i.e. the groups are homogeneous), the Conversion Functions transfer

equal-sized blocks of data.

4.1. The Butterfly Algorithm

Dongarra and Lastovetsky stated that MPI had a much richer set of communication functions and gave the example of nonblocking send [1]. However, in this study, it was seen that MPI_Send becomes a blocking function when the data size is increased too much (explained in Section 4.1.3). Therefore, the Butterfly Algorithm is used in some of the Conversion Functions.

In this section, the original Butterfly Algorithm will be explained. However, a modified version of the Butterfly Algorithm is used. The reason for modifying and the way it is modified will be mentioned in detail.

4.1.1. The Original Butterfly Algorithm

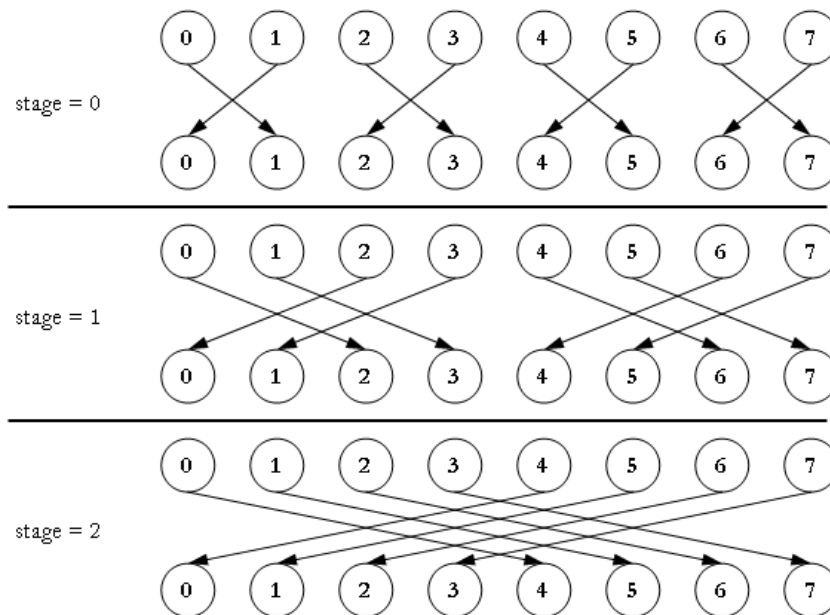


Figure 4.1. The original Butterfly Algorithm

The original Butterfly Algorithm allows processors to exchange data with **some** of the other processors. Figure 4.1 shows the data exchange among eight processors. In stage 0, the processors 0 and 1, 2 and 3, 4 and 5, 6 and 7 exchange their data. In stage 1, the exchanges

happen between processors 0 and 2, 1 and 3, 4 and 6, 5 and 7. In stage 2, the processors 0 and 4, 1 and 5, 2 and 6, 3 and 7 exchange their data.

The numbers in Table 4.1 represent the stage number in which the exchange happened. The empty cells in the table show that the original Butterfly Algorithm does not allow processors exchange data with all other processors. For example, processors 0 and 3 cannot send data to each other.

Table 4.1. The original Butterfly Algorithm among eight processors

	P0	P1	P2	P3	P4	P5	P6	P7
P0		0	1		2			
P1	0			1		2		
P2	1			0			2	
P3		1	0					2
P4	2					0	1	
P5		2			0			1
P6			2		1			0
P7				2		1	0	

4.1.2. The Modified Butterfly Algorithm

In the Conversion Functions, a processor should be able to send data to any other processor. Since the original Butterfly Algorithm does not allow this, a modified version of the algorithm is used. The difference between the original and modified Butterfly Algorithms is: The original Butterfly Algorithm does not allow a processor to send data to **any** other processor. A processor can only send data to **some** of the other processors. The Modified Butterfly Algorithm allows processors to exchange data with **all** other processors.

In the Modified Butterfly Algorithm, the processors are grouped. All the processors in a group are either senders or receivers at a time. All members of the sender group send data to all members of the receiver group. In each stage of the Modified Butterfly Algorithm, the processors are re-grouped. The groups include 2^s processors, where s is the number of

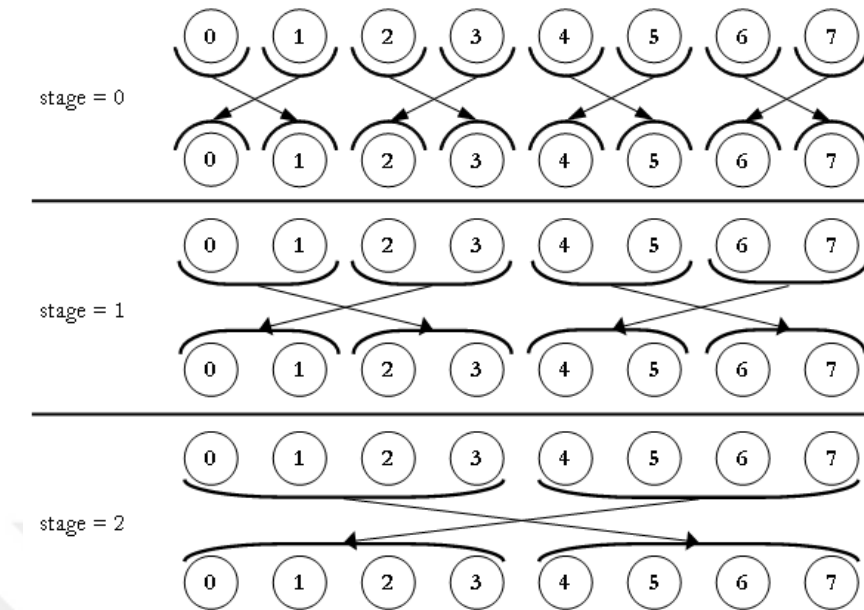


Figure 4.2. The Modified Butterfly Algorithm

Table 4.2. The Modified Butterfly Algorithm, where $p = 8$ and $q = 8$

	P0	P1	P2	P3	P4	P5	P6	P7
stage 0	S1	R0	S3	R2	S5	R4	S7	R6
stage 1	S2	S2	R0	R0	S6	S6	R4	R4
	S3	S3	R1	R1	S7	S7	R5	R5
stage 2	S4	S4	S4	S4	R0	R0	R0	R0
	S5	S5	S5	S5	R1	R1	R1	R1
	S6	S6	S6	S6	R2	R2	R2	R2
	S7	S7	S7	S7	R3	R3	R3	R3
stage 0	R1	S0	R3	S2	R5	S4	R7	S6
stage 1	R2	R2	S0	S0	R6	R6	S4	S4
	R3	R3	S1	S1	R7	R7	S5	S5
stage 2	R4	R4	R4	R4	S0	S0	S0	S0
	R5	R5	R5	R5	S1	S1	S1	S1
	R6	R6	R6	R6	S2	S2	S2	S2
	R7	R7	R7	R7	S3	S3	S3	S3
Copy	C0	C1	C2	C3	C4	C5	C6	C7

the current stage. Figure 4.2 and Table 4.2 show the data exchange among eight processors. Similar to the original Butterfly Algorithm, in stage 0, the processors 0 and 1, 2 and 3, 4 and 5, 6 and 7 exchange their data. [In Tables 4.2, 4.3 and 4.4, **P0-P7** are the processors. *S* is an abbreviation for "send to processor", *R* is an abbreviation for "receive from processor" and *C* is an abbreviation for "copy to processor".]

In the original Butterfly Algorithm - stage 1, the exchanges happen between processors 0 and 2, 4 and 6, etc. Similar to the original Butterfly Algorithm, here, the groups begin with processors 0, 2, 4 and 6. In the Modified Butterfly Algorithm, since the groups include 2^s processors, in stage 1, the groups include 2 processors. The exchanges happen between groups: 0-1 and 2-3, 4-5 and 6-7. All members in a group either send or receive data. Processor 0 and 1 send data to both 2 and 3. Processor 4 and 5 send data to both 6 and 7. Then, the senders become receiver and receivers become sender: Processor 2 and 3 send data to both 0 and 1. Processor 6 and 7 send data to both 4 and 5.

In the original Butterfly Algorithm - stage 2, the exchanges happen between processors 0 and 4, etc. Similar to the original Butterfly Algorithm, here, the groups begin with processors 0 and 4. In the Modified Butterfly Algorithm, since the groups include 2^s processors, in stage 2, the groups include 4 processors. The exchanges happen between groups: 0-1-2-3 and 4-5-6-7.

The example in Figure 4.2 shows the data transfer among 8 processors. In other words, the data are sent from 8 processors to 8 processors. Here, the previous processor number and the next processor number are equal. If data are sent from 4 processors to 16 processors, the previous processor number is 4 and the next processor number is 16. In the Conversion Functions, the previous processor number is represented by p and the next processor number is represented by q .

If p equals q , after performing all stages of the Modified Butterfly Algorithm, all processors have exchanged the necessary data (as shown in Figure 4.2). If p is less than q , the processors whose ID is less than p send data to the processors whose ID is processors whose ID is greater than or equal to p . If p is greater than q , the processors whose ID is greater than

or equal to q send data to the processors whose ID is less than q . These rules are explained below (Figures 4.3-4.4):

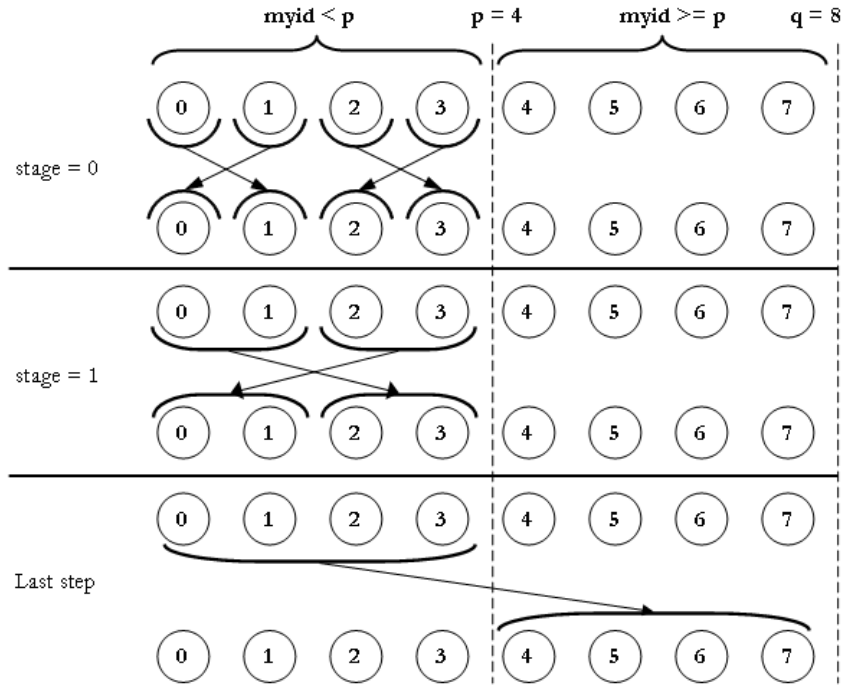


Figure 4.3. The Modified Butterfly Algorithm, where $p = 4$ and $q = 8$

In Figure 4.3 and Table 4.3, p equals 4 and q equals 8. In stages 0 and 1, 4 processors exchange data among themselves. Then, in the last step, since $p < q$, the processors whose ID is less than p (i.e. 0, 1, 2, 3 where $p = 4$) send data to the processors whose ID is greater than or equal to p (i.e. 4, 5, 6, 7 where $p = 4$).

In Figure 4.4 and Table 4.4, p equals 8, q equals 4. In stages 0 and 1, 4 processors exchange data among themselves. Then, in the last step, since $p > q$, the processors whose ID is greater than or equal to q (i.e. 4, 5, 6, 7 where $q = 4$) send data to the processors whose ID is less than q (i.e. 0, 1, 2, 3 where $q = 4$).

4.1.3. The Reason for Using Modified Butterfly Algorithm

The Conversion Functions, which were written to convert data from one decomposition type to a different decomposition type, executed correctly for small data sizes. However, using the same code but only increasing the data size, the conversion program was blocked.

Table 4.3. The Modified Butterfly Algorithm, where $p = 4$ and $q = 8$

	P0	P1	P2	P3	P4	P5	P6	P7
stage 0	S1	R0	S3	R2	-	-	-	-
stage 1	S2	S2	R0	R0	-	-	-	-
	S3	S3	R1	R1	-	-	-	-
stage 0	R1	S0	R3	S2	-	-	-	-
stage 1	R2	R2	S0	S0	-	-	-	-
	R3	R3	S1	S1	-	-	-	-
Last step	S4	S4	S4	S4	R0	R0	R0	R0
	S5	S5	S5	S5	R1	R1	R1	R1
	S6	S6	S6	S6	R2	R2	R2	R2
	S7	S7	S7	S7	R3	R3	R3	R3
Copy	C0	C1	C2	C3	-	-	-	-

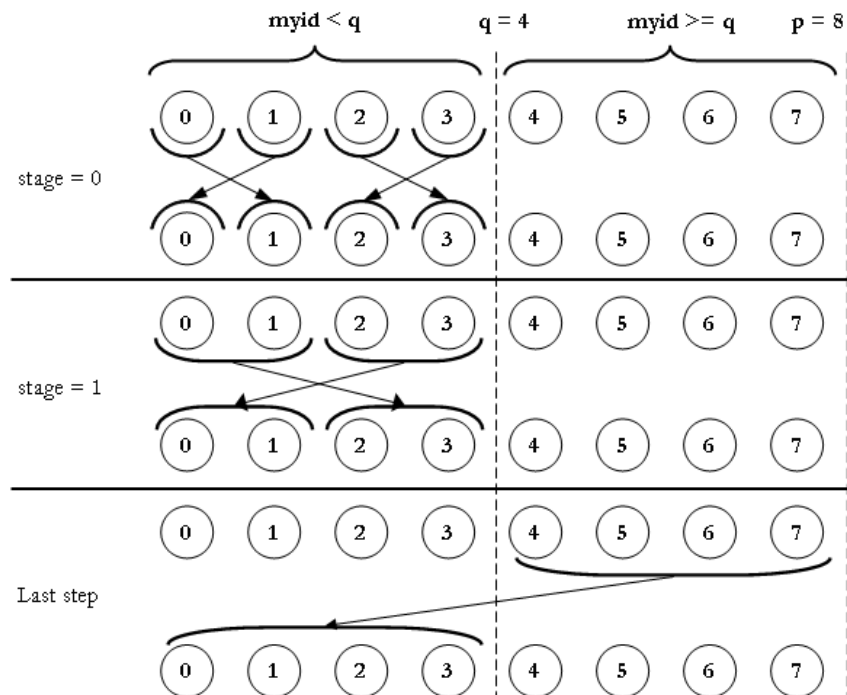
Figure 4.4. The Modified Butterfly Algorithm, where $p = 8$ and $q = 4$

Table 4.4. The Modified Butterfly Algorithm, where $p = 8$ and $q = 4$

	P0	P1	P2	P3	P4	P5	P6	P7
stage 0	S1	R0	S3	R2	-	-	-	-
stage 1	S2	S2	R0	R0	-	-	-	-
	S3	S3	R1	R1	-	-	-	-
stage 0	R1	S0	R3	S2	-	-	-	-
stage 1	R2	R2	S0	S0	-	-	-	-
	R3	R3	S1	S1	-	-	-	-
Last step	R4	R4	R4	R4	S0	S0	S0	S0
	R5	R5	R5	R5	S1	S1	S1	S1
	R6	R6	R6	R6	S2	S2	S2	S2
	R7	R7	R7	R7	S3	S3	S3	S3
Copy	C0	C1	C2	C3	-	-	-	-

The problem was caused by MPI_Send function. The manual page related with this function declares the problem as follows: "This function may block until the message is received. Whether or not MPI_Send blocks depends on factors such as how large the message is, how many messages are pending to the specific destination, whether LAMD or C2C communication is being used, etc."

During the execution of such Conversion Functions, two processors may send data to each other at the same time. For instance, processor a sends data to processor b , and processor b sends data to processor a . When the data size of the message is small, processor a does not wait for processor b to receive the data and it continues with receiving the data which were sent by processor b . However, when the data size of the message is large enough, the blocking property of MPI_Send for large messages affects the execution of the program. Processor a waits for processor b to receive the data and it cannot continue with receiving the data which were sent by processor b . Similarly, processor b waits for processor a to receive the data and it cannot continue with receiving the data which were sent by processor a . As a result, both processors are blocked because of waiting for each other to receive the data they sent.

In order to prevent two processors to send data to each other at the same time, the Modified Butterfly Algorithm is used. This algorithm allows some processors to send data and allows the other processors to receive data. Then, the senders become receiver and receivers become sender.

The Modified Butterfly Algorithm allows each processor to send data to every other processor at some time. This property is suitable for some conversions like H2V and V2H. However in most conversions, every processor does not send data to every other processor. In such conversions, the Modified Butterfly Algorithm lets processor a to send data to processor b . If the conversion algorithm requires this transfer, then processor a sends data to processor b . Otherwise, it does not send.

The Modified Butterfly Algorithm increases the duration of the send-receive process a little. However this is tolerable, because otherwise some of the Conversion Functions cannot execute with large data sizes.

Some Conversion Functions convert data from one decomposition type to the same decomposition type. Such functions are needed, because the p and q values or the group of the processors may change. For these functions, increasing the data size does not cause any problems. Because in such conversions, there does not exist a case, where two processors send data to each other. Processor a sends data to processor b , processor b sends data to processor c , etc.

4.2. Initialization, Conversion and Gather Functions

In the user's program, if an array is used for the first time, the array must be scattered to the processors which will use it in their computations. In this case, the Initialization Functions are used.

If an array was used in one of the above levels and will be used in the next level, the array is not gathered and re-distributed. Using the Conversion Functions, the pieces of the array are sent by the processors used in previous level to the processors in the next

level, directly. During the implementation of the Conversion Functions, the followings are considered:

- If the processor is a sender, to which processors it will send data and which part of its data will be sent.
- If the processor is a receiver, from which processors it will receive data and where this data will be placed.

At the end of the program, the user may want to print the results of the computations into a file. Since the arrays are decomposed and distributed to some processors, first of all, the arrays to be printed must be gathered from the processors which used them in their computations. In this case, the Gather Functions are used.

4.3. Matrix Partitioning

In this thesis, 6 types of partitioning is analyzed for matrices: Horizontal (H), Vertical (V), Rectangular (R), Horizontal Block (HB), Vertical Block (VB) and Replicated (REP). Figures 4.5-4.10 show how the data are partitioned for 4, 8 and 16 processors, for each decomposition type. Figure 4.5 shows Horizontal partitioning of a matrix on 4, 8 and 16 processors. If the size of the matrix is $N \times N$, on 4 processors with Horizontal partitioning, each processor has $N^2/4$ elements. Figure 4.6 shows Vertical partitioning of a matrix on 4, 8 and 16 processors. If the size of the matrix is $N \times N$, on 4 processors with Vertical partitioning, each processor has $N^2/4$ elements. Figure 4.7 shows Rectangular partitioning of a matrix on 4, 8 and 16 processors. If the size of the matrix is $N \times N$, on 4 processors with Rectangular partitioning, each processor has $N^2/4$ elements. Figure 4.8 shows Horizontal Block partitioning of a matrix on 4, 8 and 16 processors. If the size of the matrix is $N \times N$, on 4 processors with Horizontal Block partitioning, each processor has $N^2/2$ elements. Figure 4.9 shows Vertical Block partitioning of a matrix on 4, 8 and 16 processors. If the size of the matrix is $N \times N$, on 4 processors with Vertical Block partitioning, each processor has $N^2/2$ elements. Figure 4.10 shows Replicated partitioning of a matrix on 4, 8 and 16 processors. If the size of the matrix is $N \times N$, on 4 processors with Replicated partitioning, each processor has N^2 elements.

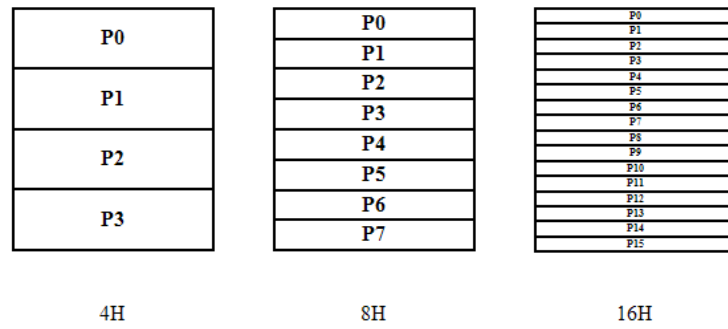


Figure 4.5. Horizontal (H) decomposition of matrices for 4, 8 and 16 processors

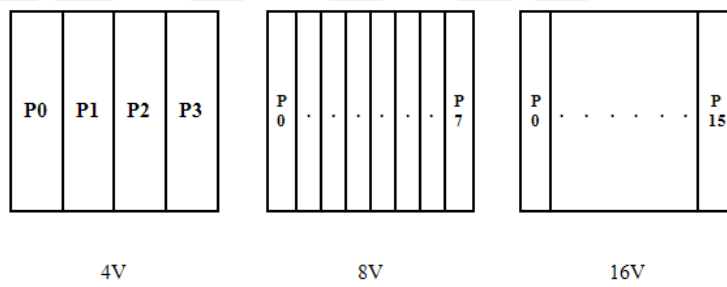


Figure 4.6. Vertical (V) decomposition of matrices for 4, 8 and 16 processors

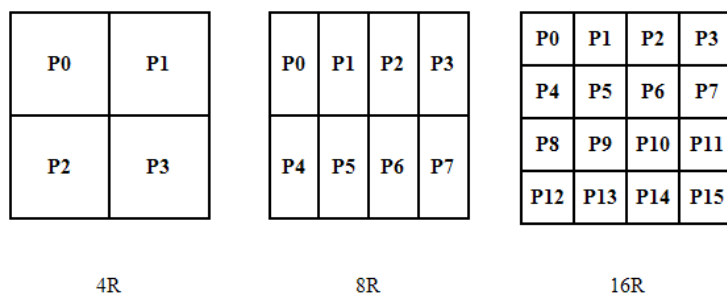


Figure 4.7. Rectangular (R) decomposition of matrices for 4, 8 and 16 processors

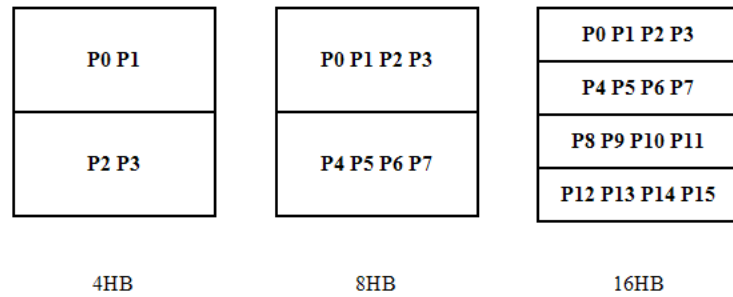


Figure 4.8. Horizontal Block (HB) decomposition of matrices for 4, 8 and 16 processors

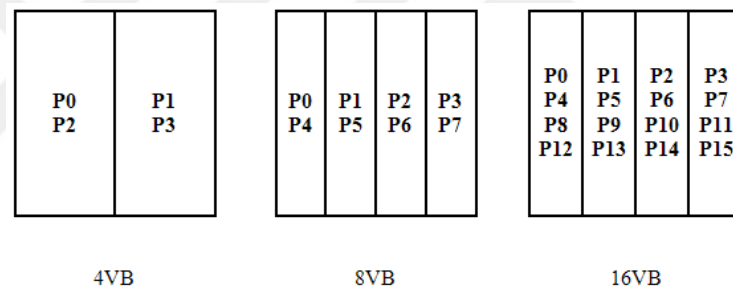


Figure 4.9. Vertical Block (VB) decomposition of matrices for 4, 8 and 16 processors

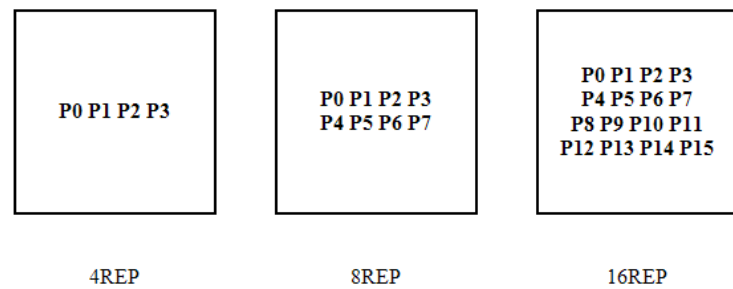


Figure 4.10. Replicated (REP) decomposition of matrices for 4, 8 and 16 processors

4.4. Conversion Functions for Matrices

The Conversion Functions transfer the data of an array such that the decomposition of the array changes from one type to another. Moreover, the data are transferred from p processors in one processor group (i.e. *previous group*) to q processors in another processor group (i.e. *next group*).

In Sections 4.4.1-4.4.36, the Conversion Functions for matrices are explained with examples. In the Figures 4.11-4.82, if the *previous group* and the *next group* are the same (i.e. if the senders and receivers are the processors of the same group), then the shaded areas are copied, not sent and received.

4.4.1. Horizontal to Horizontal Conversion for Matrices (mH2H)

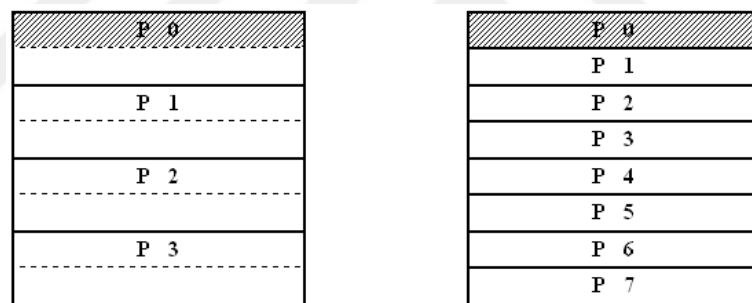


Figure 4.11. Conversion from 4H to 8H

4H $\rightarrow$ 8H: In Figure 4.11, each sender divides its data into two pieces and sends each piece to another receiver. E.g. P0 sends its data to P0 and P1. Each receiver receives one piece of data.

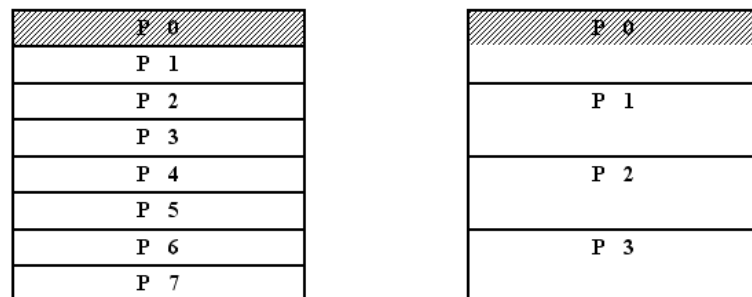


Figure 4.12. Conversion from 8H to 4H

8H $\rightarrow$ 4H: In Figure 4.12, each sender sends one piece of data. Each receiver receives two pieces of data from different processors. E.g. P0 receives its data from P0 and P1.

4.4.2. Horizontal to Vertical Conversion for Matrices (mH2V)

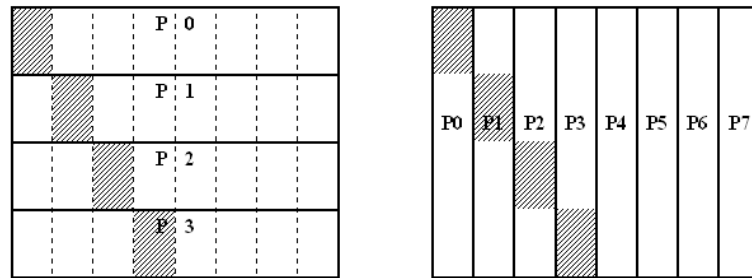


Figure 4.13. Conversion from 4H to 8V

4H $\rightarrow$ 8V: In Figure 4.13, each sender divides its data into eight pieces and sends each piece to another receiver. E.g. P0 sends its first piece to P0, its second piece to P1, its third piece to P2, etc. Each receiver receives four pieces of data, one piece from each sender. The received pieces are ordered by the senders' ID.

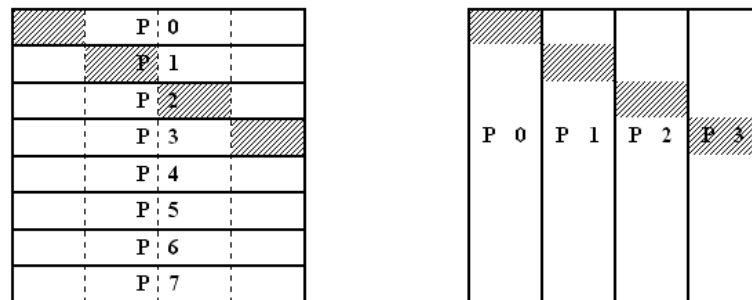


Figure 4.14. Conversion from 8H to 4V

8H $\rightarrow$ 4V: In Figure 4.14, each sender divides its data into four pieces and sends each piece to another receiver. E.g. P0 sends its first piece to P0, its second piece to P1, its third piece to P2, its fourth piece to P3. Each receiver receives eight pieces of data, one piece from each sender. The received pieces are ordered by the senders' ID.

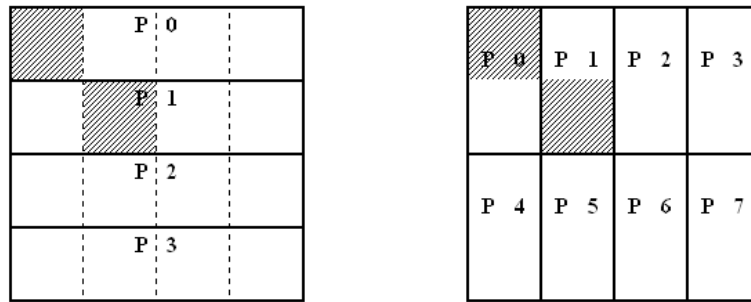


Figure 4.15. Conversion from 4H to 8R

4.4.3. Horizontal to Rectangular Conversion for Matrices (mH2R)

4H $\rightarrow$ 8R: In Figure 4.15, each sender divides its data into four pieces and sends each piece to another receiver. E.g. P0 sends its first piece to P0, its second piece to P1, its third piece to P2, its fourth piece to P3. Each receiver receives two pieces of data from different senders. E.g. P0 receives its data from P0 and P1. The received pieces are ordered by the senders' ID.

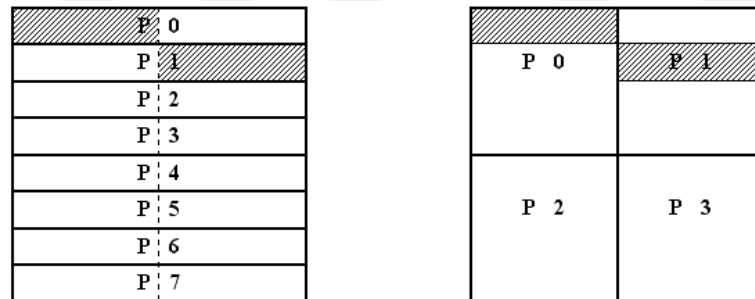


Figure 4.16. Conversion from 8H to 4R

8H $\rightarrow$ 4R: In Figure 4.16, each sender divides its data into two pieces and sends each piece to another receiver. E.g. P0 sends its first piece to P0 and its second piece to P1. Each receiver receives four pieces of data from different senders. The received pieces are ordered by the senders' ID.

4.4.4. Horizontal to Horizontal Block Conversion for Matrices (mH2HB)

4H $\rightarrow$ 8HB: In Figure 4.17, each sender sends its data to more than one receiver. E.g. P0 and P1 send their data to P0, P1, P2 and P3 (P0 and P1 may copy data instead of sending).

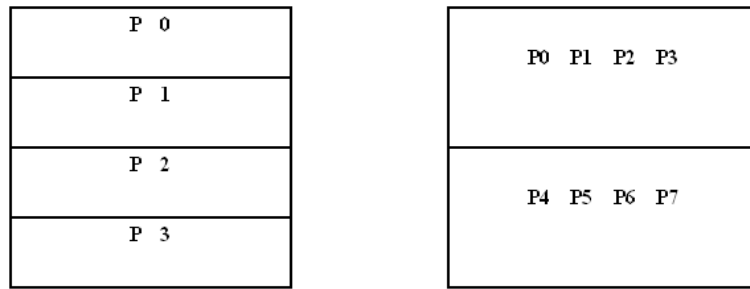


Figure 4.17. Conversion from 4H to 8HB

P2 and P3 send their data to P4, P5, P6 and P7. Each receiver receives two pieces of data. The received pieces are ordered by the senders' ID.

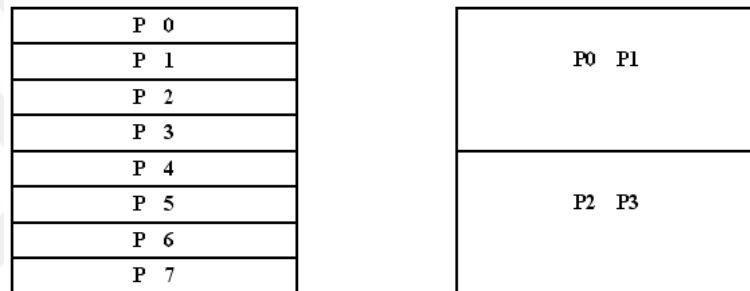


Figure 4.18. Conversion from 8H to 4HB

8H $\rightarrow$ 4HB: In Figure 4.18, each sender sends its data to more than one receiver. E.g. P0, P1, P2 and P3 send their data to P0 and P1 (P0 and P1 may copy data instead of sending). P4, P5, P6 and P7 send their data to P2 and P3. Each receiver receives four pieces of data. The received pieces are ordered by the senders' ID.

4.4.5. Horizontal to Vertical Block Conversion for Matrices (mH2VB)

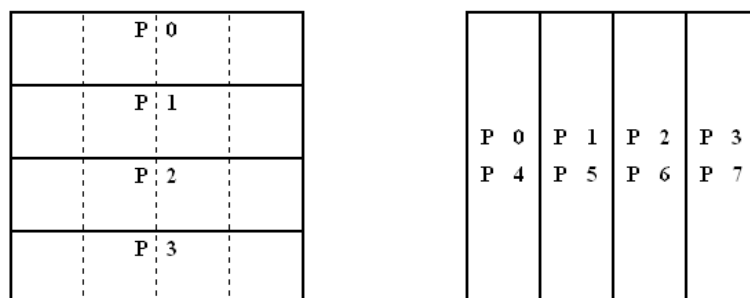


Figure 4.19. Conversion from 4H to 8VB

4H $\rightarrow$ 8VB: In Figure 4.19, each sender divides its data into four pieces and sends each piece to more than one receiver. E.g. Each sender sends its first piece to P0 and P4, its second piece to P1 and P5, its third piece to P2 and P6, its fourth piece to P3 and P7 (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives four pieces of data. The received pieces are ordered by the senders' ID.

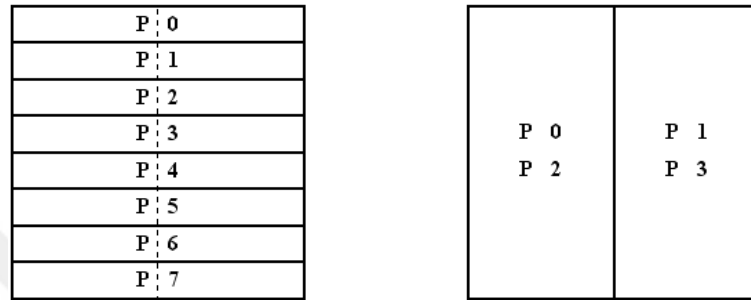


Figure 4.20. Conversion from 8H to 4VB

8H $\rightarrow$ 4VB: In Figure 4.20, each sender divides its data into two pieces and sends each piece to more than one receiver. E.g. Each sender sends its first piece to P0 and P2, its second piece to P1 and P3. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives eight pieces of data. The received pieces are ordered by the senders' ID.

4.4.6. Horizontal to Replicated Conversion for Matrices (mH2REP)

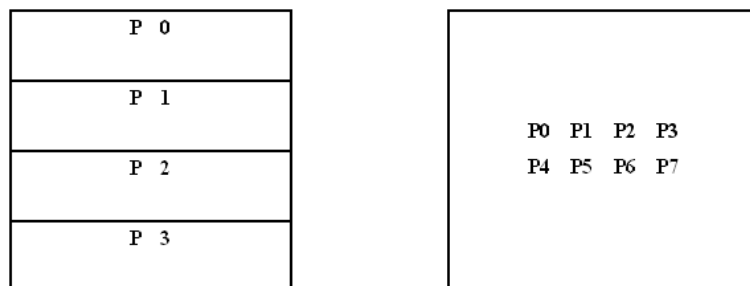


Figure 4.21. Conversion from 4H to 8REP

4H $\rightarrow$ 8REP: In Figure 4.21, each sender sends its data to all receivers. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives four pieces of data, one piece from each sender. The received pieces are ordered by the senders' ID.

8H $\rightarrow$ 4REP: In Figure 4.22, each sender sends its data to all receivers. (P0, P1, P2

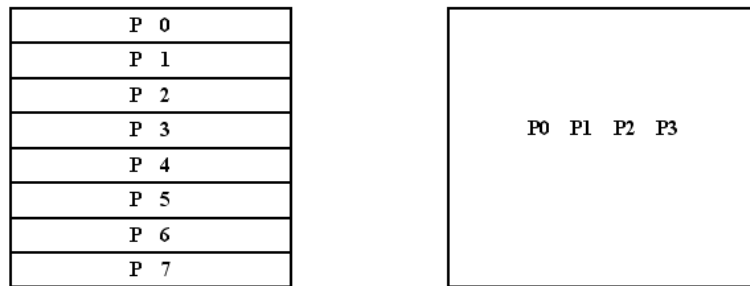


Figure 4.22. Conversion from 8H to 4REP

and P3 may copy data instead of sending). Each receiver receives eight pieces of data, one piece from each sender. The received pieces are ordered by the senders' ID.

4.4.7. Vertical to Horizontal Conversion for Matrices (mV2H)

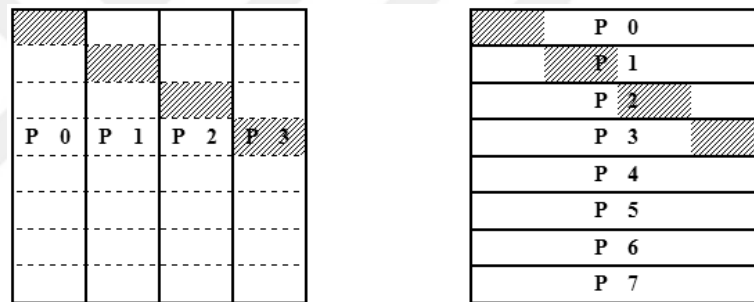


Figure 4.23. Conversion from 4V to 8H

4V $\rightarrow$ 8H: In Figure 4.23, each sender divides its data into eight pieces and sends each piece to another receiver. E.g. P0 sends its data to all receivers (from P0 to P7). Each receiver receives four pieces of data. E.g. P0 receives data from all senders (from P0 to P3).

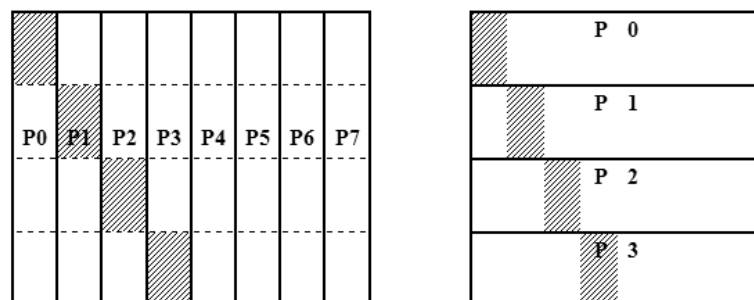


Figure 4.24. Conversion from 8V to 4H

8V $\rightarrow$ 4H: In Figure 4.24, each sender divides its data into four pieces and sends each

piece to another receiver. E.g. P0 sends its data to all receivers (from P0 to P3). Each receiver receives eight pieces of data. E.g. P0 receives data from all senders (from P0 to P7).

4.4.8. Vertical to Vertical Conversion for Matrices (mV2V)

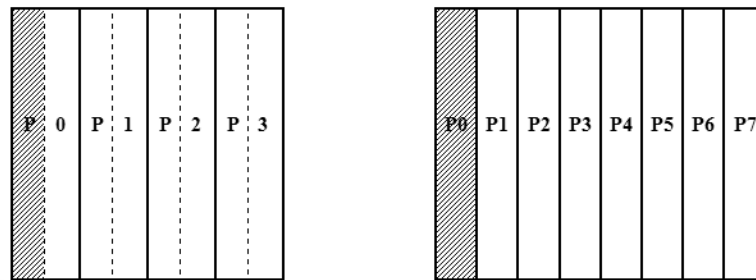


Figure 4.25. Conversion from 4V to 8V

4V $\rightarrow$ 8V: In Figure 4.25, each sender divides its data into two pieces and sends each piece to another receiver. E.g. P0 sends its first piece to P0, its second piece to P1. Each receiver receives one piece of data.

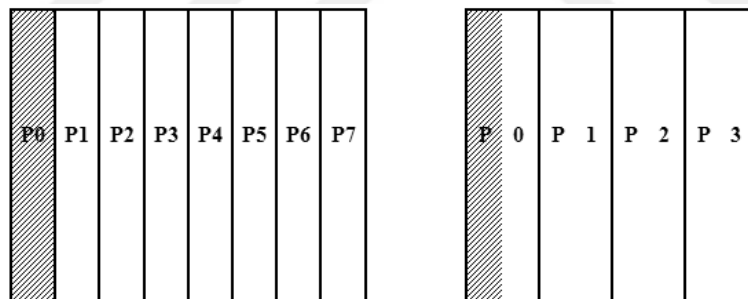


Figure 4.26. Conversion from 8V to 4V

8V $\rightarrow$ 4V: In Figure 4.26, each sender sends one of piece data. Each receiver receives two pieces of data, each piece from another sender. E.g. P0 receives its first piece from P0, its second piece from P1. The received pieces are ordered by the senders' ID.

4.4.9. Vertical to Rectangular Conversion for Matrices (mV2R)

4V $\rightarrow$ 8R: In Figure 4.27, each sender divides its data into two pieces and sends each piece to another receiver. E.g. P0 sends its first piece to P0, its second piece to P4. Each receiver receives one piece of data.

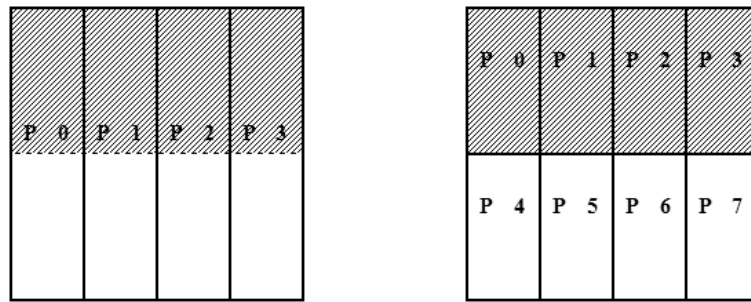


Figure 4.27. Conversion from 4V to 8R

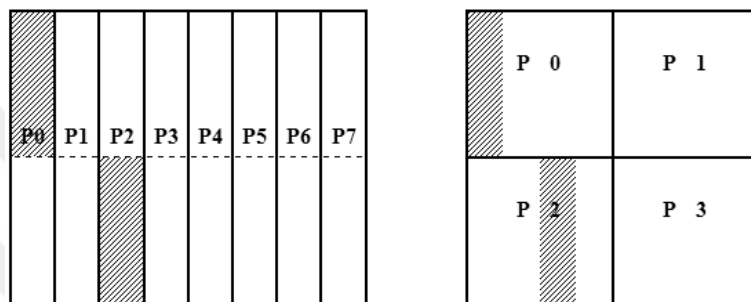


Figure 4.28. Conversion from 8V to 4R

8V $\rightarrow$ 4R: In Figure 4.28, each sender divides its data into two pieces and sends each piece to another receiver. E.g. P0 sends its first piece to P0 and its second piece to P2. Each receiver receives four pieces of data from different senders. E.g. P0 receives data from P0, P1, P2 and P3. The received pieces are ordered by the senders' ID.

4.4.10. Vertical to Horizontal Block Conversion for Matrices (mV2HB)

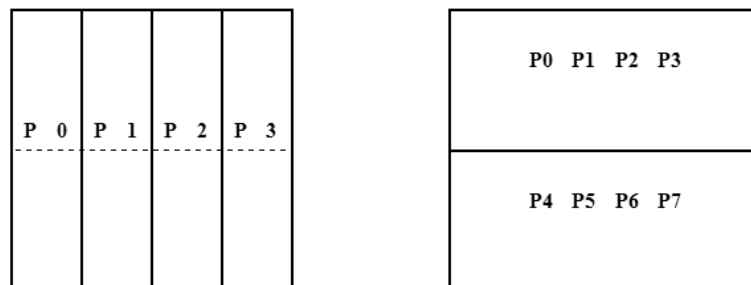


Figure 4.29. Conversion from 4V to 8HB

4V $\rightarrow$ 8HB: In Figure 4.29, each sender divides its data into two pieces and sends each piece to more than one receiver. E.g. P0, P1, P2 and P3 send their first piece to P0, P1, P2

and P3 (P0, P1, P2 and P3 may copy data instead of sending), their second piece to P4, P5, P6 and P7. Each receiver receives four pieces of data. E.g. P0 receives data from P0, P1, P2 and P3. The received pieces are ordered by the senders' ID.

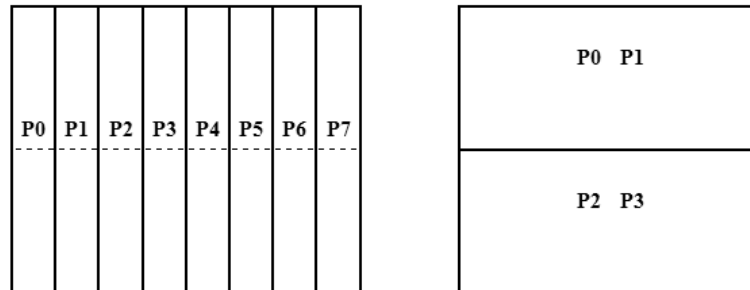


Figure 4.30. Conversion from 8V to 4HB

8V $\rightarrow$ 4HB: In Figure 4.30, each sender divides its data into two pieces and sends each piece to more than one receiver. E.g. All senders send their first piece to P0 and P1, their second piece to P2 and P3. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives eight pieces of data. E.g. P0 receives data from all senders. The received pieces are ordered by the senders' ID.

4.4.11. Vertical to Vertical Block Conversion for Matrices (mV2VB)

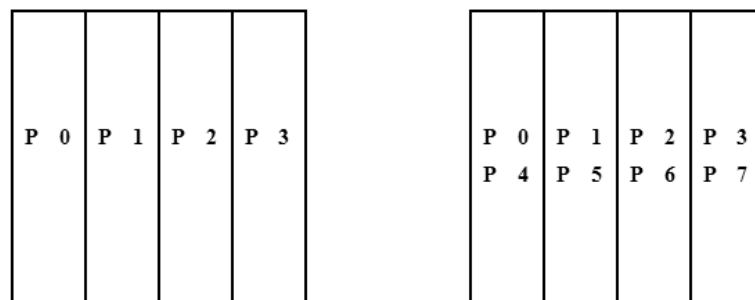


Figure 4.31. Conversion from 4V to 8VB

4V $\rightarrow$ 8VB: In Figure 4.31, senders do not divide their data. They send the data to more than one receiver. E.g. P0 sends its data to P0 and P4, P1 sends its data to P1 and P5, P2 sends its data to P2 and P6, P3 sends its data to P3 and P7. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives one piece of data.

8V $\rightarrow$ 4VB: In Figure 4.32, senders do not divide their data. They send the data to

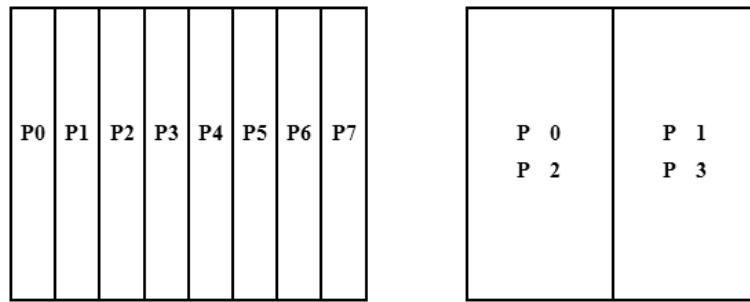


Figure 4.32. Conversion from 8V to 4VB

more than one receiver. E.g. P0, P1, P2 and P3 send their data to P0 and P2. P4, P5, P6 and P7 send their data to P1 and P3. (P0 and P2 may copy data instead of sending). Each receiver receives four pieces of data. E.g. P0 receives data from P0, P1, P2 and P3. The received pieces are ordered by the senders' ID.

4.4.12. Vertical to Replicated Conversion for Matrices (mV2REP)

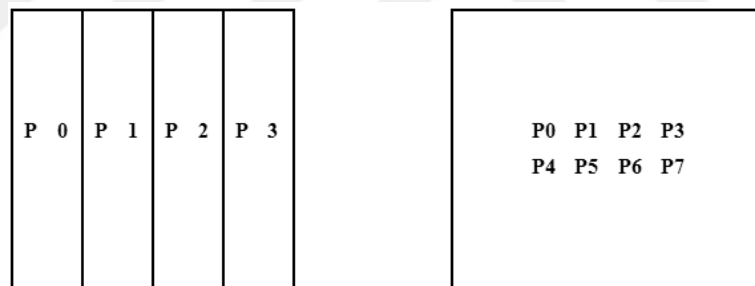


Figure 4.33. Conversion from 4V to 8REP

4V $\rightarrow$ 8REP: In Figure 4.33, each sender sends its data to all receivers. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives four pieces of data, one piece from each sender. The received pieces are ordered by the senders' ID.

8V $\rightarrow$ 4REP: In Figure 4.34, each sender sends its data to all receivers. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives eight pieces of data, one piece from each sender. The received pieces are ordered by the senders' ID.

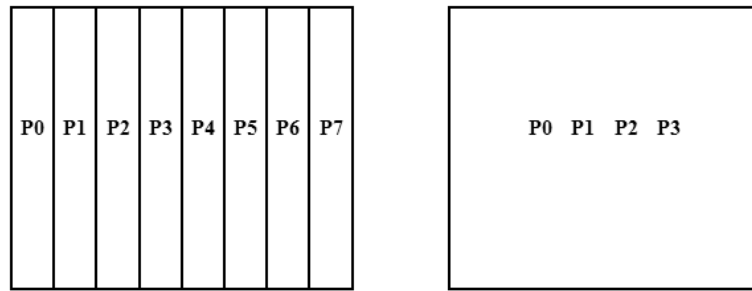


Figure 4.34. Conversion from 8V to 4REP

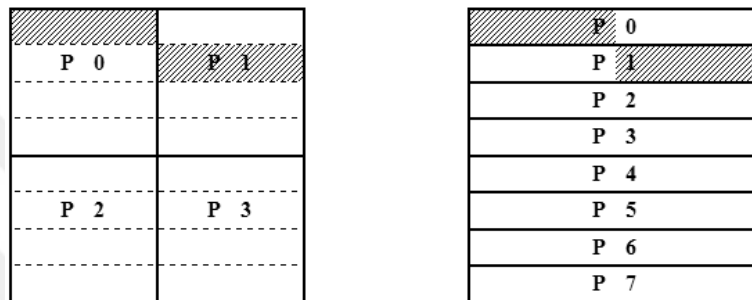


Figure 4.35. Conversion from 4R to 8H

4.4.13. Rectangular to Horizontal Conversion for Matrices (mR2H)

$4R \rightarrow 8H$: In Figure 4.35, each sender divides its data into four pieces and sends each piece to another receiver. E.g. P0 sends its pieces to P0, P1, P2 and P3. Each receiver receives two pieces of data. E.g. P0 receives data from P0 and P1. The received pieces are ordered by the senders' ID.

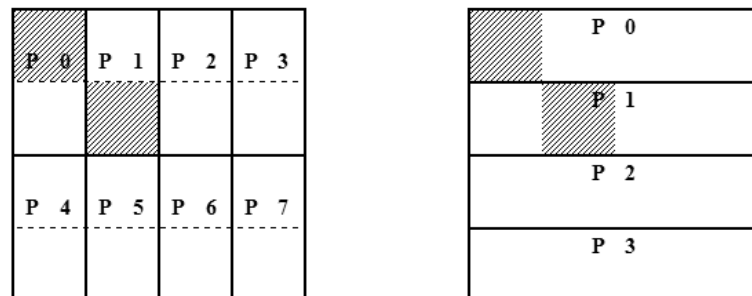


Figure 4.36. Conversion from 8R to 4H

$8R \rightarrow 4H$: In Figure 4.36, each sender divides its data into two pieces and sends each piece to another receiver. E.g. P0 sends its pieces to P0 and P1. Each receiver receives four

pieces of data. E.g. P0 receives data from P0, P1, P2 and P3. The received pieces are ordered by the senders' ID.

4.4.14. Rectangular to Vertical Conversion for Matrices (mR2V)

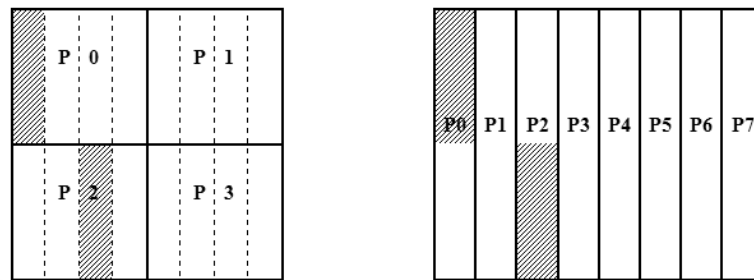


Figure 4.37. Conversion from 4R to 8V

4R $\rightarrow$ 8V: In Figure 4.37, each sender divides its data into four pieces and sends each piece to another receiver. E.g. P0 sends its first piece to P0, its second piece to P1, its third piece to P2 and its fourth piece to P3. Each receiver receives two pieces of data, each piece from another sender. E.g. P0 receives data from P0 and P2. The received pieces are ordered by the senders' ID.

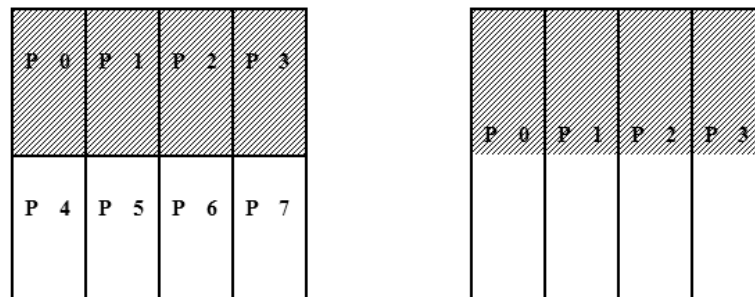


Figure 4.38. Conversion from 8R to 4V

8R $\rightarrow$ 4V: In Figure 4.38, senders do not divide their data. They send their data to a receiver. E.g. P0 sends its data to P0. Each receiver receives two pieces of data. E.g. P0 receives data from P0 and P4. The received pieces are ordered by the senders' ID.

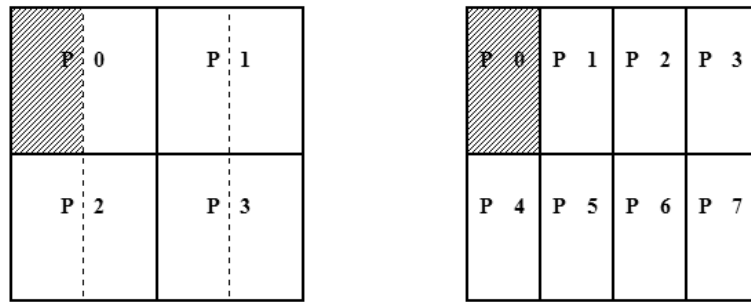


Figure 4.39. Conversion from 4R to 8R

4.4.15. Rectangular to Rectangular Conversion for Matrices (mR2R)

4R $\rightarrow$ 8R: In Figure 4.39, each sender divides its data into two pieces and sends each piece to another receiver. E.g. P0 sends its first piece to P0 and its second piece to P1. Each receiver receives one piece of data. E.g. P0 receives its data from P0.

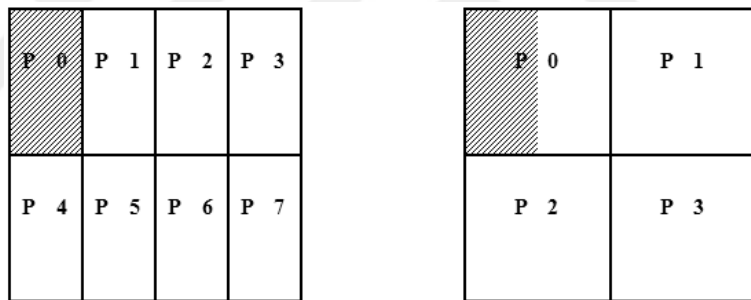


Figure 4.40. Conversion from 8R to 4R

8R $\rightarrow$ 4R: In Figure 4.40, senders do not divide their data. They send their data to a receiver. E.g. P0 sends its data to P0. Each receiver receives two pieces of data from different senders. E.g. P0 receives data from P0 and P1. The received pieces are ordered by the senders' ID.

4.4.16. Rectangular to Horizontal Block Conversion for Matrices (mR2HB)

4R $\rightarrow$ 8HB: In Figure 4.41, each sender sends its data to more than one receiver. E.g. P0 and P1 send their data to P0, P1, P2 and P3 (P0 and P1 may copy data instead of sending). P2 and P3 send their data to P4, P5, P6 and P7. Each receiver receives two pieces of data. The received pieces are ordered by the senders' ID.

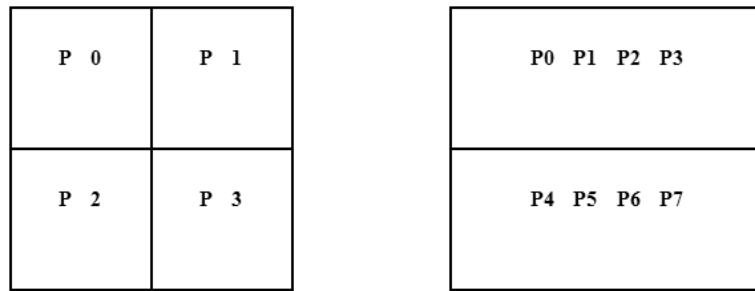


Figure 4.41. Conversion from 4R to 8HB

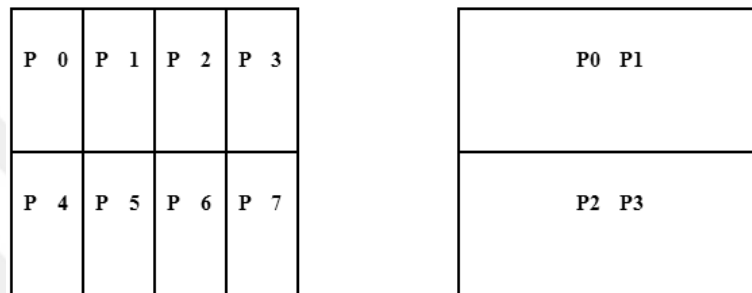


Figure 4.42. Conversion from 8R to 4HB

8R $\rightarrow$ 4HB: In Figure 4.42, each sender sends its data to more than one receiver. E.g. P0, P1, P2 and P3 send their data to P0 and P1 (P0 and P1 may copy data instead of sending). P4, P5, P6 and P7 send their data to P2 and P3. Each receiver receives four pieces of data. The received pieces are ordered by the senders' ID.

4.4.17. Rectangular to Vertical Block Conversion for Matrices (mR2VB)

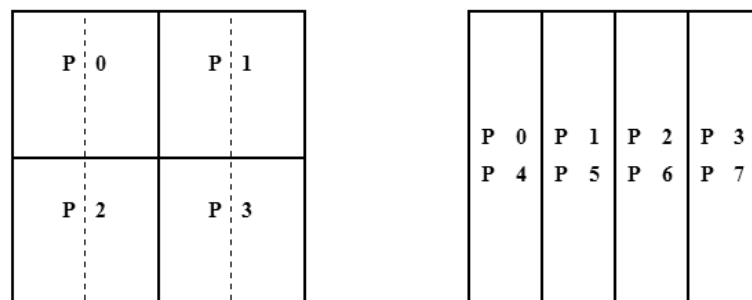


Figure 4.43. Conversion from 4R to 8VB

4R $\rightarrow$ 8VB: In Figure 4.43, each sender divides its data into two pieces and sends each piece to more than one receiver. E.g. P0 and P2 send their first piece to P0 and P4, their second piece to P1 and P5. P1 and P3 send their first piece to P2 and P6, their second piece

to P3 and P7. (P0 and P3 may copy data instead of sending). Each receiver receives two pieces of data. The received pieces are ordered by the senders' ID.

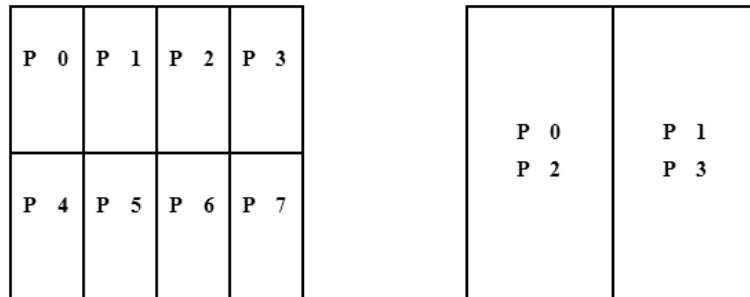


Figure 4.44. Conversion from 8R to 4VB

8R $\rightarrow$ 4VB: In Figure 4.44, senders do not divide their data and send it to more than one receiver. E.g. P0, P1, P4 and P5 send their data to P0 and P2; P2, P3, P6 and P7 send their data to P1 and P3. (P0 and P3 may copy data instead of sending). Each receiver receives four pieces of data. The received pieces are ordered by the senders' position.

4.4.18. Rectangular to Replicated Conversion for Matrices (mR2REP)



Figure 4.45. Conversion from 4R to 8REP

4R $\rightarrow$ 8REP: In Figure 4.45, each sender sends its data to all receivers. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives four pieces of data, one piece from each sender. The received pieces are ordered by the senders' position.

8R $\rightarrow$ 4REP: In Figure 4.46, each sender sends its data to all receivers. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives eight pieces of data, one piece from each sender. The received pieces are ordered by the senders' position.

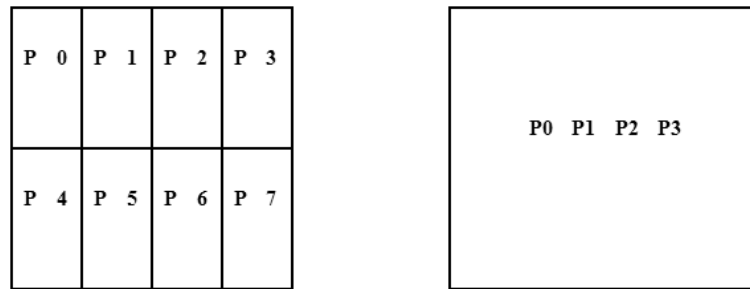


Figure 4.46. Conversion from 8R to 4REP

4.4.19. Horizontal Block to Horizontal Conversion for Matrices (mHB2H)

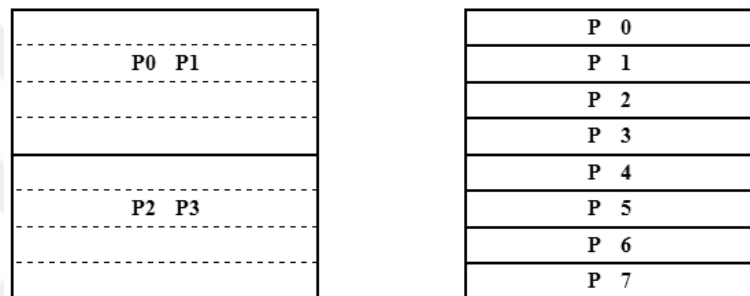


Figure 4.47. Conversion from 4HB to 8H

4HB $\rightarrow$ 8H: In Figure 4.47, each sender divides its data into four pieces and sends two of the pieces to different receivers. E.g. P0 sends its first piece to P0 and third piece to P2. P1 sends its second piece to P1 and fourth piece to P3. P2 sends its first piece to P4 and third piece to P6. P3 sends its second piece to P5 and fourth piece to P7. (P0 and P1 may copy data instead of sending). Each receiver receives one piece of data.

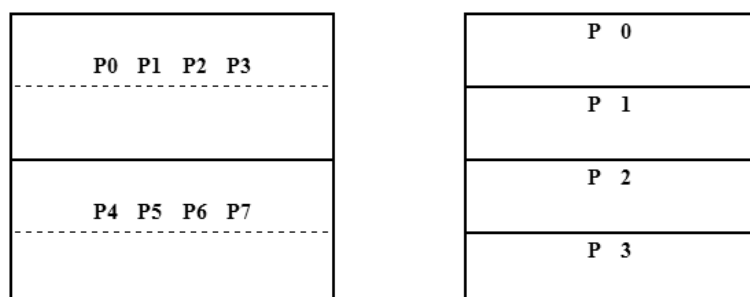


Figure 4.48. Conversion from 8HB to 4H

8HB $\rightarrow$ 4H: In Figure 4.48, each sender divides its data into two pieces and sends one of them. E.g. P0 sends its first piece to P0. P1 sends its second piece to P1. P4 sends its first

piece to P2. P5 sends its second piece to P3. (P0 and P1 may copy data instead of sending). Each receiver receives one piece of data.

4.4.20. Horizontal Block to Vertical Conversion for Matrices (mHB2V)

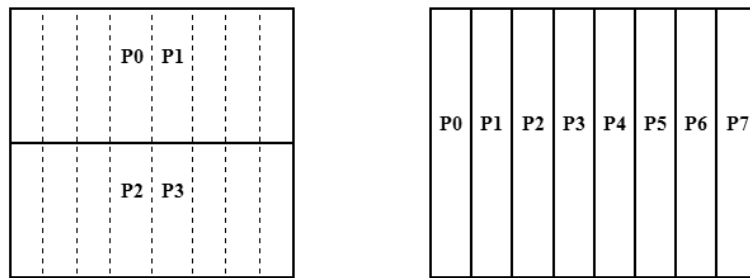


Figure 4.49. Conversion from 4HB to 8V

4HB $\rightarrow$ 8V: In Figure 4.49, each sender divides its data into eight pieces and sends four of them to different receivers. E.g. P0 and P2 send their first piece to P0, third piece to P2, fifth piece to P4 and seventh piece to P6. P1 and P3 send their second piece to P1, fourth piece to P3, sixth piece to P5 and eighth piece to P7. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives two pieces of data. The received pieces are ordered by the senders' ID.

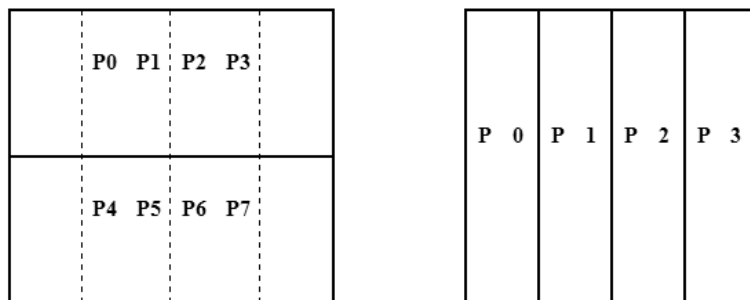


Figure 4.50. Conversion from 8HB to 4V

8HB $\rightarrow$ 4V: In Figure 4.50, each sender divides its data into four pieces and sends one piece to a receiver. E.g. P0 and P4 send their first piece to P0. P1 and P5 send their second piece to P1. P2 and P6 send their third piece to P2. P3 and P7 send their fourth piece to P3. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives two pieces of data. The received pieces are ordered by the senders' ID.

4.4.21. Horizontal Block to Rectangular Conversion for Matrices (mHB2R)

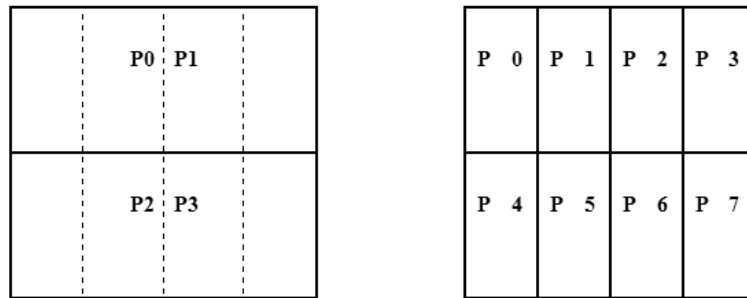


Figure 4.51. Conversion from 4HB to 8R

4HB $\rightarrow$ 8R: In Figure 4.51, each sender divides its data into four pieces and sends two of them to receivers. E.g. P0 sends its first piece to P0 and third piece to P2. P1 sends its second piece to P1 and fourth piece to P3. P2 sends its first piece to P4 and third piece to P6. P3 sends its second piece to P5 and fourth piece to P7. (P0 and P1 may copy data instead of sending). Each receiver receives one piece of data.

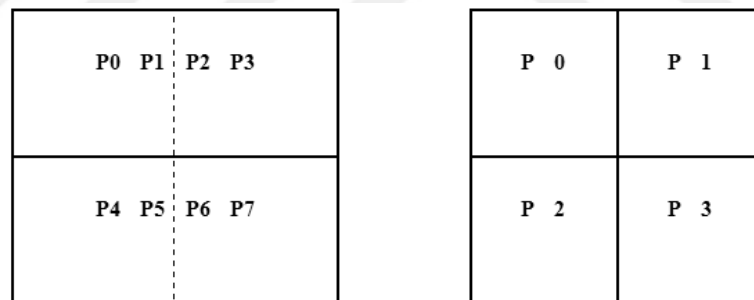


Figure 4.52. Conversion from 8HB to 4R

8HB $\rightarrow$ 4R: In Figure 4.52, each sender divides its data into two pieces and sends one piece to a receiver. E.g. P0 sends its first piece to P0. P1 sends its second piece to P1. P4 sends its first piece to P2. P5 sends its second piece to P3. (P0 and P1 may copy data instead of sending). Each receiver receives one piece of data.

4.4.22. Horizontal Block to Horizontal Block Conversion for Matrices (mHB2HB)

4HB $\rightarrow$ 8HB: In Figure 4.53, senders do not divide their data. They send their data to more than one receiver. E.g. P0 sends its data to P0 and P2. P1 sends its data to P1 and P3.

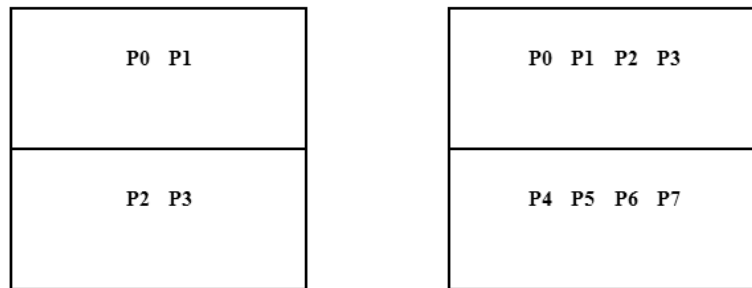


Figure 4.53. Conversion from 4HB to 8HB

P2 sends its data to P4 and P6. P3 sends its data to P5 and P7. (P0 and P1 may copy data instead of sending). Each receiver receives one piece of data.



Figure 4.54. Conversion from 8HB to 4HB

8HB $\rightarrow$ 4HB: In Figure 4.54, senders do not divide their data. E.g. P0 sends its data to P0. P1 sends its data to P1. P4 sends its data to P2. P5 sends its data to P3. (P0 and P1 may copy data instead of sending). Each receiver receives one piece of data.

4.4.23. Horizontal Block to Vertical Block Conversion for Matrices (mHB2VB)

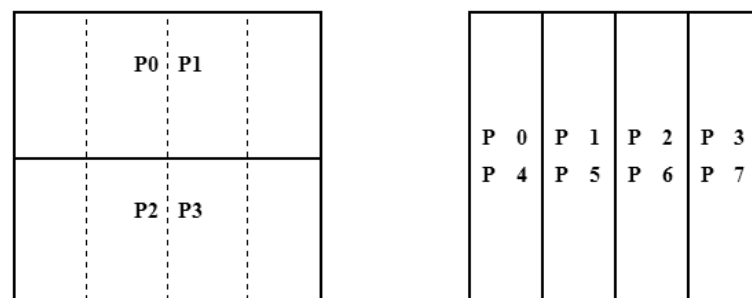


Figure 4.55. Conversion from 4HB to 8VB

4HB $\rightarrow$ 8VB: In Figure 4.55, each sender divides its data into four pieces and sends

two of them to more than one receiver. E.g. P0 and P2 send their first piece to P0 and P4, third piece to P2 and P6. P1 and P3 send their second piece to P1 and P5, fourth piece to P3 and P7. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives two pieces of data. The received pieces are ordered by the senders' ID.

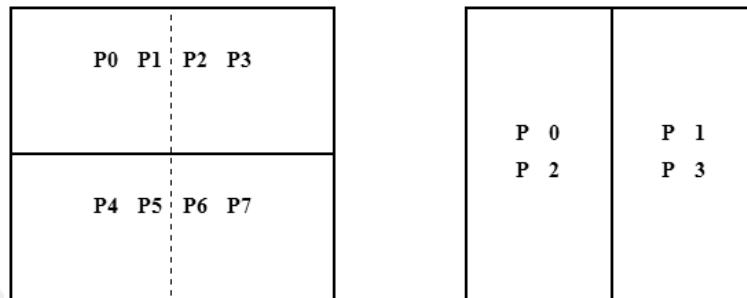


Figure 4.56. Conversion from 8HB to 4VB

8HB $\rightarrow$ 4VB: In Figure 4.56, each sender divides its data into two pieces and sends one of them. E.g. P0 and P4 send their first piece to P0. P1 and P5 send their second piece to P1. P2 and P6 send their first piece to P2. P3 and P7 send their second piece to P3. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives two pieces of data. The received pieces are ordered by the senders' ID.

4.4.24. Horizontal Block to Replicated Conversion for Matrices (mHB2REP)

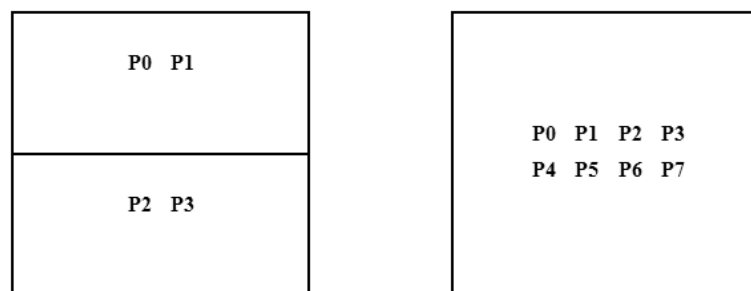


Figure 4.57. Conversion from 4HB to 8REP

4HB $\rightarrow$ 8REP: In Figure 4.57, senders do not divide their data. They send it to more than one receiver. E.g. P0 and P2 send their data to P0, P2, P4 and P6. P1 and P3 send their data to P1, P3, P5 and P7. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives two pieces of data. The received pieces are ordered by the senders' ID.

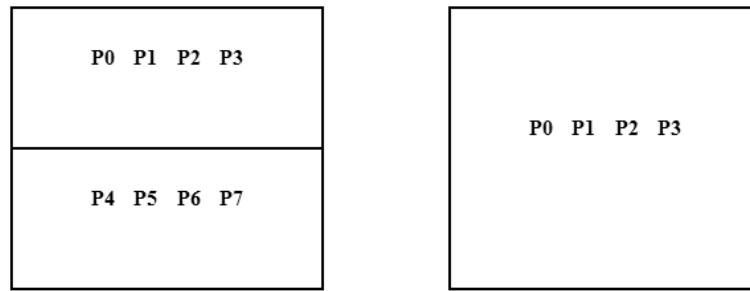


Figure 4.58. Conversion from 8HB to 4REP

8HB $\rightarrow$ 4REP: In Figure 4.58, senders do not divide their data. E.g. P0 and P4 send their data to P0. P1 and P5 send their data to P1. P2 and P6 send their data to P2. P3 and P7 send their data to P3. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives two pieces of data. The received pieces are ordered by the senders' ID.

4.4.25. Vertical Block to Horizontal Conversion for Matrices (mVB2H)

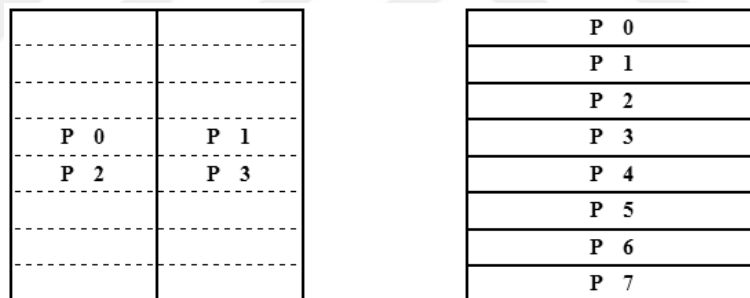


Figure 4.59. Conversion from 4VB to 8H

4VB $\rightarrow$ 8H: In Figure 4.59, each sender divides its data into eight pieces and sends four of the pieces to different receivers. E.g. P0 and P3 send their first piece to P0, fourth piece to P3, fifth piece to P4 and eighth piece to P7. P1 and P2 send their second piece to P1, third piece to P2, sixth piece to P5 and seventh piece to P6. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives two pieces of data. The received pieces are ordered by the senders' position.

8VB $\rightarrow$ 4H: In Figure 4.60, the senders divide their data into four pieces. E.g. P0 sends its first piece to P0. P1 sends its second piece to P1. P2 sends its third piece to P2. P3 sends its fourth piece to P3. (P0, P1, P2 and P3 may copy data instead of sending). P5, P6

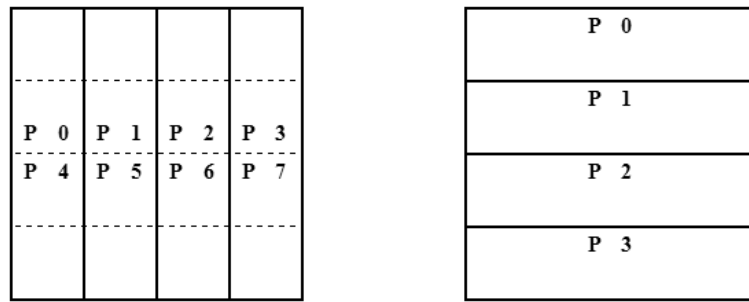


Figure 4.60. Conversion from 8VB to 4H

and P7 send their first piece to P0. P4, P6 and P7 send their second piece to P1. P4, P5 and P7 send their third piece to P2. P4, P5 and P6 send their fourth piece to P3. Each receiver receives four pieces of data. The received pieces are ordered by the senders' column.

4.4.26. Vertical Block to Vertical Conversion for Matrices (mVB2V)

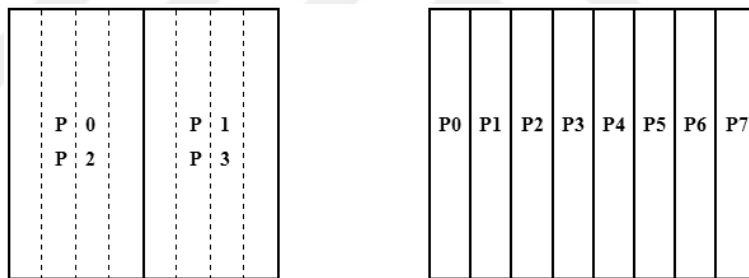


Figure 4.61. Conversion from 4VB to 8V

4VB $\rightarrow$ 8V: In Figure 4.61, each sender divides its data into four pieces and sends two of them to different receivers. E.g. P0 sends its first piece to P0, second piece to P1. P1 sends its first piece to P4, second piece to P5. P2 sends its third piece to P2, fourth piece to P3. P3 sends its third piece to P6, fourth piece to P7. (P0 and P2 may copy data instead of sending). Each receiver receives one piece of data.

8VB $\rightarrow$ 4V: In Figure 4.62, the senders do not divide their data. E.g. P0 sends its data to P0. P1 sends its data to P1. P2 sends its data to P2. P3 sends its data to P3. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives one piece of data.

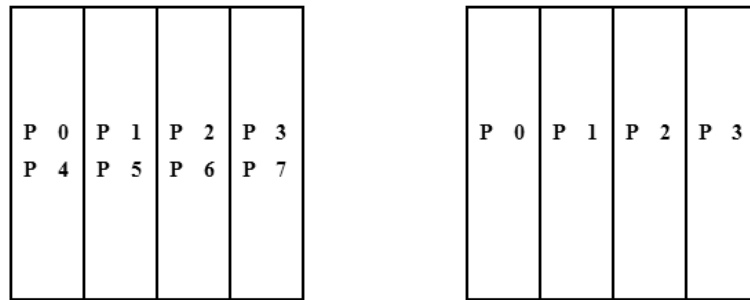


Figure 4.62. Conversion from 8VB to 4V

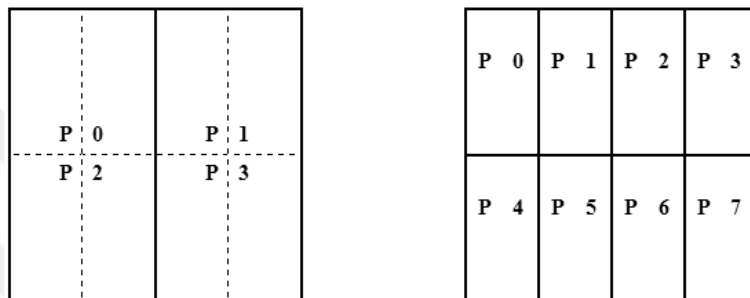


Figure 4.63. Conversion from 4VB to 8R

4.4.27. Vertical Block to Rectangular Conversion for Matrices (mVB2R)

4VB $\rightarrow$ 8R: In Figure 4.63, each sender divides its data into four pieces and sends two of them to receivers. E.g. P0 sends its first column's top piece to P0 and bottom piece to P4. P1 sends its first column's top piece to P2 and bottom piece to P6. P2 sends its second column's top piece to P1 and bottom piece to P5. P3 sends its second column's top piece to P3 and bottom piece to P7. (P0 and P3 may copy data instead of sending). Each receiver receives one piece of data.

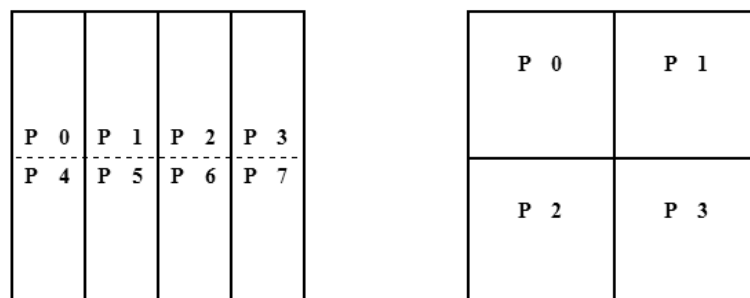


Figure 4.64. Conversion from 8VB to 4R

8VB $\rightarrow$ 4R: In Figure 4.64, each sender divides its data into two pieces and sends one piece to a receiver. E.g. P0 sends its top piece to P0. P1 sends its bottom piece to P2. P2 sends its top piece to P1. P3 sends its bottom piece to P3. P4 sends its bottom piece to P2. P5 sends its top piece to P0. P6 sends its bottom piece to P3. P7 sends its top piece to P1. (P0 and P3 may copy data instead of sending). Each receiver receives two pieces of data. The received pieces are ordered by the senders' column.

4.4.28. Vertical Block to Horizontal Block Conversion for Matrices (mVB2HB)

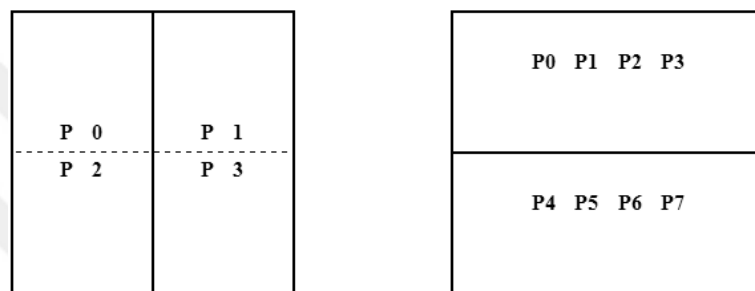


Figure 4.65. Conversion from 4VB to 8HB

4VB $\rightarrow$ 8HB: In Figure 4.65, each sender divides its data into two pieces and sends each piece to more than one receiver. E.g. P0 and P3 send their top piece to P0 and P3, bottom piece to P4 and P7. P1 and P2 send their top piece to P1 and P2, bottom piece to P5 and P6. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives two pieces of data. The received pieces are ordered by the senders' column.

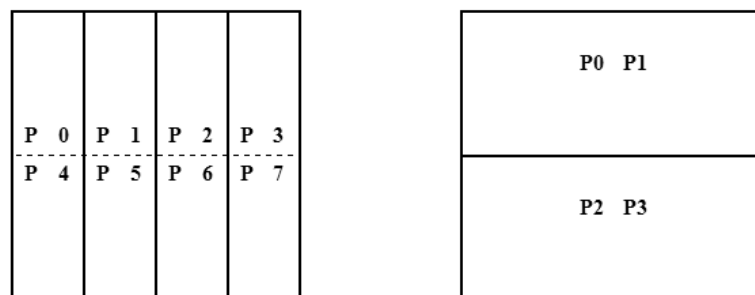


Figure 4.66. Conversion from 8VB to 4HB

8VB $\rightarrow$ 4HB: In Figure 4.66, the senders divide their data into two pieces. E.g. P0 sends its top piece to P0. P1 sends its top piece to P1. P2 sends its bottom piece to P2. P3 sends its bottom piece to P3. (P0, P1, P2 and P3 may copy data instead of sending). P5, P6

and P7 send their top piece to P0. P4, P6 and P7 send their top piece to P1. P4, P5 and P7 send their bottom piece to P2. P4, P5 and P6 send their bottom piece to P3. Each receiver receives four pieces of data. The received pieces are ordered by the senders' column.

4.4.29. Vertical Block to Vertical Block Conversion for Matrices (mVB2VB)

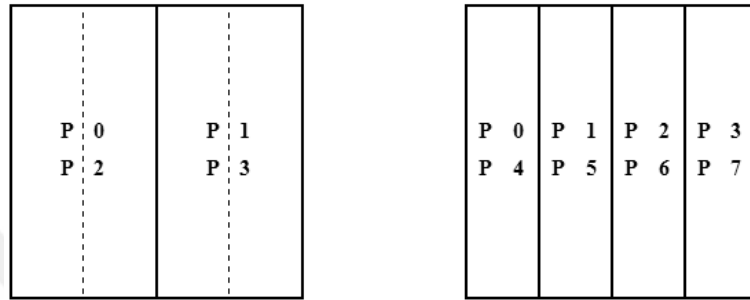


Figure 4.67. Conversion from 4VB to 8VB

4VB $\rightarrow$ 8VB: In Figure 4.67, each sender divides its data into two pieces and sends one of them to more than one receiver. E.g. P0 sends its first column to P0 and P4. P1 sends its first column to P2 and P6. P2 sends its second column to P1 and P5. P3 sends its second column to P3 and P7. (P0 and P3 may copy data instead of sending). Each receiver receives one piece of data.

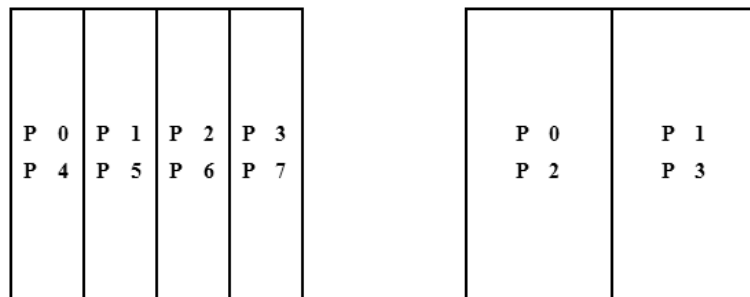


Figure 4.68. Conversion from 8VB to 4VB

8VB $\rightarrow$ 4VB: In Figure 4.68, the senders do not divide their data. E.g. P0 sends its data to P0. P1 sends its data to P2. P2 sends its data to P1. P3 sends its data to P3. P4 sends its data to P2. P5 sends its data to P0. P6 sends its data to P3. P7 sends its data to P1. (P0 and P3 may copy data instead of sending). Each receiver receives two pieces of data. The received pieces are ordered by the senders' column.

4.4.30. Vertical Block to Replicated Conversion for Matrices (mVB2REP)

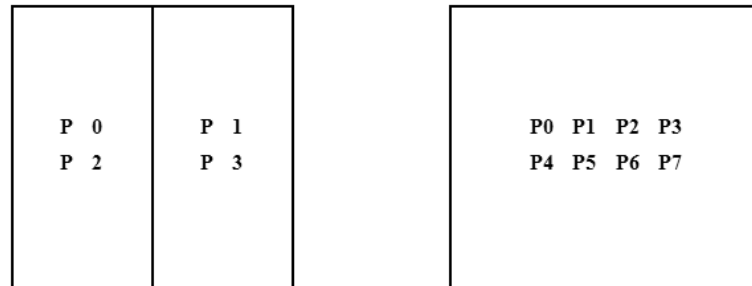


Figure 4.69. Conversion from 4VB to 8REP

4VB $\rightarrow$ 8REP: In Figure 4.69, senders do not divide their data. They send it to more than one receiver. E.g. P0 and P3 send their data to P0, P3, P4 and P7. P1 and P2 send their data to P1, P2, P5 and P6. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives two pieces of data. The received pieces are ordered by the senders' column.

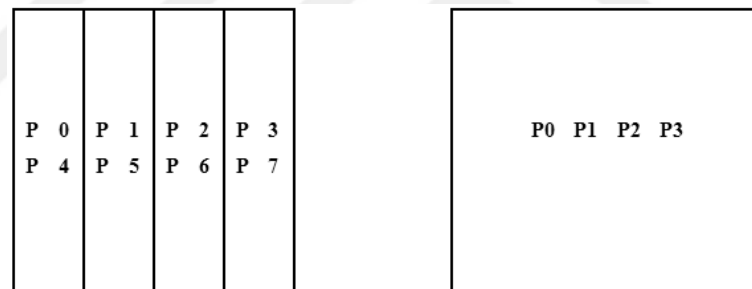


Figure 4.70. Conversion from 8VB to 4REP

8VB $\rightarrow$ 4REP: In Figure 4.70, senders do not divide their data. They send it to more than one receiver. E.g. P0 sends its data to P0. P1 sends its data to P1. P2 sends its data to P2. P3 sends its data to P3. (P0, P1, P2 and P3 may copy data instead of sending). P5, P6 and P7 send their data to P0. P4, P6 and P7 send their data to P1. P4, P5 and P7 send their data to P2. P4, P5 and P6 send their data to P3. Each receiver receives four pieces of data. The received pieces are ordered by the senders' column.

4.4.31. Replicated to Horizontal Conversion for Matrices (mREP2H)

4REP $\rightarrow$ 8H: In Figure 4.71, each sender divides its data into eight pieces and sends two of the pieces to different receivers. E.g. P0 sends its first piece to P0 and fifth piece to

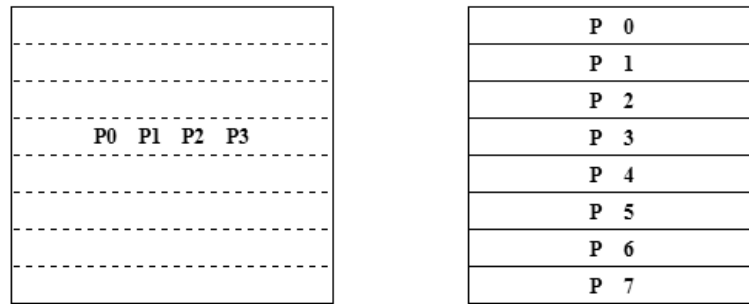


Figure 4.71. Conversion from 4REP to 8H

P4. P1 sends its second piece to P1 and sixth piece to P5. P2 sends its third piece to P2 and seventh piece to P6. P3 sends its fourth piece to P3 and eighth piece to P7. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives one piece of data.

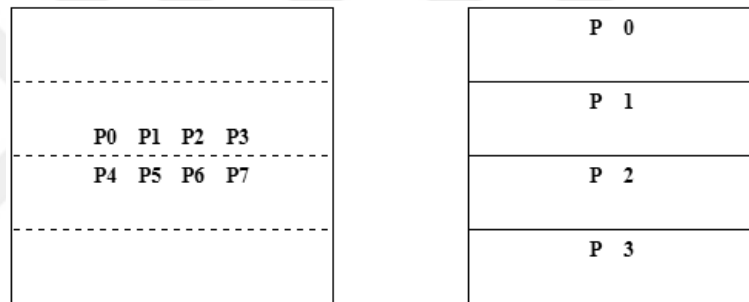


Figure 4.72. Conversion from 8REP to 4H

8REP $\rightarrow$ 4H: In Figure 4.72, the senders divide their data into four pieces and send one piece to a receiver. E.g. P0 sends its first piece to P0. P1 sends its second piece to P1. P2 sends its third piece to P2. P3 sends its fourth piece to P3. P4, P5, P6 and P7 do not send data. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives one piece of data.

4.4.32. Replicated to Vertical Conversion for Matrices (mREP2V)

4REP $\rightarrow$ 8V: In Figure 4.73, each sender divides its data into eight pieces and sends two of the pieces to different receivers. E.g. P0 sends its first piece to P0 and fifth piece to P4. P1 sends its second piece to P1 and sixth piece to P5. P2 sends its third piece to P2 and seventh piece to P6. P3 sends its fourth piece to P3 and eighth piece to P7. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives one piece of data.

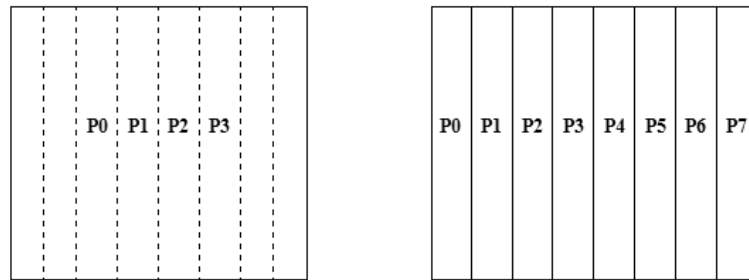


Figure 4.73. Conversion from 4REP to 8V

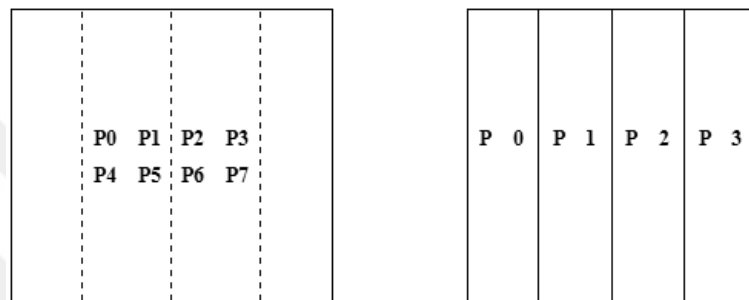


Figure 4.74. Conversion from 8REP to 4V

8REP $\rightarrow$ 4V: In Figure 4.74, the senders divide their data into four pieces and send one piece to a receiver. E.g. P0 sends its first piece to P0. P1 sends its second piece to P1. P2 sends its third piece to P2. P3 sends its fourth piece to P3. P4, P5, P6 and P7 do not send data. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives one piece of data.

4.4.33. Replicated to Rectangular Conversion for Matrices (mREP2R)

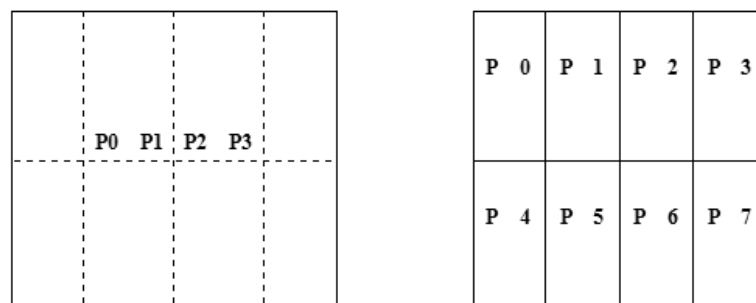


Figure 4.75. Conversion from 4REP to 8R

4REP $\rightarrow$ 8R: In Figure 4.75, each sender divides its data into eight pieces and sends

two of the pieces to different receivers. E.g. P0 sends its first column's top piece to P0 and bottom piece to P4. P1 sends its second column's top piece to P1 and bottom piece to P5. P2 sends its third column's top piece to P2 and bottom piece to P6. P3 sends its fourth column's top piece to P3 and bottom piece to P7. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives one piece of data.

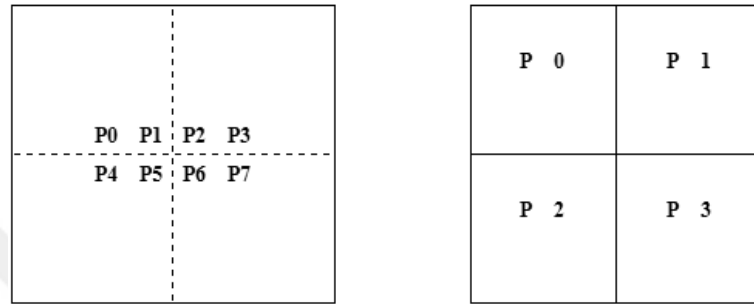


Figure 4.76. Conversion from 8REP to 4R

8REP $\rightarrow$ 4R: In Figure 4.76, the senders divide their data into four pieces and send one piece to a receiver. E.g. P0 sends its first column's top piece to P0. P1 sends its second column's top piece to P1. P2 sends its first column's bottom piece to P2. P3 sends its second column's bottom piece to P3. P4, P5, P6 and P7 do not send data. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives one piece of data.

4.4.34. Replicated to Horizontal Block Conversion for Matrices (mREP2HB)

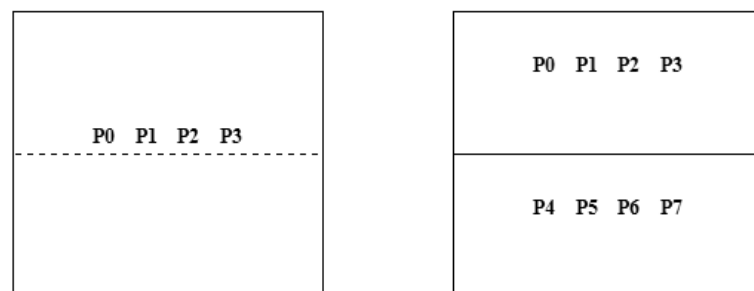


Figure 4.77. Conversion from 4REP to 8HB

4REP $\rightarrow$ 8HB: In Figure 4.77, each sender divides its data into two pieces and sends them to different receivers. E.g. P0 sends its top piece to P0 and bottom piece to P4. P1 sends its top piece to P1 and bottom piece to P5. P2 sends its top piece to P2 and bottom

piece to P6. P3 sends its top piece to P3 and bottom piece to P7. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives one piece of data.

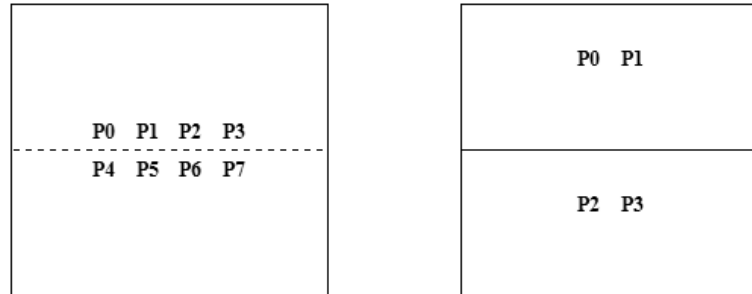


Figure 4.78. Conversion from 8REP to 4HB

8REP $\rightarrow$ 4HB: In Figure 4.78, the senders divide their data into two pieces and send one piece to a receiver. E.g. P0 sends its top piece to P0. P1 sends its top piece to P1. P2 sends its bottom piece to P2. P3 sends its bottom piece to P3. P4, P5, P6 and P7 do not send data. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives one piece of data.

4.4.35. Replicated to Vertical Block Conversion for Matrices (mREP2VB)

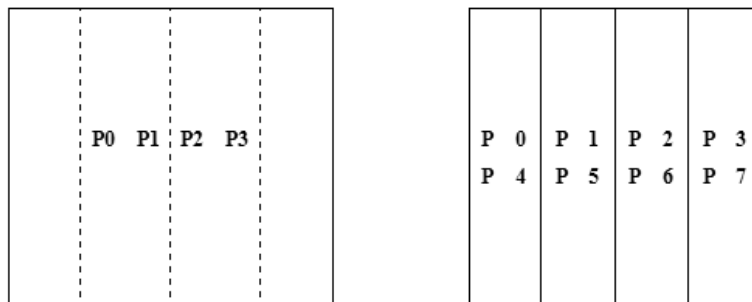


Figure 4.79. Conversion from 4REP to 8VB

4REP $\rightarrow$ 8VB: In Figure 4.79, each sender divides its data into four pieces and sends one piece to more than one receiver. E.g. P0 sends its first piece to P0 and P4. P1 sends its second piece to P1 and P5. P2 sends its third piece to P2 and P6. P3 sends its fourth piece to P3 and P7. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives one piece of data.

8REP $\rightarrow$ 4VB: In Figure 4.80, the senders divide their data into two pieces and send

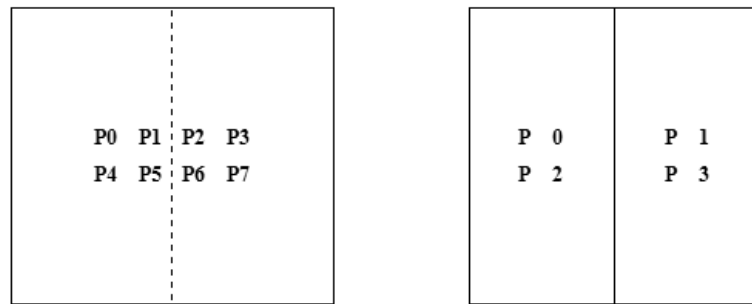


Figure 4.80. Conversion from 8REP to 4VB

one piece to a receiver. E.g. P0 sends its first piece to P0. P1 sends its second piece to P1. P2 sends its first piece to P2. P3 sends its second piece to P3. P4, P5, P6 and P7 do not send data. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives one piece of data.

4.4.36. Replicated to Replicated Conversion for Matrices (mREP2REP)



Figure 4.81. Conversion from 4REP to 8REP

4REP $\rightarrow$ 8REP: In Figure 4.81, senders do not divide their data and send it to more than one receiver. E.g. P0 sends its data to P0 and P4. P1 sends its data to P1 and P5. P2 sends its data to P2 and P6. P3 sends its data to P3 and P7. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives one piece of data.

8REP $\rightarrow$ 4REP: In Figure 4.82, senders do not divide their data. E.g. P0 sends its data to P0. P1 sends its data to P1. P2 sends its data to P2. P3 sends its data to P3. P4, P5, P6 and P7 do not send data. (P0, P1, P2 and P3 may copy data instead of sending). Each receiver receives one piece of data.

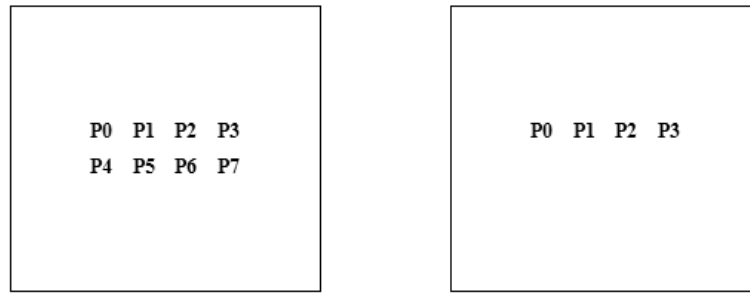


Figure 4.82. Conversion from 8REP to 4REP

4.5. Vector Partitioning

In this thesis, 3 types of partitioning is analyzed for vectors: Horizontal (H), Horizontal Block (HB) and Replicated (REP). Figures 4.83-4.85 show how the data are partitioned for 4, 8 and 16 processors, for each decomposition type. Figure 4.83 shows Horizontal partitioning of a vector on 4, 8 and 16 processors. If the size of the vector is N , on 4 processors with Horizontal partitioning, each processor has $N/4$ elements. Figure 4.84 shows Horizontal Block partitioning of a vector on 4, 8 and 16 processors. If the size of the vector is N , on 4 processors with Horizontal Block partitioning, each processor has $N/2$ elements. Figure 4.85 shows Replicated partitioning of a vector on 4, 8 and 16 processors. If the size of the vector is N , on 4 processors with Replicated partitioning, each processor has N elements.

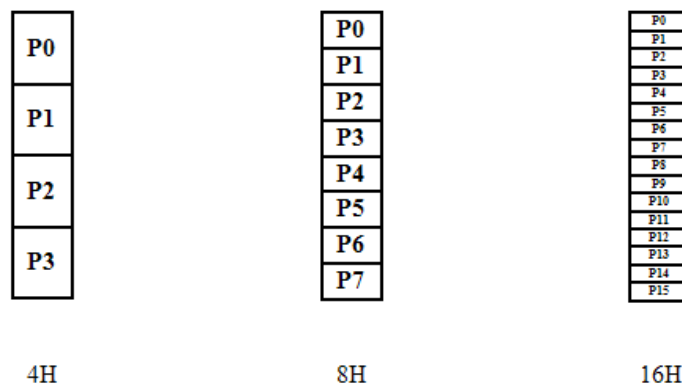


Figure 4.83. Horizontal (H) decomposition of vectors for 4, 8 and 16 processors

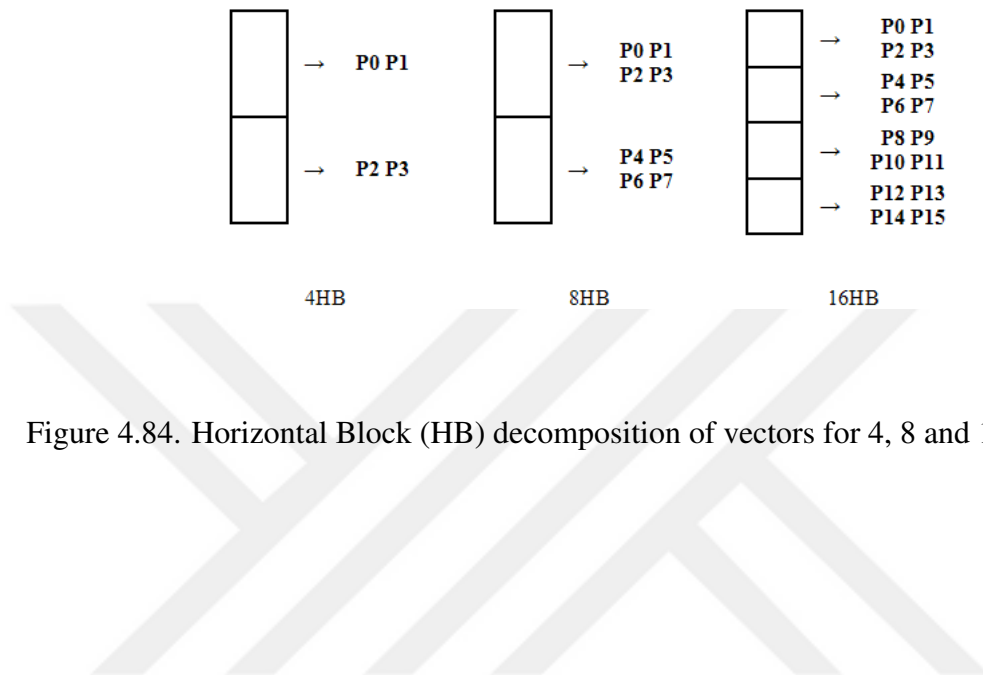


Figure 4.84. Horizontal Block (HB) decomposition of vectors for 4, 8 and 16 processors

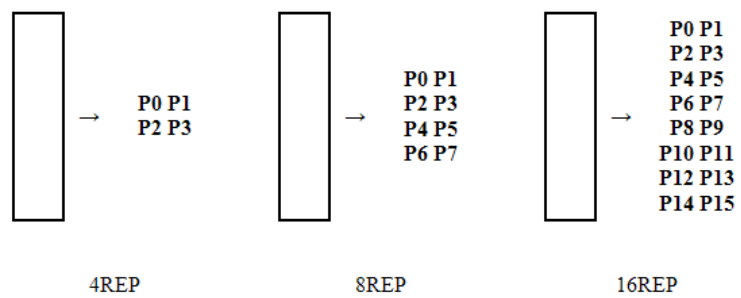


Figure 4.85. Replicated (REP) decomposition of vectors for 4, 8 and 16 processors

4.6. Conversion Functions for Vectors

The Conversion Functions for vectors are the one-dimensional versions of the Conversion Functions for matrices. These functions convert the data of a vector among Horizontal, Horizontal Block and Replicated data decomposition types.



5. PARALLEL PROGRAM OPTIMIZER

Darwin's theory about evolution inspired Genetic Algorithms. The Genetic Algorithms solve difficult problems by evolving the solution. They use some techniques, which are based on genetics and evolution. Memetic Algorithms are the combination of Genetic Algorithms and hill climbing with local search methods (explained in Section 8.2 in detail).

As the Parallel Program Optimizer, *GAlib* (*Genetic Algorithms Library*) is used, which was implemented by Assist. Prof. Dr. Ender Özcan at Yeditepe University. The problem dependent functions of the library are modified. In this problem, the chromosome length is equal to the number of levels in the user's program. The *gene* carries the following data:

- the group of processors
- the number of processors (from the selected group)
- the selected decomposition for the arrays of the level
- the communication time before the level
- the computation time of the level

The *communication time* and the *computation time* are stored in the gene, in order to make the calculations easy. The fitness for a given candidate solution is the total execution time of the parallel program. The Memetic Algorithm is explained in Figure 5.1 and in [18], in detail. The Memetic Algorithm parameters used in this thesis are presented below.

A population is generated randomly. Each individual (chromosome) is a candidate solution of the problem. As mentioned in [18], hill climbing operators are local search algorithms that are used to improve a given candidate solution. In this thesis, to improve the solutions, the chosen hill climbing method is applied after initialization and mutation. The details of hill climbing methods used in the thesis are given in Section 8.2.

The crossover operator chooses two mates (parents) and a random crossover point. The alleles before this point are copied from the first parent, and the rest is copied from the second

1. Generate initial population with N individuals
Apply hill-climbing on each individual in the initial population
2. Evolve until the termination criteria are satisfied
 - a. Evaluate each individual in the population
 - b. Generate K offspring (new individuals):
 - Select mates (individuals) from the population
 - Apply crossover on the mates generating new individuals
 - c. Apply (smart) mutation on the new individuals
 - d. Apply hill-climbing on the new individuals
 - e. Replacement: Obtain a new population from the previous population and new individuals (offsprings); N individuals from $(N+K)$ individuals

Figure 5.1. Pseudocode of a Memetic Algorithm

parent to the first offspring. For the second offspring, the alleles before the crossover point are copied from the second parent, and the rest is copied from the first parent. The mates to be used by the crossover operator are selected according to tournament mate selection method. In this selection, the mate with the best fitness is chosen from a subset of randomly selected individuals [18]. The size of the subset is called *tour size*.

One-point-crossover is used and the crossover rate is set to 1.0. The crossover rate (probability) is compared with a random number. If the random number is greater than crossover rate, the crossover operator is not used. Therefore, the offsprings become identical copies of the mates [18].

Mutation is applied to the new individuals. The traditional mutation operator scans each locus [18] and assigns a random gene with a given mutation probability, which is $1 / \text{chromosome length}$. The population size is set to 30 and the offspring count equals the population size minus 2. After the mutation, hill climbing is performed again. As a replacement strategy, Transgenerational Memetic Algorithm (TGMA) is used. In TGMA, the best of the offsprings and the previous population form the next generation [18].

Our aim is to minimize the fitness. The fitness indicates the total execution time of the parallel program. The algorithm is repeated until the termination criteria is satisfied.

6. ARRAY INFORMATION TABLE GENERATOR

The User Program Parser gathers information about the user's sequential program and writes them into output files. The Array Information Table Generator program reads the information about each level and the possible decompositions. Each individual generated by Parallel Program Optimizer is sent to Array Information Table Generator. For each level, the individuals carry the processor group ID and the number of processors on which the level will execute, and the selected possible decomposition. The Array Information Table Generator has two tables. According to the input chromosome, detailed information about the matrices and vectors of the level are written to these tables. For each chromosome generated by Parallel Program Optimizer, the contents of tables are changed. For all RHS arrays in each level, the communication time is calculated and returned to Parallel Program Optimizer to be used in calculating the fitness of the chromosome.

During the parallelization of the user's sequential program, between each level some conversion lines will be placed by the Parallelizer. These lines will be used to call the Conversion Functions with correct parameters. When the best chromosome is determined by the Parallel Program Optimizer, it is sent to Array Information Table Generator. For all RHS arrays in each level, this time instead of calculating the communication time, the conversion lines are generated. These lines are written to a file to be used by the Parallelizer.

6.1. Array Information Tables

As mentioned above, the generator has two tables: one for matrices and one for vectors. Both tables store the name and size of the arrays. The difference is the data decomposition types they include. Vectors are one dimensional arrays and they may have four types of data decomposition: Horizontal (H), Horizontal Block (HB), Replicated (REP) and None (NONE). Matrices are two dimensional arrays and they may have seven types of decomposition: Horizontal (H), Vertical (V), Rectangular (R), Horizontal Block (HB), Vertical Block (VB), Replicated (REP) and None (NONE). The tables store the following elements for each decomposition type:

- the last *level*
 - the last *group of processors*
 - the last *number of processors* in the group, and
 - the last *R/L position* of the array, where it was used in this data decomposition type.
- The *R/L position* gets values between 0 and 3. Each value has a different meaning, explained in the Section 6.1.1.

In these tables, the previous data about the arrays are copied to the current level and then updated. This approach allows us to find the decomposed array in previous levels easily. All necessary data (name, size, **for each decomposition type:** level, processor group ID, number of processors, *R/L position*) of the arrays are stored and updated in the tables. Just by looking at the data, which can be reached by the previous level's index, a quick search can be done to find an appropriate decomposition of the array, before converting it for the current level.

6.1.1. The R/L position

The *R/L position* indicates the position of the array according to the equal sign. *R* stands for *right* and *L* stands for *left*. Its value may vary between 0 and 3. An array may be on RHS, LHS or both sides of the equal sign. If the array is used more than once in a level and on both sides, the *R/L position* also gives an idea about the decomposition of the array. These values are very important, because they affect the update process of the tables.

ONLY RIGHT (R/L = 1)

- The array is only on the right-hand-side of the equal sign.
- Before the level where this array is used (in the user's program), a conversion line must be written for this array.
- This decomposition of the array may be used in next levels, since the data of the array are not changed.

ON-BOTH-SIDE-SAME (R/L = 2)

- The array is on both sides of the equal sign. On both sides, the array has the same decomposition type.
- Since the array is on right-hand-side, before the level where this array is used (in the user's program), a conversion line must be written for this array.
- Since the array is on left-hand-side (the data are changed), the previous decomposition data related with this array are deleted (previous decompositions have the old data).
- This decomposition of the array may be used in next levels, because the data of the array are updated.

ONLY LEFT or ON-BOTH-SIDE-DIFFERENT-LEFT (R/L = 3)

- The array is only on the left-hand-side of the equal sign OR the array is on both sides of the equal sign with different decomposition types (this is the one on LHS).
- Since the array is on left-hand-side, before the level where this array is used (in the user's program), a conversion line is NOT written for this array.
- Since the data of the array are changed, the previous decomposition data related with this array are deleted (previous decompositions have the old data).
- This decomposition of the array may be used in next levels, because the data of the array are updated.

ON-BOTH-SIDE-DIFFERENT-RIGHT (R/L = 0)

- The array is on both sides of the equal sign with different decomposition types (this is the one on RHS).
- Since the array is on right-hand-side, before the level where this array is used (in the user's program), a conversion line must be written for this array.
- This decomposition of the array CANNOT BE used in the next levels, because the data of the array are changed (this decomposition has the old data).

6.2. Initializing the Tables

The number of levels, the number of matrices, the number of vectors, the name and size of all arrays in the user's program are read from User Program Parser's output file. Then, each possible decomposition of all computation levels is read from the same file. The decompositions include the name, decomposition type and *R/L position* of the arrays used in the level.

6.3. Preparing the Tables for the New Individual

The Parallel Program Optimizer generates many chromosomes and sends each of them to Array Information Table Generator. Before calculating the communication time of each chromosome or before writing the conversion lines for the best chromosome, the tables must be updated according to the new individual. The input chromosome carries *processor group ID*, *number of processors* and the *selected possible decomposition* for each level. For all computation levels, the following procedure is applied:

- For all matrices and all vectors in the tables, the value of the *level*, *group of processors*, *number of processors* and *R/L position* of the array for all data decomposition types are copied from the previous level to the current level.
- From input chromosome, the *group of processors*, *number of processors* and *selected decomposition* for the current level are read.
- If the input chromosome is the best chromosome, the selected decomposition for this level is appended to an output file to be used by the Parallelizer.
- For all arrays in the selected decomposition, the update function is called with the following parameters: *R/L position* and *name* of the array, *decomposition type* offered for the array, *current level*, the *group* and the *number of processors* which the array will be sent to.

6.4. Updating the Tables

If *R/L position* equals 2 or 3 (i.e. the array is on the left-hand-side of equal sign), then in the table, the value of the *level*, *group of processors*, *number of processors* and *R/L position* of the array for all data decomposition types are deleted. The reason for this deletion can be explained as follows: Since the array is on the left-hand-side of equal sign, the content of the array will be changed. The previous decompositions will have the old values of the array. To avoid wrong calculations using old values, the "copied" decomposition data of previous level for the array are deleted.

The *decomposition type* may be H, V, R, HB, VB, REP or NONE for a matrix and H, HB, REP or NONE for a vector. According to the selected decomposition type of the array, the *level*, *group of processors*, *number of processors* and *R/L position* of the array are set with the incoming parameters.

6.5. Reading the Tables to Calculate the Communication Time

In the parallelized program, all RHS arrays of each level must be sent to the computing processors, before the computation level begins. Therefore, the communication time during the transfer of these arrays must be calculated. For each chromosome sent by the fitness function of Parallel Program Optimizer, the tables are updated and then they are examined to select an appropriate previous decomposition for the RHS arrays. (If the array was not decomposed before, the time to scatter the array is calculated.) For each computation level, for all RHS arrays, the following procedure is applied:

- According to the next decomposition type (in this level on the RHS of equal sign), the program searches for a previous decomposition type of this array in previous levels, where its *R/L position* is not 0 (0 means the decomposed data are old and not valid now). The search is done by following a search order according to the next decomposition type.
- When the previous decomposition type is determined, the communication time of transferring the array is calculated by a calculation function with the following param-

eters: the *size* of the array, the *previous group*, the *number of processors* in the previous group, the *next group*, the *number of processors* in the next group and the *conversion type* (from the previous decomposition type to the next decomposition type). If there is no previous decomposition type, the communication time of scattering this array is calculated by calculation function with the following parameters: the *size* of the array, the *next group*, the *number of processors* in the next group and the *conversion type* (scatter to next decomposition type).

- The result returned by the calculation function is the duration of converting this array. It is added to the communication time of this level.

When the communication time of each computation level is calculated, the results are returned to the fitness function of the Parallel Program Optimizer.

6.6. Calculation of the Communication Time

The communication time of converting an array from one decomposition type to another (and also transferring it from p processors in the *previous group* to q processors in the *next group*) is calculated as follows: Using the Conversion Functions as benchmarks, the slope and intercept values of total array size for all possibilities (for all *previous* and *next group*, p and q values, Conversion Functions) were determined before. The slope and intercept values are read from an input file once and stored in an array. According to the incoming parameters, the necessary slope and intercept values are searched in the array. The duration to convert the array is calculated and returned.

6.7. Reading the Tables to Write the Conversion Function Call Lines for the Best Chromosome

In the parallelized program, there must be a Conversion Function call line for all RHS arrays of each level, before the computation level begins. A Conversion Function call includes the *conversion type* parameter. This parameter indicates from which decomposition type to which type the array will be converted. To determine the previous decomposition type, the algorithm explained in Section 6.5 is used. The next decomposition type is known:

it is the decomposition type of the array in the following computation level.

After determining the previous decomposition type (and also the *conversion type*), the conversion line is generated with the correct parameters and it is written to an output file. A conversion line has the following parameters: *original ID* of the calling processor, the *previous processor group*, the *number of processors* in the previous processor group, the *next processor group*, the *number of processors* in the next processor group, the *conversion type* (from previous decomposition type to next decomposition type), the *size* of the array, the *name* of the array and the *name of the temporary array* to be used during conversion.

If there is no previous decomposition type (i.e. if the array was not decomposed before), the conversion line to scatter the array is written to the file with the following parameters: *original ID* of the calling processor, the *next processor group*, the *number of processors* in the next processor group, the *conversion type* (scatter to next decomposition type), the *size* of the array, the *name* of the array.

Examples of *conversion lines* are given in Section 7.4. These conversion lines will be placed by the Parallelizer in the user's parallelized program before each computation level, in order to convert the RHS arrays from their previous decomposition to new decomposition.

At the end of the program, the user may want to print the contents of the output arrays into files. During the execution of the parallel program, these arrays will be decomposed and each piece will be on a different processor. The arrays to be printed must be gathered by the root processor of Group 0. Therefore, the *gather lines* are prepared for all matrices and vectors. While the Parallelizer prepares the parallelized program, if there is a level where an array is printed to a file, the Parallelizer will use the gather line of the array to be printed.

For all matrices and vectors, the Array Information Table Generator searches for a previous decomposition type of this array in previous levels, where its *R/L position* is not 0 (0 means the decomposed data are old and not valid now). When the previous decomposition type is determined, the name of the array and the conversion line to gather the array (i.e. the *gather line*) is written to the file with the following parameters: *original ID* of the calling

processor, the *previous processor group*, the *number of processors* in the previous processor group, the *conversion type* (gather from previous decomposition type), the *size* of the array and the *name* of the array. Examples of *gather lines* are given in Section 7.4.



7. PARALLELIZER

The Parallelizer is used to produce the parallel version of the user's sequential program. While copying the sequential program line by line to a new file, the Parallelizer inserts some necessary lines, such as MPI and Conversion Function calls, which enables the new program to execute parallel.

The Parallelizer first reads the output files of the User Program Parser and Array Information Table Generator. Then, it begins to read the user's sequential program line by line. In each line, it looks for some keywords. These keywords are *"include"*, *"main"*, *"declaration"* (matrix or vector declaration), *"Level"* (indicating the beginning of a level), *"printf"* and *"return"*. If one of these keywords is found in the line, the function which is related with the keyword is called. In this chapter, these functions are explained in detail. If none of the above keywords are found in the line, the Parallelizer copies the line to the new file.

7.1. The Parallelization Process

The user has to enter the name of the sequential program to be parallelized as an input. The user may also enter the name of the output file, this is optional. If the user determined the name of the output file, the specified name is used to open the output file. Otherwise, *"-PARALLEL.c"* is appended to the sequential program's file name.

The user has to determine the beginning of each level in his program. He has to put some special comment lines at the beginning of the levels. These lines will help the Parallelizer to realize that a new level is beginning. Additionally, the lines where a matrix or a vector is declared must include a comment, which indicates the declaration of the array. The Parallelizer will skip such lines and declare the arrays in its own way.

From the output file of Array Information Table Generator, the Parallelizer reads the number of levels, matrices and vectors, the name and size of the arrays used in the user's program. While reading these information, the maximum size of matrices and the maximum

size of vectors are determined. During the execution of Conversion Functions, a temporary matrix or vector will be used. These temporary arrays should have the maximum size, to be capable of storing the transferred data.

For the first LHS array of all computation levels, the name of the array, the number of array's indexes, the name of each index (e.g. i, j), the number of possible decompositions for this level, the possible decompositions of the array for this level and the number of *for loops* in this level are read from the output file of User Program Parser. For each level, Array Information Table Generator appended the decomposition selected by the best chromosome to the same file. The selected possible decomposition for each level is also read from the file.

The "*number of processor groups*" and "*logarithm of the number of processors in each group in base 2*" are read from Parallel Program Optimizer's input file. (The number of processors in a group must be a power of 2). The original ID of first processor in each group is calculated to be used in during data transfers.

The user is asked whether he would like to measure the duration of each conversion, each level and whole program. Then, the user is asked whether he would like to see the host names of the computers which the parallel program executes on.

After reading the output files of the previous programs and learning the user's choices, the Parallelizer reads the sequential program line by line:

- If the line includes "*include*" keyword, *includeStatement* function is called with parameter 0.
- If the line includes "*main*" keyword, *mainStatement* function is called.
- If the line includes a comment which indicates that there is a matrix/vector declaration, the line is skipped. Because, array declarations will be done in *mainStatement* function.
- If the line includes a comment which indicates that there is a new level beginning, the level header is copied. The *currentLevel* (number) is read from the level header line and *levelStatement* function is called.
- If the line includes "*return*" keyword, *returnStatement* function is called.

- If the line does not include any of the keywords above, *copyUserStatement* function is called.

7.2. The "include" keyword

When "include" keyword is found in the line, *includeStatement* function declares some global variables and includes header files in the parallel program. *converter.h* is a header file which is included to be able to call the Conversion Functions. There are some standard C header files included in *converter.h*, such as *stdio.h*, *mpi.h*, *time.h* and *math.h*. If there is an include statement in user's sequential program which includes one of these header files and if *converter.h* is not included yet, *converter.h* and necessary lines are written. If it includes a different header file, the line is copied. *converter.h* and the necessary lines are added as follows:

- The number of processor groups and the array which stores the ID of first processor in each group are declared. These variables are global not only for the user's parallelized program, but also for the Conversion Functions.
- *converter.h* is included.
- If the user wants to measure the durations, the function prototypes and the global variables used for measuring the duration are declared. An array will be used to store the duration of each conversion, each level and whole program.
- The global variables *root* and *originalMyid* (the unique ID of each processor) are declared.

7.3. The "main" keyword

When "main" keyword is found in the line, first, some checks are done:

- If *main* function was written to the parallelized user program before, the function returns.
- There may not be an "include" statement in the user's sequential program. In that case, *includeStatement* function is called to print the necessary lines, before executing

mainStatement function.

Then, in the parallelized program, the main function begins and some variables are declared:

- All matrices and *tempMatrix* are declared in type "long **". All vectors and *tempVector* are declared in type "long *". (*tempMatrix* and *tempVector* will be used as temporary arrays during the data transfer in the Conversion Functions.)
- Some variables, which will be used in the parallelized program, are declared.
- If the user wanted to measure the durations, some local variables used for measuring the duration of whole program are declared.
- If the user wanted to see the hosts, some variables to print the hostname of the processors are declared.
- Memory allocation lines for all matrices, for all vectors, for *tempMatrix* and *tempVector* are written. *tempMatrix*'s size is equal to the maximum size of matrices and *tempVector*'s size is equal to the maximum size of vectors.
- *MPI_Init*, *MPI_Comm_size* and *MPI_Comm_rank* functions are called to initialize the MPI.
- If the durations will be measured, the root processor will start time in the user's parallelized program, to measure the duration of whole program.
- Each processor will calculate the ID of the group, which it belongs to, and its own ID (*myid*) in that group, in user's parallelized program.

7.4. The "Level" keyword

If the line includes a comment which indicates that there is a new level beginning, the comment is copied. The *currentLevel* (number) is read from the comment line and *levelStatement* function is called. The number of the level indicates the type of the level. A level may be an initialization level, a computation level or a gather-fprintf level. According to its type, the level is parallelized and copied to the new file paying attention to the indentation.

The duration of conversion (i.e. communication) and the duration of computation (us-

ing the arrays) are measured separately. If the durations will be measured, before and after each conversion/gather line and each level (except the initialization level), *startTime* and *endTime* functions are called respectively in the parallelized program to measure the duration of the operation. The durations will be stored in an array in the parallelized program.

If the level is an initialization level, the root processor in Group 0 will execute this level in the parallelized program. The statements of the level are copied line by line from the user's sequential program.

If the level is a computation level, the conversion lines to be placed before this level are read from the output file of Array Information Table Generator. Two example *conversion lines* are shown below:

```
matrixConverter(originalMyid, 1, 8, 0, 16, mH2R, 1120, b, tempMatrix);
vectorConverter(originalMyid, 0, 1, 0, 4, vHinit, 2048, d, d);
```

The format of the example *conversion lines* are as follows:

```
matrixConverter(originalMyid, previousGroup, p, nextGroup, q, conversionOption,
matrixSize, matrixName, tempMatrix);
vectorConverter(originalMyid, previousGroup, p, nextGroup, q, initializationOption,
vectorSize, vectorName, vectorName);
```

where

- *originalMyid*: The unique ID of the processor given by `MPI_Comm_rank` function.
- *previousGroup*: The previous processor group's ID, to which the array was decomposed in one of the previous levels.
- *p*: The number of processors, to which the array was decomposed in the previous group.
- *nextGroup*: The processor group's ID, which the array will be decomposed in this level.

- q : The number of processors, to which the array will be decomposed in the next group.
- *initializationOption* / *conversionOption*: Each conversion function has a unique identifier. *initializationOption* is the ID of any initialization function. *conversionOption* is the ID of any conversion function. In the example lines, the *mH2R* parameter indicates that the matrix will be converted from Horizontal to Rectangular decomposition type. The *vHinit* parameter indicates that the vector will be initialized in Horizontal decomposition type. These parameters are defined in *definitions.h* and matched with a unique ID.
- *matrixSize* / *vectorSize*: The size of the array to be converted or decomposed.
- *matrixName* / *vectorName*: The name of the array to be converted or decomposed.
- *tempMatrix* / *tempVector*: The temporary array used during conversion operation.

The last read conversion line is tokenized and the *nextGroup* and q values are set in the user's parallelized program. The processors, whose group equals *nextGroup* and whose *myid* is less than q , will execute the level. In other words, the first q processors of the *nextGroup* will do the computation.

Since the arrays will be decomposed (partitioned) in the parallelized program, in the loop statements, the "*for loop limit(s)*" must be divided by a value, according to the decomposition. In a level, there may be many arrays used for the computations. Here, the Parallelizer is interested in the decomposition type of the first expression statement's LHS array. For the first LHS array, all possible decompositions were stored. The selected possible decomposition for each level was also stored. Using these two data, the decomposition type selected for this level's first LHS array is known.

When a "*for loop*" statement is read from the user's sequential program, the line is tokenized. The "*for loop counter*" (e.g. i, j, k , etc.) is determined. The index names (e.g. i, j) of first LHS array were also known. The "*for loop counter*" is compared with the first and second index of first LHS array separately. If they are the same, the *for loop limit* is divided. Otherwise, the *for* statement is copied. To clarify this situation, the example level in Figure 7.1 will be analyzed.

```

for(i = 0; i < SIZE*; i++)
{
    for(j = 0; j < SIZE**; j++)
    {
        a[i][j] = b[i][j] + c[i][j];
        d[i][j] = e[i][j] - f[i][j];
    }
}

```

Figure 7.1. The modification of the *for loop limit* according to the decomposition type

If the decomposition type selected for this level's first LHS array (i.e. matrix *a*) is:

- Horizontal, *SIZE** is divided by q .
- Vertical, *SIZE*** is divided by q .
- Rectangular, *SIZE** is divided by h_q and *SIZE*** is divided by v_q .
- Horizontal Block, *SIZE** is divided by h_q .
- Vertical Block, *SIZE*** is divided by v_q .

When the array is partitioned, h_q is the number of rows, v_q is the number of columns. If the decomposition type of the first LHS array is Replicated or None, the *for loop limit* will not be divided. All the expression statements in the *for loop* are copied to the parallelized program.

At the end of the program, the user may want to write the result of the computations to a file. Since the arrays are decomposed during the computations, first, the pieces of the arrays (to be printed) must be gathered to the root processor. Printing a vector or a matrix requires *for loop* statements. Therefore, the statement groups, that are used to print an array, may be accepted as a level. In this project, such levels are called *gather-fprintf levels*.

The gather lines were written to an output file by Array Information Table Generator for each array. All gather lines are read from the file. Two example *gather lines* are shown

below:

```
a|matrixConverter(originalMyid, 0, 8, 0, 1, mRGather, 512, a, a);
e|vectorConverter(originalMyid, 1, 4, 0, 1, vHGather, 1792, e, e);
```

The format of the example *gather lines* are as follows:

```
<matrixName>|matrixConverter(originalMyid, previousGroup, p, nextGroup, q,
gatherOption, matrixSize, matrixName, matrixName);
<vectorName>|vectorConverter(originalMyid, previousGroup, p, nextGroup, q,
gatherOption, vectorSize, vectorName, vectorName);
```

where ”|” is the delimiter and *gatherOption* is the ID of any gather function. In the example lines, the *mRGather* parameter indicates that the matrix will be gathered in Rectangular decomposition type. The *vHGather* parameter indicates that the vector will be gathered in Horizontal decomposition type. These parameters are defined in *definitions.h* and matched with a unique ID. The other parameters used in the gather lines were explained above. In the gather lines, *nextGroup* is 0 and *q* is 1 all the time, indicating the root processor in Group 0.

After reading the gather lines from the output file, the Parallelizer begins copying the gather-fprintf level. The comment before the gather level includes the name of the array to be gathered and printed to file. The comment is tokenized and the name of the array to be gathered is determined. According to the name of the array, the gather line of the array is found and printed to the parallelized program. The level will be executed by the root processor in Group 0. The statements in the level are copied to the parallelized program.

7.5. The ”for” keyword

After a line which includes ”*Level*” keyword, there has to be a *for loop* statement. The statement is tokenized and the *for loop limit* in the condition is divided by the input parameter, *divisor*.

```

for(i = 0; i < SIZE; i++)
    for(j = 0; j < SIZE; j++)
        a[i][j] = b[i][j] + c[i][j];

```

(a) Sequential code

```

for(i = 0; i < SIZE / 4; i++)
    for(j = 0; j < SIZE / 4; j++)
        a[i][j] = b[i][j] + c[i][j];

```

(b) Parallelized code

Figure 7.2. The *for loop limit* is a string.

```

for(i = 0; i < 2048; i++)
    for(j = 0; j < 2048; j++)
        a[i][j] = b[i][j] + c[i][j];

```

(a) Sequential code

```

for(i = 0; i < 512; i++)
    for(j = 0; j < 512; j++)
        a[i][j] = b[i][j] + c[i][j];

```

(b) Parallelized code

Figure 7.3. The *for loop limit* is an integer.

In Figure 7.2 and Figure 7.3, there are two example levels. The line is tokenized to modify the second statement (the condition). The first statement (e.g. `"i = 0;"` or `"j = 0;"`) is copied to the parallelized program. The condition may be similar to `"i < SIZE"` (Figure 7.2-a) or `"i < 2048"` (Figure 7.3-a). If the *for loop limit* is a string, the condition `"i < SIZE"`, the division sign and the divisor are written to the new file. (For example, if the divisor is 4, `"i < SIZE / 4"` is written to the file (Figure 7.2-b)). If the *for loop limit* is an integer, the string part of the condition (`"i < "`) and the quotient of limit and divisor are written to the new file. (For example, if the limit is 2048 and the divisor is 4, `"i < 512"` is written to the file (Figure 7.3-b)). The third statement (e.g. `"i++"` or `"j++"`) is copied to the parallelized program.

7.6. The "return" keyword

When "return" keyword is found in the line, the durations are printed, MPI is finalized and memory for arrays is freed, in user's parallelized program.

- If the user wanted to measure the durations, the root processor prints the duration of each conversion, each level and whole program from the array, where the durations are stored. If the user wanted to see the hosts, each processor prints its original ID, host ID and host name.
- *MPI_Finalize* function is called to finalize the MPI. The memory locations allocated for all matrices, *tempMatrix*, all vectors and *tempVector* are freed. `"return 0;"` is written and the main function is closed.
- If the user wanted to measure the durations, the functions *startTime* and *endTime* functions are written to the end of the parallelized program.

7.7. Copying the User's Statements

The lines which do not include a keyword are copied to the parallelized program. The User Program Parser program gives error for file pointer declaration (*FILE *...;*), *fopen* and *fclose* function calls in the user's sequential program. Therefore, such lines are commented out. If a line includes *"FILE"*, *"fopen"* or *"fclose"* keywords, then the Parallelizer tokenizes

the string, removes the comment marks and copies the line to the parallelized program. When the User Program Parser program is adapted to such commands, these lines will not be commented out in the user's program and this part of the program will be removed. Examples for commented lines are as follows:

```
/* FILE *bFile; */  
/* bFile = fopen("./bFile.txt", "w"); */  
/* fclose(bFile); */
```

If the line is a comment (other than the ones mentioned above), the line is copied. If the line is a *"printf"* statement, the related function is called. The other lines are copied to the parallelized program.

7.8. The **"printf"** keyword

Since the levels will execute on many processors, the user may want to know which processor printed the text. The text to be printed is copied to the parallelized program with some additional identifying information, such as *originalMyid*, *myGroup*, *myid* of the printing processor.

8. EXPERIMENTS

In [17, 18], the best number of processors and the best alignment had to be determined for each level. The aim was to minimize the total execution time of the program. Moreover, a communication structure S had to be chosen for each level. To achieve this, the feasible communication structures were identified for all arrays at a level. Then, the common feasible communication structures were selected. This selection was not done randomly. Instead, the data transfer times for all these structures were calculated and the one that gave the minimum local transfer time was chosen for that level.

In this thesis, the communication structures are not used. Instead of them, the Conversion Functions are used to transfer data among the processors. However, there is a new variable: *the group of processors*. The problem includes the following variables:

- L :** The number of levels in the user's program
- D :** The alignments of data decomposition patterns at each level
- G :** The number of processor groups (where the processors of the same group are identical, but the processors of different groups are not)
- P :** The number of processors with distributed memory in each group

The Parallel Program Optimizer finds the best mapping of D alignments, G groups and P processors which minimizes the total expected execution time of the user's program.

8.1. Measurement Experiments

Many experiments are done to define the heterogeneous test environment, to measure the performance and to predict the execution time of the parallel programs. In this section, these experiments will be explained.

8.1.1. Heterogeneous Test Environment

In this study, there were 24 processors. The first 16 processors were Intel Pentium 4, 2.00 GHz. The other 8 processors were Intel Pentium 4, 2.60 GHz. Therefore, the processors could be separated into two groups, according to their computation speed. The speed of the local area network was 100 Mbit/sec. However, some of the processors in the second group were connected to a different switch. This case affected the communication speed. To communicate with some of the processors took longer time. So, the second group is separated into two groups, according to their communication speed. Finally, the processors were separated into three groups, according to both computation and communication speeds. In this section, the experiments related with the heterogeneous environment are explained in details.

In order to group the processors, computation and communication performances of the processors were measured. To measure the computation performances of the processors, the benchmark in Figure 8.1 was executed. Each processor multiplied the elements of a 2048×2048 matrix by 5 and measured the duration of this process for 5 times. The average duration calculated for each processor is shown in Table 8.1. The first group had 16 relatively slow processors. The second group had 8 relatively fast processors.

```

for(k = 0; k < loopLimit; k++)
{
    startTime();
    for(i = 0; i < SIZE; i++)
    {
        for(j = 0; j < SIZE; j++)
        {
            a[i][j] = a[i][j] * 5;
        }
    }
    endTime(k);
}

```

Figure 8.1. The benchmark which measures the computation time

To measure the communication performances of the processors, the pingpong benchmark was executed as shown in Figure 8.2. One of the processors was selected as the *root*. The root processor sent a 2048×2048 matrix to another processor. The other processor

Table 8.1. The average computation time of each processor

Host name of the processor	Average Computation Time (msec)
s60	35.63
s61	35.63
s62	35.87
s63	35.65
s64	35.67
s65	35.36
s66	35.81
s67	35.57
s68	35.75
s69	35.26
s70	35.56
s71	35.61
s72	35.63
s73	35.77
s74	35.61
s75	35.60
linux38	27.22
linux35	27.25
linux34	27.24
linux33	27.19
linux30	27.14
linux29	27.18
linux26	27.12
linux25	27.09

received the matrix and sent it back to the root processor. The root processor calculated the duration of this send-receive. This process was repeated by the root processor for all other processors. This benchmark was executed for all processors, each time a different processor was selected as root.

```

if(originalMyid == root)
{
    for(i = 0; i < nproc; i++)
    {
        if(i != root)
        {
            startTime();
            MPI_Send(&a[0][0], SIZE*SIZE, MPI_LONG,
                    i, tag, MPI_COMM_WORLD);
            MPI_Recv(&a[0][0], SIZE*SIZE, MPI_LONG,
                    i, tag, MPI_COMM_WORLD, &status);
            endTime(i);
        }
    }
}
else
{
    MPI_Recv(&a[0][0], SIZE*SIZE, MPI_LONG, root,
            tag, MPI_COMM_WORLD, &status);
    MPI_Send(&a[0][0], SIZE*SIZE, MPI_LONG, root,
            tag, MPI_COMM_WORLD);
}

```

Figure 8.2. The pingpong benchmark

Comparing the durations, the second group was separated into 2 groups, named as Group 1 and Group 2. It was seen that the communication time of the processors in Group 0 and Group 1 were close to each other. However, to communicate with the processors of Group 2 took longer time. Table 8.2 shows the average communication times (in msec) among the processors of the groups.

Table 8.2. The average communication time (in msec) between the processors of the groups

	Group 0	Group 1	Group 2
Group 0	3032.11	2998.56	3680.86
Group 1	3013.89	2975.05	3681.43
Group 2	3729.94	3670.17	3610.25

The average communication time among the processors was 3032.11 msec for Group 0,

2975.05 msec for Group 1 and 3610.25 msec for Group 2. When the upper and lower parts of Table 8.2 were compared, the average communication times to transfer data between Group 0 and Group 1 were 2998.56 msec and 3013.89 msec (where the difference was 15.33 msec). The average communication times to transfer data between Group 0 and Group 2 were 3680.86 msec and 3729.94 msec (where the difference was 49.08 msec). The average communication times to transfer data between Group 1 and Group 2 were 3681.43 msec and 3670.17 msec (where the difference was 11.26 msec). Thus, the table was symmetric, as expected. The table also showed that to communicate with Group 2 took longer time, when compared with the communication durations of Groups 0 and 1.

8.1.2. Performance Measurement of the Environment

All the Conversion Functions were executed for all possibilities of processor groups, processor numbers and array sizes. Since there were 3 processor groups, *previous group* and *next group* took 0, 1 and 2 values. The previous processor number p and the next processor number q took 1, 4, 8 and 16 values. The functions were tested with different array sizes: 512, 864, 1120, 1312, 1488, 1648, 1792, 1920 and 2048. These sizes were divisible by 4, 8 and 16. With all these possibilities, the durations were measured for 20 times as shown in Figure 8.3 and Figure 8.4. MPI.Barrier function is used to force the processors wait for each other, before executing the next statement. In *startTime* and *endTime* functions, *gettimeofday* function is called to save the instant. The difference of the instants taken by *startTime* and *endTime* gives the duration of the conversion. *matrixConverter* and *vectorConverter* functions have a switch-case structure. According to the *conversion option*, they call the related Conversion Function and pass the parameters to it. Detailed information about the parameters of these functions are given in Section 7.4.

The measured durations were sorted in increasing order. During the measurements, some durations may have high values, because of the delay caused by the load in network and the operating system. To eliminate such *peek* (high) values, the first 15 durations are considered. With the 15 measured durations, the average durations are calculated. Using these average values, the slope and intercept of each Conversion Function, for different previous and next groups and for different p and q values are obtained. For example, from

```

MPI_Barrier(MPI_COMM_WORLD);

for(d = 0; d < 20; d++)
{
    startTime();
    matrixConverter(originalMyid, ucPrevGroup, ucP,
                    ucNextGroup, ucQ, convOption, N, a, A);
    MPI_Barrier(MPI_COMM_WORLD);
    endTime(2, d);
}

```

Figure 8.3. The measurement of matrix conversion duration

```

MPI_Barrier(MPI_COMM_WORLD);

for(d = 0; d < 20; d++)
{
    startTime();
    vectorConverter(originalMyid, ucPrevGroup, ucP,
                   ucNextGroup, ucQ, convOption, N, a, A);
    MPI_Barrier(MPI_COMM_WORLD);
    endTime(2, d);
}

```

Figure 8.4. The measurement of vector conversion duration

Group 0 to Group 0, where p and q are equal to 4, Horizontal to Vertical conversion for matrices (mH2V) has the slope $2.23\text{e-}05$ and intercept $4.11\text{e+}00$. All slopes and intercepts are presented in Appendix B.

The estimated communication time for a conversion is calculated as

$$t_{comm}^c = ax + b \quad (8.1)$$

where t_{comm}^c is the estimated communication time for the conversion of an RHS array. x is the total array size in bytes. If there are N elements in one dimension of the array, x is $N \times N \times 8$ bytes for matrices, $N \times 8$ bytes for vectors (1 MPI_LONG element is 8 bytes). a is the slope and b is the intercept of the total array size in bytes, according to the previous and next groups, p and q values and the conversion.

Before the execution of a level, the arrays used in RHS must be converted to the decomposition type chosen for that level. If more than one arrays will be converted, the duration of each conversion must be added. Therefore, in order to find the total communication time before the level, the estimated communication times of the conversions for all RHS arrays in the level are added (Equation 8.2).

$$t_{comm}^l = \sum_{c=1}^C t_{comm}^c \quad (8.2)$$

where t_{comm}^l is total communication time before the level, C is the number of conversions and t_{comm}^c is the estimated communication time for the conversion of an RHS array.

8.1.3. Time Prediction of Parallel Programs

In order to predict the total execution time of a parallel program, the communication and computation times of each level must be considered. The total communication time of converting the RHS arrays in the level is calculated using Equations 8.1 and 8.2.

To estimate the computation times of each level, in the user's sequential program, the size of the arrays and the *for loop limits* (the ones which are equal to the array sizes) are decreased to a relatively small value (e.g. If the original size is 2048, it is decreased to 1024). Then, the program is executed sequentially on one processor from each processor group and the durations of each computation level is measured.

```

for (i = 0; i < N1; i++)
{
    for (j = 0; j < N2; j++)
        a[i][j] = 0.0;                                // statement 1

    for (k = 0; k < N3; k++)
        for (j = 0; j < N4; j++)
            a[i][j] = b[i][j] + c[i][k] * d[k][j];    // statement 2
}

```

Figure 8.5. Computation level example

In Figure 8.5, statement 1 will be executed $N_1 \times N_2$ times and statement 2 will be executed $N_1 \times N_3 \times N_4$ times. In such a case, to measure the computation time of the statements, the level is decomposed as shown in Figure 8.6. The durations of the new loops are measured separately.

```

for (i = 0; i < N1; i++)
{
    for (j = 0; j < N2; j++)
        a[i][j] = 0.0;                                // statement 1
}

for (i = 0; i < N1; i++)
{
    for (k = 0; k < N3; k++)
        for (j = 0; j < N4; j++)
            a[i][j] = b[i][j] + c[i][k] * d[k][j];    // statement 2
}

```

Figure 8.6. Decomposed computation level example

In the estimated computation time calculations, the ratio of the original *for loop limits* (e.g. 2048) and the measured limits (e.g. 1024) is used. The computation time is formulated

as

$$t_{comp}^l = \frac{\sum_{s=1}^S t_s^l (\prod_{f=1}^F \frac{N_f}{n_f})}{q} \quad (8.3)$$

where t_{comp}^l is the estimated computation time, S is the number of inner loops, t_s^l is the measured computation time (with a small data size), F is the number of *for loops* outside the statement, N_f is the *for loop limit* in the sequential program (real data size), n_f is the *for loop limit* used during the measurements and q is the number of processors which will execute this level. The product of the *for loop limits* in the sequential program (N_f) are divided by the product of the *for loop limits* used during the measurements (n_f). The ratio is multiplied by the measured computation time (t_s^l). If the loop is similar to Figure 8.5, there are two t_s^l values. In such cases, the products of the ratio and t_s^l are added. The sum is divided by q . The result is the estimated computation time (t_{comp}^l) for level l .

The execution time of a level in the parallel program can be expressed as the sum of the computation and communication times

$$t_{exec}^l = t_{comp}^l + t_{comm}^l \quad (8.4)$$

where t_{exec}^l denotes the total execution time, t_{comp}^l denotes the computation time and t_{comm}^l denotes the total communication time for level l [17, 18]. The total execution time of the program T is the sum of the execution times of all levels in the program

$$T = \sum_{l=1}^L t_{exec}^l \quad (8.5)$$

where L denotes the number of levels [17, 18].

Parallel Program Optimizer uses the equations explained in this chapter in its fitness function. The total execution time of the program T , found in Equation 8.5, is the fitness of the chromosome.

8.2. Optimization Experiments

Two sample programs were planned to be parallelized using this tool: the Hessenberg Reduction (Figure 8.7) and Dongarra's benchmark (Figure 8.8) programs, which were used in [17, 18]. Figures 8.9 and 8.10 show the possible decompositions (determined by User Program Parser) for the levels of the programs. The first column is the name of the array. The second column is the decomposition selected for the array (0 for None, 1 for Horizontal, 2 for Vertical, 3 for Rectangular, 4 for Replicated, 5 for Horizontal Block, 6 for Vertical Block). The third column is the *R/L position* of the array (explained in Section 6.1.1).

Using the Parallel Program Optimizer, the first experiment was done to find an expected fitness value. The population size was 30, the offspring count was 28. Before crossover, the mates were selected according to tournament mate selection method. The *tour size* was set to 4. One-point crossover was applied to the mates. The crossover rate was 1.0. The mutation rate was $1 / \text{chromosome length}$, where the *chromosome length* was equal to the number of computation levels. (The Hessenberg Reduction program had 5 computation levels, Dongarra's benchmark program had 11 computation levels). Hill climbing is applied after initialization and after mutation of the chromosomes.

During the hill climbing, the number of processors p in each gene is doubled (incremented) and halved (decremented). The fitnesses of the chromosome found in both cases and the original fitness are compared, and the p value which gave the best fitness is accepted. There are two types of local search methods. In Next Gradient method, the p values are modified beginning from the first level until the last, one by one. In Davis method, the p values are modified in a random order. To find the expected fitness, Davis local search method was used. *Maximum hill climbing iterations* was 1. Setting *run count* to 1 and *total time* of the experiment to 500 seconds, Parallel Program Optimizer was executed for each benchmark. As shown in Table 8.3, the best chromosome for Hessenberg Reduction chose the 16 processors in Group 0 for all computation levels. From the decompositions proposed by User Program Parser, the best chromosome selected 2, 2, 2, 1, 0 for the computation levels 1, 2, 3, 4 and 5, respectively. The fitness of this best chromosome was 250438.946887 msec (4 minutes and 10 seconds). The fitnesses of all possibilities in Group 0 with 1, 4 and 16 processors and for

```

/* Level 1 */
for (i=0; i<N; ++i)
{
    for (j=0; j<N; ++j)
    {
        a[i][j]=0;
        for (k=0; k<N; ++k)
        {
            a[i][j]=a[i][j]+V[i][k]*T[k][j]; /* VT */
        }
    }
}

/* Level 2 */
for (i=0; i<N; ++i)
{
    for (j=0; j<N; ++j)
    {
        b[i][j]=0;
        for (k=0; k<N; ++k)
        {
            b[i][j]=b[i][j]+a[i][k]*V[j][k]; /* VTVt */
        }
    }
}

/* Level 3 */
for (i=0; i<N; ++i)
{
    for (j=0; j<N; ++j)
    {
        c[i][j]=I[i][j]-b[i][j]; /* I-VTVt */
    }
}

/* Level 4 */
for (i=0; i<N; ++i)
{
    for (j=0; j<N; ++j)
    {
        d[i][j]=0;
        for (k=0; k<N; ++k)
        {
            d[i][j]=d[i][j]+c[k][i]*A[k][j]; /* (I-VTVt)t A */
        }
    }
}

/* Level 5 */
for (i=0; i<N; ++i)
{
    for (j=0; j<N; ++j)
    {
        e[i][j]=0;
        for (k=0; k<N; ++k)
        {
            e[i][j]=e[i][j]+d[i][k]*c[k][j]; /* (I-VTVt)t A (I-VTVt) */
        }
    }
}

```

Figure 8.7. Hessenberg Reduction

```

/* Level 1 */
for (i=0;i<nsizel;i++)
{
    for (j=0;j<nsizel;j++)
        x2[i][j]=0.0;
    for (k=0;k<nsizel;k++)
        for (j=0;j<nsizel;j++)
            x2[i][j]=x2[i][j]
                +y2[i][k]*z2[k][j];
}

/* Level 2 */
for (j=0;j<nsizel;j++)
    for (i=0;i<nsizel;i++)
    {
        x2[i][j]=y2[i][j];
        y2[i][j]=x2[i][j]*z2[i][j];
    }

/* Level 3 */
for (i=0;i<nsizel;i++)
    for (j=0;j<nsizel;j++)
        a2[j][i]=b2[j][i]+c2[j][i];

/* Level 4 */
for (i=0;i<nsizel;i++)
    for (j=0;j<nsizel;j++)
        a2[j][i]=b2[j][i]+s1*c2[j][i];

/* Level 5 */
for (i=0;i<nsizel;i++)
    for (j=0;j<nsizel;j++)
        a2[j][i]=b2[j][i]+d1[i]*c2[j][i];

/* Level 6 */
for (i=0;i<nsizel;i++)
    for (j=0;j<nsizel;j++)
    {
        a2[j][i]=b2[j][i]+c2[j][i];
        b2[j][i]=a2[j][i]+d2[j][i];
    }

/* Level 7 */
for (i=0;i<nsizel;i++)
    for (j=0;j<nsizel;j++)
    {
        a1[i]=b2[j][i]+c2[j][i];
        a2[j][i]=a1[i]*d2[j][i];
    }

/* Level 8 */
for (i=0;i<nsizel;i++)
    for (j=0;j<nsizel;j++)
    {
        a1[i]=b2[j][i]+c2[j][i];
        a2[j][i]=a1[i]*d2[j][i];
        b1[j]=e2[j][i]+f2[j][i];
        b2[j][i]=b1[j]*g2[j][i];
    }

/* Level 9 */
for (i=0;i<nsizel;i++)
    for (j=0;j<nsizel;j++)
        a2[i][j]=b2[i][j]+c2[i][j]
            +d2[i][j]+e2[i][j]-f2[i][j];

/* Level 10 */
for (i=0;i<nsizel;i++)
{
    for (j=0;j<nsizel;j++)
        a1[j]=d2[j][i]+a2[j][i];
    for (j=0;j<nsizel;j++)
        b2[j][i]=a1[j]*b2[j][i];
}

/* Level 11 */
for (i=0;i<nsizel;i++)
{
    for (j=0;j<nsizel;j++)
        a1[j]=d2[j][i]+a2[j][i];
    for (j=0;j<nsizel;j++)
        b2[j][i]=a1[j]*b2[j][i];
}

```

Figure 8.8. Dongarra's benchmark

Level: 1 Decomposition: 0	Level: 3 Decomposition: 2
V 1 1	I 3 1
T 4 1	b 3 1
a 1 2	c 3 3
Level: 1 Decomposition: 1	Level: 4 Decomposition: 0
V 4 1	c 2 1
T 2 1	A 4 1
a 2 2	d 1 2
Level: 1 Decomposition: 2	Level: 4 Decomposition: 1
V 5 1	c 4 1
T 6 1	A 2 1
a 3 2	d 2 2
Level: 2 Decomposition: 0	Level: 4 Decomposition: 2
a 1 1	c 6 1
V 4 1	A 6 1
b 1 2	d 3 2
Level: 2 Decomposition: 1	Level: 5 Decomposition: 0
a 4 1	d 1 1
V 1 1	c 4 1
b 2 2	e 1 2
Level: 2 Decomposition: 2	Level: 5 Decomposition: 1
a 5 1	d 4 1
V 5 1	c 2 1
b 3 2	e 2 2
Level: 3 Decomposition: 0	Level: 5 Decomposition: 2
I 1 1	d 5 1
b 1 1	c 6 1
c 1 3	e 3 2
Level: 3 Decomposition: 1	
I 2 1	
b 2 1	
c 2 3	

Figure 8.9. The possible decompositions for Hessenberg Reduction

Level: 1 Decomposition: 0	Level: 6 Decomposition: 0	Level: 9 Decomposition: 0
y2 1 1	c2 1 1	b2 1 1
z2 4 1	d2 1 1	c2 1 1
x2 1 2	a2 1 2	d2 1 1
Level: 1 Decomposition: 1	b2 1 2	e2 1 1
y2 4 1	Level: 6 Decomposition: 1	f2 1 1
z2 2 1	c2 2 1	a2 1 3
x2 2 2	d2 2 1	Level: 9 Decomposition: 1
Level: 1 Decomposition: 2	a2 2 2	b2 2 1
y2 5 1	b2 2 2	c2 2 1
z2 6 1	Level: 6 Decomposition: 2	d2 2 1
x2 3 2	c2 3 1	e2 2 1
Level: 2 Decomposition: 0	d2 3 1	f2 2 1
z2 1 1	a2 3 2	a2 2 3
x2 1 2	b2 3 2	Level: 9 Decomposition: 2
y2 1 2	Level: 7 Decomposition: 0	b2 3 1
Level: 2 Decomposition: 1	b2 2 1	c2 3 1
z2 2 1	c2 2 1	d2 3 1
x2 2 2	d2 2 1	e2 3 1
y2 2 2	a1 1 2	f2 3 1
Level: 2 Decomposition: 2	a2 2 3	a2 3 3
z2 3 1	Level: 7 Decomposition: 1	Level: 10 Decomposition: 0
x2 3 2	b2 1 1	d2 1 1
y2 3 2	c2 1 1	a2 1 1
Level: 3 Decomposition: 0	d2 1 1	a1 1 2
b2 1 1	a1 4 2	b2 1 2
c2 1 1	a2 1 3	Level: 10 Decomposition: 1
a2 1 3	Level: 7 Decomposition: 2	d2 2 1
Level: 3 Decomposition: 1	b2 3 1	a2 2 1
b2 2 1	c2 3 1	a1 4 2
c2 2 1	d2 3 1	b2 2 2
a2 2 3	a1 5 2	Level: 10 Decomposition: 2
Level: 3 Decomposition: 2	a2 3 3	d2 3 1
b2 3 1	Level: 8 Decomposition: 0	a2 3 1
c2 3 1	c2 2 1	a1 5 2
a2 3 3	d2 2 1	b2 3 2
Level: 4 Decomposition: 0	e2 2 1	Level: 11 Decomposition: 0
b2 1 1	f2 2 1	d2 1 1
c2 1 1	g2 2 1	a2 1 1
a2 1 3	a1 1 2	a1 1 2
Level: 4 Decomposition: 1	b2 2 2	b2 1 2
b2 2 1	b1 4 2	Level: 11 Decomposition: 1
c2 2 1	a2 2 3	d2 2 1
a2 2 3	Level: 8 Decomposition: 1	a2 2 1
Level: 4 Decomposition: 2	c2 1 1	a1 4 2
b2 3 1	d2 1 1	b2 2 2
c2 3 1	e2 1 1	Level: 11 Decomposition: 2
a2 3 3	f2 1 1	d2 3 1
Level: 5 Decomposition: 0	g2 1 1	a2 3 1
b2 1 1	a1 4 2	a1 5 2
d1 4 1	b2 1 2	b2 3 2
c2 1 1	b1 1 2	
a2 1 3	a2 1 3	
Level: 5 Decomposition: 1	Level: 8 Decomposition: 2	
b2 2 1	c2 3 1	
d1 1 1	d2 3 1	
c2 2 1	e2 3 1	
a2 2 3	f2 3 1	
Level: 5 Decomposition: 2	g2 3 1	
b2 3 1	a1 5 2	
d1 5 1	b2 3 2	
c2 3 1	b1 5 2	
a2 3 3	a2 3 3	

Figure 8.10. The possible decompositions for Dongarra's benchmark

all proposed decompositions were calculated. The minimum fitness was equal to the fitness of the best chromosome. Thus, the results of the Parallel Program Optimizer are accepted as correct.

Table 8.3. The best chromosome for Hessenberg Reduction, where the groups have 16, 4 and 4 processors, $N = 2048$. The fitness is 250438.946887 msec.

	Group of processors	Number of processors	Decomposition Number	t_{comm} in msec	t_{comp} in msec
Level 1	0	16	2	12213.07	58771.98
Level 2	0	16	2	2112.96	8658.97
Level 3	0	16	2	1333.56	3.85
Level 4	0	16	1	6697.28	98251.23
Level 5	0	16	0	1881.60	60514.44

As shown in Table 8.4, the best chromosome for Dongarra's benchmark chose the 16 processors in Group 0 and decomposition 2 for all computation levels. The fitness of the best chromosome was 30191.427704 msec (30 seconds).

Using the fitnesses of the best chromosomes, the expected fitness values were determined for both benchmarks. The experiments were performed using hill climbing with different local search methods and Genetic Algorithm. The run count was set to 50 runs and the total time of each run was set to 500 seconds.

The fitness found for Hessenberg Reduction was 250438.946887 msec. The expected fitness value for this benchmark was determined as 250438.947000 msec. Using hill climbing with Next Gradient local search method, the experiment took 63 seconds, the number of generations was 137.20. (When the Parallel Program Optimizer finds a solution whose fitness is less than the expected value, it quits that run. It does not continue the run until 500 seconds is passed.) Using hill climbing with Davis local search method, the experiment took 78 seconds, the number of generations was 169.38. Without hill climbing, using Genetic Algorithm, it took 84 seconds, the number of generations was 755.06. The frequency of finding the correct result was 1.00 for all methods. They all found the same best chromo-

Table 8.4. The best chromosome for Dongarra's benchmark, where the groups have 16, 4 and 4 processors, $N = 2048$. The fitness is 30191.427704 msec.

	Group of processors	Number of processors	Decomposition Number	t_{comm} in msec	t_{comp} in msec
Level 1	0	16	2	12213.07	8848.27
Level 2	0	16	2	19.72	96.31
Level 3	0	16	2	2654.06	84.14
Level 4	0	16	2	13.06	83.43
Level 5	0	16	2	16.31	82.06
Level 6	0	16	2	1346.62	125.86
Level 7	0	16	2	22.84	111.80
Level 8	0	16	2	4004.53	209.78
Level 9	0	16	2	32.65	7.98
Level 10	0	16	2	20.19	89.29
Level 11	0	16	2	20.19	89.27

some. According to the duration of the experiments and the number of generations, the Next Gradient method was the best.

The fitness found for Dongarra's benchmark was 30191.427704 msec. The expected fitness value for this benchmark was determined as 30191.428000 msec. Using hill climbing with Next Gradient method, the experiment took 4 hours and 28 minutes, the number of generations was 6535.80, the frequency of finding the correct result was 0.42. Using hill climbing with Davis local search method, the experiment took 4 hours and 39 minutes, the number of generations was 7345.32, the frequency of finding the correct result was 0.38. Without hill climbing, using Genetic Algorithm, it took 6 hours and 53 minutes, the number of generations was 82545.24, the frequency of finding the correct result was 0.10. They all found the same best chromosome. According to the duration of the experiments, the number of generations and the frequency of finding the correct result, the Next Gradient method was the best.

Since these benchmark programs are computation intensive, the Parallel Program Optimizer preferred the 16 processors in Group 0. In order to force Parallel Program Optimizer not to select the 16 processors in Group 0, the inputs were entered such that there were 3 groups with 4 processors. In this case, the best chromosome for Hessenberg Reduction chose 4 processors from Group 1 and decomposition 2 for all 5 levels (as shown in Table 8.5). The fitness of this best chromosome was 734366.768356 msec (12 minutes and 14 seconds). Since this benchmark is computation intensive, 4 processors are offered for all levels. The reason for choosing Group 1 is the speed of the processors and communication in that group.

As shown in Table 8.6, the best chromosome for Dongarra's benchmark offered 4 processors from Group 1 for Level 1 and 2, 4 processors from Group 0 for the other levels. It chose decomposition 2, 2, 1, 1, 1, 1, 0, 0, 1, 1, 1 for the levels 1 to 11, respectively. The fitness of the best chromosome was 45893.620401 msec (45 seconds). Parallel Program Optimizer preferred to run all the levels on 4 processors, because of the computation intensity. The reason for choosing Group 1 for Level 1 is the computation intensity of the level and the speed of the processors in that group. Group 1 is also chosen for Level 2, because Level 1

Table 8.5. The best chromosome for Hessenberg Reduction, where the groups have 4, 4 and 4 processors, $N = 2048$. The fitness is 734366.768356 msec.

	Group of processors	Number of processors	Decomposition Number	t_{comm} in msec	t_{comp} in msec
Level 1	1	4	2	7356.70	167407.68
Level 2	1	4	2	2320.83	26452.24
Level 3	1	4	2	1480.24	11.72
Level 4	1	4	2	5180.11	349180.39
Level 5	1	4	2	2325.56	172651.29

and 2 use the same arrays. For the rest of the program, Group 0 is chosen. In Levels 3, 4 and 5, matrices $a2$, $b2$ and $c2$ are used in the computations. In Levels 6 and 7, matrices $a2$, $b2$, $c2$ and $d2$ are used. In Level 8, in addition to these matrices, many arrays are used. If these levels were executed on the processors of Group 1, during the initialization, the data of all these arrays had to be transferred from Group 0 to Group 1. Since these levels will be executed on the processors of Group 0, during the initialization, some of the data will be copied and some will be transferred. The Conversion Functions copy the data where it is possible. The advantage of this property is shown by this example. The Parallel Program Optimizer preferred to copy data in Group 0, instead of transferring it from Group 0 to Group 1.

8.3. Verification of the Results

After setting the sizes of the arrays to 1024, the benchmarks are executed sequentially on one processor from each group. The duration of each computation level is measured for 5 times and the average is found. The average durations for each level (on one processor from each group) are written to a file with the number and limits of the *for loops* outside the statements of the level. These average durations will be used in computation time calculations as t_s^l in Equation 8.3.

The Hessenberg Reduction and Dongarra's benchmark programs [17, 18] are paral-

Table 8.6. The best chromosome for Dongarra's benchmark, where the groups have 4, 4 and 4 processors, $N = 2048$. The fitness is 45893.620401 msec.

	Group of processors	Number of processors	Decomposition Number	t_{comm} in msec	t_{comp} in msec
Level 1	1	4	2	7356.70	27283.16
Level 2	1	4	2	47.95	278.98
Level 3	0	4	1	2201.79	336.57
Level 4	0	4	1	64.94	333.71
Level 5	0	4	1	65.90	328.22
Level 6	0	4	1	1198.31	503.43
Level 7	0	4	0	98.37	447.19
Level 8	0	4	0	3403.24	839.13
Level 9	0	4	1	162.35	31.93
Level 10	0	4	1	99.54	357.15
Level 11	0	4	1	97.97	357.08

lelized, using the best chromosomes found in Parallel Program Optimizer experiments. The estimated and measured total execution times are compared.

The estimated execution time (i.e. the fitness of the best chromosome) for Hessenberg Reduction was 250438 msec (4 minutes and 10 seconds). The average measured execution time of this benchmark is 258311 msec (4 minutes and 18 seconds). The percentage error is 3.05%.

The estimated execution time for Dongarra's benchmark was 30191 msec (30 seconds). The average measured execution time of this benchmark is 33147 msec (33 seconds). The percentage error is 8.92%.

9. CONCLUSION

A sequential program that consists of many computation levels may use some large-sized matrices and vectors. The total execution of the program on a single processor may take too much time. Therefore, such programs are preferred to be executed in parallel.

Dongarra and Lastovetsky stated that a parallel programming style, which became increasingly popular, was to encapsulate the complexities of parallel algorithm design within libraries. An application program (e.g. the user's parallelized program) could then consist of just a sequence of calls to such library functions [1].

In this thesis, a tool is implemented, which makes "parallelizing a sequential program" an easy task. The Conversion Functions Library is written. The library consists of initialization (for scattering data), conversion (for converting an array's decomposition from one type to another), gather (for collecting the results to be printed into a file) and some utility functions. A header file is included to the parallelized program (*converter.h*) which enables the program to call the conversion functions when necessary.

During the implementation of the Conversion Functions, the followings were considered: If the processor is a sender, to which processors it will send data and which part of its data will be sent. If the processor is a receiver, from which processors it will receive data and where this data will be placed.

The tool includes User Program Parser, Conversion Functions Library, Parallel Program Optimizer (Genetic Algorithm Library), Array Information Table Generator and Parallelizer.

The User Program Parser gathers information from the sequential program, such that the number of levels, the number *for loops* in each level, the statements of each level, the name, dimension, size of each array, etc. The method to parallelize a large-sized array is to decompose it into smaller pieces. An array can be decomposed in many ways. The parser

also generates the possible decompositions for each level, according to the alignment of the arrays at that level.

The Parallel Program Optimizer is used to find an optimum solution with Memetic Algorithms. The aim is to minimize the total execution time for the parallel program to be generated. The Parallel Program Optimizer generates many chromosomes, where each gene stores the necessary data for a level, such as the *group of processors*, the *number of processors* (from the selected group) and the selected possible *decomposition* for the arrays of the level. One-point crossover, traditional mutation and some hill climbing methods are applied on the chromosomes. Each chromosome is sent to Array Information Table Generator, which writes the necessary information (the last *level*, *group of processors*, *number of processors* in the group and *R/L position* of the array according to the decomposition type) to matrix table or vector table. The Conversion Functions were used as benchmarks to determine the heterogeneous environment parameters (the slopes and intercepts of total array size for all *processor group*, *number of processors* and *Conversion Function* possibilities). By analyzing the tables and using the results of heterogeneous environment parameters, Array Information Table Generator calculates the fitness of the chromosome (i.e. total execution time of the parallelized program) and returns it to the Parallel Program Optimizer. When Parallel Program Optimizer finds the best chromosome, Array Information Table Generator updates its tables according to the optimum solution. It analyzes the tables and writes the necessary lines to call the Conversion Functions (from parallelized program) into a file.

The Parallelizer is a text processor. It reads the user's sequential program line by line. In each line, it searches for some keywords and calls the related function according to the keyword. While processing and copying lines, the function inserts MPI function calls and Conversion Function calls, in order to make it executable in parallel. At the end, the sequential program is parallelized. Then, it can be compiled and run on the heterogeneous environment.

In the parallelized program, each level may execute on a different number of processors. During the execution of the parallel program, the processors may need to transfer data before the next computation level begins. Instead of using the known data communication

structures, the Conversion Functions are proposed to transfer data directly from the sender to the receiver. They are used between levels to convert the array's decomposition from one type to another. The idea behind the Conversion Functions is to avoid the unnecessary communications of the processors, in order to reduce the communication time of the parallel program.

In this thesis, Conversion Functions Library, Array Information Table Generator and the Parallelizer are the contributions. The output of the project is the parallel version of the user's program. It is ready to compile and execute on a heterogeneous environment with an optimum execution scheme.

In the heterogeneous test environment, there were 24 processors. Using a communication benchmark (pingpong) and a computation benchmark, the processors were separated into 3 groups.

Two benchmark programs were used as the *user's sequential program*: Hessenberg Reduction and Dongarra's benchmark. Parallel Program Optimizer found the expected fitness value for each benchmark. With these values, Parallel Program Optimizer was executed using hill climbing and Genetic Algorithm. Hill climbing was performed with two different local search methods: Next Gradient and Davis. The results of all these experiments were compared. Hill climbing with Next Gradient local search method was the best.

The benchmarks were parallelized and executed according to the best chromosomes found by Parallel Program Optimizer. The estimated and measured total execution times were compared. The percentage error of Hessenberg Reduction was 3.05% and that of Dongarra's benchmark was 8.92%.

In this thesis, a sequential program, which has some levels and which does calculations on large matrices and vectors, is parallelized by the implemented tool. The levels of a sequential program are parallelized. Each level executes on a number of processors in one of the groups. Since an array may be used in many levels, the arrays can be converted from one decomposition type (used in a previous level) to another type (used in the next level).

They can be transferred from a number of processors p in the previous group to a number of processors q in the next group. If the elements of the array are changed in one of the levels, the old data of the array are not transferred. During the transfer of an array, the necessary function from Conversion Functions Library is used. The order of the levels is not changed. In other words, the levels do not execute concurrently.

To be able to parallelize a program such that the levels can execute concurrently, the tool has to be modified. There may be data dependency among the levels which use the same arrays. The dependency information can be obtained from the tables of Array Information Table Generator. The *gene* structure in Parallel Program Optimizer has to be changed. In addition to the group and number of processors on which the level will execute, the selected decomposition, the communication and computation times, the gene will have an order and a precedence for a level. The levels which can execute concurrently (i.e. there is no data dependency among them) will have the same pre-defined order. The precedence will be determined by Parallel Program Optimizer. It is the execution order of the levels which will execute on the same group of processors and which have the same order. Here, the number of processor groups is a limiting factor. The number of levels, which can execute concurrently, is limited by the number of processor groups. The level number, the order, the group of processors and the precedence may be stored in a table. The new case will also change the fitness function of the Parallel Program Optimizer. To estimate the total execution time of the parallel program, the execution time (the sum of communication and computation times) of non-concurrent levels will be added. For the concurrent levels, the maximum execution time of these levels will be added. Thus, an other version of the tool can be designed.

APPENDIX A:

A.1. Horizontal Initialization Function for Matrices

Algorithm 1 Horizontal Initialization Function for Matrices

```

1: procedure void mHorizontalInitialization(int originalMyid, int prevGroup, int p, int
   nextGroup, int q, int SIZE, long **matA, long **matB)
2: /* matA and matB refer to the same matrix. */
3: Using originalMyid, myGroup and myid (in this group) are calculated.
4: The sender constant (sc) and receiver constant (rc) are set.
5: noOfBlocks is set to ((int) (SIZE/q)).
6: blocklength is set to (SIZE).
7: stride is set to SIZE.
8: The initType is created by MPI_Type_vector and committed by MPI_Type_commit.
9: /* SENDING DATA */
10: if myid equals root AND myGroup equals prevGroup then
11:   if prevGroup equals nextGroup /* Send excluding root, i begins from 1 */ then
12:     for all  $i$  such that  $1 \leq i < q$  do
13:       Beginning from matA[i*noOfBlocks][0] address, the data in initType type is sent
       to processor (i+rc).
14:     end for
15:     /* Copying initial data for root */
16:     for all  $x$  such that  $0 \leq x < noOfBlocks$  do
17:       for all  $y$  such that  $0 \leq y < SIZE$  do
18:         matA[x][y] is copied to matB[x][y].
19:       end for
20:     end for
21:   else
22:     /* Send including root, i begins from 0 */
23:     for all  $i$  such that  $0 \leq i < q$  do
24:       Beginning from matA[i*noOfBlocks][0] address, the data in initType type is sent
       to processor (i+rc).

```

```

25:     end for
26: end if
27: else if myid is less than q AND myGroup equals nextGroup /* RECEIVING DATA */
    then
28:     Beginning from matB[0][0] address, the data in initType type is received from pro-
        cessor (root+sc).
29: end if
30: Optional: If the comments are removed, the final data on matB can be displayed.
31: initType is freed by MPI_Type_free().
32: end procedure

```

A.2. Horizontal to Horizontal Conversion Function for Matrices

Algorithm 2 Horizontal to Horizontal Conversion Function for Matrices

```

1: procedure void mHorizontalToHorizontal(int originalMyid, int prevGroup, int p, int
    nextGroup, int q, int SIZE, long **matB, long **matC)
2: Using originalMyid, myGroup and myid (in this group) are calculated.
3: The sender constant (sc) and receiver constant (rc) are set.
4: p_float is set to ((float) p).
5: q_float is set to ((float) q).
6: noOfPieces_sender is set to ((int) ceil(q_float / p_float)). The sender processors will
    divide their data into "noOfPieces_sender" pieces.
7: noOfPieces_receiver is set to ((int) ceil(p_float / q_float)). The receiver processors will
    receive "noOfPieces_receiver" pieces of data.
8: noOfBlocks is set to ((int) (SIZE/p) / noOfPieces_sender).
9: blocklength is set to ((int) SIZE).
10: /* SENDING DATA */
11: if myid is less than p AND myGroup equals prevGroup then
12:     for all  $n$  such that  $0 \leq n < noOfPieces\_sender$  do
13:          $j$  is set to floor(myid * ((float) (q_float/p_float))) + n.
14:         if (NOT (myid equals j AND prevGroup equals nextGroup)) then
15:             Beginning from matB[noOfBlocks*n][0] address, the (noOfBlocks*blocklength)
                elements in type MPI_LONG are sent to processor (j+rc).

```

```

16:     end if
17: end for
18: end if
19: /* RECEIVING DATA */
20: if myid is less than q AND myGroup equals nextGroup then
21:      $i$  is set to  $(\text{floor}(\text{myid} / ((\text{float}) (q\_float/p\_float))))$ .
22:     for all  $n$  such that  $0 \leq n < noOfPieces\_receiver$  do
23:         if  $p$  is greater than  $q$  then
24:              $j$  is set to  $i + n$ 
25:         else
26:              $j$  is set to  $i$ 
27:         end if
28:         if myid equals  $j$  AND prevGroup equals nextGroup /* Data are copied (instead of
send-receive). */ then
29:             for all  $x$  such that  $0 \leq x < noOfBlocks$  do
30:                 for all  $y$  such that  $0 \leq y < blocklength$  do
31:                     matB[ $x + (noOfBlocks * n)$ ][ $y$ ] is copied to matC[ $x + (noOfBlocks * n)$ ][ $y$ ].
32:                 end for
33:             end for
34:         else
35:             Beginning from matC[noOfBlocks*n][0] address, the (noOfBlocks*blocklength)
elements in type MPI_LONG are received from processor ( $j+sc$ ).
36:         end if
37:     end for
38: end if
39: /* matC is used as a temporary receiver matrix. At the end of the conversion, the data
must be collected on matB. Therefore, the data of matC are copied to matB. */
40: for all  $x$  such that  $0 \leq x < (noOfBlocks * noOfPieces\_receiver)$  do
41:     for all  $y$  such that  $0 \leq y < blocklength$  do
42:         matC[ $x$ ][ $y$ ] is copied to matB[ $x$ ][ $y$ ].
43:     end for
44: end for

```

45: Optional: If the comments are removed, the final data on matB can be displayed.

46: **end procedure**

A.3. Horizontal to Vertical Conversion Function for Matrices

Algorithm 3 Horizontal to Vertical Conversion Function for Matrices

```

1: procedure void mHorizontalToVertical (int originalMyid, int prevGroup, int p, int nextGroup,
   int q, int SIZE, long **matB, long **matC)
2: Using originalMyid, myGroup and myid (in this group) are calculated.
3: The sender constant (sc) and receiver constant (rc) are set.
4: noOfBlocks is set to (SIZE/p).
5: blocklength is set to (SIZE/q).
6: stride is set to SIZE.
7: The newType is created by MPI_Type_vector and committed by MPI_Type_commit.
8: /* THE MODIFIED BUTTERFLY ALGORITHM */
9: if p is greater than q then
10:   q-to-q Modified Butterfly Algorithm will be applied. Therefore, the original p value
   is stored in "origP" variable and p is equal to q, for a while.
11: end if
12: The number of stages (noOfStages) of the Modified Butterfly Algorithm is set to the
   logarithm of p in base 2.
13: The stage counter (s) is set to 0.
14: while s is less than noOfStages do
15:   2 to the power s is calculated ( $2^s$ ).
16:   if myid is less than p then
17:     Virtual ID (vid) is set to myid.
18:     if vid %  $2^s$  is not equal to 0 then
19:       vid is decremented by (vid %  $2^s$ ).
20:     end if
21:     for all g such that  $0 \leq g < 2^s$  do
22:       if (int)(myid/ $2^s$ ) % 2 equals 0 AND myGroup equals prevGroup /* this process
         is a sender, now */ then
23:         j is equal to (vid +  $2^s$  + g).

```

```

24:         Beginning from matB[0][blocklength*j] address, the data in newType type is
           sent to processor (j+rc).
25:     else if (int)(myid/2s) % 2 is not equal to 0 AND myGroup equals nextGroup /*
           this process is a receiver, now */ then
26:         j is equal to (vid - 2s + g).
27:         Beginning from matC[noOfBlocks*j][0] address, the data in newType type is
           received from processor (j+sc).
28:     end if
29: end for
30: end if
31:     s is incremented by 1.
32: end while
33: The stage counter (s) is set to 0.
34: while s is less than noOfStages do
35:     2 to the power s is calculated (2s).
36:     if myid is less than p then
37:         Virtual ID (vid) is set to myid.
38:         if vid % 2s is not equal to 0 then
39:             vid is decremented by (vid % 2s).
40:         end if
41:         for all g such that  $0 \leq g < 2^s$  do
42:             if (int)(myid/2s) % 2 equals 0 AND myGroup equals prevGroup /* the sender is
               a receiver, now */ then
43:                 j is equal to (vid + 2s + g).
44:                 Beginning from matC[noOfBlocks*j][0] address, the data in newType type is
                   received from processor (j+sc).
45:             else if (int)(myid/2s) % 2 is not equal to 0 AND myGroup equals nextGroup /*
               the receiver is a sender, now */ then
46:                 j is equal to (vid - 2s + g).
47:                 Beginning from matB[0][blocklength*j] address, the data in newType type is
                   sent to processor (j+rc).
48:             end if

```

```

49:     end for
50: end if
51:   s is incremented by 1.
52: end while
53: if origP is greater than q /* if p is greater than q */ then
54:   p is set back to its original value (origP).
55: end if
56: if p is less than q then
57:   if myid is less than p AND myGroup equals prevGroup /* this proccess is a sender */
   then
58:     for all  $g$  such that  $p \leq g < q$  do
59:       Beginning from matB[0][blocklength*g] address, the data in newType type is
       sent to processor (g+rc).
60:     end for
61:   else if myid is greater than or equal to p AND myid is less than q AND myGroup
   equals nextGroup /* this proccess is a receiver */ then
62:     for all  $g$  such that  $0 \leq g < p$  do
63:       Beginning from matC[noOfBlocks*g][0] address, the data in newType type is
       received from processor (g+sc).
64:     end for
65:   end if
66: else if p is greater than q then
67:   if myid is less than q AND myGroup equals nextGroup /* this proccess is a receiver
   */ then
68:     for all  $g$  such that  $q \leq g < p$  do
69:       Beginning from matC[noOfBlocks*g][0] address, the data in newType type is
       received from processor (g+sc).
70:     end for
71:   else if myid is greater than or equal to q AND myid is less than p AND myGroup
   equals prevGroup /* this proccess is a sender */ then
72:     for all  $g$  such that  $0 \leq g < q$  do
73:       Beginning from matB[0][blocklength*g] address, the data in newType type is

```

```

        sent to processor (g+rc).
74:     end for
75: end if
76: end if
77: if myid is less than minimum of (p, q) then
78:     if prevGroup equals nextGroup /* Data are copied (instead of send-receive). */ then
79:         for all  $x$  such that  $0 \leq x < noOfBlocks$  do
80:             for all  $y$  such that  $0 \leq y < blocklength$  do
81:                 matB[x][y + (blocklength*myid)] is copied to matC[x + (noOfBlocks*myid)][y].
82:             end for
83:         end for
84:     else
85:         if myGroup equals prevGroup /* this process is a sender */ then
86:             Beginning from matB[0][blocklength*myid] address, the data in newType type
            is sent to processor (myid+rc).
87:         else if myGroup equals nextGroup /* this process is a receiver */ then
88:             Beginning from matC[noOfBlocks*myid][0] address, the data in newType type
            is received from processor (myid+sc).
89:         end if
90:     end if
91: end if
92: /* matC is used as a temporary receiver matrix. At the end of the conversion, the data
    must be collected on matB. Therefore, the data of matC are copied to matB. */
93: for all  $x$  such that  $0 \leq x < SIZE$  do
94:     for all  $y$  such that  $0 \leq y < blocklength$  do
95:         matC[x][y] is copied to matB[x][y].
96:     end for
97: end for
98: Optional: If the comments are removed, the final data on matB can be displayed.
99: newType is freed by MPI_Type_free().
100: end procedure

```

A.4. Horizontal Gather Function for Matrices

Algorithm 4 Horizontal Gather Function for Matrices

```

1: procedure void mHorizontalGather(int originalMyid, int prevGroup, int p, int nextGroup,
   int q, int SIZE, long **matA, long **matB)
2: /* matA and matB refer to the same matrix. */
3: Using originalMyid, myGroup and myid (in this group) are calculated.
4: The sender constant (sc) and receiver constant (rc) are set.
5: noOfBlocks is set to ((int) (SIZE/p)).
6: blocklength is set to (SIZE).
7: stride is set to SIZE.
8: The initType is created by MPI_Type_vector and committed by MPI_Type_commit.
9: /* RECEIVING DATA */
10: if myid equals root AND myGroup equals nextGroup then
11:   if prevGroup equals nextGroup /* Receive excluding root, i begins from 1 */ then
12:     for all  $i$  such that  $1 \leq i < p$  do
13:       Beginning from matB[i*noOfBlocks][0] address, the data in initType type is re-
         ceived from processor (i+sc).
14:     end for
15:     /* Copying data for root */
16:     for all  $x$  such that  $0 \leq x < noOfBlocks$  do
17:       for all  $y$  such that  $0 \leq y < SIZE$  do
18:         matA[x][y] is copied to matB[x][y].
19:       end for
20:     end for
21:   else
22:     /* Receive including root, i begins from 0 */
23:     for all  $i$  such that  $0 \leq i < p$  do
24:       Beginning from matB[i*noOfBlocks][0] address, the data in initType type is re-
         ceived from processor (i+sc).
25:     end for
26:   end if
27: else if myid is less than p AND myGroup equals prevGroup /* SENDING DATA */

```

then

28: Beginning from matA[0][0] address, the data in initType type is sent to processor (root+rc).

29: **end if**

30: Optional: If the comments are removed, the final data on matB can be displayed.

31: initType is freed by MPI_Type_free().

32: **end procedure**



continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	0	mH2R	4	16	0.000019026950	1.125
0	0	mH2R	8	4	0.000025218630	-6.161
0	0	mH2R	8	8	0.000010946290	8.375
0	0	mH2R	8	16	0.000016305320	4.829
0	0	mH2R	16	4	0.000018225770	-1.624
0	0	mH2R	16	8	0.000014354850	26.642
0	0	mH2R	16	16	0.000005509583	4.152
0	0	mH2HB	4	4	0.000023508100	1.387
0	0	mH2HB	4	8	0.000087968040	-3.696
0	0	mH2HB	4	16	0.000076763320	-6.162
0	0	mH2HB	8	4	0.000049669770	1.220
0	0	mH2HB	8	8	0.000044498950	2.258
0	0	mH2HB	8	16	0.000065361660	2.709
0	0	mH2HB	16	4	0.000036151520	0.067
0	0	mH2HB	16	8	0.000061137540	-11.806
0	0	mH2HB	16	16	0.000022062470	7.027
0	0	mH2VB	4	4	0.000044461760	2.785
0	0	mH2VB	4	8	0.000037582100	28.346
0	0	mH2VB	4	16	0.000058042710	50.458
0	0	mH2VB	8	4	0.000038897560	1.577
0	0	mH2VB	8	8	0.000026240940	43.809
0	0	mH2VB	8	16	0.000040648430	94.815
0	0	mH2VB	16	4	0.000029445640	39.958
0	0	mH2VB	16	8	0.000020770570	52.827
0	0	mH2VB	16	16	0.000027248470	98.056
0	0	mH2Rep	4	4	0.000089472360	-1.490
0	0	mH2Rep	4	8	0.000154617600	-3.719
0	0	mH2Rep	4	16	0.000241506600	-18.130
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	0	mH2Rep	8	4	0.000077447100	4.891
0	0	mH2Rep	8	8	0.000109802700	8.458
0	0	mH2Rep	8	16	0.000174771400	10.335
0	0	mH2Rep	16	4	0.000059780080	40.916
0	0	mH2Rep	16	8	0.000087309870	27.753
0	0	mH2Rep	16	16	0.000118079900	75.189
0	0	mV2H	4	4	0.000022167060	6.075
0	0	mV2H	4	8	0.000018952610	12.442
0	0	mV2H	4	16	0.000013442220	46.710
0	0	mV2H	8	4	0.000019299970	8.117
0	0	mV2H	8	8	0.000013501220	12.974
0	0	mV2H	8	16	0.000009855755	77.845
0	0	mV2H	16	4	0.000011887750	99.560
0	0	mV2H	16	8	0.000012646680	11.839
0	0	mV2H	16	16	0.000014164570	-3.044
0	0	mV2V	4	4	0.000000941161	0.890
0	0	mV2V	4	8	0.000027334540	-6.160
0	0	mV2V	4	16	0.000019154400	-5.747
0	0	mV2V	8	4	0.000027820260	-4.948
0	0	mV2V	8	8	0.000000470452	1.240
0	0	mV2V	8	16	0.000019292820	-12.471
0	0	mV2V	16	4	0.000019838860	-4.343
0	0	mV2V	16	8	0.000019633450	-11.215
0	0	mV2V	16	16	0.000000273593	1.369
0	0	mV2R	4	4	0.000033460230	-4.761
0	0	mV2R	4	8	0.000006037640	0.982
0	0	mV2R	4	16	0.000008350654	0.872
0	0	mV2R	8	4	0.000025821140	-5.806
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	0	mV2R	8	8	0.000025123430	-12.327
0	0	mV2R	8	16	0.000015302710	-9.242
0	0	mV2R	16	4	0.000018823520	-0.693
0	0	mV2R	16	8	0.000018660170	-8.872
0	0	mV2R	16	16	0.000012799390	-4.780
0	0	mV2HB	4	4	0.000044519430	2.753
0	0	mV2HB	4	8	0.000076959510	5.474
0	0	mV2HB	4	16	0.000057850550	53.676
0	0	mV2HB	8	4	0.000038990990	1.516
0	0	mV2HB	8	8	0.000053891860	36.926
0	0	mV2HB	8	16	0.000040928230	87.048
0	0	mV2HB	16	4	0.000029197320	49.869
0	0	mV2HB	16	8	0.000040823650	102.138
0	0	mV2HB	16	16	0.000026833330	113.435
0	0	mV2VB	4	4	0.000066859890	-3.654
0	0	mV2VB	4	8	0.000012074910	1.235
0	0	mV2VB	4	16	0.000033764970	-2.097
0	0	mV2VB	8	4	0.000051004430	1.824
0	0	mV2VB	8	8	0.000049801930	-8.561
0	0	mV2VB	8	16	0.000060673730	-4.925
0	0	mV2VB	16	4	0.000037583820	-1.689
0	0	mV2VB	16	8	0.000037152570	-12.765
0	0	mV2VB	16	16	0.000050360670	0.569
0	0	mV2Rep	4	4	0.000089770330	-1.069
0	0	mV2Rep	4	8	0.000154979100	-2.307
0	0	mV2Rep	4	16	0.000242363600	-21.236
0	0	mV2Rep	8	4	0.000077739300	6.316
0	0	mV2Rep	8	8	0.000110123500	12.000
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	0	mV2Rep	8	16	0.000175419300	10.467
0	0	mV2Rep	16	4	0.000060496380	34.663
0	0	mV2Rep	16	8	0.000088127610	21.198
0	0	mV2Rep	16	16	0.000118532300	80.484
0	0	mR2H	4	4	0.000011809350	1.346
0	0	mR2H	4	8	0.000021778200	3.842
0	0	mR2H	4	16	0.000016134940	4.609
0	0	mR2H	8	4	0.000025234150	-6.144
0	0	mR2H	8	8	0.000011029410	6.576
0	0	mR2H	8	16	0.000011693730	17.648
0	0	mR2H	16	4	0.000019652160	-5.139
0	0	mR2H	16	8	0.000018264130	-6.910
0	0	mR2H	16	16	0.000005571561	2.368
0	0	mR2V	4	4	0.000033475630	2.477
0	0	mR2V	4	8	0.000027250140	-0.892
0	0	mR2V	4	16	0.000018861680	2.569
0	0	mR2V	8	4	0.000006254805	1.400
0	0	mR2V	8	8	0.000027748430	3.458
0	0	mR2V	8	16	0.000017937200	2.308
0	0	mR2V	16	4	0.000008751815	1.784
0	0	mR2V	16	8	0.000016730740	2.652
0	0	mR2V	16	16	0.000011658560	4.076
0	0	mR2R	4	4	0.000000618011	1.043
0	0	mR2R	4	8	0.000027229560	-5.181
0	0	mR2R	4	16	0.000030209960	-18.077
0	0	mR2R	8	4	0.000027602990	-3.759
0	0	mR2R	8	8	0.000000342657	0.716
0	0	mR2R	8	16	0.000008292714	-1.917
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	0	mR2R	16	4	0.000019571120	-6.272
0	0	mR2R	16	8	0.000008442371	-1.581
0	0	mR2R	16	16	0.000000172251	0.750
0	0	mR2HB	4	4	0.000025552310	1.408
0	0	mR2HB	4	8	0.000088117980	-3.525
0	0	mR2HB	4	16	0.000065034340	1.950
0	0	mR2HB	8	4	0.000049777010	2.375
0	0	mR2HB	8	8	0.000045628010	2.654
0	0	mR2HB	8	16	0.000047204820	46.770
0	0	mR2HB	16	4	0.000035877400	9.338
0	0	mR2HB	16	8	0.000060936010	-3.261
0	0	mR2HB	16	16	0.000022625440	6.939
0	0	mR2VB	4	4	0.000023411720	2.321
0	0	mR2VB	4	8	0.000049927230	3.180
0	0	mR2VB	4	16	0.000071261040	3.412
0	0	mR2VB	8	4	0.000049712350	-2.795
0	0	mR2VB	8	8	0.000011783600	1.812
0	0	mR2VB	8	16	0.000027990890	2.578
0	0	mR2VB	16	4	0.000035872330	5.640
0	0	mR2VB	16	8	0.000014247130	0.595
0	0	mR2VB	16	16	0.000022035800	9.404
0	0	mR2Rep	4	4	0.000089420370	-1.041
0	0	mR2Rep	4	8	0.000154777600	-3.542
0	0	mR2Rep	4	16	0.000241894000	-19.429
0	0	mR2Rep	8	4	0.000077562260	4.994
0	0	mR2Rep	8	8	0.000110036600	7.247
0	0	mR2Rep	8	16	0.000175028300	10.992
0	0	mR2Rep	16	4	0.000059685360	45.411
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	0	mR2Rep	16	8	0.000086439690	52.028
0	0	mR2Rep	16	16	0.000117405500	94.863
0	0	mHB2H	4	4	0.000000566361	0.848
0	0	mHB2H	4	8	0.000016455130	-2.134
0	0	mHB2H	4	16	0.000019094660	-6.824
0	0	mHB2H	8	4	0.000011486050	0.944
0	0	mHB2H	8	8	0.000000283270	0.743
0	0	mHB2H	8	16	0.000008270501	-2.075
0	0	mHB2H	16	4	0.000011482600	0.973
0	0	mHB2H	16	8	0.000011131010	-1.956
0	0	mHB2H	16	16	0.000000143895	0.774
0	0	mHB2V	4	4	0.000011732840	1.877
0	0	mHB2V	4	8	0.000014059030	1.620
0	0	mHB2V	4	16	0.000012399260	2.402
0	0	mHB2V	8	4	0.000006117328	1.512
0	0	mHB2V	8	8	0.000005914765	1.793
0	0	mHB2V	8	16	0.000007076193	2.338
0	0	mHB2V	16	4	0.000008678847	1.870
0	0	mHB2V	16	8	0.000007102466	2.860
0	0	mHB2V	16	16	0.000005618418	2.934
0	0	mHB2R	4	4	0.000000614116	0.937
0	0	mHB2R	4	8	0.000016498880	-1.818
0	0	mHB2R	4	16	0.000019113930	-6.668
0	0	mHB2R	8	4	0.000011520080	1.258
0	0	mHB2R	8	8	0.000000331706	0.876
0	0	mHB2R	8	16	0.000008287583	-1.895
0	0	mHB2R	16	4	0.000011340980	1.551
0	0	mHB2R	16	8	0.000008430935	-1.496
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	0	mHB2R	16	16	0.000000164930	0.808
0	0	mHB2HB	4	4	0.0000001121719	1.210
0	0	mHB2HB	4	8	0.000066086170	-2.691
0	0	mHB2HB	4	16	0.000076259360	-8.302
0	0	mHB2HB	8	4	0.000022967530	1.130
0	0	mHB2HB	8	8	0.0000001121358	1.315
0	0	mHB2HB	8	16	0.000032973060	-2.382
0	0	mHB2HB	16	4	0.000022623460	1.454
0	0	mHB2HB	16	8	0.000033296830	-0.248
0	0	mHB2HB	16	16	0.000000557488	1.102
0	0	mHB2VB	4	4	0.000023356220	2.094
0	0	mHB2VB	4	8	0.000027976320	2.476
0	0	mHB2VB	4	16	0.000049346830	1.846
0	0	mHB2VB	8	4	0.000012129990	1.862
0	0	mHB2VB	8	8	0.000011739220	1.955
0	0	mHB2VB	8	16	0.000027965960	2.616
0	0	mHB2VB	16	4	0.000017238240	2.409
0	0	mHB2VB	16	8	0.000014174940	1.687
0	0	mHB2VB	16	16	0.000022098550	7.335
0	0	mHB2Rep	4	4	0.000047025610	1.970
0	0	mHB2Rep	4	8	0.000112442900	-3.388
0	0	mHB2Rep	4	16	0.000199924100	-15.756
0	0	mHB2Rep	8	4	0.000024616820	1.756
0	0	mHB2Rep	8	8	0.000047030630	2.144
0	0	mHB2Rep	8	16	0.000112462200	-3.397
0	0	mHB2Rep	16	4	0.000034622660	2.317
0	0	mHB2Rep	16	8	0.000056596790	2.858
0	0	mHB2Rep	16	16	0.000089528900	-1.782
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	0	mVB2H	4	4	0.000011675320	1.477
0	0	mVB2H	4	8	0.000016650150	-1.092
0	0	mVB2H	4	16	0.000019342710	-4.514
0	0	mVB2H	8	4	0.000014052140	-0.440
0	0	mVB2H	8	8	0.000012546190	0.209
0	0	mVB2H	8	16	0.000012932920	1.713
0	0	mVB2H	16	4	0.000014050700	-0.402
0	0	mVB2H	16	8	0.000007011724	1.014
0	0	mVB2H	16	16	0.000006295954	0.534
0	0	mVB2V	4	4	0.000022818020	2.061
0	0	mVB2V	4	8	0.000016781830	2.103
0	0	mVB2V	4	16	0.000013644580	5.314
0	0	mVB2V	8	4	0.000000664927	0.922
0	0	mVB2V	8	8	0.000017083090	3.052
0	0	mVB2V	8	16	0.000016386840	1.400
0	0	mVB2V	16	4	0.000000665052	0.922
0	0	mVB2V	16	8	0.000017083360	3.067
0	0	mVB2V	16	16	0.000005780800	2.124
0	0	mVB2R	4	4	0.000000645499	1.005
0	0	mVB2R	4	8	0.000017045690	2.037
0	0	mVB2R	4	16	0.000013644660	6.780
0	0	mVB2R	8	4	0.000017009270	2.038
0	0	mVB2R	8	8	0.000000361885	0.699
0	0	mVB2R	8	16	0.000002905903	0.921
0	0	mVB2R	16	4	0.000017007400	2.041
0	0	mVB2R	16	8	0.000000363244	0.674
0	0	mVB2R	16	16	0.000000181959	0.739
0	0	mVB2HB	4	4	0.000023353160	1.909
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	0	mVB2HB	4	8	0.000066460510	-0.711
0	0	mVB2HB	4	16	0.000076128500	3.525
0	0	mVB2HB	8	4	0.000027912120	2.100
0	0	mVB2HB	8	8	0.000049798520	2.980
0	0	mVB2HB	8	16	0.000051619980	4.598
0	0	mVB2HB	16	4	0.000027914450	2.065
0	0	mVB2HB	16	8	0.000027912950	2.160
0	0	mVB2HB	16	16	0.000024935220	1.480
0	0	mVB2VB	4	4	0.000001286156	1.463
0	0	mVB2VB	4	8	0.000034086840	2.761
0	0	mVB2VB	4	16	0.000055805280	-0.858
0	0	mVB2VB	8	4	0.000033990310	3.002
0	0	mVB2VB	8	8	0.000000714943	0.949
0	0	mVB2VB	8	16	0.000011586520	1.540
0	0	mVB2VB	16	4	0.000033989230	2.972
0	0	mVB2VB	16	8	0.000000714753	0.954
0	0	mVB2VB	16	16	0.000000710545	1.210
0	0	mVB2Rep	4	4	0.000046688490	2.838
0	0	mVB2Rep	4	8	0.000133365200	-3.113
0	0	mVB2Rep	4	16	0.000308222500	-16.235
0	0	mVB2Rep	8	4	0.000056233760	-0.637
0	0	mVB2Rep	8	8	0.000099912600	6.163
0	0	mVB2Rep	8	16	0.000207813300	-7.244
0	0	mVB2Rep	16	4	0.000056144670	2.060
0	0	mVB2Rep	16	8	0.000056224330	-0.327
0	0	mVB2Rep	16	16	0.000100034900	1.605
0	0	mRep2H	4	4	0.000000564286	0.836
0	0	mRep2H	4	8	0.000005744853	0.868
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	0	mRep2H	4	16	0.000008201897	0.859
0	0	mRep2H	8	4	0.000000565139	0.824
0	0	mRep2H	8	8	0.000000284433	0.697
0	0	mRep2H	8	16	0.000002877857	0.884
0	0	mRep2H	16	4	0.000000565136	0.825
0	0	mRep2H	16	8	0.000000284319	0.700
0	0	mRep2H	16	16	0.000000152245	0.891
0	0	mRep2V	4	4	0.000000647936	1.198
0	0	mRep2V	4	8	0.000005806165	1.493
0	0	mRep2V	4	16	0.000008260203	1.699
0	0	mRep2V	8	4	0.000000647866	1.202
0	0	mRep2V	8	8	0.000000362085	1.173
0	0	mRep2V	8	16	0.000002938622	1.607
0	0	mRep2V	16	4	0.000000648092	1.200
0	0	mRep2V	16	8	0.000000362178	1.172
0	0	mRep2V	16	16	0.000000218149	1.301
0	0	mRep2R	4	4	0.000000607498	0.889
0	0	mRep2R	4	8	0.000005788091	1.061
0	0	mRep2R	4	16	0.000008223917	0.951
0	0	mRep2R	8	4	0.000000606877	0.901
0	0	mRep2R	8	8	0.000000329813	0.854
0	0	mRep2R	8	16	0.000002896860	0.977
0	0	mRep2R	16	4	0.000000606521	0.900
0	0	mRep2R	16	8	0.000000329819	0.853
0	0	mRep2R	16	16	0.000000166894	0.761
0	0	mRep2HB	4	4	0.000001139875	1.107
0	0	mRep2HB	4	8	0.000022967460	1.196
0	0	mRep2HB	4	16	0.000033104640	-1.826
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	0	mRep2HB	8	4	0.000001141157	1.089
0	0	mRep2HB	8	8	0.000001138933	1.177
0	0	mRep2HB	8	16	0.000011477720	1.170
0	0	mRep2HB	16	4	0.000001139776	1.112
0	0	mRep2HB	16	8	0.000001140290	1.159
0	0	mRep2HB	16	16	0.000000567363	1.006
0	0	mRep2VB	4	4	0.000001202846	1.420
0	0	mRep2VB	4	8	0.000011574830	1.356
0	0	mRep2VB	4	16	0.000033260330	-2.023
0	0	mRep2VB	8	4	0.000001201208	1.439
0	0	mRep2VB	8	8	0.000000652428	1.286
0	0	mRep2VB	8	16	0.000011556110	1.609
0	0	mRep2VB	16	4	0.000001202204	1.421
0	0	mRep2VB	16	8	0.000000652770	1.278
0	0	mRep2VB	16	16	0.000000650341	1.359
0	0	mRep2Rep	4	4	0.000002257835	1.818
0	0	mRep2Rep	4	8	0.000045989270	1.033
0	0	mRep2Rep	4	16	0.000133684900	-7.237
0	0	mRep2Rep	8	4	0.000002257189	1.854
0	0	mRep2Rep	8	8	0.000002263531	1.943
0	0	mRep2Rep	8	16	0.000046024440	3.067
0	0	mRep2Rep	16	4	0.000002255242	1.887
0	0	mRep2Rep	16	8	0.000002262230	1.956
0	0	mRep2Rep	16	16	0.000002277161	1.923
0	0	mNone2None	1	1	0.000000001914	0.554
0	0	mHinit	1	4	0.000032814360	-2.165
0	0	mHinit	1	8	0.000037584540	0.173
0	0	mHinit	1	16	0.000038430610	41.511
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	0	mVinit	1	4	0.000032872440	-2.122
0	0	mVinit	1	8	0.000037607560	0.353
0	0	mVinit	1	16	0.000038331250	45.288
0	0	mRinit	1	4	0.000032859800	-2.253
0	0	mRinit	1	8	0.000037570120	0.523
0	0	mRinit	1	16	0.000037975120	52.795
0	0	mHBinit	1	4	0.000066105700	-5.549
0	0	mHBinit	1	8	0.000153577000	-18.491
0	0	mHBinit	1	16	0.000162675200	-19.474
0	0	mVBinit	1	4	0.000066176030	-5.384
0	0	mVBinit	1	8	0.000076256140	-8.993
0	0	mVBinit	1	16	0.000162972100	-21.389
0	0	mRepInit	1	4	0.000132443700	-6.817
0	0	mRepInit	1	8	0.000307969700	-23.751
0	0	mRepInit	1	16	0.000659248400	-57.347
0	0	mHGather	4	1	0.000032872900	11.856
0	0	mHGather	8	1	0.000038359640	2.504
0	0	mHGather	16	1	0.000040257600	22.388
0	0	mVGather	4	1	0.000032373730	30.506
0	0	mVGather	8	1	0.000038474840	4.164
0	0	mVGather	16	1	0.000040934870	13.990
0	0	mRGather	4	1	0.000032780520	16.698
0	0	mRGather	8	1	0.000038430540	3.275
0	0	mRGather	16	1	0.000040775080	12.214
0	0	mHBGather	4	1	0.000032638790	19.127
0	0	mHBGather	8	1	0.000038355480	2.626
0	0	mHBGather	16	1	0.000040340240	20.324
0	0	mVBGather	4	1	0.000032919410	12.471
continued on next page						

continued from previous page

Previous Group	Next Group	Option	p	q	Slope	Intercept
0	0	mVBGather	8	1	0.000038428560	3.280
0	0	mVBGather	16	1	0.000040560050	17.462
0	0	mRepGather	4	1	0.000000000338	0.626
0	0	mRepGather	8	1	0.000000000000	0.619
0	0	mRepGather	16	1	0.000000000003	0.613
0	1	mH2H	4	4	0.000011871620	2.875
0	1	mH2H	8	4	0.000011397330	15.015
0	1	mH2H	16	4	0.000012521020	55.795
0	1	mH2V	4	4	0.000023709360	30.077
0	1	mH2V	8	4	0.000017735840	167.220
0	1	mH2V	16	4	0.000008242723	449.991
0	1	mH2R	4	4	0.000011602930	14.216
0	1	mH2R	8	4	0.000013372740	59.813
0	1	mH2R	16	4	0.000012459140	152.658
0	1	mH2HB	4	4	0.000023357410	17.590
0	1	mH2HB	8	4	0.000027508700	44.081
0	1	mH2HB	16	4	0.000022617150	191.459
0	1	mH2VB	4	4	0.000048864270	12.836
0	1	mH2VB	8	4	0.000037723220	151.328
0	1	mH2VB	16	4	0.000028722340	346.220
0	1	mH2Rep	4	4	0.000098390720	8.611
0	1	mH2Rep	8	4	0.000081695090	75.607
0	1	mH2Rep	16	4	0.000060766520	354.829
0	1	mV2H	4	4	0.000024429510	19.104
0	1	mV2H	8	4	0.000019622800	129.144
0	1	mV2H	16	4	0.000008112808	458.536
0	1	mV2V	4	4	0.000011810610	4.454
0	1	mV2V	8	4	0.000013012940	-2.528

continued on next page

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	1	mV2V	16	4	0.000010107050	91.468
0	1	mV2R	4	4	0.000022594200	2.882
0	1	mV2R	8	4	0.000015432190	73.652
0	1	mV2R	16	4	0.000014767990	132.004
0	1	mV2HB	4	4	0.000049361110	8.754
0	1	mV2HB	8	4	0.000038202720	145.457
0	1	mV2HB	16	4	0.000030523330	340.922
0	1	mV2VB	4	4	0.000045061190	2.040
0	1	mV2VB	8	4	0.000034143870	33.713
0	1	mV2VB	16	4	0.000029374020	143.177
0	1	mV2Rep	4	4	0.000098784720	5.137
0	1	mV2Rep	8	4	0.000081304830	92.262
0	1	mV2Rep	16	4	0.000059884580	354.098
0	1	mR2H	4	4	0.000012036330	8.261
0	1	mR2H	8	4	0.000012101530	92.271
0	1	mR2H	16	4	0.000011356590	71.960
0	1	mR2V	4	4	0.000016456890	5.790
0	1	mR2V	8	4	0.000012006900	13.814
0	1	mR2V	16	4	0.000011850250	52.943
0	1	mR2R	4	4	0.000011902990	5.628
0	1	mR2R	8	4	0.000012229050	13.760
0	1	mR2R	16	4	0.000010592630	69.036
0	1	mR2HB	4	4	0.000023791620	12.767
0	1	mR2HB	8	4	0.000028319190	37.719
0	1	mR2HB	16	4	0.000023312920	209.096
0	1	mR2VB	4	4	0.000023967800	6.280
0	1	mR2VB	8	4	0.000038222950	9.863
0	1	mR2VB	16	4	0.000025611600	197.092
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	1	mR2Rep	4	4	0.000098600750	5.806
0	1	mR2Rep	8	4	0.000081743720	77.517
0	1	mR2Rep	16	4	0.000060886860	320.604
0	1	mHB2H	4	4	0.000011893810	4.624
0	1	mHB2H	8	4	0.000011781340	2.731
0	1	mHB2H	16	4	0.000011550070	6.237
0	1	mHB2V	4	4	0.000011723970	18.078
0	1	mHB2V	8	4	0.000011919970	6.765
0	1	mHB2V	16	4	0.000010074050	87.485
0	1	mHB2R	4	4	0.000011790500	4.590
0	1	mHB2R	8	4	0.000011764460	5.439
0	1	mHB2R	16	4	0.000012422430	3.690
0	1	mHB2HB	4	4	0.000024201200	-1.599
0	1	mHB2HB	8	4	0.000023171480	11.315
0	1	mHB2HB	16	4	0.000024074520	1.943
0	1	mHB2VB	4	4	0.000023322140	13.779
0	1	mHB2VB	8	4	0.000024688930	-2.309
0	1	mHB2VB	16	4	0.000025318870	25.068
0	1	mHB2Rep	4	4	0.000047440730	23.703
0	1	mHB2Rep	8	4	0.000048400950	10.192
0	1	mHB2Rep	16	4	0.000047850500	53.900
0	1	mVB2H	4	4	0.000011770950	15.213
0	1	mVB2H	8	4	0.000016593960	39.016
0	1	mVB2H	16	4	0.000016574530	36.048
0	1	mVB2V	4	4	0.000012340880	-1.394
0	1	mVB2V	8	4	0.000011579750	8.187
0	1	mVB2V	16	4	0.000012119760	1.460
0	1	mVB2R	4	4	0.000011434780	15.862
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	1	mVB2R	8	4	0.000012008440	6.499
0	1	mVB2R	16	4	0.000012395560	5.868
0	1	mVB2HB	4	4	0.000023972270	7.854
0	1	mVB2HB	8	4	0.000034486140	12.878
0	1	mVB2HB	16	4	0.000033121320	34.260
0	1	mVB2VB	4	4	0.000024086250	-0.130
0	1	mVB2VB	8	4	0.000024146590	9.434
0	1	mVB2VB	16	4	0.000024655320	5.984
0	1	mVB2Rep	4	4	0.000047702360	26.690
0	1	mVB2Rep	8	4	0.000066040110	43.563
0	1	mVB2Rep	16	4	0.000066727100	27.394
0	1	mRep2H	4	4	0.000011998030	4.871
0	1	mRep2H	8	4	0.000012009010	1.006
0	1	mRep2H	16	4	0.000011761870	5.287
0	1	mRep2V	4	4	0.000011667930	7.588
0	1	mRep2V	8	4	0.000011739660	7.790
0	1	mRep2V	16	4	0.000011621010	6.536
0	1	mRep2R	4	4	0.000011681220	6.274
0	1	mRep2R	8	4	0.000011754800	3.807
0	1	mRep2R	16	4	0.000012048630	0.666
0	1	mRep2HB	4	4	0.000024194480	-2.609
0	1	mRep2HB	8	4	0.000024215170	6.251
0	1	mRep2HB	16	4	0.000023957020	1.172
0	1	mRep2VB	4	4	0.000023660850	8.435
0	1	mRep2VB	8	4	0.000023867850	1.536
0	1	mRep2VB	16	4	0.000023783560	4.291
0	1	mRep2Rep	4	4	0.000047838760	10.978
0	1	mRep2Rep	8	4	0.000047957170	-1.891
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	1	mRep2Rep	16	4	0.000047822840	1.720
0	1	mNone2None	1	1	0.000045378640	1.172
0	1	mHinit	1	4	0.000043740420	-4.815
0	1	mVinit	1	4	0.000043859510	-5.062
0	1	mRinit	1	4	0.000043805220	-5.040
0	1	mHBinit	1	4	0.000087985870	-7.882
0	1	mVBinit	1	4	0.000088066350	-7.566
0	1	mRepInit	1	4	0.000176100300	-8.268
0	1	mHGather	4	1	0.000043759900	3.977
0	1	mHGather	8	1	0.000043804500	9.585
0	1	mHGather	16	1	0.000037576120	182.496
0	1	mVGather	4	1	0.000043799430	4.080
0	1	mVGather	8	1	0.000043866860	10.093
0	1	mVGather	16	1	0.000038582920	172.398
0	1	mRGather	4	1	0.000043769320	4.620
0	1	mRGather	8	1	0.000043864740	9.031
0	1	mRGather	16	1	0.000039433460	140.334
0	1	mHBGather	4	1	0.000043741710	4.246
0	1	mHBGather	8	1	0.000043802660	9.526
0	1	mHBGather	16	1	0.000038780840	163.840
0	1	mVBGather	4	1	0.000043767570	4.353
0	1	mVBGather	8	1	0.000043841200	10.034
0	1	mVBGather	16	1	0.000038960550	162.761
0	1	mRepGather	4	1	0.000045350870	1.616
0	1	mRepGather	8	1	0.000045364930	1.376
0	1	mRepGather	16	1	0.000045363690	1.393
0	2	mH2H	4	4	0.000058283660	2.932
0	2	mH2H	8	4	0.000057570080	14.391
continued on next page						

[illegible]

continued from previous page

Previous Group	Next Group	Option	p	q	Slope	Intercept
0	2	mV2VB	4	4	0.000121622100	7.543
0	2	mV2VB	8	4	0.000117690300	29.658
0	2	mV2VB	16	4	0.000111670300	180.106
0	2	mV2Rep	4	4	0.000240688900	31.127
0	2	mV2Rep	8	4	0.000237950400	53.165
0	2	mV2Rep	16	4	0.000226409800	280.257
0	2	mR2H	4	4	0.000058028380	6.549
0	2	mR2H	8	4	0.000054807420	100.431
0	2	mR2H	16	4	0.000054307130	101.742
0	2	mR2V	4	4	0.000059056120	3.145
0	2	mR2V	8	4	0.000058027010	5.769
0	2	mR2V	16	4	0.000054354580	101.585
0	2	mR2R	4	4	0.000058275570	4.632
0	2	mR2R	8	4	0.000057779740	13.645
0	2	mR2R	16	4	0.000054357580	100.673
0	2	mR2HB	4	4	0.000115354500	23.305
0	2	mR2HB	8	4	0.000116764600	12.910
0	2	mR2HB	16	4	0.000109655600	180.843
0	2	mR2VB	4	4	0.000115987900	11.701
0	2	mR2VB	8	4	0.000118661500	19.011
0	2	mR2VB	16	4	0.000110905200	181.427
0	2	mR2Rep	4	4	0.000241177600	17.667
0	2	mR2Rep	8	4	0.000238634400	33.438
0	2	mR2Rep	16	4	0.000226725700	258.622
0	2	mHB2H	4	4	0.000058223900	4.320
0	2	mHB2H	8	4	0.000058267600	4.046
0	2	mHB2H	16	4	0.000058260470	4.768
0	2	mHB2V	4	4	0.000058058590	6.562

continued on next page

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	2	mHB2V	8	4	0.000057841060	14.089
0	2	mHB2V	16	4	0.000054488470	96.942
0	2	mHB2R	4	4	0.000058398280	5.077
0	2	mHB2R	8	4	0.000058255400	4.747
0	2	mHB2R	16	4	0.000058232450	4.836
0	2	mHB2HB	4	4	0.000116432200	4.725
0	2	mHB2HB	8	4	0.000116433600	4.943
0	2	mHB2HB	16	4	0.000115972300	7.024
0	2	mHB2VB	4	4	0.000116023800	9.739
0	2	mHB2VB	8	4	0.000115842100	8.598
0	2	mHB2VB	16	4	0.000114995300	29.066
0	2	mHB2Rep	4	4	0.000231783600	8.735
0	2	mHB2Rep	8	4	0.000231848400	6.975
0	2	mHB2Rep	16	4	0.000230930300	23.857
0	2	mVB2H	4	4	0.000058139410	5.855
0	2	mVB2H	8	4	0.000055904050	90.201
0	2	mVB2H	16	4	0.000056286860	79.675
0	2	mVB2V	4	4	0.000058234920	4.563
0	2	mVB2V	8	4	0.000058198920	4.950
0	2	mVB2V	16	4	0.000058367240	2.439
0	2	mVB2R	4	4	0.000058259490	4.830
0	2	mVB2R	8	4	0.000058064280	6.875
0	2	mVB2R	16	4	0.000057949680	8.723
0	2	mVB2HB	4	4	0.000116166900	6.057
0	2	mVB2HB	8	4	0.000117857100	16.383
0	2	mVB2HB	16	4	0.000117267100	29.294
0	2	mVB2VB	4	4	0.000116468700	5.773
0	2	mVB2VB	8	4	0.000115912700	8.142
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	2	mVB2VB	16	4	0.000115834900	9.082
0	2	mVB2Rep	4	4	0.000231920700	7.454
0	2	mVB2Rep	8	4	0.000235814400	16.121
0	2	mVB2Rep	16	4	0.000236091300	11.644
0	2	mRep2H	4	4	0.000058133670	5.248
0	2	mRep2H	8	4	0.000058268710	3.442
0	2	mRep2H	16	4	0.000058166250	5.522
0	2	mRep2V	4	4	0.000058265200	5.509
0	2	mRep2V	8	4	0.000058144700	8.628
0	2	mRep2V	16	4	0.000058327760	3.634
0	2	mRep2R	4	4	0.000058220730	4.839
0	2	mRep2R	8	4	0.000058296260	4.762
0	2	mRep2R	16	4	0.000058218930	4.855
0	2	mRep2HB	4	4	0.000116512000	3.004
0	2	mRep2HB	8	4	0.000116240800	7.447
0	2	mRep2HB	16	4	0.000116239600	6.744
0	2	mRep2VB	4	4	0.000116286200	8.291
0	2	mRep2VB	8	4	0.000116294500	8.750
0	2	mRep2VB	16	4	0.000116526700	3.635
0	2	mRep2Rep	4	4	0.000232701100	6.282
0	2	mRep2Rep	8	4	0.000232486300	10.740
0	2	mRep2Rep	16	4	0.000232593400	6.078
0	2	mNone2None	1	1	0.000063261860	-4.082
0	2	mHinit	1	4	0.000061693560	2.095
0	2	mVinit	1	4	0.000062059290	1.036
0	2	mRinit	1	4	0.000061882630	2.789
0	2	mHBinit	1	4	0.000123774200	1.937
0	2	mVBinit	1	4	0.000123901800	1.000
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	2	mRepInit	1	4	0.000249872900	-25.522
0	2	mHGather	4	1	0.000061884610	5.039
0	2	mHGather	8	1	0.000062432120	11.226
0	2	mHGather	16	1	0.000059602000	141.356
0	2	mVGather	4	1	0.000061620170	6.835
0	2	mVGather	8	1	0.000062082460	19.325
0	2	mVGather	16	1	0.000060578410	116.930
0	2	mRGather	4	1	0.000061742800	8.765
0	2	mRGather	8	1	0.000062269440	13.291
0	2	mRGather	16	1	0.000059460060	130.352
0	2	mHBGather	4	1	0.000061681270	10.942
0	2	mHBGather	8	1	0.000062216700	14.867
0	2	mHBGather	16	1	0.000058963310	153.464
0	2	mVBGather	4	1	0.000061662480	10.001
0	2	mVBGather	8	1	0.000062352580	13.139
0	2	mVBGather	16	1	0.000059486080	134.552
0	2	mRepGather	4	1	0.000063043300	-3.150
0	2	mRepGather	8	1	0.000062790650	0.419
0	2	mRepGather	16	1	0.000063113030	-2.686
1	0	mH2H	4	4	0.000011398690	1.120
1	0	mH2H	4	8	0.000011017770	1.124
1	0	mH2H	4	16	0.000010844840	1.089
1	0	mH2V	4	4	0.000025040500	-6.138
1	0	mH2V	4	8	0.000017808190	-1.204
1	0	mH2V	4	16	0.000012100660	57.953
1	0	mH2R	4	4	0.000011303030	1.852
1	0	mH2R	4	8	0.000016545730	-1.956
1	0	mH2R	4	16	0.000010862920	1.281
continued on next page						

continued from previous page

Previous Group	Next Group	Option	p	q	Slope	Intercept
1	0	mH2HB	4	4	0.000022497920	3.067
1	0	mH2HB	4	8	0.000065601930	-0.298
1	0	mH2HB	4	16	0.000043625630	-1.590
1	0	mH2VB	4	4	0.000049330190	2.476
1	0	mH2VB	4	8	0.000035021190	10.452
1	0	mH2VB	4	16	0.000054692780	53.491
1	0	mH2Rep	4	4	0.000098615580	1.137
1	0	mH2Rep	4	8	0.000141284100	-0.611
1	0	mH2Rep	4	16	0.000227248500	-6.409
1	0	mV2H	4	4	0.000025015980	-6.648
1	0	mV2H	4	8	0.000017715590	-0.613
1	0	mV2H	4	16	0.000012290250	51.367
1	0	mV2V	4	4	0.000011499640	1.615
1	0	mV2V	4	8	0.000011079150	1.916
1	0	mV2V	4	16	0.000010915800	1.879
1	0	mV2R	4	4	0.000022567200	2.101
1	0	mV2R	4	8	0.000011310570	1.321
1	0	mV2R	4	16	0.000010990040	1.085
1	0	mV2HB	4	4	0.000049302520	2.830
1	0	mV2HB	4	8	0.000070530820	3.095
1	0	mV2HB	4	16	0.000054934840	46.776
1	0	mV2VB	4	4	0.000045231860	1.560
1	0	mV2VB	4	8	0.000022712400	0.735
1	0	mV2VB	4	16	0.000044404150	-2.872
1	0	mV2Rep	4	4	0.000098839520	1.156
1	0	mV2Rep	4	8	0.000141674800	-2.852
1	0	mV2Rep	4	16	0.000227845300	-9.483
1	0	mR2H	4	4	0.000011289920	1.725

continued on next page

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
1	0	mR2H	4	8	0.000016506110	-1.848
1	0	mR2H	4	16	0.000013122620	9.511
1	0	mR2V	4	4	0.000016669990	1.909
1	0	mR2V	4	8	0.000013722760	0.873
1	0	mR2V	4	16	0.000012210380	1.543
1	0	mR2R	4	4	0.000011437620	1.558
1	0	mR2R	4	8	0.000011068980	1.336
1	0	mR2R	4	16	0.000010868960	1.239
1	0	mR2HB	4	4	0.000022574160	2.042
1	0	mR2HB	4	8	0.000065685200	-0.293
1	0	mR2HB	4	16	0.000054010820	0.678
1	0	mR2VB	4	4	0.000022619810	2.442
1	0	mR2VB	4	8	0.000032669870	1.794
1	0	mR2VB	4	16	0.000054090070	1.067
1	0	mR2Rep	4	4	0.000098729650	1.184
1	0	mR2Rep	4	8	0.000141474400	-1.822
1	0	mR2Rep	4	16	0.000227321600	-5.467
1	0	mHB2H	4	4	0.000011398070	1.202
1	0	mHB2H	4	8	0.000011016820	1.178
1	0	mHB2H	4	16	0.000010843790	1.138
1	0	mHB2V	4	4	0.000011347400	1.984
1	0	mHB2V	4	8	0.000013755550	0.140
1	0	mHB2V	4	16	0.000012245610	0.713
1	0	mHB2R	4	4	0.000011425400	1.537
1	0	mHB2R	4	8	0.000011056440	1.373
1	0	mHB2R	4	16	0.000010864280	1.240
1	0	mHB2HB	4	4	0.000022790100	1.376
1	0	mHB2HB	4	8	0.000044463230	-1.341
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
1	0	mHB2HB	4	16	0.000043694110	-2.242
1	0	mHB2VB	4	4	0.000022600520	2.354
1	0	mHB2VB	4	8	0.000027309380	2.036
1	0	mHB2VB	4	16	0.000048729600	1.159
1	0	mHB2Rep	4	4	0.000045114460	2.063
1	0	mHB2Rep	4	8	0.000110166100	-5.724
1	0	mHB2Rep	4	16	0.000196939100	-15.432
1	0	mVB2H	4	4	0.000011290810	1.666
1	0	mVB2H	4	8	0.000016553100	-2.723
1	0	mVB2H	4	16	0.000019270210	-4.711
1	0	mVB2V	4	4	0.000011476470	1.578
1	0	mVB2V	4	8	0.000011068990	1.796
1	0	mVB2V	4	16	0.000010908780	1.823
1	0	mVB2R	4	4	0.000011438900	1.547
1	0	mVB2R	4	8	0.000011068120	1.354
1	0	mVB2R	4	16	0.000010870450	1.138
1	0	mVB2HB	4	4	0.000022577970	1.989
1	0	mVB2HB	4	8	0.000065685330	-0.018
1	0	mVB2HB	4	16	0.000075860050	4.112
1	0	mVB2VB	4	4	0.000022869400	2.057
1	0	mVB2VB	4	8	0.000022229820	0.809
1	0	mVB2VB	4	16	0.000043749880	-1.212
1	0	mVB2Rep	4	4	0.000045178430	2.403
1	0	mVB2Rep	4	8	0.000132055500	-4.623
1	0	mVB2Rep	4	16	0.000306394200	-17.348
1	0	mRep2H	4	4	0.000011395120	1.209
1	0	mRep2H	4	8	0.000011015420	1.143
1	0	mRep2H	4	16	0.000010845000	1.092
continued on next page						

continued from previous page

Previous Group	Next Group	Option	p	q	Slope	Intercept
1	0	mRep2V	4	4	0.000011473080	1.629
1	0	mRep2V	4	8	0.000011072470	1.745
1	0	mRep2V	4	16	0.000010908360	1.791
1	0	mRep2R	4	4	0.000011428880	1.478
1	0	mRep2R	4	8	0.000011056050	1.343
1	0	mRep2R	4	16	0.000010865670	1.202
1	0	mRep2HB	4	4	0.000022790530	1.372
1	0	mRep2HB	4	8	0.000044476470	-1.463
1	0	mRep2HB	4	16	0.000043567370	-1.063
1	0	mRep2VB	4	4	0.000022851420	1.952
1	0	mRep2VB	4	8	0.000022209700	0.742
1	0	mRep2VB	4	16	0.000043721380	-1.118
1	0	mRep2Rep	4	4	0.000045570970	1.746
1	0	mRep2Rep	4	8	0.000089124480	-2.448
1	0	mRep2Rep	4	16	0.000176300700	-10.730
1	0	mNone2None	1	1	0.000045561750	1.192
1	0	mHinit	1	4	0.000043270880	-1.424
1	0	mHinit	1	8	0.000042895110	0.141
1	0	mHinit	1	16	0.000040684580	53.506
1	0	mVinit	1	4	0.000043370050	-1.438
1	0	mVinit	1	8	0.000042917230	0.571
1	0	mVinit	1	16	0.000041229800	41.848
1	0	mRinit	1	4	0.000043316410	-1.373
1	0	mRinit	1	8	0.000042911980	0.347
1	0	mRinit	1	16	0.000040572010	56.703
1	0	mHBinit	1	4	0.000087311720	-7.379
1	0	mHBinit	1	8	0.000174159500	-18.674
1	0	mHBinit	1	16	0.000171892000	-6.367

continued on next page

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
1	0	mVBinit	1	4	0.000087404220	-7.127
1	0	mVBinit	1	8	0.000086371020	-4.588
1	0	mVBinit	1	16	0.000172297200	-9.117
1	0	mRepInit	1	4	0.000175085800	-10.687
1	0	mRepInit	1	8	0.000349450400	-26.094
1	0	mRepInit	1	16	0.000698077700	-52.363
1	0	mHGather	4	1	0.000043138870	19.761
1	0	mVGather	4	1	0.000042942860	29.373
1	0	mRGather	4	1	0.000042711780	35.090
1	0	mHBGather	4	1	0.000042996700	24.202
1	0	mVBGather	4	1	0.000043107570	22.805
1	0	mRepGather	4	1	0.000045558640	1.260
1	1	mH2H	4	4	0.000000638339	0.347
1	1	mH2V	4	4	0.000021943780	4.419
1	1	mH2R	4	4	0.000011504000	1.137
1	1	mH2HB	4	4	0.000022914760	1.482
1	1	mH2VB	4	4	0.000043876980	2.908
1	1	mH2Rep	4	4	0.000088239730	0.121
1	1	mV2H	4	4	0.000021912610	4.991
1	1	mV2V	4	4	0.000000674760	0.376
1	1	mV2R	4	4	0.000033023840	-3.584
1	1	mV2HB	4	4	0.000043927580	2.635
1	1	mV2VB	4	4	0.000066050110	-3.409
1	1	mV2Rep	4	4	0.000088415690	-0.212
1	1	mR2H	4	4	0.000011498570	1.092
1	1	mR2V	4	4	0.000033003740	1.788
1	1	mR2R	4	4	0.000000446131	0.452
1	1	mR2HB	4	4	0.000024627840	1.242
continued on next page						

continued from previous page

Previous Group	Next Group	Option	p	q	Slope	Intercept
1	1	mR2VB	4	4	0.000022827710	1.869
1	1	mR2Rep	4	4	0.000088156350	0.257
1	1	mHB2H	4	4	0.000000418070	0.382
1	1	mHB2V	4	4	0.000011429790	1.447
1	1	mHB2R	4	4	0.000000458562	0.433
1	1	mHB2HB	4	4	0.000000830287	0.537
1	1	mHB2VB	4	4	0.000022775390	1.882
1	1	mHB2Rep	4	4	0.000045877620	1.015
1	1	mVB2H	4	4	0.000011406840	1.151
1	1	mVB2V	4	4	0.000022352650	1.324
1	1	mVB2R	4	4	0.000000486192	0.394
1	1	mVB2HB	4	4	0.000022776490	1.845
1	1	mVB2VB	4	4	0.000000971110	0.538
1	1	mVB2Rep	4	4	0.000045603100	1.289
1	1	mRep2H	4	4	0.000000416960	0.377
1	1	mRep2V	4	4	0.000000459925	0.512
1	1	mRep2R	4	4	0.000000454072	0.418
1	1	mRep2HB	4	4	0.000000834763	0.576
1	1	mRep2VB	4	4	0.000000879853	0.665
1	1	mRep2Rep	4	4	0.000001644083	0.941
1	1	mNone2None	1	1	0.000000000000	0.154
1	1	mHinit	1	4	0.000032559870	-2.015
1	1	mVinit	1	4	0.000032504350	-0.838
1	1	mRinit	1	4	0.000032488940	-0.980
1	1	mHBinit	1	4	0.000065429050	-5.072
1	1	mVBinit	1	4	0.000065488090	-5.103
1	1	mRepInit	1	4	0.000131252300	-7.702
1	1	mHGather	4	1	0.000032768420	4.559

continued on next page

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
1	1	mVGather	4	1	0.000032876790	3.861
1	1	mRGather	4	1	0.000032811390	4.690
1	1	mHBGather	4	1	0.000032778250	4.495
1	1	mVBGather	4	1	0.000032817230	4.647
1	1	mRepGather	4	1	0.000000000000	0.152
1	2	mH2H	4	4	0.000054699120	3.536
1	2	mH2V	4	4	0.000055175740	74.926
1	2	mH2R	4	4	0.000054587440	8.503
1	2	mH2HB	4	4	0.000107983800	31.550
1	2	mH2VB	4	4	0.000114896200	17.766
1	2	mH2Rep	4	4	0.000230860000	12.938
1	2	mV2H	4	4	0.000054732570	78.940
1	2	mV2V	4	4	0.000054740360	3.844
1	2	mV2R	4	4	0.000057794270	2.121
1	2	mV2HB	4	4	0.000114652900	29.358
1	2	mV2VB	4	4	0.000115653900	1.921
1	2	mV2Rep	4	4	0.000230864100	13.169
1	2	mR2H	4	4	0.000054418820	10.460
1	2	mR2V	4	4	0.000055727180	4.351
1	2	mR2R	4	4	0.000054707770	4.638
1	2	mR2HB	4	4	0.000109014800	8.871
1	2	mR2VB	4	4	0.000108822300	8.525
1	2	mR2Rep	4	4	0.000231315200	-1.155
1	2	mHB2H	4	4	0.000054645790	4.419
1	2	mHB2V	4	4	0.000054397490	11.213
1	2	mHB2R	4	4	0.000054587730	5.185
1	2	mHB2HB	4	4	0.000109266500	4.677
1	2	mHB2VB	4	4	0.000108402400	20.104
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
1	2	mHB2Rep	4	4	0.000217860200	11.808
1	2	mVB2H	4	4	0.000054093630	16.288
1	2	mVB2V	4	4	0.000054686690	4.628
1	2	mVB2R	4	4	0.000054627250	5.177
1	2	mVB2HB	4	4	0.000108622100	14.182
1	2	mVB2VB	4	4	0.000109176300	5.942
1	2	mVB2Rep	4	4	0.000217752000	8.950
1	2	mRep2H	4	4	0.000054606960	4.786
1	2	mRep2V	4	4	0.000054823180	2.893
1	2	mRep2R	4	4	0.000054768300	3.337
1	2	mRep2HB	4	4	0.000109358200	3.039
1	2	mRep2VB	4	4	0.000109087900	6.453
1	2	mRep2Rep	4	4	0.000218383200	4.972
1	2	mNone2None	1	1	0.000062387290	-3.095
1	2	mHinit	1	4	0.000060489780	0.768
1	2	mVinit	1	4	0.000060939990	-5.352
1	2	mRinit	1	4	0.000061006410	-6.606
1	2	mHBinit	1	4	0.000122167900	-9.908
1	2	mVBinit	1	4	0.000122362400	-8.891
1	2	mRepInit	1	4	0.000246273600	-27.836
1	2	mHGather	4	1	0.000061562620	0.323
1	2	mVGather	4	1	0.000061798020	-1.642
1	2	mRGather	4	1	0.000061776610	-2.409
1	2	mHBGather	4	1	0.000061617200	-1.097
1	2	mVBGather	4	1	0.000061786120	-1.306
1	2	mRepGather	4	1	0.000062584880	-2.870
2	0	mH2H	4	4	0.000048282120	0.176
2	0	mH2H	4	8	0.000047847640	1.615
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
2	0	mH2H	4	16	0.000047601980	3.724
2	0	mH2V	4	4	0.000047553220	0.691
2	0	mH2V	4	8	0.000047570470	0.929
2	0	mH2V	4	16	0.000044804980	85.655
2	0	mH2R	4	4	0.000048045970	1.303
2	0	mH2R	4	8	0.000047721610	0.855
2	0	mH2R	4	16	0.000047642040	2.561
2	0	mH2HB	4	4	0.000096126670	0.830
2	0	mH2HB	4	8	0.000190696700	1.489
2	0	mH2HB	4	16	0.000190938400	-0.029
2	0	mH2VB	4	4	0.000094854170	2.210
2	0	mH2VB	4	8	0.000094959060	2.685
2	0	mH2VB	4	16	0.000188774500	36.693
2	0	mH2Rep	4	4	0.000189652700	2.028
2	0	mH2Rep	4	8	0.000379866700	1.534
2	0	mH2Rep	4	16	0.000757368300	33.470
2	0	mV2H	4	4	0.000047452540	1.514
2	0	mV2H	4	8	0.000047286210	8.727
2	0	mV2H	4	16	0.000045105520	72.873
2	0	mV2V	4	4	0.000048369320	1.226
2	0	mV2V	4	8	0.000047952810	1.688
2	0	mV2V	4	16	0.000047715190	4.384
2	0	mV2R	4	4	0.000048393970	0.927
2	0	mV2R	4	8	0.000048118790	1.501
2	0	mV2R	4	16	0.000047705520	4.919
2	0	mV2HB	4	4	0.000094873630	1.732
2	0	mV2HB	4	8	0.000189877800	3.098
2	0	mV2HB	4	16	0.000189352100	22.219
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
2	0	mV2VB	4	4	0.000096751370	0.964
2	0	mV2VB	4	8	0.000096179030	5.199
2	0	mV2VB	4	16	0.000191429900	2.818
2	0	mV2Rep	4	4	0.000189935300	0.108
2	0	mV2Rep	4	8	0.000380669300	-6.764
2	0	mV2Rep	4	16	0.000757245800	21.504
2	0	mR2H	4	4	0.000048044450	1.072
2	0	mR2H	4	8	0.000047681400	0.832
2	0	mR2H	4	16	0.000046508440	31.696
2	0	mR2V	4	4	0.000047839120	1.559
2	0	mR2V	4	8	0.000047432340	8.066
2	0	mR2V	4	16	0.000047498660	6.703
2	0	mR2R	4	4	0.000048282920	1.508
2	0	mR2R	4	8	0.000047928930	1.367
2	0	mR2R	4	16	0.000047509870	7.025
2	0	mR2HB	4	4	0.000096182530	0.775
2	0	mR2HB	4	8	0.000190795800	0.578
2	0	mR2HB	4	16	0.000190196500	6.522
2	0	mR2VB	4	4	0.000096118010	1.905
2	0	mR2VB	4	8	0.000095049270	2.969
2	0	mR2VB	4	16	0.000190223600	2.464
2	0	mR2Rep	4	4	0.000189843400	0.468
2	0	mR2Rep	4	8	0.000380091300	-0.365
2	0	mR2Rep	4	16	0.000757168500	33.895
2	0	mHB2H	4	4	0.000048253090	0.949
2	0	mHB2H	4	8	0.000047665170	6.009
2	0	mHB2H	4	16	0.000047618780	3.041
2	0	mHB2V	4	4	0.000048006060	3.984
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
2	0	mHB2V	4	8	0.000047661690	2.970
2	0	mHB2V	4	16	0.000047475940	6.582
2	0	mHB2R	4	4	0.000048293910	1.018
2	0	mHB2R	4	8	0.000047884900	1.847
2	0	mHB2R	4	16	0.000047544470	5.614
2	0	mHB2HB	4	4	0.000096562770	1.258
2	0	mHB2HB	4	8	0.000191691900	-0.226
2	0	mHB2HB	4	16	0.000190754700	4.504
2	0	mHB2VB	4	4	0.000095946130	6.812
2	0	mHB2VB	4	8	0.000095528710	0.943
2	0	mHB2VB	4	16	0.000190511400	2.957
2	0	mHB2Rep	4	4	0.000192302200	0.709
2	0	mHB2Rep	4	8	0.000382036200	-0.268
2	0	mHB2Rep	4	16	0.000758813000	32.440
2	0	mVB2H	4	4	0.000048045090	0.810
2	0	mVB2H	4	8	0.000047709120	0.404
2	0	mVB2H	4	16	0.000047475140	1.931
2	0	mVB2V	4	4	0.000048329150	1.606
2	0	mVB2V	4	8	0.000047885550	2.039
2	0	mVB2V	4	16	0.000047293700	14.437
2	0	mVB2R	4	4	0.000048289920	1.514
2	0	mVB2R	4	8	0.000047884840	2.152
2	0	mVB2R	4	16	0.000047411920	9.543
2	0	mVB2HB	4	4	0.000096118180	0.822
2	0	mVB2HB	4	8	0.000190796400	1.369
2	0	mVB2HB	4	16	0.000190018700	3.219
2	0	mVB2VB	4	4	0.000096650090	1.403
2	0	mVB2VB	4	8	0.000095869410	1.215
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
2	0	mVB2VB	4	16	0.000191050100	0.912
2	0	mVB2Rep	4	4	0.000192317100	1.357
2	0	mVB2Rep	4	8	0.000381693800	0.655
2	0	mVB2Rep	4	16	0.000760284400	2.521
2	0	mRep2H	4	4	0.000048262580	1.087
2	0	mRep2H	4	8	0.000047869540	1.038
2	0	mRep2H	4	16	0.000047531930	5.862
2	0	mRep2V	4	4	0.000048333660	1.683
2	0	mRep2V	4	8	0.000047886250	2.413
2	0	mRep2V	4	16	0.000047510130	9.203
2	0	mRep2R	4	4	0.000048328240	0.848
2	0	mRep2R	4	8	0.000047852770	2.069
2	0	mRep2R	4	16	0.000047172490	15.629
2	0	mRep2HB	4	4	0.000096577390	0.509
2	0	mRep2HB	4	8	0.000191697400	-0.696
2	0	mRep2HB	4	16	0.000190748900	5.568
2	0	mRep2VB	4	4	0.000096640610	0.877
2	0	mRep2VB	4	8	0.000095853130	1.433
2	0	mRep2VB	4	16	0.000191002800	0.576
2	0	mRep2Rep	4	4	0.000193149200	1.255
2	0	mRep2Rep	4	8	0.000383459200	0.300
2	0	mRep2Rep	4	16	0.000763990000	-0.735
2	0	mNone2None	1	1	0.000048079720	1.408
2	0	mHinit	1	4	0.000045743720	-1.072
2	0	mHinit	1	8	0.000045411300	0.338
2	0	mHinit	1	16	0.000042753010	64.174
2	0	mVinit	1	4	0.000045847060	-1.194
2	0	mVinit	1	8	0.000045430770	0.770
continued on next page						

continued from previous page

Previous Group	Next Group	Option	p	q	Slope	Intercept
2	0	mVinit	1	16	0.000042904890	61.730
2	0	mRinit	1	4	0.000045796410	-1.126
2	0	mRinit	1	8	0.000045426430	0.567
2	0	mRinit	1	16	0.000042744150	64.726
2	0	mHBinit	1	4	0.000092288000	-7.201
2	0	mHBinit	1	8	0.000184097900	-18.302
2	0	mHBinit	1	16	0.000181650400	-3.747
2	0	mVBinit	1	4	0.000092382530	-6.991
2	0	mVBinit	1	8	0.000091275170	-3.595
2	0	mVBinit	1	16	0.000182147000	-7.492
2	0	mRepInit	1	4	0.000185097500	-11.140
2	0	mRepInit	1	8	0.000369684000	-28.139
2	0	mRepInit	1	16	0.000738876200	-63.238
2	0	mHGather	4	1	0.000045065420	37.813
2	0	mVGather	4	1	0.000044763280	50.960
2	0	mRGather	4	1	0.000045085990	39.498
2	0	mHBGather	4	1	0.000045569310	22.376
2	0	mVBGather	4	1	0.000045110630	38.778
2	0	mRepGather	4	1	0.000048074680	1.456
2	1	mH2H	4	4	0.000047711590	1.125
2	1	mH2V	4	4	0.000047168690	0.766
2	1	mH2R	4	4	0.000047517610	1.178
2	1	mH2HB	4	4	0.000094952150	3.929
2	1	mH2VB	4	4	0.000094091660	2.154
2	1	mH2Rep	4	4	0.000188178200	2.078
2	1	mV2H	4	4	0.000047123270	0.707
2	1	mV2V	4	4	0.000047783110	0.777
2	1	mV2R	4	4	0.000047677030	1.299

continued on next page

[illegible]

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
2	1	mHinit	1	4	0.000045559180	-1.776
2	1	mVinit	1	4	0.000045641950	-1.982
2	1	mRinit	1	4	0.000045601380	-1.915
2	1	mHBinit	1	4	0.000091847350	-7.391
2	1	mVBinit	1	4	0.000091926090	-7.468
2	1	mRepInit	1	4	0.000184223500	-11.261
2	1	mHGather	4	1	0.000045532050	2.537
2	1	mVGather	4	1	0.000045542690	4.055
2	1	mRGather	4	1	0.000045514960	3.823
2	1	mHBGather	4	1	0.000045501550	3.401
2	1	mVBGather	4	1	0.000045508980	3.926
2	1	mRepGather	4	1	0.000046943840	0.896
2	2	mH2H	4	4	0.000000654697	0.256
2	2	mH2V	4	4	0.000034465630	8.733
2	2	mH2R	4	4	0.000023440700	1.800
2	2	mH2HB	4	4	0.000046769320	2.162
2	2	mH2VB	4	4	0.000068885710	5.464
2	2	mH2Rep	4	4	0.000138148700	3.298
2	2	mV2H	4	4	0.000034638130	2.866
2	2	mV2V	4	4	0.000000691661	0.280
2	2	mV2R	4	4	0.000034641610	1.201
2	2	mV2HB	4	4	0.000068879150	6.161
2	2	mV2VB	4	4	0.000069363890	0.692
2	2	mV2Rep	4	4	0.000138350900	1.211
2	2	mR2H	4	4	0.000023465630	1.048
2	2	mR2V	4	4	0.000034757800	1.189
2	2	mR2R	4	4	0.000000442742	0.524
2	2	mR2HB	4	4	0.000048436640	2.195
continued on next page						

continued from previous page

Previous Group	Next Group	Option	p	q	Slope	Intercept
2	2	mR2VB	4	4	0.000046718220	1.880
2	2	mR2Rep	4	4	0.000138095500	1.965
2	2	mHB2H	4	4	0.000000418000	0.375
2	2	mHB2V	4	4	0.000023377510	1.557
2	2	mHB2R	4	4	0.000000446459	0.469
2	2	mHB2HB	4	4	0.000000830174	0.541
2	2	mHB2VB	4	4	0.000046491950	5.690
2	2	mHB2Rep	4	4	0.000093517710	2.243
2	2	mVB2H	4	4	0.000023284700	2.324
2	2	mVB2V	4	4	0.000023535820	0.841
2	2	mVB2R	4	4	0.000000482825	0.491
2	2	mVB2HB	4	4	0.000046463580	5.924
2	2	mVB2VB	4	4	0.000000960509	0.768
2	2	mVB2Rep	4	4	0.000093198270	3.353
2	2	mRep2H	4	4	0.000000417287	0.360
2	2	mRep2V	4	4	0.000000460274	0.495
2	2	mRep2R	4	4	0.000000437938	0.507
2	2	mRep2HB	4	4	0.000000832511	0.602
2	2	mRep2VB	4	4	0.000000855554	0.768
2	2	mRep2Rep	4	4	0.000001643416	0.978
2	2	mNone2None	1	1	0.000000000000	0.112
2	2	mHinit	1	4	0.000033890990	5.982
2	2	mVinit	1	4	0.000034215780	-0.626
2	2	mRinit	1	4	0.000034182640	-0.595
2	2	mHBinit	1	4	0.000068904050	-5.273
2	2	mVBinit	1	4	0.000068964340	-5.146
2	2	mRepInit	1	4	0.000138306300	-9.142
2	2	mHGather	4	1	0.000034210860	10.092

continued on next page

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
2	2	mVGather	4	1	0.000034671830	0.952
2	2	mRGather	4	1	0.000034631640	0.839
2	2	mHBGather	4	1	0.000034603100	0.602
2	2	mVBGather	4	1	0.000034635750	0.793
2	2	mRepGather	4	1	0.0000000000013	0.119
0	0	vH2H	4	4	0.0000000000000	0.614
0	0	vH2H	4	8	0.000013379390	0.535
0	0	vH2H	4	16	0.000009662760	0.619
0	0	vH2H	8	4	0.000008847739	0.525
0	0	vH2H	8	8	0.000004996275	0.516
0	0	vH2H	8	16	0.000008208262	0.628
0	0	vH2H	16	4	0.000003331101	0.609
0	0	vH2H	16	8	0.000006187754	0.616
0	0	vH2H	16	16	0.0000000000000	0.587
0	0	vH2HB	4	4	0.000039453250	0.394
0	0	vH2HB	4	8	0.000612007500	-3.724
0	0	vH2HB	4	16	0.000089058390	0.627
0	0	vH2HB	8	4	0.000025755680	0.481
0	0	vH2HB	8	8	0.000050135580	0.633
0	0	vH2HB	8	16	0.000001004319	1.921
0	0	vH2HB	16	4	0.000017302860	0.598
0	0	vH2HB	16	8	0.000023681890	0.651
0	0	vH2HB	16	16	0.000024534870	0.707
0	0	vH2Rep	4	4	0.000097978620	0.524
0	0	vH2Rep	4	8	0.000158997300	0.720
0	0	vH2Rep	4	16	0.000242416600	0.884
0	0	vH2Rep	8	4	0.000060736230	0.468
0	0	vH2Rep	8	8	0.000108091100	0.721
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	0	vH2Rep	8	16	0.000159976900	1.127
0	0	vH2Rep	16	4	0.000053342880	0.594
0	0	vH2Rep	16	8	0.000061730340	0.883
0	0	vH2Rep	16	16	0.000107727300	1.107
0	0	vHB2H	4	4	0.000001026044	0.564
0	0	vHB2H	4	8	0.000012321810	0.530
0	0	vHB2H	4	16	0.000010071220	0.619
0	0	vHB2H	8	4	0.000011694350	0.508
0	0	vHB2H	8	8	0.000000199008	0.580
0	0	vHB2H	8	16	0.000005556176	0.632
0	0	vHB2H	16	4	0.000014755100	0.488
0	0	vHB2H	16	8	0.000009940443	0.556
0	0	vHB2H	16	16	0.000000000000	0.597
0	0	vHB2HB	4	4	0.000000000000	0.568
0	0	vHB2HB	4	8	0.000049135200	0.523
0	0	vHB2HB	4	16	0.000051495940	0.579
0	0	vHB2HB	8	4	0.000027418100	0.506
0	0	vHB2HB	8	8	0.000000642889	0.625
0	0	vHB2HB	8	16	0.000029871040	0.685
0	0	vHB2HB	16	4	0.000024742480	0.521
0	0	vHB2HB	16	8	0.000024554560	0.654
0	0	vHB2HB	16	16	0.000001357648	0.581
0	0	vHB2Rep	4	4	0.000052104650	0.523
0	0	vHB2Rep	4	8	0.000131464200	0.651
0	0	vHB2Rep	4	16	0.000229002400	0.726
0	0	vHB2Rep	8	4	0.000026779690	0.553
0	0	vHB2Rep	8	8	0.000050251640	0.677
0	0	vHB2Rep	8	16	0.000124454900	0.831
continued on next page						

continued from previous page

Previous Group	Next Group	Option	p	q	Slope	Intercept
0	0	vHB2Rep	16	4	0.000041711360	0.490
0	0	vHB2Rep	16	8	0.000043661990	0.551
0	0	vHB2Rep	16	16	0.000099489160	0.785
0	0	vRep2H	4	4	0.000005177064	0.525
0	0	vRep2H	4	8	0.000005187200	0.569
0	0	vRep2H	4	16	0.000009335393	0.582
0	0	vRep2H	8	4	0.000001182043	0.580
0	0	vRep2H	8	8	0.000000050751	0.601
0	0	vRep2H	8	16	0.000000795268	0.654
0	0	vRep2H	16	4	0.000004630355	0.525
0	0	vRep2H	16	8	0.000000130441	0.624
0	0	vRep2H	16	16	0.000000000000	0.603
0	0	vRep2HB	4	4	0.000000000000	0.593
0	0	vRep2HB	4	8	0.000026318270	0.565
0	0	vRep2HB	4	16	0.000039549430	0.569
0	0	vRep2HB	8	4	0.000002244919	0.557
0	0	vRep2HB	8	8	0.000000532519	0.580
0	0	vRep2HB	8	16	0.000019656370	0.606
0	0	vRep2HB	16	4	0.000001478599	0.573
0	0	vRep2HB	16	8	0.000000000000	0.596
0	0	vRep2HB	16	16	0.000000369512	0.571
0	0	vRep2Rep	4	4	0.000000420078	0.556
0	0	vRep2Rep	4	8	0.000047580210	0.564
0	0	vRep2Rep	4	16	0.000129832400	0.738
0	0	vRep2Rep	8	4	0.000001003652	0.582
0	0	vRep2Rep	8	8	0.000002346243	0.547
0	0	vRep2Rep	8	16	0.000046860990	0.781
0	0	vRep2Rep	16	4	0.000000000000	0.588

continued on next page

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	0	vRep2Rep	16	8	0.000003118293	0.549
0	0	vRep2Rep	16	16	0.000002167389	0.579
0	0	vNone2None	1	1	0.000000101632	0.555
0	0	vHinit	1	4	0.000035486650	0.377
0	0	vHinit	1	8	0.000037230410	0.460
0	0	vHinit	1	16	0.000038708260	0.575
0	0	vHBinit	1	4	0.000066441470	0.380
0	0	vHBinit	1	8	0.000153782300	0.455
0	0	vHBinit	1	16	0.000167495600	0.512
0	0	vRepInit	1	4	0.000128213000	0.432
0	0	vRepInit	1	8	0.000303054700	0.474
0	0	vRepInit	1	16	0.000634601600	0.696
0	0	vHGather	4	1	0.000026652700	0.458
0	0	vHGather	8	1	0.000027617430	0.444
0	0	vHGather	16	1	0.000032344380	0.494
0	0	vHBGather	4	1	0.000025477450	0.466
0	0	vHBGather	8	1	0.000027549290	0.445
0	0	vHBGather	16	1	0.000031198850	0.504
0	0	vRepGather	4	1	0.000000000000	0.589
0	0	vRepGather	8	1	0.000002032005	0.567
0	0	vRepGather	16	1	0.000000000000	0.580
0	1	vH2H	4	4	0.000016691800	1.397
0	1	vH2H	8	4	0.000019663110	0.824
0	1	vH2H	16	4	0.000008346291	1.481
0	1	vH2HB	4	4	0.000101876400	0.433
0	1	vH2HB	8	4	0.000032346040	1.400
0	1	vH2HB	16	4	0.000016311720	1.512
0	1	vH2Rep	4	4	0.001315989000	-7.071
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	1	vH2Rep	8	4	0.000840576500	-2.822
0	1	vH2Rep	16	4	0.001999738000	-11.404
0	1	vHB2H	4	4	0.000017272960	1.394
0	1	vHB2H	8	4	0.000017317490	1.391
0	1	vHB2H	16	4	0.000016804210	1.397
0	1	vHB2HB	4	4	0.000054692820	0.969
0	1	vHB2HB	8	4	0.000050422000	1.001
0	1	vHB2HB	16	4	0.000090715690	0.489
0	1	vHB2Rep	4	4	0.000069484380	1.201
0	1	vHB2Rep	8	4	0.001860519000	-10.261
0	1	vHB2Rep	16	4	0.004251824000	-22.420
0	1	vRep2H	4	4	0.000016962000	1.395
0	1	vRep2H	8	4	0.000016906070	1.398
0	1	vRep2H	16	4	0.000016635700	1.398
0	1	vRep2HB	4	4	0.000057511490	0.949
0	1	vRep2HB	8	4	0.000054244710	0.979
0	1	vRep2HB	16	4	0.000054511160	0.982
0	1	vRep2Rep	4	4	0.000068288090	1.188
0	1	vRep2Rep	8	4	0.000087569270	0.904
0	1	vRep2Rep	16	4	0.000051978570	1.631
0	1	vNone2None	1	1	0.000076738620	1.040
0	1	vHinit	1	4	0.000016968850	1.402
0	1	vHBinit	1	4	0.000049035920	1.162
0	1	vRepInit	1	4	0.000169713100	0.693
0	1	vHGather	4	1	0.000035150260	1.515
0	1	vHGather	8	1	0.000028643340	1.492
0	1	vHGather	16	1	0.000027594510	1.687
0	1	vHBGather	4	1	0.000116428500	0.274
continued on next page						

[illegible]

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
0	2	vRep2HB	16	4	0.000138040200	0.369
0	2	vRep2Rep	4	4	0.000435146500	-0.603
0	2	vRep2Rep	8	4	0.000428623500	-0.631
0	2	vRep2Rep	16	4	0.000814183200	-3.423
0	2	vNone2None	1	1	0.000049726710	0.552
0	2	vHinit	1	4	0.000064176200	0.438
0	2	vHBinit	1	4	0.000116978400	0.472
0	2	vRepInit	1	4	0.000262806800	0.337
0	2	vHGather	4	1	0.000059343720	0.486
0	2	vHGather	8	1	0.000035294300	0.856
0	2	vHGather	16	1	0.004148915000	-31.867
0	2	vHBGather	4	1	0.000061209370	0.448
0	2	vHBGather	8	1	0.000045559290	0.708
0	2	vHBGather	16	1	0.004017638000	-31.692
0	2	vRepGather	4	1	0.000054235050	0.506
0	2	vRepGather	8	1	0.000052834740	0.523
0	2	vRepGather	16	1	0.000049287230	0.545
1	0	vH2H	4	4	0.000019153990	0.627
1	0	vH2H	4	8	0.000016034640	0.626
1	0	vH2H	4	16	0.000012226500	0.693
1	0	vH2HB	4	4	0.000039801500	0.517
1	0	vH2HB	4	8	0.000063748130	0.629
1	0	vH2HB	4	16	0.000053785730	0.726
1	0	vH2Rep	4	4	0.000062662650	0.501
1	0	vH2Rep	4	8	0.000117387500	0.597
1	0	vH2Rep	4	16	0.000205177900	0.680
1	0	vHB2H	4	4	0.000024444720	0.529
1	0	vHB2H	4	8	0.000020886170	0.598
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
1	0	vHB2H	4	16	0.000011267690	0.745
1	0	vHB2HB	4	4	0.000025870540	0.655
1	0	vHB2HB	4	8	0.000049534030	0.710
1	0	vHB2HB	4	16	0.000054490750	0.701
1	0	vHB2Rep	4	4	0.000049338130	0.652
1	0	vHB2Rep	4	8	0.000093621920	0.670
1	0	vHB2Rep	4	16	0.000175372800	0.817
1	0	vRep2H	4	4	0.000024461560	0.537
1	0	vRep2H	4	8	0.000016976570	0.621
1	0	vRep2H	4	16	0.000015152310	0.675
1	0	vRep2HB	4	4	0.000026391390	0.664
1	0	vRep2HB	4	8	0.000052256110	0.710
1	0	vRep2HB	4	16	0.000051553610	0.781
1	0	vRep2Rep	4	4	0.000048659500	0.708
1	0	vRep2Rep	4	8	0.000089656050	0.820
1	0	vRep2Rep	4	16	0.000171894100	0.933
1	0	vNone2None	1	1	0.000044467580	0.454
1	0	vHinit	1	4	0.000053301210	0.448
1	0	vHinit	1	8	0.000048998620	0.515
1	0	vHinit	1	16	0.000043678610	0.689
1	0	vHBinit	1	4	0.000088706000	0.551
1	0	vHBinit	1	8	0.000177047200	0.617
1	0	vHBinit	1	16	0.000182980800	0.652
1	0	vRepInit	1	4	0.000174041700	0.592
1	0	vRepInit	1	8	0.000343376900	0.701
1	0	vRepInit	1	16	0.000678844900	0.886
1	0	vHGather	4	1	0.000049063420	0.404
1	0	vHBGather	4	1	0.000048561240	0.410
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
1	0	vRepGather	4	1	0.000044878340	0.460
1	1	vH2H	4	4	0.000000490083	0.157
1	1	vH2HB	4	4	0.000036945480	0.308
1	1	vH2Rep	4	4	0.000095908110	0.457
1	1	vHB2H	4	4	0.000000342855	0.154
1	1	vHB2HB	4	4	0.000000783800	0.154
1	1	vHB2Rep	4	4	0.000051136030	0.451
1	1	vRep2H	4	4	0.000000357434	0.154
1	1	vRep2HB	4	4	0.000000658889	0.154
1	1	vRep2Rep	4	4	0.000001828853	0.152
1	1	vNone2None	1	1	0.000000000000	0.152
1	1	vHinit	1	4	0.000039180470	0.196
1	1	vHBinit	1	4	0.000066589370	0.266
1	1	vRepInit	1	4	0.000129080400	0.295
1	1	vHGather	4	1	0.000040171000	0.211
1	1	vHBGather	4	1	0.000040050640	0.209
1	1	vRepGather	4	1	0.000000000000	0.153
1	2	vH2H	4	4	0.000058942130	0.301
1	2	vH2HB	4	4	0.007710499000	-47.315
1	2	vH2Rep	4	4	0.027397810000	-169.994
1	2	vHB2H	4	4	0.000062414530	0.256
1	2	vHB2HB	4	4	0.000143100900	0.115
1	2	vHB2Rep	4	4	0.009079591000	-19.732
1	2	vRep2H	4	4	0.000060375760	0.270
1	2	vRep2HB	4	4	0.000139474900	0.152
1	2	vRep2Rep	4	4	0.000358126100	-0.401
1	2	vNone2None	1	1	0.000059571250	0.274
1	2	vHinit	1	4	0.000056600260	0.301
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
1	2	vHBinit	1	4	0.000120186200	0.240
1	2	vRepInit	1	4	0.000297908800	-0.144
1	2	vHGather	4	1	0.000054410170	0.363
1	2	vHBGather	4	1	0.000061325810	0.296
1	2	vRepGather	4	1	0.000059634990	0.282
2	0	vH2H	4	4	0.000058491640	0.525
2	0	vH2H	4	8	0.000053349030	0.686
2	0	vH2H	4	16	0.000082930090	0.805
2	0	vH2HB	4	4	0.000147660900	0.375
2	0	vH2HB	4	8	0.000621698200	-1.707
2	0	vH2HB	4	16	0.000250744500	0.955
2	0	vH2Rep	4	4	0.000254416500	0.515
2	0	vH2Rep	4	8	0.000448592400	1.143
2	0	vH2Rep	4	16	0.000798256400	2.292
2	0	vHB2H	4	4	0.000056458500	0.505
2	0	vHB2H	4	8	0.000045611310	0.682
2	0	vHB2H	4	16	0.000053561080	0.903
2	0	vHB2HB	4	4	0.000100517500	0.567
2	0	vHB2HB	4	8	0.000218855800	0.636
2	0	vHB2HB	4	16	0.000248405500	0.544
2	0	vHB2Rep	4	4	0.000226043500	0.524
2	0	vHB2Rep	4	8	0.000446784100	0.645
2	0	vHB2Rep	4	16	0.000748331500	1.975
2	0	vRep2H	4	4	0.000055589290	0.516
2	0	vRep2H	4	8	0.000045851610	0.651
2	0	vRep2H	4	16	0.000049996180	0.955
2	0	vRep2HB	4	4	0.000106432000	0.528
2	0	vRep2HB	4	8	0.000211187200	0.692
continued on next page						

continued from previous page						
Previous Group	Next Group	Option	p	q	Slope	Intercept
2	0	vRep2HB	4	16	0.000246749600	0.560
2	0	vRep2Rep	4	4	0.000231704600	0.438
2	0	vRep2Rep	4	8	0.000422844700	0.853
2	0	vRep2Rep	4	16	0.000836590800	2.137
2	0	vNone2None	1	1	0.000051460210	0.470
2	0	vHinit	1	4	0.000055868900	0.533
2	0	vHinit	1	8	0.000049465600	0.661
2	0	vHinit	1	16	0.000049940630	0.950
2	0	vHBinit	1	4	0.000109929700	0.529
2	0	vHBinit	1	8	0.000184913800	0.811
2	0	vHBinit	1	16	0.000196899300	0.764
2	0	vRepInit	1	4	0.000183567400	0.684
2	0	vRepInit	1	8	0.000395986000	0.854
2	0	vRepInit	1	16	0.000703166100	1.978
2	0	vHGather	4	1	0.000062876660	0.413
2	0	vHBGather	4	1	0.000060834120	0.429
2	0	vRepGather	4	1	0.000050690300	0.469
2	1	vH2H	4	4	0.000066387200	0.300
2	1	vH2HB	4	4	0.000134617500	0.398
2	1	vH2Rep	4	4	0.000284109300	0.238
2	1	vHB2H	4	4	0.000068831430	0.282
2	1	vHB2HB	4	4	0.000119962400	0.371
2	1	vHB2Rep	4	4	0.000260545700	0.358
2	1	vRep2H	4	4	0.000067277080	0.274
2	1	vRep2HB	4	4	0.000112731100	0.448
2	1	vRep2Rep	4	4	0.000223975300	0.436
2	1	vNone2None	1	1	0.000048488060	0.342
2	1	vHinit	1	4	0.000054011050	0.376
continued on next page						

<i>continued from previous page</i>						
Previous Group	Next Group	Option	p	q	Slope	Intercept
2	1	vHbInit	1	4	0.000094389640	0.447
2	1	vRepInit	1	4	0.000179389600	0.519
2	1	vHGather	4	1	0.000057493380	0.322
2	1	vHBGather	4	1	0.000055911380	0.339
2	1	vRepGather	4	1	0.000047215750	0.360
2	2	vH2H	4	4	0.000000575579	0.116
2	2	vH2HB	4	4	0.000053989570	0.096
2	2	vH2Rep	4	4	0.000171802400	0.065
2	2	vHB2H	4	4	0.000000311654	0.117
2	2	vHB2HB	4	4	0.000000725448	0.118
2	2	vHB2Rep	4	4	0.000103596700	0.096
2	2	vRep2H	4	4	0.000000309876	0.116
2	2	vRep2HB	4	4	0.000000646702	0.117
2	2	vRep2Rep	4	4	0.000001803745	0.115
2	2	vNone2None	1	1	0.000000000000	0.115
2	2	vHinit	1	4	0.000040714970	0.087
2	2	vHbInit	1	4	0.000076334890	0.116
2	2	vRepInit	1	4	0.000164834800	0.044
2	2	vHGather	4	1	0.000042013360	0.119
2	2	vHBGather	4	1	0.000040002180	0.133
2	2	vRepGather	4	1	0.000000000000	0.115

REFERENCES

1. Dongarra, J. and A. Lastovetsky, “*An Overview of Heterogeneous High Performance and Grid Computing*”, *Engineering the Grid*, ed. A. Hoisie, L. Yang, H. Zima, Nova Science publishers, 2004.
2. Wang, L., H. J. Siegel, V. R. Roychowdhury and A. A. Maciejewski, “*Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach*”, *J. Parallel Distrib. Comput.*, Vol. 47, No. 1, pp. 8–22, 1997.
3. Plata, O., T. F. Pena, F. F. Rivera and E. L. Zapata, “*An efficient processor allocation for nested parallel loops on distributed memory hypercubes*”, *Parallel Processing Letters*, Vol. 3, No. 2, pp. 179–187, 1993.
4. Hamidzadeh, B., D. J. Lilja and Y. Atif, “*Dynamic scheduling techniques for heterogeneous computing systems*”, *Concurrency: Practice and Experience*, Vol. 7, No. 7, pp. 633–652, 1995.
5. Braun, T. D., H. J. Siegel and N. B. et al., “*A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems*”, *Journal of Parallel and Distributed Computing*, Vol. 61, No. 6, pp. 810–837, 2001.
6. Ali, S., T. Braun, H. Siegel and A. Maciejewski, *Heterogeneous Computing*, Encyclopedia of Distributed Computing, J. Urbana and P. Dasgupta, eds., Kluwer Academic Publishers, Norwell, MA, to appear 2002.
7. Yang, C.-T. and S.-C. Chang, “*A Parallel Loop Self-Scheduling on Extremely Heterogeneous PC Clusters.*”, *J. Inf. Sci. Eng.*, Vol. 20, No. 2, pp. 263–273, 2004.
8. Lilja, D. J., “*Experiments with a Task Partitioning Model for Heterogeneous Computing*”, University of Minnesota AHPCRC Preprint no. 92-142, Minneapolis, MN, De-

cember 1992.

9. Liu, P., T.-H. Sheng and C.-H. Yang, “*Heuristic Search of Optimal Reduction Schedule in Heterogeneous Cluster Environments*”, National Computer Symposium, 1999.
10. Kim, J.-K., S. Shiple, H. J. Siegel, A. A. Maciejewski, T. D. Braun, M. Schneider, S. Tideman, R. Chitta, R. B. Dilmaghani, R. Joshi, A. Kaul, A. Sharma, S. Sripada, P. Vangari and S. S. Yellampalli, “*Dynamic Mapping in a Heterogeneous Environment with Tasks Having Priorities and Multiple Deadlines*”, *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, p. 98.1, IEEE Computer Society, Washington, DC, USA, 2003.
11. Wu, M.-Y. and W. Shu, “*A High-Performance Mapping Algorithm for Heterogeneous Computing Systems*”, *IPDPS '01: Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, p. 10074.1, IEEE Computer Society, Washington, DC, USA, 2001.
12. Kwok, Y.-K., A. A. Maciejewski, H. J. Siegel, A. Ghafoor and I. Ahmad, “*Evaluation of a Semi-Static Approach to Mapping Dynamic Iterative Tasks onto Heterogeneous Computing Systems*”, *ISPAN '99: Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '99)*, p. 204, IEEE Computer Society, Washington, DC, USA, 1999.
13. Mazzariol, M., *Computer-aided parallelization of applications*, Ph.D. thesis, Lausanne, 2001.
14. Vikram, K., K. Avijit and S. K. Aggarwal, “*OlyMPIx - A Program Parallelization Tool using MPI on Computational Grids.*”, *Parallel and Distributed Computing and Networks*, pp. 307–312, 2005.
15. Rugina, R. and M. C. Rinard, “*Automatic Parallelization of Divide and Conquer Algorithms*”, *Principles Practice of Parallel Programming*, pp. 72–83, 1999.

16. Kung, H. T. and J. Subhlok, “A new approach for automatic parallelization of blocked linear Algebra computations”, *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pp. 122–129, ACM Press, New York, NY, USA, 1991.
17. Ozcan, E. and E. Onbasioglu, “Genetic Algorithms for Parallel Code Optimization”, *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pp. 1375–1381, IEEE Press, Portland, Oregon, 20-23 June 2004.
18. Ozcan, E. and E. Onbasioglu, “Memetic Algorithms for Parallel Code Optimization”, *International Journal of Parallel Programming*, 2006.