Spring 5-31-1995

Investigation of hybrid message-passing and shared-memory architectures for parallel computer : a case study : turbonet

Xi Li

New Jersey Institute of Technology

ABSTRACT

INVESTIGATION OF HYBRID MESSAGE-PASSING AND SHARED-MEMORY ARCHITECTURES FOR PARALLEL COMPUTERS. A CASE STUDY: TURBONET

by
Xi Li

Several DSP (Digital Signal Processing) algorithms are developed for the NJIT TurboNet parallel computer. In contrast to other parallel computers that implement exclusively in hardware either the message-passing or the shared-memory communication paradigm, or employ distributed shared-memory architectures characterized by inefficient implementation of the shared-memory paradigm, the hybrid architecture of TurboNet supports direct, efficient implementation of both paradigms. Three versions of each algorithm are developed, if possible, corresponding to message-passing, shared-memory, and hybrid communications, respectively. Theoretical and experimental comparisons of algorithms are employed in the analysis of performance. The results prove that the hybrid versions generally achieve better performance than the other two versions. The main conclusion of this research is that small-scale and medium-scale parallel computers should implement directly in hardware both communication paradigms, for high performance, robustness in relation to the application space, and ease of algorithm development. To facilitate theoretical comparisons, a methodology is developed for highly accurate prediction of algorithm performance. The success of this methodology proves that such prediction is possible for complex parallel computers, such as TurboNet, if enough information is provided by the data dependence graphs.

INVESTIGATION OF HYBRID MESSAGE-PASSING AND SHARED-MEMORY ARCHITECTURES FOR PARALLEL COMPUTERS. A CASE STUDY: TURBONET

by
Xi Li

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Department of Electrical and Computer Engineering

May 1995

APPROVAL PAGE

INVESTIGATION OF HYBRID MESSAGE-PASSING AND SHARED-MEMORY ARCHITECTURES FOR PARALLEL COMPUTERS. A CASE STUDY: TURBONET

Xi Li

Dr. Sotirios G. Ziavras, Thesis Advisor	Date
Assistant Professor of Electrical and Computer Engineering, NJIT	
Dr. Constantine N. Manikopoulos, Thesis Advisor	Date
Associate Professor of Electrical and Computer Engineering, NJIT	
Dr. Edwin Hou, Committee Member	Date
Assistant Professor of Computer and Information Science, NJIT	
Dr. Michael Palis, Committee Member	Date
Associate Professor of Electrical and Computer Engineering, NJIT	
Dr. Huifang Sun, Committee Member	Date
Member of Technical Staff, David Sarnoff Research Laboratories	
Dr. Mengchu Zhou, Committee Member	Date
Assistant Professor of Electrical and Computer Engineering, NJIT	

BIOGRAPHICAL SKETCH

Author: Xi Li

Degree: Doctor of Philosophy in Electrical Engineering

Date: May 1995

Undergraduate and Graduate Education:

- Doctor of Philosophy in Electrical Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1995

- Master of Science in Computer Engineering,
  Florida Institute of Technology, Melbourne, FL, 1989

- Bachelor of Science in Electrical Engineering,
  South-China Institute of Technology, Guangzhou, P.R.China, 1983

Major: Electrical Engineering

Presentation and Publications:

1. X. Li, S. G. Ziavras and C. N. Manikopoulos, "Parallel DSP Algorithms on TurboNet: An Experimental System with Hybrid Message-Passing and Shared-Memory Architecture," *Concurrency: Practice and Experience*, accepted for publication, 1995.

2. X. Li, S. G. Ziavras and C. N. Manikopoulos, "Parallel Generation of Adaptive Multiresolution Structures for Image Processing," *Concurrency: Practice and Experience*, submitted for publication.

3. R. Hross, S. G. Ziavras, C. N. Manikopoulos, N. J. Lad and X. Li, "A Defect Identification Algorithm for Sequential and Parallel Computers," *IEEE International Symposium on Industrial Electronics*, Athens, Greece, July 10-14, 1995, to appear.

This thesis is dedicated to
my parents, my sister and Catherine Cai

ACKNOWLEDGMENT

The author would like to express his sincere gratitude to his advisors, Professors Ziavras and Manikopoulos, for their guidance, insight and support throughout the process of producing this thesis.

My thanks are due to the other members of my thesis committee, Drs. E. Hou, M. Palis, H. Sun, and M. Zhou.

I am thankful to Catherine Cai for her support and encouragement.

The work presented in this research was supported in part by the National Science Foundation under Grants CDA-9121475 and DMI-9500260.

TABLE OF CONTENTS

Chapter **Page**

1 INTRODUCTION 1

1.1 Parallel Processing Systems 1

1.1.1 Shared-Memory Multiprocessors 2

1.1.2 Message-Passing Multicomputers 3

1.2 Message-Passing Versus Shared-Memory 5

1.3 Motivation, Objectives, and Contributions 8

1.4 Outline 9

2 TURBONET: A MESSAGE-PASSING AND SHARED-MEMORY HYBRID ARCHITECTURE 10

2.1 The Texas Instruments TMS320C40 Processor 10

2.2 Hydra Boards 11

2.3 The Host System 14

2.4 The Hybrid Architecture of the TurboNet System 14

3 MULTIDIMENSIONAL CONVOLUTION 18

3.1 One-Dimensional Convolution Algorithms 18

3.2 Two-Dimensional Convolution Algorithm 20

3.3 Theoretical Analysis 27

3.3.1 1-D Convolution 27

3.3.2 2-D Convolution 30

TABLE OF CONTENTS
(Continued)

Chapter **Page**

3.4 Performance Results on TurboNet 39

3.4.1 1-D Convolution 39

3.4.2 2-D Convolution 43

4 FAST FOURIER TRANSFORM 50

4.1 One-Dimensional FFT 50

4.2 Two-Dimensional FFT 54

4.3 Theoretical Analysis 57

4.3.1 1-D FFT 57

4.3.2 2-D FFT 62

4.4 Performance Results on TurboNet 65

4.4.1 1-D FFT 65

4.4.2 2-D FFT 70

5 MATRIX MULTIPLICATION 77

5.1 Matrix Multiplication Algorithm 77

5.2 Theoretical Analysis 84

5.3 Performance Results on TurboNet 95

6 ADAPTIVE MULTIRESOLUTION STRUCTURES
FOR IMAGE SEGMENTATION 103

TABLE OF CONTENTS
(Continued)

Chapter **Page**

6.1 Introduction 103

6.2 Algorithm to Generate RAG Pyramids on TurboNet 105

6.3 Implementation on TurboNet 108

6.4 Performance Results 113

7 CONCLUSIONS 123

APPENDIX A DSP ALGORITHMS 125

REFERENCES 130

LIST OF TABLES

Table **Page**

3.1 Estimated running time for 1-D convolution with method 1 29

3.2 Estimated running time for 1-D convolution with method 2 29

3.3 The time ucal-1 (in nsecs) for 1-D convolution with method 1 30

3.4 The time ucal-2 (in nsecs) for 1-D convolution with method 2 30

3.5 The estimated running time of the hybrid version of the binary-tree algorithm 36

3.6 The estimated running time of the message-passing version of the binary-tree algorithm 37

3.7 The estimated running time of the shared-memory version of the binary-tree algorithm 37

3.8 The time u_l (in nsecs) for the binary-tree algorithm 37

3.9 The time u_{xx2} (in nsecs) for the binary-tree algorithm 38

3.10 The time u_{xx3} (in nsecs) for the binary-tree algorithm 38

3.11 The time u_{xx4} (in nsecs) for the binary-tree algorithm 38

3.12 Execution time for 1-D convolution with method 1 40

3.13 Speedup of 1-D convolution with method 1 40

3.14 Efficiency of 1-D convolution with method 1 40

3.15 Execution time for 1-D convolution with method 2 40

3.16 Speedup of 1-D convolution with method 2 41

3.17 Efficiency of 1-D convolution with method 2 41

3.18 Execution time for the hybrid version of the binary-tree algorithm 44

LIST OF TABLES
(Continued)

Table **Page**

3.19 Speedup of the hybrid version of the binary-tree algorithm 44

3.20 Efficiency of the hybrid version of the binary-tree algorithm 44

3.21 Execution time for the message-passing version of the binary-tree algorithm.........44

3.22 Speedup of the message-passing version of the binary-tree algorithm.....................45

3.23 Efficiency of the message-passing version of the binary-tree algorithm..................45

3.24 Execution time for the shared-memory version of the binary-tree algorithm...........45

3.25 Speedup of the shared-memory version of the binary-tree algorithm.......................45

3.26 Efficiency of the shared-memory version of the binary-tree algorithm....................45

3.27 The communication time (in μsec) between P_i and P_{i+1} in stage 2 49

3.28 The communication time (in μsec) between P_i and P_{i-2} in stage 3 49

3.29 The communication time (in μsec) between P_i and P_{i-4} in stage 4 49

4.1 The estimated running time for 1-D FFT with message-passing..............................61

4.2 The estimated running time for 1-D FFT with shared-memory................................61

4.3 The sequential time u (in nsecs) of 1-D FFT 61

4.4 The time ump (in nsecs) of 1-D FFT with message-passing 61

4.5 The time usm (in nsecs) of 1-D FFT with shared-memory.....................................62

4.6 The estimated running time for 2-D FFT for 2k < (N/P) with shared-memory........64

4.7 The sequential time u (in nsecs) of 2-D FFT 65

4.8 The time uDRAM (in nsecs) for matrix transposition of 2-D FFT...........................65

LIST OF TABLES
(Continued)

Table **Page**

4.9 Execution time for 1-D FFT with message-passing....66

4.10 Speedup of 1-D FFT with message-passing....66

4.11 Efficiency of 1-D FFT with message-passing....67

4.12 Execution time for 1-D FFT with shared-memory....67

4.13 Speedup of 1-D FFT with shared-memory....67

4.14 Efficiency of 1-D FFT with shared-memory....67

4.15 Execution time for 2-D FFT (without matrix transposition)....71

4.16 Speedup of 2-D FFT (without matrix transposition)....71

4.17 Efficiency of 2-D FFT (without matrix transposition)....71

4.18 Execution time for 2-D FFT (with matrix transposition)....71

4.19 Speedup of 2-D FFT (with matrix transposition)....72

4.20 Efficiency of 2-D FFT (with matrix transposition)....72

4.21 Execution time for 2-D FFT with the improved method....74

4.22 Speedup of 2-D FFT with the improved method....75

4.23 Efficiency of 2-D FFT with the improved method....75

5.1 Estimated running time for matrix multiplication on 0-D and 1-D hypercubes....87

5.2 The times $u_{(2p,\,xx)}$ (in nsecs) for matrix multiplication on the 0-D and 1-D hypercubes....87

5.3 The times u_{xx} (in nsecs) for matrix multiplication on the 1-D hypercube....87

LIST OF TABLES
(Continued)

Table **Page**

5.4 Estimated running time for matrix multiplication with the 2-D hypercube..............90

5.5 The times u $u_{(4p, xx)}$ (in nsecs) for matrix multiplication with the 2-Dhypercube......90

5.6 Estimated running time for matrix multiplication with the 3-D hypercube..............93

5.7 The times $u_{(8p, xx)}$ (in nsecs) for matrix multiplication with the 3-D hypercube........94

5.8 Execution time of matrix multiplication on the 1-D hypercube 95

5.9 Speedup of matrix multiplication for the 1-D hypercube 95

5.10 Efficiency of matrix multiplication for the 1-D hypercube 96

5.11 Execution time of matrix multiplication on the 2-D hypercube 98

5.12 Speedup of matrix multiplication for the 2-D hypercube 98

5.13 Efficiency of matrix multiplication for the 2-D hypercube 98

5.14 Execution time of matrix multiplication on the 3-D hypercube 100

5.15 Speedup of matrix multiplication for the 3-D hypercube 100

5.16 Efficiency of matrix multiplication for the 3-D hypercube 100

5.17 Two implementation results of matrix multiplication on the 0-D hypercube.
Cal.: calculation time only, Inp. & Cal.: data input and calculation time...............102

6.1 Function codes for communication operations...111

6.2 Execution time and numbers of pyramid levels for a single processor..................115

6.3 Execution time for the message-passing version with P = 2..................................116

6.4 Execution time for the shared-memory version with P = 2 116

LIST OF TABLES
(Continued)

Table **Page**

6.5 Execution time for the hybrid version with P = 2 117

6.6 Execution time for the hybrid version with P = 2 and separate Phases 4 and 5 117

6.7 Execution time for the message-passing version with P = 4 118

6.8 Execution time for the shared-memory version with P = 4 119

6.9 Execution time for the hybrid version with P = 4 119

6.10 Execution time for the hybrid version with P = 4 and separate Phases 4 and 5 119

6.11 Execution time for the message-passing version with P = 8 120

6.12 Execution time for the shared-memory version with P = 8 120

6.13 Execution time for the hybrid version with P = 8 121

LIST OF FIGURES

Figure Page

1.1 The UMA multiprocessor model 2

1.2 The NUMA multiprocessor model 3

1.3 A multicomputer 4

1.4 Examples of common network topologies 5

2.1 The Hydra block diagram 13

2.2 The interconnection diagram of TurboNet 16

3.1 An example of one-dimensional convolution 19

3.2 An example of 2-D convolution 21

3.3 The data dependencies for convolution of two 4x4 matrices 22

3.4 An example of the binary-tree method 23

3.5 An example of the grid method 25

3.6 An example of the triangle method 26

3.7 DDG of the binary-tree algorithm with 8 processors 31

3.8 Execution time of 1-D convolution (method 1 Vs method 2), with P = 8 41

3.9 A comparison of theoretical and experimental running times for method 1 42

3.10 The difference between theoretical and experimental running times for method 1 42

3.11 The difference of theoretical and experimental running times from method 2 43

3.12 The difference in execution times between the message-passing and hybrid versions, in this order, for $P = 8$. 46

LIST OF FIGURES
(Continued)

Figure **Page**

3.13 A comparison of theoretical and experimental running times of the hybrid version for $P = 8$ 46

3.14 The difference of theoretical and experimental running times from the hybrid version 47

3.15 The difference of theoretical and experimental running times for the message -passing version 47

3.16 The difference of theoretical and experimental running times for the shared-memory version 48

4.1 Data flow graph of an 8-point decimation-in-time DFT 52

4.2 An example of accessing row data for the FFT, with $P = 4$ and $2k < (N/P)$ 56

4.3 An example of transferring data for the FFT, with $P = 4$ and $2k \geq (N/P)$ 56

4.4 DDG of the 1-D FFT with $P = 8$ 58

4.5 DDG of the 2-D FFT algorithm for $2k < (N/P)$ 63

4.6 Execution time for 1-D FFT Vs array size, with $P = 8$ 68

4.7 Comparison of theoretical and experimental running times for 1-D FFT with message-passing 68

4.8 Comparison of theoretical and experimental running times for 1-D FFT with shared-memory 69

4.9 The difference of theoretical and experimental running times for the message-passing version of 1-D FFT 69

4.10 The difference of theoretical and experimental running times for the shared-memory version of the 1-D FFT 70

LIST OF FIGURES
(Continued)

Figure **Page**

4.11 Comparison of theoretical and experimental running times for 2-D FFT with shared-memory 72

4.12 The difference of theoretical and experimental running times for 2-D FFT....73

4.13 Execution time for 2-D FFT vs matrix size, with P = 8....75

4.14 Speed up of 2-D FFT vs matrix size, with P = 8....76

5.1 The decomposition for c_{ij}....79

5.2 The decomposition of the matrices A, B and C for four processors....81

5.3 The communication scheme according to Equation 5.3 82

5.4 The decomposition of the matrices A and B for the 3-D hypercube....82

5.5 The decomposition of matrices A, B and C for the 1-D hypercube....83

5.6 DDG of matrix multiplication for the 1-D hypercube 85

5.7 DDG of matrix multiplication for the 2-D hypercube 89

5.8 DDG of matrix multiplication for P_0 of the 3-D hypercube....91

5.9 Performance comparision of the three versions for the 1-D hypercube....96

5.10 The difference of theoretical and experimental timings for the 1-D hypercube....97

5.11 Performance comparision of the three versions for the 2-D hypercube....99

5.12 Difference of theoretical and experimental timings for the 2-D hypercube 99

5.13 Derformance comparision of the three versions for the 3-D hypercube....101

5.14 Difference of theoretical and experimental timings for the 3-D hypercube 101

LIST OF FIGURES
(Continued)

Figure **Page**

6.1 An example for Phases 4 and 5 107

6.2 Assignment of data for an NxN input image 110

6.3 A 32-bit long communication package 111

6.4 Four types of images used in testing: (a) Uniform, (b) Horizontal half, (c) Three objects, including the background, and (d) Checkerboard 114

6.5 Execution time vs. image types for the hybrid version with P = 4 120

6.6 Performance comparison of the three versions with image d and P = 2 121

6.7 Performance comparison of the three versions with image d and P = 4 122

6.8 Performance comparison of the three versions with image d and P = 8 122

CHAPTER 1

INTRODUCTION

Parallel processing has become the most prominent technology in achieving high performance computational power. One of the key problems to be solved with this technology is to determine how individual processes cooperate with each other efficiently when carrying out a task together. In general, message-passing and shared-memory are two techniques parallel computer systems use for coordination and communication, and they will be the focus of this dissertation. This chapter provides an introductory background in this fast growing research area. The motivations and objectives of our research, as well as an overview of its contributions, are also presented. An outline of the thesis is presented at the end.

1.1 Parallel Processing Systems

A parallel processing system consists of multiple processors (or nodes), memory modules, peripherals, and a switching or interconnection network. There are two major categories in classifying parallel processing systems: *shared-memory multiprocessors* and *message-passing multicomputers* [24]. The difference between them lies in how communication among nodes is carried out. The following two subsections give more details about these two categories.

1.1.1 Shared-Memory Multiprocessors

In a shared-memory multiprocessor multiple processors share a common memory and some peripherals, and communication is performed through the shared memory by writing and reading it. Two shared-memory multiprocessor models are primarily used: the *uniform-memory-access* (UMA) model and the *nonuniform-memory-access* (NUMA) model [36]. They differ in the way the memory and other resources are distributed. In the UMA model, as shown in Figure 1.1, all processors have equal access time to all memory locations in all shared-memory modules (marked as SM) under the condition of no network congestion, and that is why it is called uniform-memory-access model. In the NUMA model in Figure 1.2, however, accessing the local shared memory (marked as LM) is faster than accessing a remote one, because there is no need for a processor to go through the switching network when accessing the former.

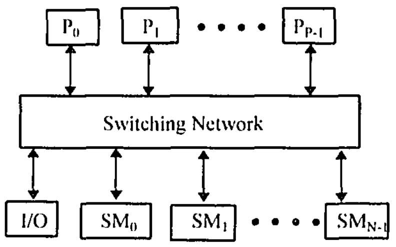


Figure 1.1 The UMA multiprocessor model.

The most popular switching networks are single bus, crossbar, and multistage. The single bus can only handle one transaction at a time, employing a single source. The crossbar and multistage networks, built with extra hardware, can have more than one on-going transaction. Hence, the single bus has low cost and low performance while the other two provide high bandwidth with higher cost.

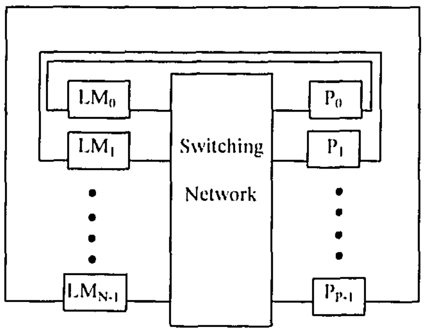


Figure 1.2 The NUMA multiprocessor model.

1.1.2 Message-Passing Multicomputers

A message-passing multicomputer consists of multiple computers (or nodes) interconnected by a point-to-point network, and each node is an autonomous computer including a processor, a private local memory, and possibly disks or I/O peripherals, as modeled in Figure 1.3. Internode communication is carried out by passing messages through the network while observing certain communication protocols. Such actions may involve multiple links (i.e. physical connections between nodes) and nodes, if the source is not directly connected to the destination.

Some common network topologies in constructing interconnection networks for multicomputers are, as shown in Figure 1.4, binary tree, star, ring, mesh, hypercube, etc. They are also called static connection networks because all links between nodes are fixed after a network is built. Among these topologies, the hypercube has very high hardware complexity but it is very popular [38]. A d-dimensional hypercube consists of 2^d nodes, each of which is connected to one other node in each dimension. For example, a 0-dimensional hypercube, a 0-cube for short, has a single node with no communication channels, i.e. it is a standard sequential computer. A 1-cube is constructed from two 0-cubes by connecting them with a single communication channel, and a 2-cube is formed with two 1-cubes by connecting their corresponding nodes via an additional channel. Figure 1.4(e) shows a 3-cube, containing two 2-cubes, and each node in each 2-cube has a connection to the corresponding node in the other 2-cube. Hence, in general, a d-cube is constructed by connecting corresponding pairs of nodes in two (d-1)-cubes with an additional channel. The number of nodes is $P = 2^d$. The number of connections per node and the maximum distance between two nodes are $d = \log_2 P$. The node number (i.e. its identification) is chosen to be a d-bit binary code where its i^{th} bit represents the coordinate of the node in the i^{th} dimension of the hypercube. The binary addresses of any two neighboring nodes differ in a single bit. The number of bits that differ between the addresses of two nodes gives the distance between them.

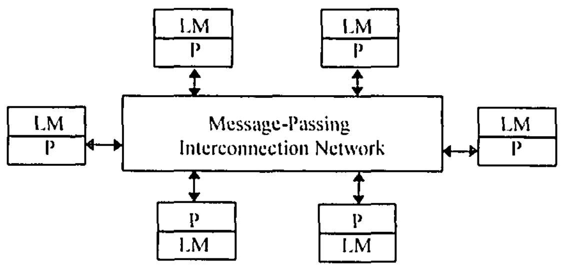


Figure 1.3 A multicomputer.

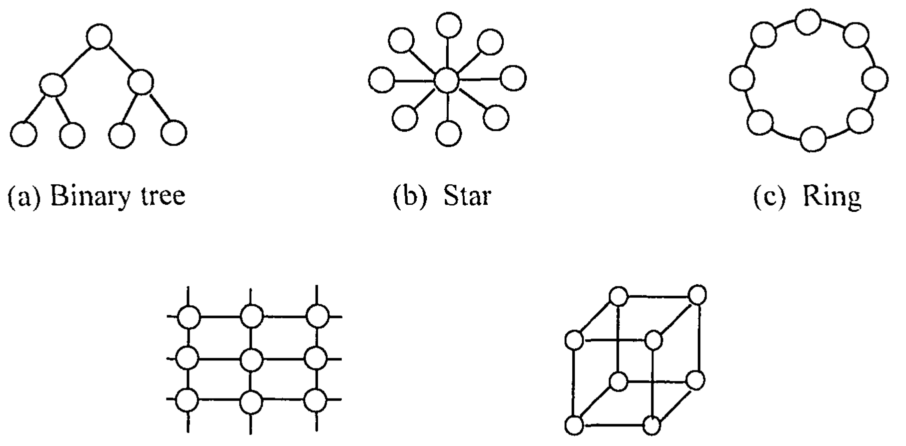


Figure 1.4 Examples of common network topologies.

1.2 Message-Passing Versus Shared-Memory

In a general sense, the message-passing architecture is efficient for communicating small amounts of data in small distance. On the other hand, shared-memory is primarily used for I/O with the host, and for otherwise distant communications with large amounts of data. Additionally, the shared-memory paradigm simplifies the development of algorithms.

Chandra et. al. [10] studied the strengths and weaknesses of these two fundamental mechanisms of message-passing and shared-memory by comparing the performance of equivalent, well-written message-passing and shared-memory programs running on similar hardware. Each application program was produced in two versions and its performance was measured on closely-related simulators of a message-passing and a shared-memory machine. They found that three of the four shared-memory programs ran at roughly the same speed as their message-passing equivalents, even though their communication patterns were different. Therefore, the advantage of message-passing over shared-memory, or vice-versa, is not clear-cut, and this strongly suggests that parallel computers should support, if possible, both communication mechanisms.

Many parallel computer systems have been developed since the early 1980's, such as the nCUBE, Intel iPSC, Wavetracer, Cray Y-MP, MultiMax, Symmetry, and Connection-Machines CM-2 and CM-5. They primarily implement either the message-passing or the shared-memory parallel processing paradigm. For example, the nCUBE, iPSC, and Connection Machine CM-2 [9] are message-passing machines, whereas the Cray Y-MP employs a shared-memory architecture. Many parallel Digital Signal Processing (DSP) algorithms have also been developed for either one of the two paradigms. For instance, Cvetanovic [7] analyzed the performance of a Fast Fourier Transform (FFT) algorithm for a shared-memory parallel architecture, and Fox [4] developed a matrix multiplication algorithm and an FFT algorithm targeting a message-passing architecture.

Recently, the complementary nature of the shared-memory and message-passing communication mechanisms, together with the advance of underlying hardware technologies have led to a growing interest in hybrid architectures supporting both paradigms within a single parallel system. Emphasis has been given to the creation of distributed shared-memory systems, where all local memories of all processors in a message-passing system form a global address space. The underlying message-passing architecture is chosen because it supports system scalability. The Stanford FLASH Multiprocessor [11] is one of the few hybrid architectures of this kind that appeared in recent years. It integrates message-passing with shared-memory in a single architecture by using a programmable node controller, called MAGIC, to efficiently support both communication paradigms. MAGIC provides a tight coupling with the network and memory, and allows for efficient protocol handling through parallel implementation of control processing and data movement. Cache coherence for the shared memory can be carried out even during message transfer. Alewife [12] is another example of a hybrid architecture. Each Alewife node has a hardware controller to handle cache coherence, and a DMA unit (within the controller) to facilitate message passing. In addition, the main processor has an efficient memory-mapped interface to the controller that is used for controlling message sends. However, both systems are based on a message-passing architecture; shared-memory transactions are simulated by software.

In related work, Dowling et. al. [3] introduced and analyzed the Hybrid Array Ring Processor (HARP) architecture. HARP is an application specific architecture centered around a host processor, shared memory, and a set of memory-mapped processors connected into both an open backplane and a bidirectional systolic ring. The architecture implements both the message-passing and shared-memory paradigms and has been benchmarked through simulation of matrix multiplication, FFT, QR Decomposition, and Singular Value Decomposition algorithms, assuming the TMS34082 floating point RISC processor. The HARP system has not been built. In these simulations, the shared memory is used only for the purpose of distributing input data and collecting final results;

data transfers during the execution of algorithms are all done through the systolic links of HARP. Thus, the HARP project does not integrate the message-passing and shared-memory paradigms during the execution of algorithms.

1.3 Motivation, Objectives, and Contributions

It is our vision that a hardware combination of message-passing with shared-memory has potential advantages for many applications, and therefore should be investigated extensively for small-scale and medium-scale parallel computers. The current version of our TurboNet system, the target system in this dissertation, is an asynchronous three-dimensional hypercube system composed of eight powerful Texas Instruments TMS320C40 (C40 in brevity) DSP processor chips. We have built this system with two VME Hydra boards of Ariel Corp., where each board contains four C40's and shared-memory, in addition to local memory attached to each C40. The shared-memory of each board is also global so that it can be accessed by any of the eight processors via the shared VME bus. TurboNet has a more general architecture that implements directly in hardware both communication paradigms, in contrast to the proposed FLASH, Alewife, and HARP systems. Details about TurboNet follow in the next section.

The objectives of this dissertation are: (1) to employ simultaneously both the message-passing and shared-memory paradigms in the development of parallel DSP algorithms, such as one-dimensional (1-D) and two-dimensional (2-D) FFT and convolution, matrix multiplication, and image segmentation; (2) to develop highly accurate methodology for the theoretical assessment of algorithms during their design; (3)

to compare the performance of our algorithms that employ the hybrid communication paradigm (i.e. both the message-passing and shared-memory paradigms) with the performance of algorithms supporting only one of the two basic paradigms, in order to illustrate the superiority of the former.

Our results prove that several algorithms can take advantage of the hybrid architecture of our target system in order to achieve very impressive speedups. Therefore the main conclusion of this dissertation is that small-scale and medium-scale parallel computers should support directly in hardware, if possible, both the message-passing and shared-memory paradigms for good performance and versatility in algorithm development.

1.4 Outline

This dissertation is organized as follows. Following this introduction, Chapter 2 provides a brief overview of our target TurboNet system, including a overview of the C40 and the Hydra architecture, and the main characteristics of the TurboNet system. In Chapters 3, 4, 5, and 6, algorithms for convolution, FFT, matrix multiplication, and image segmentation are introduced, and relevant performance analysis and experimental results are included. Chapter 7 presents the conclusions and further research objectives.

CHAPTER 2

TURBONET: A MESSAGE-PASSING AND SHARED-MEMORY HYBRID ARCHITECTURE

The TurboNet system is presented in this chapter. Three main aspects of the system are discussed here: the TMS320C40 DSP's, the Hydra boards, and the host board. From the architecture point of view, the TurboNet system implements in hardware both the message-passing and shared-memory paradigms, and this is what distinguishes it from other systems.

2.1 The Texas Instruments TMS320C40 Processor

The TurboNet system currently contains eight C40's. The C40 has the following key features [32]:

1. Six 8-bit bidirectional half-duplex communication ports for high speed interprocessor communication.
2. Six-channel DMA coprocessor for concurrent I/O and CPU operation, thereby maximizing sustained CPU performance by alleviating the CPU of burdensome I/O.
3. High-performance DSP CPU capable of 275 MOPS and 320 Mbytes/sec of data-transfer rate.

4. Two identical external data and address buses supporting shared-memory systems and high data rate, single-cycle transfers. The 32 bit data buses are called the Global Bus and the Local Bus. Each bus is capable of addressing 2 Gwords (x32 bits) of address space for a total of 4 Gwords addressable by each C40.
5. On-chip analysis module supporting efficient, state of the art parallel processing debugging.
6. On-chip program cache and dual-access/single-cycle RAM (2 KWords) for increased memory access performance.
7. Separate internal program, data, and DMA coprocessor buses for support of massive concurrent I/O of program and data throughput, thereby maximizing sustained CPU performance.

2.2 Hydra Boards

Hydra[31] is a single-slot 6U VME-based multi-digital signal processor (DSP) board, consisting of a base card and a daughter card, each one containing two C40 chips. Its block diagram is shown in Figure 2.1.

The global or local bus of each C40 is connected to a bank of fast zero-wait-state static memory which is private to the specific C40, that is, any of the DSP's can access its own SRAM without affecting the operations in any of the other DSP's. Each of the DSP's is, in fact, an independent processing node. Each of the DSP's can have 16 Kwords, 64 Kwords, or 256 Kwords (x32 bits) of SRAM in each of its two banks. In our system, the

SRAM's in Hydra 1 all contain 64 Kwords while there are 16 Kword SRAM's in Hydra 2.

Hydra has an Internal Shared Bus (ISB) to which the VME bus and the Global bus of each DSP are attached. The ISB provides access to a bank of DRAM (1 Mwords, 4 Mwords, or 16 Mwords in size; 1 Mwords in our system) and other shared resources. Any external VME master can also access the ISB. Isolation transceivers have been utilized to isolate all DSP modules and the VME bus from the ISB when they are not currently granted the ISB.

ISB requests from the VME bus always take higher priority over local DSP requests. ISB requests from the Hydra DSP's are arbitrated on a rotating priority scheme to make it impossible for one internal master to lock-out or 'starve' the other devices of the ISB. The ownership of the ISB is determined on a cycle-by-cycle basis. If the current master is a local DSP, ownership is challenged after that DSP completes its current ISB cycle. If that DSP is accessing DRAM or any ISB facility, the arbiter will test for new requesters between each access, even if the current owner is operating the DRAM in page mode.

Each C40 has six communication ports. Three of them are used within the board to implement a fully-connected system with four C40's. The remaining three communication ports are brought out to the front panel of Hydra for connections with multiple Hydra boards via microminiature, 25-pin D-sub connectors.

A monitor program, including a bootstrap, is stored in the EPROM attached to DSP1. The bootstrap will set up the VME interface, including the VME memory window as seen by the internal ISB masters and the base offset of Hydra as seen by another VME bus master. In addition to the VME interface, the monitor also initializes the RS-232 port.

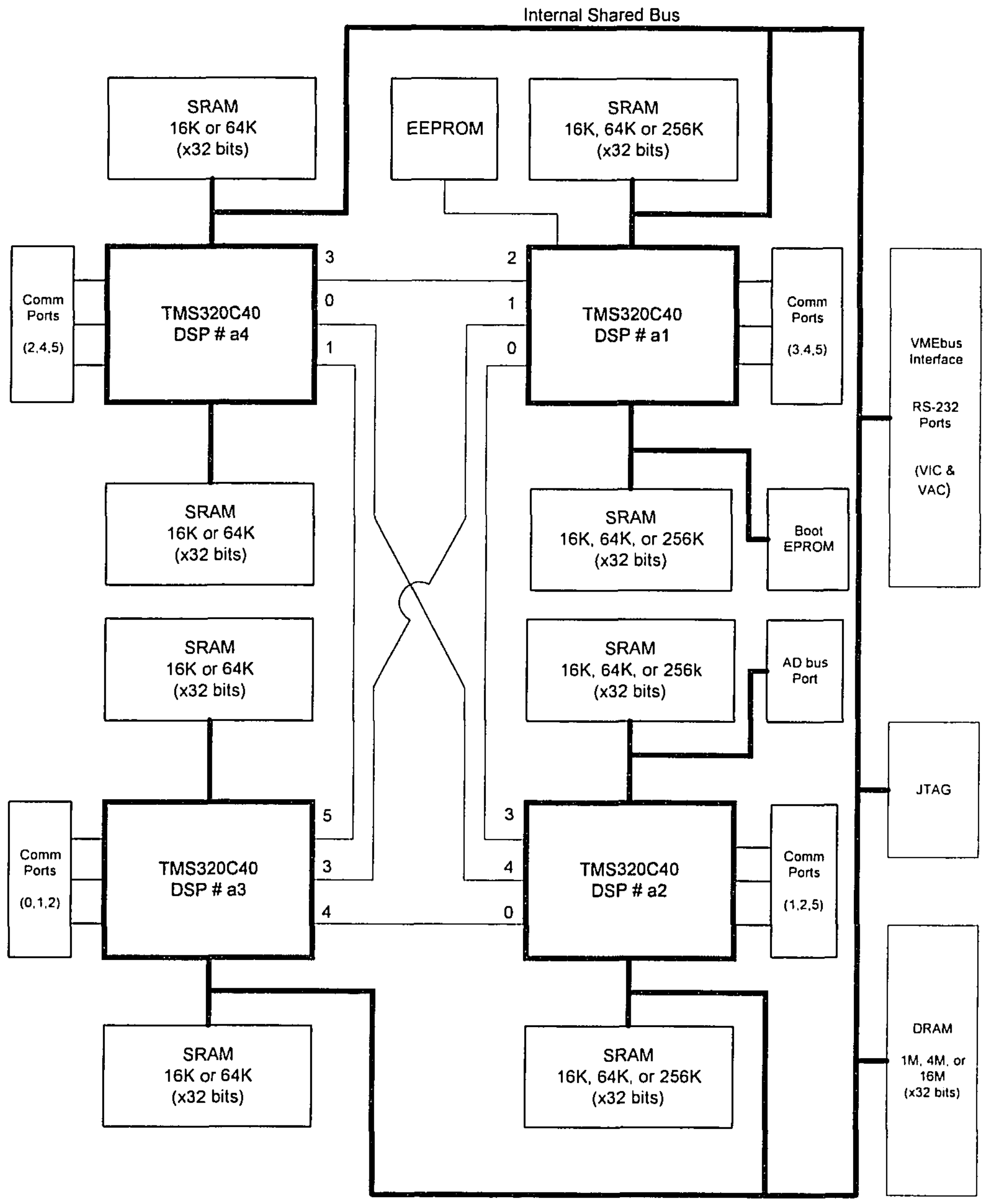

Internal Shared Bus
SRAM 16K or 64K (x32 bits)
EEPROM
SRAM 16K, 64K or 256K (x32 bits)
Comm Ports (2,4,5)
TMS320C40 DSP # a4
3
0
1
2
1
0
TMS320C40 DSP # a1
Comm Ports (3,4,5)
VMEbus Interface
RS-232 Ports
(VIC & VAC)
SRAM 16K or 64K (x32 bits)
SRAM 16K, 64K, or 256K (x32 bits)
Boot EPROM
SRAM 16K or 64K (x32 bits)
SRAM 16K, 64K, or 256k (x32 bits)
AD bus Port
JTAG
Comm Ports (0,1,2)
TMS320C40 DSP # a3
5
3
4
3
4
0
TMS320C40 DSP # a2
Comm Ports (1,2,5)
DRAM
1M, 4M, or 16M (x32 bits)
SRAM 16K or 64K (x32 bits)
SRAM 16K, 64K, or 256K (x32 bits)

Figure 2.1 The Hydra block diagram.

After DSP1 is boot up by bootstrap, it will boot up the other three DSP's through their parallel communication ports. All the necessary configuration information for Hydra is stored in an EEPROM that is also attached to DSP1.

In our system, the Hydra device driver and the utility library are both made available to users. The former is a traditional UNIX device driver while the latter is a resource-mapping library, and they both are independent to each other. The driver can be a loadable one or it can be configured into the SunOS kernel.

2.3 The Host System

The host system, a SPARC CPU-2CE board, is a complete VME-based SPARCstation 2 architecture with Sbus expansion. It offers the same I/O interface as the SPARCstation 2, including DMA supported SCSI and Ethernet ports, along with audio, keyboard/mouse, and two serial channels with full modem support. Two Sbus sockets allow the installation of standard Sbus modules, such as graphics frame buffers, or accelerators, or any other of over 300 Sbus cards available. In our case, these two sockets are used for a monitor and a cache memory expansion card.

2.4 The Hybrid Architecture of the TurboNet System

The TurboNet system comprises a VME backplane, a SPARC CPU-2CE [29] host board, two Hydra boards, two hard disk drives, a floppy drive, a CDROM, a VME bus logic analyzer, and a dumb terminal, as depicted in Figure 2.2. There are four bidirectional links between the two Hydra boards, each of them connecting two C40's together in the two different boards. As mentioned earlier, three of the six communication ports of each DSP are dedicated to the interconnection of the four DSP's inside Hydra. A fourth port is used for an interboard connection. Ignoring the diagonal intraboard connections, the processors form a 3-dimensional hypercube. It is our intention to build a larger hypercube system with up to 64 C40's, thus the diagonal intraboard connections will be removed in the future in order to facilitate this goal. Therefore, these diagonal connections are ignored in this research.

As shown in Figure 2.1, each Hydra board has a shared memory (the DRAM) connected to all four DSP's by a single bus, the ISB. It can be accessed by the DSP's from inside and outside the board. The DSP's, however, when accessing the shared memory from the other board, would have to go through the VME bus before reaching it, i.e. the access time will be longer compared to accessing the local shared memory. This indicates that we have a UMA shared-memory model when dealing with one board and a NUMA shared-memory model when using both boards. And let us not forget that all these coexist with the message-passing hypercube connection in our TurboNet system. Both communication mechanisms (paradigms) can perform their functions at the same time, so that DSP's can talk to each other through communication ports and have their DMA controllers competing for the shared memory via their ISB's and/or VME bus. This hybrid architecture is like adding a new communication media to a system which has only one overcrowded communication space. It also provides options of communication paradigms to applications that may have interconnection bias in performance, such as 1-D FFT, as explained in Chapter 4.

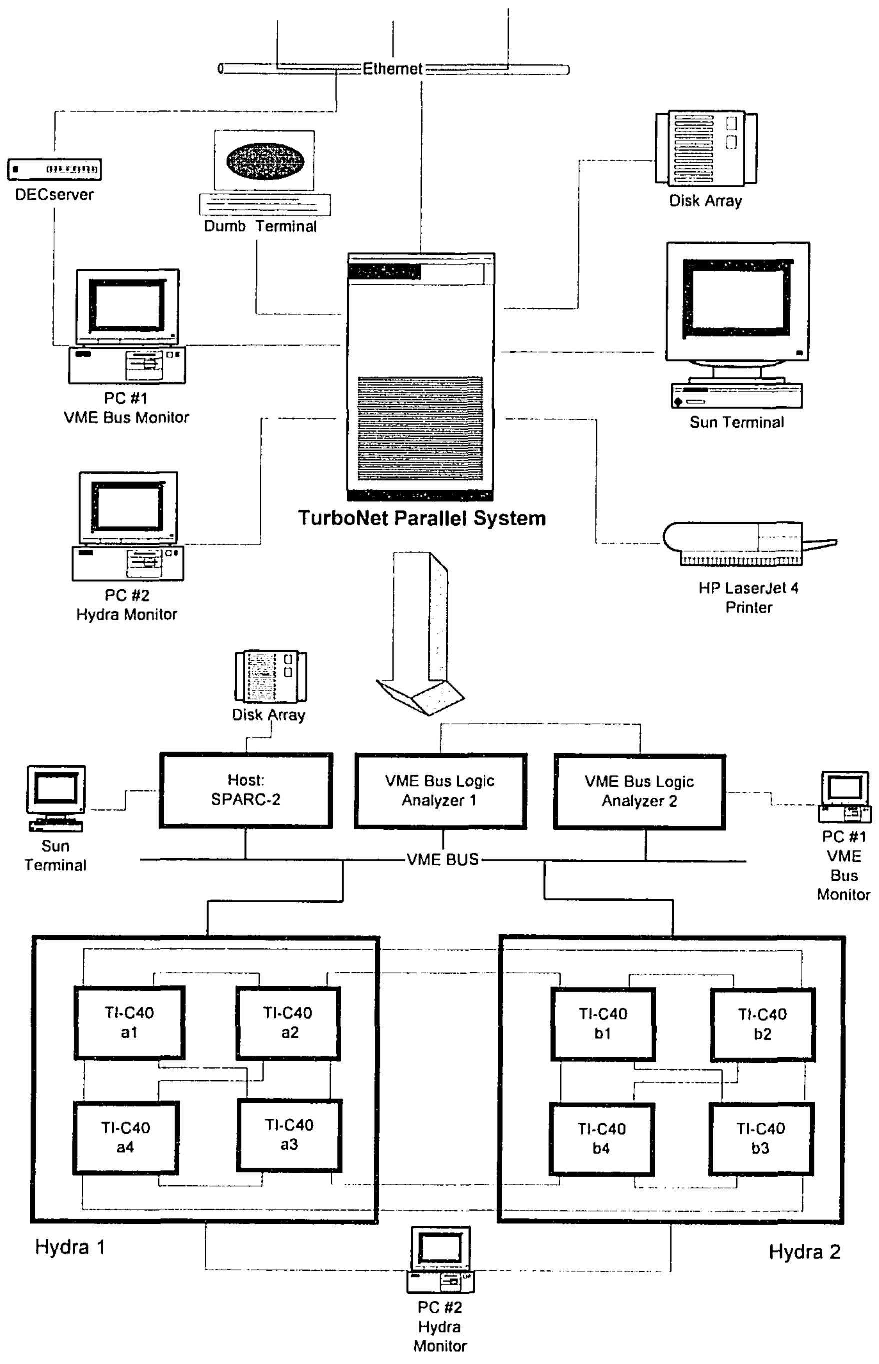


Figure 2.2 The interconnection diagram of TurboNet.

In the course of the system integration, two major problems occurred. The two Hydra boards are different versions, and the older one can take only the low base address in the VME bus while the recent version can use any base address available. The other problem is the malfunction of three communication ports in DSP7 (b3) of the second Hydra board. Due to this reason, its connection with DSP4 (a4) of the first Hydra board goes through DSP5 (b1) of the second Hydra board instead of going to a4 directly. This is not shown in Figure 2.2.

CHAPTER 3

MULTIDIMENSIONAL CONVOLUTION

In this chapter algorithms are presented for 1-D and 2-D convolution on the TurboNet system. Theoretical analysis of the algorithms and performance results of their implementation are also included. All running times in this thesis are expresses in seconds unless specified otherwise. There are two algorithms for 1-D convolution. They do not require any communication between processors. For 2-D convolution, all three communication paradigms have been implemented, namely message-passing, shared-memory and a combination of both (the hybrid paradigm).

3.1 One-Dimensional Convolution Algorithms

Given two sequences of numbers $f[0], f[1], \ldots, f[N-1]$ and $g[0], g[1], \ldots, g[M-1]$, 1-D convolution produces a sequence y of length $N+M-1$, where

$$y[i] = \sum_{j=0}^{N-1} f[j]g[i-j] \tag{3.1}$$

for $i = 0, 1, \ldots, N+M-2$. Let us consider an example of this convolution with $f = \{1,4,2\}$ and $g = \{1,2,3,1\}$. The result of convolution is shown in Figure 3.1.

From the figure we can see that each element of y is generated by one or more multiplications of some elements from f and g, and zero or more subsequent additions. Suppose we have two processors to perform this convolution. One way to implement this is to let processor P_0 calculate $y[0]$, $y[1]$, $y[4]$ and $y[5]$, while let processor P_1 calculate $y[2]$ and $y[3]$. Another way is to let processor P_0 calculate $y[0]$, $y[1]$ and $y[2]$, while let processor P_1 calculate $y[3]$, $y[4]$ and $y[5]$. In *method* 1, if there are P processors, where $1 < P \leq M$, each one of them calculates M/P results in the first phase. For example, P_0 calculates $y[0], y[1], \ldots, y[M/P\text{-}1]$, P_1 calculates $y[M/P], y[M/P+1], \ldots, y[2M/P\text{-}1]$, etc. In the second phase, each of the first $\lceil (N-1)P/M \rceil$ processors calculates up to M/P results from $y[M]$ to $y[M+N\text{-}2]$. In *method* 2, P_0 calculates the first $(M+N\text{-}1)/P$ results while P_1 calculates the second $(M+N\text{-}1)/P$ results, and so on; we assume that M/P and $(M+N\text{-}1)/P$ are integer numbers. In method 1 each processor has almost the same amount of computation, but in method 2 processors that generate the central portion of the output array have more calculation load than the others, and therefore good load balancing is rarely accomplished. As a result, method 1 will generally have better performance than method 2. In both methods, as the algorithms were designed, no communication is necessary between processors.

$$
\begin{array}{llll}
y[0] = & f[0]g[0] & = & 1\text{x}1 \\
y[1] = & f[0]g[1] + f[1]g[0] & = & 1\text{x}2 + 4\text{x}1 \\
y[2] = & f[0]g[2] + f[1]g[1] + f[2]g[0] & = & 1\text{x}3 + 4\text{x}2 + 2\text{x}1 \\
y[3] = & f[0]g[3] + f[1]g[2] + f[2]g[1] & = & 1\text{x}1 + 4\text{x}3 + 2\text{x}2 \\
y[4] = & f[1]g[3] + f[2]g[2] & = & 4\text{x}1 + 2\text{x}3 \\
y[5] = & f[2]g[3] & = & 2\text{x}1
\end{array}
$$

Figure 3.1 An example of one-dimensional convolution.

3.2 Two-Dimensional Convolution Algorithm

The basic definition of two-dimensional convolution is

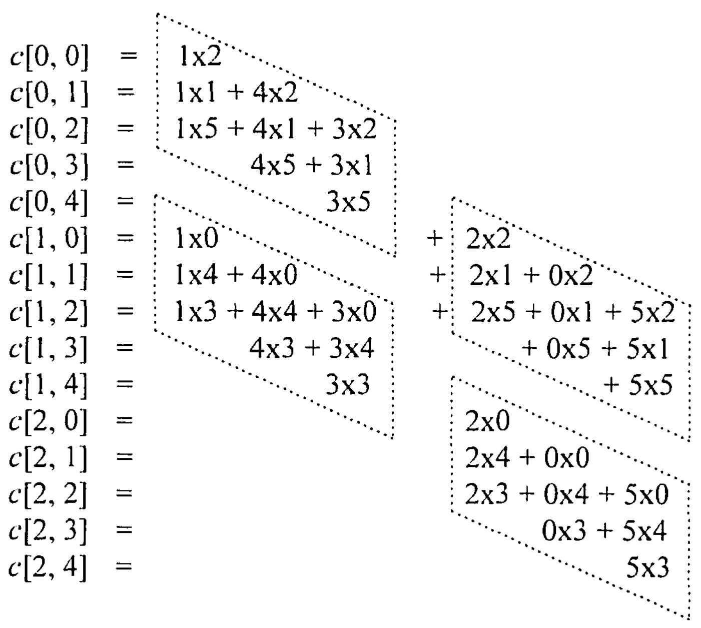
$$c[x,y] = \sum_{i=0}^{N_1-1} \sum_{j=0}^{N_2-1} f[i,j]g[x-i,y-j] \tag{3.2}$$

where $x = 0, 1, 2, \ldots, N_1 + M_1 - 2$, $y = 0, 1, 2, \ldots, N_2 + M_2 - 2$, and the lengths of $f[n_1, n_2]$ and $g[m_1, m_2]$ in the first dimension are N_1 and M_1, respectively, whereas they are N_2 and M_2, respectively, in the second dimension. In the following discussion we assume that N_1 and M_1 are powers of 2.

As an example, let us consider two-dimensional convolution with $f = \begin{Bmatrix} 1,4,3 \\ 2,0,5 \end{Bmatrix}$ and $g = \begin{Bmatrix} 2,1,5 \\ 0,4,3 \end{Bmatrix}$. Figure 3.2 (a) shows the operations to be performed. If operations involving distinct pairs of rows from f and g, respectively, are grouped together with distinct parallelograms, then four such parallelograms are obtained, as in Figure 3.2 (b). If we rotate this figure counter-clockwise by 90 degrees, then the parallelograms derived for the convolution of two 4x4 matrices and the data dependencies for obtaining the resulting matrix are shown in Figure 3.3. Again each parallelogram in Figure 3.3 represents the convolution of a pair of rows from f and g, respectively.

$$
\begin{array}{lllll}
c[0,0] = & f[0,0]g[0,0] & & & \\
c[0,1] = & f[0,0]g[0,1]+f[0,1]g[0,0] & & & \\
c[0,2] = & f[0,0]g[0,2]+f[0,1]g[0,1]+f[0,2]g[0,0] & & & \\
c[0,3] = & f[0,1]g[0,2]+f[0,2]g[0,1] & & & \\
c[0,4] = & f[0,2]g[0,2] & & & \\
c[1,0] = & f[0,0]g[1,0] & +f[1,0]g[0,0] & & \\
c[1,1] = & f[0,0]g[1,1]+f[0,1]g[1,0] & +f[1,0]g[0,1]+f[1,1]g[0,0] & & \\
c[1,2] = & f[0,0]g[1,2]+f[0,1]g[1,1]+f[0,2]g[1,0] & +f[1,0]g[0,2]+f[1,1]g[0,1]+f[1,2]g[0,0] & & \\
c[1,3] = & f[0,1]g[1,2]+f[0,2]g[1,1] & +f[1,1]g[0,2]+f[1,2]g[0,1] & & \\
c[1,4] = & f[0,2]g[1,2] & +f[1,2]g[0,2] & & \\
c[2,0] = & & f[1,0]g[1,0] & & \\
c[2,1] = & & f[1,0]g[1,1]+f[1,1]g[1,0] & & \\
c[2,2] = & & f[1,0]g[1,2]+f[1,1]g[1,1]+f[1,2]g[1,0] & & \\
c[2,3] = & & f[1,1]g[1,2]+f[1,2]g[1,1] & & \\
c[2,4] = & & f[1,2]g[1,2] & &
\end{array}
$$

(a) An expansion of Equation 3.2 with the data arrays f and g.



(b) The convolution parallelograms.

Figure 3.2 An example of 2-D convolution.

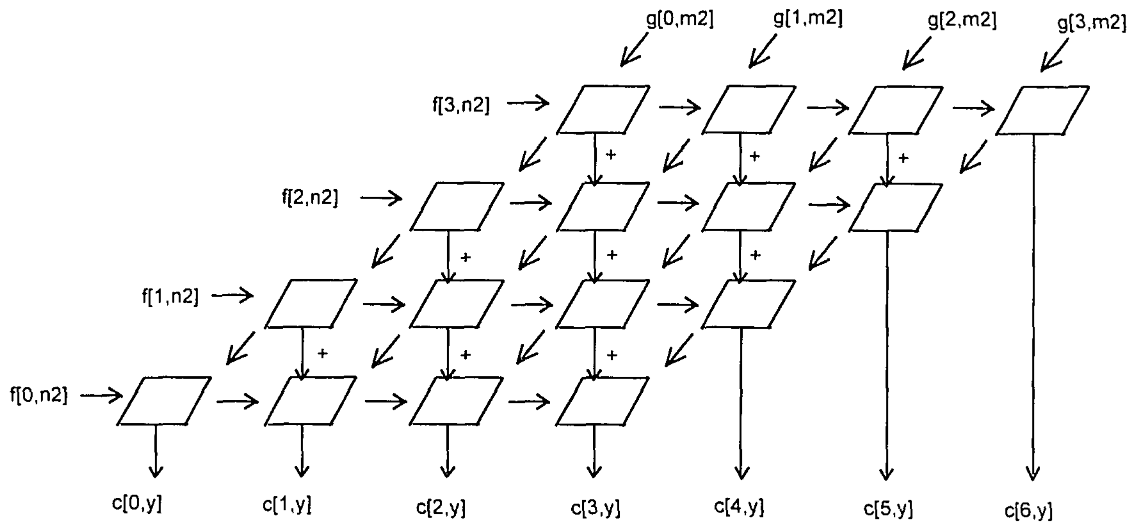


Figure 3.3 The data dependencies for convolution of two 4x4 matrices.

Suppose we have a hypercube system of four processors to perform this convolution. Each processor could be responsible for the calculations in one row of parallelograms in Figure 3.3. The final result will spread unevenly among the processors, so that processor P_0 will have $c[0,y]$, $c[1,y]$, $c[2,y]$ and $c[3,y]$, whereas processors P_1, P_2 and P_3 will have $c[4,y]$, $c[5,y]$ and $c[6,y]$, respectively. This is shown in Figure 3.4. In the first stage of this algorithm, each processor performs the operations of all the parallelograms assigned to it. In the second stage, processors P_1 and P_3 send the results of their first three parallelograms to processors P_0 and P_2, respectively, and the receivers add them to their corresponding results. In the final stage, processor P_2 sends the results of its first two parallelograms to processor P_0 and the third one to processor P_1, and again the receivers add them to their corresponding results. We call this method of parallel convolution the *binary-tree method* because of the pairwise additions in rows.

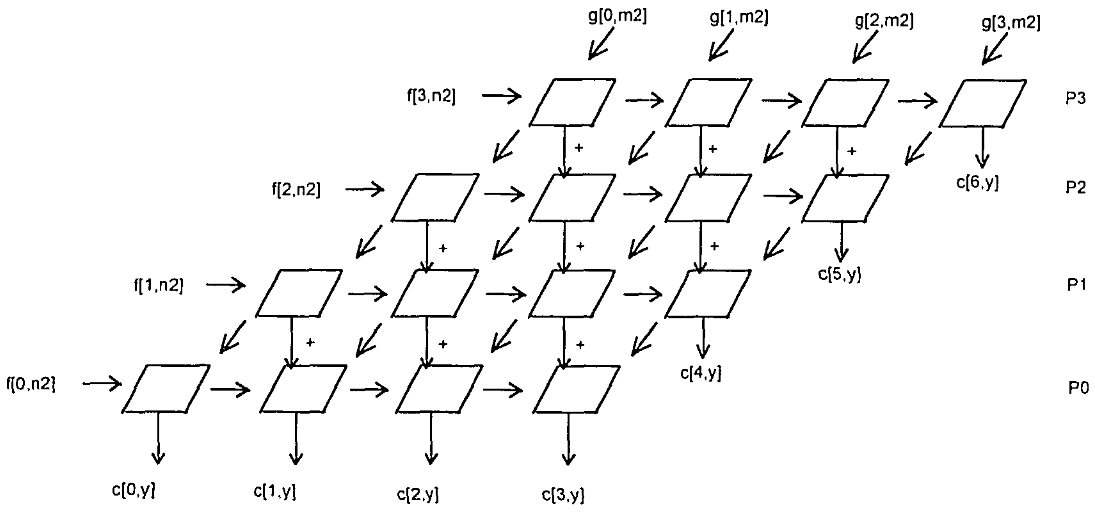


Figure 3.4 An example of the binary-tree method.

It is important to mention that data transfers can be avoided altogether by combining pairs of columns, so that the number of parallelograms in each column becomes N_1 (similar to method 1 of 1-D convolution). This is accomplished in Figure 3.3 by combining the linear arrays that produce the pairs ($c[0,y]$, $c[4,y]$), ($c[1,y]$, $c[5,y]$) and ($c[2,y]$, $c[6,y]$). However, such an implementation is neglected here because programming such a task is more difficult in the general case. Also such an implementation could not be used to investigate the capabilities of TurboNet's hybrid communication architecture.

With P processors ($P > 1$), where P is assumed to be a power of 2, the binary-tree algorithm will have $\log_2 P + 1$ stages, and each processor will have N_1/P rows of parallelograms (N_1/P is an integer). P_0 will have $c[0,y]$, $c[1,y]$, ..., $c[M_1\text{-}2+N_1/P, y]$ of the final result, and P_i will have $c[M_1\text{-}1+ i\ N_1/P, y]$, ..., $c[M_1\text{-}2+ (i+1)N_1/P, y]$, for i = 1, 2, ..., P-1.

Now suppose we have a 4-dimensional hypercube with 16 processors. To perform the two-dimensional convolution, we can assign each of the 16 parallelograms in Figure 3.3 to a processor in the hypercube, so that the convolution of $f[3, n_2]$ and $g[0, m_2]$ goes to processor P_0, the convolution of $f[3, n_2]$ and $g[1, m_2]$ goes to processor P_1 and so on, as shown in Figure 3.5. The arrows for values of f and g do not represent systolic array processing as in [37, 38, 39]. Each arrow implies that all processors on the corresponding row or column initially obtain the same values of f or g. After each processor finishes its 1-D convolution, except for those processors on the south and east borders, it has to send the result to its south-east neighbor, and the final result will be distributed among the processors on the south and east borders. We call this the *grid method.*

In the grid method, all communications are non-local, and they are most likely done efficiently through shared-memory. A binary Gray code is used for the assignment of parallelograms to processors, and each communication is between processors whose distance is equal to 2. The Gray codes for rows and columns are chosen, so that each set of four processors in each quadrant belong to the same Hydra board. This scheme minimizes the number of VME bus accesses.

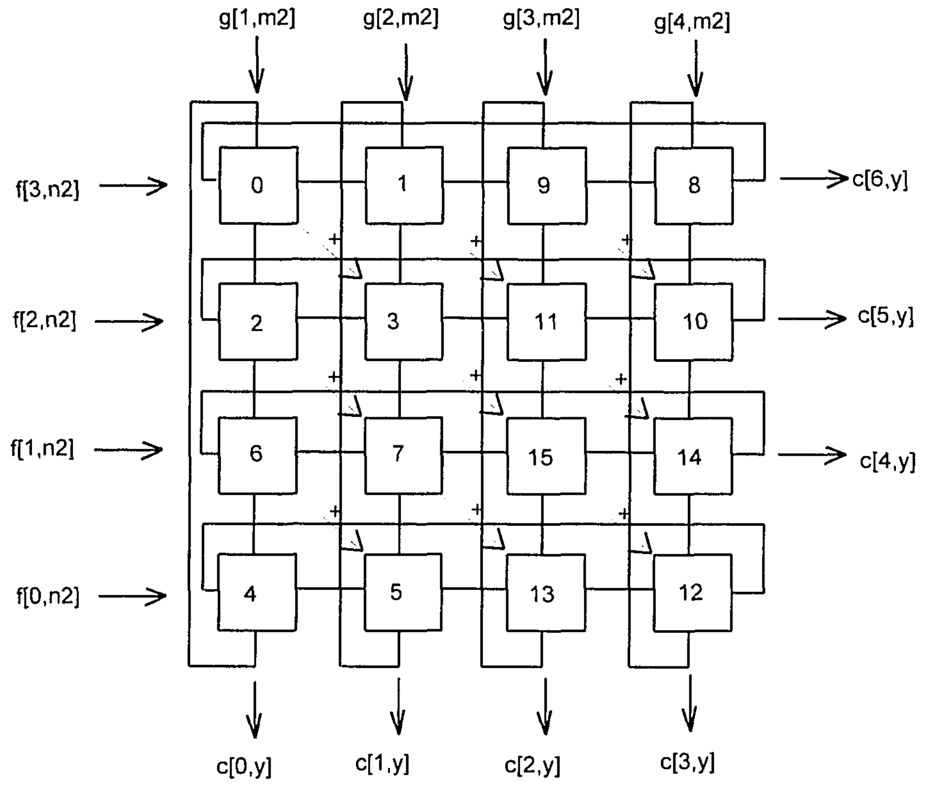


Figure 3.5 An example of the grid method.

To also make efficient use of the message-passing capability of our target system, we invent another method, called the *triangle method*. The profile in Figure 3.3 is also a parallelogram. If we cut its right angle along the dashed line and paste it in the left side of the parallelogram, we have a square configuration. We then assign each parallelogram to a processor of the hypercube as shown in Figure 3.6. In the figure each row and column number is transformed according to the binary reflected Gray code. To obtain the address of a hypercube processor, we concatenate the corresponding transformed column and row numbers. This process results in optimal mapping of a mesh to the hypercube. Again, the address assignment in this algorithm intents to let all the processors in the same column of Figure 3.6 share the same-board DRAM, and therefore avoid going through the VME bus.

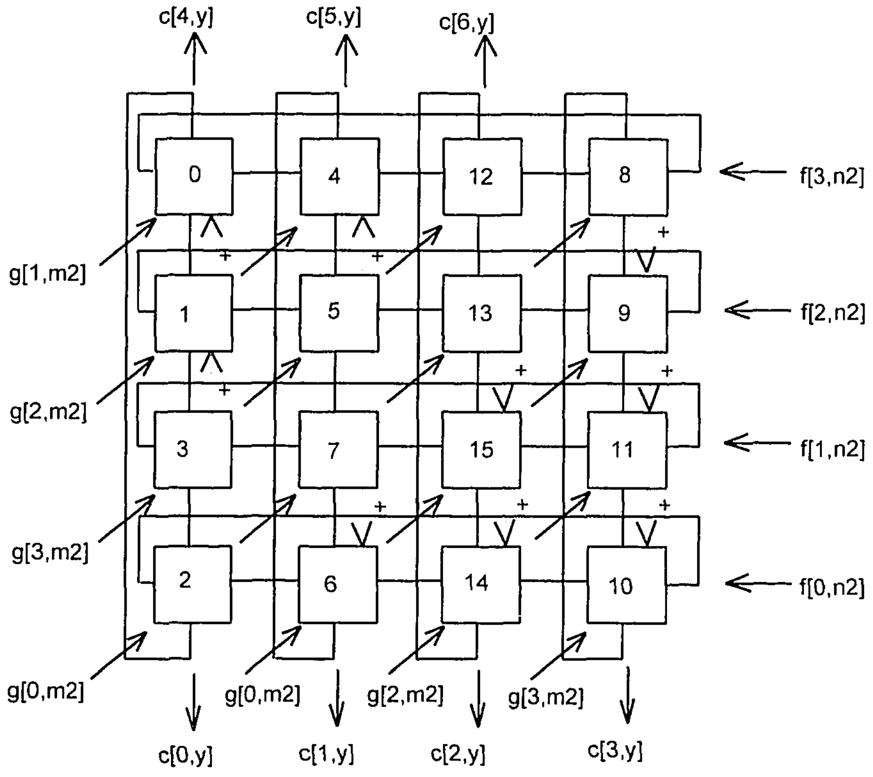


Figure 3.6 An example of the triangle method.

After each processor is done with its 1-D convolution, all processors in the lower-right triangle (including the diagonal processors) transfer their results to their lower-vertical neighbors, while the processors in the upper-left triangle send theirs to their upper neighbors, as indicated in Figure 3.6. The final result will be distributed among the processors in the upper and lower sides of the mesh. All data transfers are between neighbors.

In general, for both the grid and triangle methods each processor might be assigned more than one parallelogram. In this case, after calculating the convolutions of its assigned parallelograms, the processor has to transfer different parallelograms to different processors. We did not implement these two methods. Only the binary-tree method was implemented on TurboNet because of its simplicity in programming. We, however, consider these as interesting problems for further research. Algorithms for all methods are shown in Appendix A.

3.3 Theoretical Analysis

Theoretical analysis for 1-D and 2-D convolution is presented in this section. Since the algorithms for 1-D convolution do not need any communication, no Data Dependence Graph (DDG) is presented. For the algorithm of 2-D convolution, three communication strategies are discussed and the DDG is also provided for theoretical estimation of the algorithm's performance.

3.3.1 1-D Convolution

1-D convolution of two arrays is as defined in Equation 3.1. Its running time on a single processor is

$$t_{\mathrm{sq1}}(M) = 2\,MN\,t_{\mathrm{cal}} = O(M^2) \qquad (3.3)$$

for $M \geq N$, where M and N are the sizes of the two data arrays, and t_{cal} is the time for one multiplication or one addition. Since no communication is involved, the execution times of both methods for 1-D convolution consist of calculation times only. In both methods, let P, the number of processors, be a power of 2, $P \leq M$ and $M \geq N$. In method 1, let M/P be an integer. Because the workload is balanced among the processors, each one of them performs MN/P multiplications and additions. The total running time is therefore

$$t_{m1}(M) = 2\,MN/P\,t_{\mathrm{cal}} = O(M^2/P) \tag{3.4}$$

In method 2, $n = (M + N - 1)/P$ is assumed to be an integer. Here each processor calculates n consecutive elements of the result. Since the numbers of operations required for different output elements may differ, this method does not generally balance the workload among the processors. The processor(s) that produces the central portion of the output array y corresponds to the longest running time. The maximum number of multiplications/additions the processor might have to perform is $Nn-((P/2+1)n-M)((P/2+1)n-M+1)/2$ if $(P/2-1)n < N < nP/2$, or Nn if $N < (P/2-1)n$. Then the running time of method 2 is

$$t_{m2}(M) = \begin{cases} 2\,(Nn-((P/2+1)n-M)((P/2+1)n-M+1)/2\,)\,t_{\mathrm{cal}} \\ 2\,N\,n\,t_{\mathrm{cal}} \end{cases}$$

$$= \begin{cases} O(M^2/P) & \text{if} \quad (P/2-1)n < N < nP/2 \\ O(M^2/P) & \text{if} \quad N < (P/2-1)n \end{cases} \tag{3.5}$$

In Tables 3.1 and 3.2 the calculated running times for both methods of 1-D convolution are presented. They are estimated from the following two equations

$$t_{E-m1}(M) = MN/P\,u_{cal-1}(N,P) \tag{3.6}$$

and

$$t_{E-m2}(M) = \begin{cases} (Nn-((P/2+1)n-M)((P/2+1)n-M+1)/2)\,u_{cal-2}(N,P) \\ \qquad \text{if } (P/2-1)n < N < nP/2 \\ Nn\,u_{cal-2}(N,P) \qquad \text{if} \quad N < (P/2-1)n \end{cases}$$

(3.7)

where $u_{cal-1}(N, P)$ and $u_{cal-2}(N, P)$, obtained from experiment, represent the amount of time for one multiplication and one addition, including the time to load and store the input and output data in methods 1 and 2, respectively. Due to the pipelined structure of the C40 [33], the unit time may change according to the number of operations needed to be performed. This is why the u parameters are functions of N and P. Their values are derived from experiment and are listed in Tables 3.3 and 3.4.

Table 3.1 Estimated running time for 1-D convolution with method 1.

M=512	N=8	N=32	N=128	N=384
P=1	0.005132	0.020509	0.082040	0.246151
P=2	0.003689	0.013160	0.053300	0.174860
P=4	0.001861	0.006710	0.028462	0.090753
P=8	0.000954	0.003519	0.015033	0.046197

Table 3.2 Estimated running time for 1-D convolution with method 2.

M=513	N=8	N=32	N=128	N=384
P=1	0.005132	0.020509	0.082040	0.246151
P=2	0.004470	0.014063	0.052554	0.154966
P=4	0.002266	0.007400	0.032507	0.114776
P=8	0.001112	0.003708	0.016281	0.065480

Table 3.3 The time $u_{\text{cal-1}}$ (in nsecs) for 1-D convolution with method 1.

$u_{\text{cal-1}}$	N=8	N=32	N=128	N=384
P=1	1250.7	1251.8	1251.8	1252.0
P=2	1801.3	1606.5	1626.6	1778.8
P=4	1817.4	1638.2	1737.2	1846.4
P=8	1863.3	1718.3	1835.1	1879.8

Table 3.4 The time $u_{\text{cal-2}}$ (in nsecs) for 1-D convolution with method 2.

$u_{\text{cal-2}}$	N=8	N=32	N=128	N=384
P=2	2178.4	1720.0	1600.7	1573.3
P=4	2178.9	1700.4	1587.3	1566.0
P=8	2138.5	1704.0	1590.0	1563.5

3.3.2 2-D Convolution

The 2-D convolution of two matrices performed on a uniprocessor has running time

$$t_{\text{sq}2}(M_1) = 2N_1N_2M_1M_2t_{\text{cal}} = O(M_1^4) \tag{3.8}$$

where N_1, N_2 , M_1 and M_2 are as defined in Section 3.2, and M_1 is assumed to be the largest among them. The binary-tree algorithm for 2-D convolution, as described in the previous section, has

$$s = \log_2 P + 1 \tag{3.9}$$

stages.

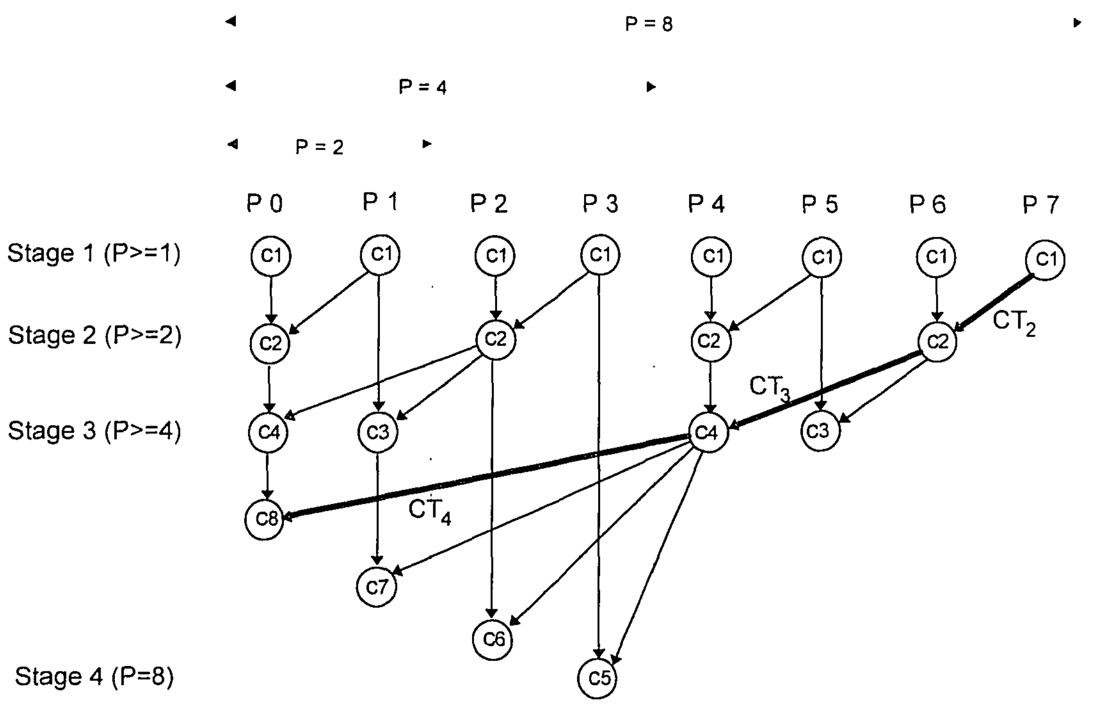


Figure 3.7 DDG of the binary-tree algorithm with 8 processors.

Although the only restriction of the algorithm is that the number of processors P be a power of 2, the DDG in Figure 3.7 is for 8 processors, the number of processors available in TurboNet. This diagram also includes the data dependence information for $P = 2$ and $P = 4$. From Equation 3.9, there are 4 stages in the algorithm for $P = 8$. In stage 1 each processor computes the convolution for each parallelogram assigned to it. As indicated in Section 3.2, each processor has N_l/P rows of parallelograms, and from Figure 3.2 it can be seen that each row contains M_l parallelograms, i.e. each processor is assigned M_lN_l/P parallelograms. Inside each one of them, it is simply a convolution of two rows from the two input matrices, with lengths of N_2 and M_2, respectively, as seen in Figure 3.1. Hence, there are $N_2M_2M_1N_1/P$ multiplications and the same number of additions in stage 1 for each processor. With the same assumption as in Equation 3.8, we have the running time of stage 1 as

$$c_1(M_1) = 2N_2M_2\ M_1N_1/P\ t_{\mathrm{cal}} = O(M_1^4/P) \tag{3.10}$$

The convolution result in each processor is then treated locally as a 1-D array of length $(N_1/P+M_1-1)(N_2+M_2-1)$, where N_2+M_2-1 is the number of elements of the result in the 1-D convolution of vectors (i.e. one parallelogram) having length N_2 and M_2.

In stage 2, each processor P_i whose LSB of its binary address i is 0 (i.e. P_0, P_2, P_4 or P_6) receives from P_{i+1} (i.e. P_1, P_3, P_5 or P_7) the first $(M_1-1)(N_2+M_2-1)$ elements of the array that resulted in stage 1 in P_{i+1}, and adds them to the last $(M_1-1)(N_2+M_2-1)$ elements of its own array from stage 1. Since the binary addresses of P_i and P_{i+1} differ in a single bit, they are neighbors if TurboNet is viewed as a hypercube, and therefore their communications can be easily carried out through direct links between them. This is true only for the hybrid and message-passing versions of the algorithm. In the shared-memory version, because each pair of P_i and P_{i+1} are located in the same board, no accessing of the shared memory through the VME bus is necessary. But the shared memory in each Hydra board is shared by two pairs of P_i and P_{i+1} for $P \geq 4$, therefore delay due to congestion on the ISB occurs (no delay for $P = 2$). As described in Section 2.2, the ownership of the ISB is determined on a cycle-by-cycle basis, so the maximum delay for $P \geq 4$ is about $(M_1-1)(N_2+M_2-1)$ cycles. The running times of the three versions in stage 2 are

$$c_{(2,\mathrm{hy})}(M_1) = (M_1 - 1)(N_2 + M_2 - 1)(t_{\mathrm{cal}} + t_{\mathrm{com}}) = O(M_1^2) \tag{3.11}$$

$$c_{(2,\mathrm{mp})}(M_1) = (M_1 - 1)(N_2 + M_2 - 1)(t_{\mathrm{cal}} + t_{\mathrm{com}}) = O(M_1^2) \tag{3.12}$$

$$c_{(2,\mathrm{sm})}(M_1) = \begin{cases} (M_1 - 1)(N_2 + M_2 - 1)(t_{\mathrm{cal}} + t_{\mathrm{DRAM}}) = O(M_1^2) & \text{for } P = 2 \\ \\ (M_1 - 1)(N_2 + M_2 - 1)(t_{\mathrm{cal}} + t_{\mathrm{DRAM}} + t_{\mathrm{del}}) = O(M_1^2) & \text{for } P \geq 4 \end{cases} \tag{3.13}$$

where hy, mp and sm mean hybrid, message-passing and shared-memory, respectively. Also, t_{com} is the time to transfer a single word between neighbors in the hypercube via the direct link that connects them, and $t_{\mathrm{del}} = t_{\mathrm{DRAM}}$ is the time to transfer the same word but through the shared memory (the DRAM).

In stage 3 each processor P_i whose LSB of its i is 0 and the next bit is 1 (i.e. P_2 or P_6) sends to P_{i-1} (i.e. P_1 or P_5) $N_1/P(N_2+M_2-1)$ elements of its resulting array from stage 2 (starting at element $(M_1-N_1/P-1)(N_2+M_2-1)$) and to P_{i-2} (i.e. P_0 or P_4) the first $(M_1-N_1/P-1)(N_2+M_2-1)$ elements. P_{i-1} and P_{i-2} will then add the received data to their own arrays. The communication process in this stage can be carried out in three ways. In the hybrid way, the communication between P_i and P_{i-1} is done through the shared memory while P_i sends data to P_{i-2} via a direct link between them, and the later is assumed to dominate the running time in this stage. In this case, there is no communication congestion in either the link or the shared memory. In using message-passing, data being sent from P_i to P_{i-1} may need to go through P_{i-2}, that is P_{i-2} is an intermediate node in the path between P_i and P_{i-1}. This implies a delay of $N_1/P(N_2+M_2-1)\, t_{\mathrm{com}}$ in the communication between P_i and P_{i-1}. It is assumed that this is the dominant time. In the shared-memory implementation P_i, P_{i-1}

and P_{i-2} are all in the same board, therefore there will be no accessing to out-board shared memory. The ISB delay due to congestion exists when P_{i-1} and P_{i-2} try to read data at the same time from the on-board shared memory through the ISB bus. Since P_i starts sending data to P_{i-2} only after it finishes with P_{i-1}, P_{i-2} will finish last with a maximum delay of $N_1/P(N_2+M_2-1)\ t_{\text{del}}$. The running times for this stage are

$$c_{(4,\text{hy})}(M_1) = (M_1 - N_1/P - 1)(N_2 + M_2 - 1)(t_{\text{cal}} + t_{\text{com}}) = O(M_1^2) \tag{3.14}$$

$$\begin{aligned} c_{(4,\text{mp})}(M_1) &= (M_1 - N_1/P - 1)(N_2 + M_2 - 1)(t_{\text{cal}} + t_{\text{com}}) + N_1/P(N_2 + M_2 - 1)\ t_{\text{com}} \\ &= O(M_1^2) + O(M_2^2/P) \end{aligned} \tag{3.15}$$

$$\begin{aligned} c_{(4,\text{sm})}(M_1) &= (M_1 - N_1/P - 1)(N_2 + M_2 - 1)(t_{\text{cal}} + t_{\text{DRAM}}) + N_1/P(N_2 + M_2 - 1)\ t_{\text{DRAM}} \\ &= O(M_1^2) + O(M_2^2/P) \end{aligned} \tag{3.16}$$

Stage 4 is similar to stage 3. Each processor P_i whose lowest two bits of i are 0's and the next bit is 1 (i.e. P_4) sends three different sets of $N_1/P(N_2+M_2-1)$ elements of its resulting array from stage 3 (starting at element $(M_1-3N_1/P-1)(N_2+M_2-1)$) to P_{i-1}, P_{i-2} and P_{i-3} (i.e. P_3, P_2 and P_1), and then the first $(M_1-3N_1/P-1)(N_2+M_2-1)$ elements to P_{i-4} (i.e. P_0). The receivers will add the received data to their own data arrays and the algorithm will be terminated at this point. In this stage, all the receivers are in one board while P_i, the sender, is in another board, and P_i has a direct link to P_{i-4} only. Hence, in the hybrid version of the algorithm P_i sends data to P_{i-1}, P_{i-2} and P_{i-3} through the shared memory by using three of its DMA controllers, and then to P_{i-4} via the link. The three DMA controllers work simultaneously and produce congestion delay on the VME bus in this case as much as $2N_1/P(N_2+M_2-1)\ t_{\text{o-DRAM}}$, where $t_{\text{o-DRAM}}$ is the time to access the out-board DRAM for a single value. When using message-passing only, P_i dumps all the data to P_{i-4}, and P_{i-4} then distributes them to P_{i-1}, P_{i-2} and P_{i-3} through communication links. So the delay in the link between P_i and P_{i-4} is $3N_1/P(N_2+M_2-1)\ t_{\text{com}}$. The shared-memory only case is similar to the hybrid one in terms of using the shared-memory, but with a bigger delay of $3N_1/P(N_2+M_2-1)\ (t_{\text{o-DRAM}} + t_{\text{DRAM}})$ in both the VME bus and the ISB for P_{i-4}. Since P_{i-4} will finish last, the running times for stage 4 are

$$\begin{aligned} c_{(8.\text{hy})}(M_1) &= (M_1 - 3N_1/P - 1)(N_2 + M_2 - 1)(t_{\text{cal}} + t_{\text{com}}) + 2N_1/P(N_2 + M_2 - 1)\ t_{\text{o-DRAM}} \\ &= O(M_1^2) + O(M_2^2/P) \end{aligned} \tag{3.17}$$

$$\begin{aligned} c_{(8.\text{mp})}(M_1) &= (M_1 - 3N_1/P - 1)(N_2 + M_2 - 1)(t_{\text{cal}} + t_{\text{com}}) + 3N_1/P(N_2 + M_2 - 1)\ t_{\text{com}} \\ &= O(M_1^2) + O(M_2^2/P) \end{aligned} \tag{3.18}$$

$$\begin{aligned} c_{(8.\text{sm})}(M_1) &= (M_1 - 3N_1/P - 1)(N_2 + M_2 - 1)(t_{\text{cal}} + t_{\text{DRAM}}) \\ &\quad + 3N_1/P(N_2 + M_2 - 1)\ (t_{\text{o-DRAM}} + t_{\text{DRAM}}) = O(M_1^2) + O(M_2^2/P) \end{aligned} \tag{3.19}$$

The running times of P_{i-1}, P_{i-2} and P_{i-3} will be very close to each other in the hybrid and shared-memory cases. In the message-passing case, P_{i-1} will finish first and then P_{i-2} will be followed by P_{i-3}.

From the discussion above we can see that the hybrid version has less delay than the message-passing and shared-memory versions, and therefore would provide better performance over the other two. As described above, the dominant running times in stages 2, 3 and 4 are c_2, c_4 and c_8, respectively, that is, the longest path in the DDG of Figure 3.7 is from c_1 of P_7 to c_2 of P_6, c_4 of P_4 and c_8 of P_0 (as marked by thick lines in the figure). Hence, the overall running times of the binary-tree algorithm with $P = 8$, listed in Tables 3.5 to 3.7, are estimated from the following equations:

$$\begin{aligned} t_{\text{E-hy}}(M_1) = {} & (N_2 M_2 M_1 N_1/P)\, u_1(N,P) + (N_2 + M_2 - 1)((M_1 - 1)\, u_{\text{hy}2}(N,P) \\ & + (M_1 - N_1/P - 1)\, u_{\text{hy}3}(N,P) + (M_1 - 3N_1/P - 1)\, u_{\text{hy}4}(N,P)) \end{aligned} \tag{3.20}$$

$$\begin{aligned} t_{\text{E-mp}}(M_1) = {} & (N_2 M_2 M_1 N_1/P)\, u_1(N,P) + (N_2 + M_2 - 1)((M_1 - 1)\, u_{\text{mp}2}(N,P) \\ & + (M_1 - N_1/P - 1)\, u_{\text{mp}3}(N,P) + (M_1 - 3N_1/P - 1)\, u_{\text{mp}4}(N,P)) \end{aligned} \tag{3.21}$$

$$\begin{aligned} t_{\text{E-sm}}(M_1) = {} & (N_2 M_2 M_1 N_1/P)\, u_1(N,P) + (N_2 + M_2 - 1)((M_1 - 1)\, u_{\text{sm}2}(N,P) \\ & + (M_1 - N_1/P - 1)\, u_{\text{sm}3}(N,P) + (M_1 - 3N_1/P - 1)\, u_{\text{sm}4}(N,P)) \end{aligned} \tag{3.22}$$

where all the u parameters represent the times for one addition and one communication (except for u_1 which is only the time for one multiplication and one addition) in corresponding methods and stages, including relevant delay and time to load and store data. They are derived from experiment and are listed in Tables 3.8 to 3.11.

Table 3.5 The estimated running time of the hybrid version of the binary-tree algorithm.

N_1=N_2=8	M_1=M_2=8	M_1=M_2=16	M_1=M_2=24	M_1=M_2=28	M_1=M_2=32
P=1	0.008228	0.031727	0.070469	0.095581	0.124509
P=2	0.004277	0.016290	0.036129	0.048954	0.063698
P=4	0.002381	0.008957	0.019737	0.026673	0.034641
P=8	0.001545	0.005595	0.012062	0.016266	0.021081

Table 3.6 The estimated running time of the message-passing version of the binary-tree algorithm.

$N_1=N_2=8$	$M_1=M_2=8$	$M_1=M_2=16$	$M_1=M_2=24$	$M_1=M_2=28$	$M_1=M_2=32$
P=1	0.008228	0.031727	0.070469	0.095581	0.124509
P=2	0.004273	0.016281	0.036136	0.048881	0.063701
P=4	0.002425	0.008979	0.019720	0.026669	0.034672
P=8	0.001579	0.005644	0.012138	0.016368	0.021174

Table 3.7 The estimated running time of the shared-memory version of the binary-tree algorithm.

$N_1=N_2=8$	$M_1=M_2=8$	$M_1=M_2=16$	$M_1=M_2=24$	$M_1=M_2=28$	$M_1=M_2=32$
P=1	0.008228	0.031727	0.070469	0.095581	0.124509
P=2	0.004418	0.016678	0.036901	0.049883	0.065020
P=4	0.002773	0.010128	0.022053	0.029819	0.038701
P=8	0.002011	0.007060	0.015154	0.020352	0.026261

Table 3.8 The time u_1 (in nsecs) for the binary-tree algorithm.

u_1	$M_1=M_2=8$	$M_1=M_2=16$	$M_1=M_2=24$	$M_1=M_2=28$	$M_1=M_2=32$
P=1	2008.8	1936.5	1911.6	1904.9	1899.9
P=2	1999.1	1922.5	1897.8	1890.9	1885.7
P=4	2005.4	1923.8	1898.6	1891.4	1886.1
P=8	2017.2	1928.0	1900.7	1892.1	1887.1

Table 3.9 The time u_{xx2} (in nsecs) for the binary-tree algorithm.

u_{xx2}		$M_1=M_2=8$	$M_1=M_2=16$	$M_1=M_2=24$	$M_1=M_2=28$	$M_1=M_2=32$
hy		1741.9	1567.8	1610.5	1604.1	1570.0
mp		1707.6	1542.9	1620.9	1526.2	1580.2
sm	**P=2**	3078.1	2694.2	2691.3	2586.5	2671.6
	P=4	3960.0	3719.4	3642.2	3661.5	3669.0
	P=8	3976.2	3785.2	3771.4	3759.7	3719.6

Table 3.10 The time u_{xx3} (in nsecs) for the binary-tree algorithm.

u_{xx3}		$M_1=M_2=8$	$M_1=M_2=16$	$M_1=M_2=24$	$M_1=M_2=28$	$M_1=M_2=32$
hy	**P=4**	1928.0	1793.3	1676.2	1636.5	1627.5
	P=8	1807.8	1725.2	1586.5	1589.2	1597.2
mp	**P=4**	2149.3	1531.3	1411.5	1397.1	1402.8
	P=8	1718.9	1393.2	1330.0	1329.5	1345.0
sm	**P=4**	4054.7	3219.7	3068.4	3010.5	2972.6
	P=8	3717.8	3225.8	3067.6	3025.6	2987.7

Table 3.11 The time u_{xx4} (in nsecs) for the binary-tree algorithm.

u_{xx4}	$M_1=M_2=8$	$M_1=M_2=16$	$M_1=M_2=24$	$M_1=M_2=28$	$M_1=M_2=32$
hy	2776.7	1992.8	1731.3	1710.1	1698.5
mp	3171.7	2129.3	1806.3	1759.5	1746.6
sm	3856.7	2763.0	2629.5	2590.5	2563.2

3.4 Performance Results on TurboNet

The algorithms for 1-D and 2-D convolution were implemented on TurboNet. Relevant performance results are presented in this section. In all implementations each processor employs its DMA controllers to fetch the values of *f* and *g* from the shared memory, and starts the calculation after both data arrays are stored in the local memory. Due to their lack of communication, the 1-D convolution algorithms cannot demonstrate the use of message-passing and/or shared-memory, and therefore will not be evaluated in significant detail. They are, however, efficient algorithms that serve as introductory for the algorithm of 2-D convolution.

3.4.1 1-D Convolution

Tables 3.12 to 3.17 contain execution times, speed-ups and efficiencies for both methods of 1-D convolution, with variant data sizes and numbers of processors. The execution time listed here for each case is the longest one among the processors. It can be seen from the results that method 1 has better performance than method 2. This agrees with our analysis in Section 3.1. In Figure 3.8, the execution times for both methods are plotted for 8 processors. It can also been seen that the estimated running times in Section 3.3.1 are very close to the experimental ones. In Figure 3.9, such a comparison for method 1 is plotted. The difference between the running times of methods 1 and 2 are also shown in Figures 3.10 and 3.11.

Table 3.12 Execution time for 1-D convolution with method 1.

M=512	N=8	N=32	N=128	N=384
P=1	0.005129	0.020513	0.082049	0.246145
P=2	0.003682	0.013148	0.053314	0.174866
P=4	0.001858	0.006716	0.028450	0.090712
P=8	0.000946	0.003500	0.015048	0.046178

Table 3.13 Speedup of 1-D convolution with method 1.

M=512	N=8	N=32	N=128	N=384
P=2	1.39	1.56	1.54	1.41
P=4	2.76	3.05	2.88	2.71
P=8	5.42	5.86	5.45	5.33

Table 3.14 Efficiency of 1-D convolution with method 1.

M=512	N=8	N=32	N=128	N=384
P=2	0.70	0.78	0.77	0.70
P=4	0.69	0.76	0.72	0.68
P=8	0.68	0.73	0.68	0.67

Table 3.15 Execution time for 1-D convolution with method 2.

M=513	N=8	N=32	N=128	N=384
P=1	0.005139	0.020553	0.082209	0.246625
P=2	0.004482	0.014087	0.052504	0.154948
P=4	0.002250	0.007413	0.032529	0.114757
P=8	0.001129	0.003711	0.016269	0.065491

Table 3.16 Speedup of 1-D convolution with method 2.

M=513	N=8	N=32	N=128	N=384
P=2	1.15	1.46	1.57	1.59
P=4	2.29	2.77	2.52	2.15
P=8	4.55	5.54	5.05	3.77

Table 3.17 Efficiency of 1-D convolution with method 2.

M=513	N=8	N=32	N=128	N=384
P=2	0.57	0.73	0.78	0.80
P=4	0.57	0.69	0.63	0.54
P=8	0.57	0.69	0.63	0.47

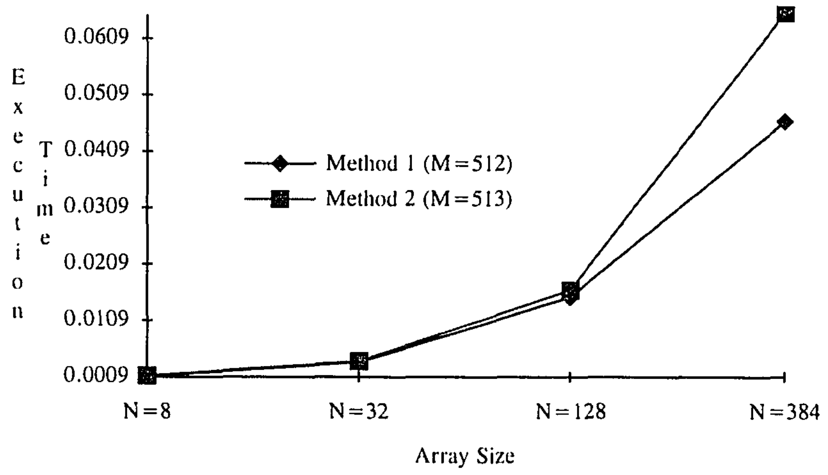


Figure 3.8 Execution time of 1-D convolution (method 1 vs method 2), with P = 8.

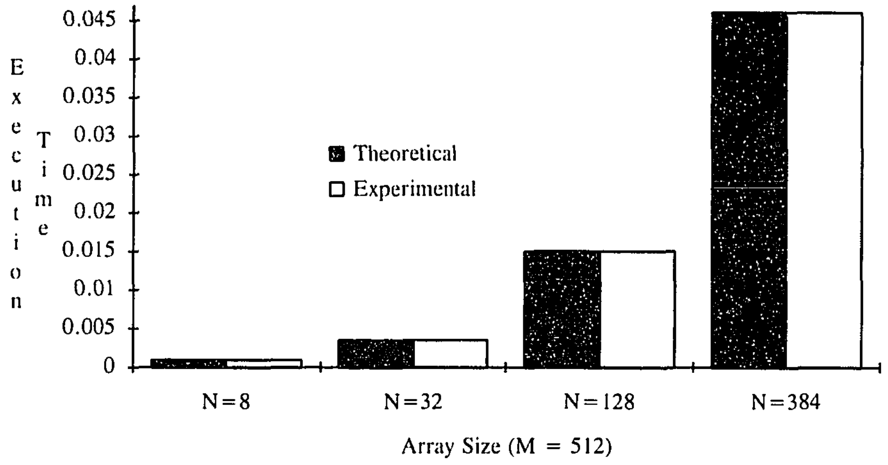


Figure 3.9 A comparison of theoretical and experimental running times for method 1.

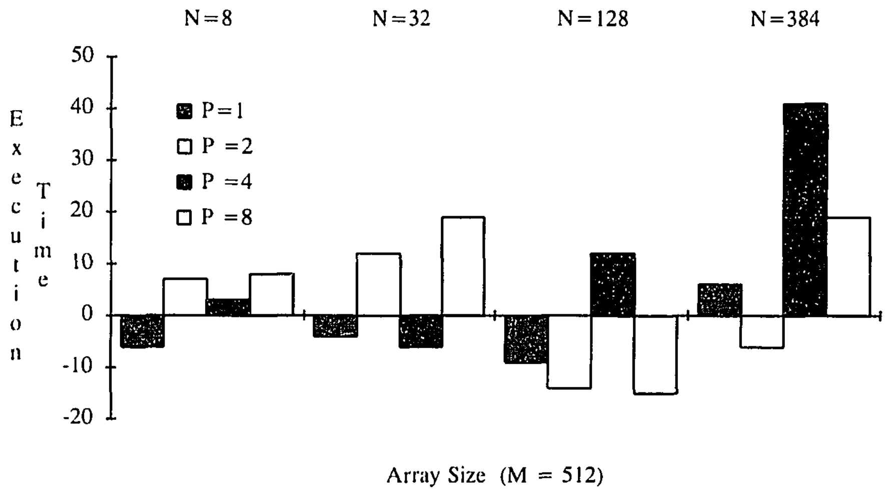


Figure 3.10 The difference between theoretical and experimental running times for method 1.

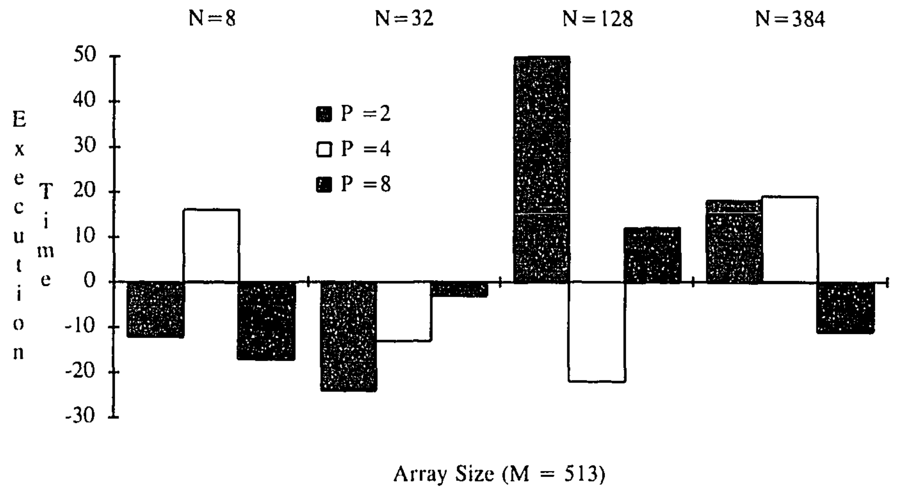


Figure 3.11 The difference of theoretical and experimental running times from method 2.

3.4.2 2-D Convolution

Tables 3.18 to 3.26 contain execution times, speed-ups and efficiencies for the three versions of the binary-tree algorithm, with variant data sizes and numbers of processors (N_1 and N_2 are always equal to 8). From the results we see that the performance of the hybrid version is slightly better than that of the message-passing one, while the shared-memory one trails both of them with a noticeable margin. This matches our prediction in Section 3.3.2. From Tables 3.18 and 3.24, it is seen that the hybrid and message-passing versions are much faster than the shared-memory version. In order to show clearly that the hybrid version is also faster than the message-passing version, in Figure 3.12 we plot the difference in execution times between the message-passing and hybrid versions, in this order, for 8 processors. Again the theoretical and experimental running times of the binary-tree method are close to each other. A direct comparison of both running times for the hybrid version with $P = 8$ is shown in Figure 3.13, and the differences of these running times for the three versions are plotted in Figures 3.14 to 3.16.

Table 3.18 Execution time for the hybrid version of the binary-tree algorithm.

$N_1=N_2=8$	$M_1=M_2=8$	$M_1=M_2=16$	$M_1=M_2=24$	$M_1=M_2=28$	$M_1=M_2=32$
P=1	0.008237	0.031713	0.070460	0.095560	0.124481
P=2	0.004265	0.016292	0.036094	0.048911	0.063672
P=4	0.002388	0.008948	0.019696	0.026638	0.034628
P=8	0.001553	0.005565	0.012071	0.016254	0.021066

Table 3.19 Speedup of the hybrid version of the binary-tree algorithm.

$N_1=N_2=8$	$M_1=M_2=8$	$M_1=M_2=16$	$M_1=M_2=24$	$M_1=M_2=28$	$M_1=M_2=32$
P=2	1.93	1.95	1.95	1.95	1.96
P=4	3.45	3.54	3.58	3.59	3.59
P=8	5.30	5.70	5.84	5.88	5.91

Table 3.20 Efficiency of the hybrid version of the binary-tree algorithm.

$N_1=N_2=8$	$M_1=M_2=8$	$M_1=M_2=16$	$M_1=M_2=24$	$M_1=M_2=28$	$M_1=M_2=32$
P=2	0.97	0.97	0.98	0.98	0.98
P=4	0.86	0.89	0.89	0.90	0.90
P=8	0.66	0.71	0.73	0.73	0.74

Table 3.21 Execution time for the message-passing version of the binary-tree algorithm.

$N_1=N_2=8$	$M_1=M_2=8$	$M_1=M_2=16$	$M_1=M_2=24$	$M_1=M_2=28$	$M_1=M_2=32$
P=1	0.008237	0.031713	0.070460	0.095560	0.124481
P=2	0.004267	0.016294	0.036096	0.048914	0.063675
P=4	0.002417	0.008989	0.019746	0.026695	0.034690
P=8	0.001595	0.005628	0.012149	0.016343	0.021163

Table 3.22 Speedup of the message-passing version of the binary-tree algorithm.

$N_1=N_2=8$	$M_1=M_2=8$	$M_1=M_2=16$	$M_1=M_2=24$	$M_1=M_2=28$	$M_1=M_2=32$
P=2	1.93	1.95	1.95	1.95	1.95
P=4	3.41	3.53	3.57	3.58	3.59
P=8	5.16	5.63	5.80	5.85	5.88

Table 3.23 Efficiency of the message-passing version of the binary-tree algorithm.

$N_1=N_2=8$	$M_1=M_2=8$	$M_1=M_2=16$	$M_1=M_2=24$	$M_1=M_2=28$	$M_1=M_2=32$
P=2	0.96	0.97	0.98	0.98	0.98
P=4	0.85	0.88	0.89	0.89	0.90
P=8	0.65	0.70	0.72	0.73	0.74

Table 3.24 Execution time for the shared-memory version of the binary-tree algorithm.

$N_1=N_2=8$	$M_1=M_2=8$	$M_1=M_2=16$	$M_1=M_2=24$	$M_1=M_2=28$	$M_1=M_2=32$
P=1	0.008237	0.031713	0.070460	0.095560	0.124481
P=2	0.004409	0.016666	0.036872	0.049907	0.064997
P=4	0.002763	0.010133	0.022097	0.029843	0.038714
P=8	0.002017	0.007077	0.015137	0.020329	0.026273

Table 3.25 Speedup of the shared-memory version of the binary-tree algorithm.

$N_1=N_2=8$	$M_1=M_2=8$	$M_1=M_2=16$	$M_1=M_2=24$	$M_1=M_2=28$	$M_1=M_2=32$
P=2	1.87	1.90	1.91	1.91	1.92
P=4	2.98	3.13	3.19	3.20	3.22
P=8	4.08	4.48	4.65	4.70	4.74

Table 3.26 Efficiency of the shared-memory version of the binary-tree algorithm.

$N_1=N_2=8$	$M_1=M_2=8$	$M_1=M_2=16$	$M_1=M_2=24$	$M_1=M_2=28$	$M_1=M_2=32$
P=2	0.94	0.95	0.96	0.96	0.96
P=4	0.75	0.78	0.80	0.80	0.80
P=8	0.51	0.56	0.58	0.59	0.59

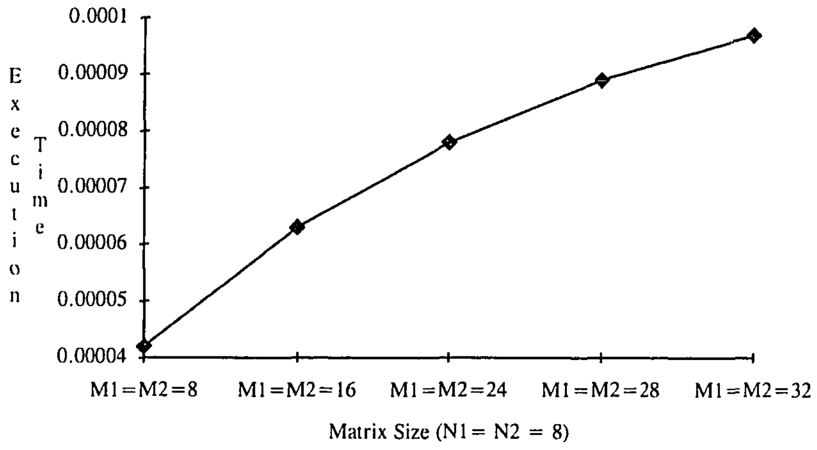


Figure 3.12 The difference in execution times between the message-passing and hybrid versions, in this order, for $P = 8$.

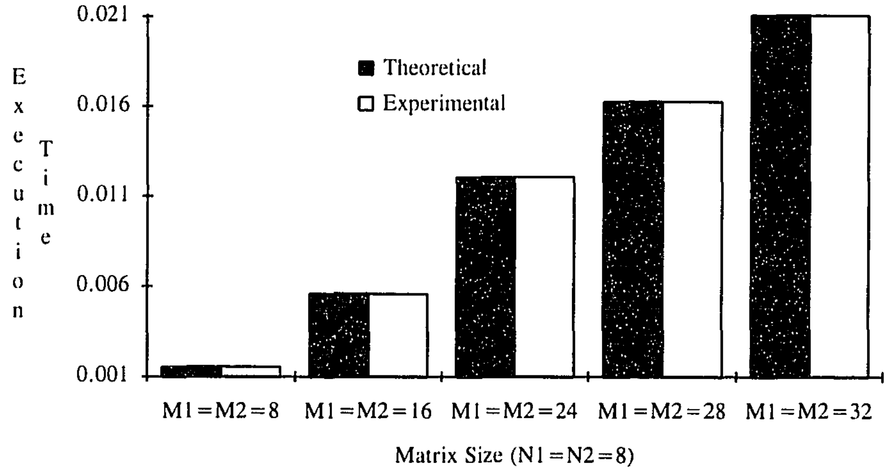


Figure 3.13 A comparison of theoretical and experimental running times of the hybrid version for $P = 8$.

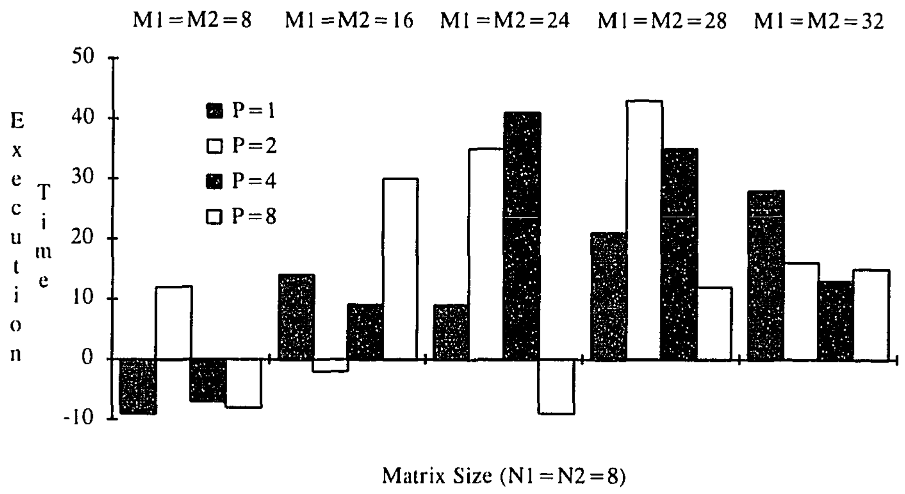


Figure 3.14 The difference of theoretical and experimental running times for the hybrid version.

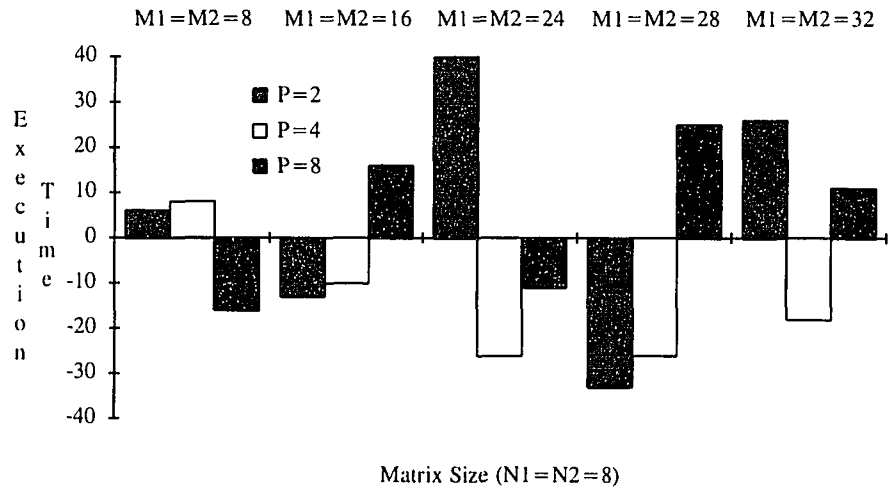


Figure 3.15 The difference of theoretical and experimental running times for the message-passing version.

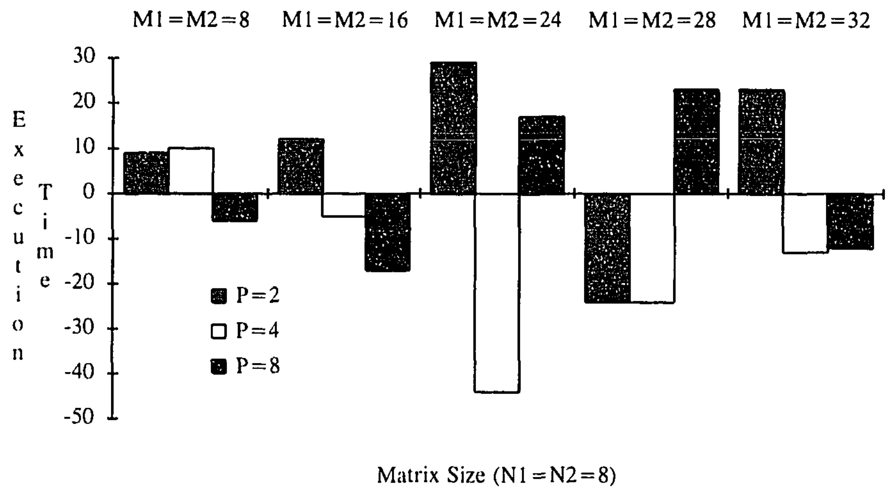


Figure 3.16 The difference of theoretical and experimental running times for the shared-memory version.

In Figure 3.7, CT_2, CT_3 and CT_4 represent the communication times of the corresponding edges in stages 2, 3 and 4. Their values were measured in the implementations and are displayed in Tables 3.27 to 3.29. In the tables, hy, mp and sm are abbreviations for hybrid, message-passing and shared-memory. From Equations 3.11 to 3.13, it is seen that the number of operations in stage 2 is not a function of P. P affects the communication times only in the shared-memory implementation; that is why in Table 3.27 only the sm case is categorized for different values of P. Also, notice that Table 3.29 is only for $P = 8$.

Table 3.27 The communication time (in μsec) between P_i and P_{i+1} in stage 2.

CT_2		$M_1=M_2=8$	$M_1=M_2=16$	$M_1=M_2=24$	$M_1=M_2=28$	$M_1=M_2=32$
hy		133.30	424.50	880.38	1182.12	1545.88
mp		133.40	424.55	880.41	1182.35	1545.98
sm	P=2	274.30	796.75	1654.61	2175.25	2868.48
	P=4	370.90	1164.95	2383.61	3179.15	4103.78
	P=8	379.60	1190.15	2448.71	3252.95	4162.48

Table 3.28 The communication time (in μsec) between P_i and P_{i-2} in stage 3.

CT_3		$M_1=M_2=8$	$M_1=M_2=16$	$M_1=M_2=24$	$M_1=M_2=28$	$M_1=M_2=32$
hy	P=4	132.90	409.55	867.37	1148.65	1517.02
	P=8	138.50	414.80	877.04	1170.96	1539.90
mp	P=4	161.20	457.85	918.87	1222.45	1586.62
	P=8	154.70	448.60	907.04	1209.80	1573.70
sm	P=4	270.60	860.55	1804.67	2374.95	3051.82
	P=8	302.40	931.00	1862.04	2464.20	3172.00

Table 3.29 The communication time (in μsec) between P_i and P_{i-4} in stage 4.

CT_4	$M_1=M_2=8$	$M_1=M_2=16$	$M_1=M_2=24$	$M_1=M_2=28$	$M_1=M_2=32$
hy	153.80	410.40	868.80	1161.60	1521.05
mp	178.50	476.60	940.80	1246.60	1609.04
sm	210.60	673.00	1418.70	1906.60	2497.24

CHAPTER 4

FAST FOURIER TRANSFORM

The implementation of Fast Fourier Transform (FFT) algorithms on parallel systems is found in many applications [1, 2, 4, 6, 7, 23]. This chapter deals with the 1-D and 2-D FFT algorithms and their implementation on TurboNet. Here the number of processors and the sizes of any data arrays or matrices are assumed to be powers of 2.

A 2-D FFT is usually performed by 1-D row FFT's followed by 1-D column FFT's, that is, as long as we know how to do the 1-D FFT, the 2-D FFT is also solved. The problem with the 2-D FFT here, however, is how to overlap the input, calculation and output processes, so that the performance can be improved, especially when the input data to a processor is bigger than its internal memory and then such overlaps become vital. Our 2-D FFT algorithm will also focus on how to perform efficiently the transposition of the intermediate matrix between the row and column FFT's.

4.1 One-Dimensional FFT

The Discrete Fourier Transform (DFT) of a finite-length digital sequence x is a sequence y of the same length as x such that

$$y[k] = \sum_{n=0}^{N-1} x[n] w_N^{kn} \tag{4.1}$$

for $k = 0, 1, ..., N\text{-}1$, where N is the length of the digital sequence and w_N is a primitive N^{th} root of unity, that is $w = e^{i2\pi/N}$, where $i = \sqrt{-1}$. Its computation requires on the order of N^2 multiplications and N^2 additions.

The FFT is the most popular algorithm in the implementation of the DFT. For a parallel implementation of the FFT, let us look at the data flow graph of an 8-point decimation-in-time (DIT) FFT. Its butterfly configuration is shown in Figure 4.1, where the input array is shuffled while the output is in increasing-index order. In the figure, as an example, four processors are used to perform the 8-point FFT. Each one of them calculates two results as shown. Any arrow that crosses a dashed line separating two processors indicates a communication between the two end-processors of the arrow. In stage 1, each processor performs a butterfly computation without any communication involved. In stage 2, each processor carries out the low or upper halves of two butterfly computations and communicates with one of its two neighbors the results of both computations. Stage 3 is just like stage 2 except that this time it involves another neighbor. The W_N^x 's are called *butterfly-twiddle factors* (these coefficients are complex numbers). There is a trade-off between speed and memory in dealing with the twiddle factors. In order to speed up the FFT operation, they are usually generated and stored in a table before stage 1 starts. To save memory, however, they can be calculated each time when they are needed. For an N-point FFT, $N/2$ coefficients are needed. The butterfly network can also be mapped onto a hypercube with the same number of processors (i.e. $P = 8$ in this example) in an optimal manner. Therefore, assuming a hypercube system with

8 processors, each processor will have to exchange data once with each one of its $\log_2 8 = 3$ neighbors.

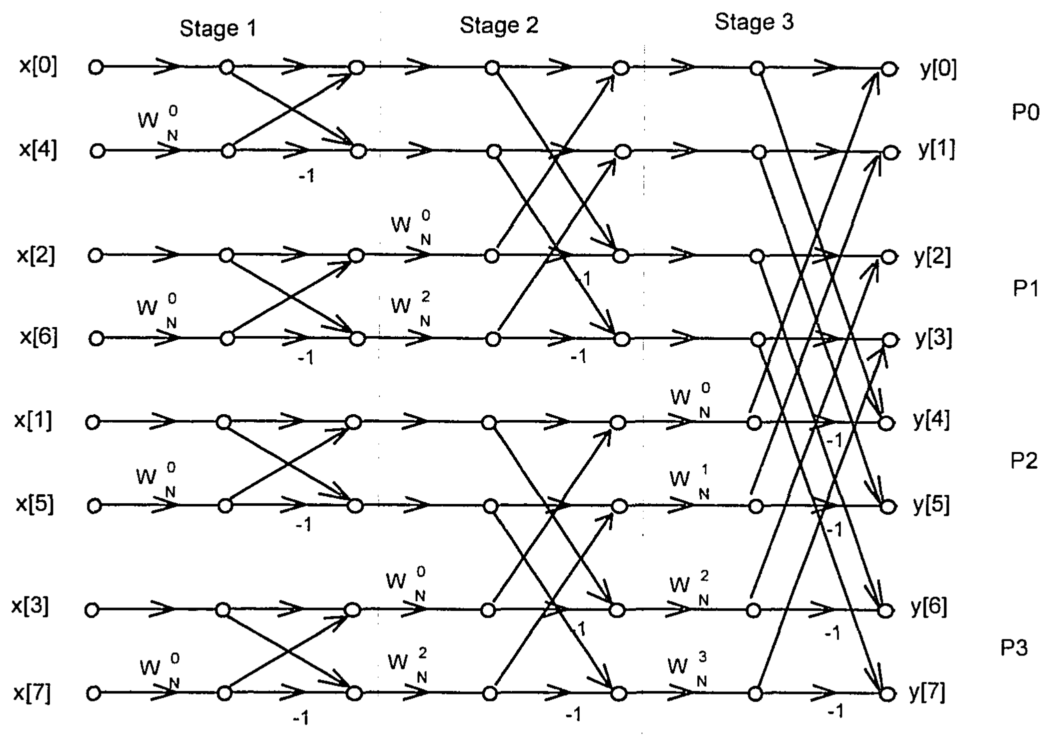


Figure 4.1 Data flow graph of an 8-point decimation-in-time DFT.

In general, if there are P processors in performing an N-point, radix-2, decimation-in-time FFT, then each one of them calculates N/P results by performing $N/(2P)\log_2 N$ butterfly computations, so that P_0 produces $y[0]$, $y[1]$, ..., $y[N/P-1]$, P_1 produces $y[N/P]$, $y[N/P+1]$, ..., $y[2N/P-1]$, etc. Note that after the first $\log_2(N/P)$ butterfly computations, which is a sequential N/P-point 1-D FFT, the remaining butterfly computations are split in half between neighboring processors. This means that when calculating half of a butterfly computation, a processor has to exchange data with its neighbor performing the other half of the butterfly computation.

The parallel 1-D FFT algorithm has a strong communication bias toward hypercube systems, since all communications in the algorithm are between neighboring processors, and this results in good performance for the message-passing method. The FFT algorithm is described briefly below.

ONE-DIMENSIONAL FFT ALGORITHM:

```
(1)    for all processors do in parallel
           set up a DMA controller to read $N/P$ input data from the DRAM with bit-
           reverse addressing (global memory);
(2)    for $s = 1$ to $\log_2(N/P)$ do
           for $i = 0$ to $2^s/2$ do
               read the coefficient from the DRAM;
               perform the basic butterfly computation;
           end for
       end for;
(3)    for $s = \log_2(N/P) + 1$ to $\log_2 N$ do
           for $i = 0$ to $N/P-1$ do
               read the coefficient from the DRAM;
               exchange data with a neighbor;
```

```
                perform the basic butterfly computation;
            end for
        end for.
```

4.2 Two-Dimensional FFT

The definition of the two-dimensional DFT is given by

$$y[k_1,k_2]=\sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1}x[n_1,n_2]w_{N_1}^{k_1n_1}w_{N_2}^{k_2n_2} \tag{4.2}$$

for $k_1 = 0, 1, ..., N_1-1$ and $k_2 = 0, 1, ..., N_2-1$, where $w_{N_1}=e^{i2\pi/N_1}$, $w_{N_2}=e^{i2\pi/N_2}$, and N_1 and N_2 are the lengths of the 2-D matrix x.

To perform a 2-D FFT, we first perform 1-D FFT's row-by-row on the input matrix, and then on the resulting matrix we perform 1-D FFT's column-by-column; this is called the *row-column* method [26]. The 2-D FFT is, thus, decomposed into N_1+N_2 1-D FFT's. In order to reduce the communication overhead and simplify the programming task, we let each 1-D FFT be performed by a single processor rather than dividing it between two or more processors; we assume that the number of processors is less than the number of rows or columns, which is usually the case in many applications. This means that each processor will perform 1-D FFT's in a sequential way, and communication is needed only during the transposition of the resulting matrix of the row FFT's.

As mentioned at the beginning of this chapter, due to the limited amount of processor internal memory, all required input data for a processor may not be fully stored in it initially. Instead the processor may have to read few rows or columns at a time, and after completion of the current task it will have to start the next read cycle. In this situation, the efficiency of reads and writes is vital to the performance of the algorithm. Hence, our research focus here is on how efficiently input data are read from and output data are written back to the DRAM, for both row and column FFT's.

Suppose we have P processors to perform a 2-D FFT on a NxN matrix, where N/P is an integer. Assume row FFT's for simplicity. Also, assume that each processor can store internally $2k$ rows of data, where $2k < (N/P)$, and that $(N/P)/k$ is an integer. In order to save CPU time, we let a DMA controller of each processor read from the DRAM k rows of data assigned to it, while the processor operates on k preceding rows. Each time the DMA controller reads another k rows, Pk rows down from the first row of the last k rows read. While the processor works on these newly read k rows, another DMA controller writes back to the DRAM the result for the last k rows processed. Of course, the processor's internal memory should accommodate $4k$ rows if the memory space occupied by the $2k$ rows of the result is also considered. Each processor will repeat this process $(N/P)/k$ times. Such a memory interleaving example for input rows, with $P = 4$, is shown in Figure 4.2. For the column FFT, the procedure is the same except that the input is the columns of the resulting matrix of the row FFT's. This means we have to use different address indexing in the DMA controllers when reading or writing the DRAM, or perform transposition of the matrix directly.

Under the condition of $2k < (N/P)$, we have assumed the use of shared-memory for communication, and it is obviously a practical choice over any message-passing method.

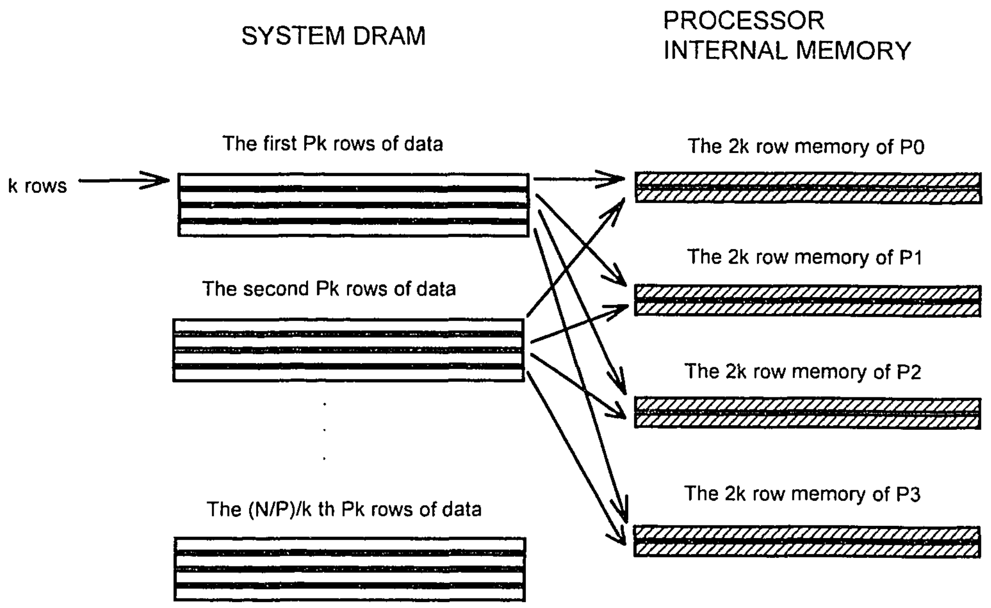


Figure 4.2 An example of accessing row data for the FFT, with $P = 4$ and $2k < (N/P)$.

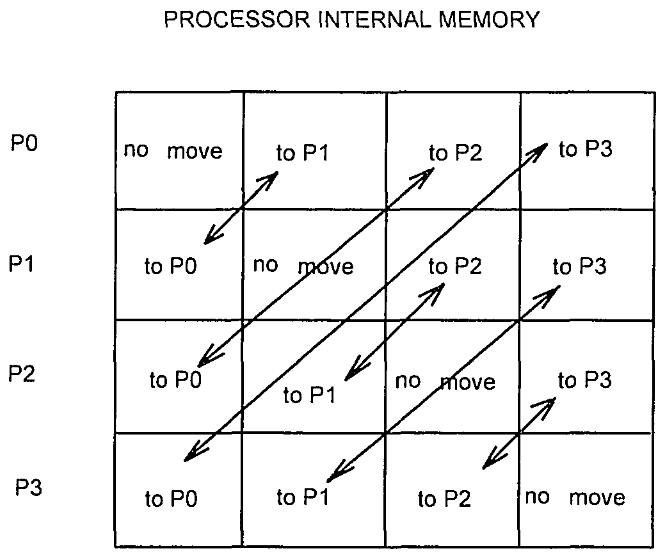


Figure 4.3 An example of transferring data for the FFT, with $P = 4$ and $2k \geq (N/P)$.

Now consider the case of $2k \geq (N/P)$. Each processor now can fully contain all N/P rows of its data in its internal memory. In this case the only problem is how to carry out the transposition of the intermediate matrix. After finishing the row FFT's, each processor divides the resulting N/P rows into P sections of columns, and then sends each one of them to that processor whose identification number (ID) is the same as the section number. Transposition of columns into rows is also involved. Each processor keeps the section with the same section number as its own ID. Figure 4.3 gives an example for this case. In this case, each processor has to talk to every other processor, which results in complex programming structures. Hence, this method was not implemented on TurboNet and can be considered in further research.

4.3 Theoretical Analysis

Theoretical analysis of the 1-D and 2-D FFT algorithms is presented in this section.

4.3.1 1-D FFT

From Figure 4.1 we can see that there are $\log_2 N$ stages in the N-point FFT. In each stage there are $N/2$ butterfly computations, and each one includes two multiplications and two additions of complex numbers. Hence, its running time on a single processor is

$$t_{\mathrm{sql}}(N) = 6N(\log_2 N)\, t_{\mathrm{cal}} = O(N \log_2 N) \qquad (4.3)$$

where t_{cal} is the time for one multiplication or addition.

The DDG for $P = 8$ is depicted in Figure 4.4. Parts of the figure can also be used for 2 and 4 processors. If $N/P \geq 2$, each arrow connecting two processors represents the communication of N/P values. As indicated in Section 4.1, the parallel implementation of the N-point FFT by P processors has two parts. The first part of the first $\log_2(N/P)$ stages is an N/P-point sequential FFT with running time

$$c_1(N) = 6(N/P)\log_2(N/P)t_{\text{cal}} = O(N/P \log_2(N/P)) \qquad (4.4)$$

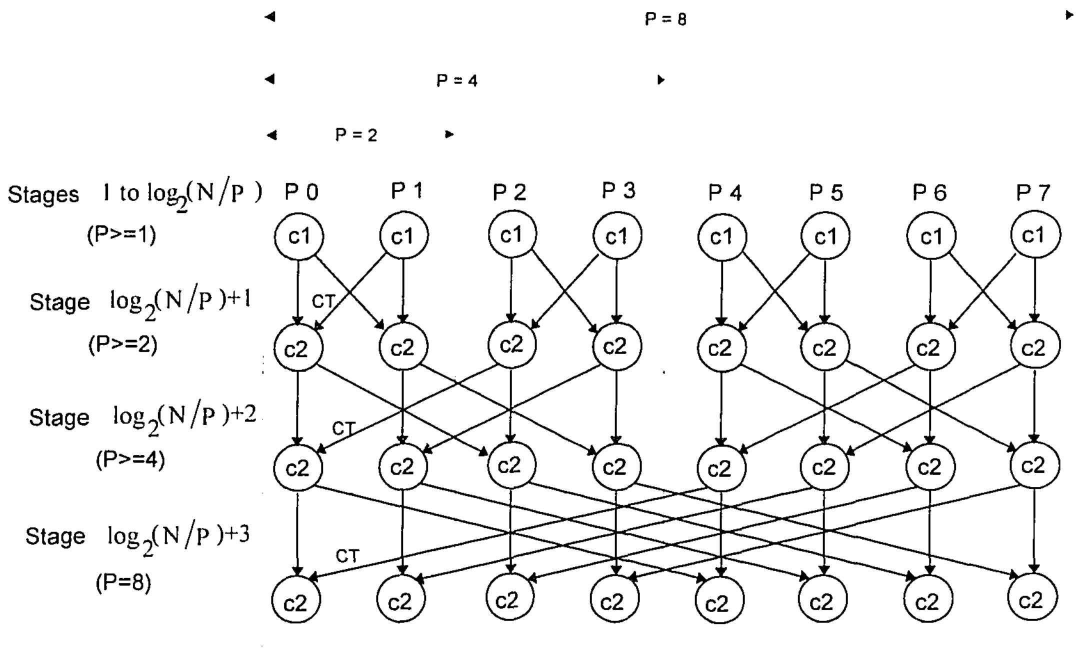


Figure 4.4 DDG of the 1-D FFT with $P = 8$.

The second part is the last $\log_2 P$ stages (they are the last three layers for $P = 8$ in Figure 4.4). In this part each butterfly computation in every stage is divided into two halves carried out by two neighboring processors. All stages in this part should have the same running time since the numbers of calculations are equal and the communication time between any two neighbors is assumed to be the same. Each processor will have *N*/*P* values of the butterfly computations in each stage, and therefore the running time for each stage of the second part is

$$c_2(N) = N/P\,(6t_{\text{cal}} + 2t_{\text{com}}) = O(N/P) \tag{4.5}$$

where t_{com} is the time to send one word (a complex number is considered to have two words) from a processor to its neighbor via the direct link that connects them, and the running time of the parallel algorithm for the message-passing system adds up to

$$t_{\text{mp}}(N) = c_1(N) + (\log_2 P)c_2(N) = O(N/P \log_2 N) \tag{4.6}$$

As mentioned in Section 4.1, the nature of the parallel FFT makes good use of the hypercube structure in terms of efficient communication, since all data exchanges are between neighbors. If the communication channels between neighboring processors are full-duplex, there will be no additional delay when they exchange data at the same time. For a half-duplex channel, which is the case for the TurboNet system, if two neighboring processors want to send data to each other at the same time, one has to wait until the other one finishes. However, we can safely assume that the communication between two neighbors in the TurboNet system never happens at the same time, because one of them has to multiply the outgoing data with the twiddle-factors while the other does not. It is obvious that the integration of the shared-memory paradigm with this message-passing paradigm for the 1-D FFT would result in performance degradation. For the shared-memory only method, there will be a total bus delay as much as $(P-1)\,N/P(\log_2 P)t_{\text{del}}$, where $N/P \log_2 P$ is the total number of complex numbers each processor has to send, and $t_{\mathrm{del}} = t_{\mathrm{DRAM}}$ is the time to transfer one complex number from/to the DRAM. Hence, the estimated running time for the shared-memory method is

$$t_{\mathrm{sm}}(N) = 6((N/P\log_2 P + N/P\log_2(N/P))t_{\mathrm{cal}} + 2N(\log_2 P)t_{\mathrm{DRAM}})$$

$$= N/P(\log_2 N)t_{\mathrm{cal}} + N(\log_2 P)t_{\mathrm{DRAM}} = O(N/P\log_2 N + N\log_2 P) \tag{4.7}$$

From Equations 4.6 and 4.7, it is obvious that the message-passing method will have better performance than the shared-memory one. Tables 4.1 and 4.2 give the estimated running times of the 1-D FFT for the message-passing and shared-memory methods, respectively. They are estimated from the following two equations:

$$t_{\mathrm{E-mp}}(N) = N/P(\log_2(N/P))u(N,P) + N/P(\log_2 P)u_{\mathrm{mp}}(N,P) \tag{4.8}$$

$$t_{\mathrm{E-sm}}(N) = N/P(\log_2(N/P))u(N,P) + N/P(\log_2 P)u_{\mathrm{sm}}(N,P) \tag{4.9}$$

where $u(N,P)$, $u_{\mathrm{mp}}(N,P)$ and $u_{\mathrm{sm}}(N,P)$, obtained from experiment, represent the amount of time for half a butterfly computation (i.e. one multiplication and one addition of complex numbers) in the sequential, parallel with message-passing and parallel with shared-memory implementations, respectively. The u parameters also include the time to load and store input and output data. Appropriate values are derived from experiment and are listed in Tables 4.3, 4.4 and 4.5.

Table 4.1 The estimated running time for the 1-D FFT with message-passing.

	N=64	**N=128**	**N=256**	**N=512**	**N=1024**
P=1	0.001163	0.002610	0.005828	0.012921	0.028499
P=2	0.000783	0.001665	0.003563	0.007708	0.016668
P=4	0.000523	0.001050	0.002159	0.004526	0.009605
P=8	0.000372	0.000688	0.001318	0.002678	0.005468

Table 4.2 The estimated running time for the 1-D FFT with shared-memory.

	N=64	**N=128**	**N=256**	**N=512**	**N=1024**
P=1	0.001163	0.002610	0.005828	0.012921	0.028499
P=2	0.000923	0.001928	0.004079	0.008786	0.018734
P=4	0.000820	0.001616	0.003275	0.006523	0.013444
P=8	0.000886	0.001667	0.003250	0.006515	0.013198

Table 4.3 The sequential time u (in nsecs) of 1-D FFT.

u	**N=64**	**N=128**	**N=256**	**N=512**	**N=1024**
P=1	3028.6	2912.9	2845.7	2804.0	2783.1
P=2	3300.0	3044.3	2916.3	2841.8	2805.8
P=4	3765.6	3300.0	3046.9	2917.4	2842.3
P=8	4666.7	3781.3	3306.3	3046.9	2917.7

Table 4.4 The time u_{mp} (in nsecs) of 1-D FFT with message-passing.

u_{mp}	**N=64**	**N=128**	**N=256**	**N=512**	**N=1024**
P=2	7968.8	7750.0	7421.9	7375.0	7302.7
P=4	8812.5	8156.3	7726.6	7468.8	7390.6
P=8	10833.3	9291.7	8218.8	7854.2	7432.3

Table 4.5 The time u_{sm} (in nsecs) of 1-D FFT with shared-memory.

u_{sm}	N=64	N=128	N=256	N=512	N=1024
P=2	12343.8	11859.4	11453.1	11586.0	11337.9
P=4	18093.8	17000.0	16445.3	15269.5	14888.7
P=8	32250.0	29687.5	28343.8	27838.5	27562.5

4.3.2 2-D FFT

A sequential 2-D FFT on an NxN matrix is actually $2N$ N-points 1-D FFT's, and therefore its running time is

$$t_{sq2}(N) = 2N(6N(\log_2 N)t_{cal}) = O(N^2 \log_2 N) \qquad (4.10)$$

Here the matrix transposition is assumed to be done through reading input columns. The proposed 2-D FFT algorithm for $2k < (N/P)$, described in Section 4.2, has a DDG as shown in Figure 4.5. In order to speed up the operations, the algorithm overlaps its data input, FFT calculation and data output phases for the row and column FFT's. This is indicated in the figure by dashed circles. Inside each circle, the three phases are represented by smaller circles marked with "r", "ffts" and "w" for read, 1-D FFT's and write, respectively. Overlapping of the three phases iterates until the row (or column) FFT's are completed. Since the time for reading and writing k rows (or columns) of data from and to the shared memory is less than the time for their 1-D FFT's, after the first k rows (or columns) of data are fetched each processor can complete without pause all of its N/P row (or column) FFT's. For example, from experiment, with $N = 64$, $P = 8$ and $k = 2$, the two 64-point 1-D FFT's take 2338 μsecs, while the reading and writing of data from the shared memory can take up to $(2P\text{-}1)kN$ 660(nsecs) = 1267.2 μsecs, where the 660(nsecs) is the value of t_{DRAM}. Therefore, the running time of the row (or column) FFT stage is

$$t_{\mathrm{fft}}(N) = (N/P)\,N(\log_2 N)6\,t_{\mathrm{cal}} = O(N^2/P\log_2 N) \tag{4.11}$$

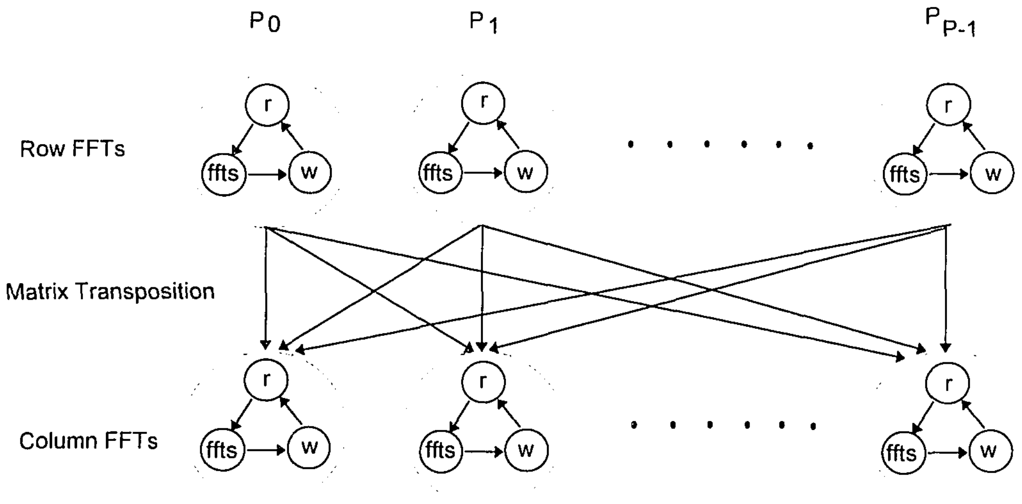


Figure 4.5 DDG of the 2-D FFT algorithm for $2k < (N/P)$.

As pointed out earlier, the transposition can be performed during the input of columns by either using another addressing scheme for the DMA controllers, different from the one used for row FFT's, or directly transposing the matrix in the DRAM. The latter is chosen in our algorithm for the following two reasons: (1) re-utilizing the program of row FFT's for column FFT's without any change; (2) avoiding the initialization of the DMA controllers for the reading of each column with different starting addresses, since it is rather time consuming. In the transposition stage, each processor has to read and write

$N\,N/P$ complex numbers with a possible maximum delay in the DRAM of $(P-1)\,N^2/P\,2\,t_{\mathrm{DRAM}}$, that is, the running time of this stage is

$$t_{\mathrm{mt}}(N) = (N^2/P + (P-1)\,N^2/P)2\,t_{\mathrm{DRAM}} = 2N^2\,t_{\mathrm{DRAM}} = O(N^2) \tag{4.12}$$

The total running time of the algorithm is then the summation of $2t_{\mathrm{fft}}(N)$ and $t_{\mathrm{mt}}(N)$, as

$$\begin{aligned} t_{\mathrm{fft}2}(N) &= N^2/P(\log_2 N)12\,t_{\mathrm{cal}} + N^2\,2\,t_{\mathrm{DRAM}} = O(N^2/P(\log_2 N + P)) \\ &= O(N^2/P\log_2 N) \end{aligned} \tag{4.13}$$

for N >> P. The estimated running time of the 2-D FFT algorithm for $2k$ < (N/P) is presented in Table 4.6. It is based on Equation 4.14:

$$t_{\mathrm{E\text{-}fft}2}(N) = 2\,N^2/P(\log_2 N)u(N,P) + N^2/P\;u_{\mathrm{DRAM}}(N,P) \tag{4.14}$$

where $u(N, P)$ is defined as in Equation 4.9 and listed in Table 4.7. The parameter $u_{\mathrm{DRAM}}(N, P)$ is the unit time of reading and writing a complex number from and to the DRAM with congestion delay found from experiment, and is listed in Table 4.8.

Table 4.6 The estimated running time for 2-D FFT for $2k < (N/P)$ with shared-memory.

	8 x 8	**16 x 16**	**32 x 32**	**64 x 64**	**128 x 128**
P=1	0.002021	0.009048	0.037552	0.161429	0.683195
P=2	0.001228	0.005172	0.020977	0.088500	0.372199
P=4	0.000848	0.003398	0.013740	0.057035	0.228818
P=8	0.000686	0.002735	0.010733	0.045891	0.182538

Table 4.7 The sequential time u (in nsecs) of 2-D FFT.

u	8 x 8	16 x 16	32 x 32	64 x 64	128 x 128
P=1	4685.8	3688.0	3329.8	3003.8	2792.0
P=2	4903.2	3819.4	3383.8	3184.4	3007.6
P=4	5431.6	4120.2	3493.8	3493.4	3193.8
P=8	6490.0	4738.2	4028.4	4295.9	4028.3

Table 4.8 The time u_{DRAM} (in nsecs) for matrix transposition for 2-D FFT.

u_{DRAM}	8 x 8	16 x 16	32 x 32	64 x 64	128 x 128
P=1	3461.8	5840.8	3373.9	2621.3	1083.1
P=2	8967.0	9849.6	7132.9	5000.0	3328.1
P=4	20438.5	20126.0	18733.3	13777.3	11150.6
P=8	46870.0	47561.5	43571.3	38080.0	32733.9

4.4 Performance Results on TurboNet

The 1-D FFT and 2-D FFT algorithms for $2k < (N/P)$ have been implemented on TurboNet. Results are presented in this section. In all of our FFT implementations, the twiddle factors (or coefficients) are calculated and stored in a table in the DRAM. Each processor fetches the table values only once and stores them in its internal memory before starting any calculation.

4.4.1 1-D FFT

The 1-D FFT algorithm has been tested in two ways. One way uses message-passing only, whereas another one uses shared-memory only. From the experimental results of the

1-D FFT in Tables 4.9 to 4.12, as our theoretical analysis predicted, the message-passing method is shown indeed to be superior to the shared-memory one, and both experimental running times are close to the theoretical estimations. Figure 4.6 displays the experimental results for both versions, under $P = 8$. In Figures 4.7 and 4.8, the theoretical times of both communication methods for $P = 8$ are shown together with the experimental times. The differences between the theoretical and experimental times for both versions are shown in Figures 4.9 and 4.10. In our implementation, programs for complex and real FFT's were written and tested separately. The results show that the real FFT is a little faster than the complex one. All results given here are for the complex FFT.

The potential advantage of using simultaneously both the shared-memory and message-passing methods in 1-D FFT is not obvious; further research is needed.

Table 4.9 Execution time for 1-D FFT with message-passing.

	N=64	**N=128**	**N=256**	**N=512**	**N=1024**
P=1	0.001169	0.002613	0.005820	0.012928	0.028471
P=2	0.000781	0.001660	0.003558	0.007712	0.016638
P=4	0.000530	0.001049	0.002155	0.004527	0.009590
P=8	0.000377	0.000682	0.001314	0.002656	0.005483

Table 4.10 Speedup of 1-D FFT with message-passing.

	N=64	**N=128**	**N=256**	**N=512**	**N=1024**
P=2	1.50	1.57	1.64	1.68	1.71
P=4	2.21	2.49	2.70	2.86	2.97
P=8	3.10	3.83	4.43	4.87	5.19

Table 4.11 Efficiency of 1-D FFT with message-passing.

	N=64	N=128	N=256	N=512	N=1024
P=2	0.75	0.79	0.82	0.84	0.86
P=4	0.55	0.62	0.68	0.71	0.74
P=8	0.39	0.48	0.55	0.61	0.65

Table 4.12 Execution time for 1-D FFT with shared-memory.

	N=64	N=128	N=256	N=512	N=1024
P=1	0.001169	0.002613	0.005820	0.012928	0.028471
P=2	0.000924	0.001926	0.004092	0.008741	0.018736
P=4	0.000823	0.001617	0.003270	0.006512	0.013419
P=8	0.000882	0.001676	0.003260	0.006545	0.013221

Table 4.13 Speedup of 1-D FFT with shared-memory.

	N=64	N=128	N=256	N=512	N=1024
P=2	1.26	1.36	1.42	1.48	1.52
P=4	1.42	1.62	1.78	1.99	2.12
P=8	1.33	1.56	1.79	1.99	2.15

Table 4.14 Efficiency of 1-D FFT with shared-memory.

	N=64	N=128	N=256	N=512	N=1024
P=2	0.63	0.68	0.71	0.74	0.76
P=4	0.36	0.40	0.44	0.50	0.53
P=8	0.17	0.19	0.22	0.25	0.27

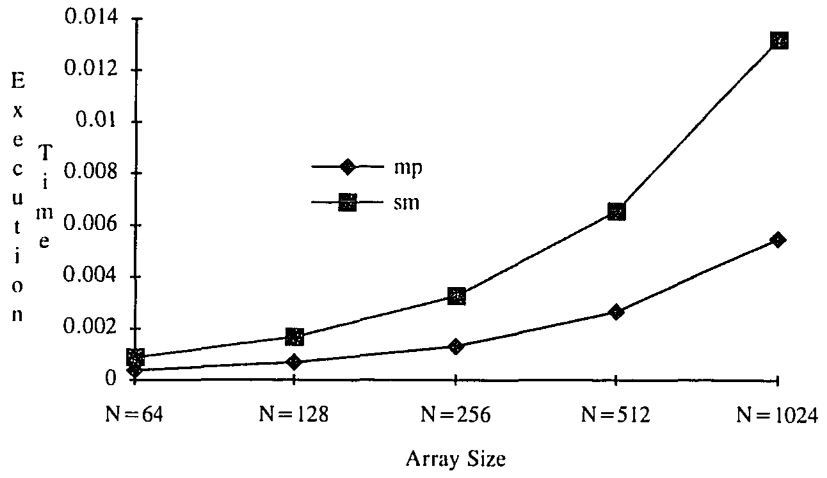


Figure 4.6 Execution time for 1-D FFT Vs array size, with P = 8.

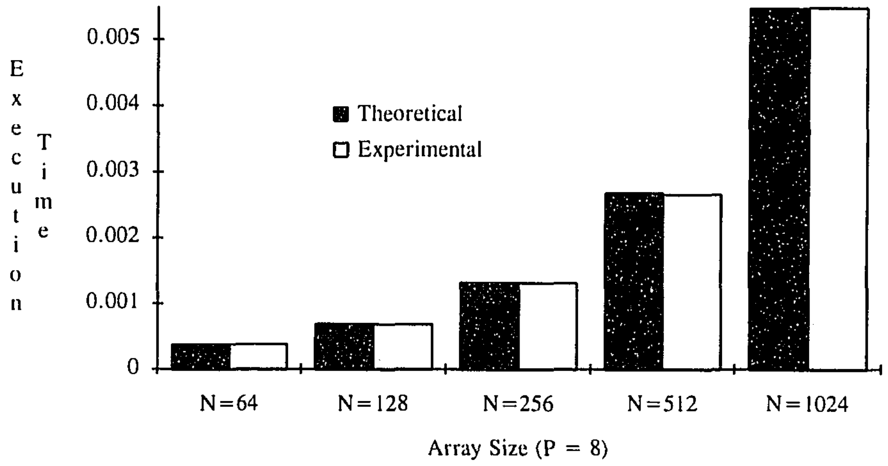


Figure 4.7 Comparison of theoretical and experimental running times for 1-D FFT with message-passing.

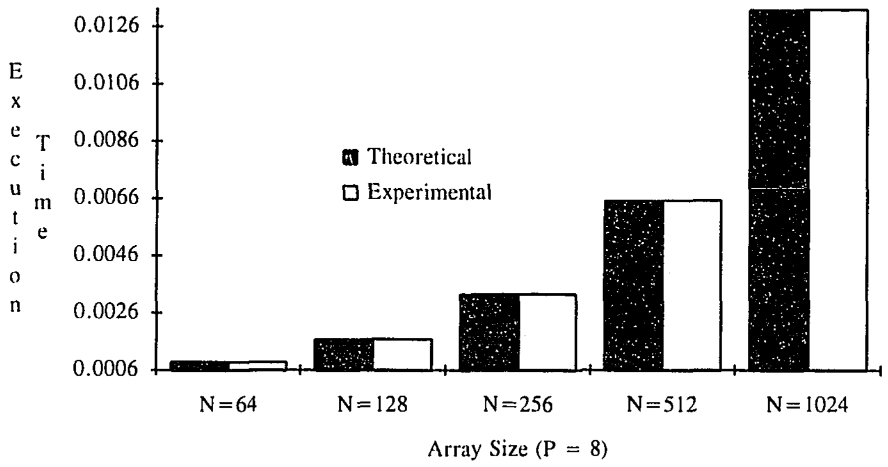


Figure 4.8 Comparison of theoretical and experimental running times for 1-D FFT with shared-memory.

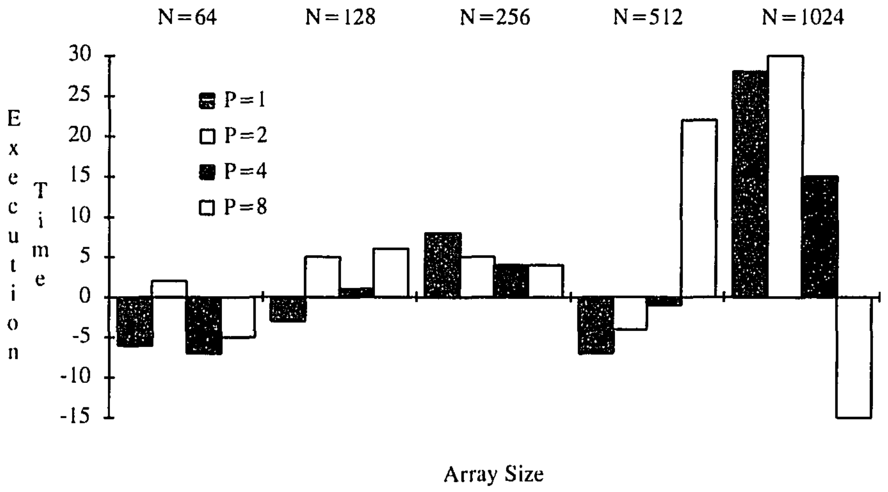


Figure 4.9 The difference of theoretical and experimental running times for the message-passing version of the 1-D FFT.

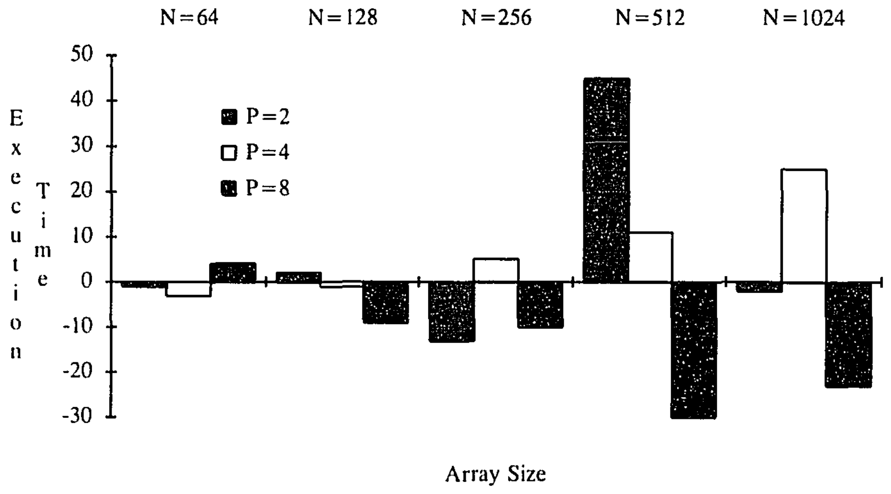


Figure 4.10 The difference of theoretical and experimental running times the shared-memory version of 1-D FFT.

4.4.2 2-D FFT

The 2-D FFT algorithm for $2k < (N/P)$ also has been implemented on TurboNet. Two groups of execution times are given; one does not include the time for matrix transposition (Table 4.15) while the other does (Table 4.18). The purpose of this is to show how much time is needed for transposition. The experimental times (with matrix transposition) are also compared with the theoretical ones in Figure 4.11 to show that they are very close to each other. The differences between them are shown in Figure 4.12.

Table 4.15 Execution time for 2-D FFT (without matrix transposition).

	8 x 8	16 x 16	32 x 32	64 x 64	128 x 128
P=1	0.001799	0.007557	0.034086	0.150738	0.665432
P=2	0.000942	0.003910	0.017320	0.078233	0.344841
P=4	0.000522	0.002108	0.008952	0.043000	0.183083
P=8	0.000312	0.001214	0.005155	0.026388	0.115527

Table 4.16 Speedup of 2-D FFT (without matrix transposition).

	8 x 8	16 x 16	32 x 32	64 x 64	128 x 128
P=2	1.91	1.93	1.97	1.93	1.93
P=4	3.45	3.58	3.81	3.51	3.63
P=8	5.77	6.22	6.61	5.71	5.76

Table 4.17 Efficiency of 2-D FFT (without matrix transposition).

	8 x 8	16 x 16	32 x 32	64 x 64	128 x 128
P=2	0.95	0.97	0.98	0.96	0.96
P=4	0.86	0.9	0.95	0.88	0.91
P=8	0.72	0.78	0.83	0.71	0.72

Table 4.18 Execution time for 2-D FFT (with matrix transposition).

	8 x 8	16 x 16	32 x 32	64 x 64	128 x 128
P=1	0.002078	0.009052	0.037546	0.161483	0.683177
P=2	0.001229	0.005171	0.020971	0.088475	0.372105
P=4	0.000849	0.003396	0.013748	0.057108	0.228764
P=8	0.000687	0.002736	0.010732	0.045888	0.182566

Table 4.19 Speedup of 2-D FFT (with matrix transposition).

	8 x 8	**16 x 16**	**32 x 32**	**64 x 64**	**128 x 128**
P=2	1.78	1.84	1.89	1.91	1.94
P=4	2.77	3.16	3.28	3.29	3. 45
P=8	3.74	4.22	4.60	5.03	5.50

Table 4.20 Efficiency of 2-D FFT (with matrix transposition).

	8 x 8	**16 x 16**	**32 x 32**	**64 x 64**	**128 x 128**
P=2	0.89	0.92	0.94	0.96	0.97
P=4	0.69	0.79	0.82	0.82	0.86
P=8	0.47	0.53	0.57	0.63	0.69

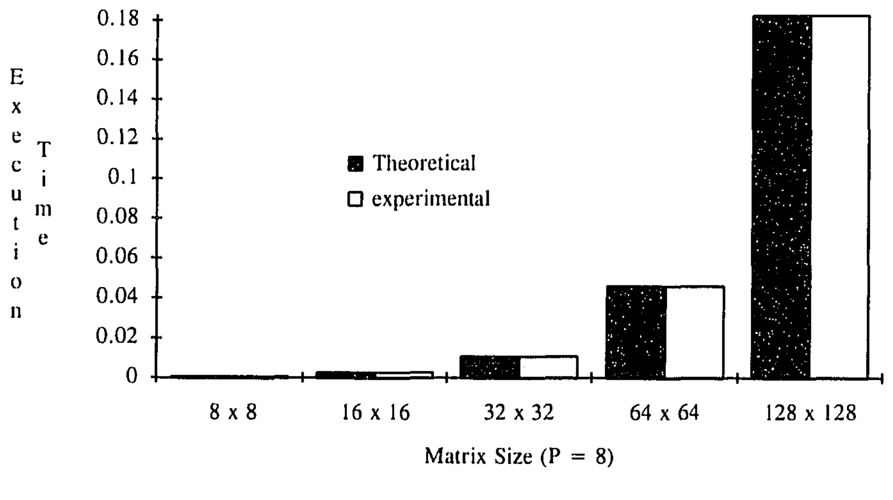


Figure 4.11 Comparison of theoretical and experimental running times for 2-D FFT with shared-memory.

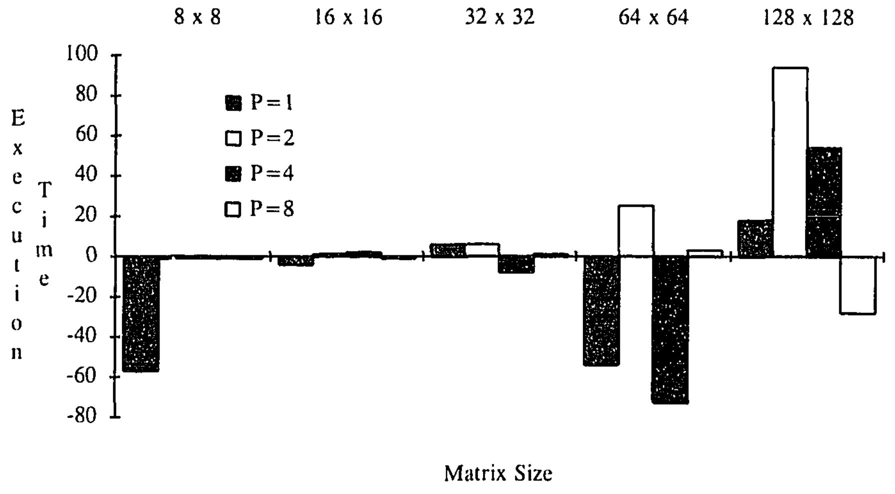


Figure 4.12 The difference of theoretical and experimental running times for the 2-D FFT.

In the above implementations, elements of the intermediate matrix were written back to the shared memory by each processor, and the transposition was carried out there by all processors. To reduce the contention in the shared memory incurred by this action, in an improved implementation each processor using its DMA controller writes its intermediate matrix into its own local memory, then reads data column-by-column from the latter matrix and writes them row-by-row into the shared memory. This not only carries out matrix transposition, but also cuts down on the number of shared-memory accesses for each element of data by each processor from 4 to 2. However, this implementation has assumes that the internal memory of each processor is big enough to hold all its output data plus $2k$ rows of input. Furthermore, the overlapping of input, calculation and output in the algorithm has changed its purpose solely for the speed of execution.

We have also taken an additional step to reduce the contention in the shared memory, that is, we delay each processor by a certain amount of time according to:

$$T_d = (\text{ Processor ID})\, t_{\text{DMA}}\, M \,\alpha\, \beta \qquad (4.15)$$

where t_{DMA} is the average time for a DMA controller to transfer a word from the shared memory to the local memory, M is the number of words to be transferred ($M = 2N^2/P$, if the size of the internal RAM of each processor is larger than the size of its matrix; otherwise, M equals the half size of the RAM), α is a fraction ranging from 0.2 to 0.33 and is determined by experiment, and β is a number proportional to $\log_2 P$, also determined by experiment. Results for the resulting improved method are shown in Tables 4.21 to 4.23. In Figures 4.10 and 4.11, the execution times and speedups are compared for the original (from Table 4.18) and improved methods. It can be concluded that the improved method results in significant performance improvement.

Table 4.21 Execution time for 2-D FFT with the improved method.

	8 x 8	**16 x 16**	**32 x 32**	**64 x 64**	**128 x 128**
P=1	0.002008	0.008182	0.034783	0.151509	0.667038
P=2	0.001126	0.004450	0.018400	0.079082	0.343463
P=4	0.000700	0.002530	0.010500	0.045300	0.190800
P=8	0.000587	0.001800	0.006800	0.029372	0.118000

Table 4.22 Speedup of 2-D FFT with the improved method.

	8 x 8	16 x 16	32 x 32	64 x 64	128 x 128
P=2	1.78	1.84	1.89	1.92	1.94
P=4	2.87	3.23	3.31	3.34	3.50
P=8	3.42	4.55	5.12	5.16	5.65

Table 4.23 Efficiency of 2-D FFT with the improved method.

	8 x 8	16 x 16	32 x 32	64 x 64	128 x 128
P=2	0.89	0.92	0.95	0.96	0.97
P=4	0.72	0.81	0.83	0.84	0.87
P=8	0.43	0.57	0.64	0.65	0.71

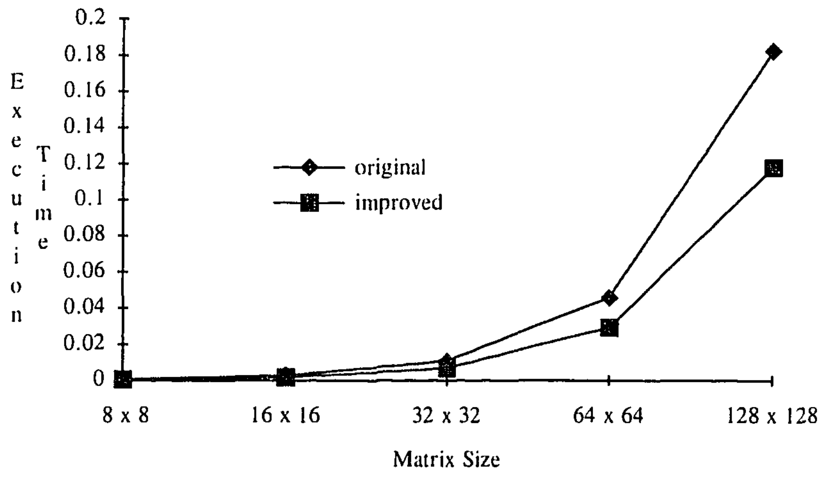


Figure 4.13 Execution time for 2-D FFT Vs matrix size, with $P = 8$.

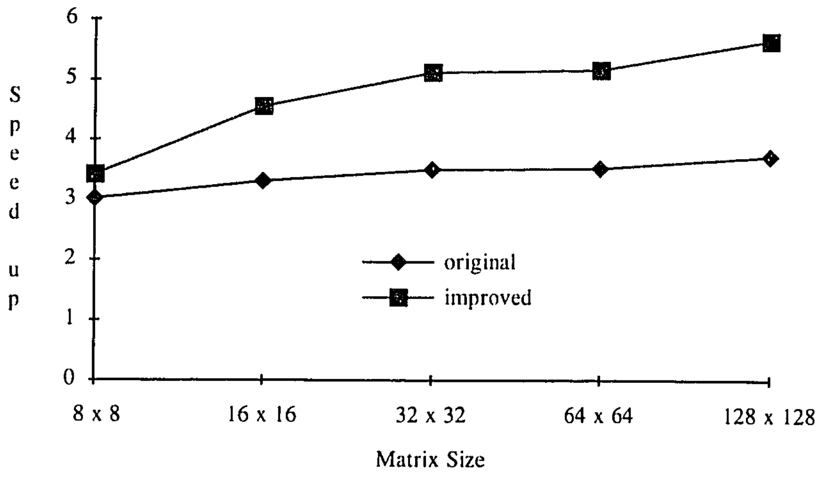


Figure 4.14 Speed up of 2-D FFT Vs matrix size, with $P = 8$.

CHAPTER 5

MATRIX MULTIPLICATION

The numerical solution of many problems reduces in part or fully to various matrix operations. A very common operation is the multiplication of matrices. A matrix can be "full" (with an insignificant number of zero elements), banded (non-zero elements clustered along the diagonal of the matrix), or sparse (with a more complicated structure in the locations of non-zero elements). In this chapter we present an algorithm for the multiplication of full matrices, its performance analysis, and experimental results on TurboNet. Naturally for TurboNet we have developed three versions of the algorithm based on message-passing, shared-memory and hybrid communication paradigms.

5.1 Matrix Multiplication Algorithm

The multiplication of two matrices A and B of size L by N and N by M, respectively, is a matrix C of size L by M whose elements are given by

$$c_{ij} = \sum_{k=0}^{N-1} a_{ik} b_{kj} \tag{5.1}$$

for $i = 0, 1, ..., L\text{-}1$, $j = 0, 1, ..., M\text{-}1$ and $k = 0, 1, ..., N\text{-}1$. In the following discussion, we assume that L, N, M and P (the number of processors) are always powers of 2.

The fastest possible algorithm for the multiplication of two N by N matrices on the hypercube requires N^3 processors and consumes time $O(\log_2 N)$ [5]. The basic algorithm chosen here is the popular simple algorithm of Fox [4, 37] that assumes a message-passing architecture. The assignment of matrix elements to the hypercube processors assumes the mapping of the matrix mesh to the hypercube. The run time of this algorithm is $O(N^3/P + \sqrt{P}\log_2 P + N^2/\sqrt{P}\log_2 P)$, where P is the number of processors. It uses broadcasts of blocks of A in rows, and circular upward shifts of block B in columns. Initially, diagonal blocks A_{ii} are selected for broadcast. The algorithm performs $\sqrt{P}$ iterations of the following steps:

- Broadcast the selected block of A among $\sqrt{P}$ processors in the same row.
- Multiply the received block of A with the contained block of B.
- Shift the block of B to the processor directly above, assuming wraparound in columns.
- Select the block $A_{\mathrm{i,(j+1)mod}\,P}$ for the next row broadcast.

To understand the basic idea behind this algorithm, relevant calibration follows.

To perform the multiplication in a parallel way, let us expand Equation 5.1 as follows

$$c_{ij} = \sum_{k=0}^{N/2-1} a_{ik}b_{kj} + \sum_{k=N/2}^{N-1} a_{ik}b_{kj} \tag{5.2}$$

The sum in the calculation of c_{ij} is split into two parts. The first part involves the first $N/2$ elements from the i^{th} row of A and the j^{th} column of B, while the second part involves the last $N/2$ elements of each of them, as illustrated in Figure 5.1. Both parts can be further decomposed successively into halves until each resulting part contains only one element from each matrix, thus achieving a complete expansion of Equation 5.1.

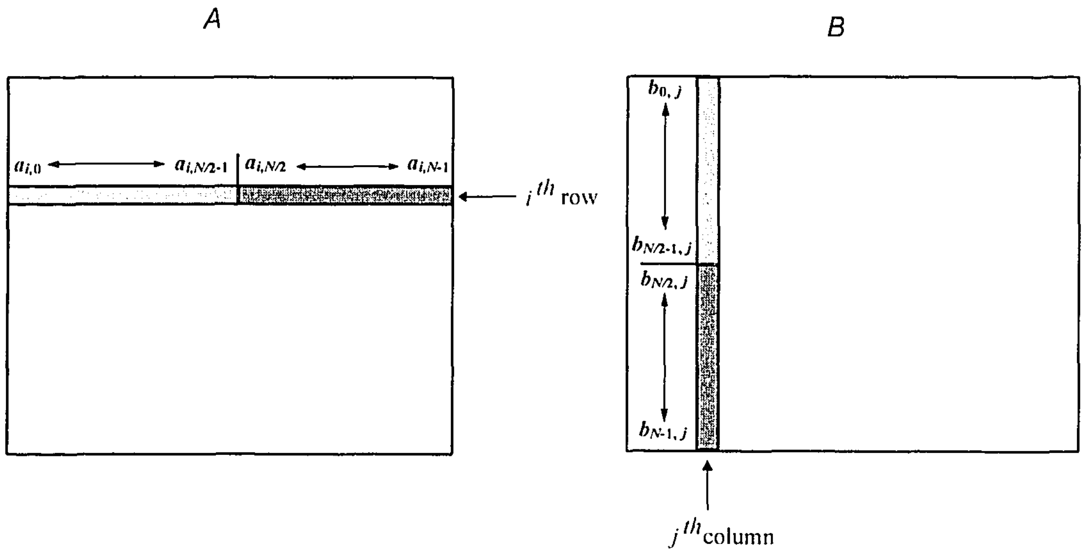


Figure 5.1 The decomposition for c_{ij}.

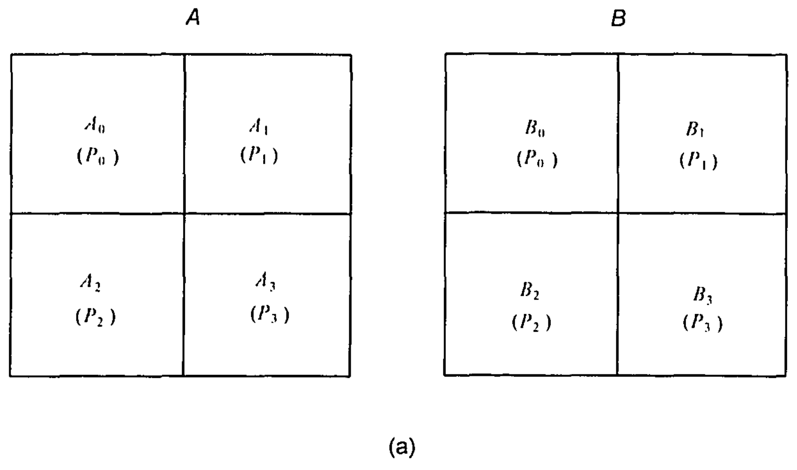
Now assume a 2-D hypercube (i.e. four processors) for the matrix multiplication in Equation 5.1. Both matrices A and B are decomposed into four submatrices, and each one of them is assigned to a processor as indicated in Figure 5.2 (a). Also the product C will be distributed among the processors, as shown in Figure 5.2 (b). From Equation 5.2, we know that each c_{ij} of C_0 is produced from elements of the i^{th} rows of A_0 and A_1 and the j^{th} columns of B_0 and B_2, as in the first line of the following equation:

$$\begin{cases} C_0 = A_0 B_0 \;\vdots + A_1 B_2 \\ C_1 = A_0 B_1 \;\vdots + A_1 B_3 \\ C_2 = A_3 B_2 \;\vdots + A_2 B_0 \\ C_3 = A_3 B_3 \;\vdots + A_2 B_1 \end{cases} \tag{5.3}$$

$$\underleftrightarrow{\text{Phase 1} \;\vdots\; \text{Phase 2}}$$

From this Equation it is seen that Phase 1 (i.e. generating A_0B_0 for C_0, A_0B_1 for C_1, etc.) requires communication from $P_0(P_3)$ to $P_1(P_2)$ for submatrix $A_0(A_3)$, while in Phase 2 $P_1(P_2)$ sends $A_1(A_2)$ to $P_0(P_3)$. Also notice that P_0 and P_2 (P_1 and P_3) need to exchange their submatrices B_0 and B_2 (B_1 and B_3) with each other before the calculation in Phase 2 can start (this scheme is shown in Figure 5.3). In order to save time, these exchanges are initiated at the beginning of Phase 1, and the processors have to wait for the submatrices of B before starting the calculation in Phase 2 only if the exchanges are still going on.

In the previous example each submatrix of A had the same number of columns as the number of rows in each submatrix of B, because the square root of P (i.e. 4 in the example) was an integer (i.e. 2). Therefore, the processors in the system could be arranged in a square mesh configuration. However, this square mesh configuration is not possible if the square root of P is not an integer. For example, the processors in the 3-D hypercube (P = 8) can only be arranged in a 2x4 rectangular mesh. As a result in the latter case, each submatrix in each processor needs to be divided again in half in the column or row direction, so that each resulting submatrix of A has the same number of columns as the number of rows in each resulting submatrix of B, as shown in Figure 5.4. Virtual processing is then needed, because the 3-D hypercube will emulate a 4-D hypercube (i.e. P = 16) in order to perform the calculations. Each physical processor will emulate two virtual processors by dividing its time between two submatrices assigned to it (for example, P_0 will first produce A_{00} B_{00} for C_{00} and then A_{00} B_{01} for C_{01}). There are four phases of operation in this 3-D hypercube example.



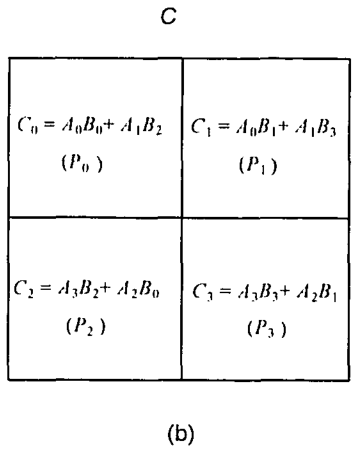
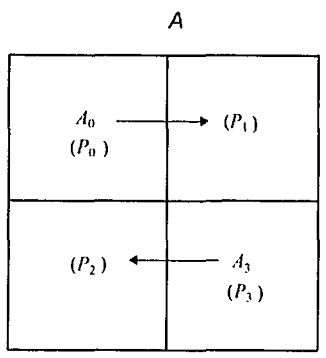
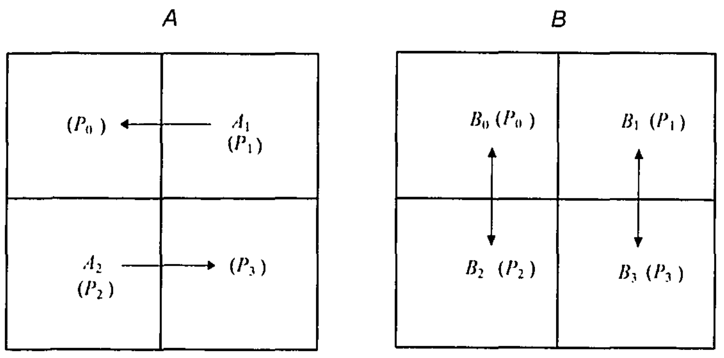

C
$C_0 = A_0B_0 + A_1B_2$ (P_0)
$C_1 = A_0B_1 + A_1B_3$ (P_1)
$C_2 = A_3B_2 + A_2B_0$ (P_2)
$C_3 = A_3B_3 + A_2B_1$ (P_3)
(b)

Figure 5.2 The decomposition of the matrices A, B and C for four processors.

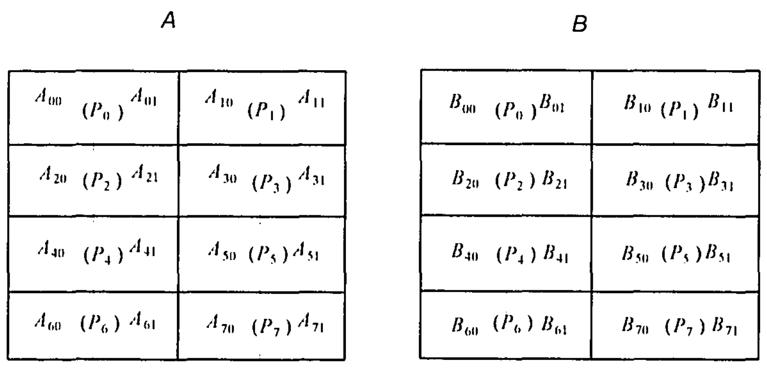


(a) Phase 1



(b) Phase 2

Figure 5.3 The communication scheme according to Equation 5.3.



A
B
A_{00} (P_0) A_{01}
A_{10} (P_1) A_{11}
A_{20} (P_2) A_{21}
A_{30} (P_3) A_{31}
A_{40} (P_4) A_{41}
A_{50} (P_5) A_{51}
A_{60} (P_6) A_{61}
A_{70} (P_7) A_{71}
B_{00} (P_0) B_{01}
B_{10} (P_1) B_{11}
B_{20} (P_2) B_{21}
B_{30} (P_3) B_{31}
B_{40} (P_4) B_{41}
B_{50} (P_5) B_{51}
B_{60} (P_6) B_{61}
B_{70} (P_7) B_{71}

Figure 5.4 The decomposition of the matrices A and B for the 3-D hypercube.

Using two processors to implement Equation 5.1 is a special case which can be speeded up by selecting a decomposition scheme for matrices different from the previously presented one for the 3-D hypercube. More specifically, we divide the matrices A and B in half by rows and columns, respectively, and no further decomposition of submatrices inside each processor is needed. In this case communication is necessary only for the submatrices of B. This process is shown in Figure 5.5.

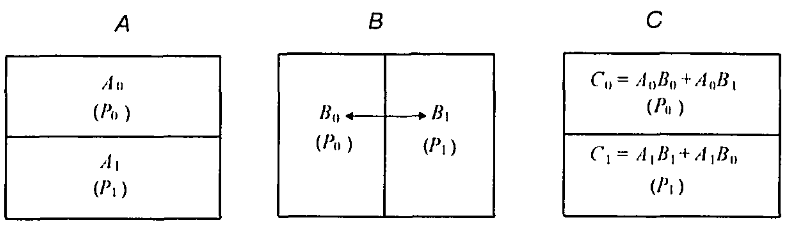

A
B
C
A_0 (P_0)
A_1 (P_1)
B_0 (P_0)
B_1 (P_1)
$C_0 = A_0B_0 + A_0B_1$ (P_0)
$C_1 = A_1B_1 + A_1B_0$ (P_1)

Figure 5.5 The decomposition of matrices A, B and C for the 1-D hypercube.

To summarize the basic algorithm, the matrices A and B are first divided in such a way that the number of columns in each submatrix of A and the number of rows in each submatrix of B are the same, and then (except for the 1-D hypercube case) an iterative process is employed that each time broadcasts one of the submatrices of A horizontally (i.e. to all other processors in the same row) and shifts upward the submatrices of B (no shifting is needed in the last phase), so that each processor generates one of the submatrices of C. The number of phases is equal to the number of rows in the processor mesh.

When the number of dimensions in the hypercube is less than or equal to three, all communications are local (for higher dimensional hypercubes, the broadcasting of submatrices of A is not a local communication when utilizing the message-passing paradigm). This means that the message-passing communication should be expected to be efficient. We have developed three versions of the matrix multiplication algorithm based on the message-passing, shared-memory and hybrid paradigms. The following sections present a theoretical analysis that includes estimated running times, and implementation results on TurboNet.

5.2 Theoretical Analysis

Since TurboNet is currently a 3-D hypercube system, our theoretical analysis assumes at most eight processors. For simplicity, we assume that all matrices are square of size of N by N. Based on Equation 5.1, the sequential multiplication time is

$$t_{\text{sq}}(N) = 2N^3\, t_{\text{cal}} = O(N^3) \tag{5.4}$$

where t_{cal} is the time for one multiplication or one addition.

Let us first look at the 1-D hypercube case. Figure 5.6 shows the DDG of the matrix multiplication algorithm for the 1-D hypercube, derived from Figure 5.5. There are two phases in this algorithm. In the first phase, P_0 and P_1 produce A_0B_0 and A_1B_1, respectively. In the last phase the processors exchange B_0 and B_1 with each other and generate the final results by adding A_0B_1 and A_1B_0 to their previous products, respectively. The thick arrows crossing the vertical dotted line in the figure represent the exchanges. The purpose of putting the exchange process in the last phase instead of in the first one is to see how much effect the communication overhead has on the speedup and efficiency of the algorithm, since the calculation time is expected to be larger than the communication overhead (the exchange process would finish before the calculation of phase 1 if it is initiated at the beginning of phase 1).

The calculation time in each phase is $(N^3/2)\,t_{\text{cal}}$. The communication in the last phase can be implemented in three ways. In the hybrid way, P_0 can use a communication channel while P_1 can use the shared memory (or vice verse). The communication times for P_0 and P_1 are $(N^2/2)\,t_{\text{com}}$ and $(N^2/2)\,t_{\text{DRAM}}$, respectively, where t_{com} is the time to transfer one word between neighbors in the hypercube via the direct link, and t_{DRAM} is the time to transfer the same word but through the shared memory (the DRAM). From our experiment on TurboNet, t_{com} is greater than t_{DRAM}, therefore the total running time for the hybrid implementation is

$$t_{(2\text{p},\text{hy})}(N) = 2\,(N^3/2)\,t_{\text{cal}} + (N^2/2)\,t_{\text{com}} = O(N^3) \tag{5.5}$$

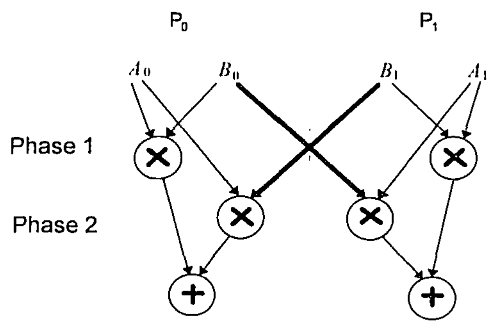


Figure 5.6 DDG of matrix multiplication for the 1-D hypercube.

When using message-passing only with the half-duplex physical channel between the two processors, the communication time is $2(N^2/2)\,t_{\mathrm{com}} = N^2\,t_{\mathrm{com}}$. The total running time for the message-passing implementation is

$$t_{(2\,\mathrm{p,mp})}(N) = 2\,(N^3/2)\,t_{\mathrm{cal}} + N^2\,t_{\mathrm{com}} = O(N^3) \qquad (5.6)$$

With the shared-memory implementation that involves congestion delay, the communication time would be as much as $2(N^2/2)\,t_{\mathrm{DRAM}} = N^2\,t_{\mathrm{DRAM}}$, and the total running time in this case could be at most

$$t_{(2\,\mathrm{p,sm})}(N) = 2\,(N^3/2)\,t_{\mathrm{cal}} + N^2\,t_{\mathrm{DRAM}} = O(N^3) \qquad (5.7)$$

From Equations 5.5 to 5.7, it is seen that the performance of the algorithm depends heavily on the communication time, and the hybrid version of the algorithm is estimated to have better performance than the other two versions. The actual estimation formulas are

$$t_{\mathrm{E}-(2\,\mathrm{p,hy})}(N) = N^3\,u_{(2\,\mathrm{p,hy})}(N) + (N^2/2)\,u_{\mathrm{hy}}(N) \qquad (5.8)$$

$$t_{\mathrm{E}-(2\,\mathrm{p,mp})}(N) = N^3\,u_{(2\,\mathrm{p,mp})}(N) + N^2\,u_{\mathrm{mp}}(N) \qquad (5.9)$$

$$t_{\mathrm{E}-(2\,\mathrm{p,sm})}(N) = N^3\,u_{(2\,\mathrm{p,sm})}(N) + N^2\,u_{\mathrm{sm}}(N) \qquad (5.10)$$

where $u_{(2\mathrm{p,hy})}(N)$, $u_{(2\mathrm{p,mp})}(N)$ and $u_{(2\mathrm{p,sm})}(N)$ are the amounts of times for one multiplication or one addition, including the time for loading operands and storing results, in the hybrid, message-passing and shared-memory implementations, respectively. Also, $u_{\mathrm{hy}}(N)$, $u_{\mathrm{mp}}(N)$ and $u_{\mathrm{sm}}(N)$ represent the communication times for one word in the three communication paradigms, respectively. They are obtained from experiment and are listed in Tables 5.2 and 5.3. The estimated running times are listed in Table 5.1. The calculation time for a single processor is estimated from Equation 5.11 and is also included in Table 5.1. It is obtained by

$$t_{E\text{-sq}}(N) = 2N^3\, u_{sq}(N) \qquad (5.11)$$

where $u_{sq}(N)$ is defined similarly to $u_{2p}(N)$ but now with one processor. Its values are also obtained from experiment and are listed in Table 5.2.

Table 5.1 Estimated running time for matrix multiplication on the 0-D and 1-D hypercubes.

		8 x 8	**16 x 16**	**32 x 32**	**64 x 64**	**128 x 128**
P = 1		0.000520	0.003866	0.029443	0.224348	1.687490
P = 2	**hy**	0.000404	0.002880	0.022187	0.175391	1.396054
	mp	0.000521	0.003720	0.027400	0.202813	1.500705
	sm	0.000509	0.003398	0.024712	0.187066	1.467966

Table 5.2 The times $u_{(2p,\ xx)}$ (in nsecs) for matrix multiplication on the 0-D and 1-D hypercubes.

	8 x 8	**16 x 16**	**32 x 32**	**64 x 64**	**128 x 128**
u_{sq}	507.82	471.92	449.26	427.90	402.33
$u_{(2p,hy)}$	728.52	673.83	664.79	662.98	662.71
$u_{(2p,mp)}$	886.72	846.68	806.00	759.10	708.63
$u_{(2p,sm)}$	882.81	782.72	728.52	701.42	693.98

Table 5.3 The times u_{xx} (in nsecs) for matrix multiplication on the 1-D hypercube.

P = 2	**8 x 8**	**16 x 16**	**32 x 32**	**64 x 64**	**128 x 128**
u_{hy}	968.75	937.50	787.11	778.81	763.86
u_{mp}	1046.87	984.37	966.38	932.37	890.66
u_{sm}	875.00	750.00	820.30	779.30	768.20

The DDG for the 2-D hypercube is shown in Figure 5.7. As discussed in the example of using the 2-D hypercube in Section 5.1, there are two types of communication in this case; the communication in both phases of submatrices of A between processors in the same row and the exchange in phase 1 of submatrices of B between processors in the same column (the exchange finishes before the end of phase 1). All these communications are represented by thick arrows in the DDG. In phase 1, P_1 (P_2) receives one row of A_0 (A_3) from P_0 (P_3) and performs multiplications of this row with all columns of B_1 (B_2). This process is repeated $N/2$ times, until all rows of A_0 (A_3) have been multiplied by B_1 (B_2). In the meantime, the exchange of the submatrices of B is also in process. Therefore, the running time of each iteration in this phase is the calculation time $N^2/2\, t_{\text{cal}}$ plus the time for transferring $N/2$ words between two neighboring processors. In the hybrid implementation, as in the 1-D hypercube case, submatrices of A are transferred via the shared memory while submatrices of B are exchanged over the communication channels. This leads to a communication time in each iteration of $N/2\, t_{\text{DRAM}}$ and the same amount of maximum delay. In the message-passing implementation, since submatrices of A and B go through different channels, i.e. there is no congestion on the channels, the time for communication is $N/2\, t_{\text{com}}$. When only shared memory is used for communication, the amount of time is $N/2\, t_{\text{DRAM}}$, plus the maximum delay of $5N/2\, t_{\text{DRAM}}$ due to the congestion from exchanges of submatrices of A and B. The time for phase 2 is similar to the one for phase 1, except that there is a time delay of $N/2\, t_{\text{DRAM}}$ when using the shared memory in the shared-memory implementation only; since the exchange of submatrices of B has finished, the delay is caused by the communication of a submatrix of A between processors in another row. The total running times of the three implementations, therefore, are

$$t_{(4\,\mathrm{p,hy})}(N) = (N^3/2)\,t_{\mathrm{cal}} + N^2\,t_{\mathrm{DRAM}} = O(N^3) \tag{5.12}$$

$$t_{(4\,\mathrm{p,mp})}(N) = (N^3/2)\,t_{\mathrm{cal}} + (N^2/2)\,t_{\mathrm{com}} = O(N^3) \tag{5.13}$$

$$t_{(4\,\mathrm{p,sm})}(N) = (N^3/2)\,t_{\mathrm{cal}} + 2N^2\,t_{\mathrm{DRAM}} = O(N^3) \tag{5.14}$$

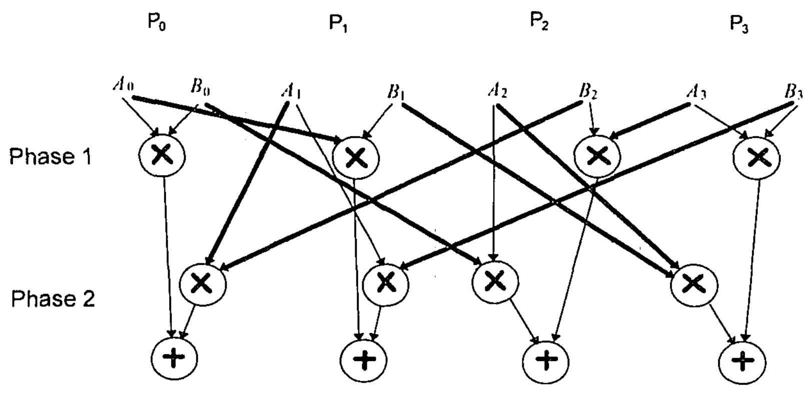


Figure 5.7 DDG of matrix multiplication for the 2-D hypercube.

From the three equations above, since from experiment $(t_{\mathrm{com}}/2) > t_{\mathrm{DRAM}}$, we predict that the hybrid version of the algorithm will provide the best performance. The estimated running times of the three implementations, based on Equations 5.15 to 5.17 below, are presented in Table 5.4.

$$t_{\mathrm{E}-(4\,\mathrm{p,hy})}(N) = N^3/2\;u_{(4\,\mathrm{p,hy})}(N) \tag{5.15}$$

$$t_{\mathrm{E}-(4\,\mathrm{p,mp})}(N) = N^3/2\;u_{(4\,\mathrm{p,mp})}(N) \tag{5.16}$$

$$t_{E-(4p,sm)}(N) = N^3/4\,(u_{(4p,sm1)}(N) + u_{(4p,sm2)}(N)) \tag{5.17}$$

In the equations, $u_{(4p,hy)}(N)$ and $u_{(4p,mp)}(N)$ are the amounts of time for one multiplication or one addition, and the transfer of at most $2/N$ and $1/N$ word with the hybrid and message-passing paradigms, respectively. The parameters $u_{(4p,sm1)}(N)$ and $u_{(4p,sm2)}(N)$ are the amounts of time for one multiplication or one addition, and the transfer in phases 1 and 2 of at most $6/N$ and $2/N$ word with the shared-memory paradigm, respectively. Their values are from experiment and are recorded in Table 5.5.

Table 5.4 Estimated running time for matrix multiplication with the 2-D hypercube.

P = 4	**8 x 8**	**16 x 16**	**32 x 32**	**64 x 64**	**128 x 128**
hy	0.000254	0.001459	0.010211	0.076225	0.567676
mp	0.000199	0.001398	0.010636	0.080952	0.637976
sm	0.000319	0.001713	0.011101	0.079013	0.604105

Table 5.5 The times $u_{(4p,xx)}$ (in nsecs) for matrix multiplication with the 2-D hypercube.

P = 4	**8 x 8**	**16 x 16**	**32 x 32**	**64 x 64**	**128 x 128**
$u_{(4p,hy)}$	992.19	712.40	623.23	581.55	541.38
$u_{(4p,mp)}$	777.33	682.62	649.17	617.61	608.42
$u_{(4p,sm1)}$	1531.91	939.64	713.62	607.04	579.41
$u_{(4p,sm2)}$	960.28	733.21	641.48	598.60	572.83

The 3-D hypercube implementation of matrix multiplication is similar to the 2-D case, except for the addition of two more phases, more shared memory delay, and that each processor has to perform matrix multiplication for two submatrices of A. In order to avoid confusion from line crossings, instead of drawing the entire DDG for eight processors, we show in Figure 5.8 the DDG for one processor (P_0) only. In the figure, a submatrix is identified with a processor number if it comes from another processor. The structures of the DDGs for the other processors are similar to this one except that processor numbers identifying some submatrices are different.

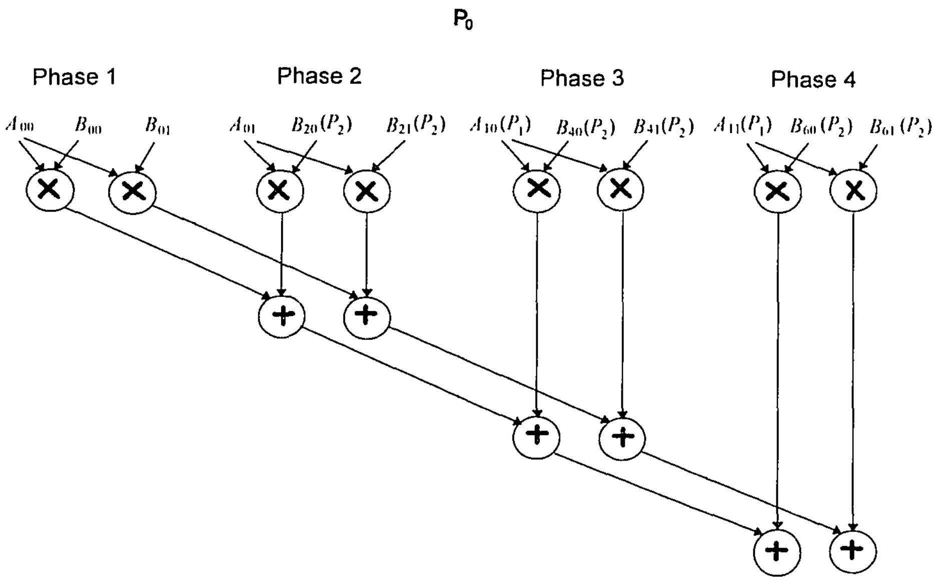


Figure 5.8 DDG of matrix multiplication for P_0 of the 3-D hypercube.

In each iteration of phase 1, each processor performs multiplications of two pairs of submatrices of A and B, as indicated in Figure 5.8. Also the communication of $N/4$ values is needed. The calculation time in each iteration is $N^2/4\, t_{\mathrm{cal}}$. For the hybrid implementation, the amount of time for the communication of $N/4$ words through the shared memory (the exchanges of submatrices of B are via communication channels) is

$N/4\ t_{\text{DRAM}}$ plus the same amount of time for maximum delay. In the message-passing implementation, it takes $N/4\ t_{\text{com}}$ to transfer the $N/4$ words through a channel. Since the number of these "calculate-communicate" iterations is $N/4$, and phases 2 to 4 are the same as phase 1, the running times of the hybrid and message-passing implementations, respectively, are

$$t_{(8\text{p},\text{hy})}(N) = (N^3/4)\ t_{\text{cal}} + (N^2/2)\ t_{\text{DRAM}} = O(N^3) \tag{5.17}$$

$$t_{(8\text{p},\text{mp})}(N) = (N^3/4)\ t_{\text{cal}} + (N^2/4)\ t_{\text{com}} = O(N^3) \tag{5.18}$$

The shared-memory implementation is a little more complicated than its counterpart in the 2-D hypercube case, due to the introduction of another shared memory module. As shown in Figure 5.4, processors P_0 to P_3 are in one Hydra board while the rest of the processors are in another, and hence the communications of submatrices of A need not cross the VME bus. This limits the congestion delay contributed by each processor when transferring submatrices of A to $N/4\ t_{\text{DRAM}}$ for one iteration. This delay exists in all four phases. In exchanges of submatrices of B, two processors in one board have to access the shared memory in another board, while the other two processors use only the shared memory in the same board. All these operations generate a maximum delay of $6\,N/4\ t_{\text{DRAM}}$ for each iteration in the ISB (which is connected to the shared memory) of each board, and this delay appears only in the first three phases. Therefore, with $N/4$ iterations in each phase, the total running time for the four phases in the shared-memory implementation is

$$t_{(8\text{p},\text{sm})}(N) = (N^3/4)\ t_{\text{cal}} + 13N^2/8\ t_{\text{DRAM}} = O(N^3) \tag{5.19}$$

From Equations 5.17 to 5.19, we see that the hybrid version of the algorithm has less communication time than the other two (since $(t_{\text{com}}/2) > t_{\text{DRAM}}$), hence its execution will be the fastest of all three. The estimation of running times for the 3-D hypercube will again be similar to the one for the 2-D hypercube. The estimated times are based on the following equations and are listed in Table 5.6:

$$t_{E-(8p,hy)}(N) = N^3/4\ u_{(8p,hy)}(N) \tag{5.20}$$

$$t_{E-(8p,mp)}(N) = N^3/4\ u_{(8p,mp)}(N) \tag{5.21}$$

$$t_{E-(8p,sm)}(N) = N^3/16\,(3u_{(8p,sm1)}(N) + u_{(8p,sm2)}(N)) \tag{5.22}$$

In the equations above, the definitions of $u_{(8p,hy)}(N)$ and $u_{(8p,mp)}(N)$ are the same as the ones for $u_{(4p,hy)}(N)$ and $u_{(4p,mp)}(N)$ in Equations 5.15 and 5.16, while $u_{(8p,sm1)}(N)$ and $u_{(8p,sm2)}(N)$ are the amounts of time for one multiplication or one addition, plus the transfer of at most $8/N$ and $2/N$ word, with the shared-memory paradigm, in phases 1 to 3 and 4, respectively. Their values are from experiment and are recorded in Table 5.7.

Table 5.6 Estimated running time for matrix multiplication with the 3-D hypercube.

P = 8	**8 x 8**	**16 x 16**	**32 x 32**	**64 x 64**	**128 x 128**
hy	0.000207	0.000874	0.004812	0.035357	0.263544
mp	0.000210	0.000876	0.005229	0.040400	0.326750
sm	0.000478	0.001800	0.008074	0.048980	0.315953

Table 5.7 The times $u_{(8p,xx)}$ (in nsecs) for matrix multiplication with the 3-D hypercube.

P = 8	8 x 8	16 x 16	32 x 32	64 x 64	128 x 128
$u_{(8p,hy)}$	1617.19	853.52	587.40	539.51	502.67
$u_{(8p,mp)}$	1640.63	855.47	638.31	616.46	623.23
$u_{(8p,sm1)}$	4552.08	2089.84	1132.32	829.14	640.00
$u_{(8p,sm2)}$	1281.25	761.72	545.41	502.01	490.52

In general, the running time of the matrix multiplication algorithm on a hypercube with P processors, where $P > 2$,

$$t_G(N) = (2N^3/P)\,t_{cal} + (N^2/r)\,t_x + T_{del} = O(N^3/P) \tag{5.23}$$

where t_x is the amount of time to transfer a word through the shared memory or communication channel(s), r is the number of rows in the processor array or the number of phases, and T_{del} is the congestion delay in the communication. In the hybrid implementation of the algorithm, T_{del} is induced only from the communication of submatrices of A through the shared memory. With the message-passing paradigm only, if P/r (the number of columns in the processor array) is less than or equal to 2, T_{del} is zero (i.e. all communications are local), otherwise it is the delay due to channel congestion. For the shared-memory implementation, T_{del} is caused by the communication of submatrices of A and B that requires sharing of the bus or memory.

5.3 Performance Results on TurboNet

The algorithms of matrix multiplication were implemented on TurboNet. This section presents experimental results, speedups, efficiencies, and comparisons of theoretical and experimental results.

In the 1-D hypercube implementation, as predicted in the last section, the hybrid version has the best performance among all three versions. The experimental results are shown in Table 5.8. Figure 5.9 compares execution times of the three versions of the algorithm on the 1-D hypercube, and the difference between the theoretical and experimental results is plotted in Figure 5.10.

Table 5.8 Execution time of matrix multiplication on the 0-D and 1-D hypercubes.

		8 x 8	16 x 16	32 x 32	64 x 64	128 x 128
P = 1		0.000523	0.003881	0.029438	0.224369	1.687509
P = 2	hy	0.000381	0.002851	0.022170	0.175384	1.396061
	mp	0.000513	0.003700	0.027417	0.202837	1.500771
	sm	0.000514	0.003387	0.024688	0.187088	1.467935

Table 5.9 Speedup of matrix multiplication for the 1-D hypercube.

P = 2	8 x 8	16 x 16	32 x 32	64 x 64	128 x 128
hy	1.37	1.36	1.33	1.33	1.22
mp	1.02	1.05	1.07	1.10	1.12
sm	1.02	1.15	1.19	1.20	1.15

Table 5.10 Efficiency of matrix multiplication for the 1-D hypercube.

P = 2	8 x 8	16 x 16	32 x 32	64 x 64	128 x 128
hy	0.68	0.68	0.66	0.66	0.61
mp	0.51	0.52	0.54	0.55	0.56
sm	0.51	0.57	0.59	0.60	0.57

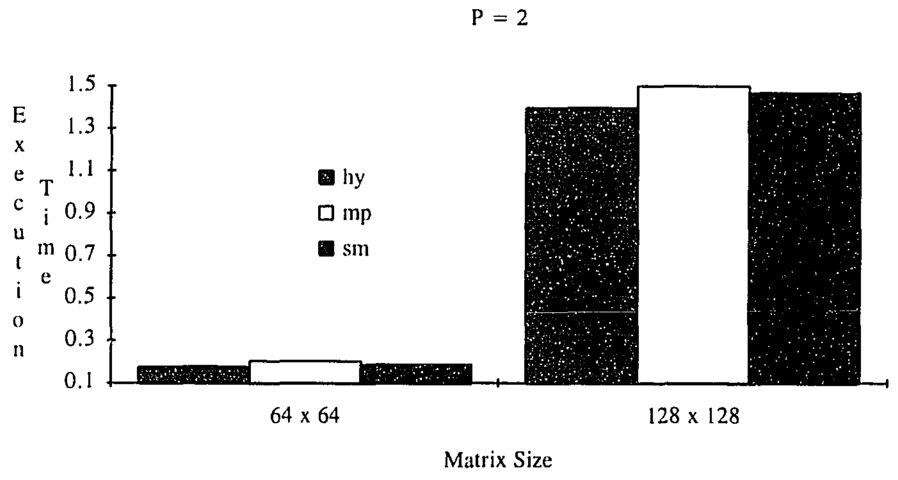


Figure 5.9 Performance comparison of the three versions for the 1-D hypercube.

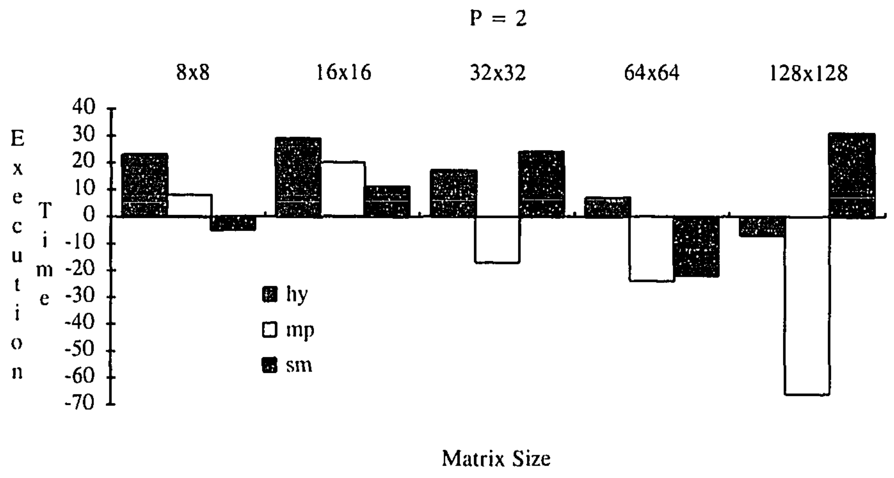


Figure 5.10 The difference of theoretical and experimental timings for the 1-D hypercube.

In the implementations of matrix multiplication on both the 2-D and 3-D hypercubes, the hybrid versions of the algorithm again have better performance than the others. These results agree with our earlier analysis according to Equations 5.12 to 5.14 and 5.17 to 5.19. Also notice that in Tables 5.11 and 5.14, where the experimental results for the 2-D and 3-D hypercubes are presented, as the size of the matrices increases, the message-passing version become slower than the shared-memory one. This is because the message-passing paradigm is not suitable for the communication of large amounts of data. Relevant speedups, efficiencies, performance comparisons, and differences between theoretical and experimental timings are all given below.

Table 5.11 Execution time of matrix multiplication on the 2-D hypercube.

P = 4	8 x 8	16 x 16	32 x 32	64 x 64	128 x 128
hy	0.000258	0.001483	0.010170	0.076173	0.567643
mp	0.000204	0.001390	0.010616	0.080993	0.638016
sm	0.000323	0.001702	0.011051	0.078957	0.604139

Table 5.12 Speedup of matrix multiplication for the 2-D hypercube.

P = 4	8 x 8	16 x 16	32 x 32	64 x 64	128 x 128
hy	2.03	2.62	2.89	2.95	2.97
mp	2.56	2.79	2.77	2.77	2.64
sm	1.62	2.28	2.66	2.84	2.79

Table 5.13 Efficiency of matrix multiplication for the 2-D hypercube.

P = 4	8 x 8	16 x 16	32 x 32	64 x 64	128 x 128
hy	0.51	0.65	0.72	0.74	0.74
mp	0.64	0.70	0.69	0.69	0.66
sm	0.40	0.57	0.67	0.71	0.69

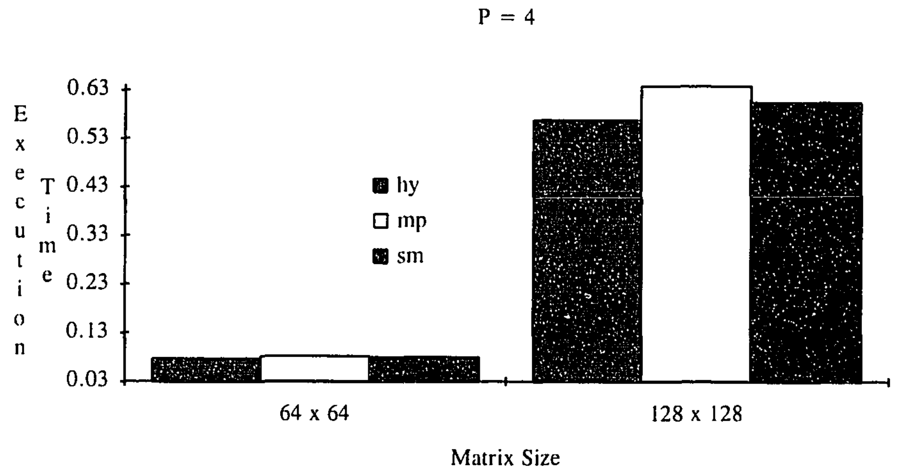


Figure 5.11 Performance comparison of the three versions for the 2-D hypercube.

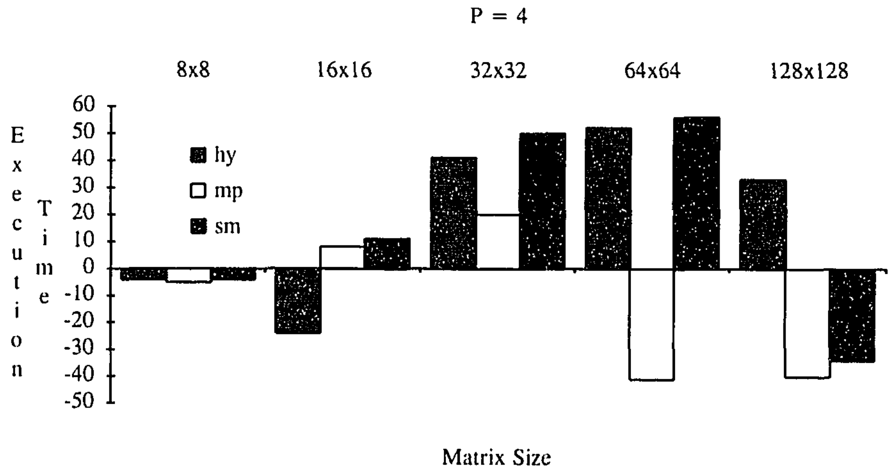


Figure 5.12 Difference of theoretical and experimental timings for the 2-D hypercube.

Table 5.14 Execution time of matrix multiplication on the 3-D hypercube.

P = 8	**8 x 8**	**16 x 16**	**32 x 32**	**64 x 64**	**128 x 128**
hy	0.000217	0.000897	0.004878	0.035379	0.263526
mp	0.000205	0.000889	0.005242	0.040346	0.326720
sm	0.000472	0.001789	0.008110	0.049000	0.316000

Table 5.15 Speedup of matrix multiplication for the 3-D hypercube.

P = 8	**8 x 8**	**16 x 16**	**32 x 32**	**64 x 64**	**128 x 128**
hy	2.41	4.33	6.03	6.34	6.40
mp	2.55	4.37	5.62	5.56	5.17
sm	1.12	2.17	3.63	4.58	5.34

Table 5.16 Efficiency of matrix multiplication for the 3-D hypercube.

P = 8	**8 x 8**	**16 x 16**	**32 x 32**	**64 x 64**	**128 x 128**
hy	0.30	0.54	0.75	0.79	0.80
mp	0.32	0.55	0.70	0.70	0.65
sm	0.14	0.27	0.45	0.57	0.67

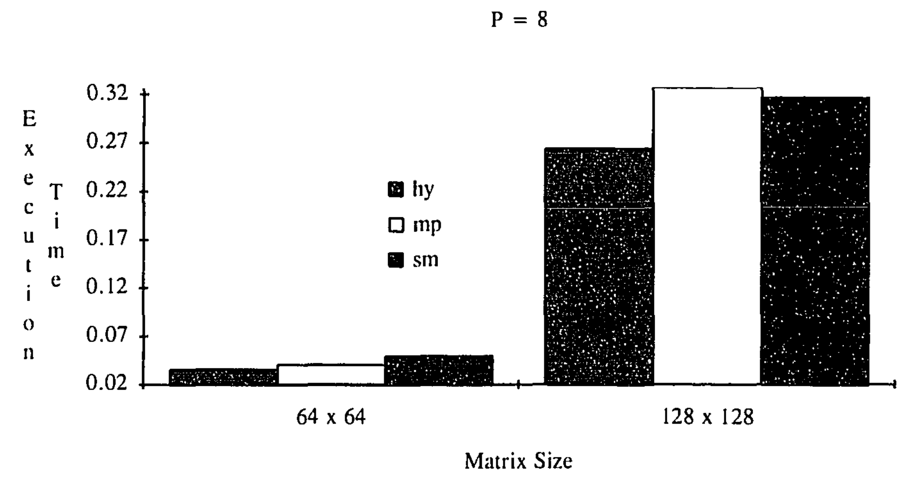


Figure 5.13 Performance comparison of the three versions for the 3-D hypercube.

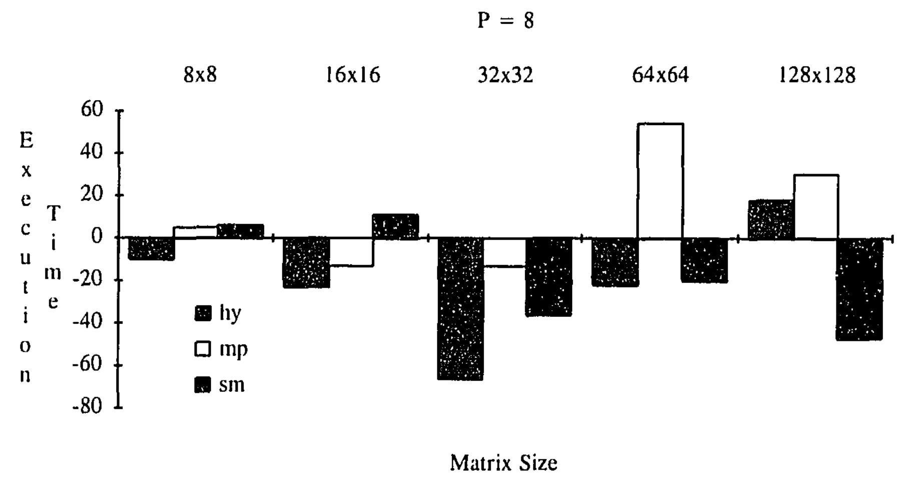


Figure 5.14 Difference of theoretical and experimental timings for the 3-D hypercube.

The execution time on the 0-D hypercube, shown in Table 5.8, involves only calculation time, i.e. the timer starts after both input matrices *A* and *B* are stored in the processor's memory. For the sake of comparison, another algorithm was also developed for a single processor so that data inputs and calculations are overlapped. Results for the latter case are presented in Table 5.17, together with pure calculation times.

Table 5.17 Two implementation results of matrix multiplication on the 0-D hypercube. Cal.: calculation time only, Inp. & Cal.: data input and calculation time.

P = 1	**8 x 8**	**16 x 16**	**32 x 32**	**64 x 64**	**128 x 128**
Cal.	0.000523	0.003881	0.029438	0.224369	1.687509
Inp. & Cal.	0.000590	0.004067	0.029824	0.224966	1.689134

CHAPTER 6

ADAPTIVE MULTIRESOLUTION STRUCTURES FOR IMAGE SEGMENTATION

In this chapter, an algorithm is described for the creation of region-adjacency-graph (RAG) pyramids on TurboNet. The implementation of the algorithm is discussed and performance results are presented for the three communication techniques which are supported by TurboNet's hybrid architecture. Each level of these hierarchies of irregular tessellations is generated by independent stochastic processes that adapt the structure of the pyramid to the content of the image. RAG pyramids can be used in multiresolution image analysis to extract connected components from labeled images. The experimental results indicate that efficient communication is vital to good performance of the algorithm.

6.1 Introduction

The structure of conventional multiresolution representations of images, i.e. image pyramids [15], is regular, with all parents having the same number of children arranged in a square neighborhood. The rigid structure of image pyramids is not optimal for feature extraction tasks where homogeneous images have to be delineated. Techniques have been presented that modify the links of the image pyramid in order to alleviate this problem, however they do not provide satisfactory or efficient solutions. In contrast, if the child-parent links are established according to local homogeneity criteria, a region adjacency graph (RAG) is used to represent each level of the pyramid [13]. RAG pyramids are hierarchies of RAG's [14]. The irregular structure of RAG pyramids adapts to the image content and is, therefore, optimal for feature extraction; it also preserves the logarithmic processing time property of regular pyramids [13].

For the sake of simplicity, this research assumes binary images. The entire image array is the RAG corresponding to the highest resolution. To build the next level of the RAG pyramid, each node in the RAG of the current level has to determine if each one of its neighbors has the same value (i.e. 0 or 1) as its own. With this class information, stochastic processes are applied to decide if this node will survive and become a node in the next lower-resolution level. Links between survivors and non-survivors (i.e. child-parent links) are then established by connecting the survivors to their non-surviving neighbors in the same class. The survivors also establish new neighborhoods in the new level. Eventually, each connected component will have only one survivor in the highest-level RAG. By starting at an apex of the highest-level RAG and traversing the pyramid through previously established links, we can find all the pixels in the corresponding connected component.

Algorithms for generating RAG pyramids on the CRCW (Concurrent-Read, Concurrent-Write) PRAM (Parallel Random-Access Machine) and implementations on the Connection Machine CM-2 message-passing hypercube supercomputer have been presented in [14]. In the rest of this chapter, we present results for the creation of RAG pyramids on our TurboNet system.

6.2 Algorithm to Generate RAG Pyramids on TurboNet

In this section, the algorithm for generating RAG pyramids on TurboNet is described. There are 5 phases in this algorithm. Let us assume that a node c_0 in level l has r neighbors ($r = 8$ in level 0, assuming 8-connectedness) labeled by c_i, where $i = 1, ..., r$. c_0 is associated with two binary state variables, q_0 and p_0, as well as with one random variable x_0, uniformly distributed between [0, 1].

In *Phase* 1, we assign to each c_i of c_0 a binary state variable λ_i, for $i = 1, ..., r$, according to the following:

$$\lambda_i = \begin{cases} 1, & \text{if } c_i \text{ belongs to the same class with } c_0 \\ 0, & \text{otherwise.} \end{cases} \tag{6.1}$$

The stochastic process mentioned in Section 6.1 is applied iteratively until no more changes take place. Every iteration has two phases, called *Phase* 2 and *Phase* 3. If c_0 survives at the end of the k^{th} iteration, its $p_0(k)$ value will be 1, otherwise it will be 0. In the beginning of each stochastic process, $p_i(0) = 0$, $\forall i = 0, ..., r$.

In *Phase* 2, $q_0(k)$ in the k^{th} iteration is updated as follows:

$$q_0(k) = \begin{cases} 1, & \text{if } \lambda_i p_i(k-1) = 0 \quad \forall i = 0, \cdots, r \\ 0, & \text{otherwise.} \end{cases} \tag{6.2}$$

This means $q_0(k)$ is 1 if there is no survivor among all neighbors of c_0, including itself, that belong to its own class. If $q_0(k)$ is 1, a random number will be generated according to the uniform distribution and will be stored in $x_0(k)$.

In *Phase* 3, $p_0(k)$ is determined based on the updated $q_i(k)$ and $x_i(k)$:

$$p_0(k) = \begin{cases} 1, & \text{if } q_0(k)x_0(k) = \max(\lambda_i q_i(k) x_i(k)) > 0 \quad \forall i = 0, \cdots, r \\ p_0(k-1), & \text{otherwise.} \end{cases} \tag{6.3}$$

Thus, c_0 becomes a survivor if the outcome of the random variable for c_0 is a local maximum in the similarity graph.

All survivors produced in each level by the stochastic processes satisfy the following two conditions that are required for the RAG pyramids:

- any two survivors in the same level must not be neighbors, and
- each non-survivor has to have at least one survivor among its neighbors in the same level.

The stochastic processes will terminate if all $q_0(k)$'s are 0.

The task of *Phase* 4 is to establish the child-parent links between survivors (parents) in the new level and non-survivors (children) in the preceding level. These links are actually a subset of the edges in the RAG of the preceding level. A survivor c_0 looks at each one of its non-surviving neighbors $n1c_i$, for $i = 1, ..., r$, to see if $n1c_i$ has any surviving neighbor, other than c_0 (a non-survivor may have more than one neighboring survivor as long as they are not neighbors themselves). Let the neighbors of $n1c_i$ be $n2c_j$, for $j = 1, ..., r'$. If one or more $n2c_j$'s are found to be survivors, their $x_0(k)$ values will be compared with the one of c_0, and $n1c_i$ will become the child of c_0 or one of the $n2c_j$'s, whichever has the largest random value of $x_0(k)$ drawn when it was chosen as a survivor. If no survivor is found from the $n2c_j$'s, c_0 will simply connect to $n1c_i$ as its parent. For example, ignoring the solid lines in Figure 6.1, we use dotted lines to represent connections between neighbors (represented by circles). Since the survivors (dark circles) c_0 and $n2c_1$ have $x_0(k)$ values of 0.3387 and 0.1956, respectively, $n1c_1$ becomes the child of c_0. There is a child flag for each non-survivor which is set when it is chosen as a child of its parent. Hence, by checking the child flag, c_0 does not have to go through Phase 4 for $n1c_i$ if it already has a parent.

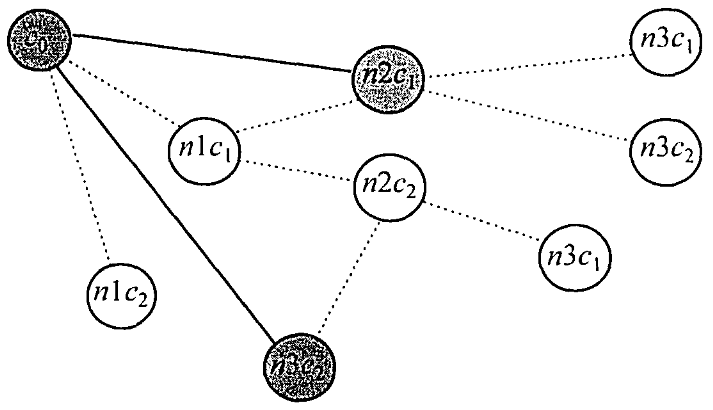


Figure 6.1 An example for *Phases* 4 and 5.

In *Phase* 5, all survivors in the new level establish connections to create the new RAG. Two survivors are neighbors in the RAG of the new level if they were not more than three edges away in the preceding level. It is implemented as follows. A survivor c_0 looks into the $n2c_j$'s of each one of its $n1c_i$'s (it must not be the survivor c_0 itself). If an $n2c_j$ is also a survivor, then survivor c_0 takes it as a neighbor and moves to the next $n2c_j$. If not, c_0 goes one step further into the neighbors of $n2c_j$, represented by $n3c_l$, for $l = 1, \ldots, r''$. If an $n3c_l$ is a survivor, it becomes a neighbor of c_0, otherwise c_0 goes to the next $n3c_l$ until $l = r''$, the number of neighbors of the current $n2c_j$. As an example, let us look at Figure 6.1. Because $n2c_1$ is a survivor, c_0 connects to $n2c_1$ (this connection is represented by a solid line). Then, since $n2c_2$ is not a survivor, c_0 goes into its neighbors and finds $n3c_2$ as a new neighbor. By setting up connections between these survivors, the construction of the RAG is completed, and a new iteration of the algorithm will start with *Phase* 1 assuming this RAG as the input.

A survivor becomes an apex root in the hierarchical structure when it can no longer find any neighbor. The algorithm quits when all survivors are roots.

6.3 Implementation on TurboNet

At the beginning of the algorithm, an input image (or matrix) is divided into P sections as shown in Figure 6.2, where P is the number of processors used. Each section has N^2/P pixels and is assigned to one processor, where NxN is the size of the square matrix. An optimal mapping of the ring of image sections to the TurboNet's hypercube structure is carried out using the binary reflected Gray code. Therefore, neighboring sections are assigned to neighboring processors in the hypercube [16]. Before *Phase* 1 for level 0 can start, each pixel in the input image has to find all its 8-connected neighbors by recording their addresses in a list. This takes $O(N^2/P)$ time.

In *Phase* 1, each processor finds the λ_i's for all neighbors of its pixels c_0 for the class similarity criterion. This time is proportional to

$$t_1(N) = r(N^2/P)(t_{\text{cal}} + \alpha_0 t_{\text{com}}) = O(r\,N^2/P) \qquad (6.4)$$

where r is the number of neighbors of c_0, α_0 is the number of neighbors of c_0 outside of the current processor, and t_{cal} and t_{com} are times for one calculation and one communication between two processors, respectively. In *Phases* 2 and 3, the stochastic processes select survivors. Again if the neighbor c_i of c_0 is not within the current processor, communication between the respective two processors is needed for the values of q_i, p_i and x_i. Both phases have similar asymptotic time complexities as show below

$$t_2(N) = r(N^2/P)(2\,t_{\text{cal}} + 3\alpha_0 t_{\text{com}}) = O(r\,N^2/P) \tag{6.5}$$

and

$$t_3(N) = r(N^2/P)(4t_{\text{cal}} + 3\alpha_0\,t_{\text{com}}) = O(r\,N^2/P) \tag{6.6}$$

In *Phase* 4, if $n1c_i$ is not in the same processor as c_0, then the addresses, the p_j's and x_j's of the $n2c_j$'s have to be transferred. Otherwise, only the values of the p_j's and x_j's for those $n2c_j$'s that are not in the current processor are needed from other processors. Its time complexity is

$$\begin{aligned} t_4(N) &= (N^2/P - d_0)(\alpha_i\,3t_{\text{com}} + r\,t_{\text{cal}} + d_i(\alpha_j\,2t_{\text{com}} + r'\,t_{\text{cal}} + s_j\,t_{\text{cal}})) \\ &= O(r^2\,N^2/P) \end{aligned} \tag{6.7}$$

assuming that r >> (r', d_i and s_j) and N^2/P >> d_0, where d_i is the number of non-survivors among the $n1c_i$'s, s_j is the number of survivors among the $n2c_j$'s, and α_i and α_j are the numbers of pixels that are not in the current processor in d_i and s_j, respectively. Also, d_0 is the total number of non-survivors of the original N^2/P. The time complexity of *Phase* 5 is similar to that of *Phase* 4 except for one additional step; c_0 may look into the $n3c_l$'s if $n2c_j$ is not a survivor. The time consumed is

$$\begin{aligned} t_5(N) &= (N^2/P - d_0)(\alpha_0\,t_{\text{com}} + r(r'\,t_{\text{cal}} + d_j(\alpha_{0l}\,t_{\text{com}} + r''\,t_{\text{cal}}))) \\ &= O(r^3\,N^2/P) \end{aligned} \tag{6.8}$$

also assuming that r >> (r', r'' and d_j) and N^2/P >> d_0, where d_j is the number of non-survivors among the $n2c_j$'s, and α_{0l} is the number of $n3c_l$'s outside of the current processor.

Since the total number of levels in the hierarchy is $O(\log_2 N)$, Phases 1, 4 and 5 are performed $O(\log_2 N)$ times, while Phases 2 and 3 are performed $O((\log_2 N)^2)$ times. Therefore, the running time of the entire algorithm is

$$\begin{aligned} t(N) &= O(\log_2 N(t_1(N) + t_4(N) + t_5(N)) + (\log_2 N)^2(t_2(N) + t_3(N))) \\ &= O(r(N^2/P)\log_2 N(r^2 + \log_2 N)) \end{aligned} \tag{6.9}$$

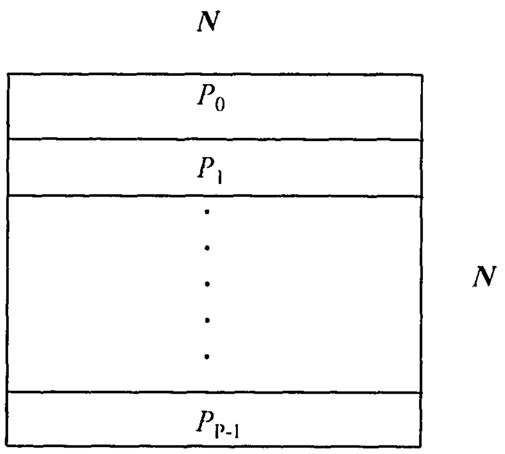


Figure 6.2 Assignment of data for an *N*x*N* input image.

We have developed message-passing, shared-memory and hybrid versions of the algorithm. In the message-passing version, all communications are done through communication channels. If there is no direct link between the source and destination processors, a communication package has to go through intermediate processor(s). When communicating between boards, the source first sends the package to the processor in the other board that is directly linked to it. If this processor is not the destination of the package, it will pass it to another processor. For example, in Figure 2.2 from Chapter 2, if a2 wants to transmit a package to b3, it will send it to b1, and b1 will transfer it to b2 (or b4). b2 (or b4) then passes it to b3. A communication package is 32 bits long and is divided into four fields, as depicted in Figure 6.3. In the diagram, D, S and FC stand for destination, source and function code, and they are 3, 3 and 4 bits long, respectively. DATA, representing either p_i, x_i, $n1c_i$ or $n3c_l$ depending on the function code, takes 16 bits. When DATA is just one item of a group of data, the 6-bit INDEX gives its order. For example, when a source requests the neighbors of $n1c_i$, the destination will reply by sending the $n2c_j$'s one-by-one with their indices. Because x_i is a 32-bit floating point number, it can only be sent in two parts with 16 bits each and INDEX is also used here. Table 6.1 shows all function codes.

DATA	**INDEX**	**FC**	**S**	**D**

Figure 6.3 A 32-bit long communication package.

Table 6.1 Function codes for communication operations.

FC	**Request**	**Reply**
x	0	1
p	2	3
$n1c$	4	5
$n3c$	6	7

When using only shared-memory for communication, each processor copies all values of p_i, q_i, x_i and the neighbor addresses of all its pixels to the shared memory using DMA controllers. This happens only at the end of *Phases* 2, 3 and 5.

The major advantage of TurboNet is that it has both message-passing and shared-memory capabilities. This allows users to choose between communication methods based on their particular tasks, and hence benefit form both methods at the same time. In our hybrid version of the algorithm, each processor has the ability to decide whether to use a communication link or the shared memory when communicating with others by observing the following rules:

- if the source and destination processors are neighbors in the system, i.e. in the 3-dimensional hypercube, communication is done via communication links;
- otherwise, data is sent through the shared memory.

Because in this system, a processor cannot interrupt another processor unless it sends a message to a communication port of the latter processor, all requests for communication are delivered via communication links in all of our implementations. Hence, the processors can choose what communication method to use only when replying to requests. As mentioned earlier in this section, the values of x_i have to be split and sent in two parts. In order to reduce the communication overhead and simplify the programming task, in the hybrid version of the algorithm the x_i's are always transmitted through the shared memory as in the shared-memory implementation.

All processors have to be synchronized at the end of *Phases* 2, 3 and 5 before continuing. This is done by using the shared memory. If a processor finishes its job earlier, it may not quit until all others are also done, because it has to stay "active" to respond to requests by other processors directed to it.

6.4 Performance Results

All three versions of the algorithm have been implemented on TurboNet. The performance results are presented in this section. All running times are expressed in seconds. Four types of binary images, named a, b, c and d, were used to test the algorithm. The image sizes were 4x4, 8x8, 16x16, 32x32, 64x64 and 128x128. The image types are depicted in Figure 6.4. The execution times for one processor and the numbers of pyramid levels are given in Table 6.2. Because we have obtained the same numbers of pyramid levels for 2, 4 and 8 processors as those shown in Table 6.2, these numbers are not repeated in the following tables.

In Tables 6.3 to 6.5, execution times for the three versions of the algorithm with two processors are shown. We can see that the shared-memory version has the best performance over the other two, especially as the image size increases, and the hybrid one comes second. As detailed in Section 6.3, to send a piece of data a processor has to construct a header and then merge it with the data. This includes at least 7 logical operations and doubles the amount of time for just sending and receiving a piece of data through communication ports. It is also longer than just reading from or writing to the shared memory the same piece of data when there is little or no collision on the ISB bus connecting the processors to the memory. The second reason for which the message-passing version is slower than the others is the limited amount of memory that forces us to discard data after use, instead of keeping them in the memory. This implies that two or more communications for the same piece of data may be needed. When only two processors are used, two neighboring nodes in the hypercube are always chosen, hence in the hybrid version communications are almost the same as those in the message-passing one, except for transferring the x_i's through the shared memory. This means that the hybrid version will have similar performance with the message-passing version. Also, when $P = 2$, the possibility of bus collisions is low in the shared-memory version. All these factors make the shared-memory version a winner over the others.

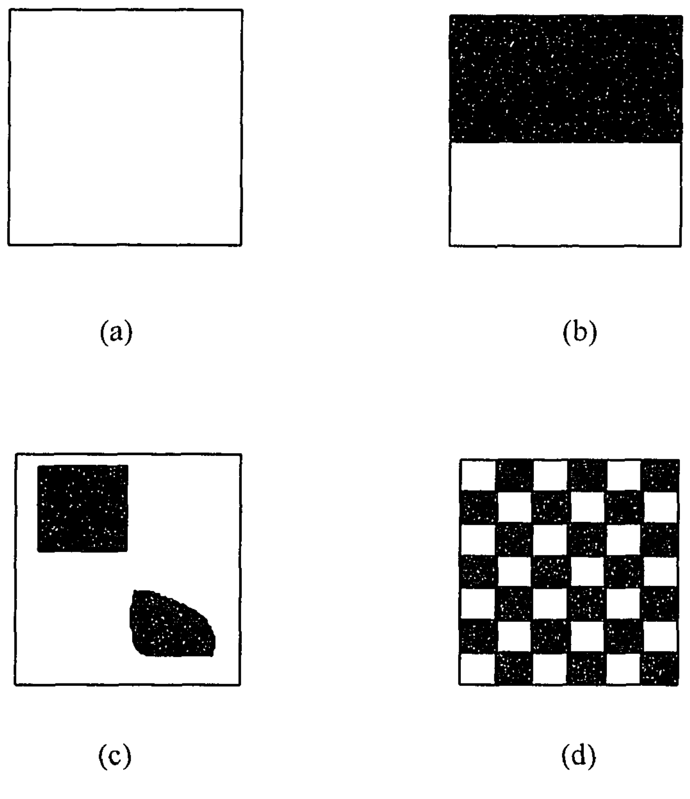


Figure 6.4 Four types of images used in testing: (a) Uniform, (b) Horizontal half, (c) Three objects, including the background, and (d) Checkerboard.

Table 6.2 Execution time and numbers of pyramid levels for a single processor.

	a	b	c	d
4x4	0.002417	0.001267	0.002415	0.002416
	2	2	2	2
8x8	0.015711	0.012314	0.009935	0.010288
	3	3	3	3
16x16	0.075226	0.067247	0.059171	0.031162
	4	4	4	4
32x32	0.349460	0.329755	0.310074	0.194625
	5	5	5	5
64x64	1.472487	1.428048	1.351442	1.098416
	6	6	6	6
128x128	5.919033	5.888514	5.752992	5.087174
	6	6	6	6

Let us look at the results from the viewpoint of image type. In an image, due to the nature of connected components, pixels on edges have fewer neighbors than non-boundary pixels, i.e. r (or r', or r'') is small. This is why the execution times for image d, a checkerboard image, are the smallest among the results. Due to the way an image is divided for the algorithm, image b requires no communication between processors when $P = 2$ (or less communication than other images when $P = 4$ or 8). Hence, the performance for image b is better than the one for image c as the size of the image increases, although the latter might have a large number of edges than the former (this is not true for a single processor since no communication is involved in that situation). This indicates that there are two major factors having influence on the performance: the number of communication operations and the value of r. Generally, the bigger the value of r, the higher the frequency of communication operations unless a similar situation as that for image b with $P = 2$ happens.

As explained in the last paragraph, one of the major problems with the performance of the algorithm is the necessity for frequent communications in the general case. All implementation results in Tables 6.3 to 6.5 are obtained with *Phases* 4 and 5 merged due to the fact that both phases search the $n1c_i$'s and their neighbors in a similar way. As a comparison, the execution times for the hybrid version with separation of *Phases* 4 and 5 is presented in Table 6.6. Improvement as high as 9 percent is gained from this merging of phases.

Table 6.3 Execution time for the message-passing version with $P = 2$.

Image size	a	b	c	d
4x4	0.004822	0.001088	0.004898	0.004901
8x8	0.020987	0.008731	0.012005	0.011019
16x16	0.075012	0.047472	0.051081	0.035916
32x32	0.278019	0.224320	0.240187	0.159799
64x64	1.150380	0.952072	1.023819	0.831675
128x128	4.413166	4.068171	4.241350	3.659837

Table 6.4 Execution time for the shared-memory version with $P = 2$.

Image size	a	b	c	d
4x4	0.004820	0.001201	0.004909	0.004880
8x8	0.020717	0.008967	0.012413	0.010863
16x16	0.074750	0.046817	0.050675	0.033301
32x32	0.261943	0.218780	0.233138	0.147998
64x64	1.099478	0.939378	0.973961	0.777961
128x128	4.183290	3.918180	4.107810	3.496321

Table 6.5 Execution time for the hybrid version with $P = 2$.

Image size	a	b	c	d
4x4	0.004838	0.001111	0.004839	0.004851
8x8	0.020617	0.008697	0.012394	0.011036
16x16	0.075134	0.046943	0.050850	0.034060
32x32	0.276149	0.216675	0.235188	0.150298
64x64	1.110316	0.951002	1.005726	0.786590
128x128	4.363920	4.061670	4.149989	3.556343

Table 6.6 Execution time for the hybrid version with $P = 2$ and separate *Phases* 4 and 5.

Image size	a	b	c	d
4x4	0.005547	0.001226	0.005503	0.005547
8x8	0.022122	0.009569	0.013490	0.012660
16x16	0.081270	0.050118	0.055276	0.037902
32x32	0.297830	0.232514	0.253200	0.168105
64x64	1.187579	1.048526	1.077222	0.883491
128x128	4.013880	4.326028	4.420089	3.891362

In the 4-processor case, bus collisions incurred from accessing the shared memory increase, and communication channel congestion may also happen in the message-passing version due to non-local communications (e.g. when a1 sends a group of data to a3 via a4, and a4 sends a group of data to a2 via a3 almost at the same time, congestion could result in a3). However, the hybrid version can ease the channel congestion by using the shared memory for non-local communications when reducing the number of bus collisions by moving local communications from the shared memory to the links. From Tables 6.7 to 6.9, we can see that the hybrid version has better performance than the other two, and the earlier discussion about the performance for $P = 2$ resulting from the image types still holds as depicted in Figure 6.5. Again the results for the hybrid version with separation of *Phases* 4 and 5 are shown in Table 6.10.

When $P = 8$, similarly to the 4-processor case the ISB bus and the communication links become more overloaded, and the hybrid version again has better performance than the other two versions of the algorithm. Results for $P = 8$ are displayed in Tables 6.11 to 6.13. The performance differences of the three versions for image d are visualized by the charts in Figures 6.6, 6.7 and 6.8 with respect to different numbers of processors.

Better performance improvement should be expected for the hybrid version running on a system with more than eight processors. Improvement of the rules that determine the employment of shared-memory or message-passing in the hybrid algorithm could definitely enhance the performance further. This task is an objective for future research.

Table 6.7 Execution time for the message-passing version with $P = 4$.

Image size	a	b	c	d
4x4	0.007121	0.003567	0.007577	0.006661
8x8	0.032992	0.021199	0.021617	0.021018
16x16	0.116961	0.078435	0.088001	0.038952
32x32	0.309354	0.246113	0.276843	0.137059
64x64	1.029711	0.886206	0.912341	0.646127
128x128	3.594540	3.397266	3.465665	2.826207

Table 6.8 Execution time for the shared-memory version with $P = 4$.

Image size	a	b	c	d
4x4	0.007301	0.003532	0.007436	0.007412
8x8	0.033181	0.020987	0.020871	0.020140
16x16	0.116385	0.077981	0.085413	0.036887
32x32	0.301351	0.240361	0.256259	0.131873
64x64	0.998678	0.873288	0.897125	0.610231
128x128	3.439710	3.265150	3.356454	2.705943

Table 6.9 Execution time for the hybrid version with $P = 4$.

Image size	a	b	c	d
4x4	0.007041	0.003423	0.007335	0.007338
8x8	0.032782	0.020600	0.020287	0.020163
16x16	0.115116	0.076685	0.081354	0.036087
32x32	0.297675	0.236533	0.249481	0.126750
64x64	0.969237	0.813678	0.882079	0.534873
128x128	3.314809	3.024618	3.099592	2.505213

Table 6.10 Execution time for the hybrid version with $P = 4$ and separate Phases 4 and 5.

Image size	a	b	c	d
4x4	0.008185	0.004167	0.008260	0.008261
8x8	0.036674	0.022218	0.023210	0.021983
16x16	0.124501	0.081513	0.089198	0.041297
32x32	0.321667	0.253449	0.271251	0.141354
64x64	0.986330	0.982225	0.949851	0.583320
128x128	3.523334	3.203883	3.302786	3.099013

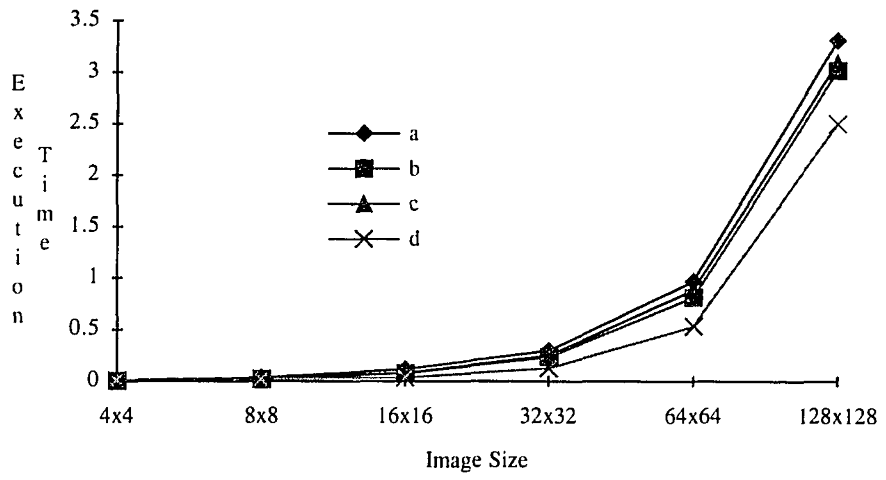


Figure 6.5 Execution time vs. image types for the hybrid version with $P = 4$.

Table 6.11 Execution time for the message-passing version with $P = 8$.

Image size	a	b	c	d
8x8	0.052370	0.031772	0.039740	0.041152
16x16	0.188065	0.130690	0.155713	0.043280
32x32	0.349411	0.272126	0.322993	0.118673
64x64	0.931953	0.828863	0.819055	0.506182
128x128	2.930214	2.839261	2.876496	2.192747

Table 6.12 Execution time for the shared-memory version with $P = 8$.

Image size	a	b	c	d
8x8	0.056110	0.049703	0.035482	0.038130
16x16	0.179111	0.130550	0.147927	0.042110
32x32	0.349392	0.267737	0.287105	0.117954
64x64	0.920304	0.819539	0.829105	0.481761
128x128	2.845688	2.720958	2.765861	2.119655

Table 6.13 Execution time for the hybrid version with $P = 8$.

Image size	a	b	c	d
8x8	0.052370	0.049730	0.034258	0.038103
16x16	0.179109	0.125714	0.095437	0.039445
32x32	0.326547	0.261412	0.269629	0.106352
64x64	0.846256	0.704534	0.776690	0.367363
128x128	2.551307	2.278012	2.329146	1.778732

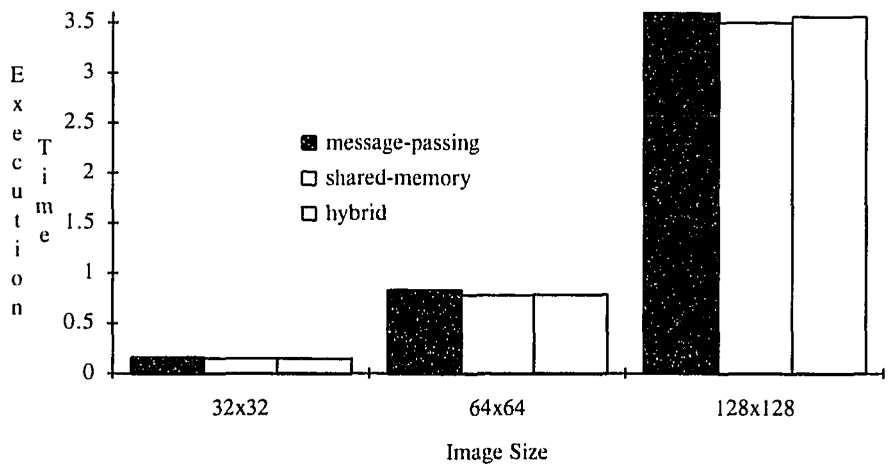


Figure 6.6 Performance comparison of the three versions with image d and $P = 2$.

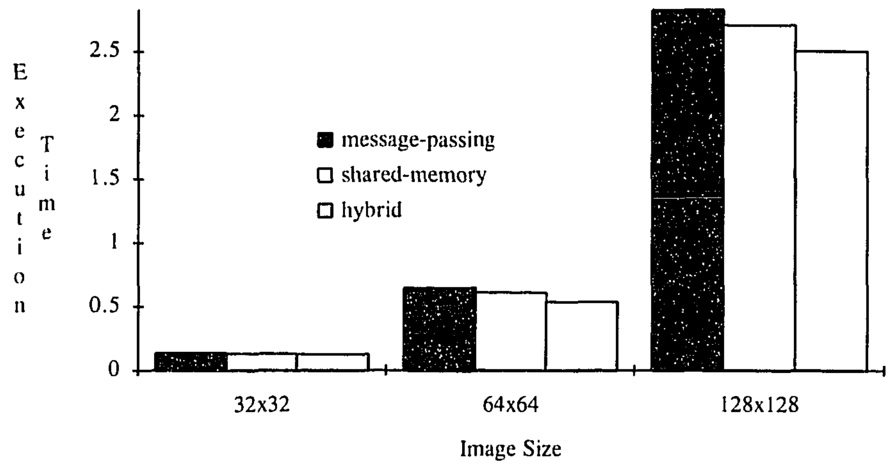


Figure 6.7 Performance comparison of the three versions with image d and $P = 4$.

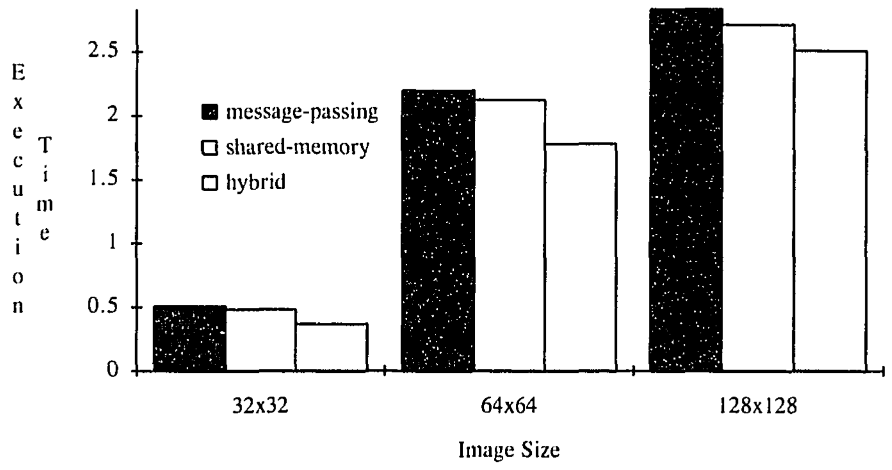


Figure 6.8 Performance comparison of the three versions with image d and $P = 8$.

CHAPTER 7

CONCLUSIONS

In this dissertation, several DSP algorithms were developed for a new experimental parallel system, namely TurboNet. The hybrid architecture of the TurboNet system facilitates direct implementation of both the message-passing and shared-memory fundamental parallel-processing communication paradigms. In contrast, other existing systems support directly in hardware only one of these two fundamental communication paradigms. The main objective of this dissertation was to show that hybrid architectures that implement directly both the message-passing and shared-memory paradigms introduce flexibility in algorithm development that often results in very good performance. Three versions of each algorithm were developed, if possible, that employ message-passing, shared-memory and hybrid communications, respectively. Theoretical and experimental comparisons of these algorithms were also included. The theoretical evaluation of algorithms was based on the framework of a methodology developed here for good performance prediction. The success of this methodology proves that highly accurate performance prediction is possible for complex systems, such as TurboNet, if enough information is provided by data dependence graphs. This methodology can also be applied in the algorithm design phase to assess the performance before program coding.

The theoretical and experimental results show that the hybrid versions of the algorithms generally have better performance than versions that employ only one of the two fundamental communication paradigms. Therefore, not only do hybrid systems of this type lead to significant performance gains for applications that can take advantage of the hybrid architecture, but these systems also support directly each fundamental paradigm for applications with communication bias toward either one of the two paradigms.

In general, the shared-memory paradigm is good for coarse-grain parallelism, while message-passing is more efficient for communicating small amounts of data within groups of nearby processors. To be able to use both paradigms for very high performance, an application should employ a mixture of local and/or remote communications with simultaneous presence of coarse-grain and fine-grain parallelism. The main conclusion of this research is that small-scale and medium-scale parallel computers should implement directly in hardware, if possible, both communication mechanisms for high performance, robustness and ease of algorithm development.

The current TurboNet system has eight processors. However, its architecture supports straightforward system scalability for up to sixty-four processors. In general, as the system size increases, the communication overhead for the system also increases. This is because the bandwidths of communication channels and shared memories are limited. However, this effect of system size is more preeminent in the implementation of the shared-memory paradigm. Further research is needed to find the maximum system size for which the hybrid architecture is still superior. To achieve this objective, both theoretical and experimental results must be produced.

APPENDIX A

DSP ALGORITHMS

1-D Convolution Algorithm 1:

```
(1)    Assume that the input data of f and g are stored in the shared memory of the
       system. Each processor gets the values of f through its DMA, and its g values as
       shown below, where i is the ID number of a processor with $0 \le i \le P-1$:
       m = M/P;
       $t_1$ = im;
       $t_2$ = (i+1)m-1;
       for j = 0 to N+m-2 do in parallel
           if $t_2$-j $\ge$ 0
               $g_i$[j] = g[$t_2$-j];
           else
               $g_i$[j] = g[M+$t_2$-j];
           end if
           if j $\le$N-1
               $f_i$ [j] = f[j];
           end if
       end for
(2)    for j = $t_1$ to $t_2$ do in parallel
           y[j-$t_1$] = 0;
           if j $\le$ N-1
               T = j;
           else
               T = N-1;
           end if
           for k = $t_2$-j to $t_2$-j+T do in parallel
               y[j-t1] = y[j -$t_1$]+$g_i$[k]f[k-$t_2$+j];
           end for
       end for
(3)    This step is for processor that has 2m elements of y[t].
       for n = 0 to (N-1)/m do in parallel
           if i = n
               for j = $t_1$+M to $t_2$+M-1 do
                   yy[j-$t_1$-M] = 0;
                   if j $\le$ N+M=2
                       for k = $t_2$-1  to $t_2$ +N+M-1-j do
                           yy[j-$t_1$-M] = yy[j-$t_1$-M]+$g_i$[k]f[j-M+k-$t_2$];
```

```
                    end for
                else
                    return;
                end if
                return;
            end for
        end if
    end for
```

1-D Convolution Algorithm 2:

(1) Assume that the input data are stored in the shared memory of the system. Each processor gets its f and g values through its DMA as follows:

```
    m = (N+M-1)/P;
    $t_1$ = im;
    $t_2$ = (i+1)m-1;
    $t_3$ = $t_1$-(N-1);
    for all the processors do in parallel
        if $t_1$ < (N-1) or $t_2$ > (M-1)
            if $t_1$ < (N-1)
                for j = $t_2$ to 0 do
                    $g_i$[j-$t_2$] = g[j];
                    if j-$t_2$ $\leq$ N-1
                        $f_i$[j-$t_2$] = f[j-$t_2$];
                    end if
                end for
            else
                for j = $t_3$ to M-1 do
                    $g_i$[j-$t_3$] = g[j]
                    if j-$t_3$ $\leq$ N-1
                        $f_i$[j-$t_3$] = f[N-1-j+$t_3$];
                end if
                end for
            end if
        else
            for j = 0 to N+m-2 do
                $g_i$[j] = g[$t_2$-j];
                if j $\leq$ N-1
                    $f_i$[j] = f[j];
                end if
            end for
        end if
(2) for j = $t_1$ to $t_2$ do in parallel
        y[j-$t_1$] = 0;
        if j ≥ (M/m)m
            if j ≥ M
                T = M+N-2-j;
            else
                T = N-1;
            end if
            for k = j-t1 to j-t1+T do
                y[j-t1] = y[j-t1]+g_i[k]f_i[k-j+t1];
            end for
        else
            if j ≤ N-1
                T = j;
            else
                T = N-1;
            end if
            for k = t2-j to t2-j+T do
                y[j-t1] = y[j-t1]+g_i[k]f_i[k+j-t2];
            end for
        end if
    end for
```

2-D Convolution Algorithm (binary-tree):

```
(1)     For i = 0 to P-1 do in parallel:
            read the g matrix;
            for j = i×( N_1/P) to (i+1) ×( N_1/P)-1 do
                read f[j,y];
            end for
        end for
(2)     For i = 0 to P-1 do in parallel:
            perform the convolution of f and g;
        end for
(3)     For s = 1 to $\log_2$ P
            send result to lower neighbor or receiver result from upper neighbor;
            if ( ID>(P-v1))
                break;
            end if
          end for
```

2-D Convolution Algorithm (grid):

```
(1)     Processors in the first row and first column get g and f respectively, from the
        system DRAM and pass them to the other processors accordingly through DMAs
        and communication ports.
(2)     For all the processors do in parallel
                for i = 0 to N2-1 do
                        for j = 0 to M2-1 do
                                c[i+j] = c[i+j]+f[i]g[j];
                        end for
                end for
(3)     if ( you are in the first row or the first column )
                if ( not in the last row or the last column)
                        send your c[] to your south-east neighbor;
                        return;
                else
                        return;
                end if
        end if
        wait until you receive result from your north-west neighbor;
        add the received result to yours;
        if ( you have a south-east neighbor)
                send your c[] to your south-east neighbor;
                return;
        else
                return;
        end if
        return;
```

2-D Convolution Algorithm (triangle):

```
(1)     Processors in the first row, first column and last column get g and f respectively,
        from the system DRAM and pass them to the other processors accordingly
        through DMAs and communication ports.
(2)     For all the processors do in parallel
                for i = 0 to N2-1 do
                        for j = 0 to M2-1 do
                                c[i+j] = c[i+j]+f[i]g[j];
                        end for
                end for
(3)     if ( you are in the diagonal or right above it )
                if ( you are in the diagonal )
                        if ( have lower neighbor )
                send your c[] to it;
                return;
            else
                return;
            end if
        else
            if ( have upper neighbor )
                send your c[] to it;
                return;
            else
                return;
            end if
        end if
end if
wait until you receive result from your upper or lower neighbor;
add the received result to yours;
if ( you have received result from your upper neighbor )
        if ( have lower neighbor )
            send your c[] to it;
            return;
        else
            return;
        end if
else
        if ( have upper neighbor )
            send your c[] to it;
            return;
        else
            return;
        end if
end if
```

REFERENCES

1. S. Horguchi and T. Nakada, "Performance Evaluation of Parallel Fast Fourier Transform on a Multiprocessor Workstation," *Journal of Parallel and Distributed Computing*, pp. 158-163, December 1993 .

2. C. Tong and P. N. Swarztrauber, "Ordered Fast Fourier Transform on a Massively Parallel Hypercube Multiprocessor," *Journal of Parallel and Distributed Computing*, pp. 50-59, December 1991 .

3. E. M. Dowling and Z. Fu, "HARP: An Open Architecture for Parallel Matrix and Signal Processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 1081-1091, October 1993.

4. G. Fox, et. al., *Solving Problems on Concurrent Processors*, vol. 1, Prentice-Hall, Englewood Cliffs, NJ, 1988.

5. S. G. Akl, *The Design and Analysis of Parallel Algorithms*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

6. D. H. Bailey, "FFTs in External or Hierarchical Memory," *Journal of Supercomputing*, vol. 4, No. 1, pp. 23-35, March 1990.

7. Z. Cvetanovic, "Performance Analysis of the FFT Algorithm on a Shared-Memory Parallel Architecture, " *IBM Journal of Research and Development*, vol. 31, No. 4, pp. 245-451, July 1987.

8. R. E. Blahut, *Fast Algorithms for Digital Signal Processing*, Addison-Wesley, Reading, MA, 1987.

9. S. G. Ziavras and M. A. Siddiqui, "Pyramid Mapping onto Hypercubes for Computer Vision: Connection Machine Comparative Study," *Concurrency: Practice and Experience*, Vol. 5, No. 6, pp. 471-489, September 1993.

10. S. Chandra, J. R. Larus and A. Rogers, "What Is Time Spent in Message-Passing and Shared-Memory Programs?" *6th Int'l Conference on Architectural Support for Program Langs. & OS's*, October 4-7, 1994.

11. J. Heinlein, K. Gharachorloo, S. Dresser and A. Gupta, "Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor," *6th Int'l Conference on Architectural Support for Program Langs. & OS's*, October 4-7, 1994.

12. D. Kranz, K. Jonhson, A. Agarwal, J. Kubiatowicz and B. Lim, "Integrating Message-Passing and Shared-Memory: Early Experience," *5th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pp. 54-63, May 1993.

13. A. Montanvert, P. Meer and A. Rosenfeld, "Hierarchical Image Analysis Using Irregular Tessellations," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 13, No. 4, pp. 307-316, 1991.

14. S. G. Ziavras and P. Meer, "Adaptive Multiresolution Structures for Image Processing on Parallel Computers," *Journal of Parallel and Distributed Computing*, Vol. 25, pp. 475-483, 1994.

15. A. Rosenfeld (Ed.), *Multiresolution Image Processing and Analysis*, Springer-Verlag, Berlin, 1984.

16. S. G. Ziavras and D. P. Shah, "High Performance Emulation of Hierarchical Structures on Hypercube Supercomputers," *Concurrency: Practice and Experience*, Vol. 6, No. 2, pp. 85-100, 1994.

17. T. H. Cormen, *Introduction to Algorithms*, McGraw-Hill, New York, NY, pp. 784-796, 1990.

18. R. Sedgewick, *Algorithms*, Addison-Wesley, Reading, MA, pp. 583-592, 1988.

19. J. H. McClellam, *Number Theory in DSP*, Addison-Wesley, Reading, MA, pp.194-196, 1979.

20. K. M. Chandy, *Parallel Program Design: a Foundation*, Addison-Wesley, Reading, MA, pp. 148-151, 1989.

21. A. Y. Grama, " Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures," *IEEE Parallel and Distributed Technology*, August 1993.

22. A. Gupta and V. Kumar, " The Scalability of FFT on Parallel Computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, August 1993.

23. R. Hross, S. G. Ziavras, C. N. Manikopoulos, N. J. Lad and X. Li, "A Defect Identification Algorithm for Sequential and Parallel Computers," *IEEE International Symposium on Industrial Electronics*, Athens, Greece, July 10-14, 1995, to appear.

24. T. G. Lewis and H. El-Rewini, *Introduction to Parallel Computing*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

25. P. A. Lynn and W. Fuerst, *Introduction to Digital Signal Processing with Computer Applications*, John Wiley & Son, Chichester, NY, pp.211-246, 1989.

26. J. S. Lim, *Two-Dimensional Signal and Image Processing*, PTR Prentice-Hall, Englewood Cliffs, NJ, pp.163-172, 1990.

27. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, Addison-Wesley, Reading, MA, 1992.

28. X. Li, S. G. Ziavras and C. N. Manikopoulos, "Parallel DSP Algorithms on TurboNet: An Experimental System with Hybrid Message-Passing and Shared-Memory Architecture," *Concurrency: Practice and Experience*, accepted for publication, 1995.

29. Force Computers Inc, San Jose, CA, *SPARC CPU-2CE Technical Reference Manual*, April 1993.

30. X. Li, S. G. Ziavras and C. N. Manikopoulos, "Parallel Generation of Adaptive Multiresolution Structures for Image Processing," *Concurrency: Practice and Experience*, submitted for publication.

31. Ariel Corporation, Highland Park, NJ, *User's Manual for the V-C40 Hydra*, version 0.60, February, 1994.

32. Texas Instruments, Houston, TX, *TMS320C4x User's Guide*, pp.1.4-1.6, 1993.

33. Texas Instruments, Houston, TX, *TMS320C4x User's Guide*, pp.10.1-10.16, 1993.

34. Texas Instruments, Houston, TX, *TMS320 Floating-Point DSP Optimal C Compiler User's Guide*, 1991

35. Texas Instruments, Houston, TX, *TMS320 Floating-Point DSP Assembly Language Tools User's Guide*, 1991

36. K. Hwang, *Advanced Computer Architecture with Parallel Programming*, Ed. 2. McGraw Hill, Englewood Cliffs, NJ, 1993.

37. G. C. Fox, S. W. Otto and A. J. Hey, "Matrix Algorithms on Hypercube (I): Matrix Multiplication," *Parallel Computing*, pp. 17-31, 1987.

38. S. G. Ziavras, " RH: A Versatile Family of Reduced Hypercube Interconnection Networks," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 11. pp. 1210-1220, November 1994.