New Jersey Institute of Technology
Digital Commons @ NJIT

Theses | Electronic Theses and Dissertations

Fall 1995

Algorithms for the NJIT turbonet parallel computer

Nitin J. Lad
New Jersey Institute of Technology

Follow this and additional works at: https://digitalcommons.njit.edu/theses

Part of the Electrical and Electronics Commons

Recommended Citation
Lad, Nitin J., "Algorithms for the NJIT turbonet parallel computer" (1995). *Theses*. 1172.
https://digitalcommons.njit.edu/theses/1172

This Thesis is brought to you for free and open access by the Electronic Theses and Dissertations at Digital Commons @ NJIT. It has been accepted for inclusion in Theses by an authorized administrator of Digital Commons @ NJIT. For more information, please contact digitalcommons@njit.edu.

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

NJIT
New Jersey's Science & Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

UMI Number: 1378357

Copyright 1995 by
Lad, Nitin J.

All rights reserved.

UMI Microform 1378357
Copyright 1996, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

ALGORITHMS FOR THE NJIT TURBONET PARALLEL COMPUTER

by
Nitin J. Lad

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering

Department of Electrical and Computer Engineering

October 1995

Copyright © 1995 by Nitin J. Lad

ALL RIGHTS RESERVED.

APPROVAL PAGE

PARALLEL ALGORITHMS FOR THE NJIT TURBONET PARALLEL COMPUTER

Nitin J. Lad

Dr. Sotirios Ziavras, Thesis Advisor — Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. John Carpinelli, Committee Member — Date
Associate Professor of Electrical and Computer Engineering and
Director of Computer Engineering, NJIT

Dr. MengChu Zhou, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT

ABSTRACT

ALGORITHMS FOR THE NJIT TURBONET PARALLEL COMPUTER

by
Nitin J. Lad

Element selection for arrays, array merging, and sorting are very frequent operations in many of today's important applications. These operations are of interest to scientific, as well as other applications where high-speed database search, merge, and sort operations are necessary and frequent. Therefore, their efficient implementation on parallel computers should be a worthwhile objective. Parallel algorithms are presented in this thesis for the implementation of these operations on the NJIT TurboNet system, an in-house built experimental parallel computer with TMS320C40 Digital Signal Processors interconnected in a 3-D hypercube structure. The first algorithm considered is selection. It involves finding the k-th smallest element in an unsorted sequence of n elements, where $1 \leq k \leq n$. The second algorithm involves the merging of two sequences sorted in nondecreasing order to form a third sequence, also sorted in nondecreasing order. The third parallel algorithm is sorting. For a given unsorted sequence S of size n, we want to sort the sequence such that $s_i' \leq s_{i+1}'$, for all n elements. Performance results show that the robust structure of TurboNet results in significant speedups.

BIOGRAPHICAL SKETCH

Author: Nitin J. Lad

Degree: Master of Science in Electrical Engineering

Date: October 1995

Undergraduate and Graduate Education:

- Master of Science in Electrical Engineering,
  New Jersey Institute of Technology,
  Newark, NJ, October 1995

- Bachelor of Science in Electrical Engineering,
  New Jersey Institute of Technology,
  Newark, NJ, May 1991

Major: Electrical Engineering

Presentations and Publications:

R. Hross, S. Ziavras, C. 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.

This thesis is dedicated to my family and friends.

ACKNOWLEDGMENT

I would like to thank all the professors I had taken classes with and graduate students I worked with during my study period at NJIT. Special thanks to Dr. Ziavras, Dr. Xi Li, and Ralf Hross for their guidance and advice.

I also want to thank Dr. Carpinelli and Dr. Zhou for serving in my thesis defense committee.

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 1

1.1.1 Shared-Memory Multiprocessors 2

1.1.2 Message-Passing Multiprocessors 3

1.2 Motivation, Objectives, and Contributions 5

1.3 Outline 7

2 THE NJIT TURBONET SYSTEM 8

2.1 System Description 8

2.2 The Host System 10

2.3 Parallel Digital Signal Processor: TI-TMS320C40 11

3 PARALLEL SELECTION 13

3.1 Parallel Selection Algorithm 13

3.2 TurboNet Implementation and Timing Results 19

3.2.1 TurboNet Implementation of Parallel Select 20

4 PARALLEL MERGE 23

4.1 Parallel Merging 23

4.1.1 Sequential Merging 24

4.1.2 Finding the Median of Two Sorted Sequences 26

4.1.3 Parallel Merging on an EREW Computer 28

TABLE OF CONTENTS
(Continued)

Chapter **Page**

4.2 TurboNet Parallel Merge....33

5 PARALLEL SORT....38

5.1 Parallel Sorting....38

5.1.1 Sequential Sorting....38

5.1.2 Parallel Sort Algorithm....40

5.2 TurboNet Parallel Sort....44

6 CONCLUSIONS....51

APPENDIX A PARALLEL SELECTION PROGRAMS....53

APPENDIX B PARALLEL MERGING PROGRAMS....65

APPENDIX C PARALLEL SORTING PROGRAMS....75

REFERENCES....91

LIST OF TABLES

Table **Page**

3.1 Averaged Execution Time(μsec) for Parallel Selection..21

4.1 Execution Time(μsec) for Parallel Merge..37

5.1 Summary of x and k value computations..45

5.2 Execution Time (μsec) for Parallel Sort..50

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 NJIT TurboNet System	9
2.2 The Hydra Board	12
3.1 Finding *19*th element using PARALLEL_SELECT	16
3.2 Finding *19*th element using PARALLEL_SELECT recursively	18
3.3 Second recursive call to PARALLEL_SELECT	18
3.4 Call to SEQUENTIAL_SELECT in step 1	19
3.5 Plot of average execution time (in μsec) versus the number of elements	22
4.1 Two sorted sequences S1 and S2 to be merged	30
4.2 S1 and S2 are divided during the first pass of step 1.2	31
4.3 S1 and S2 are divided among all four processors for merging	32
4.4 Plot of execution time (in μsec) versus the number of elements	36
5.1 Initial unsorted sequece S of size n=40 elements	42
5.2 Dividing the S into four subsequences for sorting by selection	42
5.3 Recursive call to parallel sort for sorting S1 and S2 simultaneously	43
5.4 Recursive call to parallel sort for sorting S3 and S4 simultaneously	43

LIST OF FIGURES
(Continued)

Figure **Page**

5.5 Final sorted sequence S of size n=40 elements (after step 6)....43

5.6 Initial unsorted sequence of size n=40 elements....47

5.7 Dividing the S into four subsequence for sorting by selection....48

5.8 Recursive call to parallel sort by each processor....48

5.9 Final sorted sequence S of size n=40 elements (after step 6)....49

5.10 Plot of execution time (in microseconds) versus the number of elements....50

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, shared-memory and message-passing are two techniques parallel computer systems use for coordination and communication. The shared-memory technique in parallel computing will be the focus of this thesis. This chapter provides introductory background in this fast growing research area. The most important issues in this chapter are the motivation and objectives of our research. An outline of the thesis is also 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 [7]. 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 the processors share a common memory and some peripherals, and communication is performed through the shared-memory. These multiprocessor models are primarily used: the uniform-memory-access (UMA) model and the nonuniform-memory-access (NUMA) model [2]. They differ in the way the memory and other resources are distributed. In the UMA model, as shown in Figure 1.1, all the processors have equal access time to all the memory locations in all the 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 of 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.

The most popular switching networks are the single bus , crossbar, and multistage [6]. 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 ongoing transaction. Hence, the single bus has low cost and low performance while the other two provide high bandwidth with higher cost.

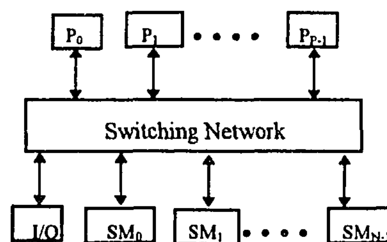

Figure 1.1 The UMA multiprocessor model

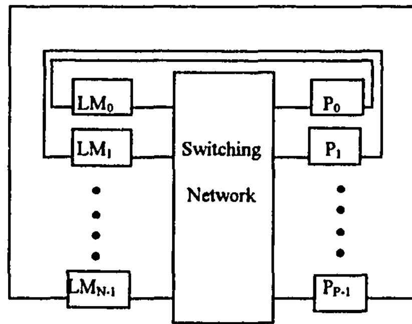

Figure 1.2 The NUMA multiprocessor model

1.1.2 Message-Passing Multicomputers

A message-passing multicomputer [6] 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, the binary tree, star, ring, mesh, hypercube, etc. They are also called static connection networks because all the links between nodes are fixed after a network is built. Among these topologies, the hypercube is one of the most complicated but yet very popular. 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-node for short, has a single node with no communication channel, i.e., 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 connection to the corresponding node in the other 2-cube. Hence, in general, a *d*-cube is constructed by connecting the corresponding nodes of 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 the ith bit of it represents the coordinate of the node in the ith dimension of the hypercube. For a pair of directly connected nodes, their node numbers are different in only one bit. The number of bits that differ between the node numbers of two nodes gives the distance between them.

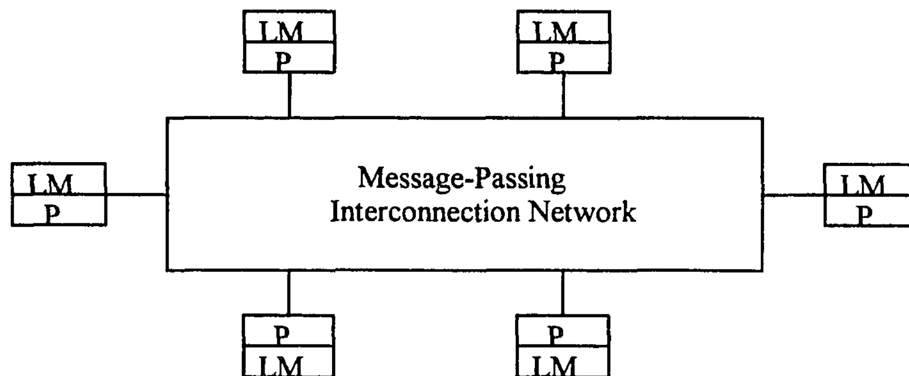

Figure 1.3 A multicomputer

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

data. Additionally, the shared-memory paradigm simplifies the development of algorithms.

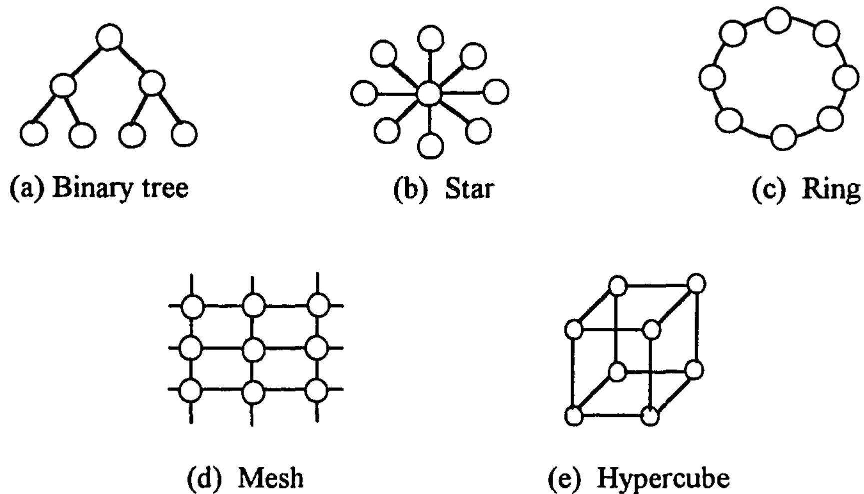

Figure 1.4 Examples of common network topologies

1.2 Motivation, Objectives, and Contributions

Array processing operations, such as finding the k-th smallest element (i.e., the selection problem) or merging two arrays to form a single sorted array, are very frequent in today's applications. For example, in the stock market, the medical and weather prediction fields, and other real-time applications, database operations are frequent. And, the operations need to be completed very quickly. Parallel processing is much better suited to such compute intensive applications. The tasks can be completed in significantly less time because the workload is distributed among all available processors and processors work in parallel, whereas in traditional one-processor systems the processing is done serially, thus

taking much longer, and therefore their implementation on parallel computers should be investigated.

The target system in this thesis is an asynchronous, three-dimensional hypercube system composed of eight powerful Texas Instruments TMS320C40(C40) Digital Signal Processor(DSP) chips. We have built this system with two VME Hydra boards of Ariel Corporation, where each board contains four C40 DSPs and shared-memory, in addition to local memory attached to each C40 DSP. 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 the message-passing and shared-memory communication paradigms, in contrast to other proposed systems such as FLASH and HARP. Details about TurboNet follow in the next chapter.

The objectives of this thesis are: (1) to employ the shared-memory paradigm in the implementation of parallel algorithms for the selection, array merging, and sorting problems; (2) to compare the performance of our parallel algorithms that employ the shared-memory communication paradigm with the performance of sequential algorithms, in order to illustrate the superiority of the former.

Our results prove that several algorithms can take advantage of the shared-memory capability of the hybrid architecture of our target system, TurboNet, in order to achieve significant speedup. Therefore, the main conclusion of this thesis is that the implementation of such operations on a relatively small shared-memory parallel computer is very practical and cost effective if the applications process large amounts of data.

1.3 Outline

This thesis is organized as follows. Following this introduction, Chapter 2 provides a brief review of our target TurboNet system, including a review of the C40, Hydra and TurboNet architectures. In chapters 3, 4, and 5, algorithms for selection, merging, and sorting are discussed, and their implementations on TurboNet are presented along with relevant performance analysis results. Chapter 6 presents the conclusions and further research objectives.

CHAPTER 2

THE NJIT TURBONET SYSTEM

The NJIT TurboNet system is presented in this chapter. The main aspects of the system are discussed here as follows: the entire system, the host system, and the Texas Instrument's TMS320C40 Digital Signal Processor. From the architecture point of view, the TurboNet system implements in hardware both the shared-memory and message-passing paradigms, and this is what distinguishes it from other parallel systems.

2.1 System Description

The TurboNet computer system comprises a SPARC CPU-2CE host system board, a VME backplane, two Ariel-VC40 Hydra Digital Signal Processor(DSP) boards, two hard disk drives, a floppy drive, a CD-ROM, a VME Bus Logic Analyzer, and a set of PC-AT computers as depicted in Figure 2.1. There are four communication links between the two DSP system boards, each link connecting corresponding C40 processors on the two different Hydra boards. Each Texas Instruments TMS320C40 (C40 for brevity) Digital Signal Processor has six bi-directional communication ports. Three of the six communication ports of each processor are being used to interconnect the DSPs within the board to implement a fully-connected four-processor system. A fourth communication port is used for an interboard connection, that is to link with a C40 processor in the second Hydra board so that a 3-D hypercube system with eight processors is formed.

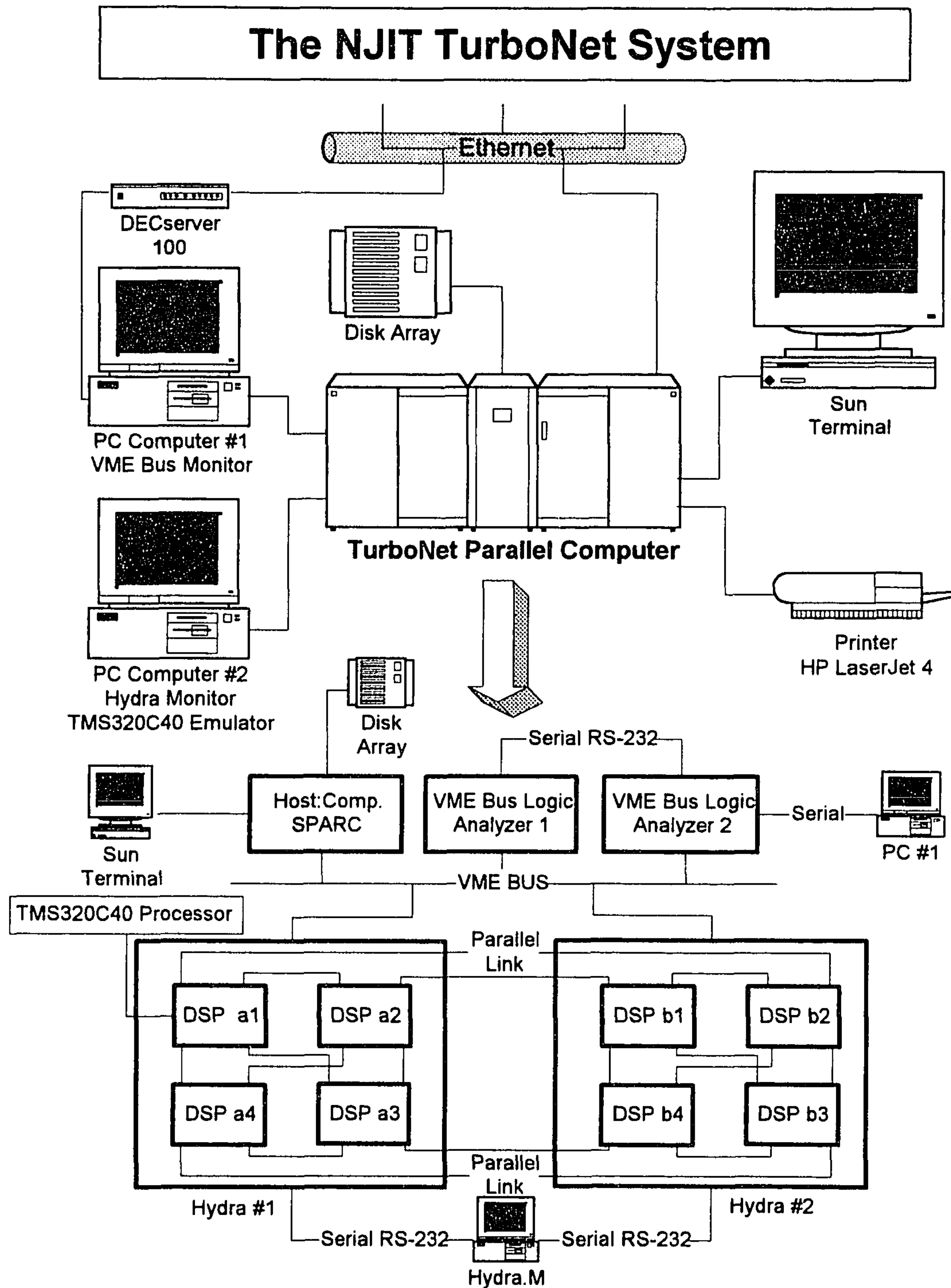

Figure 2.1 The NJIT TurboNet System

Each Hydra board (see Figure 2.2) is a single-slot VME-based multi-digital signal processor system, containing four C40 DSP chips. An Internal Shared Bus (ISB) is included, to which the VME bus and the global bus of each DSP are connected. The ISB provides access to DRAM memory and other shared resources to all four processors within the Hydra board. Because of the Shared-DRAM memory and the hypercube architecture, the TurboNet system implements both the shared-memory and message-passing communication paradigms. This is one of the features that make this system unique.

The TurboNet system is monitored by two PC-AT computers. These units are linked to the VME boards and the Hydra boards. Their purpose is to display, using customized software, the VME status of the system and the Hydra board activity. Any error within the system will alert the programmer for troubleshooting and maintenance purposes.

2.2 The Host System

The host system, a 40 MHz SPARC CPU-2CE board, is a complete VME-based SPARC 2 architecture with Sbus expansion. The SPARC CPU-2CE runs the SunOS/Solaris operating system version 4.1.3. The main processor unit is based on a SPARC 32-bit RISC architecture. It comprises an integrated Integer Unit/Floating Point Unit, a Sun standard SRAM-based memory management unit, a cache controller, and two Cache RAM chips. Operating at 40 MHz, the Integer Unit/Floating Point Unit provides 28.5 MIPS integer performance and 4.2 MFLOPS floating point performance. The purpose of

the host system is to compile and download the C40 programs to the Hydra boards using the VME bus.

All DSPs have the capability of becoming the VME Bus Master as well as the VME System Controller. The Hydra VME interface has a built-in DMA controller that can be set to move data to/from the shared internal DRAM from/to another VME card autonomously. This helps to relieve the DSPs from the task of data movement. The VMEbus analyzers are logic analyzers that are designed specifically to interface and troubleshoot the VME bus. The VME bus testing can be done at the software and hardware levels. Hardware handshaking and timing problems can be traced, analyzed and displayed on an independent terminal without interference to the VME bus.

2.3 Parallel Digital Signal Processor: TMS320C40

The TMS320C40 is a 32-bit processor designed specifically for parallel-processing and other real time embedded applications. It has six communications ports for high-speed interprocessor communications with a 20-Mbyte/sec maximum asynchronous transfer rate and a six-channel DMA coprocessor for concurrent I/O and CPU operation, thereby maximizing sustained CPU performance by alleviating the CPU of burdensome I/O. The high-performance DSP CPU is capable of 275 MOPS and 320 Mbytes/sec. Two identical external data and address buses supporting shared memory systems and high data rate, single-cycle transfer are also designed into the chip. The six communication ports, under DMA coprocessor supervision, allow the CPU to perform other tasks in parallel, utilizing

its computational power to the maximum. This benefits the computational timing and data transfer throughput.

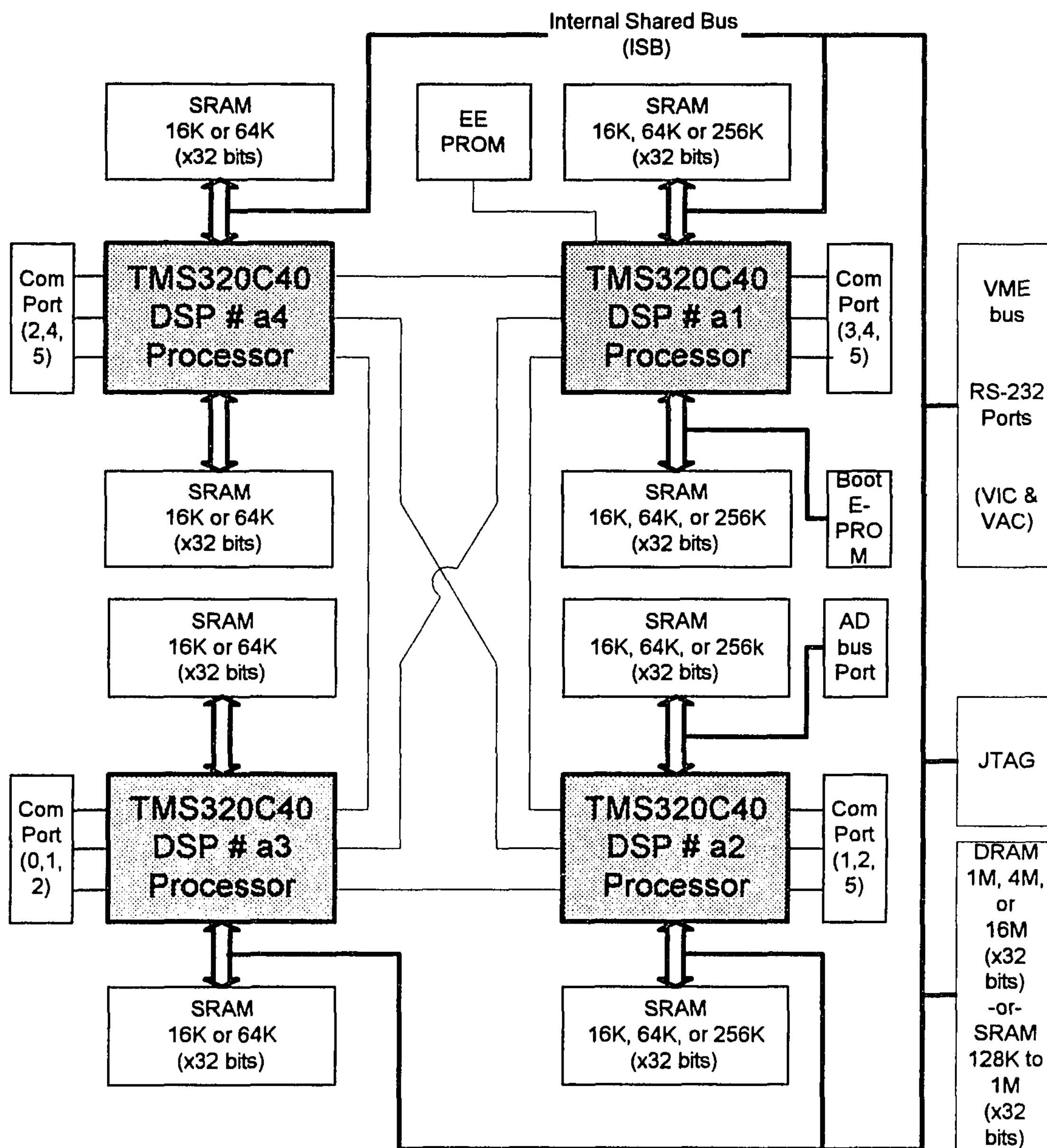

Figure 2.2 The TurboNet's Hydra Board

CHAPTER 3

PARALLEL SELECTION

3.1 Parallel Selection

The selection problem involves finding the k-th smallest element in a given sequence of unsorted elements. That is, given an unsorted sequence S of n elements and an integer k, where $1 \le k \le n$, we want to find the k-th smallest element in S. This problem arises in many commercial applications, especially financial applications where statistical methods are used. The simplest way of finding the k-th element is by first sorting the sequence and then selecting the k-th smallest element. For a sequence of size n, it requires $O(n^2)$ comparisons. The divide-and-conquer approach can solve the problem in $O(n)$ time [1]. An algorithm for the EREW (Exclusive-Read, Exclusive-Write) shared-memory SIMD computer has running time $t(n) = O(n^x)$, where x is computed from $N = n^{1-x}$, and N is the number of processors [1]. The parallel algorithm uses a sequential algorithm for the selection problem in individual processors. This SEQUENTIAL_SELECT algorithm finds the k-th element in $O(n)$ time. Let the constant integer Q have any value greater than or equal to 5, as determined in [1] for the best possible asymptotic performance. The algorithm works as follows [1]:

SEQUENTIAL_SELECT(S, k)

(1) **if** $|S| < Q$ **then** sort S and find the k-th element

else divide S into $|S|/Q$ subsequences of size Q each

end if.

(2) For each subsequence find its median m_i, after its sorting.

(3) Find the median m of these $|S|/Q$ medians by calling SEQUENTIAL_SELECT.

(4) Generate the following sequences that contain elements of S chosen as follows:

S_G: Elements greater than m

S_E: Elements equal to m

S_L: Elements less than m

(5) **if** $|S_L| \geq k$ **then** call SEQUENTIAL_SELECT(S_L, k)

else if $|S_L| + |S_E| \geq k$ **then** return m as the k-th element

else call SEQUENTIAL_SELECT $(S_G, k - |S_L| - |S_E|)$

end if.

end if.

The running time of the SEQUENTIAL_SELECT algorithm was shown in [1] to be $t(n) = O(n)$, which is optimal. A parallel algorithm is cost optimal if the product of the number of processors and the running time has asymptotic complexity equal to the asymptotic running time of the best known sequential algorithm for the problem.

Using parallel processing techniques, the selection task can be done more efficiently. A parallel algorithm for selection on a EREW shared-memory SIMD

computer is introduced as procedure PARALLEL_SELECT [1]. The following assumptions are made. There are N processors in the computer, where $N \geq 1$. M is an array of size N in the shared memory. Each processor knows the size n of the sequence S and computes x from $N = n^{1-x}$, where $0 < x < 1$. Each of the n^{1-x} processors can store a sequence of size n^x in its local memory. Two other procedures are used. Procedure BROADCAST broadcasts information to all processors in $O(\log n^{1-x})$ time. The procedure ALLSUMS finds all prefix sums in the same amount of time [1].

The algorithm is as follows [1]:

procedure PARALLEL_SELECT (S, k)

(1) **if** $|S| \leq 4$ **then** P_1 uses at most five comparisons to return the k-th element

else (i) S is subdivided into $|S|^{1-x}$ subsequences S_i of length $|S|^x$ each, where $1 \leq i \leq |S|^{1-x}$, and

(ii) subsequence S_i is assigned to processor P_i.

end if.

(2) **for** $i = 1$ **to** $|S|^{1-x}$ **do in parallel**

(2.1) {P_i obtains the median m_i, i.e. the $\lceil |S_i| / 2 \rceil$th element, of its associated subsequence}

SEQUENTIAL_SELECT $(S_i, \lceil |S_i| / 2 \rceil)$

(2.2) P_i stores m_i in ***M(i)***

end for.

(3) {The procedure is called recursively to obtain the median m of M}

$$\text{PARALLEL_SELECT } (M, \lceil |M|/2 \rceil).$$

(4) The sequence S is subdivided into the three subsequences:

$S_L = \{s_i \in S : s_i < m\}$, $S_E = \{s_i \in S : s_i = m\}$, and $S_G = \{s_i \in S : s_i > m\}$.

(5) **if** $|S_L| \geq k$ **then** PARALLEL_SELECT (S_L, k)

else if $|S_L| + |S_E| \geq k$ then return m

else PARALLEL_SELECT $(S_G, k - |S_L| - |S_E|)$

end if

end if.

The running time of the algorithm is $t(n) = O(n^x)$ for $n > 4$. An example that follows illustrates the workings of the PARALLEL_SELECT algorithm.

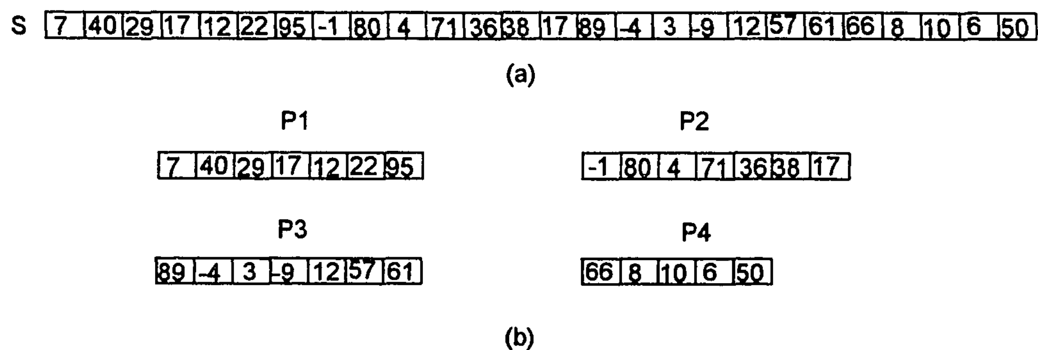

Figure 3.1 Finding the *19*th element using PARALLEL_SELECT

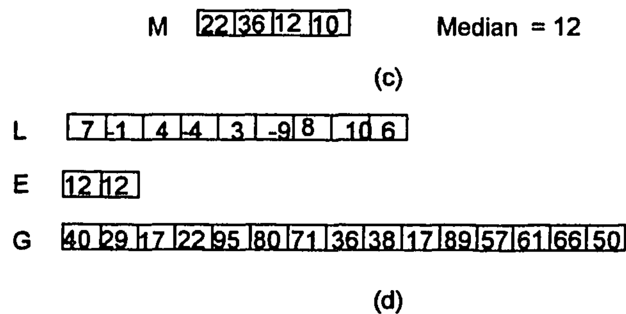

Figure 3.1 Finding the *19*th element using PARALLEL_SELECT (continued)

We want to find the *19*th element of an unsorted sequence S (in Figure 3.1(a)) of size 26 using a FOUR processor EREW SM parallel computer. Initially, we have $k = 19$, $|S| = n = 26$, and $N = 4$. Therefore, the value of x is computed as $x = 1 - (\log N / \log n) = 0.5745$ and $1 - x = 0.4255$. The size of the sequence S_i received by each of the n^{1-x} processors is $\lceil |S|^x \rceil = 7$. After step 1, the first three processors have seven elements each, and the fourth has five, as in Figure 3.1(b). Next, each processor calls SEQUENTIAL_SELECT to find the median of the subsequence it has and places the median m_i in the array M (Figure 3.1(c)). In step 3, PARALLEL_SELECT is called recursively to find the median of medians, which is 12. In step 4, S is divided into three subsequences (namely S_L, S_E, and S_G), as in Figure 3.1(d). Since $|S_L| = 9$ and $|S_E| = 2$, we have $|S_L| + |S_E| < k$ and PARALLEL_SELECT is called recursively in step 5. At this point, the input to PARALLEL_SELECT is a new sequence S (Figure 3.2(a)) and $k = 8$. Since $\lfloor |S|^{1-x} \rfloor$, the sequence is subdivided into $\lfloor |15|^{1-.5745} \rfloor = 3$ subsequences S_i of length

$\lceil |S|^x \rceil = \lceil |15|^{.5745} \rceil = 5$. Each of the three processors receives five elements as in Figure 3.2(b).

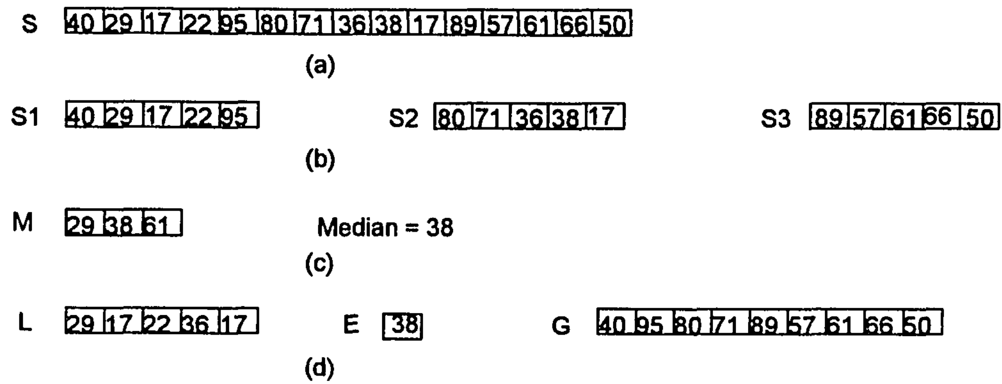

Figure 3.2 Finding the *19th* element using PARALLEL_SELECT recursively

In step 2, all three processors find medians and place them in the shared-memory array M. In step 3, PARALLEL_SELECT is called and processor 1 finds the median of medians as in Figure 3.2(c). In step 4, S is divided into three subsequences (namely S_L, S_E, and S_G), as in Figure 3.2(d). Since $|S_L| = 5$ and $|S_E| = 1$, we have $|S_L| + |S_E| < k$ and PARALLEL_SELECT is again called recursively in step 5.

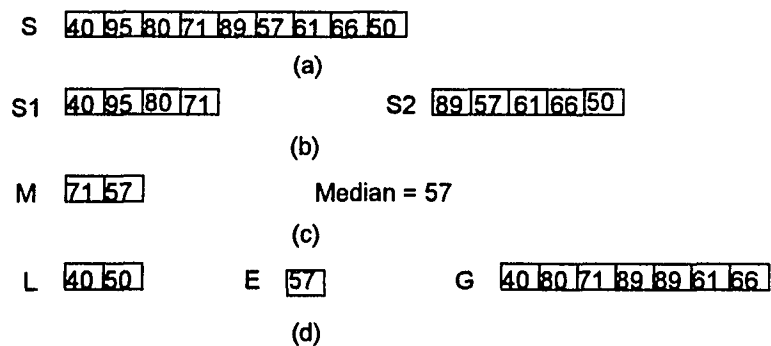

Figure 3.3 Second recursive call to PARALLEL_SELECT

At this point, the input to PARALLEL_SELECT is a new sequence S, as shown in Figure 3.3(a), and $k = 2$. Since $\lfloor |S|^{1-x} \rfloor$, the sequence is subdivided into $\lfloor |9|^{1-.5745} \rfloor = 2$ subsequences S_i of length $\lceil |S|^x \rceil = \lceil |9|^{.5745} \rceil = 4$. The first processor receives four elements and the second receives five elements, as in Figure 3.3(b). The median value (M=57) is found and the sequence is again divided into three subsequences, as shown in Figure 3.3(c-d). Since $|S_L| = 2$ and k=2, $|S_L| \geq k$ and PARALLEL_SELECT is called recursively in step 5. For the new sequence (Figure 3.4) the k-th (second) value, which is 50, is found in step 1 since the size of the sequence is less than Q, which is 4.

S	40	50

Figure 3.4 Call to SEQUENTIAL_SELECT in step 1

3.2 TurboNet Implementation and Timing Results

The parallel select algorithm discussed in the previous subsection has been implemented on the NJIT TurboNet parallel computer. The TurboNet implementation of the algorithm and its performance results are presented in this section. The original theoretical algorithms have been enhanced to give a robust performance on the TurboNet parallel computer. This is done without changing the basic workings of the original parallel algorithms. Various practical problems were considered in the implementations; these include bottlenecks due to the TurboNet's single bus architecture, limited size of the local SRAM and of the global DRAM memories, and communications and synchronization among the processors. For the implemented parallel algorithm, performance

measurements are shown for mainly three cases: single processor, 2 processors, and 4 processors. For each of the three cases, the algorithm's execution times (in microseconds) are shown for various sizes of input arrays, from 400 to 4000 elements.

3.2.1 TurboNet Implementation of Parallel Select

The implementation of the parallel select algorithm on TurboNet and its relevant performance results are presented in this subsection. One, two, or four processors use the DRAM shared-memory for inter-communication.

Contrary to the original parallel algorithm in Section 3.1, in Step 1 instead of keeping the value of Q constant(Q=5), in the implementation of parallel select it is varied from Q=5, for small input sequences, to Q=7, for very large sequences. The Q value variation is mainly done so that the program counter stack does not overflow when frequent and continuous recursive calls to the algorithm are made. In Step 4, only processor 1 is assigned the task of dividing the original sequence into three subsequences S_L, S_E, and S_G. In the original parallel selection algorithm (see Section 3.1) all processors are used to complete this task. The modification is made to reduce the communications overheads (DRAM usage) among the processors, thus boosting the overall performance. This is especially very apparent for very large size input arrays (see Tables 3.1-3.3). Also, we should note that there are some hidden or implied tasks that are required in the implementation. For instance, in Step 4, when the sequence S is divided into subsequences, the new resulting subsequences S_L, S_E, and S_G, are first copied into temporary memory locations. When the subdividing task is completed the

subsequences are copied back into their original memory locations in the order of S_L, S_E, and S_G. Even with these extra hidden tasks, the overall performance of the parallel algorithm on TurboNet is very impressive compared to its sequential counterpart. The parallel select algorithm implementation was tested and execution time data were taken for the following case: the size of input unsorted arrays varies from n=400 to n=4000 elements.

The Table 3.1 contains the average execution time (in μsec) where all cases were considered. The plots of the execution time (in μsec) versus the input array size (n elements) is shown in Figures 3.5.

Table 3.1: Averaged Execution Time for Parallel Selection

Array size (elements)	Execution Time (μsec)		
	1 processor	**2 processors**	**4 processors**
400	28055	26983	23620
800	71120	57419	47849
1200	130614	92781	73908
1600	205767	139364	102164
2000	296603	182056	134959
2400	401020	236330	167830
2800	520787	297421	205397
3200	654855	353659	247622
3600	804961	431191	288228
4000	879412	515278	337798

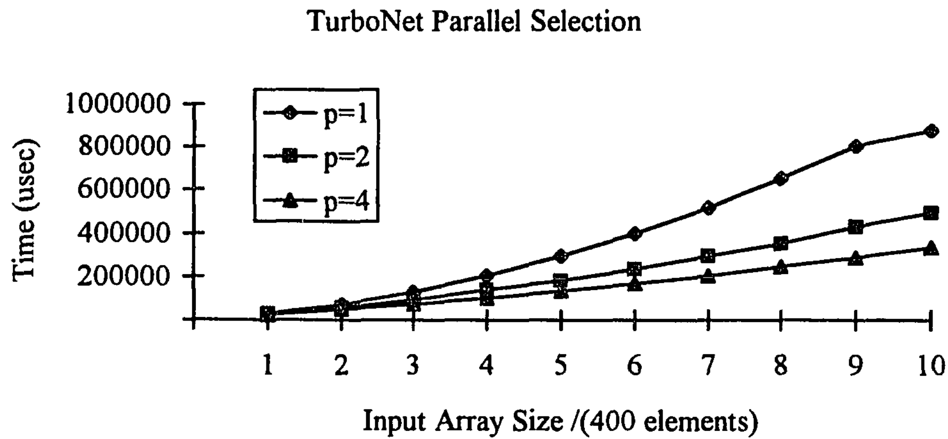

Figure 3.5 Plot of average execution time (in μsec) versus the number of elements

The results obtained do show the significant reduction of the execution time that is achieved by applying parallel processing. The dramatic performance improvement is very apparent for sequences of very large size.

CHAPTER 4

PARALLEL MERGE

4.1 Parallel Merging

The merging problem arises in many areas, especially in database applications and file management [1]. Although many applications involve the merging of non-numeric data (for example: merging of two alphabetically sorted mailing lists), this task can be handled easily once the basics of the problem have been resolved. Merging involves taking two sequences of numbers sorted in non-decreasing order and merging them to form a third resulting sequence, also sorted in nondecreasing order. For two given sorted sequences $A = \{a_1, a_2, ..., a_r\}$ of length r and $B = \{b_1, b_2, ..., b_s\}$ of length s, the resulting merged the sequence is $C = \{c_1, c_2, ..., c_{r+s}\}$ also sorted in nondecreasing order, such that each c_i in the sequence C belongs to either the sequence A or B, and each a_i and each b_i appears exactly once in the sequence C.

For very large size arrays, this task is very time consuming when done on a single processor system. However, it can be done very efficiently on a parallel computer by distributing the workload among all available processors and having each processor utilize an efficient sequential algorithm. In the subsection that follows, sequential and parallel merge algorithms are discussed. First, two sequential algorithms are discussed, an algorithm to do sequential merge (SEQUENTIAL_MERGE) and an algorithm to find the median of two sequences (TWO_SEQUENCE_MEDIAN). Then, a fast EREW parallel merge algorithm is introduced which utilizes the sequential algorithms.

4.1.1 Sequential Merging

For two given sequences of numbers $A = \{a_1, a_2, ..., a_r\}$ and $B = \{b_1, b_2, ..., b_s\}$ sorted in nondecreasing order, we want to merge them to form a third sequence C, also sorted in nondecreasing order. The merging process is to be performed by a single processor. This can be done as follows: Define two pointers, one pointer for each sequence. At the start of the merge, the pointers are initialized so that they are positioned at elements a_1 and b_1, respectively. Next, the smaller of a_1 and b_1 is found and is assigned to c_1, and the pointer to the sequence from which c_1 came is advanced by one position. Again, the two pointed elements are compared to find the smaller of the two. The smaller element is assigned to c_2 and, again, the corresponding pointer is advanced by one. This process is repeated until one of the two input sequences is exhausted. The leftover elements in the other sequence are now appended to C. The algorithm is given in what follows as procedure SEQUENTIAL_MERGE. Its description is simplified by assuming that a_{r+1} and b_{s+1} exist [1].

procedure SEQUENTIAL_MERGE (A, B, C)

(1) (1.1) $i \leftarrow 1$

(1.2) $j \leftarrow 1$

(2) **for** $k = 1$ to $r + s$ **do**

if $a_i < b_j$ **then** (i) $c_k \leftarrow a_i$

(ii) $i \leftarrow i + 1$

else (i) $c_k \leftarrow b_j$

(ii) $j \leftarrow j+1$

end if

end for.

The following numerical example shows how the algorithm works. We want to merge two sorted sequences, $A = \{a_1, a_2, ..., a_5\} = \{-3,0,2,5,8\}$ and $B = \{b_1, b_2, ..., b_6\} = \{-2,1,5,7,11\}$ to form a third sequence C, also in sorted order. Here, $r = s = 5$. Therefore, the resulting sequence C will be of size $r + s = 5 + 5 = 10$.

Step 1: $i = 1$ and $j = 1$

Step 2: For $k = 1$, $a_1 < b_1$ is true, so $c_1 = -3$ and $i = 2$

$k = 2$, $a_2 < b_1$ is false, so $c_2 = -2$ and $j = 2$

$k = 3$, $a_2 < b_2$ is true, so $c_3 = 0$ and $i = 3$

$k = 4$, $a_3 < b_2$ is false, so $c_4 = 1$ and $j = 3$

$k = 5$, $a_3 < b_3$ is true, so $c_5 = 2$ and $i = 4$

$k = 6$, $a_4 < b_3$ is false, so $c_6 = 5$ and $j = 4$

$k = 7$, $a_4 < b_4$ is true, so $c_7 = 5$ and $i = 5$

$k = 8$, $a_5 < b_4$ is false, so $c_8 = 7$ and $j = 5$

$k = 9$, $a_5 < b_5$ is true, so $c_9 = 8$ and $i = 6$

Therefore, at $k = 10$, $c_{10} = 11$.

In the worst case($r = s = n$), this algorithm requires n comparisons to generate the sequence C. Therefore, the algorithm runs in $O(n)$ time.

4.1.2 Finding the Median of Two Sorted Sequences

We want to find the median of two given sequences (sorted in nondecreasing order) without actually sorting the sequences. An algorithm is introduced in [1] as procedure TWO-SEQUENCE_MEDIAN(A, B, x, y). For two given sorted sequences $A = \{a_1, a_2, ..., a_r\}$ and $B = \{b_1, b_2, ..., b_s\}$, where $r, s > 1$, we let $m = r + s$ represent the length of the resulting sequence C which is formed after merging the two sequences. We want to find the median, the $\lceil m/2 \rceil$th element, of the sequence C without actually forming C. This algorithm returns an index pair (a_x, b_y). One of the two indices is a median of C, depending on the one that satisfies following properties:

(1) a_x is the median of $A.B$ if a_x is larger than $\lceil m/2 \rceil - 1$ elements and smaller than $\lfloor m/2 \rfloor$ elements. b_y is the median of $A.B$ if b_y is larger than $\lceil m/2 \rceil - 1$ elements and smaller than $\lfloor m/2 \rfloor$ elements.

(2) If a_x is the median, then b_y is either (i) the largest element in B smaller than or equal to a_x, or (ii) the smallest element in B larger than or equal to a_x.

Similarly, if b_y is the median, then a_x is either (i) the larger element in A smaller than or equal to b_y, or (ii) the smallest element in A larager than or equal to b_y.

(3) If more than one pair satisfies 1 and 2, then the algorithm returns the pair for which $x + y$ is the smallest.

The algorithm is as follows [1]:

TWO-SEQUENCE_MEDIAN(A, B, x, y)

(1) (1.1) $low_A \leftarrow 1$

(1.2) $low_B \leftarrow 1$

(1.3) $high_A \leftarrow r$

(1.4) $hgih_B \leftarrow s$

(1.5) $n_A \leftarrow r$

(1.6) $n_B \leftarrow s$

(2) **while** $n_A > 1$ **and** $n_B > 1$ **do**

(2.1) $u \leftarrow low_A + \lceil (high_A - low_A - 1)/2 \rceil$

(2.2) $v \leftarrow low_B + \lceil (high_B - low_B - 1)/2 \rceil$

(2.3) $w \leftarrow \min(\lfloor n_A / 2 \rfloor, \lfloor n_B / 2 \rfloor)$

(2.4) $n_A \leftarrow n_A - w$

(2.5) $n_B \leftarrow n_B - w$

(2.6) **if** $a_u \geq b_v$

then (i) $high_A \leftarrow high_A - w$

(ii) $low_A \leftarrow low_B + w$

else (i) $low_A \leftarrow low_A + w$

(ii) $high_B \leftarrow high_B - w$

end if

end while.

(3) Return as x and y the indices of the pair from $\{a_{u-1}, a_u, a_{u+1}\} \times \{b_{v-1}, b_v, b_{v=1}\}$

satisfying properties 1-3 of a median pair.

In the worst case (r=s=n), the algorithm requires $c_a + c_b \log(\min\{r,s\})$ time units which is $O(\log n)$. The algorithm is being utilized in the next section to form a fast parallel merging algorithm.

4.1.3 Parallel Merging on an EREW Computer

By applying parallel processing techniques and utilizing the efficient sequential algorithms discussed in previous subsections, namely SEQUENTIAL_MERGE and TWO-SEQUENCE_MEDIAN, the merging task can be done very efficiently. An algorithm is presented as procedure EREW_PARALLEL_MERGE which uses this approach [1]. It has small running time and optimal cost. It requires a sublinear number of processors and adapts to the actual number of processors available. The algorithm assumes N processors, where N is a power of 2. The merging is done in two stages. The sorted sequences A and B are partitioned into N subsequences $A_1, A_2, \ldots, A_N$ and $B_1, B_2, \ldots, B_N$ such that $|A_i| + |B_i| = (r+s)/N$. Also, all $A_i.B_i$ subsequences' elements are smaller than or equal to those of $A_{i+1}.B_{i+1}$ for $1 \le i \le N-1$. Next, all A_i and B_i are merged simultaneously.

The algorithm is as follows [1]:

procedure EREW_PARALLEL_MERGE (A, B, C)

(1) (1.1) Processor P_1 obtains the quadruple $(1, r, 1, s)$

(1.2) **for** $j = 1$ **to** $\log N$ **do**

for $i = 1$ **to** 2^{j-1} **do in parallel**

Processor P_i has received the quadruple (e, f, g, h)

(1.2.1) { Finds the median pair of two sequences, A_i and B_i }

TWO-SEQUENCE_MEDIAN ($A[e,f]$, $B[g,h]$, x, y)

(1.2.2) { Computes four pointers p_1, p_2, q_1, and q_2 as follows}

if a_x is the median

then (i) $p_1 \leftarrow x$

(ii) $q_1 \leftarrow x+1$

(iii) **if** $b_y \leq a_x$ **then** { $p_2 \leftarrow y$, $q_2 \leftarrow y+1$ }

else { $p_2 \leftarrow y-1$, $q_2 \leftarrow y$ }

end if

else (i) $p_2 \leftarrow y$

(ii) $q_2 \leftarrow y+1$

(iii) **if** $a_x \leq b_y$ **then** { $p_1 \leftarrow x$, $q_1 \leftarrow x+1$ }

else { $p_1 \leftarrow x-1$, $q_1 \leftarrow x$ }

end if

end if

(1.2.3) Communicates the quadruple (e, p_1, g, p_2) to P_{2i-1}

(1.2.4) Communicates the quadruple ($_1$, , $_2$,) to

end for

end for.

(2) **for** i = 1 **to** N **do in parallel**

Processor P_i having received the quadruple (a,b,c,d)

(2.1) $w \leftarrow 1+((i-1)(r+s))/N$

(2.2) $z \leftarrow \min\{i(r+s)/N,(r+s)\}$

(2.3) SEQUENTIAL MERGE ($A[a,b]$, $B[c,d]$, $C[w,z]$)

end for.

Working of the parallel merge algorithm is described by an example that follows. For two given sorted sequences S1 of size r=16 and S2 of size s = 14 as shown in Figure 3.3.1, we want to merge the sequences using a four processor parallel computer and produce a third resulting sorted sequence C of size $r+s$. In step 1.1, processor P1 receives the two sequences, (e,f,g,h) = (1,16,1,14). During the first interation of Step 1.2, P1 calls the procedure TWO-SEQUENCE_MEDIAN and finds the median indices (a_x, b_y) = (9,6) as shown in Figure 4.1. It keeps part of S1 and S2 (S1[1,9] and S2[1,6]) and communicates S1[10,16] and S2[7,14] to P2 using the shared memory.

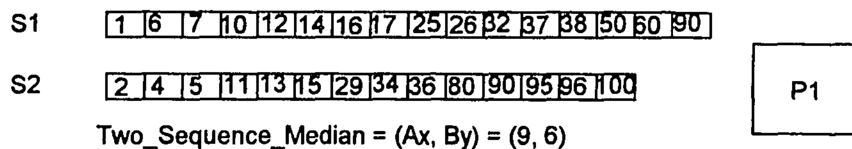

Figure 4.1 Two sorted sequences S1 and S2 to be merged

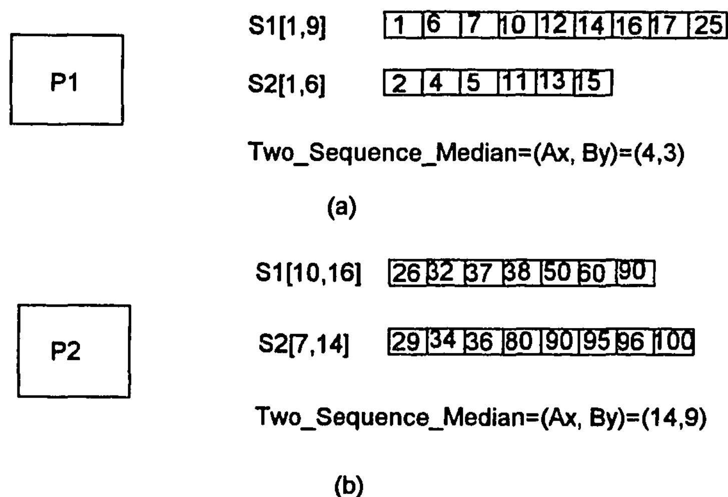

Figure 4.2 S1 and S2 are subdivided during the first pass of step 1.2

During the second iteration P1 again calls the TWO-SEQUENCE_MEDIAN and finds the indices of the median pair S1[1,9] and S2[1,6], namely (a_x, b_y)= (4,3), as shown in Figure 4.2 (a). Simultaneously, P2 calls the TWO-SEQUENCE_MEDIAN and finds the indices of the median pair S1[10,16] and S2[7,14], namely (a_x, b_y) = (14, 9), also shown in Figure 4.2(b). Processor P1 keeps S1[1,4] and S2[1,3] and communicates (5,9,4,6), namely S1[5,9] and S2[4,6], to P2. At the same time, P2 communicates (10,14,7,9) to P3 and (15,16,10,14) to P4. At this point, all processors have a part of S1 and S2 as shown in Figure 4.3 (a)-(d). In Step 2, each processor calls the SEQUENTIAL MERGE to merge the parts of S1 and S2, producing the resulting sequence C_i, as shown in Figure 4.3 (a)-(d).

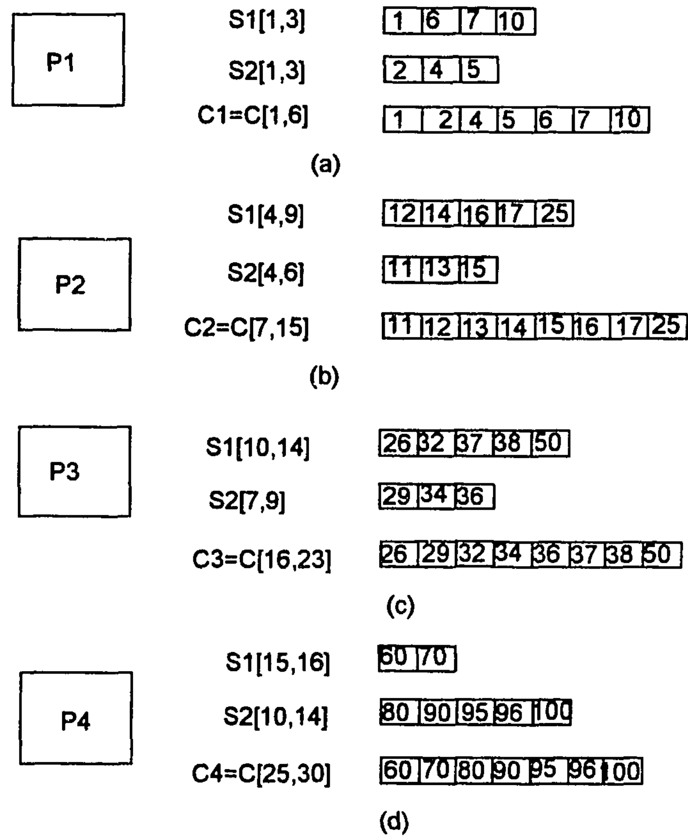

Figure 4.3 S1 and S2 are divided among all four processors for merging

The algorithm is analyzed as follows: P1 reads from memory in constant time c_1 in Step 1.1. TWO-SEQUENCE_MEDIAN is called in Step 1.2 which takes $\log(r+s)$ time units. Steps 1.2.2, 1.23, and 1.2.4 take constant time c_2. Therefore, Step 1 is completed in $\log N \times (c_1 + c_2 + \log(r+s))$ time units since there are $\log N$ iterations. In step 2, each processor calls the Sequential Merge procedure to merge at most $(r+s)/N$ elements in $(r+s)/N + c_3$ time units. In the worst case (r=s=n), step 1 and step 2 take $\log N \times (c_1 + c_2 + \log 2n) + 2n$ time units which can be expressed as follows [1]: $t(2n) = O(n/N + \log^2 n)$

This EREW Parallel Merge algorithm adapts to the number of available processors in the parallel computer. In addition to being adaptive, the algorithm is also cost optimal. Later, in section 4.2, an implementation of the parallel merge on TurboNet is introduced and performance results are summarized.

4.2 TurboNet Parallel Merge

In this section, an implementation of the parallel merge algorithm on TurboNet is discussed in detail. Relevant performance results are summarized later in a table and figure which show dramatic speedups. The parallel merge TurboNet implementation is done for one, two, and four processors. The shared memory is used to communicate data and program control information (various flags status) among processors. Enhancements are made to the original algorithm to simplify its implementation on TurboNet. This is done without affecting the basic workings of the original parallel merge algorithm (as in section 4.1).

Since we already know that a maximum of four processors will be used on TurboNet, the for-do loop limit is set to 2 passes in step 1.2. Further modifications are made (in step 2.1 and step 2.2) on how the size of the resulting merged sequence ($w = |C_i|$) and its offset value (z) are computed just before invoking the sequential merge algorithm in step 2.3. In the original merge algorithm in section 4.1, each of the two subsequences, A and B, is partitioned into N subsequences of equal size based on their original sequence size (r and s) and regardless of the quadruple they receive in step 1. Hence, the size of the resulting merged subsequence ($|C_i|$) computed by each processor is

also of equal value. However, after rigorously testing the workings of the algorithm, it is found that this is not the case in many instances. The merged subsequences C_i produced by the processor have different sizes when its $\|A_i| - |B_i\| > 0$, thus the incorrect offset (*z*) value is computed. In many cases where $\|A_i| - |B_i\| = D_i > 0$ and D_i is different for each processor, the resulting subsequence formed by each processor is written into the merged sequence C starting at an incorrect index position (or offset). To correct this problem and expand the algorithm so that it can merge any order of sorted elements as well as subsequences of different size, the following enhancements are introduced: (1) In step 2.1, instead of having each processor compute the offset value from the equation $w = 1 + ((i-1)(r+s))/N$, now the quadruple (*a*, *b*, *c*, *d*) indices received by each processor are also examined and the offset (*w*) is computed from $w = a + c$. (2) Similarly, the corresponding size of the resulting subsequence is computed from $z = b - a + d - c$ in step 2.2. (3) Sequential merge is enhanced so that it can now merge two sorted sequences by just making a single comparison for the case when the last element of one subsequence is less than the first element of the other subsequence, for example where $A_i(r) < B_i(1)$.

The TurboNet (4 processor) version of the parallel merge algorithm is as follows:

procedure TURBONET _ PARALLEL_MERGE (*A*, *B*, *C*)

Stage 1: (1.1) Processor P_1 obtains the quadruple (1, *r*, 1, *s*)

(1.2) **for** j = 1 **to** 2 **do**

for i = 1 **to** 2^{j-1} **do in parallel**

Processor P_i has received the quadruple (*e*, *f*, *g*, *h*)

(1.2.1) { Finds the median pair of two sequences, A_i and B_i }

TWO-SEQUENCE_MEDIAN (*A*[*e*,*f*], *B*[*g*,*h*], *x*, *y*)

(1.2.2) { Compute four pointers p_1, p_2, q_1, and q_2

as follows }

if a_x is the median

then (i) $p_1 \leftarrow x$

(ii) $p_1 \leftarrow x+1$

(iii) **if** $b_y \le a_x$ **then** { $p_2 \leftarrow y,\ q_2 \leftarrow y+1$ }

else { $p_2 \leftarrow y-1,\ q_2 \leftarrow y$ }

end if

else (i) $p_2 \leftarrow y$

(ii) $q_2 \leftarrow y+1$

(iii) **if** $a_x \le b_{y_x}$ **then** { $p_1 \leftarrow x,\ \ q_1 \leftarrow x+1$ }

else { $p_1 \leftarrow x-1, q_1 \leftarrow x$ }

end if

end if

(1.2.3) Communicates the quadruple (e, p_1, g, p_2) to P_{2i-1} using

the shared memory.

(1.2.4) Communicates the quadruple (q_1, f, q_2, h) to P_{2i} using

the shared memory.

end for

end for.

Stage 2: **for** $i = 1$ **to** N **do in parallel**

Processor P_i having received the quadruple (a,b,c,d)

(2.1) $w \leftarrow a + c$

(2.2) $z \leftarrow b - a + d - c$

(2.3) SEQUENTIAL MERGE ($A[a,b]$, $B[c,d]$, $C[w,z]$)

end for.

The algorithm was tested and performance results were collected for all the following cases: (1) the size of the input sequences, A and B, ranged from 200 to 2600 elements, (2) one, two, and four processors were used, and (3) the input sequences were of different sizes.

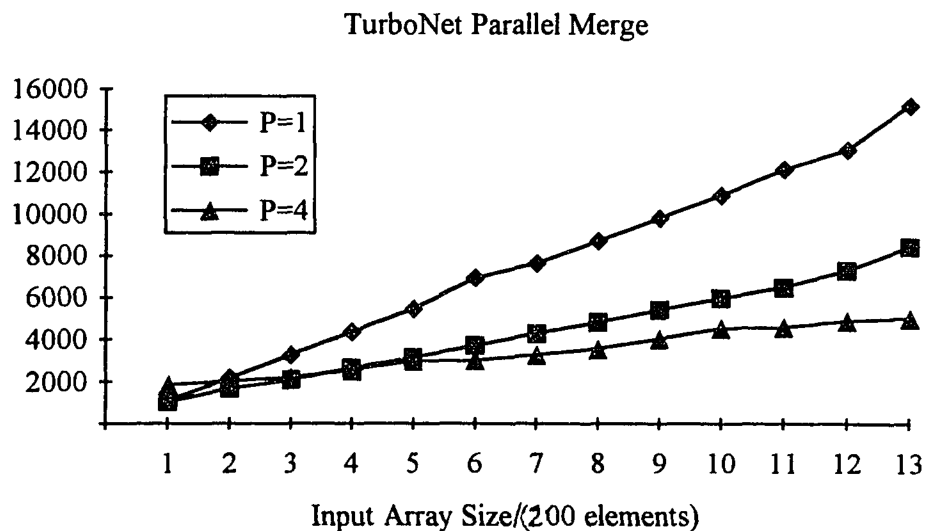

Figure 4.4 Plot of the execution time (in μsec) versus the number of elements

Table 4.1 Execution Time(μsec) for Parallel Merge

Array size (elements)	Execution Time (μsec)		
	1 processor	**2 processors**	**4 processors**
200	1097	1057	1668
400	2187	1701	2050
600	3277	2098	2212
800	4367	2646	2549
1000	5457	3139	2989
1200	6942	3725	3023
1400	7637	4292	3283
1600	8727	4829	3564
1800	9817	5407	4039
2000	10907	5955	4544
2200	12152	6497	4590
2400	12635	7018	4772
2600	13495	7589	5023

The performance results are summarized in Table 4.1 and Figure 4.4. For small size arrays (200 to 600 elements) the execution time is large for two and four processors cases because of the overhead due to shared-memory communications among processors. However, the data show significant reduction in the execution time achieved for the two and four processors cases, when the input sequences are of very large size, especially with 1400 to 2600 element sequences.

CHAPTER 5

PARALLEL SORT

5.1 Parallel Sorting

Among all computational tasks in today's applications sorting appears to be one of the most important. The sorting problem arises in many areas, from database applications to file management. In fact, sorting is so basic that many other computational tasks, such as selection, require sorting. Although many applications involve sorting of non-numeric data (for example: sorting of records, lists, etc.), this task can be handled easily once the basics of the problem have been resolved. Sorting is defined as follows. For a given sequence $S = \{s_1, s_2, \ldots, s_n\}$ of size n elements in random order, we want to sort the sequence such that $s_i' < s_{i+1}'$ for all n elements, where $1 \leq i \leq n-1$.

In the next section, a sequential sort algorithm is discussed. The objective is to adapt the sequential algorithm to run on a parallel computer. Later, the algorithm will be utilized by a parallel version of the sort algorithm.

5.1.1 Sequential Sorting

There are many algorithms for sorting on a sequential computer. One of the well known and easy techniques of sorting is bubble sort. It requires one pass through the sequence or n comparisons to sort each element. Hence, for the sequence of size n, the running time for bubble sort is $O(n^2)$, which is very high. The sorting task can be done more

efficiently on a single-processor computer by using an algorithm called QUICKSORT which is discussed here. It is recursive and has a running time of $O(n \log n)$ [1].

procedure QUICKSORT (S)

if $|S| = 2$ **and** $s_2 < s_1$

then $s_1 \leftrightarrow s_2$

else if $|S| > 2$ **then**

Step 1: {Determine m, the median element of S }

SEQUENTIAL_SELECT ($S, \lceil |S| / 2 \rceil$)

Step 2: { Divide S into two subsequences, S_1 and S_2}

$S_1 \leftarrow \{s_i : s_i \leq m\}$ and $|S_1| = \lceil |S| / 2 \rceil$

$S_2 \leftarrow \{s_i : s_i \geq m\}$ and $|S_2| = \lfloor |S| / 2 \rfloor$

Step 3: QUICKSORT (S_1)

QUICKSORT (S_2)

end if

end if

It works as follows: QUICKSORT first finds the median of S and then divides S into S1 and S2. S1 is a sequence of elements smaller than or equal to the median. S2 is a sequence of elements larger than or equal to the median. Next, QUICKSORT is called

recursively for S1 and then S2. This continues until each sequence has one or two elements.

5.1.2 Parallel Sort Algorithm

Because sorting is an important task, several algorithms have been developed for sorting on parallel computers. In this section, a parallel sort algorithm is introduced. It adapts to the number of available processors on the parallel computer at hand. Each processor in the system uses efficient sequential algorithms to complete the sorting tasks. In addition to being adaptive, it is also cost optimal. The basic idea behind the workings of the algorithm is as follows. For a given unsorted sequence of n elements, first k-1 medians are found using Parallel Select (described in section 3.1), where $k = 2^{\lceil 1/x \rceil}$ and x is computed from $N = n^{1-x}$. The sequence is then divided into k subsequences of size $n / 2^{\lceil 1/x \rceil}$ elements each. This is done so that every element of the subsequence S_i is smaller than or equal to every element of the subsequence S_{i+1}, for $i = 1, \ldots, k-1$. The algorithm is applied in parallel to each of the k subsequences using $N / (k-1)$ processors per subsequence. This is continued recursively for each of the subsequences until the entire original sequence S is sorted in nondecreasing order. When the size of the sequence and subsequence is less than or equal to k, the sequence is sorted using the sequential sort algorithm QUICKSORT.

The parallel sort algorithm [1] is as follows:

procedure EREW_SORT (S)

```
if |S| ≤ k then
        QUICKSORT (S)
else    (1)   for i = 1 to k - 1 do
                PARALLEL_SELECT (S, ⌈i|S|/k⌉) { Obtain m_i }
              end for
        (2)   S_1 ← {s ∈ S : s ≤ m_1}
        (3)   for i = 2 to k - 1 do
                  S_i ← {s ∈ S : m_{i-1} ≤ s ≤ m_i}
              end for
        (4)   S_k ← {s ∈ S : s ≥ m_{k-1}}
        (5)   for i = 1 to k/2 do in parallel
                  EREW_SORT (S_i)
              end for
        (6)   for i = (k/2) + 1 to k do in parallel
                  EREW_SORT (S_i)
              end for
end if
```

A simple numerical example that follows shows the workings of the Parallel Sort algorithm. For a given unsorted sequence S of n=40 elements (see Figure 5.1), we want to sort the sequence in non-decreasing order using a N=4 processor parallel computer. First, x=0.624 and k= 4 are computed from $N = n^{1-x}$ and $k = 2^{\lceil 1/x \rceil}$. In step 1, parallel

select is invoked to find k-1 = 3 medians ($m_1 = 10, m_2 = 20$, and $m_3 = 30$) as shown in Figure 5.2(a).

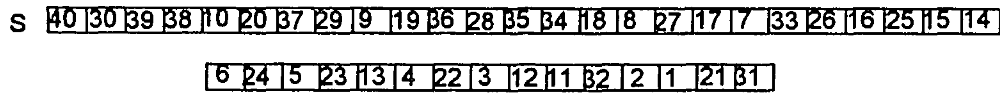

Figure 5.1 Initial unsorted sequence S of size n=40 elements

Next, in steps 2 to 4, the original sequence S is divided into four subsequences S1 to S4 as shown in Figure 5.2(b). Note that every element of S1 is smaller than or equal to the median $m_2 = 10$. Every element of S2 is smaller than or equal to $m_2 = 20$, and greater than $m_2 = 10$. Similarly, the subsequences S3 and S4 are formed as shown in Figure 5.2(b). In step 5, the algorithm is called recursively to sort S1 and S2 simultaneously. Processors P1 and P2 are used to sort S1 in parallel (see Figure 5.3(a)). And, simultaneously, processors P3 and P4 are used to sort S2 as shown in Figure 5.3(b).

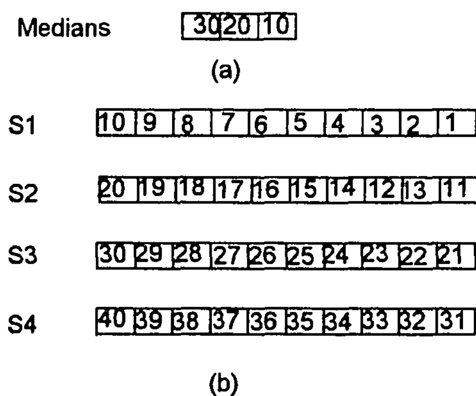

Figure 5.2 Dividing S into four subsequences for sorting by selection

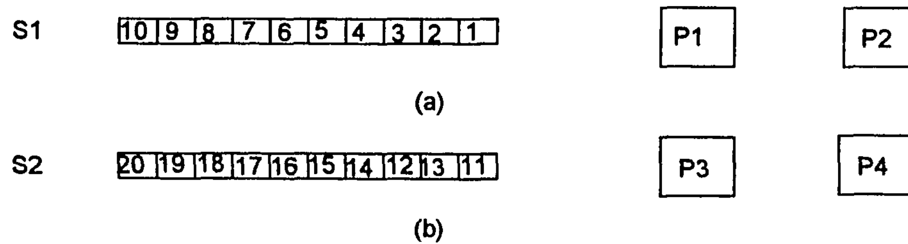

Figure 5.3 Recursive call to parallel sort for sorting S1 and S2 simultaneously

In step 6, parallel sort is again called recursively to sort S3 and S4 in parallel. The algorithm uses processors P1 and P2 simultaneously to sort S3. At the same time, processors P3 and P4 sort S4 simultaneously using the algorithm. This is shown in Figure 5.4(a) and Figure 5.4(b), respectively. The final sorted sequence is formed when Step 6 is completed (see Figure 5.5).

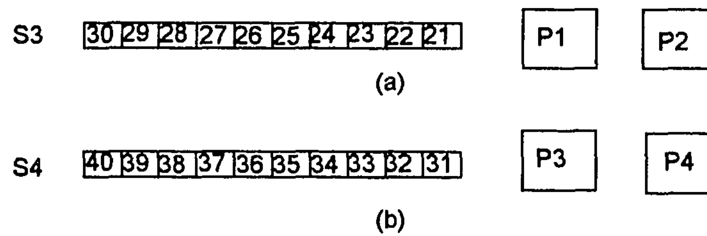

Figure 5.4 Recursive call to parallel sort for sorting S3 and S4 simultaneously

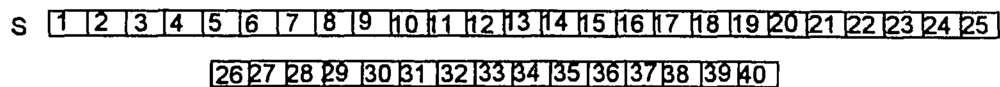

Figure 5.5 Final sorted sequence S of size n=40 elements (after step 6)

The call to QUICKSORT takes constant time, c_a, for k elements. In step 1, k-1 calls to parallel select take $c_b n^x$ time. Steps 2 to 4 also take constant time, c_c. The parallel sort is called recursively twice in steps 5 and 6. This part takes $2t(n/k)$ time. The total running time for the parallel EREW sort is $t(n) = c_a + c_b n^x + c_c + 2t(n/k)$. Therefore, the final running time for the algorithm is $t(n) = n^x \log n$ when the equation is solved for $t(n)$.

5.2 TurboNet Parallel Sort

Since sorting is a very important operation that has received a lot of attention, its implementation on parallel computers is a worthwhile objective and an interesting challenge. This section deals with the implementation of the Parallel Sort algorithm (previously discussed in Section 5.1) on the NJIT TurboNet System. The TurboNet version of the parallel sort algorithm is discussed in detail and relevant performance results are presented.

Several practical factors have been taken into consideration in enhancing the original algorithm to give good performance on TurboNet and simplify its implementation. Because of limited availability of local and shared memory, the maximum input array size is limited to 5000 elements, and the memory allocation is made to satisfy this requirement. Also, it is computed that for a very large input array size of 5000 to 8000 elements, the value of k is to remain constant, equal to 4. The computation is summarized in Table 5.1 below for 100 to 8000 elements, for two cases with $N = 4$ and $N = 2$.

A second enhancement is made to determine how the procedure QUICKSORT is used. Instead of calling QUICKSORT when the sequence size (n) is a constant k, as done in the original algorithm in Section 5.1, in the TurboNet implementation QUICKSORT is invoked when the input sequence size is C or smaller, where C is varied from 4, for $n = 10$, to 250, for $n = 4000$ elements. This modification is made mainly to minimize the program stack overflow that can occur during frequent recursive calls of the algorithm. The third enhancement is done to determine how the parallel select is invoked in step 1 of the parallel sort. In the TurboNet implementation of parallel sort, the parallel select is invoked by the sort algorithm only during its first pass, rather than in every pass, as done in the original sort algorithm.

Table 5.1 Summary of x and k value computations

$N = n^{1-x}$ and $k = 2^{\lceil 1/x \rceil} = 4$			
CPU (N)	**Array Size (n)**	x	**$1/x$**
4	100	0.69	1.43
4	1000	0.79	1.25
4	4000	0.83	1.20
4	8000	0.85	1.18
2	100	0.84	1.17
2	1000	0.89	1.11
2	4000	0.91	1.09
2	8000	0.92	1.08

During the next pass or a recursive call, the sequential select is called simultaneously by each processor in step 1. This is mainly done to reduce the SBUS usage by all processors, thus significantly reducing the shared-memory communication needed among

processors. This is further clarified in a numerical example discussed later. The enhancement has significantly reduced the overall communication overhead, thus giving a very impressive performance boost.

The TurboNet version of the parallel sort is presented as follows: NOTE: When the procedure is called the first time, the size of the sequence is the longest, $|S|$ = *ORIGINAL*. The sequence size changes (becomes smaller) when the procedure is called the second time (recursively), and so on. Therefore, initially the variable *size* is equal to *ORIGINAL*.

procedure TURBONET_PARALLEL_SORT (*S*, *size*)

if $|S| \leq C$ **then**

    QUICKSORT (S)

else

    (1) **if** *size* = *ORIGINAL* **then**

        for $i = 1$ **to** $k - 1$ **do**

            PARALLEL_SELECT $(S, \lceil i|S|/k \rceil)$ { Obtain m_i }

        end for

    else if *size* < *ORIGINAL* **then**

        for $i = 1$ **to** $k - 1$ **do**

            SEQUENTIAL_SELECT $(S, \lceil i|S|/k \rceil)$ { Obtain m_i }

        end for

    end if

(2) $S_1 \leftarrow \{s \in S: s \leq m_1\}$

for $i = 2$ **to** k -1 **do**

$S_i \leftarrow \{s \in S: m_{i-1} \leq s \leq m_i\}$

end for

$S_k \leftarrow \{s \in S: s \geq m_{k-1}\}$

(3) **for** i = 1 **to** k **do in parallel**

Processor P_i calls TURBONET_PARALLEL_SORT $(S_i, size)$

end for

end if

An example that follows shows the actual workings of the TurboNet Parallel Sort. For a given unsorted sequence S (see Figure 5.6) we want to sort the sequence using the TurboNet Parallel Sort algorithm. When the algorithm is first called, the sequence size n is 40 elements and the parallel select is called in step 1 to find three medians, $m_1 = 10$, $m_2 = 20$, and $m_3 = 30$, as shown in Figure 5.7(a).

S | 40 | 30 | 39 | 38 | 10 | 20 | 37 | 29 | 9 | 19 | 36 | 28 | 35 | 34 | 18 | 8 | 27 | 17 | 7 | 33 | 26 | 16 | 25 | 15 | 14 |

| 6 | 24 | 5 | 23 | 13 | 4 | 22 | 3 | 12 | 11 | 32 | 2 | 1 | 21 | 31 |

Figure 5.6 Initial Unsorted Sequence of size n=40 elements

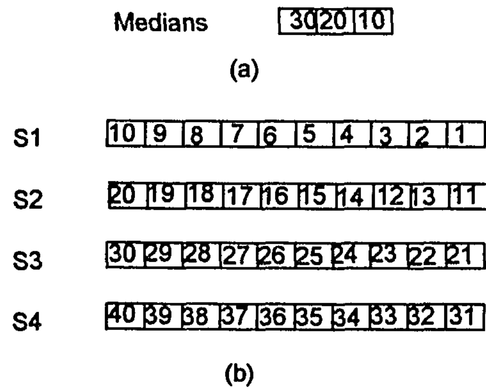

Figure 5.7 Dividing the S into four subsequences for sorting by selection

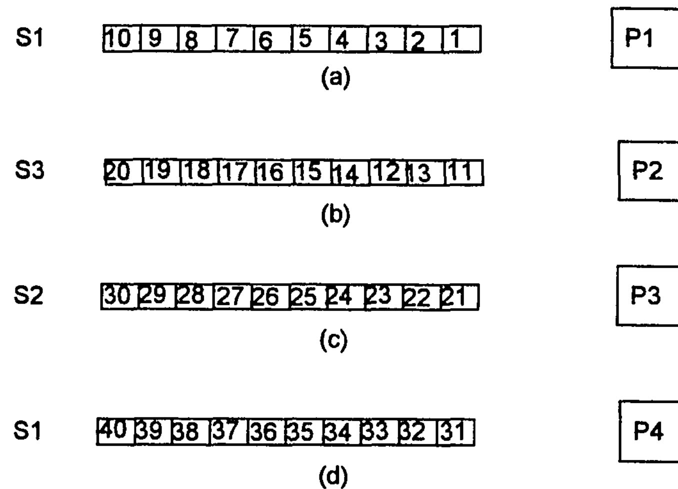

Figure 5.8 Recursive call to Parallel Sort by each processor

The original sequence is divided into four subsequences by processor one as shown in Figure 5.7(b). Note that this is done in just one pass (one step) through the original sequence S, rather than four passes through S to form four subsequences as done by the original algorithm in section 3.3(steps 2-4). In step 3, each processor reads its subsequence S_i and invokes the TurboNet parallel sort recursively to sort the

subsequence S_i in parallel. Now, during this recursive call the sort algorithm becomes sequential, but all processors are working in parallel on sorting their subsequences, as shown in Figure 5.8(a)-(d). Finally, at the end of step 3, the sequence is completely sorted, as shown in Figure 5.9.

Performance measurements were obtained for the following combinations of cases: (1) The input array size (n) ranged from 400 to 4000 elements, (2) 1, 2, and 4 processors were used, (3) the sequences were unsorted and of decreasing order.

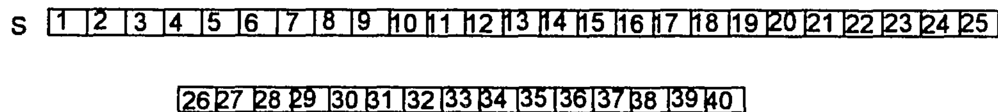

Figure 5.9 Final sorted sequence S of size n=40 elements (after step 6)

Table 5.2 summarizes the execution times (in microseconds) for unsorted input sequences of 400 to 4000 elements for a single processor, dual processors, and four processors. A plot of the data is also shown in Figure 5.9. As shown in Tables 5.2 and Figure 5.10 below, the results indicate very impressive performance improvement when 2 and 4 processors are used. This is very apparent for very large size of the input sequences, namely n=2000 to 4000 elements.

Table 5.2 Execution Time (μsec) for Parallel Sort

Array size (elements)	Execution Time (μsec)		
	1 processor	**2 processors**	**4 processors**
400	144593	142307	106419
800	350680	316603	211676
1200	649954	554155	341676
1600	1044320	843713	504710
2000	1503596	1166348	678995
2400	2017688	1534807	877654
2800	2552638	1962075	1093700
3200	3162698	2440201	1341916
3600	3856251	2965354	1609624
4000	4626148	3540366	1893207

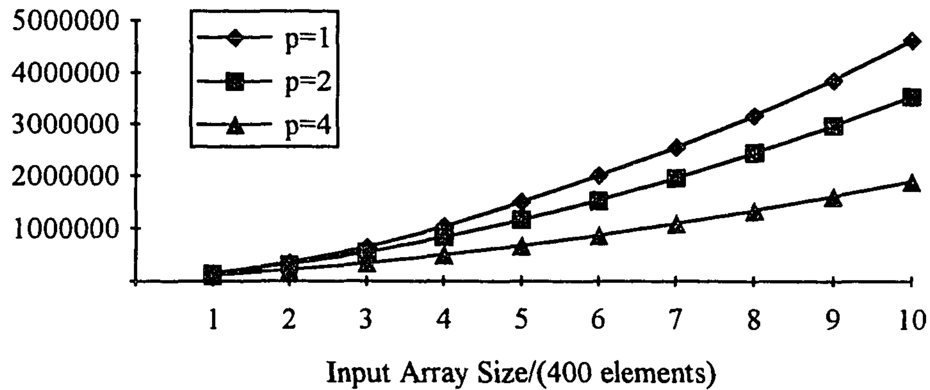

Figure 5.10 Plot of the execution time (in μsec) versus the number of elements

CHAPTER 6

CONCLUSIONS

In this thesis, parallel algorithms were implemented on a new experimental parallel computer, namely TurboNet, for selection, merge, and sort applications. The main objective of this thesis was to show that several algorithms can take advantage of the shared-memory capability of the hybrid architecture to achieve significant speedup. Three versions of each algorithm were developed and implemented employing shared-memory communications, namely for four processors, two processors, and a single processor (sequential). Experimental comparisons of these algorithms were also included. The performance results obtained from the implementation of the algorithms on TurboNet show considerable performance benefits that are possible in turns of execution time reduction. The implementations are very cost effective if the applications process very large amounts of data. Further study should involve implementation of the algorithms using the message-passing or hybrid communication paradigms available on TurboNet.

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

PARALLEL SELECTION PROGRAMS

```
/*--------------------------------------------------------------------
FILE:         dsp0.c
DESCRIPTION: DSP0 parallem selection program
PROCESSORS:   4
DATE:         November 10, 1994
PROGRAMMER:   Nitin Lad
--------------------------------------------------------------------*/

#include "newdef.h"

int _SELECT_LOCAL_DSP0(s_inaddr, s_size, s_k, CPU_USE, del_x)
signed long *s_inaddr;
int s_size, s_k, *CPU_USE;
float *del_x;
{
	int i, j, k, tot_seq_cpu, seq_size, m_index, n, N = 0;
	int first, last, itemp;
	float x, temp;
	signed long njit, median = 0;

	if ( s_size  <= MIN )
	{
		/* copy the sequence/sub-sequence to local memory */
		for ( i=0; i< s_size; i++)  t_addr[i] = s_inaddr[i];

		/* ONLY DSP0 execute seq. select to find kth sm. */
		median = seq_sort( &t_addr[0], s_size);
		/* median found */
		select_median[dsp_id] = median;

		/* copying kth element to select_median arr 1-4 */
		if ( s_k <= s_size)
			for ( i = 1; i < 4; i++)
				select_median[i] = t_addr[s_k-1];
		gen_flag = t_addr[s_k-1];
		gen_flag2 = gen_flag3 = gen_flag4 = t_addr[s_k-1];

		if ( s_k <= s_size)
			select_val_k[MAIN] = t_addr[ s_k -1];
		else
			select_val_k[MAIN] = 1000;

		return EXIT;

	}
	else
	{

		/* --- compute N from seq size and 1 - x factor  ----- */
		if ( loop_count == ZERO )
		{
			tot_seq_cpu = SEL_CPU;
			loop_count++;
		}
		else
		{
```

```
                        tot_seq_cpu = floor( pow(s_size, *del_x) );
                        loop_count++;
                }

                if ( tot_seq_cpu >= SEL_CPU ) tot_seq_cpu = SEL_CPU;
                *CPU_USE = N = tot_seq_cpu;

                /* have DSP 1 to N-1 start selection */
                for (i=0; i < N; i++) select_local_flag[i] = START;

                /* ------ compute sequence size for DSP0 ----- */
                seq_size =      ceil( pow(s_size, 1 - *del_x) );
                m_index = ceil( ceil( pow(s_size, 1- *del_x) )/2 );

                /* copy the sequence to local memory */
                for ( i=0; i< seq_size; i++)  t_addr[i] = s_inaddr[i];

                /*  sort and find median */
                select_median[dsp_id] = seq_select( t_addr, seq_size, m_index);

                select_local_flag[dsp_id]= DONE;
                return DONE;
        }       /* end of if */
}               /* end of routine */


/* re-declare the routine */
int _SELECT_LOCAL_DSP0(
signed long *s_inaddr,
int s_size, int s_k, int *CPU_USE,
float *del_x);

int _SELECT_LOCAL_CONTROL_DSP0(s_inaddr, s_outaddr, s_size, s_k, del_x)
signed long *s_inaddr, *s_outaddr;
int s_size, s_k;
float *del_x;
{
        int i, j, k, tot_seq, seq_size, med, n, N = 4;
        int first, last, itemp;
        float x, temp;
        signed long *new_addr, median = 0;
        int *CPU_USE = (int *)(EXT_LRAM1+40002);
        int *Lsize = (int *)(EXT_LRAM1+40004);
        int *Esize = (int *)(EXT_LRAM1+40006);
        int *Bsize = (int *)(EXT_LRAM1+40008);

        *CPU_USE = 0;

        /* copy address & size to local shared locations */
        select_local_size[MAIN] = s_size;

        /* set select_local_flag to start for DSP0 */
        select_local_flag[dsp_id] = START;

        /* initialize all medians value to zero */
        for (i=0; i < SEL_CPU; i++) select_median[i] = 0;

        /* following if is for debugging purpose */
        if ( s_size == 0 ) {
                gen_flag = s_inaddr[0];
                gen_flag2 = s_inaddr[1];
                gen_flag3 = s_inaddr[2];
                gen_flag4 = s_inaddr[3];
                return DONE;
        }

        /* call select_local_dsp0 to find kth element  */
```

```
	if (_SELECT_LOCAL_DSP0(s_inaddr, s_size, s_k, CPU_USE, del_x) == EXIT)
	{
		return DONE;
	}


	/* ELSE, wait until all DSPs have found medians */
	for (i=1; i < *CPU_USE; i++)
		while ( select_local_flag[i] != DONE);

	/* find the Median of medians */
	if ( *CPU_USE >= 3 )
		median = seq_select( &select_median[0], *CPU_USE, 2);
	else
		median = seq_select( &select_median[0], *CPU_USE, 1);

	select_med = median;

	/* divide the sequence into THREE sub-sequences */
	if( sub_seq(s_inaddr, median, s_size, Lsize, Esize, Bsize) == DONE)
	{
		/* determine if the k-th element is in 1st, 2nd, or 3rd seq. */
		if ( *Lsize >= s_k )
		{
			/* k-th element is in 1st sub-sequence */
			*select_offset = 0;
			s_size = *Lsize;
			/* call parallel select recursively */
			if (_SELECT_LOCAL_CONTROL_DSP0(s_inaddr,
				s_outaddr,s_size,s_k,del_x)==DONE)
				return DONE;
		}
		else if ( (*Lsize + *Esize) >= s_k )
		{
			/* k-th element is median */
			*select_offset = 0;
			gen_flag = gen_flag2 = gen_flag3 = gen_flag4 = median;
			select_med = median;
			return DONE;
		}
		else
		{
			/* k-th element is in the 3rd subsequence */
			s_size = *Bsize;
			s_k = s_k - *Lsize - *Esize;	/* re-compute k value */
			*select_offset = *Lsize + *Esize;
			*select_base = *select_base + *select_offset;
			/* call parallel select recursively */
			if (_SELECT_LOCAL_CONTROL_DSP0(s_inaddr+*select_offset,
				s_outaddr,s_size,s_k,del_x)==DONE);
		}
	}
	else
	{
		/* sub-dividing of sequence has failed, so return error */
		return FAIL;
	}	/* end if */

}		/* end of routine */


int _SELECT_DSP0(s_inaddr, s_outaddr, s_size, s_k)
signed long *s_inaddr, *s_outaddr;
int s_size, s_k;
{
	int i, j, k, tot_seq, seq_size, med, n, N = 4;
	int first, last, itemp, flag = FAIL;
```

```
        signed long njit, median = 0;

        float *del_x = (float *)(EXT_LRAM1+40000);

        s_inaddr++;
        s_outaddr++;

        for (i=0; i < s_size; i++) s_outaddr[i] = s_inaddr[i];

        select_local_size[MAIN] = s_size;
        gen_flag = gen_flag2 = gen_flag3 = gen_flag4 = 0;
        *select_offset = 0;
        for (i=0; i < SEL_CPU; i++) select_main_flag[i] = CONTINUE;
        for (i=0; i < SEL_CPU; i++) select_local_flag[i] = HOLD;
        select_wait_flag[0] = s_k;

        /* wait until signal received from host program */
        while ( !inter_flag );

        if ( s_size  <= MIN )
        {
              /* size of sequence is less than or equal to Q */

              /* copy the sequence into DSP's local memory */
              for ( i=0; i< s_size; i++) t_addr[i] = s_inaddr[i];

              /* ONLY DSP0 execute seq. select to find kth sm. */
              median = seq_sort( &t_addr[0], s_size);
              select_med = select_median[dsp_id] = median;

              /* copying kth element to select_median arr 1-4 */
              if ( s_k <= s_size)
                    for ( i = 1; i < 4; i++)
                          select_median[i] = t_addr[s_k-1];
              /* set all output flags to the k-th value */
              gen_flag = t_addr[s_k-1];
              gen_flag = gen_flag2 = gen_flag3 = gen_flag4 = t_addr[s_k-1];

              if ( s_k <= s_size)
                    select_val_k[MAIN] = t_addr[ s_k-1 ];
              else
                    select_val_k[MAIN] = 1000;

              return DONE;
        }
        else
        {
              /* the sequence size is greater than Q */

              /* compute original 1 - x value */
              *del_x =  log(SEL_CPU) / log(s_size);
              *select_base = 0;

              if (_SELECT_LOCAL_CONTROL_DSP0(s_inaddr,s_outaddr,s_size,
                    s_k,del_x) == DONE)
                    return DONE;
        }     /* end if */
}             /* end of routine */


main()           /* _main:  dsp0.c  parallel_sort */
{
      int i, j=0, local_broad = 0, k_value, tempa;
      signed long tempb;
      unsigned long timerStart, timerEnd, temp;
      float elapsed_time = 0.0, temp_float;
      signed long last_index, k_small, k_large, k_mid;
```

```
signed long savetime, savetime2, savetime3, xx, save_size;
int choose = 0;

/* enable the DSP's interrupt facility */
GIE_ON();
SET_PERIOD(0xffffffff);         /* set timer periof */

while (1)
{
      /* wait until signal is received from the host */
      while ( (main_flag != GO) );

      if ((main_flag == GO) && (select_flag == GO))
      {
            /* intialize all output flags */
            select_wait_flag[0] = JUNK;
            select_wait_flag[1] = JUNK;
            select_wait_flag[2] = JUNK;
            select_wait_flag[3] = JUNK;
            sort_main_flag[0] = HOLD;
            sort_main_flag[1] = HOLD;
            sort_main_flag[2] = HOLD;
            sort_main_flag[3] = HOLD;
            temp_float = select_size;
            k_small = k_large = k_mid = 0;
            for (i = 0; i <8; i++) meds[i] = 44444;
            for (i=0; i <= select_size; i++)
                        sort_input[i] = select_input[i];

            /* --------- find smallest k-th element -------- */
            k_value = select_size/10;
            RESET_TIMER;                 /* reset timer */
            timerStart = GET_TIMER;      /* get initial timer value */
            /* call parallel select */
            select_flag = _SELECT_DSP0(select_input, select_output,
                          select_size, k_value);
            /* get timer value */
            timerEnd = GET_TIMER;
            /* compute the elapsed time */
            savetime = ELAPSED_TIME(timerStart, timerEnd) * 1000000;
            k_small = gen_flag;

            /* --------- find median element -------- */
            k_value = select_size/2;
            RESET_TIMER;          /* reset the timer */
            timerStart = GET_TIMER;      /* get initial timer value */
            /* call parallel select */
            select_flag = _SELECT_DSP0(select_input, select_output,
                                select_size, k_value);
            timerEnd = GET_TIMER;   /* get elapsed time valued /
            /* compute the elapsed time */
            savetime2 = ELAPSED_TIME(timerStart, timerEnd) * 1000000;
            k_mid = gen_flag;

            /* --------- find smallest k-th element -------- */
            k_value = select_size - select_size/10;
            RESET_TIMER;                 /* reset timer */
            timerStart = GET_TIMER;      /* get initial timer value */
            /* call parallel select */
            select_flag = _SELECT_DSP0(select_input, select_output,
                                      select_size, k_value);
            timerEnd = GET_TIMER;        /* get elapsed time value */
            /* compute the elapsed time */
            savetime3 = ELAPSED_TIME(timerStart, timerEnd) * 1000000;
            k_large = gen_flag;


            /* ------- signal other DSPs to exit selection ------ */
```

```
                        if ( select_flag == DONE) main_flag = DONE;
                        for (i=1; i < SEL_CPU; i++) select_local_flag[i] = DONE;
                        for (i=1; i < SEL_CPU; i++) select_main_flag[i] = EXIT;

                        gen_flag = k_small;
                        select_wait_flag[0] = savetime;

                        gen_flag2 = k_mid;
                        select_wait_flag[1] = savetime2;

                        gen_flag3 = k_large;
                        select_wait_flag[2] = savetime3;
                        select_wait_flag[3] = 111111;

                        /* ------- signal all DSPs to exit --------- */
                        select_main_flag[MAIN] = EXIT;
                }       /* end if */
        }               /* end while */
}                       /* end of main */
```

```
/*-------------------------------------------------------------------
FILE:         dsp1.c
DESCRIPTION:  DSP1 parallel selection program
PROCESSORS:   4
DATE:         November 10, 1994
PROGRAMMER:   Nitin Lad
-------------------------------------------------------------------*/

#include "newdef.h"

int _SELECT_LOCAL_DSP1(s_inaddr, s_outaddr, s_size, del_x)
signed long *s_inaddr, *s_outaddr;
int s_size;
float del_x;
{
      int i, j, k, tot_seq_cpu, seq_size, m_index, n, N = 4;
      float x, temp;
      signed long median = 0;
      int first, last, itemp;


      /* wait until signal received from DSP0 */
      while ( (select_local_flag[dsp_id] != START ) );

      if ( s_size < 9)
            for ( i =0; i < s_size; i++) select_wait_flag[i] = s_inaddr[i];

      /* --- compute x from N = n **(x-1)   where 0 < x < 1 ----- */
      if ( loop_count == ZERO )
      {
            tot_seq_cpu = SEL_CPU;
            loop_count++;
      }
      else
      {
            tot_seq_cpu = floor( pow(s_size, del_x) );
      }

      if ( tot_seq_cpu >= SEL_CPU ) tot_seq_cpu = SEL_CPU;
      N = tot_seq_cpu;

      /* if the number of CPU to be utilized is less than 2 (= 1)
            then do nothing -- return */
      if ( N < 2 ) return DONE;

      /* Otherwiese: */
      /* ------ compute sequence size = n**(x) = Si ----- */
      seq_size =      ceil( pow(s_size, 1- del_x) );
      m_index = ceil( ceil( pow(s_size, 1- del_x) )/2 );

      first = dsp_id * seq_size;
      if ( N == 2 ) seq_size = s_size - first + 1;

      /*  sort and find median  */
      for ( i=0; i< seq_size; i++)
            t_addr[i] = s_inaddr[first+i];

      /* call sequential select */
      select_median[dsp_id] = seq_select( &t_addr[0], seq_size, m_index);
      select_local_flag[dsp_id] = DONE;

      return DONE;

}     /* end of select_local routine */
```

```
int _SELECT_DSP1(s_inaddr, s_outaddr, s_size)
signed long *s_inaddr, *s_outaddr;
int s_size;
{
      float del_x;
      unsigned long *l_addr = 0;

      /* sequence size is less then Q then do nothing and return */
      if ( s_size <= MIN ) return DONE;
      l_addr = malloc(sizeof(unsigned long));


      /* compute original 1 - x value */
      del_x =  log(SEL_CPU) / log(s_size);
      loop_count = ZERO;
      s_inaddr++;
      s_outaddr++;
      median_count = 0;

      /* wait until signal received from host */
      while ( !inter_flag );

      while ( (select_main_flag[dsp_id] != EXIT ) )
      {
            if ( select_local_flag[dsp_id] == START )
            {
                  s_size = select_local_size[MAIN];
                  _SELECT_LOCAL_DSP1(s_inaddr + *select_base,
                              s_outaddr, s_size, del_x);
            }     /* end if */
      }           /* end while */

      if (select_main_flag[dsp_id] == EXIT)
                              return DONE;
}                 /* enf of routine */


main()          /* _main:  dsp1.c  parallel_sort */
{
      int i, j=0, local_broad = 0;
      signed long first_index, last_index;
      signed long first_element, last_element;
      signed long local_size = 0;
      signed long xx, save_size = 0;
      int choose = 0;

      /* enable the DSP's interrupt facility */
      GIE_ON();

      while (1)
      {
            /* wait until signal is received from host */
            while ( (main_flag != GO) );

            /* call parallel selection routine */
            if ((main_flag == GO) && (select_flag == GO))
                  select_flag = _SELECT_DSP1(select_input,
                              select_output, select_size);

      }     /* end while */
}           /* end main */
```

```
/*-----------------------------------------------------------------
FILE:        dsp2.c
DESCRIPTION: DSP2 parallem selection program
PROCESSORS:  4
DATE:        November 10, 1994
PROGRAMMER:  Nitin Lad
-----------------------------------------------------------------*/

#include "newdef.h"

int _SELECT_LOCAL_DSP2(s_inaddr, s_outaddr, s_size, del_x)
signed long *s_inaddr, *s_outaddr;
int s_size;
float del_x;
{

        int i, j, k, tot_seq_cpu, seq_size, m_index, n, N = 4;
        float x, temp;
        signed long *x_addr, njit, median = 0;
        int first, last, itemp;

        /* wait until signal is received from DSP0 */
        while ( (select_local_flag[dsp_id] != START ) );

        /* --- compute x from N = n **(x-1)   where 0 < x < 1 ----- */
        if ( loop_count == ZERO )
        {
                tot_seq_cpu = SEL_CPU;
                loop_count++;
        }
        else
                tot_seq_cpu = floor( pow(s_size, del_x) );

        if ( tot_seq_cpu >= SEL_CPU ) tot_seq_cpu = SEL_CPU;
        N = tot_seq_cpu;

        /* if the number of CPU to be utilized is less than 3 (= 2)
                then do nothing -- return */
        if ( N < 3 ) return DONE;

        /* ------ compute sequence size n**(x) = Si ----- */
        seq_size =      ceil( pow(s_size, 1- del_x));
        m_index = ceil( ceil( pow(s_size, 1- del_x) )/2 );

        first = dsp_id * seq_size;
        if ( N == 3 ) seq_size = s_size - first + 1;

        /*  sort and find median */
        for ( i=0; i< seq_size; i++)
                t_addr[i] = s_inaddr[first+i];

        /* call sequential select */
        select_median[dsp_id] = seq_select( &t_addr[0], seq_size, m_index);
        select_local_flag[dsp_id] = DONE;

        return DONE;
}       /* end of routine */




int _SELECT_DSP2(s_inaddr, s_outaddr, s_size)
signed long *s_inaddr, *s_outaddr;
int s_size;
{
```

```
        float del_x;
        /* if the sequence size is <= Q,
              do nothing and return */
        if ( s_size <= MIN ) return DONE;

        /* compute original 1 -x value */
        del_x =  log(SEL_CPU) / log(s_size);
        loop_count = ZERO;

        s_inaddr++;
        s_outaddr++;


        /* wait until signal is received from host program */
        while ( !inter_flag );

        while ( (select_main_flag[dsp_id] != EXIT ) )
        {
              if ( select_local_flag[dsp_id] == START )
              {
                    /* get the sequence size value */
                    s_size = select_local_size[MAIN];
                    /* call parallel selection local routine */
                    _SELECT_LOCAL_DSP2(s_inaddr+ *select_offset,
                                 s_outaddr, s_size, del_x);
              }     /* end if */
        }           /* end while */

        if (select_main_flag[dsp_id] == EXIT)  return DONE;
}                   /* end of routine */


main()          /* _main:  dsp2.c  parallel_sort */
{
        int i, j=0, local_broad = 0, choose=0;
        signed long first_index, last_index;
        signed long local_size = 0;
        signed long xx, save_size = 0;

        /* enable the DSP's interrupt facility */
        GIE_ON();

        while (1)
        {
              /* wait until a signal is received from host */
              while ( (main_flag != GO) );

              /* call parallel selection routine */
              if ((main_flag == GO) && (select_flag == GO))
                    select_flag = _SELECT_DSP2(select_input,
                                 select_output, select_size);
        }     /* end while */
}             /* end main  */
```

```
/*-----------------------------------------------------------------
FILE:         dsp3.c
DESCRIPTION:  DSP3 parallel selection program
PROCESSORS:   4
DATE:         November 10, 1994
PROGRAMMER:   Nitin Lad
-----------------------------------------------------------------*/

#include "newdef.h"

int _SELECT_LOCAL_DSP3(s_inaddr, s_outaddr, s_size, del_x)
signed long *s_inaddr, *s_outaddr;
int s_size;
float del_x;
{

	int i, j, k, tot_seq_cpu, seq_size, m_index, n, N = 4;
	float x, temp;
	signed long *x_addr, njit, median = 0;
	int first, last, itemp;

	/* wait until signal is received from the host */
	while ( (select_local_flag[dsp_id] != START ) );

	/* --- compute x from N = n **(x-1)   where 0 < x < 1 ----- */
	if ( loop_count == ZERO )
	{
		tot_seq_cpu = SEL_CPU;
		loop_count++;
	}
	else
		tot_seq_cpu = floor( pow(s_size, del_x) );

	if ( tot_seq_cpu >= SEL_CPU ) tot_seq_cpu = SEL_CPU;
	N = tot_seq_cpu;

	/* if the number of CPU to be utilized is less than 4 (= 3)
		then do nothing -- return */
	if (N < 4 ) return DONE;

	/* ------ compute sequence size n**(x) = Si ----- */
	seq_size =      ceil( pow(s_size, 1- del_x) );
	m_index = ceil( ceil( pow(s_size, 1- del_x) )/2 );

	first = dsp_id * seq_size;
	seq_size = s_size - first + 1;

	m_index = ceil( seq_size/2 );

	/*  sort and find median */
	for ( i=0; i< seq_size; i++)
		t_addr[i] = s_inaddr[first+i];

	/* call sequential select */
	select_median[dsp_id] = seq_select( &t_addr[0], seq_size, m_index);
	select_local_flag[dsp_id] = DONE;
	return DONE;
}



int _SELECT_DSP3(s_inaddr, s_outaddr, s_size)
signed long *s_inaddr, *s_outaddr;
int s_size;
{
```

```
        float del_x;

        /* if the sequence size is <= Q,
              do nothing and return */
        if ( s_size <= MIN ) return DONE;

        /* compute original 1 -x value */
        del_x =  log(SEL_CPU) / log(s_size);
        loop_count = ZERO;
        s_inaddr++;
        s_outaddr++;


        /* wait until a signal is received from the host */
        while ( !inter_flag );

        while ( (select_main_flag[dsp_id] != EXIT ) )
        {
              if ( select_local_flag[dsp_id] == START )
              {
                    /* get the sequence size */
                    s_size = select_local_size[MAIN];
                    /* call parallel selection local routine */
                    _SELECT_LOCAL_DSP3(s_inaddr+ *select_offset,
                                s_outaddr, s_size, del_x);
              }     /* end if */
        }           /* end while */

        if ( select_main_flag[dsp_id] == EXIT ) return DONE;
}                   /* end of routine */



main()        /* _main:  dsp3.c  parallel_sort */
{
        int i, j=0, local_broad = 0;
        signed long first_index, last_index, xx;
        int choose, save_size;

        /* enable the DSP's interrupt facility */
        GIE_ON();

        while (1)
        {
              /* wait until a signal is received from the host */
              while ( main_flag != GO);

              /* call parallel select  routine */
              if ((main_flag == GO) && (select_flag == GO))
              {
                    select_flag = _SELECT_DSP3(select_input,
                                select_output, select_size);
              }     /* end if */
        }           /* end while */
}                   /* end main */
```

APPENDIX B

PARALALEL MERGE DSP PROGRRMS

```
/*---------------------------------------------------------------

FILE:           dsp0.c
DESCRIPTION: DSP1 parallem merge program
PROCESSORS:     4
DATE:           June 4, 1995
PROGRAMMER:     Nitin Lad
---------------------------------------------------------------*/

/*    Using data array in local memory */

#include "newdef.h" /* Including header file */

int _MERGE_DSP0(dataA, dataB, dataC, sizeA, sizeB)
signed long *dataA, *dataB, *dataC;
int sizeA, sizeB;
{
      int i, j, k, tot_seq, seq_size, n, N = 4;
      int first, last, itemp, flag = FAIL;
      signed long njit, median = 0;
      int quad[5], e, f, g, h, dela, delb;
      int w, z, dsp, check = 0, lsizea, lsizeb;
      double temp, temp2;
      unsigned long timerStart=0, timerEnd=0;
      float elapsed_time = 0.0;

      *x = *y = 0;

      GIE_ON();
      SET_PERIOD(0xffffffff);          /* set timer periof */


      while ( !inter_flag );

      dataA++;  dataB++;  dataC++;

      for (i=0; i < sizeA; i++) mdataA[i] = dataA[i];
      for (i=0; i < sizeB; i++) mdataB[i] = dataB[i];


/*---------------------------------------------------------------
                              Reset DSP timer
---------------------------------------------------------------*/
      RESET_TIMER;
      timerStart = GET_TIMER;          /* Read currect time */

      e = g = 1;
      f = sizeA;  h =sizeB;
      dela = delb = 0;

      for ( j = 1; j <= (log(N)/log(2)); j++)
      {
            i = 1;
            two_seq_med( &mdataA[e+dela-1], &mdataB[g+delb-1],
                                 f-dela, h-delb, x, y, med);
```

```
        *x = *x + dela;
        *y = *y + delb;

        /* Used for debugging purpose */
        check = 0;
        if ( check == 1)
        {
              gen_flag = *x;
              gen_flag2 = *y;
        }


/* step 1.2.2: Compute four pointers p1, p2, q1, q2  */
        *p1 = *p2 = *q1 = *q2 = 1122;

        find_merge_ptrs( mdataA, mdataB, sizeA, sizeB,
                                x, y, med, p1, p2, q1, q2);

        /* Used for debugging purpose */
        check = 0;
        if ( check == 1)
        {
              gen_flag = *x;
              gen_flag2 = *y;
        }


        /* If this is SECOND iteration than
              wait until DSP2 is ready  */
        if ( j == 1)
              while ( merge_local_flag[1] != READY);
        else if ( j >= 2)
              while ( merge_local_flag[1] != NEXT);


/* --------- communicate quadruple to DSP2 ---------*/
        /*    P2 = P(2i)  <---------- (q1, f, q2, h)  */
        dsp = 2*4;
        merge_quad[dsp+1] = *q1;
        merge_quad[dsp+2] = f;
        merge_quad[dsp+3] = *q2;
        merge_quad[dsp+4] = h;
        merge_local_flag[1] = GO;

        /* compute the new quadruple for next iteration */
        /*    P1 = P(2i-1)  <---------- (e, p1, g, p2)      */
        f = *p1;
        h = *p2;

}                     /* end of for loop */


lsizea = f - e + 1;
if ( lsizea <= 0)
        lsizea = 0;

lsizeb = h - g + 1;

if ( lsizeb <= 0)
        lsizeb = 0;

seq_merge(&mdataA[e-1], &mdataB[g-1],
                  mdataC, lsizea, lsizeb);

/* wait until all FOUR DSPs are done */
i = 1;
flag = WAIT;
```

```
        while (flag != DONE)
        {

                if ( merge_local_flag[i] == DONE)
                            i++;

                if ( i >= 4 )
                            flag = DONE;
        }       /* end of while loop */

/*-----------------------------------------------------------
                Read current time and compute elapsed time.
-----------------------------------------------------------*/
        timerEnd = GET_TIMER;      /* Read current time */
        elapsed_time =  ELAPSED_TIME(timerStart, timerEnd);

        for (i=0; i <(lsizea+lsizeb); I++)
                    dataC[i] = mdataC[i];

        /* Increment Error Count if there is error in DSP0
                                merge outputs ---- */
        j = 0;
        for ( i =0; i < (sizeA+sizeB); i++)
                if ( dataC[i] != i)
                    j++;

/*----------------------------------------------------------
                        Recompute elapsed time.
-----------------------------------------------------------*/
                gen_flag = j;
                gen_flag2 =  ELAPSED_TIME(timerStart, timerEnd) * 1000000;
                gen_flag3= elapsed_time * 1000000;
                gen_flag4 =  ELAPSED_TIME(timerStart, timerEnd) * 1000000;

                return DONE;

}               /* end of routine */




main()
{
        int i, local_broad = 0;

        GIE_ON();     /* Enable DSP Interrupt facility */

        while (1)
        {

                /*      wait until host program has completed the
                        downloading of the data into all DSPs  */
                while ( (main_flag != GO) );

               if ((main_flag == GO) && (merge_flag == GO))
               {
                merge_flag = _MERGE_DSP0(dataA, dataB,
                                        dataC, sizeA, sizeB);
                    if ( merge_flag == DONE)
                          main_flag = DONE;

               }                  /* end if */
        }                         /* end while loop */

}                                 /* end of main */
```

```
/*----------------------------------------------------------------
FILE:               dsp1.c
DESCRIPTION:        DSP1 parallel merge program
PROCESSORS:         4
DATE:               June 4, 1995
PROGRAMMER:         Nitin Lad
---------------------------------------------------------------*/


#include "newdef.h"


int _MERGE_DSP1(dataA, dataB, dataC, sizeA, sizeB)
signed long *dataA, *dataB, *dataC;
signed long sizeA, sizeB;
{
      int i, j, k, n, N = 4, iteration;
      int first, last, itemp, flag = FAIL;
      int lsizea, lsizeb, lsizec;
      int w, z, e, f, g, h, dsp, dela, delb, check, offa, offb, offc;
      double temp, temp2;
      float temp3;
      signed long njit, median = 0;

      dela = delb = 0;
      *x = *y = *p1 = *p2 = *q1 = *q2 = 0;
      gen_flag = gen_flag2 = gen_flag3 = gen_flag4 = JUNK;
      merge_local_flag[dsp_id] = WAIT;

      while ( !inter_flag );

      dataA++;
      dataB++;
      dataC++;

      /* copy data from SHARED memory to LOCAL DSP memory */
      for (i =0; i < sizeA; I++)
            mdataA[i] = dataA[i];
      for (i =0; i < sizeB; I++)
            mdataB[i] = dataB[i];

      merge_local_flag[dsp_id] = READY;

      /* wait until DSP0 communicates the quadruple */
      while( merge_local_flag[dsp_id] != GO);
      dsp = 2*4;

      /* reads the quadruple received from DSP0 */
      gen_flag  = e = merge_quad[dsp+1];
      gen_flag2 = f = merge_quad[dsp+2];
      gen_flag3 = g = merge_quad[dsp+3];
      gen_flag4 = h = merge_quad[dsp+4];
      merge_local_flag[dsp_id] = NEXT;

      offa = e - 1;  offb = g - 1;

      two_seq_med( &mdataA[e+dela-1], &mdataB[g+delb-1],
                   f-e+1, h-g+1, x, y, med);

      *x = *x + dela + offa;
      *y = *y + delb + offb;

      check = 0;
      if ( check == 1)
      {
            gen_flag = *x;
            gen_flag2 = *y;
```

```
}

/* step 1.2.2:  Compute four pointers p1, p2, q1, q2  */
*p1 = *p2 = *q1 = *q2 = 1122;

find_merge_ptrs(mdataA, mdataB, sizeA, sizeB,
             x, y, med, p1, p2, q1, q2);

/* Used for debugging purpose */
check = 0;
if ( check == 8)
{
       gen_flag  = *p1;
       gen_flag2 = *p2;
       gen_flag3 = *q1;
       gen_flag4 = *q2;
}


dsp = 4*4;
/* communicate the quadruple to DSP4 */
/*     P4 = P(2i)  <---------- (q1, f, q2, h)  */

while ( merge_local_flag[3] != READY);

merge_quad[dsp+1] = *q1;
merge_quad[dsp+2] = f;
merge_quad[dsp+3] = *q2;
merge_quad[dsp+4] = h;
merge_local_flag[3] = GO;

/* Used for debuggin purpose */
check = 0;
if ( check == 4)
{
       gen_flag  = *q1;
       gen_flag2 = f;
       gen_flag3 = *q2;
       gen_flag4 = h;
}


/* communicate the quadruple to DSP3 */
/*      P3 = P(2i-1)  <---------- (e, p1, g, p2)        */
dsp = 3*4;
while ( merge_local_flag[2] != READY);

merge_quad[dsp+1] = e;
merge_quad[dsp+2] = *p1;
merge_quad[dsp+3] = g;
merge_quad[dsp+4] = *p2;
merge_local_flag[2] = GO;

/* waite until DSP1 communicates next set of quadruple */
while( merge_local_flag[dsp_id] != GO);
dsp = 2*4;

/* reads the quadruple received from DSP1 */
gen_flag  = e = merge_quad[dsp+1];
gen_flag2 = f = merge_quad[dsp+2];
gen_flag3 = g = merge_quad[dsp+3];
gen_flag4 = h = merge_quad[dsp+4];

/* Used for debuggin purpose */
check = 0;
if ( check == 3)
```

```
        {
                gen_flag  = e;
                gen_flag2 = f;
                gen_flag3 = g;
                gen_flag4 = h;
        }

        lsizea = f - e + 1;
        if ( lsizea <= 0)
                lsizea = 0;

        lsizeb = h - g + 1;
        if ( lsizeb <= 0)
                lsizeb = 0;

        offc = e - 1 +  g - 1;

        seq_merge(&mdataA[e-1], &mdataB[g-1], mdataC, lsizea, lsizeb);

        merge_local_flag[dsp_id] = DONE;

        for (i=0; i < (lsizea+lsizeb); I++)
                dataC[offc+i] = mdataC[i];

        return DONE;
}       /* end of merge routine */



main()
{
        int i, local_broad = 0;

        GIE_ON();     /* Enable DSP interrupt facility */


        while (1)
        {
                /*    wait until host program has completed the
                      downloading of the data into all DSPs   */
        while ( (main_flag != GO) );

              if ((main_flag == GO) && (merge_flag == GO))
              {
               merge_flag = _MERGE_DSP1(dataA, dataB,
                                        dataC, sizeA, sizeB);
                      if ( merge_flag == DONE) main_flag = DONE;
              }
         }

}       /* end of main */
```

```
/*-------------------------------------------------------------------
FILE:                dsp2.c
DESCRIPTION:         DSP1 parallel merge program
PROCESSORS:          4
DATE:                June 4, 1995
PROGRAMMER:          Nitin Lad
------------------------------------------------------------------*/

#include "newdef.h"


int _MERGE_DSP2(dataA, dataB, dataC, sizeA, sizeB)
signed long *dataA, *dataB, *dataC;
signed long sizeA, sizeB;
{
      int i, j, k, n, N = 4;
      int first, last, itemp, flag = FAIL;
      int w, z, e, f, g, h, dsp, check;
      int offc, lsizea, lsizeb;
      signed long njit, median = 0;
      float *del_x;
      double temp, temp2;

      *x = *y = 0;

      merge_local_flag[dsp_id] = WAIT;


      while ( !inter_flag );

      dataA++;
      dataB++;
      dataC++;

      /* copy data from SHARED memory to LOCAL DSP memory */
      for (i =0; i < sizeA; I++)
            mdataA[i] = dataA[i];

      for (i =0; i < sizeB; I++)
            mdataB[i] = dataB[i];

      merge_local_flag[dsp_id] = READY;


      /* wait until DSP1 communicates the quadruple */
      while( merge_local_flag[dsp_id] != GO);
      dsp = 3*4;

      /* reads the quadruple received from DSP1 */
      gen_flag  = e = merge_quad[dsp+1];
      gen_flag2 = f = merge_quad[dsp+2];
      gen_flag3 = g = merge_quad[dsp+3];
      gen_flag4 = h = merge_quad[dsp+4];


      /* Used for debugging purpose */
      check = 3;

      if ( check == 3)
      {
            gen_flag  = e;
            gen_flag2 = f;
            gen_flag3 = g;
            gen_flag4 = h;
      }
```

```
        lsizea = f - e + 1;
        if ( lsizea <= 0)
               lsizea = 0;

        lsizeb = h - g + 1;
        if ( lsizeb <= 0)
               lsizeb = 0;

        offc = e - 1 +  g - 1;
        gen_flag = offc;
        gen_flag2 = offc+lsizea+lsizeb;


        seq_merge(&mdataA[e-1], &mdataB[g-1], mdataC, lsizea, lsizeb);

        merge_local_flag[dsp_id] = DONE;

        for (i=0; i < (lsizea+lsizeb); I++)
               dataC[offc+i] = mdataC[i];

        return DONE;
}



main()
{
        int i, local_broad = 0;

        GIE_ON();     /* Enable the DSP interrup facility */

        while (1)
        {
               /*    wait until host program has completed the
                     downloading of the data into all DSPs   */
             while ( (main_flag != GO) );

             if ((main_flag == GO) && (merge_flag == GO))
             {
                     merge_flag = _MERGE_DSP2(dataA, dataB,
                                         dataC, sizeA, sizeB);

                     if (merge_flag == DONE)
                          main_flag = DONE;

             }            /* end if */
          }               /* end while loop */

}                         /* end of main */
```

```
/*-------------------------------------------------------------
FILE:         dsp3.c
DESCRIPTION: DSP1 parallel merge program
PROCESSORS:  4
DATE:         June 4, 1995
PROGRAMMER:  Nitin Lad
-------------------------------------------------------------*/

#include "newdef.h"

int _MERGE_DSP3(dataA, dataB, dataC, sizeA, sizeB)
signed long *dataA, *dataB, *dataC;
signed long sizeA, sizeB;
{
      int i, j, k, check, n, N = 4;
      int first, last, itemp, flag = FAIL;
      signed long njit, median = 0;
      int w, z, e, f, g, h, dsp;
      int offc, lsizea, lsizeb;
      double temp, temp2;

       *x = *y = 0;
       merge_local_flag[dsp_id] = WAIT;

      while ( !inter_flag );

      dataA++;
      dataB++;
      dataC++;

      /* copy data from SHARED memory to LOCAL DSP memory */
      for (i =0; i < sizeA; i++)
                  mdataA[i] = dataA[i];

      for (i =0; i < sizeB; I++)
                  mdataB[i] = dataB[i];

      merge_local_flag[dsp_id] = READY;

      /* wait until DSP2 communicates the quadruple */
      while( merge_local_flag[dsp_id] != GO);
      dsp = 4*4;

      /* reads the quadruple received from DSP1 */
      gen_flag  = e = merge_quad[dsp+1];
      gen_flag2 = f = merge_quad[dsp+2];
      gen_flag3 = g = merge_quad[dsp+3];
      gen_flag4 = h = merge_quad[dsp+4];

      check = 3;
      if ( check == 3)
       {
             gen_flag  = e;
            gen_flag2 = f;
            gen_flag3 = g;
            gen_flag4 = h;
       }

      lsizea = f - e + 1;
      if ( lsizea <= 0)
             lsizea = 0;

      lsizeb = h - g + 1;
      if ( lsizeb <= 0)
             lsizeb = 0;

       offc = e - 1 +  g - 1;
```

```
        gen_flag = offc;
        gen_flag2 = offc + lsizea + lsizeb;

        seq_merge(&mdataA[e-1], &mdataB[g-1], mdataC, lsizea, lsizeb);

        merge_local_flag[dsp_id] = DONE;

        for (i=0; i < (lsizea+lsizeb); i++)
                dataC[offc+i] = mdataC[i];

        return DONE;
}


main()
{
        int i, local_broad = 0;

        GIE_ON();     /* Enable the DSP interrupt facility */

        while (1)
        {
                /* wait until host program has completed the
                        downloading of the data into all DSPs */
                while ( main_flag != GO);


                if ((main_flag == GO) && (merge_flag == GO))
                {
                        merge_flag = _MERGE_DSP3(dataA, dataB,
                                                dataC, sizeA, sizeB);

                        if ( merge_flag == DONE) main_flag = DONE;

                }                       /* end if */
        }                               /* end of while loop */
}                                       /* end of main */
```

APPENDIX C

PARALLEL SORTING PROGRAMS

```
/*-----------------------------------------------------------------
FILE:         dsp0.c
DESCRIPTION: DSP1 parallel sort program
PROCESSORS:   4
DATE:         June 4, 1995
PROGRAMMER:   Nitin Lad
---------------------------------------------------------------*/

#include "newdef.h"


/* NOTE: Only the main program included here.  All selection routine are
included in Appendix A */


main()          /* _main:  dsp0.c  parallel_sort */
{
      int i, j=0, local_broad = 0, k_value, tempa;
      signed long tempb;
      unsigned long timerStart, timerEnd, temp;
      float elapsed_time = 0.0, temp_float;
      signed long last_index;
      signed long xx, save_size;
      int choose = 0;

      /* enable DSP interrupt facility */
      GIE_ON();
      SET_PERIOD(0xffffffff);         /* set timer periof */

      while (1)
      {
            /* wait until a signal received from the host */
            while ( (main_flag != GO) );

            if ((main_flag == GO) && (select_flag == GO))
            {
                  /* initialize flags */
                  select_wait_flag[0] = JUNK;
                  select_wait_flag[1] = JUNK;
                  select_wait_flag[2] = JUNK;
                  select_wait_flag[3] = JUNK;
                  sort_main_flag[0] = HOLD;
                  sort_main_flag[1] = HOLD;
                  sort_main_flag[2] = HOLD;
                  sort_main_flag[3] = HOLD;
                  temp_float = select_size;

                  for (i = 0; i <8; i++) meds[i] = 44444;
                  /* Copy the sequence */
                  for (i=0; i <= select_size; i++)
                              sort_input[i] = select_input[i];

                  RESET_TIMER;              /* reset timer */
                  timerStart = GET_TIMER;   /* Get start timer value */
```

```
/* call parallel select to find 1st median value */
meds[3] = k_value = ceil( temp_float/K_VALUE);
select_flag = _SELECT_DSP0(select_input, select_output,
                    select_size, k_value);
meds[0] = gen_flag;


/* call parallel select to find 2nd median value */
meds[4] = k_value = ceil( 2*temp_float/K_VALUE);
select_flag = _SELECT_DSP0(select_input, select_output,
                    select_size, k_value);
meds[1] = gen_flag;

/* call parallel select to find 3rd median value */
meds[5] = k_value = ceil( 3*temp_float/K_VALUE);
select_flag = _SELECT_DSP0(select_input, select_output,
                     select_size, k_value);
meds[2] = gen_flag;

/* indexes */
select_wait_flag[0] = meds[3];
select_wait_flag[1] = meds[4];
select_wait_flag[2] = meds[5];
select_wait_flag[3] = 7777777;

/* medians */
gen_flag  = meds[0];
gen_flag2  = meds[1];
gen_flag3  = meds[2];
gen_flag4  = 77777;

/* generate FOUR sequences based on THREE medians */


/* ------- signal other DSPs to exit selection ------ */
if ( select_flag == DONE) main_flag = DONE;
for (i=1; i < SEL_CPU; i++) select_local_flag[i] = DONE;
for (i=1; i < SEL_CPU; i++) select_main_flag[i] = EXIT;

sort_main_flag[1] = GO;
sort_main_flag[2] = GO;
sort_main_flag[3] = GO;

last_index = meds[3]+1;
for(j=0; j< last_index; j++)
      t_addr[j] = sort_input[j];

/*  ------- sort the sequence ----------------- */
xx = 0;
save_size = j;
/* call local fast_sort routine */
fast_sort(xx, save_size);

for(j=0; j< last_index; j++)
      sort_input[j]= t_addr[j];

/* wait until all other DSPs are done */
while( sort_main_flag[1] != DONE);
while( sort_main_flag[2] != DONE);
while( sort_main_flag[3] != DONE);


timerEnd = GET_TIMER;      /* Get timer value */
select_wait_flag[0] = ELAPSED_TIME(timerStart,
timerEnd) * 1000000;       /* compute the elapsed time */
```

```
                    /* Copy the sorted sequence to the output array */
                    for (i=0; i <= select_size; i++)
                                select_output[i] = sort_input[i];

                    /* ------- signal all DSPs to exit --------- */
                    select_main_flag[MAIN] = EXIT;
              }     /* end if */
       }            /* end while */
}                   /* end of main */



/*---------------------------------------------------------------
FILE:         dsp1.c
DESCRIPTION:  DSP1 parallel sort program
PROCESSORS:   4
DATE:         June 4, 1995
PROGRAMMER:   Nitin Lad
---------------------------------------------------------------*/

#include "newdef.h"


main()          /* _main:  dsp1.c  parallel_sort */
{
       int i, j=0, local_broad = 0;
       signed long first_index, last_index;
       signed long first_element, last_element;
       signed long local_size = 0;
       signed long xx, save_size = 0;
       int choose = 0;

       GIE_ON();    /* enable the DSP's interrupt facility */

       while (1)
       {
              /* wait until a signal is received from the host */
              while ( (main_flag != GO) );

              /* call local parallel select to find medians */
              if ((main_flag == GO) && (select_flag == GO))
                     select_flag = _SELECT_DSP1(select_input,
                                   select_output, select_size);

              if ((main_flag == GO) && (sort_main_flag[1] == GO))
              {
                     /* Get the first and last index for the sequence
                            to be sorted */
                     first_index = meds[3]+1;
                     last_index = meds[4]+1;

                     /* read the unsorted sub-sequence from shared memory */
                     for(i=first_index; i < last_index; i++)
                     {
                            t_addr[j] = sort_input[i];
                            j++;
                     }

                     /* sort the sequence */

                     xx = 0;
                     save_size = j;
                     /* call fast sort the sort the subsequence */
                     fast_sort(xx, save_size);
```

```
                    /* write the sorted sequence to shared memory */
                    for (i=0; i < j; i++)
                    {
                          sort_input[first_index+i] = t_addr[i];
                    }

                    sort_main_flag[1] = DONE;
              }     /* end if */

              if ((sort_main_flag[1] == DONE) && (select_flag ==DONE))
                    main_flag = DONE;
       }            /* end while */
}                   /* end of main */




/*------------------------------------------------------------------
FILE:         dsp2.c
DESCRIPTION: DSP2 parallel sort program
PROCESSORS:   4
DATE:         June 4, 1995
PROGRAMMER:   Nitin Lad
------------------------------------------------------------------*/

#include "newdef.h"


main()          /* _main:  dsp2.c  parallel_sort */
{
       int i, j=0, local_broad = 0, choose=0;
       signed long first_index, last_index;
       signed long local_size = 0;
       signed long xx, save_size = 0;

       GIE_ON();     /* enable the DSP interrupt */

       while (1)
       {
              /* wait until a signal is received from the host */
              while ( (main_flag != GO) );

              /* call parallel select to find median values */
              if ((main_flag == GO) && (select_flag == GO))
              select_flag = _SELECT_DSP2(select_input, select_output,
                                         select_size);

              if ((main_flag == GO) && (sort_main_flag[2] == GO))
              {
                    /* Get first and last index of the sequence
                          to be sorted */
                    first_index = meds[4]+1;
                    last_index = meds[5]+1;

                    /* read the unsorted sub-sequence from shared memory */
                    for(i=first_index; i < last_index; i++)
                    {
                          t_addr[j] = sort_input[i];
                          j++;
                    }

                    /* sort the sequence */
                    xx = 0;
                    save_size = j;
                    fast_sort(xx, save_size);
```

```
			/* write the sorted sub-sequence to shared memory */
			for (i=0; i < j; i++)
			{
				sort_input[first_index+i] = t_addr[i];
			}

			sort_main_flag[2] = DONE;
		}	/* end if */

		if ((sort_main_flag[2] == DONE) && (select_flag ==DONE))
			main_flag = DONE;
	}		/* end while */
}			/* end main */
```

```
/*---------------------------------------------------------------
FILE:        dsp3.c
DESCRIPTION: DSP3 parallel sort program
PROCESSORS:  4
DATE:        June 4, 1995
PROGRAMMER:  Nitin Lad
---------------------------------------------------------------*/

#include "newdef.h"


main()		/* _main:  dsp3.c  parallel_sort */
{
	int i, j=0, local_broad = 0;
	signed long first_index, last_index, xx;
	int choose, save_size;

	GIE_ON();	/* enable DSP interrupt */

	while (1)
	{
		/* wait until a signal is received from the host */
		while ( main_flag != GO);

		/* call parallel select to find medians */
		if ((main_flag == GO) && (select_flag == GO))
			select_flag = _SELECT_DSP3(select_input,
					select_output, select_size);

		if ((main_flag == GO) && (sort_main_flag[3] == GO))
		{
			/* Get the first and last index value of the
				sequence to be sorted from DSP0 */
			first_index = meds[5]+1;
			last_index = select_size+1;

			/* read the unsorted sub-sequence from shared memory */
			for(i=first_index; i < last_index; i++)
			{
				t_addr[j] = sort_input[i];
				j++;
			}

			/* sort the sequence */
			xx = 0;
			save_size = j;
			fast_sort(xx, save_size);
```

```
                        /* write the sorted sequence to shared memory */
                        for (i=0; i < j; i++)
                        {
                              sort_input[first_index+i] = t_addr[i];
                        }


                        sort_main_flag[3] = DONE;
                  }     /* end if */
                        if ((sort_main_flag[3] == DONE) && (select_flag ==DONE))
                        main_flag = DONE;
            }           /* end while */
      }                 /* end main */



      1
```

INCLUDE FILE

```
/*------------------------------------------------------------------
FILE:         newdef.h
DESCRIPTION: Include file for parallel selection program
PROCESSORS:  4
DATE:         November 10, 1994
PROGRAMMER:  Nitin Lad
------------------------------------------------------------------*/

#include <math.h>
#include <stdlib.h>
#include "/usr/local/hydra/include/hydra.h"

/*-----------------------------------------------------------------------
                           CONSTANT declarations
-----------------------------------------------------------------------*/
#define HOLD                0
#define MAIN                0
#define ALL                 0
#define ZERO                0
#define GO                  1
#define START               1
#define READY               7
#define MAXX                6000

#define CONTINUE            20
#define EXIT                17

#define DONE                9
#define NOT_DONE            6
#define FAIL                -9
#define C                   0
#define CLEAR               0
#define JUNK                77777777

#define CPU                 8
#define MIN                 4
#define SEL_CPU             1      /* Totoal CPU use for par_select */
#define NEXT_CPU            2

#define GET_READY           44
#define START_BROAD         47
#define STACK_SIZE          200          /* stack size */
#define MAX                 4000         /* Max array size limit */
#define K_VALUE             4            /* default k-th value */

/*-----------------------------------------------------------------------
                    MEMORY ADDRESS CONSTANT declarations
-----------------------------------------------------------------------*/
/* ----- Shared Memory for Board 1 -  For control information ---- */
#define CRAM0             0x8d000000
#define CRAM0_SIZE        0xfff          /* 4K   */

/* ---- Shared Memory for Board 1 -  General Data storage use ----- */
#define CRAM1             0x8d000fff     /* starting address */
#define CRAM1_SIZE        0xfefc0-0xfff  /* 1020K-4K */

/* -------------  Shared Memory for Board 2 -------------------- */
#define CRAM0_B           0xa0000000
#define CRAM0_B_SIZE      0xfefc0        /* 1020K */

/* ----------  Internal LOCAL RAM BLK0 & BLK1------------------  */
#define INT_RAM           0x2ff800
#define INT_RAM_SIZE      0x800          /*  2K */

/* ----------  External LOCAL RAM  ----------------------------- */
```

```
#define EXT_LRAM1       0x40001200
#define EXT_LRAM1_SIZE  0x2dff         /* 59.5K  */

/* ----------- External LOCAL RAM Supplemental ------------------ */
#define EXT_LRAM2       0xc0000000
#define EXT_LRAM2_SIZE  0xffff         /* 16K  */


#define SHARED_ADDR     0x8d000000     /* starting address      */
#define SHARED_SIZE     0xfefbf        /* 1020K                 */

#define SHARED_ADDR2    0xa0000000     /* starting address Board 2 */
#define SHARED_SIZE2    0xfefbf        /* 1020K            Board 2 */

#define LOCAL_RAM       0x2ff800       /* starting LRAM addres */
#define LOCAL_RAM_SIZE  0x7ff          /* 2Kwords               */

/* following is reserved for text of programs */
#define LOCAL_ADDR      0x40001200     /* starting address      */
#define LOCAL_SIZE      0xedff         /* 59.5K                 */

/*-----------------------------------------------------------------
                    TIMER SETUP VARIABLE DECLARATIONS
-----------------------------------------------------------------*/
#define ELAPSED_TIME( start, end )     (((end) - (start))*0.0000001)
#define GET_TIMER   (*(unsigned long *)0x00100024)
#define RESET_TIMER (*(unsigned long *)0x00100020 |= 960)
#define SET_PERIOD(X)     (*(unsigned long *)0x00100028 = (unsigned long) X)


/*-----------------------------------------------------------------
             General local input/output array declarations
-----------------------------------------------------------------*/
signed long *in_addr = (signed long *)INT_RAM;
signed long *out_addr = (signed long *)(INT_RAM+INT_RAM_SIZE/2);

/*-----------------------------------------------------------------
                        Local array declarations
-----------------------------------------------------------------*/
signed long *_small = (signed long *)EXT_LRAM1;
signed long *_equal = (signed long *)(EXT_LRAM1+5000);
signed long *_big = (signed long *)(EXT_LRAM1+10000);

signed long *small = (signed long *)EXT_LRAM1;
signed long *equal = (signed long *)(EXT_LRAM1+5000);
signed long *big = (signed long *)(EXT_LRAM1+10000);

signed long *seq1 = (signed long *)EXT_LRAM1;
signed long *seq2 = (signed long *)(EXT_LRAM1+5000);
signed long *seq3 = (signed long *)(EXT_LRAM1+10000);
signed long *seq4 = (signed long *)(EXT_LRAM1+15000);

/* - Local Memory Array: used by seq_select & fast_sort -- */
signed long *t_addr = (signed long *)(EXT_LRAM1+20000);
signed long *loc_seq = (signed long *)(INT_RAM+210);

signed long *broad_input = (signed long *)INT_RAM;
signed long *broad_output = (signed long *)(INT_RAM+INT_RAM_SIZE/2);

/*---------- Local median array size 10 used by fast_sort */
signed long *medM   = (signed long *)INT_RAM;


/************** EXternal Local memory for LOCAL VARIABLES *******/

int D_SIZE,         /* size of sequence */
dsp_id;             /* DSP identification */
```

```
/*---------------------------------------------------------------
      Variables used to communication with other DSP
      and host programs
---------------------------------------------------------------*/
int main_flag = HOLD, select_flag, merge_flag;
int inter_flag = HOLD;    /* Signal flag */
int select_size;          /* size of selection sequence */
int tot_cpu = 0;
int k_value;              /* k-th value */
signed long select_med = 0;

signed long gen_flag = 0;
signed long gen_flag2 = 0;
signed long gen_flag3 = 0;
signed long gen_flag4 = 0;
signed long median_count = 0;


signed long stackA[STACK_SIZE];
signed long stackA2[STACK_SIZE];
int global_count = 0;
signed long f_index = 0;
signed long ssize = 0;
signed long f_size = 0;

unsigned sig_flag = CONTINUE;
signed long loop_count = 0;
unsigned long timerStart, timerEnd;


#define Q 4         /* Q value */
#define K 21        /* Default constant K */

/****************************************************************/
/*_______________GLOBAL_FLAGS/VARIABLES/ARRAY_declaration_________*/
/****************************************************************/

/*______NEXT BLOCK of GLOBAL MEMORY== CRAM0______    */
/*______select_median_size = 8 ___________          */
/*  Each processor stored median values here         */
/*  SIZE:    Next Eight long words                   */
signed long *select_median=(signed long *)(CRAM0);

/* Intermediate local shared mem. flags for selection routine */
signed long *select_local_flag = (signed long *)(CRAM0 +
                   8*sizeof(signed long));
/*---------------------------------------------------------------
      Shared-Memory flags usage for communications
---------------------------------------------------------------*/
/*______select_wait_flag_size = 8 __________ */
signed long *select_wait_flag = (signed long *)(CRAM0 +
                   16*sizeof(signed long));

/*______select_broad_flag_size = 8 __________ */
signed long *select_broad_flag = (signed long *)(CRAM0 +
                   24*sizeof(signed long));

/*______select_k = 2 _________________________ */
signed long *select_k = (signed long *)(CRAM0 +
                   32*sizeof(signed long));

/*______select_val_k_size = 2 ________________ */
signed long *select_val_k = (signed long *)(CRAM0 + 34*sizeof(signed long));

/*______select_local_size_size = 2 __________ */
```

```
signed long *select_local_size = (signed long *)(CRAM0 + 36*sizeof(signed
long));

/*______select_local_median_size = 10__________ */
signed long *select_local_median = (signed long *)(CRAM0 + 38*sizeof(signed
long));

/*______select_main_flag_size = 8______________ */
unsigned long *select_main_flag = (unsigned long *)(CRAM0 + 48*sizeof(unsigned
long));

/*______select_offset_size = 2_________________ */
unsigned long *select_offset = (unsigned long *)(CRAM0 + 56*sizeof(unsigned
long));
/*______select_base_size = 2___________________ */
unsigned long *select_base = (unsigned long *)(CRAM0 + 58*sizeof(unsigned
long));

/*______Shared_Median_Array_size = 8________________*/
unsigned long *meds = (unsigned long *)(CRAM0 + 60*sizeof(unsigned long));

/*______Sort_flag_size = 8______________________*/
unsigned long *sort_main_flag = (unsigned long *)(CRAM0 + 68*sizeof(unsigned
long));


/*---------------------------------------------------------------
              Shared-memory input/output arrays declaration
----------------------------------------------------------------*/
signed long *in_saddr=(signed long *)CRAM1;
signed long *out_saddr=(signed long *)(CRAM1+ CRAM1_SIZE/2);


/*---------------------------------------------------------------
       Shared-memory input/output arrays declaration for Parallel
       Selection
----------------------------------------------------------------*/
signed long *select_output=(signed long *)(CRAM1);   /* Global Addr. space */
signed long *select_input=(signed long *)(CRAM1+MAXX);
signed long *sort_input=(signed long *)(CRAM1+2*MAXX);
signed long *ssmall=(signed long *)(CRAM1+4*MAXX);
signed long *sequal=(signed long *)(CRAM1+6*MAXX);
signed long *sbig=(signed long *)(CRAM1+8*MAXX);


/*---------------------------------------------------------------
glob_push(): This local routine is used to push values onto a
             predefined stack in the sort operation */
----------------------------------------------------------------*/
int glob_pushA( adrA, offsetA)
signed long *adrA, *offsetA;
{
       int i, j,  k;
       if ( global_count == STACK_SIZE)
       {
              return 1;
       }
       else
       {
              stackA[global_count] = *adrA;
              stackA2[global_count] = *offsetA;
              global_count = global_count + 1;
              return 0;
       }      /* end if */
}             /* end of push routine*/
```

```
/*------------------------------------------------------------------
glob_pop():  This local routine is used to pull values from a
             predefined stack in the sort operation */
-------------------------------------------------------------------*/
int glob_popA( adrA, offsetA)
signed long *adrA, *offsetA;
{
      int i, j,  k;
      if (global_count == 0)
      {
            return 1;
      }
      else
      {
            global_count = global_count - 1;
            *adrA = stackA[global_count];
            *offsetA = stackA2[global_count];
            return 0;
      }     /* end if */
}           /* end of pop routine */




/*-------------------------------------------------------------
Sequential Sort():  Local routine to perform sequential
                          sort operations.
             S:     Starting of address of an input array
             Size:  Size of the array(number of elements)
--------------------------------------------------------------*/
signed long seq_sort(S, size)
signed long *S;
int size;
{
      int i, j, lim;
      signed long temp;
      for ( i= 0; i< size; i++)
            for ( j=i+1; j< size; j++)
                  if ( S[i] > S[j] )
                  {
                        temp = S[i];
                        S[i] = S[j];
                        S[j] = temp;
                  }

      if ( size > 1)
            lim = size/2;
      else
            return S[0];

      if ( size/2 > lim)
            return S[lim];
      else
            return S[lim-1];

}     /*end of bubble sort */



/*-----------------------------------------------------------------------
seq_select():       This routine is used to find a k-th smallest
                    element using a sequential technique.
```

```
-------------------------------------------------------------------*/
signed long seq_select(S, size, k) /* ----- sequential select ------- */
signed long *S;
int size, k;
{
      int i, j = 0, a, first, last,  t_sm, t_eq, t_big;
      signed long median = 0;
      signed long seq_select( signed long *S, int size, int k);

      for ( i = 0; i < size; i++) small[i] = equal[i] = big[i] = 0;

      if ( size < Q )
      {
            median =  seq_sort( &S[0], size );
            return S[k-1];
      }
      else
      {
            /* ----- divide and sort the sequences ---- */
            for ( a = 0; a < size; a += Q )
            {
                  first = a;
                  last = a + Q;
                  if ( last > size )  last = size;
                  equal[j] = seq_sort( &S[first], last - first );
                  j++;
            }     /* end for */

            median =  seq_sort( &equal[0], j );
            t_big = t_eq = t_sm = 0;

            for ( i = 0; i < size; i++)
            {
                  if ( S[i] > median )
                  {
                        big[t_big] = S[i];
                        t_big++;
                  }
                  else if ( S[i] == median )
                  {
                        equal[t_eq] = S[i];
                        t_eq++;
                  }
                  else if ( S[i] < median )
                  {
                        small[t_sm] = S[i];
                        t_sm++;
                  }     /* end if */
            }           /* end for */

            if ( t_sm >= k )
            {
                  for (i=0; i < t_sm; i++) S[i] = small[i];
                  return seq_select(S, t_sm, k);
            }
            else if ( (t_sm+t_eq) >= k )
            {
                  return median;
            }
            else
            {
                  for (i=0; i < t_big; i++) S[i] = big[i];
                  return seq_select(S, t_big, k - t_sm - t_eq );
            }     /* end if */
      }           /* end if */

}                 /* end sequential select */
```

```
/*-----------------------------------------------------------------
sub_seq():    This routine is used to divided a sequence into
              three subsequences during selection operatiotn:  ]
              (1) < median  (2) = median (3) > median
----------------------------------------------------------------*/
signed long  sub_seq(S, med, size, Lsize, Esize, Bsize)
signed long  *S, med;
int size, *Lsize, *Esize, *Bsize;
{
       int i, j = 0;
       int l, e, b;

       l=e=b = 0;
       *Lsize = *Esize = *Bsize = 0;

       for ( i = 0; i < size; i++) _small[i] = _equal[i] = _big[i] = 0;

       for ( i = 0; i < size; i++)
              if ( S[i] > med )
              {
                     _big[b] = S[i];
                     b++;
              }
              else if ( S[i] == med )
              {
                     _equal[e] = S[i];
                     e++;
              }
              else if ( S[i] < med )
              {
                     _small[l] = S[i];
                     l++;
              }
              else
              {
                     return FAIL;  /* subdivision has failedreturn error */
              }

       j = 0;
       for ( i = 0; i < l; i++)
       {
              S[j] = _small[i];
              j++;
       }
       for ( i = 0; i < e; i++)
       {
              S[j] = _equal[i];
              j++;
       }
       for ( i = 0; i < b; i++)
       {
              S[j] = _big[i];
              j++;
       }
       *Lsize = l;
       *Esize = e;
       *Bsize = b;

       return DONE;
}      /* end of routine */
```

```
/*-------------------------------------------------------------------
div_seq():   This routine is used to divide a sequence into four
             equal size subsequences during parallel sort operation.
             Suquences such that each element of S1 < S2 < S3 < S4.

-------------------------------------------------------------------*/
signed long  div_seq(S, m, size)
signed long  *S, *m;
int size;
{
      /* NOTE:   Max array size limit of seqX[] is 5000 */
      int i, j = 0;
      int mone, mtwo, mthree, mfour;
      int med, adj=0;
      signed long a, b, c, d;

      for ( i = 0; i < (size/4+adj); i++)
            seq1[i] = seq2[i] = seq3[i] = seq4[i]=0;

      /*      m[0]    index -----> median m[1]    */
      /*      m[2]    index -----> median m[3]    */
      /*      m[4]    index -----> median m[5]    */

      mone = mtwo = mthree = mfour = 0;

      for ( i = 0; i < size; I++)
      {
            if ( S[i] <= m[0] )
                  seq1[mone++] = S[i];
            else if ( (S[i] > m[0]) && (S[i] <= m[1]) )
                  seq2[mtwo++] = S[i];
            else if ( (S[i] > m[1]) && (S[i] <= m[2]) )
                  seq3[mthree++] = S[i];
            else if ( S[i] > m[2] )
                  seq4[mfour++] = S[i];
      }     /* end for */

      j = 0;
      for ( i = 0; i < mone; i++)
      {
            S[j] = seq1[i];
            j++;
      }     /* end for */
      for ( i = 0; i < mtwo; i++)
      {
            S[j] = seq2[i];
            j++;
      }     /* end for */
      for ( i = 0; i < mthree; i++)
      {
            S[j] = seq3[i];
            j++;
      }     /* end for */
      for ( i = 0; i < mfour; i++)
      {
            S[j] = seq4[i];
            j++;
      }     /* end for */

      return 0;
}           /* end of routine */


/*-------------------------------------------------------------------
fast_sort(): This routine sorts a given sequence using fast sort
             techniques.
```

```
-------------------------------------------------------------------*/
signed long fast_sort(l_index, size)
signed long l_index;
int size;
{
	signed long temp;
	signed long m1, m2, m3;
	signed long  size1, size2, size3;
	float limf = 0.0, k_val;   /* jj */
	int x, xsize, i, j, lim = 0, N = 150;

	ssize = size;
	f_index = l_index;

	if ( ssize <= N)
	{
		seq_sort(&t_addr[f_index], ssize);
		medM[8] = 4444;
		medM[9] = 5555;
		if ( glob_popA(&medM[8], &medM[9]) == 0)
		{
			/* reference below has to be global  */
			fast_sort(medM[8], medM[9]);
		}
		else
			return 0;
	}
	else if ( ssize > N)
	{

		/* STEP A:      find m1, m2, and m3 */
		k_val = ssize/4;
		i = ssize/4;
		if ( k_val > i) i++;

		/* STEP B: push the sequence index and size for m3, m2, and m1 */
		medM[0] = i+f_index;
		medM[1] = 2*i+f_index;
		medM[2] = 3*i+f_index;

		m1 = i;
		m2 = 2*i;
		m3 = 3*i;

		size3 = ssize - m3;
		size2 = m3 - m2;
		size1 = m2 - m1;

		medM[6] = m3 + f_index;
		medM[7] = size3;
		if ( glob_pushA(&medM[6], &medM[7]) == 0);

		medM[6] = m2 + f_index;
		medM[7] = size2;
		if ( glob_pushA(&medM[6], &medM[7]) == 0);

		medM[6] = m1 + f_index;
		medM[7] = size1;
		if ( glob_pushA(&medM[6], &medM[7]) == 0);

		/* NOTE:   loc_seq size ls limited by INT_RAM size */
		for ( i = 0; i < ssize; i++)  loc_seq[i] = t_addr[i+f_index];
		medM[0] = seq_select(loc_seq, ssize, m1);

		for ( i = 0; i < ssize; i++)  loc_seq[i] = t_addr[i+f_index];
		medM[1] = seq_select(loc_seq, ssize, m2);
```

```
        for ( i = 0; i < ssize; i++)  loc_seq[i] = t_addr[i+f_index];
        medM[2] = seq_select(loc_seq, ssize, m3);

        /* STEP C:  divide the sequence */
        if ( div_seq(&t_addr[f_index], medM, ssize) == 0);

        /* STEP D:  call fast_sort recursively */
        ssize = size1;
        /* call fast_sort recursively */
        fast_sort(f_index, ssize);

    } /* end if */
    return 0;
}   /* end of fast sort */
```

REFERENCES

[1] S. G. Akl, *The Design and Analysis of Parallel Algorithms.* Englewood Cliffs, NJ: Prentice Hall, 1989.

[2] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, and Programmability.* Englewood Cliffs, NJ: McGraw-Hill, 1993.

[3] *V-C40 Hydra User's Manual*, Ariel Corporation, Version 0.54, Highland Park, NJ, Aug. 1993.

[4] R. Hross, S. Ziavras, C. Manikopoulos, N. J. Lad, and X. Li, "A Defect Identification Algorithm for Sequential and Parallel Computers," in *IEEE International Symposium on Industrial Electronics*, Athens, Greece, July 1995.

[5] R. Simar, P. Koeppen, J. Leach, S. Marshall, D. Francis, G. Mekras, J. Rosenstrauch, S. Anderson, "Floating-Point Processors Join Forces in Parallel Processing Architectures," *IEEE Micro*, pp. 60-69, Aug. 1992.

[6] X. Li, "Investigation of Hybrid Message-Passing and Shared-Memory Architectures for Parallel Computers. A case study: TurboNet," Ph.D. dissertation, Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ, May 1995.

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

[8] X. Li, S. G. Ziavras, and C. N. Manikopoulos, "Parallel DSP Algorithms on TurboNet: An Experimental Hybrid Message-Passing/Shared-Memory Architecture," *Concurrency: Practice and Experience*, in appearance.