

9-30-1990

Implementation of parallel sorting algorithms on a transputer network

Venkatraman U. Calidas
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

Calidas, Venkatraman U., "Implementation of parallel sorting algorithms on a transputer network" (1990).
Theses. 2535.
<https://digitalcommons.njit.edu/theses/2535>

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

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.

Abstract

Thesis title: Implementation of Parallel Sorting Algorithms on a Transputer Network

Venkatraman U. Calidas, Master of Science in Electrical Engineering, 1990

Thesis Directed by: Dr. John Carpinelli, Assistant Professor
Department of Electrical and Computer Engineering

Many sorting algorithms, such as the bubble sort and the binary tree selection sort, are n -squared algorithms and take $O(n^2)$ time to sort n values. These can be easily parallelized; the resulting sorts are called the odd-even transposition sort and the parallel tree selection sort, respectively. These have $O(n \log n)$ time complexity. The quicksort is an $O(n \log n)$ algorithm and it can be parallelized to make it an $O(\log n)$ algorithm.

A transputer is a microprocessor which has serial links to transmit and receive data from other transputers. The above algorithms are implemented on a network of transputers and the link connections for the topology for each algorithm is suggested. The hardware and software tools available for this thesis enabled us to implement the above parallel sorting algorithms.

2) Implementation of Parallel Sorting Algorithms on a
Transputer Network

1) by
Venkatraman U. Calidas

Thesis submitted to the Faculty of the Graduate School of the New Jersey
Institute of Technology in partial fulfillment of the requirement for the
degree of Master of Science in Electrical Engineering

1990


APPROVAL SHEET


TITLE OF THESIS: **Implementation of Parallel Sorting
Algorithms on a Transputer Network**


NAME OF CANDIDATE: **Venkatraman U. Calidas**
Master of Science in Electrical Engineering, 1990

THESIS & ABSTRACT

APPROVED:


Dr. John Carpinelli 9/6/90
Assistant Professor Date
Department of Electrical and Computer Engineering


Dr. Anthony Robbi 9/13/90
Associate Professor Date
Department of Electrical and Computer Engineering


Dr. Edwin Hou 9-6-90
Assistant Professor Date
Department of Electrical and Computer Engineering

Vita

Name: Venkatraman U. Calidas

Permanent Address:

Degree to be conferred: M.S.E.E., 1990

Date of birth:

Secondary Education: Vidya Mandir, Madras, India

| Collegiate institutions attended | Dates | Degree | Date of Degree |
|---|---------|------------|----------------|
| S.D.M. College of Engineering Dharwad, India | 1984-88 | B.S.E.C.E. | May, 1988 |
| New Jersey Institute of Technology | 1988-90 | M.S.E.E. | October, 1990 |
| Major: Electrical Engineering | | | |

~~Contents~~

| | | |
|---|--|----|
| | List of Figures | iv |
| | List of Tables. | v |
| 1 | Introduction | 1 |
| 2 | Transputer Basics | 4 |
| | 2.1 Overview. | 4 |
| | 2.2 Transputer is a programmable device | 4 |
| | 2.3 Generic Architecture of the transputer family | 5 |
| | 2.4 IMS T800 Architecture | 8 |
| | 2.5 Floating point instructions | 9 |
| | 2.6 Concurrent operation of the FPU and CPU | 9 |
| | 2.7 Communication Links. | 10 |
| | 2.8 Operation of transputer hardware used with PC as host | 11 |
| | 2.8.1 PART.2 BOARD | 11 |
| | 2.8.2 PART.6 BOARD. | 15 |
| 3 | Software Development Tools | 20 |
| | 3.1 Introduction | 20 |
| | 3.2 Compiling a C program | 21 |
| | 3.2.1 PP The preprocessor | 21 |
| | 3.2.2 TCX "C" compiler | 22 |
| | 3.2.3 TASM transputer assembler | 22 |
| | 3.2.4 TLNK Transputer | 23 |
| | 3.2.5 TLIB Librarian | 24 |
| | 3.2.6 Network loader | 24 |
| 4 | Sorting and Parallel sorting | 32 |
| | 4.1 Introduction | 32 |
| | 4.1.1 Judging sorting algorithms | 34 |
| | 4.1.2 Example of exchange sort | 34 |
| | 4.1.3 Sorting by selection | 36 |
| | 4.1.4 Example of insertion sort | 37 |
| | 4.2 Improved sorts | 38 |
| | 4.2.1 Shell sort | 38 |
| | 4.2.2 Quicksort | 39 |
| | 4.3 Parallel sorting | 41 |
| | 4.3.1 The odd-even transposition sort | 42 |
| | 4.3.2 Parallel Tree algorithm | 43 |
| | 4.3.3 Parallel Quicksort algorithm | 45 |

| | | |
|---|--|----|
| 5 | Implementation | 46 |
| | 5.1 Introduction to processes and channels | 47 |
| | 5.1.1 Concurrency | 47 |
| | 5.1.2 Interprocess communication. | 49 |
| | 5.1.3 Alternation | 51 |
| | 5.1.4 Semaphores | 52 |
| | 5.1.5 Reliable communication | 53 |
| | 5.2 An example program | 55 |
| | 5.3 Implementation of odd-even transposition sort | 56 |
| | 5.4 Implementation of parallel tree sort algorithm | 60 |
| | 5.5 Implementation of parallel quicksort algorithm | 62 |
| 6 | Conclusion | 64 |
| | Bibliography | 66 |
| | Appendix | 68 |

List of Figures

| | | |
|------|---|----|
| 2.1 | Plot of performance against number of processors | 6 |
| 2.2 | Plot of performance for a multi-Transputer system | 6 |
| 2.3 | Transputer generic architecture | 7 |
| 2.4 | T800 Architecture | 9 |
| 2.5 | Elements of communication protocol | 10 |
| 2.6 | Signal when acknowledged packet overlaps | 10 |
| 2.7 | Block diagram of Link adaptor cell | 12 |
| 2.8 | Block diagram of PART.6 board. | 14 |
| 2.9 | A Transputer chain | 16 |
| 2.10 | A binary tree connection | 19 |
| | | |
| 3.1 | Network topology for 'nif' file | 30 |
| 3.2 | Network topology for 'nif' file | 31 |
| | | |
| 4.1 | Bubble sort algorithm | 35 |
| 4.2 | Selection sort algorithm | 36 |
| 4.3 | Insertion sort algorithm | 37 |
| 4.4 | Shell sort algorithm | 39 |
| 4.5 | Operation of Quicksort | 40 |
| 4.6 | Binary tree data structure | 44 |
| | | |
| 5.1 | Example program | 54 |
| 5.2 | 'nif' file used for example program | 55 |
| 5.3 | Network topology used for odd-even sort | 56 |
| 5.4 | Odd-even transposition sort | 57 |
| 5.5 | Transputer link connection in a binary tree | 60 |
| 5.6 | 'nif' file for binary tree topology. | 61 |

List of Tables

| | | |
|-----|--|----|
| 2.1 | Relationship between system services signals . . . | 17 |
| 2.2 | Allowable system services connections . . . | 18 |

Chapter 1

Introduction

Sorting is a very fundamental problem while dealing with computers and computer programs. The process of sorting or ordering a list of objects according to some linear order is done very frequently. These routines are used in almost all database programs as well as in compilers and operating systems.

Due to the fundamental nature of the sorting problem it has been extensively studied. A very detailed treatment is given by Knuth [9]. As most computers were of the Von-Neuman type, the sorting algorithms were originally serial in nature. However with the design of SIMD and MIMD computers, parallel algorithms too were investigated.

With the advent of VLSI technology it became possible to build complex processors on a single chip. One such chip is the powerful microprocessor T800 by INMOS. The T800 is different from other conventional processors such as the MC68020 from Motorola. The T800 has been designed with parallel processing in mind. The interprocess communication is done by the 4 links available on the T800. Thus it is possible to connect the T800 in a variety of topologies such as linear, mesh, ring and tree connections.

Transputer boards with typically four T800s are available commercially. These boards fit in the bus of any PC/AT compatible computer,

and with the help of software we can create a parallel processing environment. One such board from the company Computer Systems Architects, which has 4 T-800s is used in this work.

In this thesis parallel sorting algorithms have been implemented on a network of transputers. Many fast parallel sorting algorithms require $O(n^2)$ processing elements to sort n elements in $O(\log n)$ time [10]. For large values of n the number of processors needed would be prohibitively high. In some cases sorting networks have been suggested. Not all computers can afford to have such a sorting network or a large number of processors. Thus an attempt has been made to perform parallel sorting with the number of processors much less than the number of elements to be sorted.

The bubble sort algorithm is a simple serial sort algorithm. This algorithm may be parallelized; this is called the odd-even transposition sort [12]. Another serial algorithm which can be parallelized is the parallel tree algorithm [5]. The method to adapt this algorithm to run on a transputer tree network is shown. The quicksort algorithm which is by far one of the most popular sort algorithms, can also be parallelized. One method of the parallel quicksort algorithm is discussed and implemented.

This thesis dissertation is organized according to the following chapters.

Chapter 1: This is the introduction to this thesis work.

Chapter 2: This chapter gives the basic overview of the transputer architecture.

Chapter 3: This chapter describes the software tools to compile, load and run programs on a transputer network.

Chapter 4: This chapter discusses the various sorting methods in general use and their algorithms. Some of these algorithms can be parallelized, and these algorithms are also discussed.

Chapter 5: This chapter deals with the implementation of parallel algorithms, dealt with in the previous chapter, to run on a transputer network.

Chapter 6: This is the final chapter with results of this work and conclusions.

Chapter 2

Transputer Basics

2.1 Overview

The Transputer is designed to exploit the potential of VLSI technology. This technology allows a large number of identical devices to be manufactured cheaply. For this reason it is attractive to implement a concurrent system using a number of identical components, each customized by an appropriate program. The transputer is a VLSI device with a processor, memory to store the program executed by the processor, and communication links for direct connection to other transputers.

2.2 Transputer is a programmable device

Transputer systems can be designed and programmed using the *OCCAM* language. Occam allows an application to be described as a collection of processes operating concurrently and communicating through channels. The transputer can be used as a building block for concurrent processing systems, and Occam as the associated design formalism.

Occam was developed to program concurrent, distributed systems. Emphasis is placed on the word distributed because many previous languages were unsuitable for this area. Occam enables a system to be described as a collection of concurrent processes which communicate with each other and

with peripheral devices through channels. Concurrent processes do not communicate via shared variables, and thus Occam is a suitable language for programming systems where there is no memory shared between processors in the systems.

2.3 Generic architecture of the Transputer family

Speedup of an array processor may be roughly defined as the ratio of the time taken by a uniprocessor to complete a given job to the time taken by an array processor for the same job. The speedup that can be achieved by a parallel computer with n identical processors working concurrently on a single problem is at most n times faster than a single processor. In practice the speedup is much less, since some processors are idle at a given time because of conflicts over memory access or communication paths. Figure 2.1 shows the various estimates of the actual speedup, ranging from a lower-bound $\log_2 n$ (Minsky's conjecture) to an upper-bound $n/\ln(n)$ [7]. Attempts to build multiprocessor systems from conventional microprocessors have always been fraught with problems, not the least of which have been those with inter-process communication. Bandwidth problems are usually encountered when more than four processors are linked via a shared bus. Typical commercial multi-processor systems consist of only two to four processors.

Transputers have their own built-in serial links, each capable of concurrently inputting and outputting at up to 10 Mbit/sec. A typical transputer has four such links, giving it a communications capacity of 40 Mbit/sec into the device and 40 Mbit/sec out. As these links are an integral part of each individual transputer, the greater the number of transputers in a network the greater the total bandwidth of the system. Due to the serial links available on every transputer, they can easily be designed to form array

structured SIMD computers. The array may take different shapes such as hypercube, binary tree, mesh and so on. Analysis by Hwang and Briggs [7] of SIMD array processors shows that speedup increases monotonically with respect to the number of available processors. Figure 2.2 shows the speedup versus the number of processors available. Thus we can expect increase in speedup while using five transputers available for this work.

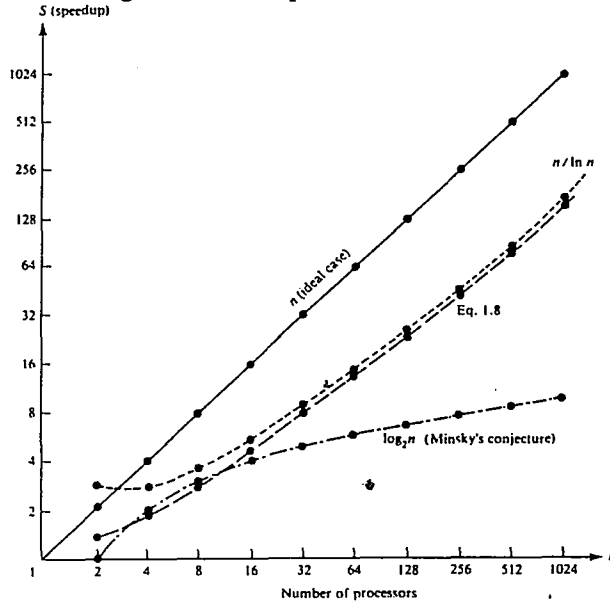


Figure 2.1 Plot of speedup versus number of processors in a multi-processor system

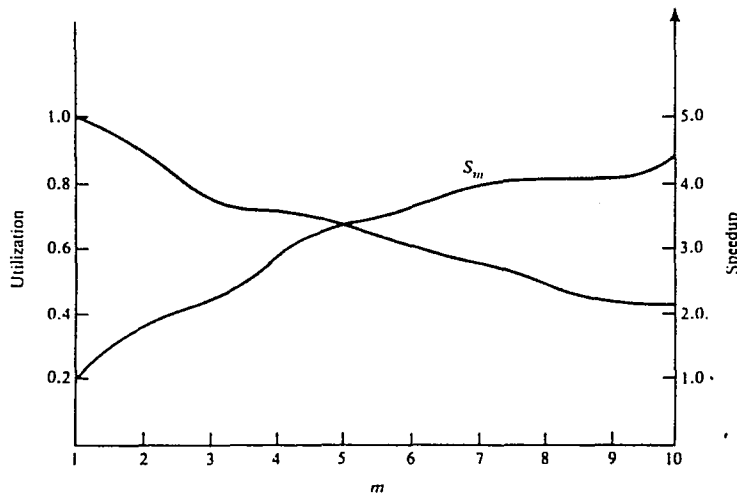


Figure 2.2 Plot of speedup versus number of processors in an array processor

Figure 2.3 shows the generic architecture of the transputer family. For example, the T424 32-bit transputer, with 4 serial links and 4 Kbytes of on-chip static RAM, is usually taken as being the standard general purpose transputer, but various combinations of on-chip facilities are possible. Transputers always include a processor, system services and one or more serial links, but they may have 4 Kbytes of on-chip RAM, 2 Kbyte or no on-chip memory. A more powerful transputer is the T800, which has an on-chip floating point computation unit.

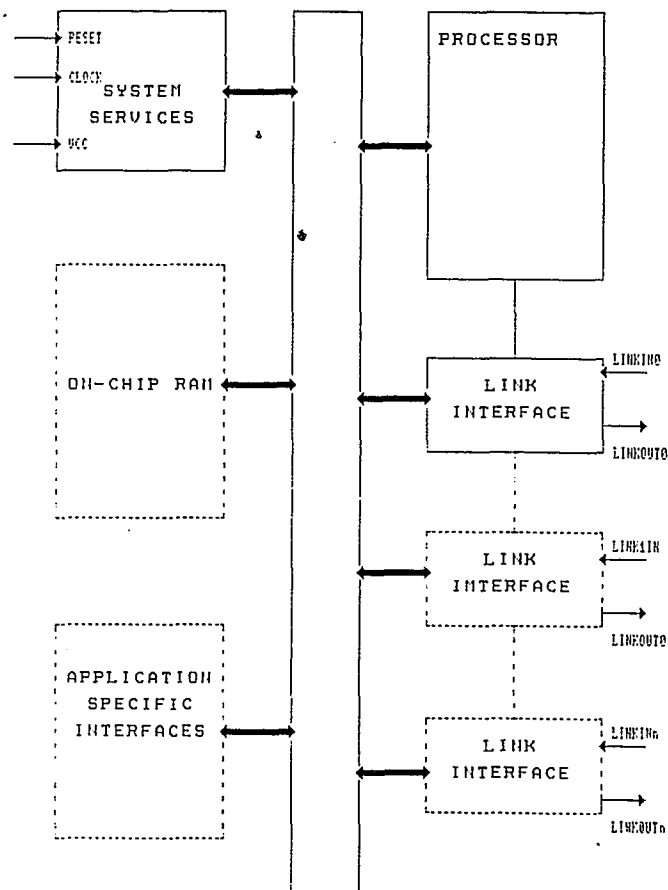


Figure 2.3 Transputer generic architecture

All transputers have a timer which runs off a 5MHz external clock. Both the processor clock frequency and the link clock (which controls the transfer rate of the serial links) are derived from the external clock frequency by internal scaling. All transputers support the 10 Mbit/sec standard. The link protocol, like the link speed, is standard across the whole range of transputers. Data is transmitted one byte at a time in packets, with each packet being acknowledged by the receiving transputer. The acknowledgement packet is sent as soon as the data packet is identified and, since each link is bi-directional, data packets and acknowledgement packets can be communicated concurrently.

2.4 IMS T800 Architecture

The T800 is a microprocessor which has an on-chip floating point computation unit. The small size and high performance comes from a design which takes careful note of silicon economics. Many contemporary microprocessors come with a co-processor which occupies more area than the microprocessor itself. The block diagrams of the interconnection between the major blocks of the IMS T800 is given in Figure 2.4 [14]. The T800 CPU contains three registers (A, B and C) used for integer and address arithmetic, which form a hardware stack. Loading a value onto the stack pushes B into C, and A into B before loading A. Storing a value from A pops B into A, and C into B. There are three registers in the FPU which are called AF, BF and CF, and they push and pop the same way as the A, B and C registers.

All transputers share the same basic instruction set. It contains a small number of instructions, each with the same format, chosen to give a compact representation of the operations most frequently occurring in programs. In this respect the transputer family follows principles of Reduced Instruction

Set Computers (RISC). The four most significant bits are function code, and the four least significant bits are a data value. The 16 functions include loads, stores, jumps, and calls, and enable the most common instructions to be represented in a single byte.

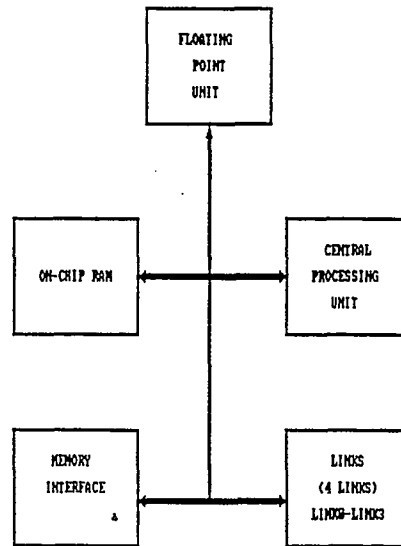


Figure 2.4 T800 architecture

2.5 Floating point instructions

The core of floating instructions includes simple load, store and arithmetic instructions. The transputer designers ran several FORTRAN programs whose results suggested that the addition of some more complex instructions would improve performance and code density. The resulting instruction set reflects this philosophy.

2.6 Concurrent operation of FPU and CPU

In the IMS T800 the FPU operates concurrently with the CPU. It is therefore possible to perform an address calculation while the FPU performs a floating point calculation.

2.7 Communication Links

Two transputers are linked by connecting their respective link interfaces with two one-directional signal wires, along which data is transmitted serially. The two wires provide two Occam channels, one in each direction. Each Occam channel requires data to be transmitted in one direction and control information in the other. A simple protocol is used to multiplex data for one channel and control information for the other channel onto the wires. Messages are transmitted as a sequence of bytes. Each byte must be acknowledged before the next is transmitted. A byte of data is transmitted as a start bit, followed by a one bit, followed by eight bits of data, followed by a stop bit. An acknowledgement is transmitted as a start bit followed by a stop bit. Figure 2.5 shows the elements of the communication protocol used and Figure 2.6 shows the signals when the acknowledged packet overlaps the data. Thus the IMS T800 floating point transputer provides a very high performance building block for concurrent systems.

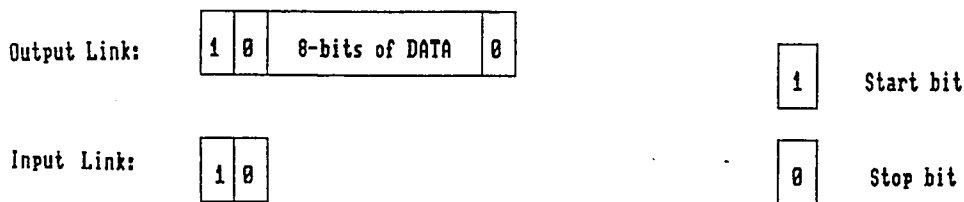


Figure 2.5 Elements of communication protocol

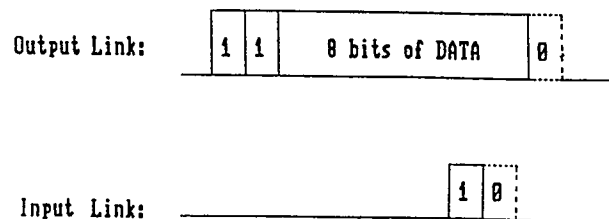


Figure 2.6 Signal when acknowledged packet overlaps

2.8 A brief theory of operation of the transputer hardware used with a PC as host

There are two boards available for this research. They are the Link Adapter Board and the multiple transputer board with four Transputers. These are known as the PART.2 and PART.6 boards, respectively. A brief description of the two boards follows.

The PART.2 board from CSA is one of a series of flexible tools designed for development projects using the Inmos transputer. At the same time it is a formidable application accelerator providing an aggregate 10 MIPS and 1.5 MFLOPS of computing power and 256 Kbytes of memory.

2.8.1 The Part.2 board

The block diagram of the Link Adaptor cell is shown in Figure 2.7. The PART.2 board is an Inmos compatible implementation of the Transputer environment. Each board consists of two cells, the transputer cell and the link adaptor cell. The transputer cell contains the transputer microprocessor, 256 Kbytes of 100ns DRAM, differential link buffers, and a small amount of control logic. The link adaptor cell contains the an Inmos C012 link adaptor, differential link buffers and PC bus interface logic. The transputer and link adaptor cells may work independently of each other but are normally configured for connection via one link.

Communication between the PC and the transputer is performed through a Link Adaptor. This device, similar in concept to a high speed UART, converts 8 bit parallel information from the PC into a bit stream for transmission over a transputer link. The function of the Link adaptor is just to communicate between the PC and the transputer. Two methods of communication can be established at any time between this board and the PC.

Information may be transmitted to or from the link adaptor by programmed (or Channel I/O) or by direct memory access (DMA) using the PC DMA controller. The link adaptor is auxiliary to the transputer on this board.

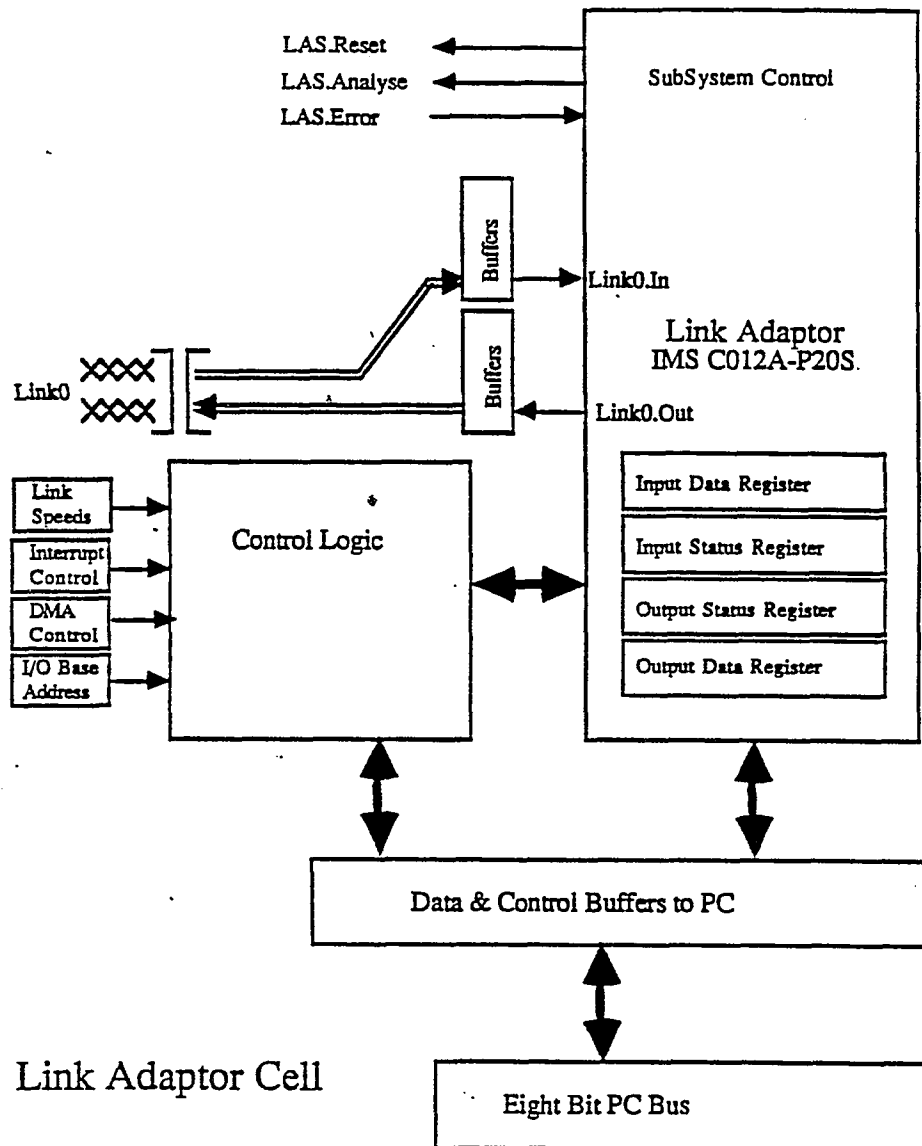


Figure 2.7 Block diagram of link adaptor cell

The transputer on a PART.2 has no connection between the PC or any other transputer than that provided across transputer links. This communication occurs transparent to the computation engine of the transputer as a type of direct memory access. Link information is placed directly into the transputer memory as it is received. In the same manner, information may be transmitted from memory across the link to a link adaptor or another transputer without burdening the computation engine. In some configurations, links are chained from transputer to transputer. When results of computations are returned to the host transputer, the information passes through the link structure of each transputer between the transmitting transputer and the receiver.

2.8.1.1 Theory of Operation

The PART.2 is designed as a simple transputer evaluation cell with a single transputer, 256 Kbytes of external memory and a PC interface. For convenience during the initial use of the PART.2 board, standard connections between the transputer and Link adaptor cells have been pre-wired on the etched circuit card. PC link adaptor connection, link speeds and interrupt selections have also been pre-selected and wired via the circuit board.

The transputer is a separate computer running independent from the PC. It is attached to the PC through a PC Link adaptor. The link adaptor is a device from Inmos that converts 8 bits parallel from the PC bus, to serial data for receipt by transputer links. On a standard PART.2, link 0 of the transputer is hard wired on the transputer to the link adaptor. The link speeds for both the link adaptor and transputer are set at 10 MBits/sec. The interface used between the PC and link adaptor is Direct memory access. The information is then directly placed from PC memory into the transputer

memory. Several PART.2 transputers may be connected together by connecting links 1-3. Enhancements possible with the PART.2 include: Switch selectable link speeds 10/20 MBits/sec, switch selectable connection between link adaptor and PC, and selectable link adaptor to PC DMA and PC interrupt addresses.

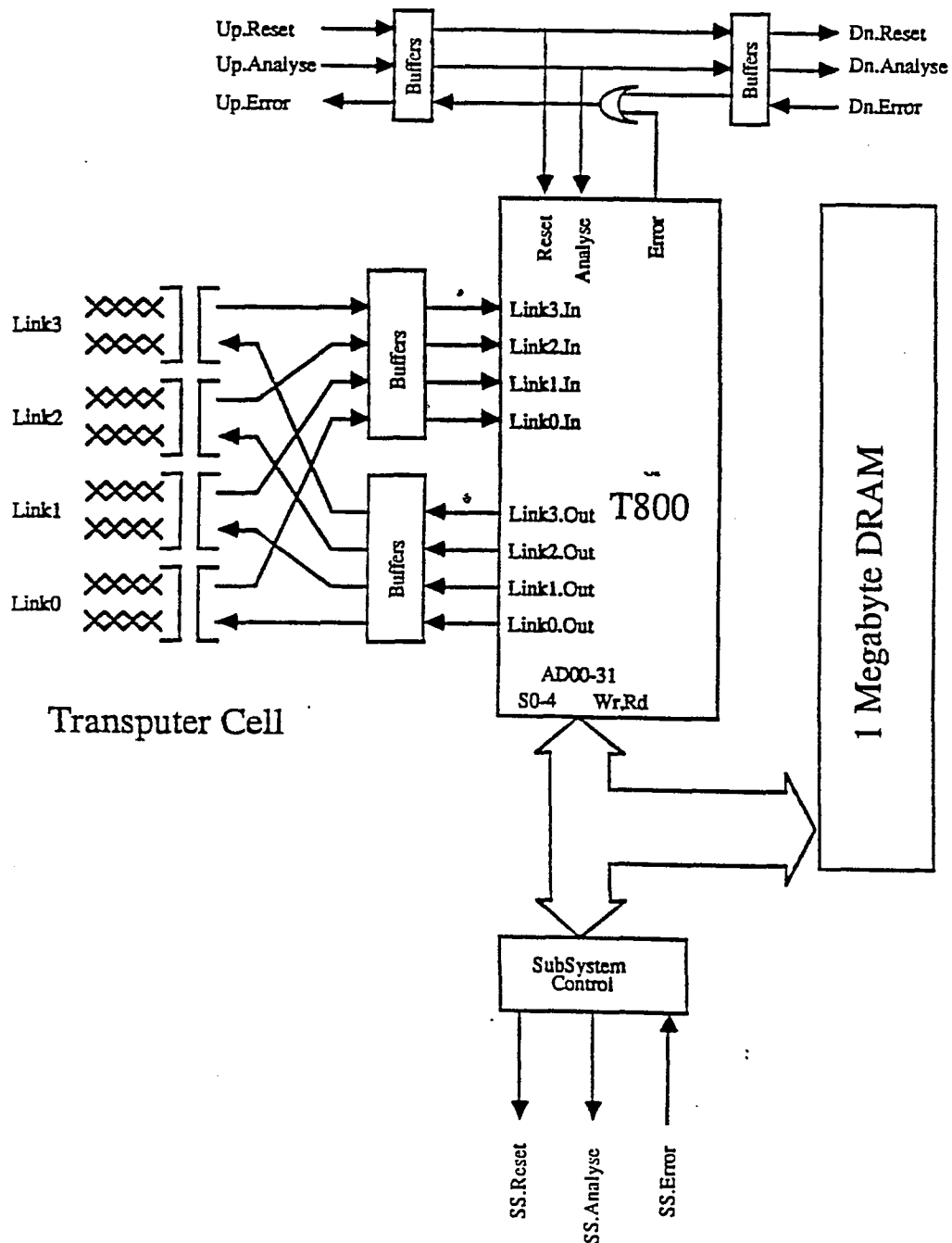


Figure 2.8 Transputer cell in a PART.6 board having four transputer cells

2.8.2 The Part.6 board

The block diagram of the PART.6 board is shown in Figure 2.8. The PART.6 board from CSA is a flexible tool designed for development projects on the forefront of processor technology. At the same time, PART.6 is a formidable application accelerator providing an aggregate 40 MIPS / 8 MFLOPS of computing power and 1MByte of memory. The PART.6 is a straightforward implementation of the transputer parallel processing environment. Each transputer cell consists of the transputer microprocessor, 256 Kbyte of 100 ns DRAM, differential link buffers, and a small amount of miscellaneous logic. The four transputer cells are completely independent of each other except for the single clock oscillator. The board available for the work carried out in this dissertation is the PART.6 which has 4 transputers and 1 Megabyte of memory per board.

The PART.6 board allows for great flexibility in arranging arbitrary systems of Transputers. No predefined interconnection topology has been assumed. The user may have any combination of systems and subsystems that cross large numbers of PART.6 boards, or may opt for a simple, one level system of just one board.

Figure 2.9 shows the connections for a network configured as a Transputer Chain.

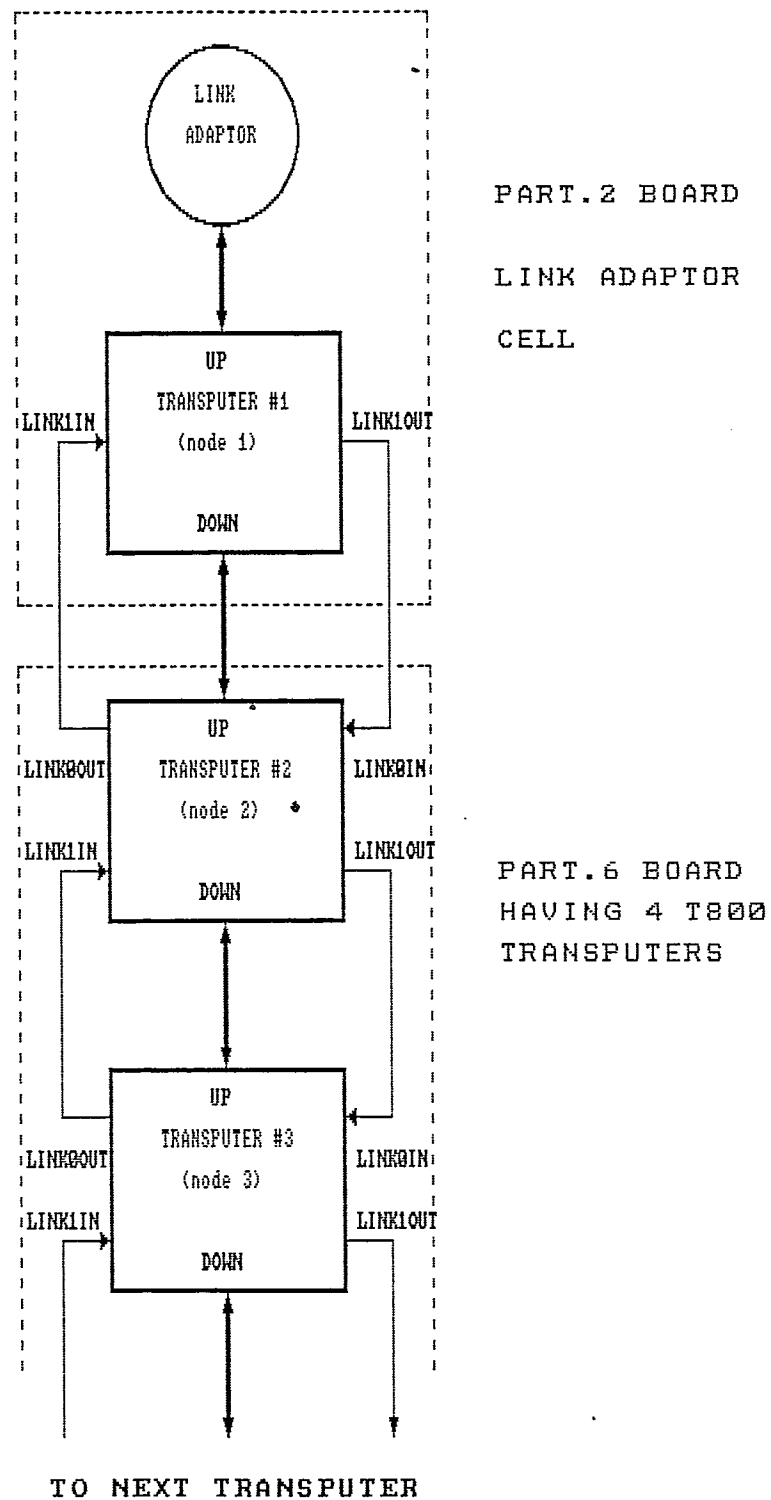


Figure 2.9 A transputer chain

2.8.2.1 System Services

In defining system services, the Inmos philosophy and terminology has been adopted. The system services network consists of three signal lines; RESET, ANALYZE and ERROR. Reset and Analyze always travel in the same direction, while Error travels in the opposite direction. There are three distinct type of Systems services connections; UP, DOWN and SUBSYSTEM. Table 2.1 [17] describes the relationship between the system services signals and types of connections.

| | UP | DOWN | SUB-SYSTEM |
|---------|--------|--------|------------|
| RESET | INPUT | OUTPUT | OUTPUT |
| ANALYZE | INPUT | OUTPUT | OUTPUT |
| ERROR | OUTPUT | INPUT | INPUT |

Table 2.1 Relationship between system services signals

It is intended that the UP and DN connections provide a chain of buffered control signals. In a chain of Transputer cells, the first cell (usually the PART.2 board) is considered the master and assumes the uppermost position. Reset and Analyze signals propagate down the chain to cells positioned lower in the chain, while the error signal propagates up the chain to the master position. The UP.Reset provides the master reset input for any Transputer cell. The signal is buffered and passed on directly to the DN.Reset output. The same is true for the UP.Analyze and DN.Analyze. The error signal is received as an input from other Transputer cells at DN.Error. This signal is ORed with the local error flag and sent out on UP.Error. The SS

connection provides for the beginning of a new chain of control. These signals originate under software control, in the Transputer cell itself.

The combination of UP, DN, and SS connections allow for the hierarchical control structures, such as trees, to be constructed. Figure 2.10 illustrates a binary tree. Table 2.2 [17] summarizes the allowable connections. A 4-pin connector is used to interconnect the system services signals.

| | UP | DN | SS | |
|----|----|----|----|----------------------------------|
| UP | -- | OK | OK | DN: DOWN |
| DN | OK | -- | -- | SS: SUBSYSTEM |
| SS | OK | -- | -- | --: MEANS CONNECTION NOT ALLOWED |
| | | | | OK: MEANS CONNECTION IS ALLOWED |

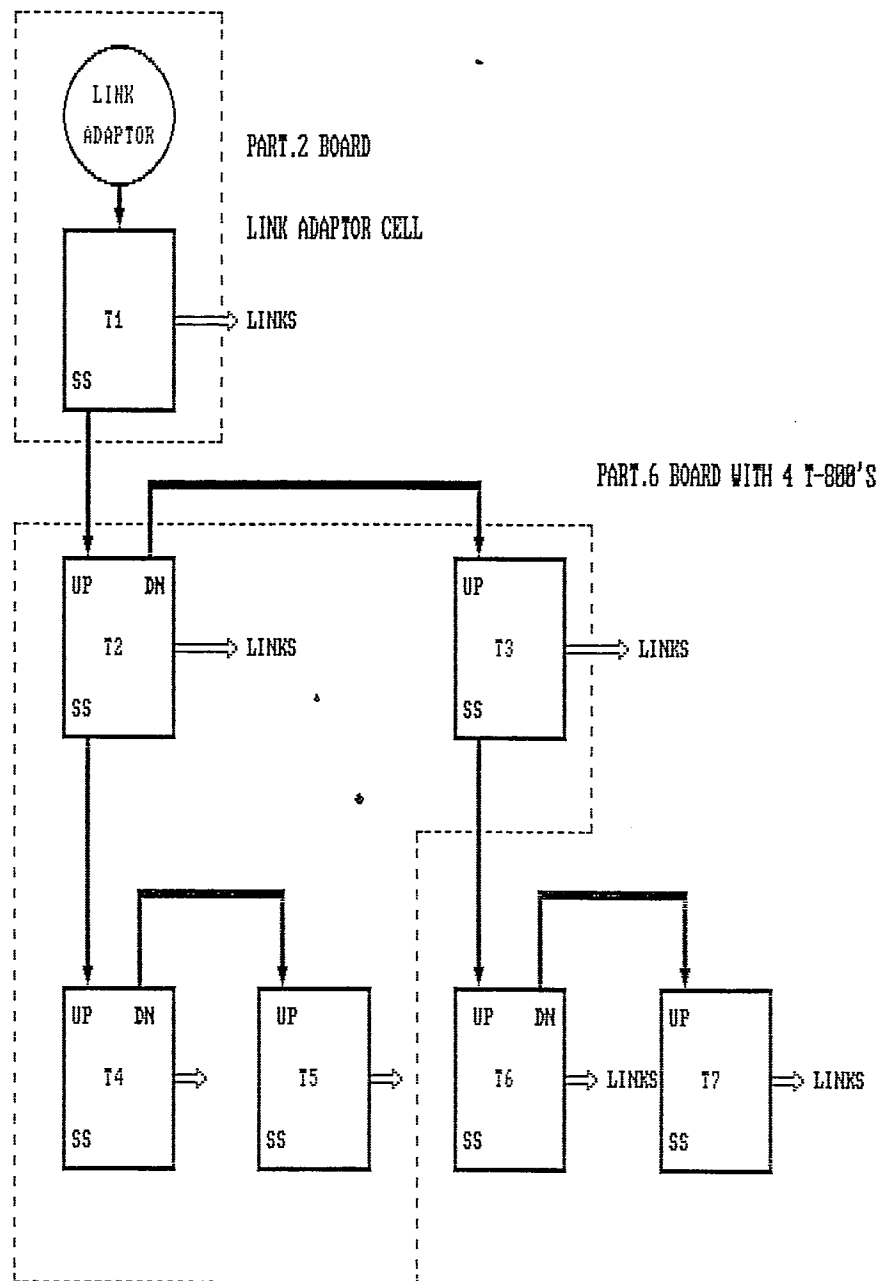
Table 2.2 Allowable systems services connections

2.8.2.2 Links

A 4-pin connector is used to provide link interconnection. The interconnection cables for system services are completely interchangeable with those for the links. The link signals are differentially buffered to provide a higher reliability interconnect.

2.8.2.3 Selections of Link speeds and Processor speeds

DIP switches are provided on the PART.6 boards to enable a change in Link speeds. The links speeds can be selected to operate at 5 MBits/sec, 10 MBits/sec or 20 MBits/sec. Jumpers are provided to change the processor speed.



T T-800 TRANSPUTER
 SS SUBSYSTEM
 DN DOWN

Figure 2.10 A binary tree connection

Chapter 3

Software development tools.

3.1 Introduction:

The transputer toolset functions as a cross development environment with the supplied tools executed on a host system and run on the attached Transputer array. Tools are provide for both program development and loading the executable files. The development tools are as follows.

| | |
|------|---------------------------------|
| PP | The "C" preprocessor |
| TCX | The "C" compiler |
| TASM | The assembler |
| TLNK | The linker/locater |
| TLIB | The relocatable file librarian |
| MAKE | The program maintenance utility |

The loading and execution tools are

| | |
|--------|---|
| LD-ONE | The single transputer loader |
| LD-NET | The 'network' loader for arrays of transputers. |
| CIO | The host I/O driver for "C" I/O on the transputer |

3.2 Compiling and running a C program

First, PP is run on the source code to handle "C" preprocessing functions. Then TCX is run to convert the code to transputer assembly language.

Next, TASM is run to assemble the code. At this point the code may be optionally included in a library using TLIB or linked into a executable file using TLNK (or both).

Finally the code is executed using the LD-ONE loader (for the case of a single transputer), or the LD-NET loader (if more than one transputer is to be loaded). In either case, since the program probably does runtime I/O, the CIO "C" host I/O driver will be used with the selected loader.

We shall now describe briefly how each of these tools works and give their special features.

3.2.1 PP The "C" preprocessor.

PP is a preprocessor for the "C" programming language. It conforms to the ANSI Draft Proposed "C" standard and contains options for backward compatibility with programs written for non-ANSI preprocessors.

The basic function of the preprocessor is reading "C" source code files and handling the file inclusion, conditional compilation, macros and other lexical manipulations which are requested by the programmer. When finished the preprocessor writes out a simplified version of the input source code file for later compilation by the "C" compiler proper.

3.2.2 TCX "C" compiler

The TCX is a "C" compiler for the INMOS 32-bit Transputers. TCX does not include preprocessing facilities and is intended to run as a post pass to the preprocessor. The general form of the TCX command line is:

```
tcx <input_filename> [-options]
```

The input_filename field must contain the name of the program being compiled. The option facility allows us to control the operation of TCX to match our needs.

By default the TCX generates code for the T414 Transputer. To have the TCX compile code for the T800 we must specify the "-p8" option.

The "-c" option compresses the output file by removing source debugging information. Unless the debugging information is required this option will substantially reduce the size of the output file and will also improve the code optimization since source statement boundaries are no longer important to preserve in the generalized code.

3.2.3 TASM Transputer Assembler

TASM is a relocatable assembler for the INMOS 32-bit Transputers. It supports standard INMOS mnemonics and allows splitting a program into separate pieces which are combined at linkage time. TASM is designed to be used in two ways:

- 1) As a post-pass to the TCX "C" compiler. The compiler generates an assembly language output file and TASM is used to turn it into relocatable format.

- 2) As a stand-alone tool for programming in the Transputer assembly language.

The architecture of the transputer requires that some of the code generation be delayed until the linker/locator stage to insure minimum length prefix strings are generated for all instructions.

The general form of the TASM command line is:

```
tasm <input_filename> [<temp_directory>] [- [options]]
```

3.2.4 TLNK Transputer Linker

TLNK is a linker designed for use with the INMOS 32-bit Transputers. It supports and allows complete control of load addresses for each program, library or code fragment. TLNK is designed for use with the TASM assembler and the TLIB librarian. It is also the linker used with the TCX compiler. TLNK may be run either as an interactive program or in a batch mode with a user specified command file.

The architecture of the transputer requires that some of the code generation be delayed until the linker/locator stage to insure minimum length prefix strings are generated for all instructions. TLNK supports this by processing information which TASM puts in the relocatable files to determine which instructions have been completed at assembly time and which TLNK has to use during the linkage operation. The general form of the TLNK command is:

```
tlmk [command_file]
```

If a command file is provided TLNK assumes default values and proceeds to link the program. Otherwise TLNK runs interactively and asks for input values of such parameters as temporary file path, names of the input files, library files to be linked and the name of the output file.

3.2.5 TLIB Transputer Librarian.

TLIB is a librarian for use with the TLNK Transputer linker. It allows the creation, modification and examination of libraries of relocatable files. TLIB operates in a fashion similar to a line oriented text editor with commands to append, insert, delete and manipulate libraries. The general form of the TLIB command line is:

```
tlib <library_filename> [-options] [option_specific]
```

The library_filename field must contain the name of the library with the extension ".tll".

In general, one type of operation may be done to the library for each invocation of TLIB. We may append multiple files at a time for example, but if we wish to delete a file we must do it in a separate operation. The libraries must be structured such that the TLNK linker only gets one defining instance of each external symbol. The TLIB can be instructed to read a series of files to be placed in a library and automatically determine a correct ordering.

3.2.6 LD-NET The Transputer Network Loader

LD-NET is used to load a collection of executable files onto a network of Transputers. Executable files are created as a result of linking relocatable modules using TLNK and have default file name extensions of ".tld".

Transputers networks to be used with LD-NET must conform to the standard INMOS system/subsystem reset/analyze topology and must have a subset of communications links in the network which mimics the system/subsystem connections (the remaining links may be connected in any arbitrary fashion).

The system/subsystem topology organizes a Transputer network into a binary tree with each node having one reset/analyze input and two

reset/analyze outputs (system and subsystem). Each node also has two error inputs and one error output associated with the reset/analyze connections but reversed in direction. Many configurations may be mapped into this configuration including linear and two dimensional arrays. Since links not involved in the reset process may be connected in an arbitrary fashion, they may be used to construct forms which wouldn't map easily, such as rings and hypercubes.

The basic LD-NET process is as follows:

1. LD-NET reads the specified "network information file" ('nif' file) to determine the Transputer network topology, and the programs to be loaded, and computes an optimum bootstrapping and load order.
2. The Transputer connected to the host system link adapter is RESET (the "root" node). This also causes the "system" reset output of the root node to be activated since the "system" output is a daisy chained version of the root node reset input.
3. A primary bootstrap is loaded onto the node and executed.
4. A secondary bootstrap is loaded onto the node and executed.
5. The root node performs a RESET operation on the software controlled (subsystem) output. At this point both system and subsystem reset outputs have been activated.
6. A primary and secondary bootstrap is sent to the root node for use with one of its children.
7. Steps 3 through 6 are repeated for each node down the tree until all nodes have been bootstrapped. This phase of the network load process is point to point since each bootstrap has node specific information, such as the children to be loaded, the desired program to keep during loading phase and the user assigned node number.

8. After all nodes are bootstrapped one copy of each different program to be loaded onto the network is sent to the root node.
9. The root node sends a copy of each program to each of its children who do the same thing in turn. Each node keeps the copy of the program it will eventually be running as it goes by. The part of the bootstrap process is sometimes called the program "flooding" phase since the entire network is parallel loaded with all the programs to be run on any node.
10. When all programs have been loaded the host sends the root node (and all other nodes) an EXECUTE command which starts all the programs on the network running. The actual execution of the user programs starts with the leaf nodes and works back to the root as the command is acknowledged up the tree.
11. At this point the Transputer network is loaded and running. LD-NET then does an "EXEC" to allow the specified host system driver program to run and handle I/O requests from the Transputer. The general form of the LD-NET command line is:

```
ld-net <network_information_file> <command line arguments>
```

The "network_information_file" (nif file) specifies the file to read to determine the network topology and the list of programs to be loaded. Any remaining parameters on the command line are passed to host driver program, which in our case is the host program running on the PC. In this project we use the CIO program which functions as an I/O server for "C" programs running on the root node.

The following information must be provided in the network information file:

1. **Buffer_size**
2. **Host_server**
3. **decode_timeout**
4. **level_timeout**
5. **Node descriptions**

The node description is the most important information as it describes the network topology. Here the line begins with a node number which describes four things about the node:

1. The user supplied node number to address it by.
2. The program to download to it.
3. Which node provides the reset signal for it and whether that signal originates from the "system" ("R") or the "subsystem" ("S"), reset output of the parent node.
4. What other nodes the links from this node talk to. Although only the links involved in the network loading process must be described, it is a good practice to completely describe the connections for use by other programs which might wish to read the network information ("nif") file.

The above information is contained in seven comma separated fields within the line, the last field on a line is terminated with a semicolon.

Field #1: Node Number

Contains the node number of the users choice. This number must be a positive non-zero integer in either "C" style decimal, hexadecimal or octal format. This number may range from 1 to 1000 and non-consecutive numbers may be used. The node number "0" is reserved to represent the host interface

and may not be used in any other fashion. By convention the node number "1" should be the root transputer in the network. The node number provided in the ".nif" file is available to user "C" programs during the execution as the contents of the "_node_number" external integer variable which is quite useful for runtime configuration chores.

Field #2: Program to Download

This field contains the name of the program to download to that particular node. By default, a file name extension of ".tld" is assumed if none is specified.

Field #3: Parent node

This field contains both the number of the parent node (the node which resets this node), and an indication about whether the "system" or "sub-system" output of the parent controls the reset for this node. This is indicated by a "R" (connected to a "system" output), or "S" (connected to sub-system output), letter followed by the parent node number. As usual, node number may be in "C" style decimal, hexadecimal or octal format. For example S1 would indicate that the current node has its reset input connected to the "subsystem" output of parent node 1.

No more than one node may be connected to the "system" or subsystem" output of any given node in a network.

Field #4: Link 0 connection

Field #5: Link 1 connection

Field #6: Link 2 connection

Field #7: Link 3 connection

The last four fields are used to specify which node each link is connected to (by node number). If a link is unconnected its entry may be left

blank. Since each node must have a direct link with the node which boots it, one link entry must have the same node number as the parent field.

Examples illustrating the above rules incorporated in the ".nif" files are given below.

```
1, test, R0, 1, 0, 2, ;
```

This line in a ".nif" file indicates that the current node is node 1, the program to load is "test.tld", this node is reset by the host interface (which is connected by link 1), and it has a link connection to itself (link 0) and link 2 is connected to node 2. As Link 3 is not connected to any other node it is not mentioned. The connection is shown in figure 3.1.

When two nodes have more than one link connection, a different approach is taken to fill in the fields.

For example

```
1, test, R0, 1, 0, 2[1], 2[3];
```

In this case two links exist between node 1 and node 2. Link 2 of node 1 is connected to link 1 of node 2, and link 3 of node 2 is connected to link 3 of node 2. When we use the "[]" notation to describe a network we need list only one end of a connection as it fully defines the linkage. This is illustrated in figure 3.2.

The input channel addresses from which a node was bootstrapped by its parent are available during "C" program execution as the contents of the "_boot_chan_in" and the "_boot_chan_out" external variables. The variables

are of type "Channel" and are declared in "conch.h". More about the channels and how channels interact with processes will be discussed in chapter 5.

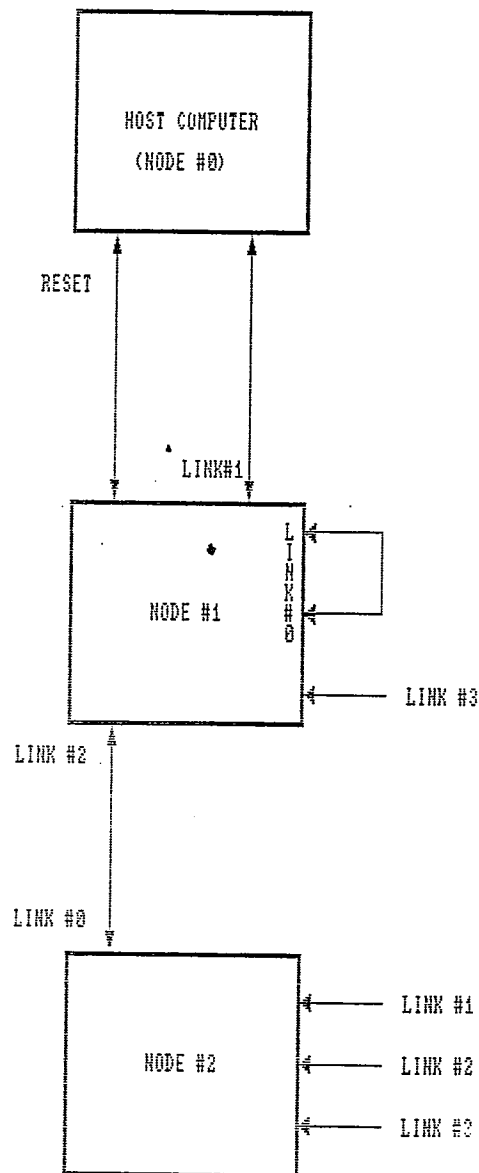


Figure 3.1 Transputer network configuration for 'nif' file containing the line: 1, test, R0, 1, 2, ;

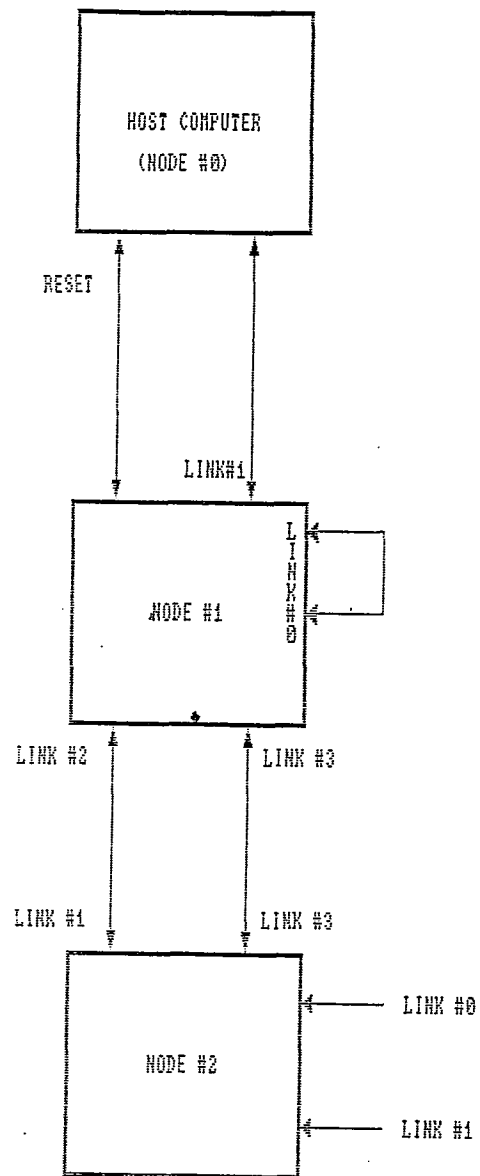


Figure 3.2 Transputer network configuration for 'nif' file containing the line: 1, test, R0, 0, 2[1], 2[3];

Chapter 4

Sorting and Parallel Sorting

In the world of computer science, perhaps no other tasks are more fundamental or as extensively analyzed as sorting and searching. These routines are used in almost all database programs as well as in compilers, interpreters and operating systems. There are two types of sorting: internal sorting and external sorting. When the data is available in the random access memory of the computer the sorting is called internal sorting and when it is available in sequential access disks or tapes it is called external sorting.

In this research we are only concerned about the internal sorting mechanisms. The first part of this chapter discusses some of the basic serial sorts available. In the second part of this chapter we discuss how some of these sorts can be parallelized. In the next chapter we discuss how these parallel sorts are implemented on the Transputer network.

4.1 Introduction

Sorting is the process of arranging a set of similar information into an increasing or decreasing order. Specifically, given a sorted list i of n elements, then

$$i_1 \geq i_2 \geq i_3 \geq \dots \geq i_n \quad \text{or} \quad i_1 \leq i_2 \leq i_3 \leq \dots \leq i_n$$

Most C compilers supply the function `qsort()` which is included as part of the standard library. The study and understanding of sorting is important for three reasons. First, a generalized function like `qsort()` cannot be applied to all situations. Second, as the `qsort()` function is generalized to run on a wide variety of data, it runs more slowly than a similar sort that operates on only one type of data. This is because the generalization process inherently increases run time because of the extra processing time needed to handle various data types. Third, although the `qsort()` program is very effective in the general case, it may not be the best sort for specialized applications.

There are two general categories of sorting algorithms: algorithms that sort arrays (both in memory and in random access disk files) and algorithms that sort sequential disk or tape files. In this research, we are concerned only with the former, the arrays in random access memory.

Generally when the information is sorted, only a portion of that information is used as the sort "key". This key is used in comparisons, but when an exchange is made the entire data structure is swapped. For example, in an address database the zip code field may be used as the key, but the entire address is sorted.

There are three general methods that can be used to sort arrays:

- 1) Exchange
- 2) Selection
- 3) Insertion

All other methods are usually a combination of the above three methods.

4.1.1 Judging Sorting Algorithms

There are many different algorithms for each classification of sorting method. They all have some merits, but the general criteria for a sorting algorithm are

- 1) How fast can it sort information in an average case?
- 2) How fast are its best and worst cases?
- 3) Does it exhibit natural or unnatural behavior?
- 4) How much memory will be needed for its working?

Clearly, how fast a particular algorithm sorts is of great concern. It can be shown that the speed with which an array is sorted is directly related to the number of comparisons and the number of exchanges with the number of exchanges taking more time. A *comparison* occurs when one element is compared to another; an *exchange* occurs when two elements are swapped in an array. A *move* occurs when one element is moved from one location in memory to another which may or may not be a temporary memory location. Exchange of two elements during sorting requires three moves.

A sort is said to exhibit natural behavior if it works least when the list is already in order, works harder as the list becomes less ordered, and the hardest when the list is in inverse order. How hard a sort works is based on the number of comparisons and exchanges that are executed. Memory requirements for sorting varies with each algorithm. Some algorithms need a lot of internal memory to sort efficiently.

4.1.2 Example of exchange sort

The most well known sort is the bubble sort. The bubble sort is in the exchange category of sorting. The general concept is the repeated comparison and, if necessary, exchange of adjacent elements. It is slowest among sorts but

has the advantage of very low memory requirements. The algorithm of the bubble sort is given in figure 4.1.

```

for  $i := 1$  to  $n-1$  do
    for  $j := n$  downto  $i+1$  do
        if  $a[j] < a[j-1]$  then
             $\text{swap}(a[j], a[j-1]);$ 
        end;
    end;
end;

 $n$  = number of values to be sorted.
 $i, j$  = counters.
 $\text{swap}$  = A function that interchanges the position
          of the two parameters.

```

Figure 4.1 Algorithm for bubble sort.

The bubble sort is said to be an n -squared algorithm. In analyzing any sort it is important to determine how many comparisons will be performed for the best and the worst case. In the bubble sort the number of comparisons is always the same because the two 'for' loops (see figure 4.1) in the algorithm repeat the specified number of times whether the list is initially ordered or not. Hence the bubble sort always performs

$$(1/2) * (n^2 - n) \text{ comparisons}$$

where n is the number of elements to be sorted. This formula is derived from the fact that the outer loop executes $(n-1)$ times and the inner loop executes $(n/2)$ times.

The number of exchanges is zero for the best case for an already sorted list. The number of moves for the worst and average case exchanges are [13]

worst: $(3/2) * (n^2 - n)$

average: $(3/4) * (n^2 - n)$

4.1.3 Example of sorting by selection

A selection sort selects the element with a lowest value and exchanges it with that of the first element. Then from the remaining $(n-1)$ elements the element with the least key is found and exchanged with the second element, and so forth. The exchanges continue to the last two elements. The basic algorithm is shown in figure 4.2.

```
var
    lowkey : keytype; { currently smallest key found
                        on a pass through a[i]...a[n] }
    lowindex : integer; { the position of lowkey }
begin
    for i := 1 to n-1 do begin
        {select the lowest among a[i]...a[n] & swap
         it with a[i]}
        lowindex := i;
        lowkey := a[i];
        for j := i+1 to n do
            {compare each key with current lowkey}
            if a[j] < lowkey then begin
                lowkey := a[j];
                lowindex := j
            end;
        swap (a[j],a[lowindex])
    end
end;
```

Figure 4.2 Algorithm for selection sort

The outer loop executes $(n-1)$ times and the inner loop $((1/2)n)$ times. The result is that the selection sort has $((1/2)*(n^2-n))$ as the number of comparisons. The best and worst cases for moves are [13]

best: $3(n-1)$

worst: $((n^2)/4) + 3(n-1)$

For the best case, if the list is ordered, then only $(n-1)$ elements need to be moved, and each exchange requires three moves. The average case is

average: $n(\ln(n) + y)$

where y is the Euler's constant, about 0.577216.

4.1.4 Example of insertion sort

The insertion sort is the third and last of the simple sorting algorithms. The insertion sort initially sorts the first two members of the array. Next, the algorithm inserts the third member into the sorted position. Then the fourth element is inserted into the list of three elements. The process continues until all the elements are sorted. The algorithm of the insertion is given in figure 4.3.

```
a[0] := -∞ ;
for i := 2 to n do begin
    j := i;
    while a[j] < a[j-1] do begin
        swap (a[j],a[j-1]);
        j := j-1;
    end
end
```

Figure 4.3 Algorithm for insertion sort

In the selection sort, the number of comparisons that occur while using the insertion sort depends upon how the list is initially ordered. If the list is in order then the number of comparisons is $(n-1)$. If it is totally out of order (or sorted in the reverse direction), then it is

$$((1/2)(n^2+n))-1$$

For the average case the number of comparisons is

$$\text{average: } ((1/4)(n^2 + n - 2))$$

The number of exchanges are [9]

$$\text{best: } 0$$

$$\text{worst: } 1/2(n^2 + 3n - 4)$$

Therefore, for worst cases the insertion sort is as bad as the bubble sort and the selection sort. The advantage of the insertion is that it works the least

when the array is initially sorted and the hardest when the array is sorted in inverse order. This means that it behaves naturally.

4.2 Improved Sorts

All the algorithms discussed so far have an execution time of order n -squared. This means that for large amounts of data the sorts are very slow. Fortunately there are sorts which are faster than those discussed so far. We shall describe two of them in detail. The first one is the Shellsort and the other is the quicksort.

4.2.1 Shell sort

The Shell sort is named after its inventor, D. L. Shell. The general method is derived from the insertion sort based on diminishing increments. First, all the elements that are three positions apart are sorted. Then all elements that are two positions apart are sorted. Finally, all those adjacent to each other are sorted. For example consider the array x_1, x_2, \dots, x_n . In the first step the elements (x_1, x_4, x_7, \dots) , (x_2, x_5, x_8, \dots) and (x_3, x_6, x_9, \dots) are sorted. In the second step elements $(x_1, x_3, x_5, x_7, \dots)$ and elements $(x_2, x_4, x_6, x_8, \dots)$ are sorted. Finally the elements adjacent to each other are sorted. The exact sequence for the increments can be changed. The only rule is that the last increment must be one. For example the sequence

9, 5, 3, 2, 1

is widely used. The execution time for the Shell sort is of the order of $(n^{1.5})$ [3]. The algorithm of the Shell sort is given in figure 4.4.

```

procedure shellsort(var a:array[1..n] of integer);
var
    i,j,incr: integer;
begin
    incr := n div 2;
    while incr > 0 do begin
        for i:= incr + 1 to n do begin
            j := i - incr;
            while j > 0 do
                if a[j] > a[j+incr] then begin
                    swap(a[j],a[j+incr]);
                    j := j - incr;
                end
                else
                    j := 0 {break}
                end
            end
            incr := incr div 2
        end
    end; {shellsort}

```

Figure 4.4 Algorithm for shell sort

4.2.2 The Quicksort

The quicksort, invented by C. A. R. Hoare, is a superior sort algorithm. It is generally considered the best sorting algorithm currently available.

The quick sort is built on the idea of partitions. The general procedure is to select a value called the comparand and then to partition the array into two sections. All the elements equal to or greater than the comparand are on one side and those less than the value on the other. This process is repeated for each remaining section until the array is sorted.

The process of selecting a comparand is essentially recursive and therefore quicksort algorithms are recursive algorithms. The selection of the middle comparand value can be accomplished in two ways. It can either be chosen at random or it can be by averaging a small set of values taken from the array. For optimal sorting it is desirable to select a value in the middle of a range of values. In the worst case the value is chosen at one extreme. The operation of quicksort is shown figure 4.5 [3]. It can be derived that the

average number of comparisons is $n \log(n)$ [13] and the number of exchanges is approximately $(n/6) \log(n)$ [13]. This is better than all the other sorting algorithms discussed so far. Thus the quicksort is an $O(n \log n)$ algorithm in its average case and an $O(n^2)$ algorithm in its worst case.

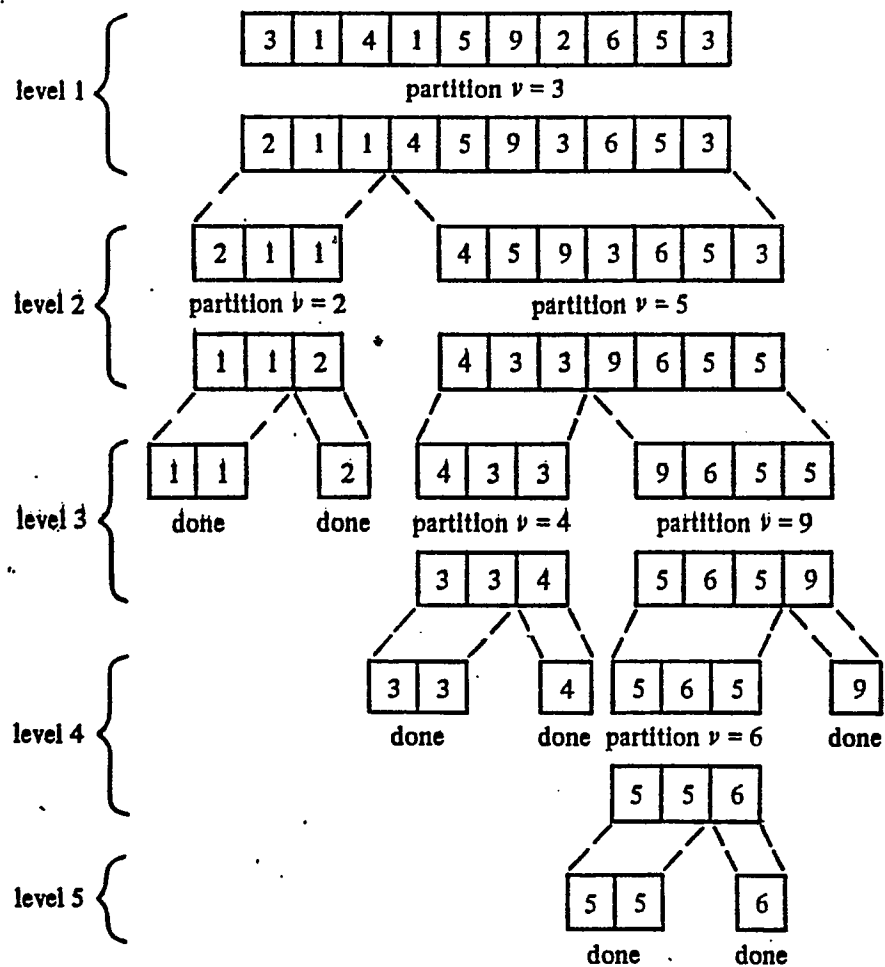


Figure 4.5 Operation of quicksort

4.3 Parallel sorting algorithms.

As seen from the examples in the previous section the algorithms were serial. However with the advent of parallel processing, parallel sorting algorithms were also investigated.

In order to state the problem of parallel sorting clearly, we must define what is meant by a sorted sequence in a parallel processor. When processors share a common memory the idea of contiguous memory location is similar to that in a serial processor. Thus, as in the serial case, the time complexity of a sorting algorithm can be expressed in terms of a number of comparisons (performed in parallel by some or all the processors) and the number of exchanges. Whereas in some architectures the processors do not share memory and communicate along the lines of an interconnection network, the definition of the sorting problem requires a convention to order the processors and thus the union of their memory locations. When parallel processors are used, the time complexity of a sorting algorithm is expressed in terms of parallel comparisons and exchanges between processors that are adjacent in the interconnection network [4].

In the previous section we discussed several sorting algorithms that were essentially serial in nature. In the discussion to follow we shall see how some of these algorithms can be parallelized to give us a faster sort. The speedup of a parallel sorting algorithm can be defined as the ratio between the number of comparison-moves required by a serial sorter and the number of comparison-exchanges required by the parallel algorithm [4].

As parallel processing enables us to perform more than a single comparison, many models had several comparisons being done at the same time. In one model a key is compared to $(n-1)$ other keys in a single time unit using $(n-1)$ processors [10]. However the concurrency that is achieved this way

is limited by the number of processors or by the interconnection scheme between the processors.

There are mainly two types of serial algorithms, those which take $O(n \log n)$ comparisons and those that take $O(n^2)$. It would appear that by using n processors, n elements can be sorted in $O(\log n)$ parallel comparisons. However these algorithms have serial constraints.

On the other hand parallelizing algorithms of the order $O(n^2)$ comparisons can be easily done. This approach produces $O(n)$ -time sorting when n processors are used to sort n elements. This is illustrated by parallelizing the bubble sort. This sort is also called the odd-even transposition sort and is discussed shortly. Another serial algorithm which can be easily parallelized is the sort using tree selection. The result is a parallel tree sort which will be shortly described. Finally, one implementation of the parallel quicksort algorithm is suggested. The algorithms are described in the following sections and their implementations are discussed in the next chapter.

4.3.1 The odd-even transposition sort

The serial "bubble sort" algorithm has been explained in the previous section. The serial odd-even transposition is a variation of the bubble sort, with a total of n phases, each requiring $n/2$ comparisons. Odd and even phases alternate. During the odd phase, odd elements are compared with their right adjacent neighbors, thus the pairs $(x_1, x_2), (x_3, x_4), \dots$ are compared. During an even phase, even elements are compared with their right adjacent neighbors; that is, the pairs $(x_2, x_3), (x_4, x_5), \dots$ are compared. To completely sort the sequence, n phases (alternately odd and even) are required [9].

This algorithm can be easily parallelized. Let us take the case to sort n elements. Consider n linearly connected processors and label them P_1, P_2, \dots, P_n . Assume that the links are bidirectional so that P_i can communicate with both P_{i-1} and P_{i+1} . Also assume that initially x_i resides in P_i for $i = 1, 2, \dots, n$. To sort (x_1, x_2, \dots, x_n) in parallel, let P_1, P_3, P_5, \dots be active during the odd time steps and execute the odd phases of the serial odd-even transpositions. Similarly let P_2, P_4, \dots be active during the even time steps, and perform the even phases in parallel. A single comparison-exchange requires two transfers. For example, during the first step x_2 is transferred to P_1 and compared with x_1 . If $x_1 > x_2$, x_1 is transferred to P_2 ; otherwise x_2 is transferred back to P_2 . Thus the parallel odd-even transposition algorithm sorts n numbers with n processors in n comparisons and $2n$ transfers.

A method to implement this algorithm with five Transputers is discussed in the next chapter. The Transputers are connected in a linear fashion to form a network.

4.3.2 Parallel Tree sort Algorithm

In a serial tree selection sort, a binary tree data structure with $(2n-1)$ nodes is used to sort n numbers. The tree has n leaves and initially one number is stored in each leaf. We then map all the elements on a binary tree data structure. Sorting is performed by selecting the minimum of the n numbers, the minimum of the remaining $(n-1)$ remaining numbers and so on.

Figure 4.6 shows a part of the binary tree structure. The binary tree structure is used to find the minimum by iteratively comparing the numbers in two sibling nodes and moving the smaller number to the parent node. By simultaneously performing all the comparisons at the same level of the binary tree, a parallel tree sort is obtained [Ref 5].

Consider a set of $(2n-1)$ processors interconnected to form a binary tree with one processor at each of n leaf nodes and at each interior node of the tree. By starting with one number at each leaf processor, the minimum can be transferred to the root processor in $\log_2 n$ parallel comparison and transfer steps [Ref 5]. At each step, a parent receives an element from each of its two children, performs a comparison, retains the smaller element, and returns the larger one. After the minimum has reached the root it is written out. From then on empty processors are instructed to accept data from non empty children and select the minimum if they receive two elements. At every other step, the next element in increasing order reaches the root. Thus sorting is completed in time $O(n)$.

The implementation of the above algorithm is discussed in the next chapter. As we do not have n processors available it would not be a straightforward implementation. The load has to be evenly distributed among all the processors.

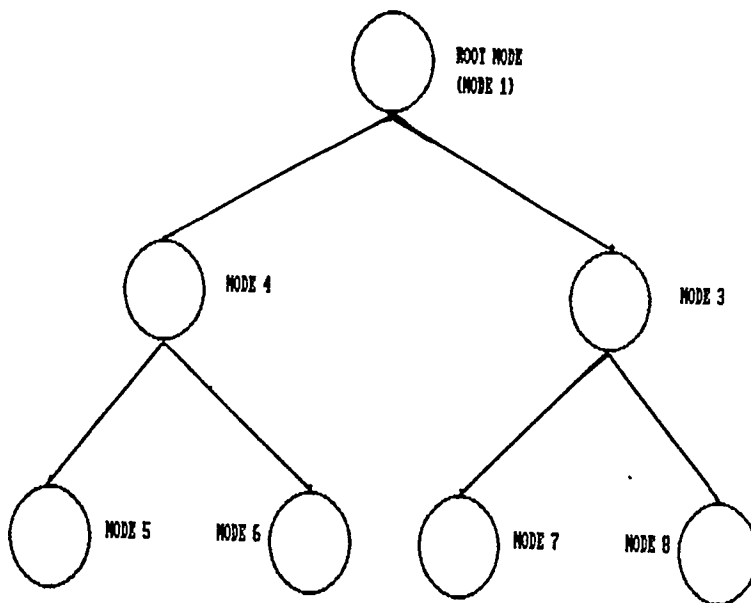


Figure 4.6 Binary tree structure

4.3.3 Parallel Quicksort algorithm

The Quicksort algorithm discussed in section 4.2.2 is an $O(n \log n)$ algorithm and probably the most efficient for internal sorting. We have used a straightforward implementation of parallel quicksort. A binary tree structure of figure 4.6 is used for our implementation.

The root node or parent node goes through a list of numbers to be sorted. We assume that the data set is evenly distributed for a predetermined range. The root node sends the data points to its child processors depending on where the points lie within the predetermined range. For example, if we assume that the numbers are evenly distributed from 0 to N we can divide the data set into three parts. One part containing values below $N/3$, the second part containing values between $N/3$ and $2N/3$, and the third part containing values between $2N/3$ and N . The root node starts scanning the numbers in the array to be sorted. It send the elements to its respective children depending on where they lie in the range. The elements in the range 0 to $N/3$ are kept in the root node itself. All elements between the values $N/3$ and $2N/3$ are sent to say the right child. The rest of the elements are sent to the left child. Once the root has finished sending out data to its children all three nodes do a quicksort on the data available within each node. The results are sent back to the root node. The efficiency of this algorithm like in all other quicksort algorithms depends on the selection of good pivot points. The many ways a pivot can be chosen has been discussed in section 4.2.2. In this case the computation of pivots can take place simultaneously in the children as they receive data from their parent.

Chapter 5

Implementation

In this chapter we shall see how the parallel sort algorithms discussed in the previous chapters are implemented on a network of transputers. The transputer assembly language has many features which help compiler writers take full advantage of concurrent nature of a parallel algorithm. Occam is the language which was specifically designed with the transputer in mind [8]. It is intended by its devisers as the "assembly code" of the transputer. Hence it is with a Occam program that one will realize the full potential transputers. But Occam has its drawbacks. It is a very specialized language and not a very popular language. The most popular language in academic circles is the "C" language. The compiler for Occam was not available for this research. However the TCX "C" discussed in chapter 3 has many features to implement concurrency. Hence we choose "C" to code the algorithms to be implemented.

This chapter is divided into two parts. In the first part, an interface to the concurrency features of the transputers is described. The interface is a library of C functions that allow the user to implement multiple processes, interprocess communication through message passing or shared data, and semaphores. Once a user understands the working of these concurrent mechanisms, it is then a matter of simple programming in C and running the program on the network. A description of how to compile, load and run the C programs has been given in chapter 3. In the second part we discuss the

implementations of parallel sorting algorithms which have been the subject of this research.

5.1 Introduction to Processes, Channels and Semaphores.

The transputer has a large set of instructions for implementing concurrent systems. This set includes instructions to start, stop, and pass messages between processes. The hardware includes features to block and restart processes waiting on communication and to select a new process after a predetermined timeslice has elapsed, placing the timesliced process at the end of an active queue.

5.1.1 Concurrency

Before a new process can be executed, a stack frame must be allocated for it. The transputer contains a single 32 bit linear address space that all processes execute within. Stack space for a process is allocated using the `malloc()` command in the C language. This is a function which allocates a region of heap memory large enough to hold an object of a specified number of bytes. After allocating the space, the stack frame is initialized to a valid state for the process to begin executing. A new process is allocated as follows:

```
# include <conc.h>
Process *ProcAlloc(func, sp, nparam, p1, p2, ... , pn)
    int (*func)();
    int sp;
    int nparam;
```

`ProcAlloc()` takes a pointer to a function that contains the code for the process. The parameter 'sp' indicates the amount of stack space required for that process. `nparam` specifies the number of words of space off the stack initially for parameters `p1, p2, p3,...` to the function.

On successful completion, ProcAlloc() returns a pointer to the structure that constitutes the process. ProcAlloc() uses malloc() to allocate the space for the Process structure and the stack space for the process. Once a process is initiated or allocated the parameters can be altered. There are several routines for executing processes. They are listed below:

```
# include <conc.h>

ProcRun(p)
    Process *p;

ProcRunHigh(p)
    Process *p;

ProcRunLow(p)
    Process *p;

ProcPar(p1, p2, p3, ..., pn, 0)
    Process *p1, *p2, *p3, ... *pn;

ProcParList(plist)
    Process **plist;

ProcPriPar(phigh, plow)
    Process *phigh, *plow;
```

We shall briefly explain the working of the above commands. ProcRun(), ProcRunHigh(), and ProcRunLow() execute unsynchronized processes. The process begins execution and is out of control of the initiating process. The initiating process has no means for determining or altering the state of the created process except through a communication means the user explicitly establishes. ProcRun() executes the process at the priority of the current process, ProcRunHigh() executes the process at high priority and ProcRunLow() executes the process at low priority.

ProcPar(), ProcParList(), and ProcPriPar() start a group of processes. Control is returned to the initiating process when all of the initiated processes terminate. ProcPar() takes an explicit null terminated list of processes; all of

the processes are executed at priority of the current process. ProcPriPar() takes two parameters. The first process is executed at high priority and the second is executed at low priority. ProcPriPar() returns when both processes complete.

5.1.2 Interprocess communication

The transputer supports a message passing protocol for interprocess communication. A *channel* is a unidirectional message stream between two processes. When a process performs input or output to a channel, the process is blocked until the corresponding process performs its respective output or input. This way the channels can be used as a synchronization mechanism. The only caveat is that the two channels must perform operations of the same data size. If one process attempts to output 50 bytes while the corresponding process tries to input 49 bytes, an unpredictable operation will result. There are six routines for performing communication along channels:

```
# include <conc.h>
ChanOut (c, cp, cnt)
    Channel *c;
    char *cp;
    int cnt;

ChanOutChar(c, ch)
    Channel *c;
    char ch;

ChanOutInt(c,n)
    Channel *c;
    int n;

ChanIn(c, cp, cnt)
    Channel *c;
    char *cp;
    int cnt;

int
ChanInInt(c)
    Channel *c;
```

```

char
ChanInChar(c)
    Channel *c;

```

We shall briefly describe the working of the above commands. The ChanIn() function reads *n* bytes of data, from the "channel" pointed to by *c*, to the buffer pointer *cp*. The ChanInChar and ChanInInt functions may be used to read, and return the value of a byte or word, respectively, read from the "channel" pointed to by *c*. The function ChanOut() is used to write *n* bytes of data to the "channel" pointed to by *c*, from the buffer pointed to by *cp*. ChanOutChar and ChanOutInt functions may be used to write the value of a byte or word of type character or integer, respectively, to the "channel" pointed to by *c*.

As channels require initialization before they can be used for communication, there are two more routines provided for channel allocation and initialization. ChanReset() resets a channel, returning information already contained in the channel, ChanAlloc() returns a pointer to an initialized channel.

The concept of channels extends beyond the bounds of a single transputer. The four serial links of a T800 correspond to eight channels (four input and four output) with specific hardware addresses. These addresses are contained in the 'conch.h' file that must be included in the "C" program if the above routines are to be used. The physical addresses of these links are given below:

```

/* Addresses for physical links on T800 */
#define LINK0OUT ((Channel *) 0x80000000)
#define LINK0IN  ((Channel *) 0x80000004)
#define LINK1OUT ((Channel *) 0x80000008)
#define LINK1IN  ((Channel *) 0x8000000c)
#define LINK2OUT ((Channel *) 0x80000010)
#define LINK2IN  ((Channel *) 0x80000014)
#define LINK3OUT ((Channel *) 0x80000018)
#define LINK3IN  ((Channel *) 0x8000001c)

```

5.1.3 Alternation

A series of calls are available to determine the status of channels and possibly wait until a channel is ready for input. There are six alternation routines:

```
int
ProcAlt(c1, c2, ..., cn, 0)
    Channel *c1, *c2, ..., *cn;

int
ProcAltList(c1list)
    Channel **c1list;

int
ProcSkipAlt(c1, c2, ..., cn, 0)
    Channel *c1, *c2, ..., *cn;
int
ProcSkipAltList(c1list)
    Channel **c1list;

int
ProcTimerAlt(time, c1, c2, ..., 0)
    Channel *c1, *c2, ..., *cn;

int
ProcTimerAltList(time, c1list)
    Channel **c1list;
```

ProcAlt() and ProcAltList() cause the current process to block until one of its channels in its argument list is ready for input. On completion the routine returns an index into the parameter list for the channel ready for input.

ProcSkipAlt() and ProcSkipAltList() check specified channels. If one of the channels is ready for input, an index into parameter list is returned, otherwise a -1 is returned. These routines do not block waiting for one of the channels, they return immediately.

ProcTimerAlt() and ProcTimerAltList() block the current process until one of the channels is ready for input or until the timer times out the value

specified. If the timer times out, a -1 is returned, otherwise an index into the parameter list is returned.

5.1.4 Semaphores

The library provides a semaphore facility. This has been implemented by the "C" compiler although a semaphore facility is not explicitly supported by the transputers. A semaphore can be initialized in two ways:

```
#include <conc.h>
Semaphore sem = SEMAPHOREINIT;

/*      or      */

Semaphore *newsem;
newsem = SemAlloc();
```

A semaphore is acquired with the following command

```
#include <conc.h>
SemP(sem)
    Semaphore sem;
```

A semaphore can be released with the following command

```
#include <conc.h>
SemV(sem)
    Semaphore sem;
```

SemP() and SemV() take the semaphore as a parameter rather than a pointer to the semaphore. SemP() blocks the current process and places it on a queue if the semaphore is in use, otherwise it sets the semaphore acquired and execution continues. This routine will not return until the process acquires the semaphore. SemV() releases the semaphore and runs the first process on the queue if any processes are waiting.

5.1.5 Reliable communication

On occasion it may be necessary to attempt communication along a channel that may be inoperative. The standard routines such as ChanIn() and ChanOut() are inadequate for this task. The standard routines causes the initiating processes to stall until the communication completes. If the communication never completes due to a faulty connection, the processes will never continue. Four routines are provided to ensure that reliable communication is achieved.

```
ChanOutTimeFail(chan, cp, cnt, time)
    Channel *chan;
    char *cp;
    int cnt;
    int time;

int
ChanOutChanFail(chan, cp, cnt, failchan)
    Channel *chan, *failchan;
    char *cp;
    int cnt;

ChanInTimeFail(chan, cp, cnt, time)
    Channel *chan;
    char *cp;
    int cnt;
    int time;

int
ChanInChanFail(chan, cp, cnt, failchan)
    Channel *chan, *failchan;
    char *cp;
    int cnt;
```

These routines are similar to the ChanIn() and ChanOut() functions discussed above. They have an additional "time" parameter that allows the process to reschedule based on a terminating condition. In the case of ChanInTimeFail() and ChanOutTimeFail(), communication is attempted until the clock reaches the value of the "time" parameter. If communication completes before the timeout, the routines return a status of 0. If the

communication has not completed when the timeout occurs, the offending channel is reset and a status of 1 is returned from the function.

The above routines should be used to establish the integrity of the link between two unfamiliar processes and not as a standard method of communicating between two processors.

```
#include <stdlib.h>
#include <conc.h>
#define WS_SIZE 512

main()
{
    if (_node_number==1)
    {
        int i,j;

        j = ChanInInt(LINKLIN);

        printf("received from node 2: j= %d\n",j+100);

        printf("All Done\n");
    }
    if (_node_number == 2)
    {
        int i;
        i = ChanInInt(LINKLIN);
        i = i + 100;
        ChanOutInt(LINKOOUT, i);
    }
    if (_node_number == 3)
    {
        int i;
        i=ChanInInt(LINKLIN);
        i = i + 100;
        ChanOutInt(LINKOOUT, i);
    }
    if (_node_number == 4)
    {
        int i;
        i= ChanInInt(LINKLIN);
        i = i + 100;
        ChanOutInt(LINKOOUT, i);
    }
    if (_node_number == 5)
    {
        int i;
        i = 100;
        ChanOutInt(LINKOOUT, i);
    }
}
```

RESULT:

C:\> ld-net ['nif' file]

received from node 2: j=500

Figure 5.1 Example program and results

5.2 An example program

An example program is shown in figure 5.1. This program illustrates how data can be passed from one transputer to another by means of link connections. The variable "_node_number" is declared in the concurrency library file "conch.h". Hence this file is included in the program header. This variable holds the network address for this node. There are two other variables which are of interest to us. They are "_boot_chan_in" and "_boot_chan_out". These variables are pointers to the "hard" input and output channels which were used to bootstrap the transputer node.

The transputer network configuration used for this program is the chain shown in figure 2.9. The program is compiled and loaded using the commands described in chapter 3. These are the compiler, assembler, linker and the network loader. The network information file ('nif' file) used while loading the program is shown in figure 5.2. The network is bootstrapped and each transputer node has the executable program it is supposed to run. In this case the executable program is the same for every transputer. Now the programs begin to run in each node. In this case there are five and so five processes start running simultaneously.

```
1,   testrun,  R0,  0,  2,   ,   ;
2,   testrun,  R1,  1,  3,   ,   ;
3,   testrun,  R2,  2,  4,   ,   ;
4,   testrun,  R3,  3,  5,   ,   ;
5,   testrun,  R4,  4,   ,   ,   ;
```

Figure 5.2 'nif' file used for the example program

The 'if' statement in the program decides which part of the code segment is to be executed by which transputer. The node five sends a data out on its link 0 known to the program as LINK0OUT. This link is connected to the link 1 known to the program as LINK1IN of the node four transputer. The node four executes its program segment which directs it to read the data from the output LINK0OUT of node five. Thus node 4 is able to get the data from node five. Now node four send the modified data to node three on its output link which is connected to the input link of node 3. Node 3 reads this data and modifies it.

This way the data is transmitted from one transputer node to another. Finally node one gets the data and prints it on the screen of the host computer. Thus this example illustrates how processes can be run simultaneously on a transputer network.

5.3 Implementation of odd-even transposition sort

This algorithm has already been discussed in section 4.3.1. The transputer network configuration for this program is shown Figure 5.3. The modules used for this program are shown in figure 5.4. In the ideal case we would be dealing with n processors where n is the number of items to be sorted. Let us first discuss the implementation for the ideal case and then let us see how it is applied to our case of five transputer nodes available.

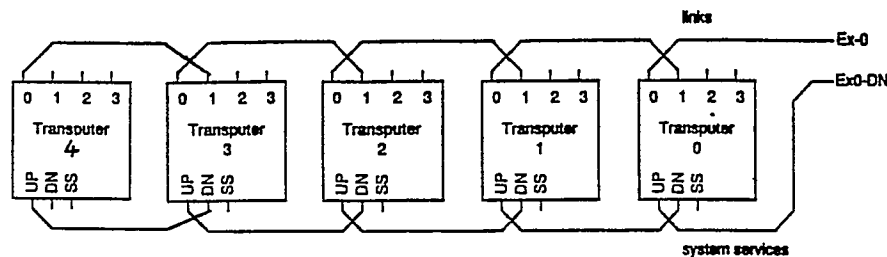
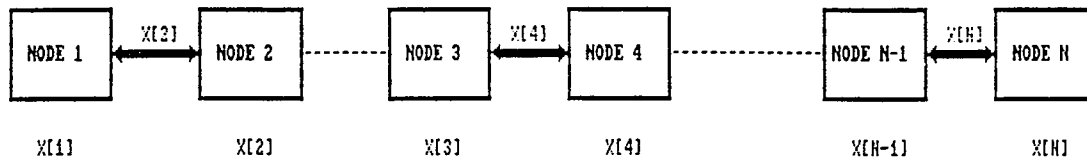
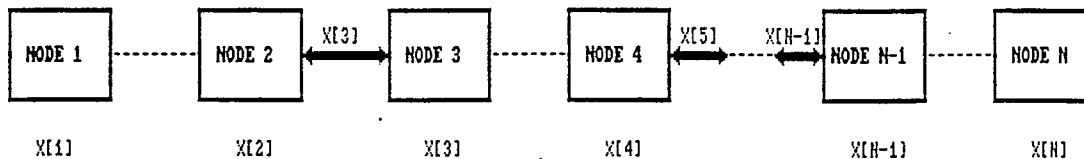


Figure 5.3 Network topology used for the odd-even transposition sort

As the same copy of the executable program is loaded onto each node of the transputer network all the nodes have a copy of the data to be sorted. There are two modules in the program. Each module is executed consecutively in a loop n times. The two modules are named 'odd' and 'even'.



ODD MODULE : Each odd numbered node gets a data value from the node on its right. This value and the value resident in each node is compared. The smaller value is retained and larger value sent back.



EVEN MODULE : Each even numbered node gets a data value from the node on its right. This value and the value resident in each node is compared. The smaller value is retained and larger value sent back.

Figure 5.4 Odd-even transposition sort

The 'odd' module initiates processes on the transputer nodes to get the data item from the processor to its right. For example, node one will get its data from node two, and so on. The data sent out is compared to the data each processor has. The comparisons are done only by the odd numbered nodes. Hence the name odd cycle. If the data sent out is greater than the data resident in each processor the data is sent back on the same link. If the comparison shows the data received to be smaller than the data resident in node two, the greater value is sent out on the output link and the lower value now resides in each active processor.

During the execution of the 'even' module all the even numbered processors are 'active' in the sense that they do the comparisons. The data is sent out by the odd numbered processors are compared with the data present in the even numbered processors. The value which is greater is sent back to the node that sent the data and the lower value is resident in the node which receives data.

The 'odd' modules and 'even' modules alternately perform their tasks n times. At the end of this cycle the data is sorted and printed. As we only have 5 nodes the above scheme has to be modified to be able to run successfully. This modification is discussed below.

The basic idea to do the sort now is the same in that 'odd' and 'even' modules are alternately executed until the data is sorted. The operation of the modules are changed slightly. Let the numbers to be sorted be stored in the array $x[0], x[1], x[2], \dots, x[n]$. We start by making the assumption that $n=49$. We have implemented the algorithm for a five transputer chain. The values $x[0]$ to $x[9]$ are present in node 1 and the values $x[10]$ to $x[19]$ are present in node 2 and so on. The description of odd and even phases follow.

During the 'odd' phase each node performs the 'odd' cycle of the general case to the set of ten numbers it contains. For example node 1 compares and exchanges $x[0]$ and $x[1]$ in such a way that the smaller value is stored in $x[0]$. Then it compares and exchanges $x[2]$ and $x[3]$ in such a way that the smaller value is stored in $x[2]$. The last comparison for node one would be that of $x[8]$ and $x[9]$. All the other nodes do similar operations simultaneously. Thus at the end of the 'odd' phase each node has performed an 'odd' cycle on its set of ten numbers.

During the 'even' phase each node once again performs an 'even' cycle on the ten numbers it contains. For example the node one compares and exchanges $x[1]$ and $x[2]$ in such a way that $x[1]$ contains the smaller value. It then does the same operation for $x[3]$ and $x[4]$. The last comparison would be of the elements $x[7]$ and $x[8]$. The other nodes do the same operations with the set of ten numbers resident in each node. However, the number $x[9]$ is present in node 1 and $x[10]$ is present in node 2. Similarly the numbers $x[19]$ and $x[20]$ are resident in node 2 and node 3 respectively. The 'even' phase must be performed in these numbers too, to sort the elements correctly. This is accomplished by sending these values to the node on left of every node. The nodes compare the elements received with the elements in the top of its array and sends back the larger value on to the node on its right. Node 1 does not send the value $x[0]$ and node 5 does not receive a value to be compared with $x[49]$. Thus the 'even' phase is accomplished.

The 'odd' and 'even' phases are executed in a loop $n/2$ times. In this example we have 50 numbers to sort, so this looping is performed 25 times and the data is sorted. This program can be easily modified to sort any set of numbers greater than that considered in its implementation. The program implementing this algorithm is given in the appendix.

5.4 Implementation of the parallel tree-sort algorithm

The parallel tree algorithm is discussed in section 4.3.2. The implementation is fairly straightforward. The fact that the transputer has four links helps in the implementation. The network topology is a binary tree structure as shown in figure 2.10. The tree structure with the link connections is shown in figure 5.5. Every node except the nodes in the last row and the root node use 3 links. Every parent uses two links to connect to two of its children. Each child has two links to connect to its children.

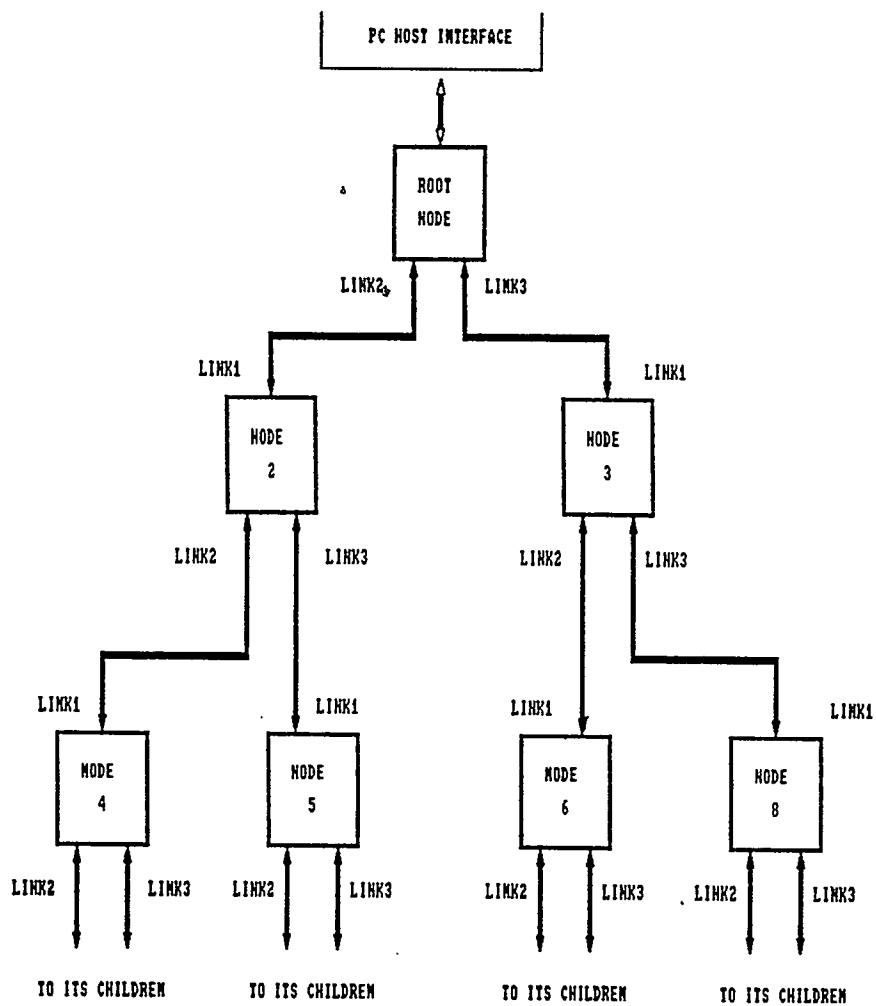


Figure 5.5 Transputer link connections in binary tree structure

Consider a set of $(2n-1)$ nodes as shown in the figure 5.6. At each step, a parent receives an element from each of its two children. Consider a parent node. It receives an element from each of its two children, performs a comparison retains the smaller element and returns the larger element. In this topology each transputer has its link 1 connected to its parent and each parent has link 2 connected to its left child and link 3 connected to its right child. This way the minimum can be transferred to the root node in $\log_2(n)$ parallel comparison and transfer steps. After the minimum has reached the root it is written out. From then on, empty nodes are instructed to accept data from non-empty children and select the minimum if they receive two elements. The program is given in the appendix. This way every next step will bring the minimum in increasing order. Thus sorting is achieved using the binary tree structure. The network information file for such a network is shown in figure 5.6.

```

1,  program,  R0,  0,  2,  3,  ;
2,  program,  R1,  ,  1,  4,  5;
3,  program,  S1,  ,  1,  6,  8;
4,  program,  R2,  ,  2,  9,  10;
5,  program,  S2,  ,  2,  11, 12;
6,  program,  R3,  ,  1,  13, 14;

```

Figure 5.6 'nif' file for the binary tree topology

5.5 Implementation of the parallel quicksort algorithm

The network topology used for the implementation of the quicksort is similar to the topology used for the parallel tree selection sort given in figure 5.5. Node 1 is connected to its children, node 2 and node 3 with link 1 and link 2 respectively.

While running a sort on a set of numbers one usually knows the range of values one is dealing with. For example, a meteorologist may know the range of values of temperatures collected over a period of time. This knowledge of the data one is dealing with greatly helps the programmer who codes sorting algorithms. We have assumed that the data set to be sorted is evenly distributed in a given range. This assumption is necessary so that the processors may share an equal or almost equal load. If the data set is not evenly distributed then the program still works, but not at its best case performance. In other words, the quicksort degenerates into a serial quicksort which might take $O(n^2)$ time. But usually this is not the case and the average time taken will be of the order of $O(n \log n)$.

As we use three processors, the algorithm we use would be of the order of $O((n/3) \log n)$ [5]. There are many ways of finding a pivot to partition the data points in the quicksort algorithm. The pivot is the element around which we rearrange the elements in the array in a recursive fashion. In some cases the median is computed so that it is preceded by about half the keys and followed by about half [3]. In some cases it is chosen at random [13]. In our case as we have assumed an even distribution; we have taken specific values as pivot points.

In our implementation, we sorted a set of numbers in the range 0 to 100. The numbers were divided by the root node in such a way that all numbers in the range 30 to 60 were sent to node two and all numbers in the range 60 to

100 were sent to node three. The numbers in the range 0 to 30 were retained by the root node. Quicksort is performed by the three nodes independently and the results are sent back to node 1 for display. This program is given in the appendix. Thus parallel quicksort was implemented using three nodes.

Chapter 6

Conclusion

Sorting is a candidate for parallel solution because many algorithms have an element of divide-and-conquer. That means the task is carried out by dividing it into some number of smaller, simpler tasks each of which is repeatedly divided until only trivial tasks remain. Such a strategy identifies independent parts of the original problem, which can be tackled concurrently.

In this thesis parallel sorting algorithms have been implemented on a network of transputers. By using transputers and a "C" language compiler which exploits the concurrency features of transputers, one can code algorithms which perform parallel processing. The way to realize the full performance potential of any multiprocessor system lies in the even distribution of the workload. It has been seen that the transputer offers a good programming environment to work with parallel algorithms. Any application which runs at present on an array processor can be implemented on an array of transputers.

It has been reported that a lot of time has been spent by system designers trying to devise mechanisms for interprocess communications, task scheduling and process synchronization [11]. By providing a software environment such as the one used for this research the system designer can concentrate on aspects of concurrency and architecture of large systems.

Many parallel algorithms such as for sorting can be developed and implemented.

In this thesis the parallel implementation of serial sort algorithms such as the bubble sort and the binary tree selection sort have been implemented. An algorithm for running parallel quicksort has also been implemented. The connections of the links of each transputer has also been discussed. The concurrency features of the TCX "C" compiler were used while programming the algorithms. These included features to start parallel processes and the use of channels to achieve synchronization between processes and process allocation routines.

The advantage of implementing algorithms with transputers is that the code can be easily modified to run whenever more processors are added to the network. There are many applications such as sorting that can be run faster using parallel processing. The study of such algorithms and techniques is suggested for future work in this area.

Bibliography

- 1.) "C" *Users Guide*, 1989. Book I, "Compiler Assembler and Network Loader". Computer Systems Architects, Provo, Utah.
- 2.) "C" *Users Guide*, 1989. Book II, "Transputer "C" library description and Library documentation". Computer Systems Architects, Provo, Utah.
- 3.) Aho, V. A., Hopcroft, E. J., and Ullman, D. J., 1987. *Data Structures and Algorithms*, Addison-Wesley, Reading, Massachusetts.
- 4.) Bentley, J. L. and Kung, H. T., 1979. "A tree structure for searching problems". In the *Proceedings of the 1979 Internatioanl Conference on Parallel Processing* (Aug).
- 5.) Bitton, D., DeWitt, D. J., Hsiao, D. K. and Menon, J., 1984. "A taxonomy of Parallel sorting", *Computing Surveys*, vol 16. No 3, (Sept 1984).
- 6.) Desrochers, R. G., 1987. *Principles of parallel and Multiprocesing*, McGraw-Hill Book company, New-York, New-York.
- 7.) Hwang, K. and Briggs, A. F., 1984. *Computer Architecture and Parallel Processing*, McGraw-Hill Book Company, New-York, New-York.
- 8.) Jones, G., 1987. *Programming in Occam*, Prentice-Hall, Englewoods Cliffs, New-Jersey.
- 9.) Knuth, D. E. 1973. Sorting and searching. In *The Art of Computer Programming*, vol 3. Addison-Wesley, Reading, Masschusetts.
- 10.) Muller, D. E., and Preparata, F. P., 1975. "Bounds to complexities of networks for sorting and switching". *Journal of ACM* vol 22.
- 11.) Newport, J. P., "The Inmos Transputer", *32-bit Micrprocessors*, CAP Scientific Limited.
- 12.) Rich, P. R. 1972. *Internal Sorting Methods Illustrated with PL/I programs*, Prentice-Hall, Englewood Cliffs, New-Jersey.
- 13.) Schildt, H., C: 1987. "C" *The Complete Reference*, Osborne McGraw-Hill, Berkeley, California.
- 14.) Homewood, M., May, D., Shepherd, R. and Shepherd, D., 1987. "The IMS T800 Transputer". *IEEE Micro* (October).
- 15.) Nicoud, D. J., and Tyrell, A. M., 1989. "The transputer T414 Instruction set". *IEEE Micro*, vol 9, (June).

- 16.) "*PART.6*" *Installation guide and users manual*, 1987. Computer Systems Architects, Provo, Utah.
- 17.) "*PART.2*" *Installation guide and users manual*; 1987. Computer Systems Architects, Provo, Utah.

• Appendix

•

```
/*****
```

ODD EVEN TRANSPOSITION SORT

This program sorts 50 numbers. It uses a 5 node transputer chain. The 'nif' file used while loading the executable program is shown in figure 5.2.

```
*****/
```

```
#include <stdio.h>
```

```
#include <conc.h>
```

```
main()
```

```
{
```

```
    int x[50],i;
```

```
x[0]=10; x[1]=50; x[2]=36; x[3]=16; x[4]= 19; x[5] = 11;
x[6]=30; x[7]=50; x[8]=48; x[9]=25; x[10]=99; x[11]= 31;
x[12]=100; x[13]=500; x[14]=360; x[15]=160; x[16]= 190;
x[17]=110; x[18]=300; x[19]=500; x[20]=34; x[21]=85;
x[22]=90; x[23]=40; x[24]=70; x[25]=3; x[26]=5; x[27]=340;
x[28]=850; x[29]=900;x[30]=109;x[31]=508; x[32]=365;
x[33]=163; x[34]= 189;x[35] =211; x[36]=303; x[37]=506;
x[38]=487; x[39]=250; x[40]=103; x[41]=504; x[42]=367;
x[43]=167; x[44]= 191; x[45] = 111; x[46]=304; x[47]=501;
x[48]=482; x[49]=252;
```

```
    if(_node_number==1)
```

```
    {
```

```
        int stop,a,tmp;
```

```
        stop=25;
```

```
        while(stop > 0)
```

```
        {
```

```
            for(i=0;i<10;i=i+2)
```

```
            {
```

```
                if( x[i] > x[i+1] )
```

```
                {
```

```
                    tmp = x[i];
```

```
                    x[i]= x[i+1];
```

```
                    x[i+1]=tmp;
```

```
                }
```

```
            }
```

```
            for(i=0; i<8; i=i+2)
```

```
            {
```

```
                if( x[i+1] > x[i+2])
```

```
                {
```

```
                    tmp=x[i+1];
```

```
                    x[i+1]=x[i+2];
```

```
                    x[i+2]=tmp;
```

```
                }
```



```

    }

    a=ChanInInt(LINK1IN);
    if(x[9]<a)
    {
        ChanOutInt(LINK1OUT,a);
    }
    else
    {
        tmp=a;
        a=x[9];
        x[9]=tmp;
        ChanOutInt(LINK1OUT,a);
    }
    printf("stop%d ",stop);
    stop-- ;
}
printf("\n");

for(i=10;i<50;i++)
{
    x[i]=ChanInInt(LINK1IN);
}

for(i=0;i<50;i++)
{
    printf(" %d ",x[i]);
}
printf("\n");
}
if(_node_number==2)
{
    int stop,tmp,i,a;
    stop=25;
    while(stop>0)
    {
        for(i=10; i < 20; i=i+2)
        {
            if(x[i] > x[i+1])
            {
                tmp=x[i];
                x[i]=x[i+1];
                x[i+1]=tmp;
            }
        }

        for(i=10; i < 18; i=i+2)
        {
            if(x[i+1] > x[i+2])
            {
                tmp=x[i+1];
                x[i+1]=x[i+2];

```

```

        x[i+2]=tmp;
    }
}

ChanOutInt(LINK0OUT,x[10]);
a= ChanInInt(LINK1IN);
if(x[19]<a)
{
    ChanOutInt(LINK1OUT,a);
}
else
{
    tmp=a;
    a=x[19];
    x[19]=tmp;
    ChanOutInt(LINK1OUT,a);
}
x[10]=ChanInInt(LINK0IN);
stop--;
}

for(i=20;i<50;i++)
    x[i]=ChanInInt(LINK1IN);
for(i=10;i<50;i++)
    ChanOutInt(LINK0OUT,x[i]);
}

```

```

if(_node_number==3)
{
    int stop,tmp,i,a;
    stop=25;
    while(stop>0)
    {
        for(i=20; i < 30; i=i+2)
        {
            if(x[i] > x[i+1])
            {
                tmp=x[i];
                x[i]=x[i+1];
                x[i+1]=tmp;
            }
        }

        for(i=20; i < 28; i=i+2)
        {
            if(x[i+1] > x[i+2])
            {
                tmp=x[i+1];
                x[i+1]=x[i+2];
                x[i+2]=tmp;
            }
        }
    }
}

```

```

    }

    ChanOutInt(LINK0OUT,x[20]);
    a= ChanInInt(LINK1IN);
    if(x[29]<a)
    {
        ChanOutInt(LINK1OUT,a);
    }
    else
    {
        tmp=a;
        a=x[29];
        x[29]=tmp;
        ChanOutInt(LINK1OUT,a);
    }
    x[20]=ChanInInt(LINK0IN);
    stop--;
}

for(i=30;i<50;i++)
    x[i]=ChanInInt(LINK1IN);
for(i=20;i<50;i++)
    ChanOutInt(LINK0OUT,x[i]);
}

if(_node_number==4)
{
    int stop,tmp,i,a;
    stop=25;
    while(stop>0)
    {

        for(i=30; i < 40; i=i+2)
        {
            if(x[i] > x[i+1])
            {
                tmp=x[i];
                x[i]=x[i+1];
                x[i+1]=tmp;
            }
        }

        for(i=30; i < 38; i=i+2)
        {
            if(x[i+1] > x[i+2])
            {
                tmp=x[i+1];
                x[i+1]=x[i+2];
                x[i+2]=tmp;
            }
        }

        ChanOutInt(LINK0OUT,x[30]);
    }
}

```

```

        a= ChanInInt(LINK1IN);
        if(x[39]<a)
        {
            ChanOutInt(LINK1OUT,a);
        }
        else
        {
            tmp=a;
            a=x[39];
            x[39]=tmp;
            ChanOutInt(LINK1OUT,a);
        }
        x[30]=ChanInInt(LINK0IN);
        stop--;
    }

    for(i=40;i<50;i++)
        x[i]=ChanInInt(LINK1IN);
    for(i=30;i<50;i++)
        ChanOutInt(LINK0OUT,x[i]);
}

```

```

if(_node_number==5)
{
    int stop,tmp,i;
    stop=25;
    while(stop>0)
    {

        for(i=40; i < 50; i=i+2)
        {
            if(x[i] > x[i+1])
            {
                tmp=x[i];
                x[i]=x[i+1];
                x[i+1]=tmp;
            }
        }

        for(i=40; i < 48; i=i+2)
        {
            if(x[i+1] > x[i+2])
            {
                tmp=x[i+1];
                x[i+1]=x[i+2];
                x[i+2]=tmp;
            }
        }

        ChanOutInt(LINK0OUT,x[40]);
    }
}

```

```

        x[40]=ChanInInt(LINK0IN);
        stop--;
    }
    for(i=40;i<50;i++)
        ChanOutInt(LINK0OUT,x[i]);
}
}

```

```
/******
```

This program uses 3 nodes to implement a parallel tree sort algorithm. The 'nif' file used to load the executable program is given in figure 5.6

```
*****/
```

```
# include <stdio.h>
# include <conc.h>
```

```
main()
{
```

```
    int a[2];
    a[0]=20;
    a[1]=10;
```

```
    if(_node_number==1)
    {
```

```
        int comp1,comp2,i,b[2];
```

```
        comp1=ChanInInt(LINK1IN);
```

```
        comp2=ChanInInt(LINK2IN);
```

```
        i=0;
```

```
        if(comp1 < comp2)
```

```
        {
```

```
            b[i]=comp1;
```

```
            ChanOutInt(LINK1OUT,1);
```

```
            ChanOutInt(LINK2OUT,0);
```

```
        }
```

```
        else
```

```
        {
```

```
            b[i]=comp2;
```

```
            ChanOutInt(LINK1OUT,0);
```

```
            ChanOutInt(LINK2OUT,1);
```

```
        }
```

```
        i++;
```

```
        if(comp1 < comp2)
```

```
        {
```

```
            b[i]=ChanInInt(LINK2IN);
```

```
        }
```

```
        else
```

```
        {
```

```
            b[i]=ChanInInt(LINK1IN);
```

```
        }
```

```

        printf("sorted values are b[0]= %d, b[1]=
%d\n",b[0],b[1]);
    }

    if(_node_number==2)
    {
        int empty;

        ChanOutInt(LINK0OUT,a[0]);
        empty=ChanInInt(LINK0IN);
        if (empty==1)
        {
            ; /* Get data values from children if any */
        }
        else
        {
            ChanOutInt(LINK0OUT,a[0]);
        }
    }
    if(_node_number==3)
    {
        int empty;

        ChanOutInt(LINK0OUT,a[1]);
        empty=ChanInInt(LINK0IN);
        if(empty==1)
        {
            ; /* Get data values from children if any */
        }
        else
        {
            ChanOutInt(LINK0OUT,a[1]);
        }
    }
}

```

/*****

This program does a parallel quicksort on 30 numbers using 3 nodes of transputers arranged in a binary tree. The 'nif' file used while loading the network is given in figure 5.6.

*****/

```
#include <stdio.h>
#include <conc.h>
```

```
quick(item, count)
int *item;
int count;
{
    qs(item, 0, count-1);
}
qs(item, left, right)
int *item;
int left, right;
{
    register int i,j;
    char x,y;

    i=left; j=right;
    x=item[(left+right)/2];

    do
    {
        while(item[i]<x && i<right) i++;
        while(x<item[j] && j>left) j--;

        if(i<=j)
        {
            y=item[i];
            item[i]=item[j];
            item[j]=y;
            i++;j--;
        }
        }while (i<=j);

    if(left<j) qs(item,left,j);
    if(j<right) qs(item,i,right);
}
```



```

main()
{
    int x[30],i,j,k,l;
    int a[30],b[30],c[30];

    if(_node_number==1)
    {
        for(i=0;i<30;i++)
        {
            scanf("%d",&x[i]);
        }
        /*for(i=0;i<30;i++)
        {
            printf("  %d  ",x[i]);
        }*/
        printf("\n");

        j=0;k=0;l=0;
        for(i=0;i<30;i++)
        {
            if(x[i] <= 30)
            {
                a[j]=x[i];
                j++;
            }
            else if(x[i] > 30 && x[i] <= 60)
            {
                b[k]=x[i];
                k++;
            }
            else
            {
                c[l]=x[i];
                l++;
            }
        }

        /*** data sent to node two ***/
        ChanOutInt(LINK1OUT,k);
        for(i=0;i<k;i++)
        {
            ChanOutInt(LINK1OUT,b[i]);
        }

        /*** data sent to node three ***/
        ChanOutInt(LINK2OUT,l);
        for(i=0;i<l;i++)
        {
            ChanOutInt(LINK2OUT,c[i]);
        }

        printf("\n\nQuicksorting by node #1\n\n");
    }
}

```

```

quick(a,j);
printf("\n");
for(i=0;i<j;++i)
printf("  %d  ",a[i]);

/**/ data back from node two /**/
for(i=0;i<k;i++)
{
    b[i]=ChanInInt(LINK1IN);
}
printf("\n\nQuicksorting by node #2\n\n");
for(i=0;i<k;++i)
printf("  %d  ",b[i]);

/**/ data back from node three /**/
for(i=0;i<l;i++)
{
    c[i]=ChanInInt(LINK2IN);
}
printf("\n\nQuicksorting by node #3\n\n");
for(i=0;i<l;++i)
printf("  %d  ",c[i]);
}

if(_node_number==2)
{
    int k1,i1;
    int z1[20];

    k1=ChanInInt(LINK0IN);
    for(i1=0;i1<k1;i1++)
    {
        z1[i1]=ChanInInt(LINK0IN);
    }

    quick(z1,k1);

    for(i1=0;i1<k1;i1++)
    {
        ChanOutInt(LINK0OUT,z1[i1]);
    }
}

if(_node_number==3)
{
    int k1,i1;
    int z1[20];

    k1=ChanInInt(LINK0IN);
    for(i1=0;i1<k1;i1++)
    {
        z1[i1]=ChanInInt(LINK0IN);
    }
}

```

```
    quick(z1,k1);  
    for(i1=0;i1<k1;i1++)  
    {  
        ChanOutInt(LINK0OUT,z1[i1]);  
    }  
}  
}
```