STUDY OF PARALLEL COMPUTING IN A TRANSPUTER BASED ENVIRONMENT

USING INMOS C TOOLSET

by

Tariq Asrar Alvi

Submitted in Partial Fulfillment of the Requirements
for the Degree of
Master of Science in Engineering
in the
Electrical Engineering
Program

Salvatore Pansino 11/20/92
Advisor Date

Sally M. Hotchkiss November 24, 1992
Dean of the Graduate School Date

YOUNGSTOWN STATE UNIVERSITY

December, 1992

Y-10-4
(4)

ABSTRACT

STUDY OF PARALLEL COMPUTING IN A TRANSPUTER BASED ENVIRONMENT USING INMOS C TOOLSET

Tariq Asrar Alvi
Master of Science in Engineering
Youngstown State University, 1992

This thesis builds upon the thesis work done by Mr. S. V. Chala, who simulated a control process on a transputer network using the INMOS Occam Toolset. Here, the hardware and software aspects of parallel processing are presented, as well as their application in a control problem, using the pole placement technique. The hardware used is a network of T800 transputers and the supporting hardware, manufactured by the INMOS corporation. The software is the INMOS C TOOLSET, which incorporates a full ANSI C compiler that also supports parallel processing.

Comparison is made between the serial and parallel versions of the same program. The parallel version of the simulation program is found to be faster than the serial version.

ACKNOWLEDGEMENTS

Most of all, the author would like to thank Dr. Dilip K. Singh for initiating the study of transputer-based parallel processing at Youngstown State University. The author's sincere thanks are also due to Dr. S. R. Pansino for his guidance as the advisor.

The author sincerely wishes to thank Mr. S. V. Chala for lending a hand in setting up the hardware, and Ms. V. Kotwal for helping with the analysis of the control problem. The author would also like to thank all the friends from all over the world using Net-News, who offered great advice when he was learning the TOOLSET.

TABLE OF CONTENTS

	PAGE
ABSTRACT	ii
ACKNOWLEDGEMENTS	iii
TABLE OF CONTENTS	iv
LIST OF FIGURES	vi
LIST OF TABLES	vii
CHAPTER	
I. OVERVIEW	1
Transputers	1
Transputers and C	5
II. HARDWARE DESCRIPTION	10
IMS T800	10
TRAM	19
TRAM Motherboard	21
III. ANSI C TOOLSET	25
Features of the Toolset	26
Toolset Summary....................................	28
Program Development Using the Toolset ...	32
IV. PARALLEL PROCESSING	39
Elementary Parallel Operations	40
Matrix Multiplication	42

Parallel Evaluation of Arithmetic Expressions 43

Recursive Doubling 46

V. CONTROL SYSTEM DESIGN VIA POLE PLACEMENT 50

Integral Control 50

VI. APPLICATION DESIGN 59

Problem 59

Derivation 61

Hardware Setup 62

Software 63

Results 66

Conclusion 68

Recommendations 70

APPENDIX 71

Batch File 71

Configuration File 71

Control Software 74

REFERENCES 86

LIST OF FIGURES

FIGURE		PAGE
1.1	Transputer Architecture	3
1.2	Transputer Networks	4
1.3	Parallel Programming Model	7
2.1	IMS T800 Block Diagram	12
2.2	Schematic Diagram of a Size 1 TRAM	20
2.3	Schematic Diagram of IMS B403	22
2.4	Block Diagram of IMS B008	23
3.1	Steps in Development of a Program	34
4.1	Serial Evaluation	45
4.2	Parallel Evaluation	45
4.3	Recursive Doubling on Parallel Computer	48
4.4	Recursive Doubling for 8 Numbers	48
5.1	Integral Controller	51
5.2	Type 1 Servo System	53
6.1	Inverted Pendulum System	60
6.2	Hardware Setup	64
6.3	Response Curve for $x_3(t)$	67

LIST OF TABLES

TABLE		PAGE
1	Floating Point Operation Times for IMS T800 ...	14
2	Speed Variation in IMS T800	16
3	Data Rates for Each Link Speed	18
4	Summary of Tools Used in the Project	29
5	Toolset File Extensions	37
6	Comparison of Serial and Parallel Evaluation ..	46

CHAPTER I

OVERVIEW

This chapter introduces transputers and parallel processing. All details relevant to this thesis will be provided in the succeeding chapters.

Parallel processing is a powerful way of increasing system performance. The combination of hardware parallel support and a compiler package, which makes the hardware features easily accessible from software, makes the transputer and the Toolset powerful vehicle for the development of parallel applications.

Transputers

Transputers are high-performance microprocessors that support parallel processing through on-chip hardware. They can be connected by their serial links in application specific ways and can be used as the building blocks for complex parallel processing systems.

The transputer is a complete microcomputer on a single chip. It contains the hardware support for processor communications, a very fast (single cycle) on-chip memory, and a programmable memory interface that allows external memory to

be added with minimal supporting logic. Figure 1.1 shows the generalized architecture of the IMS T4 family of transputers.

Multitransputer Systems

Multitransputer systems can be built very simply. The four high-speed links allow transputers to be connected to each other in arrays, trees, and many other configurations. The circuitry to drive the links is all on the transputer chip and only two wires are needed to connect a pair of transputers. Some possible arrangements of transputers are illustrated in Figure 1.2.

Links

Transputer links provide a communication and synchronization path between processors, allow memory to be examined directly by debugging programs, and permit programs to be loaded onto whole networks of transputers via a single transputer link. Each individual transputer also supports communication between parallel processes through a system of internal links, implemented as words in memory.

Hardware Parallel Support

Each transputer has a highly efficient built-in runtime scheduler for processes running in parallel on the same transputer and supports channel communication through single words in memory. Processes waiting for input or output, or waiting on a timer, consume no CPU resources, and process

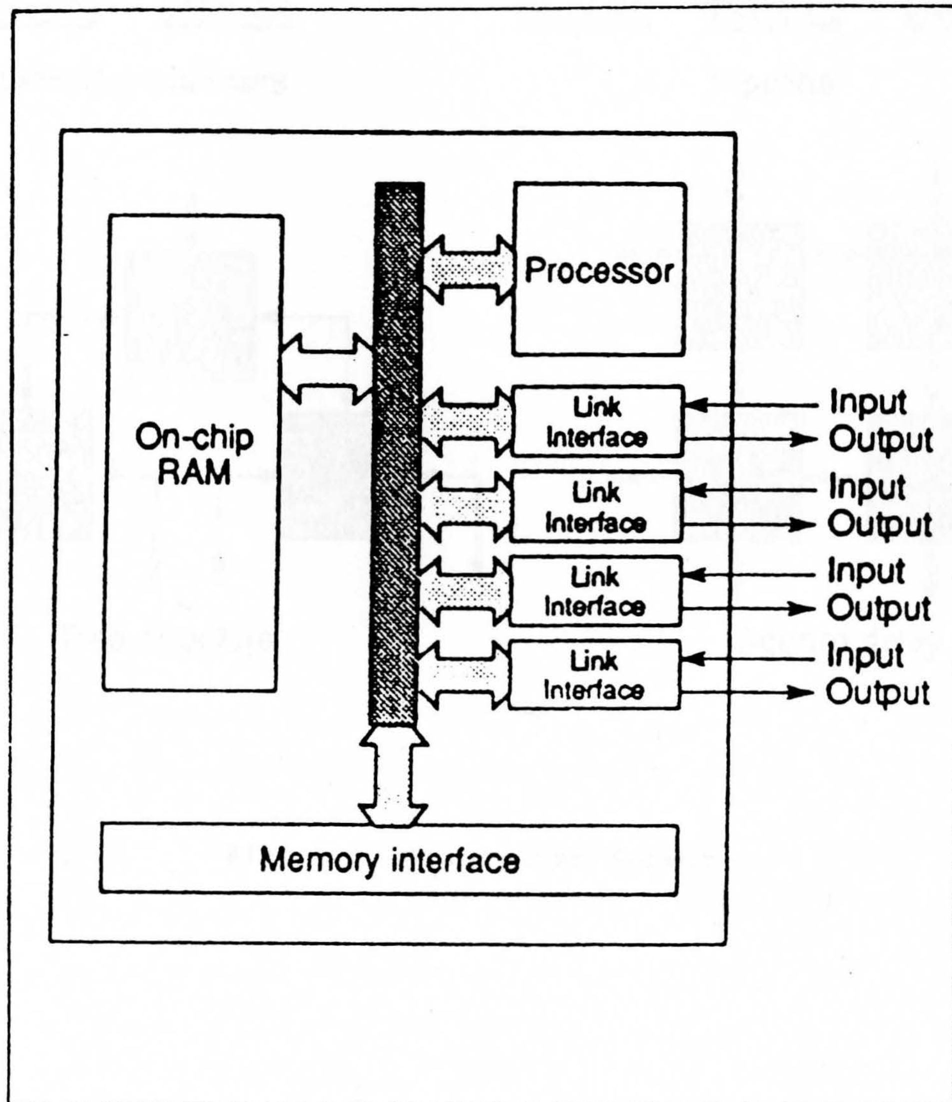

Fig. 1.1. Transputer Architecture

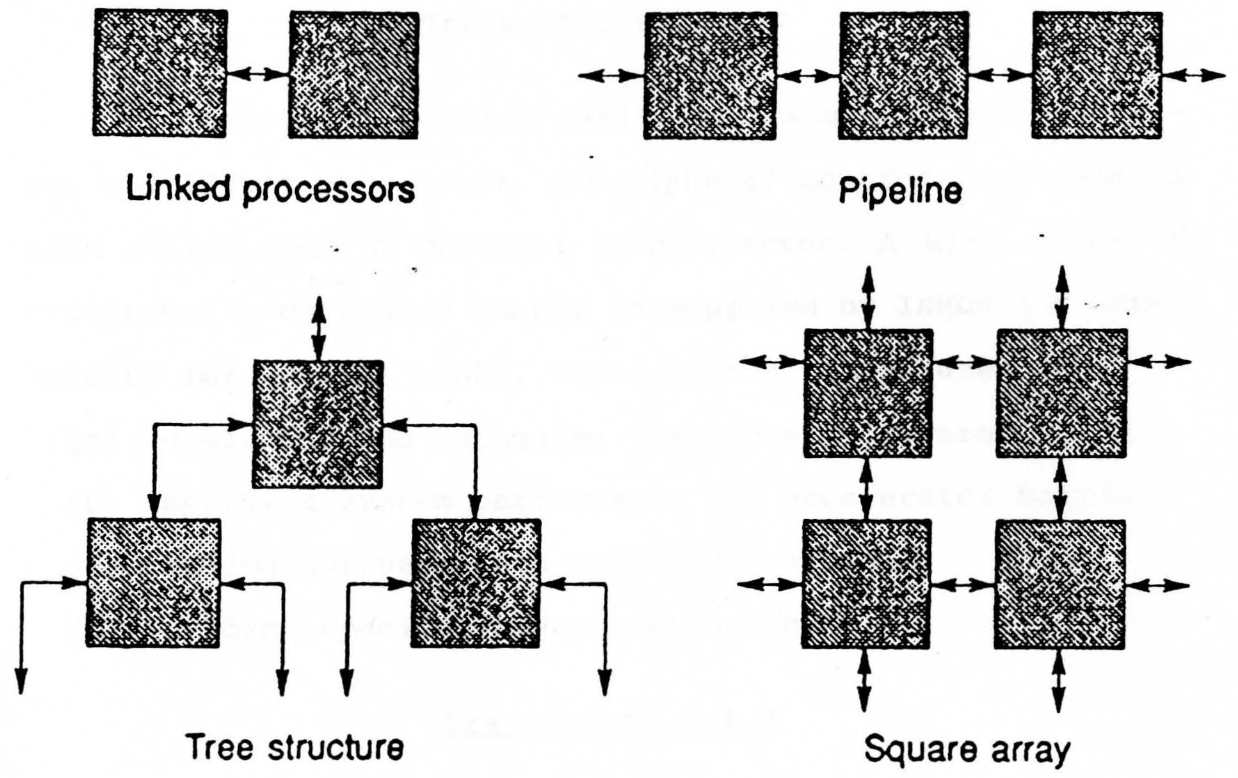

Fig. 1.2. Transputer Networks

context switching time can be as little as one microsecond. The communication links between processors operate in parallel with the processing unit and can transfer data simultaneously on all links without the intervention of the CPU.

Transputer Products

There is a complete family of transputer devices: 32-bit and 16-bit processors, a peripheral control processor, a link switch, and a parallel link adaptor. A wide range of transputer programming boards is supplied by INMOS and other vendors for several hosts. These boards can be used for:

(a) Developing and debugging transputer software.
(b) Improving system performance (as accelerator boards).
(c) Loading software onto embedded systems.
(d) Building specific transputer networks.[1]

Transputers and C

The ANSI C Toolset has been designed to reflect the parallel processing model of communicating sequential processes (CSP). The inherent flexibility of the C language, the capacity to mix code from different languages, and the ability to use the parallel features of the transputer, make the Toolset a powerful tool for programming parallel systems.[2]

Programming Model

The parallel programming model consists of a number of independent processes executing simultaneously and communica-

ting through channels. A process can be built from any number of other parallel processes, so that an entire software system can be described as a hierarchy of intercommunicating parallel processes. This model is consistent with many modern software design methods.

Communication between processes is synchronized. When data is passed between two processes the output process does not proceed until the input process is ready. Buffered communication and multiplexing can be achieved by inserting a specific buffer or multiplexing process between the two processes. Library functions are provided for the input and output of data on channels. Figure 1.3 illustrates the main elements of the programming model. The figure shows that modules can be made up of any number of sub-modules. The arrows represent the direction of communication between modules.

Real Time Programming

The parallel features of the transputer provide direct support for real time programming. The key features are listed below:

(a) Direct and efficient implementation of parallel processes in hardware.

(b) Prioritization of parallel processes.

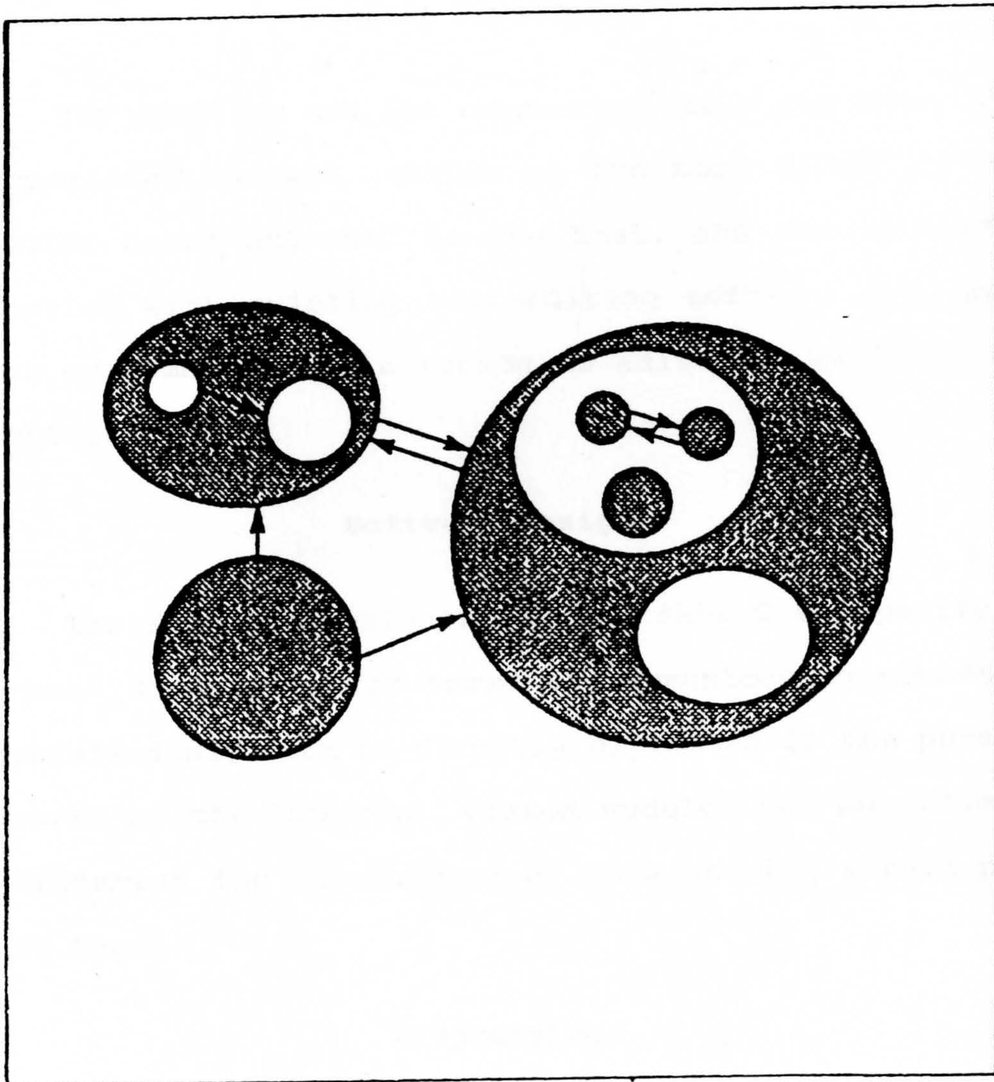

Fig. 1.3. Parallel Programming Model

WILLIAM F. MAAG LIBRARY
YOUNGSTOWN STATE UNIVERSITY

(c) The ability to implement software interrupts as high priority processes.

(d) Easy programming of software timers, allowing close control of timing and non-busy polling.

(e) Placement of variables at specific addresses in memory-mapped devices.

Program Development

The compiler and its supporting tools run under standard operating systems, either on the host itself or on a transputer board attached to the host, and can be used in conjunction with existing text-editing software and source control systems. For this reason no editor is provided with the Toolset.

Software Design

The software designer can use ANSI C to specify the components of a system in terms of communicating processes. The overall design can be directly expressed in the parallel constructs of the language. Common modules can be collected into libraries for the purpose of code sharing within programming teams.

Programming

Code for single transputers is linked using the linker tool and loadable programs are generated using the collector tool. For multitransputer systems the collector tool reads and

processes a configuration data file created by the configurer tool, while for single transputer programs the collector adds bootstrap code for a single processor. Single processor bootstrapping by the collector is controlled by a command line option. Software processes and channels are allocated using the configuration language and loadable code ready for distribution on the network is generated using the configurer.

Debugging

Programs for multitransputer systems can be debugged at the symbolic level using the network debugger that allows a halted program to be analyzed in terms of its source code. A low level debugging environment using direct memory display, instruction disassembly, and processor data is also provided. Breakpoint debugging allows programs to be executed interactively. Post-mortem debugging allows stopped programs to be debugged from the contents of the transputers' memory. The debugger inserts no additional code into the program, but rather writes the data in a description file. This guarantees that the code generated when debugging is disabled will run the way it was originally designed to run.[1]

CHAPTER II

HARDWARE DESCRIPTION

This chapter will discuss the hardware used in the thesis. The following hardware was used:

(a) IMS T800 Transputer

(b) IMS B403 TRAM (Transputer Module)

(c) IMS B008 TRAM Motherboard

IMS T800

The IMS T800 transputer is a 32-bit CMOS microcomputer with a 64-bit floating point unit and graphics support. It has 4 Kbytes of on-chip RAM for high-speed processing, a configurable memory interface, and four standard INMOS communication links. The instruction set achieves efficient implementation of high-level languages and provides direct support to the parallel programming model when using either a single transputer or a network. The transputer is designed to implement the Occam language but also supports other languages such as C, Fortran, and Pascal.

The processor speed of a device can be pin-selected in stages from 17.5 MHz up to the maximum allowed for the part. A device running at 30 MHz achieves an instruction throughput

of 15 MIPS. The high performance arithmetic and floating point unit enables the T800 to achieve 2.25 Mflops at 30 MHz. For convenience of description, the IMS T800 operation is split into the basic blocks (Figure 2.1) described below.

Processor

The 32-bit processor contains instruction processing logic, instruction and work pointers, and an operand register. It directly accesses the high-speed, 4 Kbyte on-chip memory, which can store data or programs. Where larger amounts of memory or programs in ROM are required, the processor has access to 4 Gbytes of memory via the external memory interface (EMI).

Registers

The design of the Transputer processor exploits the availability of fast on-chip memory by having only six registers that are used in the execution of a sequential process. The small number of registers, together with the simplicity of the instruction set, enables the processor to have relatively simple (and fast) data-paths and control logic. The six registers are:

(a) The workspace pointer, which points to an area of storage where local variables are kept.

(b) The instruction pointer, which points to the next instruction to be executed.

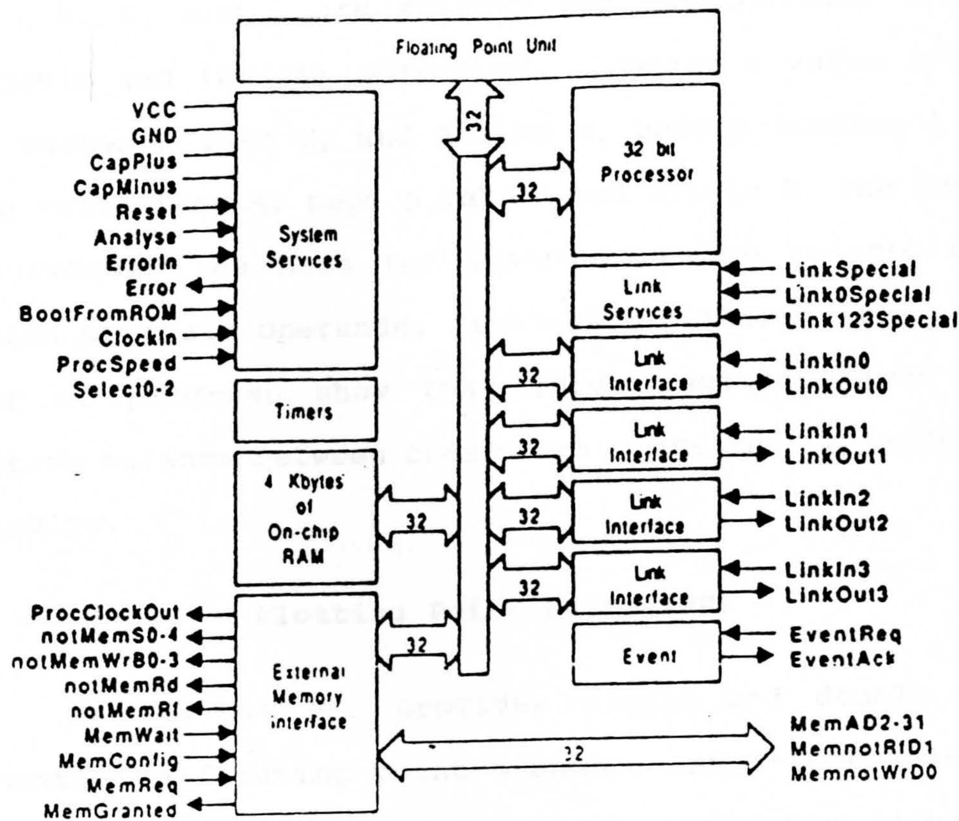

Fig. 2.1. IMS T800 Block Diagram

(c) The operand register, used in the formation of instruction operands.

(d) The A, B, and C registers, which form an evaluation stack.

A, B, and C are sources and destinations for most arithmetic and logical operations. Loading a value into 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. The use of a stack removes the need for instructions to re-specify the location of their operands. Statistics gathered from a large number of programs show that three registers provide an effective balance between code compactness and implementation complexity.

Floating Point Unit (FPU)

The 64-bit FPU provides single and double length arithmetic to floating point standard (ANSI-IEEE 754-1985). It is able to perform floating point arithmetic in parallel with the CPU, sustaining in excess of 2.25 Mflops on a 30 MHz device. All data communication between memory and the FPU occurs under control of the CPU.

The FPU consists of a microcoded computing engine with a three deep floating point evaluation stack for manipulation of floating point numbers. These three stack registers are FA, FB, and FC, each of which can hold either a 32-bit or 64-bit data; and an associated flag, set when a floating point value is loaded.

The FPU has been designed to operate on both single length (32-bit) and double length (64-bit) floating point numbers, and returns results which fully conform to the ANSI-IEEE 754-1985 floating point arithmetic standard. Denormalized numbers are fully supported in the hardware. All rounding modes defined by the standard are implemented. The basic addition, subtraction, multiplication and division are performed by single instructions. The floating point operation times for the IMS T800 are illustrated in Table 1.

TABLE 1

FLOATING POINT OPERATION TIMES FOR IMS T800

	T800-20		T800-30	
Operation	**Single length**	**Double length**	**Single length**	**Double length**
add	350 ns	350 ns	233 ns	233 ns
subtract	350 ns	350 ns	233 ns	233 ns
multiply	550 ns	1000 ns	367 ns	667 ns
divide	850 ns	1600 ns	567 ns	1067 ns

Timers

The transputer has two 32-bit timer clocks. The timers provide accurate process timing, allowing processes to de-schedule themselves until a specific time.

One timer is accessible only to high priority processes and is incremented every microsecond, cycling com-

pletely in approximately 4295 milliseconds. The other is accessible only to low priority processes and is incremented every 64 microseconds, giving exactly 15625 cycles in one second. It has a full period of approximately 76 hours.

System Services

System services include all the necessary logic to initialize and sustain operation of the device. They also include error handling and analysis facilities.

The pins shown in Figure 2.1 are described below. Power is supplied to the device via VCC and GND pins. CapPlus and CapMinus are connected externally by a low leakage, low inductance, 1 μF capacitor for the internally derived power supply for the internal clocks. Reset is assertive high and the falling edge initializes the transputer, triggers the memory configuration sequence and starts the bootstrap routine. Analyze will halt the transputer at the next descheduling point if it is taken high while the transputer is running. ErrorIn and Error, together, indicate that an error was detected. An internal error can be caused, for example, by arithmetic overflow, divide by zero, array bounds violation, or software setting the flag directly. The error pin carries the OR'ed output of the internal error flag and the error input. BootFromROM allows the transputer to be externally bootstrapped when connected to high (e.g., to VCC). ClockIn is the standard clock input supplied by the user. High frequency internal clocks are derived from ClockIn and it must be derived

from a crystal oscillator, since stability is important. ProcSpeedSelect0-2 pins are used to vary the processor speed in discrete steps as shown in Table 2.

TABLE 2

SPEED VARIATION IN IMS T800

Proc Speed Select2	Proc Speed Select1	Proc Speed Select0	Processor Clock Speed MHz	Processor Cycle Time ns	Notes
0	0	0	20.0	50.0	
0	0	1	22.5	44.4	
0	1	0	25.0	40.0	
0	1	1	30.0	33.3	
1	0	0	35.0	28.6	
1	0	1			Invalid
1	1	0	17.5	57.1	
1	1	1			Invalid

Memory

The IMS T800 has 4 Kbytes of on-chip RAM (static memory) for high rates of data throughput. Each internal memory access takes one processor cycle. The transputer can also access 4 Gbytes of external memory space. Internal and external memory are part of the same linear address space.

Internal memory starts at the most negative address 80000000 and extends to 80000FFF. User memory begins at 80000070; this location is given the name MemStart. External

memory space starts at 80001000 and extends up through 00000000 to 7FFFFFFF.

External Memory Interface

The external memory interface (EMI) allows access to a 32-bit address space, supporting dynamic and static RAM as well as ROM and EPROM.

Link Interface(s)

Four identical INMOS bi-directional serial links provide synchronized communication among the processors and with the outside world. Each link comprises an input channel and an output channel. A link between two transputers is implemented by connecting a link interface on one transputer to a link interface on the other. Every byte of data sent on a link is acknowledged on the input of the same link; thus each signal line carries both data and control information.

The quiescent state of a link output is low. Each data byte is transmitted as a high start-bit, followed by another high bit, followed by eight data bits, followed by a low stop-bit. The least significant bit of data is transmitted first. After transmitting a data byte, the sender waits for the acknowledge, which consists of a high start-bit, followed by a zero-bit. The acknowledge signifies both that a process was able to receive the acknowledged data-byte and that the receiving link is able to receive another byte. The sending

link reschedules the sending process only after the acknowledge for the final byte of the message has been received.

Link speeds can be set by LinkSpecial, Link0Special, and Link123Special. Link 0 speed can be set independently. Table 3 shows uni-directional and bi-directional data rates in Kbytes/second for each link speed. LinknSpecial is to be read as Link0Special when selecting link 0 speed and as Link123-Special for the others. Data rates are quoted for a transputer using internal memory, and will be affected by a factor depending on the number of external memory accesses and the length of the external memory cycle.

TABLE 3

DATA RATES FOR EACH LINK SPEED

Link Special	**Linkn Special**	**Mbits/sec**	**Kbytes/sec** **Uni**	**Bi**
0	0	10	910	1250
0	1	5	450	670
1	0	10	910	1250
1	1	20	1740	2350

Event

EventReq and EventAck provide an asynchronous handshake interface between an external event and an internal process. When an external event takes EventReq high, the external event channel (additional to the external link channels) is made ready to communicate with a process. When both the event

channel and the process are ready, the processor takes EventAck high and the process, if waiting, is scheduled. EventAck is removed after EventReq goes low.

TRAM

TRAMs are small, cost-effective sub-assemblies of transputers and other circuitry (often RAM) with a simple but efficient 16-signal-interface standard profiled in modular sizes. The interface accommodates 4 serial transputer links for interprocessor communication, power supply, and system signals.

This standard allows the TRAMs to be mounted onto a variety of motherboards which provide specific host interface hardware. Each motherboard can connect to a number of TRAMs and provides facilities for configuring a network of TRAMs for the user specified topology, under software control. A software package is provided for motherboards which allows this task to be undertaken with the minimum effort.

All TRAMs are based upon a single module profile with a defined pin layout. This single format is known as size 1. The schematic figure of the size 1 TRAM is shown in Figure 2.2.

Larger TRAMs are simply a multiple of the size 1 footprint. Thus, a size 2 TRAM occupies two of the sockets into which a size 1 TRAM will plug. In order to avoid confusion, discussions about motherboards always refer to "slots". A slot is one position into which a size 1 TRAM may be

plugged. Thus, a motherboard which has ten slots may have ten size 1 TRAMs, or five size 2 TRAMs, or two size 4 and two size 1 TRAMs, or one size 8 TRAM, or even six size 1 TRAMs and one size 4 TRAM. The common pins that are available from the TRAMs are described below.

```
o Link2out                    Link3in o
o Link2in                    Link3out o
o VCC                             GND o
o Link1out                    Link0in o
o Link1in                    Link0out o
o LinkSpeedA                 notError o
o LinkSpeedB                    Reset o
o ClockIn (5 MHz)             Analyze o
```

Fig. 2.2. Schematic Diagram of a "size 1" TRAM

Standard TRAM Pins

Transputers, and therefore TRAMs, require three signals to be connected to them to allow them to initialize, and debug so that they can signal an error. These signals are Reset for resetting, Analyze to allow debugging, and NotError to signal an error on a transputer or a TRAM. These three signals are collectively known as system services. The system services for a TRAM are treated as a single signal conceptually, although they are actually three signals.

IMS B403

The IMS B403 is a very compact TRAM providing 2 Mbytes of memory, but still providing maximum performance capability. This is achieved by extending the principle of fast on-chip

RAM to include 32 Kbytes of static RAM, which cycles as fast as possible. So, any technique which puts the most frequently accessed memory locations near the bottom of memory will speed up the processing. This TRAM is the most popular board for running INMOS' Toolset.

The IMS B403 packs 11 square cm of silicon onto a board the size of a credit card. Four IMS B403's fit onto the IMS B008 in a single slot of the IBM PC. The schematic of the IMS B403 appears in Figure 2.3.

TRAM Motherboard

A TRAM motherboard provides a number of slots into which TRAMs can be plugged. Each of these slots provides the necessary connections to power, clock, reset signals and the transputer links. The motherboard provides a method of connecting TRAMs together and may also include special circuitry to provide an interface to something other than a transputer system.

IMS B008

The IMS B008 is a motherboard designed to plug into a PC or PC/AT bus. The board has ten TRAM slots, an interface to the PC bus, and a programmable link switch to allow a network of TRAMs to be set up under software control. Figure 2.4 provides a functional block diagram of the IMS B008.

The interface to the PC provides a single transputer link and a system services port. This allows software running

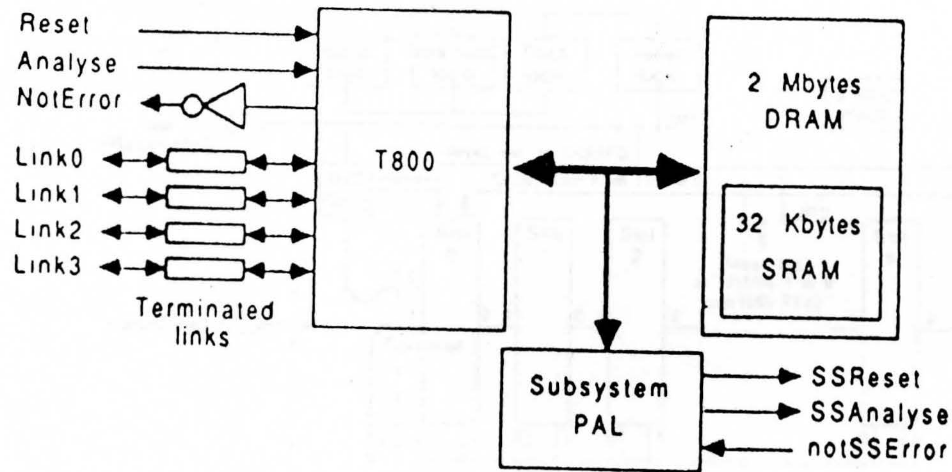

Fig. 2.3. Schematic Diagram of IMS B403

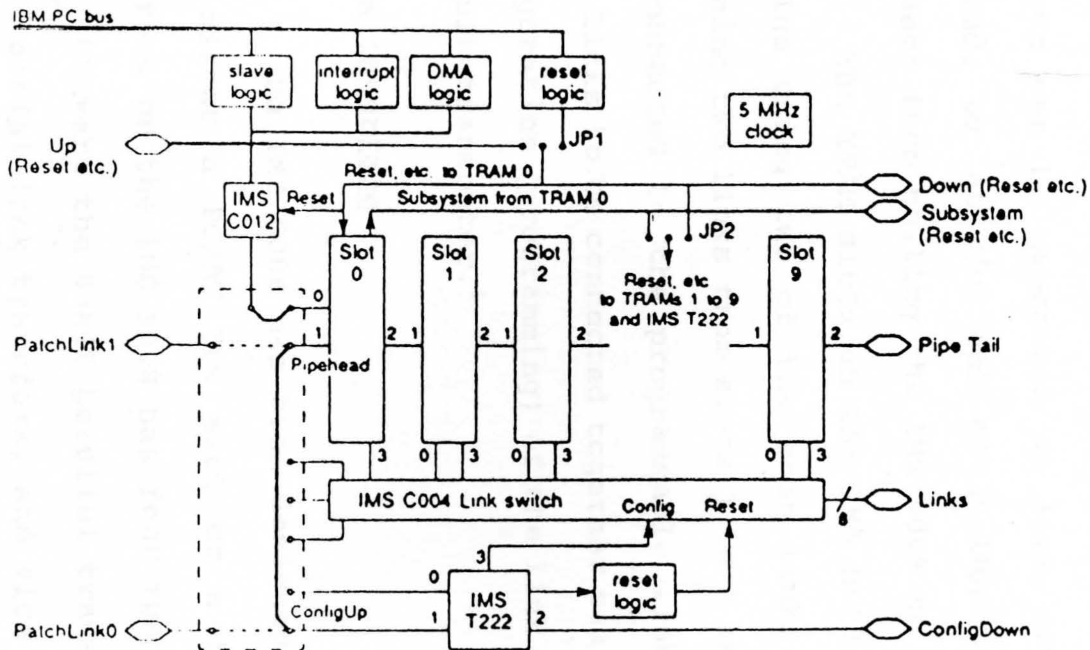

Fig. 2.4. Block Diagram of IMS B008

on the PC to reset, analyze, communicate with, and monitor the error flag of a transputer network connected to or on the IMS B008. Data can be transferred to and from the link interface using programmed I/O or a DMA transfer mechanism, allowing data transfer to go on without processor intervention. Interrupts can be generated on link events, on error being asserted, or at the end of a DMA transfer, freeing the processor from polling the IMS B008 to detect these events.

The TRAM slots on the IMS B008 are connected into a pipeline using two of the four links from each slot. The remaining two links from slots 1 to 9 and link 3 from slot 0, are connected to the programmable link-switch which allows these links to be connected together via software. Control and configuration (programming) of the link switch is performed by a 16-bit transputer.[3]

PC Bus Interface

The IMS B008 has been designed to work when plugged into either a PC/AT bus slot or a PC bus slot. The bus interface on the IMS B008 has four functions to perform:

(a) Convert the 8-bit parallel transfers on the PC bus to serial link transfers, and vice versa.

(b) Provide a system services port.

(c) Control DMA transfers.

(d) Generate interrupts on link interface events, on the assertion of transputer error, or on DMA transfer end.

CHAPTER III

ANSI C TOOLSET

This chapter gives an overview of the ANSI C Toolset. It briefly describes each tool, outlines its purpose, and explains how the tools are used together to develop, configure, load, and run transputer programs. The chapter also introduces the run-time library, outlines the standards for error reporting, and summarizes host-specific characteristics.

The ANSI C Toolset is a software cross-development system for transputers, hosted on several systems, among them the PC/MS-DOS system. It consists of a full ANSI C compiler with parallel support, a multilanguage linker, a configurer for mapping programs onto transputer networks, a code collector tool for generating directly loadable files, and a program loader and host server tool. The Toolset also includes a fully interactive debugger, program building tools, and EPROM programming tools. Together, the compiler and its supporting tools form an integrated environment for the development of programs on transputers and transputer-based hardware.

Features of the Toolset

The ANSI C Toolset is an integrated development system for transputer programs incorporating a new standard object file format, a C-like configuration language, a comprehensive run-time library, and support for parallel programming based on the communicating process model. It represents a broad enhancement of the approach to parallel programming in C and introduces standards for the generation of object code for transputers and transputer-based hardware.

Standard Object File Format

The ANSI C compiler generates object code in an intermediate form known as TCOFF (Transputer Common Object File Format). The adoption of this format introduces a standard for the development of future transputer compilers and enables code generated by compatible compilers to be freely mixed in the same system.

Configuration Language

The configuration language allows software and hardware networks to be described separately and joined by a software-to-hardware description. The language is a simple declarative language that has the syntactic flavor of C and can be used on any size network. A full range of high-level language constructs including replicative and conditional statements make it easy to explore different configurations before committing to hardware.

Run-time Library

A comprehensive run-time library is supplied with the Toolset providing full ANSI C support with additional support for parallel programming. The library of parallel functions provides channel-based communication. An optimized library with no server support is available for embedded code.

Parallel Programming

The abstract model used in ANSI C reflects the Communicating Sequential Process (CSP) model of parallel programming. The model maps easily onto the transputer to provide efficient parallel code. Software is broken down into independent processes which exchange data and synchronize their activity via channels. Processes can be mapped onto one, several, or many transputers using the configuration language.

Transputer Targets

The ANSI C Toolset can be used to write programs targeted at IMS M212, T212, T222, T225, T400, T414, T425, T800, T801, and T805 transputers. Code can also be written to run on a group of processor types by compiling for a transputer class.

Toolset Summary

The tools that were used are summarized in Table 4 and are briefly described here.

Compiling the Code

ANSI C Compiler - icc

The compiler icc is an ANSI standard C compiler with additional support for parallel programming. It conforms fully to ANSI standard X3.159 1989.

The ANSI standard for C formalizes the original implementation of C as described in "The C programming Language" by Kernighan and Ritchie. It further extends it to include a run-time library, some language extensions already in common usage, and many other improvements designed to standardize the language.

ANSI C supports parallel programming through a series of C structures and a comprehensive set of process handling and channel communication functions. Some useful non-ANSI functions are also provided in the run-time library.

The compiler produces code for specific processor types or transputer classes. The compiled object file must be linked, configured, and made executable before the program can be run. The executable file consists of code which can be directly loaded onto a transputer network.

TABLE 4

SUMMARY OF TOOLS USED IN THE PROJECT

Tool	**Description**
icc	The ANSI C compiler. A full ANSI standard compiler with parallel support. Generates object code for specific transputer targets.
icconf	The configurer. Analyzes the configuration description and produces a configuration data file for the code collector.
icollect	The code collector. Collects linked units into a single file for loading on a transputer network. Takes as input a configuration data file or a single linked unit.
ilink	The Toolset linker. Resolves external references and links separately compiled code into a second file.
iserver	The host file server. Loads programs onto transputer hardware and provides runtime access to the host.

Generating Executable Code

Three tools are used in sequence (or two for a single transputer program) to generate the loadable file from compiled object code. They are described below.

Linker - ilink

The Toolset linker, ilink, links separately compiled modules and libraries into a single code unit, resolving external references and generating a linked unit. Linked units can be used in configuration descriptions to map software onto specific arrangements of transputers, or can be bootstrapped for a single transputer using icollect.

Library modules are linked with the program by the C start-up file which must be specified on the linker command line. The correct start-up file must be specified for the transputer type.

Configurer - icconf

The configurer, icconf, generates configuration information for transputer networks from a configuration description written in the transputer configuration language. The tool prepares the program for configuring on a specific arrangement of transputers by analyzing the configuration description and producing a data file for the code collector tool.

Code Collector - icollect

The code collector tool, icollect, takes the data file generated by icconf and generates a single file that can be loaded and run on a transputer network. The file contains bootable code modules for all processors on the network, along with distribution information that is used by the loader to place the modules on each processor.

icollect is also used to generate bootable code for single transputer programs from linked units by appending single transputer bootstrap code. The single transputer mode of operation is selected by a command-line option.

Loading and Running Programs

Bootable code for single transputers and transputer networks is loaded onto the transputer hardware using the host file server tool, iserver, which both loads the program and starts the run-time environment that supports interaction with the host.

Host File Server - iserver

The host file server, iserver, is a combined host server and loader tool. When invoked to load a program it both loads the code onto the transputer hardware and provides run-time services on the host (such as program input/output) for the transputer program.

Program Development Using the Toolset

The ANSI C Toolset is a cross-development system for transputers. Creation of executable code for a transputer or transputer network takes several stages involving the use of specific tools at each stage. Program development is supported by tools which provide facilities for debugging and creating object code libraries.

The main stages in developing a program and the tools used are listed below:

(a) **Write the source**

Source code can be written using any ASCII editor. Code can be divided between any number of source files. Source code must conform to ANSI standard. Source code syntax can be checked prior to compilation by invoking the compiler with the check option.

(b) **Compile the source**

Each source file is compiled using the ANSI C compiler, icc, to produce one or more compiled object files. Each file must be compiled for the same transputer type or for a transputer class covering several compatible types.

(c) **Link the compiled units**

The compiled source files are linked together using ilink. This generates a single file called a linked unit in which all external references are resolved. The linking operation also links the library modules

required by the program, which are selected by transputer type from the compiled library code.

(d) **Configure the program**

For multitransputer programs a configuration description must be constructed in order to assign linked units to specific nodes on the transputer network and link them by channel variables. The description is processed by the configurer tool, icconf, to produce a configuration data file. Single transputer programs can also be configured.

(e) **Generate an executable file**

The configuration data file generated by icconf is read by the code collector, icollect, which generates a single executable file for a transputer network. The same tool is used to directly generate bootable files for single transputer programs from linked units.

(f) **Load and run the program**

The executable or bootable file is loaded or run on the transputer network down a host link using iserver. Once loaded, code begins to execute immediately. The server tool also starts up and maintains the environment that supports the programs communication with the host.

Figure 3.1 illustrates the development in terms of the architecture of the toolset. The default file extensions assumed and generated by the tools are used to represent source and target files.

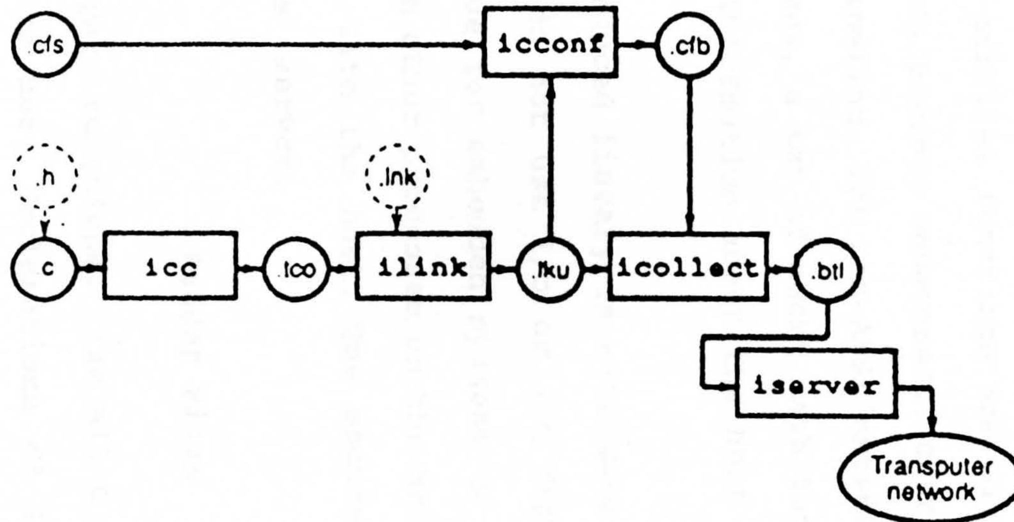

Fig. 3.1. Program Development Using the Toolset

Run-time Library

The run-time library is a library of compiled C functions that perform common programming operations. The library contains the complete set of ANSI standard functions plus functions to support parallel programming and some non-ANSI extensions.

The parallel functions are divided into three functional groups: process management, channel communication, and semaphore handling. The non-ANSI extensions include a set of i/o primitives, a set of short math functions, functions for retrieving information about the host system, and debugging functions.

A reduced library is available for linking with programs that do not use i/o or i/o dependent functions, for example, code for embedded systems or code that only communicates with other processes on the network and has no direct interaction with the host. The reduced library contains no calls to the iserver.

Header Files

Library functions, like all C functions, must be declared before use. Declarations of library functions with associated constants, macros, and definitions are held in a number of library header files to ensure that function declarations are of the correct form and that supporting macros and constants are included. Header files are given the extension .h.

The library header files contain groups of routines collected together according to common usage. For example, routines that control standard i/o operations are grouped in the file stdio.h. Most header files also contain definitions of constants and macros that are associated with the functions' use.

Many of the header files and function groupings are defined in the ANSI STANDARD. The library extensions which support parallel programming and other non-ANSI operations are also grouped for programming convenience, for example, functions for sending data down channels are grouped separately from those which manipulate semaphores. Similarly, non-ANSI functions such as short math functions and low level i/o functions are grouped separately. Parallel programming functions are in fact grouped into three files covering process handling, channel communication, and semaphore handling.

Some library functions are implemented as macros, and a few are implemented as both functions and macros. The decision about which to use depends on the programming style and personal choice.

Toolset File Extensions

The toolset uses a standard set of file extensions to identify specific source, intermediate, and object files. Certain file extensions are assumed on input, and generated on output if extensions are not specified on the command line. For example, the compiler assumes the extension .c for the

input source file and adds the extension .tco to the output file, unless otherwise specified. The adoption of a standard system allows file extensions to be omitted on the command line and permits host file handling systems to manipulate the files. The system forms an integrated whole and is designed to reflect the architecture of toolset compilation. The main file extensions are listed in Table 5.

TABLE 5

TOOLSET FILE EXTENSIONS

Extension	Description
.btl	Bootable code file.
.c	C source files.
.cfb	Configuration data file.
.cfs	Configuration description.
.lku	Linked unit.
.lnk	Linker indirect file.
.tco	Compiled code file.

Error reporting

All errors are reported in a standard format containing the name of the tool, a severity level, and some explanatory text explaining why the error occurred. Errors found in files or the file system may also generate a filename and line number. For example:

```
Warning-icc-prog.c(25) inventing 'extern int foo();'
```

Host dependencies

The ANSI C Toolset can be hosted on several platforms, and is designed to blend in as far as possible with each host operating system. Source and object code is portable between all systems.

The Toolset is available for the following systems:

(a) IBM PC and NEC PC running MS-DOS.

(b) VAX running VMS.

(c) Sun 3 running SunOS.

(d) Sun 4 running SunOS.[1]

CHAPTER IV

PARALLEL PROCESSING

Parallel processing is a technique for increasing the computation speed for a task, by dividing the algorithm into several sub-tasks and allocating multiple processors to execute multiple sub-tasks simultaneously. Compared to serial systems, parallel systems permit more freedom of expression. A foundation in the skills of thinking in parallel is basic to the understanding of such systems.

The following general principles may provide some useful guidelines:

(a) The most obvious approach is to examine a serial method and convert it into a procedure that operates on composite mathematical objects such as vectors and matrices, so that many data are processed simultaneously. However, the latest and most efficient serial method is not always best suited for such adaptation; often an earlier less efficient, serial method already possesses a high degree of parallelism, and so is much more adaptable to parallel computation.

(b) It may turn out that an iterative algorithm equivalent to a certain non-iterative method possesses a higher

degree of parallelism (more independent computations) and can be organized systematically on a parallel machine.

(c) The computations may be broken down into smaller units and distributed among processors.

These ideas are discussed in connection with the following topics:

(a) elementary parallel operations,

(b) matrix multiplication,

(c) parallel evaluation of arithmetic expressions, and

(d) recursive doubling.

<u>Elementary Parallel Operations</u>

The following operations are likely to be available on most parallel computers, and will, therefore, be defined:

(a) Arithmetic operations: +, -, *, ÷ denote term-wise addition, subtraction, multiplication, and division respectively on matrices and vectors.

(b) Row and column selection operations: A_{i-} and A_{-j} denote the ith row and jth column of matrix A.

If,

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \tag{4-1}$$

then,

$$A_{1\blacksquare} = [1 \;\; 2] \tag{4-2}$$

and

$$A_{\blacksquare 1} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \tag{4-3}$$

(c) Row and column operations: ΣrA and ΣcA denote the row and column vector obtained by summing the rows and columns of A, respectively; for example, for A as in (b) above,

$$\Sigma rA = [4 \;\; 6] \tag{4-4}$$

and

$$\Sigma cA = \begin{bmatrix} 3 \\ 7 \end{bmatrix} \tag{4-5}$$

(d) Matrix formation operations: r(V1T,...,VnT) and c(V1,...,Vn) denote those matrices whose rows or columns respectively are vectors V1T,...,VnT and V1,...,Vn, where T denotes transpose.

(e) Maximum and minimum operations: max A and min A denote the maximum and minimum elements of a vector or matrix A.

(f) Logical operations: AND and OR denote the term-wise Boolean operations on objects such as logical matrices and vectors; for example, for logical vectors M and P:

M AND P = (M1,...,Mn) AND (P1,...,Pn)

= (M1 AND P1,...,Mn AND Pn).

Matrix Multiplication

Let A and B be matrices of size m × n and n × p respectively. In forming the matrix product C = AB with elements

$$C_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj} \quad (1 \le i \le m,\ 1 \le j \le p) \tag{4-6}$$

there are mnp products $a_{ik}b_{kj}$ to be calculated.

A strategy for forming this product on a parallel computer with N > mnp processors will be considered.

The matrix C = AB has mp entries; each is the sum of products of n pairs of numbers; and the total number of scalar multiplications is mnp. Since N > mnp, all multiplications can be performed with a single application of *, by multiple positioning of data entries, as

$$\uparrow m \downarrow \begin{bmatrix} A & \dots & A \end{bmatrix} * \begin{bmatrix} B1T_ & \dots & B1T_ \\ B1T_ & \dots & B1T_ \end{bmatrix} \uparrow m \downarrow$$

$$\leftarrow np \rightarrow \qquad \leftarrow np \rightarrow$$

where BiT_ denotes the ith row of BT.

The p successive copies of A are placed in adjacent positions and columns of B are placed horizontally and repeated m times. Various additions must be made, and the sums assigned to correct positions in C. In general there are mp results in the result matrix, and each entry consists of n numbers.

Parallel Evaluation of Arithmetic Expressions

Arithmetic expressions are central to any type of computation, and therefore, it is important to consider their evaluation on a parallel machine.

An arithmetic expression is built up from variables $x_1, \ldots, x_n$ by means of the operations of addition, subtraction, multiplication, and division. Two expressions are equivalent if they take the same value for every assignment of values to the variables.

On a parallel computer, the evaluation of an arithmetic expression E is based on the selection of an equivalent expression α for which several operations can be carried out simultaneously.

Consider,

$$E = (x_1x_2+x_3)x_4+x_5 \quad (4-7)$$

The order of evaluation of (4-7) on a serial computer may be indicated by

$$E = ((((x_1 . x_2) + x_3) . x_4) + x_5) \tag{4-8}$$

The equivalent

$$\alpha = x_1 . x_2 . x_4 + x_3 . x_4 + x_5 \tag{4-9}$$

suitable for parallel computation, is evaluated as

$$\alpha = (((x_1 . x_2) . x_4) + ((x_3 . x_4) + x_5)) \tag{4-10}$$

where the rule is that all the inner brackets are computed at step 1, all the next brackets at step 2, and so on. The serial and parallel evaluations using E and α respectively are shown in figures 4.1 and 4.2. In serial computation, the operations are carried out sequentially, leading to expression E as a result, as in Figure 4.1. In the parallel case (figure 4.2), at step t=1, the products x_1x_2 and x_3x_4 are computed simultaneously; at step t=2, the product $x_1x_2.x_4$ and the sum $x_3x_4+x_5$ are computed simultaneously; finally, at step t=3, the sum is computed to produce α.

We can compare the serial and parallel evaluations as follows:

Let

t = number of parallel or serial steps,

p = number of processors used,

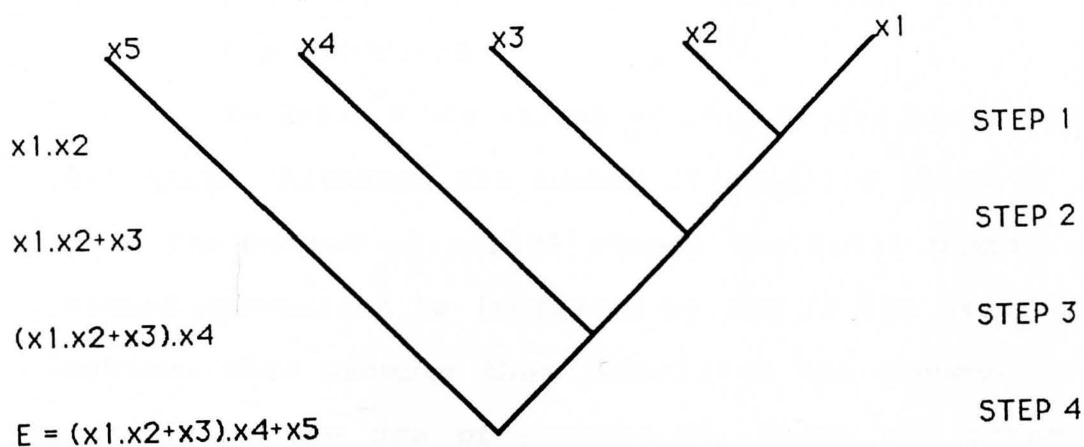

Fig. 4.1. Serial Evaluation

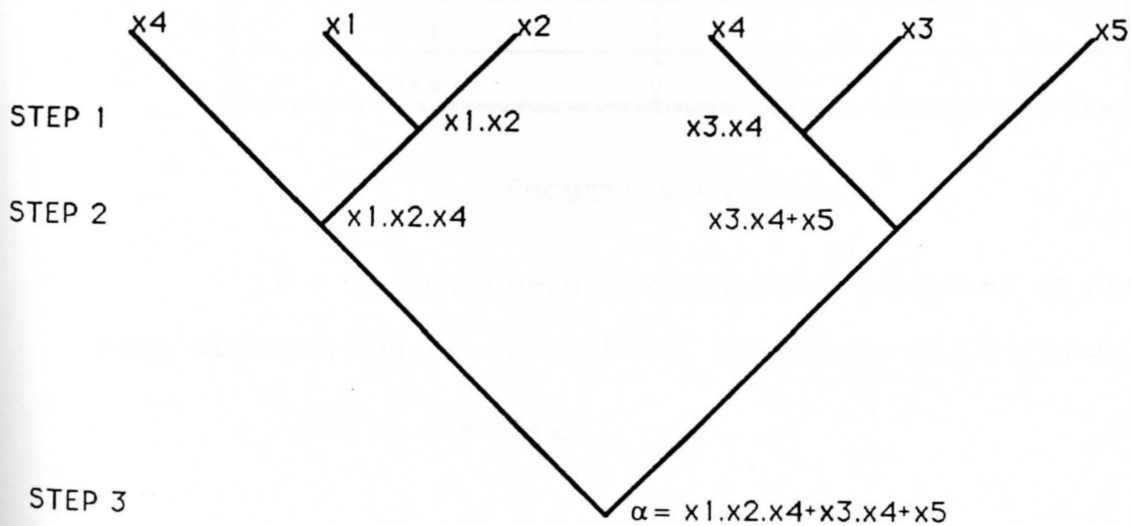

Fig. 4.2. Parallel Evaluation

s = total number of operations performed by the algorithm.

In Table 6 the values of (t,p,s) are displayed for the two cases. Although the number of parallel steps is one less than the number of serial steps, the total number of individual operations is increased by one in the parallel computation. This example thus underlines the connection between optimizing the use of processors, which may otherwise lie idle, and minimizing the total number of steps.

TABLE 6

COMPARISON OF SERIAL AND PARALLEL EVALUATION

Serial evaluation, E	**Parallel evaluation, α**
t:4	3
p:1	2
s:4	5

Recursive Doubling

If $\circ$ is an associative operation on pairs of mathematical objects (numbers, vectors, matrices, etc.), thus

$$(a \circ b) \circ c = a \circ (b \circ c) \quad (4\text{-}11)$$

then the "product"

$$a_1 \circ a_2 \circ \ldots \circ a_n \quad (4\text{-}12)$$

is uniquely defined, independent of the bracketing; for example,

$$((a_1 \circ a_2) \circ a_3) \circ a_4 = (a_1 \circ a_2) \circ (a_3 \circ a_4) \tag{4-13}$$

The left-hand side represents the natural way of calculating the product; the right-hand side is an alternative method, such that on a parallel machine the operations in the brackets can be carried out simultaneously, as in Figure 4.3.

Computations within each level are performed in parallel, and (in general) if the size of the set of objects is n, then the result is produced in $\log_2 n$ steps. This is the basic idea behind recursive doubling, whereby the total computation is repeatedly divided into two seperate computations of equal complexity that can be executed in parallel. The natural means of carrying out these operations is to use a binary interconnection of processors. If we take $2^3 = 8$ numbers then the calculations may be arranged as shown in Figure 4.4, where each of the processors, not necessarily distinct, P1,...,P7 performs an associative operation $\circ$ on distinct pairs of objects chosen from $a_1, \ldots, a_8$.

At step 1, P_4: $a_1 \circ a_2$, store result as x_1,

P_5: $a_3 \circ a_4$, store result as x_2,

P_6: $a_5 \circ a_6$, store result as x_3,

P_7: $a_7 \circ a_8$, store result as x_4.

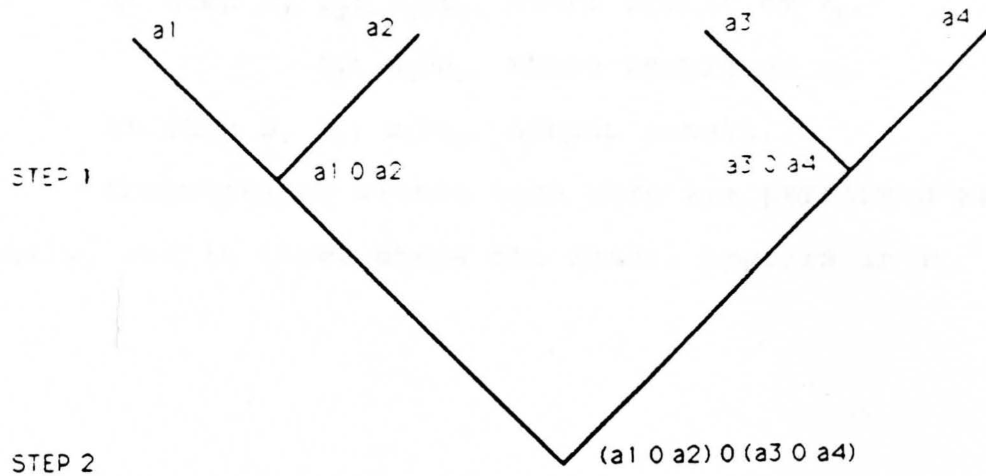

Fig. 4.3. Recursive Doubling on Parallel Computer

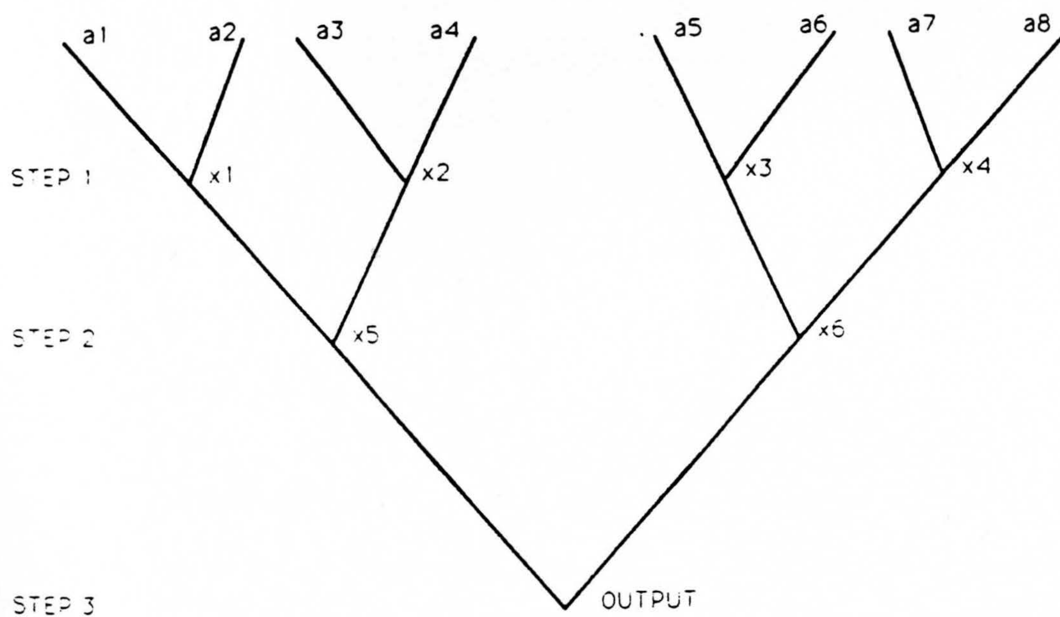

Fig. 4.4 Recursive Doubling for 8 Numbers

At step 2, P_2: $a_1 \circ a_2$, store result as x_5,

P_3: $a_3 \circ a_4$, store result as x_6.

At step 3, P_1: $a_5 \circ a_6$, output result.

Computations within each step are performed simultaneously, and in three steps the result appears in P_1.[3]

CHAPTER V

CONTROL SYSTEM DESIGN VIA POLE PLACEMENT

This chapter will present a design method commonly called the pole placement or pole assignment technique. This method will be applied to control a fourth order system, to be presented in Chapter 6. It will be assumed that all states variables are measurable and are available for feedback. It will be shown that if the system is completely state controllable, then poles of the closed-loop system may be placed at any desired locations by means of state feedback through an appropriate state feedback gain matrix.

Since the system to be considered in Chapter 6 is not type 1, an integral controller will be used.

Integral Control

In an integral controller the value of output u(t) is changed at a rate proportional to the actuating error signal e(t). That is,

$$\frac{du(t)}{dt} = K_i e(t) \tag{5-1}$$

or

$$u(t) = K_i \int_0^t e(t)\,dt \tag{5-2}$$

Here K_i is an adjustable constant.

The transfer function of the integral controller is

$$\frac{U(s)}{E(s)} = \frac{K_i}{s} \tag{5-3}$$

If the value of e(t) is doubled, then the value of u(t) varies twice as fast. For zero actuating error, the value of u(t) remains stationary. Figure 5.1 shows a block diagram of an integral controller.

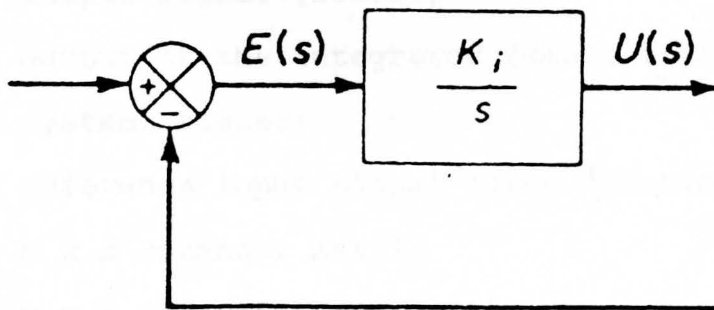

Fig. 5.1. Integral Controller

Design of Type 1 Servo System Where Plant has no Integrator

Since the plant has no integrator (type 0 plant), the basic principle of the design of a type 1 servo system is to

insert an integrator in the feed forward path between the error comparator and the plant as shown in figure 5.2. From the diagram,

$$\dot{\mathbf{x}} = \mathbf{Ax}+\mathbf{B}u \tag{5-4}$$

$$y = \mathbf{Cx} \tag{5-5}$$

$$u = -\mathbf{Kx}+k_1\xi \tag{5-6}$$

$$\dot{\xi} = r-y = r-\mathbf{Cx} \tag{5-7}$$

Here **x** = state vector of the plant (n-vector)

u = control signal (scalar)

y = output signal (scalar)

ξ = output of the integrator (state variable of the system, scalar)

r = reference input signal (step function, scalar)

A = n x n constant matrix

B = n x 1 constant matrix

C = 1 x n constant matrix

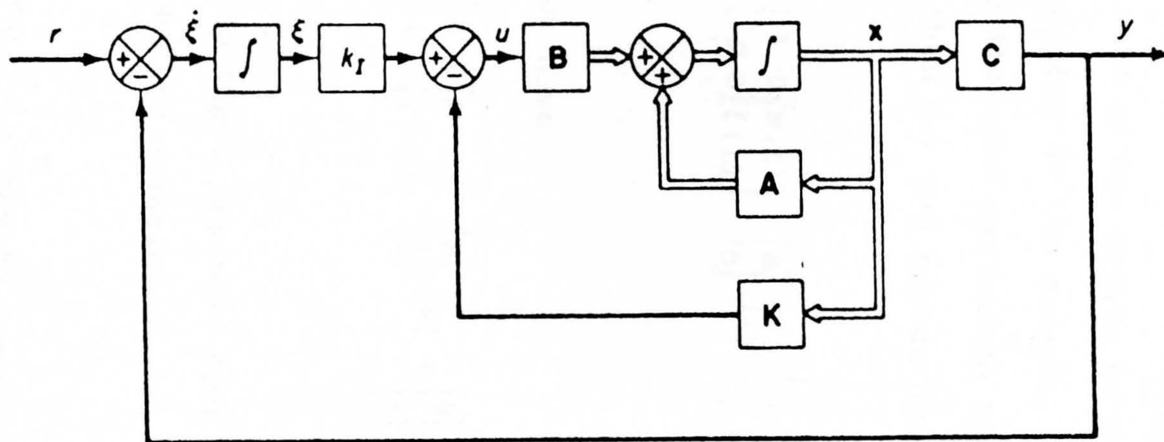

Fig. 5.2. Type 1 Servo System

Assuming that the plant given by equation (5-4) is completely state controllable, the transfer function can be given by

$$G_p(s) = \mathbf{C}(s\mathbf{I}-\mathbf{A})^{-1}\mathbf{B} \tag{5-8}$$

To avoid possibility of the inserted integrator being canceled by the zero at the origin of the plant, it is assumed that $G_p(s)$ has no zero at the origin.

Assuming that the reference input (step function) is applied at t=0, then for t>0 system dynamics can be described by

$$\begin{bmatrix}\dot{\mathbf{x}}(t)\\ \xi(t)\end{bmatrix} = \begin{bmatrix}\mathbf{A} & \mathbf{0}\\ -\mathbf{C} & 0\end{bmatrix}\begin{bmatrix}\mathbf{x}(t)\\ \xi(t)\end{bmatrix} + \begin{bmatrix}\mathbf{B}\\ 0\end{bmatrix}u(t) + \begin{bmatrix}\mathbf{0}\\ 1\end{bmatrix}r(t) \tag{5-9}$$

At steady state we have

$$\begin{bmatrix}\dot{\mathbf{x}}(\infty)\\ \xi(\infty)\end{bmatrix} = \begin{bmatrix}\mathbf{A} & \mathbf{0}\\ -\mathbf{C} & 0\end{bmatrix}\begin{bmatrix}\mathbf{x}(\infty)\\ \xi(\infty)\end{bmatrix} + \begin{bmatrix}\mathbf{B}\\ 0\end{bmatrix}u(\infty) + \begin{bmatrix}\mathbf{0}\\ 1\end{bmatrix}r(\infty) \tag{5-10}$$

Since r(t) is a step input, we have r(∞) = r(t) = r (constant) for t>0. By subtracting equation (5-10) from equation (5-9), we obtain

$$\begin{bmatrix}\dot{\mathbf{x}}(t) - \dot{\mathbf{x}}(\infty)\\ \xi(t) - \xi(\infty)\end{bmatrix} = \begin{bmatrix}\mathbf{A} & \mathbf{0}\\ -\mathbf{C} & 0\end{bmatrix}\begin{bmatrix}\mathbf{x}(t) - \mathbf{x}(\infty)\\ \xi(t) - \xi(\infty)\end{bmatrix} + \begin{bmatrix}\mathbf{B}\\ 0\end{bmatrix}[u(t) - u(\infty)] \tag{5-11}$$

Define

$$\begin{aligned} \boldsymbol{x}(t) - \boldsymbol{x}(\infty) &= x_e(t) \\ \xi(t) - \xi(\infty) &= \xi_e(t) \\ u(t) - u(\infty) &= u_e(t) \end{aligned} \tag{5-12}$$

Then equation (5-10) can be written as

$$\begin{bmatrix} \dot{\boldsymbol{x}}_e(t) \\ \xi_e(t) \end{bmatrix} = \begin{bmatrix} \boldsymbol{A} & 0 \\ -\boldsymbol{C} & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_e(t) \\ \xi_e(t) \end{bmatrix} + \begin{bmatrix} \boldsymbol{B} \\ 0 \end{bmatrix} u_e(t) \tag{5-13}$$

where

$$u_e(t) = -\boldsymbol{K}\boldsymbol{x}_e(t) + k_I \xi_e(t) \tag{5-14}$$

Define a new (n + 1)th-order error vector **e**(t) by

$$\boldsymbol{e}(t) = \begin{bmatrix} \boldsymbol{x}_e(t) \\ \xi_e(t) \end{bmatrix} \tag{5-15}$$

The equation (5-13) then becomes

$$\dot{\boldsymbol{e}} = \hat{\boldsymbol{A}}\boldsymbol{e} + \hat{\boldsymbol{B}}u_e \tag{5-16}$$

where

$$\hat{\boldsymbol{A}} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{0} \\ -\boldsymbol{C} & 0 \end{bmatrix}, \quad \hat{\boldsymbol{B}} = \begin{bmatrix} \boldsymbol{B} \\ 0 \end{bmatrix} \tag{5-17}$$

and equation (5-14)becomes

$$u_e = -\hat{\boldsymbol{K}}\boldsymbol{e} \tag{5-18}$$

where

$$\hat{\boldsymbol{K}} = [\boldsymbol{K} \vdots -k_I] \tag{5-19}$$

Equations (5-16) and (5-18) describe the dynamics of the (n+1)th-order regulator system. If the system defined by equation (5-16) is completely state controllable, then, by specifying the desired characteristic equation for the system, matrix **K** can be determined by the pole placement technique. The steady state values of **x**(t), ξ(t), and u(t) can be found as follows: At steady state (t = ∞), from equations (5-4) and (5-7), we derive the equations (5-20) and (5-21).

$$\dot{\boldsymbol{x}}(\infty) = \boldsymbol{0} = \boldsymbol{A}\boldsymbol{x}(\infty) + \boldsymbol{B}u(\infty) \tag{5-20}$$

$$\dot{\xi}(\infty) = 0 = r - \boldsymbol{C}\boldsymbol{x}(\infty) \tag{5-21}$$

These two can be combined into one vector-matrix equation:

$$\begin{bmatrix} \mathbf{0} \\ 0 \end{bmatrix} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ -\boldsymbol{C} & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{x}(\infty) \\ u(\infty) \end{bmatrix} + \begin{bmatrix} 0 \\ r \end{bmatrix} \tag{5-22}$$

If matrix **P**, defined by

$$\boldsymbol{P} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ -\boldsymbol{C} & 0 \end{bmatrix} \tag{5-23}$$

is of rank n + 1, then its inverse exists and

$$\begin{bmatrix} \boldsymbol{x}(\infty) \\ u(\infty) \end{bmatrix} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ -\boldsymbol{C} & 0 \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{0} \\ -r \end{bmatrix} \tag{5-24}$$

Also, from Equation (5-6) we have

$$u(\infty) = -\boldsymbol{Kx}(\infty) + k_I \xi(\infty) \tag{5-25}$$

Therefore, we have

$$\xi(\infty) = \frac{1}{k_I}[u(\infty) + \boldsymbol{Kx}(\infty)] \tag{5-26}$$

It is noted that, if matrix **P** given by equation (5-23) has rank n+1, then the system defined by equation (5-16) becomes completely state controllable. Therefore, if the rank

of matrix **P** given by Equation (5-23) is n + 1, then the solution to this problem can be obtained by the pole placement approach.

The state error equation can be obtained by substituting equation (5-18) into equation (5-16) .

$$\dot{e} = (\hat{A} - \hat{B}\hat{K})\, e \tag{5-27}$$

If the desired eigenvalues of matrix **A - BK** (that is, the desired close loop poles) are specified as $\mu_1, \mu_2, \ldots, \mu_{n+1}$, then the state feedback gain matrix **K** and the integral gain constant k_I can be determined.[5] In the actual design, it is necessary to consider several different matrices **K** (which correspond to several different sets of desired eigenvalues) and carry out computer simulations to find the one that yields the best overall system performance. Then the best one is chosen as the matrix **K**.

CHAPTER VI

APPLICATION DESIGN

This chapter presents the application of the information put forth in the preceding chapters. A program is presented that will utilize the pole placement technique to control a system. First, a sample problem and its transfer function and its state-space equations are presented. Following that, all the variables needed for the pole placement method are found. Then the hardware and the software implementation are discussed. Finally, the graphical results are presented.

Problem

An inverted pendulum system, as shown in figure 6.1, is considered. Here, only the two-dimensional problem is considered. The inverted pendulum is unstable in that it may fall over any time unless a suitable control force is applied. It is assumed that the pendulum mass is concentrated at the end of the rod as shown in the figure. The rod is assumed to be massless. The control force u is applied to the cart.

In the diagram, Θ is the angle of the rod from the vertical. It is assumed that Θ is small enough so that it is

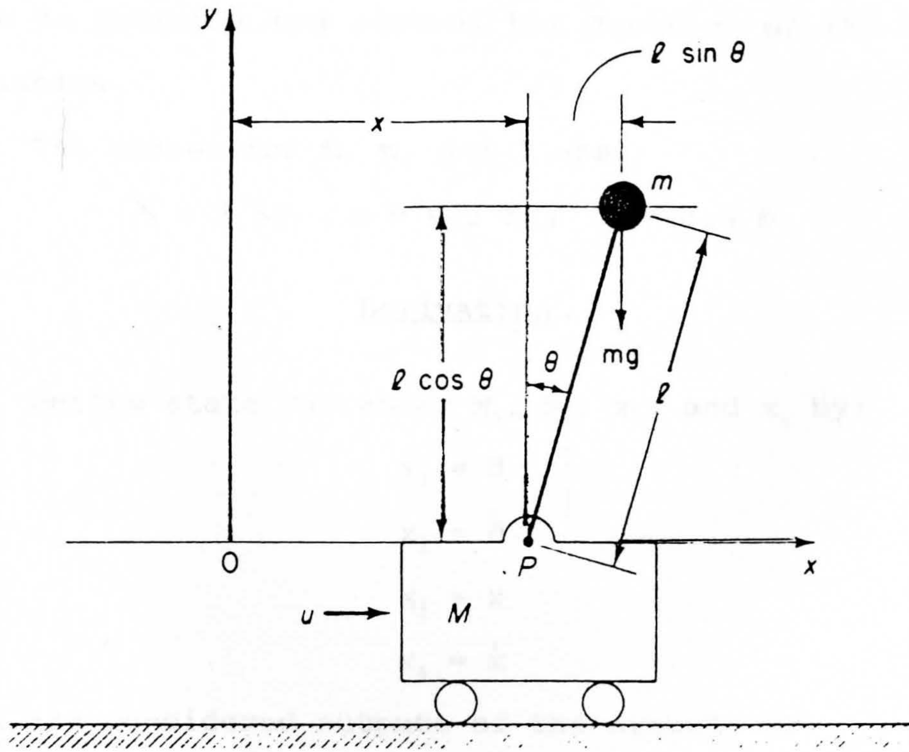

Fig. 6.1. Inverted Pendulum System

reasonable to approximate sin θ by θ, cos θ by 1, and that $\theta\dot{\theta}^2 \approx 0$. These assumptions will linearize the systems non-linear equations. It is desired to keep the inverted pendulum upright as much as possible and control the position of the cart in step fashion.

The values for M, m, and l are:

$$M = 2 \text{ Kg}, \quad m = 0.1 \text{ Kg}, \quad l = 0.5 \text{ m}$$

Derivation

Define state variables x_1, x_2, x_3, and x_4 by:

$$x_1 = \theta$$
$$x_2 = \dot{\theta}$$
$$x_3 = x$$
$$x_4 = \dot{x}$$

θ and x are considered outputs of the system, or

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \theta \\ x \end{bmatrix} = \begin{bmatrix} x_1 \\ x_3 \end{bmatrix}$$

The state space representation of the system is:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{M+m}{Ml} g & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{m}{M} g & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} 0 \\ -\frac{1}{Ml} \\ 0 \\ \frac{1}{M} \end{bmatrix} u \tag{6-1}$$

and

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \tag{6-2}$$

After substituting the numerical values:

$$\boldsymbol{A} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 20.601 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -0.4905 & 0 & 0 & 0 \end{bmatrix}, \quad \boldsymbol{B} = \begin{bmatrix} 0 \\ -1 \\ 0 \\ 0.5 \end{bmatrix}, \quad \boldsymbol{C} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Using the pole placement method outlined in chapter V, the following values for **K** and k_I are determined.

$$\boldsymbol{K} = [-157.6336 \ -35.3733 \ -56.0652 \ -36.7466]$$

$$k_I = -50.9684$$

These values were used as input to the program.

Hardware Setup

After considering both the cost and program efficiency it was decided that four T800 transputers would be used. They

were arranged in a loop as shown in figure 6.2. The links were connected as follows:

(a) Link 0 of the master processor was connected to the host (IBM PC) for input and output.

(b) Link 2 of processor1 was connected to link 1 of processor2.

(c) Link 2 of processor2 was connected to link 1 of processor3.

(d) Link 3 of processor3 was connected to link 3 of processor1 to complete the loop.

Software

Software consisted of three parts:

(a) A batch file to run all the different tools.

(b) The configuration file to describe the hardware software relationship to be used by the configure tool.

(c) The four programs, comprising the control software, which were placed on the four transputers.

Batch File

First, all the four C programs were compiled, using icc, for the T800 transputer. Then, all the four compiled programs were linked, using ilink with the appropriate libraries. It is important to note that the program thes1.c had to be linked with the startup.lnk library. This was done because this program would reside on the master processor and would need to interact with the host for input and output. The other

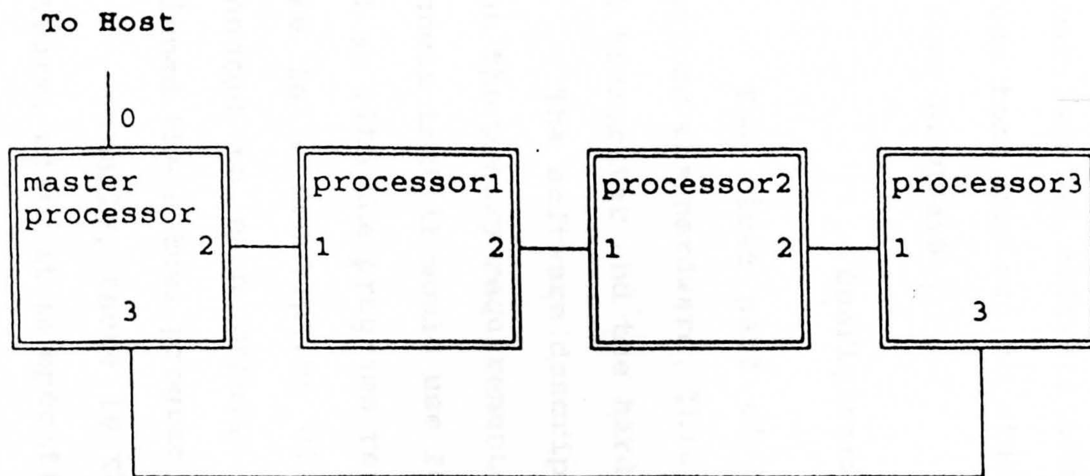

Fig. 6.2. Hardware Setup

three programs were linked with the reduced library because they only needed interprocessor communication.

Following this, the config tool was run on the configuration description. This would check for any irregularities in the description. Then, icollect tool was run for the marriage of software and hardware description. Finally, iserver tool was run to start the simultaneous execution of the four programs.

Configuration Description

The first part of the configuration file thes.cfs describes the hardware. This includes the memory available on each transputer and the hard link connection information.

The software description starts with the information about the memory requirements of each module and the software channels that it would use for communication with the host as well as with the programs residing on other transputers. Then there is a description of how the software channels are connected to each other. Following that, each module is assigned the actual program that it would be using.

Finally, there is the marriage between hardware and software, where it is specified that which program will reside on which transputer.

Control Software

Of the four programs, thes1 and thes4 did the actual computation while thes2 and thes3 acted as buffers so that

idle time was minimized for the master processor and processor3. This was done to take advantage of parallel processing where the overall run-time is reduced due to computation load being shared over multiple processors.

Program thes1 was solely responsible for input and output since it resided on the master processor which had access to the host input and output devices. During input, if at all possible, processor3 was kept busy in some other calculation.

The actual solution to the problem was done by an improvised version of the Runge-Kutta method. The computations were shared by the master processor and processor1.

All the results were saved on a disk resident on the host for possible graphing at a later time. It should be noted that this version of C does not have graphics capabilities.

Results

The reference input to the plant was 0.5. From figure 6.2 it can be seen that the state that was to be controlled ($x_3(t)$) reached 0.5 in less than 3 seconds. Moreover, the behavior in the transient region is acceptable.

Another point to note is that a dry run (that is, without printing the output) of this program ran more than twice as fast on the transputer system as compared to a similar program run on the IBM 486.

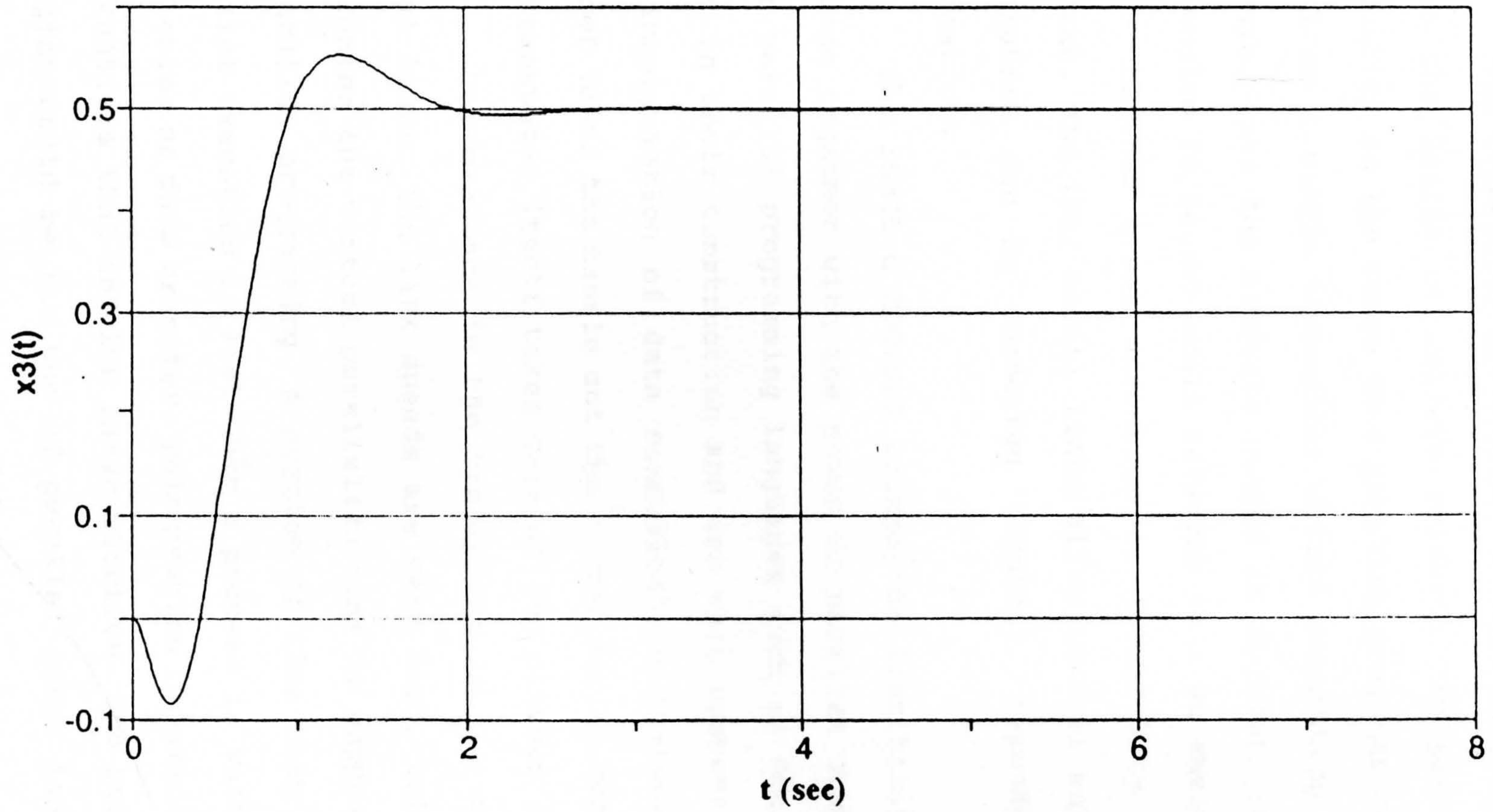

Fig. 6.3. Response Curve for $x_3(t)$

Conclusion

The transputer environment brings parallel computing within the realms of college research. It provides great flexibility in the sense that parallel programs can first be tested on a single transputer before committing oneself to hardware. Once the hardware needed is decided, it is fairly inexpensive to set up small networks in a PC environment.

Another great advantage is flexibility in forming networks. The four serial links allow several ways in which transputers can be connected together, depending on the application.

The INMOS C TOOLSET brings the flexibility of the C language together with the power of parallel computing. The other parallel programming languages such as Occam are very rigid in their construction and are very cumbersome to use. The incorporation of data communication statements in the toolset takes the hassle out the normal serial communication. The transputer itself takes care of any clashes in timing.

One drawback in the transputer is the speed of the serial links. The link speeds are very high, but they still compromise the virtual parallelism that is supposed to exist in parallel programming. A portion of time that is saved by parallel execution is lost when a process is waiting for the completion of data transfer. This problem is specially evident in transfers that involve large matrices. One possible solution could be the use of parallel links instead of the serial links by INMOS in the future.

Another drawback is the way the toolset is set up. It should be on a par with other packages available today, such as Borland TURBO C. Running the tools from the operating system seems very primitive after having worked with much more advanced systems.

Finally, the literature available on both hardware and software leaves a lot to be desired. The information is very sketchy and there is a profound lack of examples. This, however, is quite understandable since this is a fairly new system.

Recommendations

It takes considerable amount of time to learn to program in a parallel environment. It is recommended that anyone willing to do further research on this subject should consult this thesis, as well as that of Mr. S. V. Chala, in order to cut down the amount of time needed to learn to use the INMOS C or Occam Toolsets as well as getting used to programming in a parallel multiprocessor environment.

A good follow up on this thesis would be the use of a multitransputer system in an actual control process where a very high sampling rate is required. The maximum clock speed available on a single T800 transputer is 35 MHz. A network of transputers sampling the same signal could deliver sampling rates not attainable by any other means. If it were an embedded system containing no calls to the host system, the performance could be improved even further.

APPENDIX

<u>Software Used in the Thesis</u>

Batch File

This file consists of all the commands that were needed to build the program. It is much easier to have the commands in a batch file, rather than run them individually. The batch file can be edited any time. The contents of the batch file are given below:

```
icc thes1.c /t800
icc thes2.c /t800
icc thes3.c /t800
icc thes4.c /t800
ilink thes1.tco /t800 /f startup.lnk
ilink thes2.tco /t800 /f startrd.lnk
ilink thes3.tco /t800 /f startrd.lnk
ilink thes4.tco /t800 /f startrd.lnk
icconf thes.cfs
icollect thes.cfb
iserver /sb thes.btl /se
```

Configuration File

The configuration file, thes.cfs, contains the description that will be needed by the icconf tool. This description is written in a C like language. It contains the hardware description, the software description, the placement of software channels on hard-links, and the placement of software programs on individual processors.

The file contents are given below:

```
/* Hardware Description */
T800 (memory = 2M) master_processor;
T800 (memory = 1M) mult_processor1;
T800 (memory = 1M) mult_processor2;
T800 (memory = 1M) mult_processor3;
connect master_processor.link[0], host;
connect master_processor.link[2], mult_processor1.link[1];
connect mult_processor1.link[2], mult_processor2.link[1];
connect mult_processor2.link[2], mult_processor3.link[1];
connect mult_processor3.link[3], master_processor.link[3];
/* Software description */
process (stacksize = 400k, heapsize = 400K,
interface (input in, output out, input frommult1, output
tomult1, input frommult3, output tomult3)) master;
process (stacksize = 400k, heapsize = 400k,
interface (input frommaster, output tomaster,
input frommult2, output tomult2)) mult1;
process (stacksize = 400k, heapsize = 400k,
interface (input frommult1, output tomult1,
input frommult3, output tomult3)) mult2;
process (stacksize = 20k, heapsize = 10k,
interface (input frommult2, output tomult2,
input frommaster, output tomaster)) mult3;
input from_host;
output to_host;
connect master.in, from_host;
```

```
connect master.out, to_host;
connect master.frommult1, mult1.tomaster;
connect master.tomult1, mult1.frommaster;
connect mult1.frommult2, mult2.tomult1;
connect mult1.tomult2, mult2.frommult1;
connect mult2.frommult3, mult3.tomult2;
connect mult2.tomult3, mult3.frommult2;
connect mult3.frommaster, master.tomult3;
connect mult3.tomaster, master.frommult3;
use "thes1.lku" for master;
use "thes2.lku" for mult1;
use "thes3.lku" for mult2;
use "thes4.lku" for mult3;
/* Module Placement */
place master on master_processor;
place mult1 on mult_processor1;
place mult2 on mult_processor2;
place mult3 on mult_processor3;
place to_host on host;
place from_host on host;
```

Control Software

The control software consisted of four programs, one for each transputer. Thes1.c and Thes4.c did the actual calculation using the Runge-Kutta method, while Thes2.c and Thes3.c acted as buffer programs to minimize processor idle time. The listing of each of the programs is given below.

Thes1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <misc.h>
#include <channel.h>
#include <time.h>
#define CLEAR "\x1B[2J"
double  MAT2[4][4],MAT4[4],X[4];
int     NUMVAR,NUMU;
double func1(int EQNUM)
/* Calculates (A-BK)X + BKiG */
{
 double k;
 int    count;
 k=0;
 for (count=0;count<=NUMVAR-1;count++)
    k=k+MAT2[EQNUM][count]*X[count];
 k=k+MAT4[EQNUM];
 return k;
}
int main(int argc, char* argv[])
{
 Channel *tomult1, *frommult1;
 Channel *tomult3, *frommult3;
 int    i, j, k, N;
 double  a, b, h, t;
 double  K1[4], K2[4], K3[4], K4[4];
```

```
double  A[4][4], B[4][4], K[4][4], Ki[4][4], G[4];
double  MAT5[4],MAT6[4],U[4];
FILE    *stream;
tomult1= (Channel *) get_param(4);
frommult1= (Channel *) get_param(3);
tomult3= (Channel *) get_param(6);
frommult3= (Channel *) get_param(5);
/**OPEN DATA FILE FOR GRAPH**/
if ((stream = fopen("odeser.dat", "w"))==NULL)
   {
    printf("Could not open file");
    abort();
   }
/******************************/
/*        GET DATA           */
printf(CLEAR);
printf("Enter starting point: ");
scanf("%lf",&a);
printf("Enter ending point: ");
scanf("%lf",&b);
printf("Enter number of intervals: ");
scanf("%d",&N);
printf("Enter system order: ");
scanf("%d",&NUMVAR);
printf("Enter number of inputs(<=%d): ",NUMVAR);
scanf("%d",&NUMU);
ChanOutInt(tomult3,NUMVAR);
```

```
ChanOutInt(tomult3,NUMU);
printf(CLEAR);
for (i=0;i<=NUMVAR-1;i++)
    {
     printf("Enter initial condition for X%d: ",i+1);
     scanf("%lf",&X[i]);
    }
 printf(CLEAR);
 printf("\nA Matrix\n");
 for (i=0;i<=NUMVAR-1;i++)
     {
      for(j=0;j<=NUMVAR-1;j++)
        {
         printf("\nA[%d][%d]: ",i+1,j+1);
       /* READ A MATRIX */
         scanf("%lf",&A[i][j]);
        }
     }
 printf(CLEAR);
 printf("\nB Matrix\n");
 for (i=0;i<=NUMVAR-1;i++)
     {
      for(j=0;j<=NUMU-1;j++)
        {
```

```
            printf("\nB[%d][%d]: ",i+1,j+1);
          /* READ B MATRIX */
            scanf("%lf",&B[i][j]);
           }
       }
  printf(CLEAR);
  printf("\nK Matrix\n");
  for (i=0;i<=NUMU-1;i++)
       {
        for(j=0;j<=NUMVAR-1;j++)
           {
            printf("\nK[%d][%d]: ",i+1,j+1);
         /* READ K MATRIX */
            scanf("%lf",&K[i][j]);
           }
       }
  ChanOut(tomult3,(void *) A,16*sizeof(double));
  ChanOut(tomult3,(void *) B,16*sizeof(double));
  ChanOut(tomult3,(void *) K,16*sizeof(double));
  printf(CLEAR);
  printf("\nKi Matrix\n");
  for (i=0;i<=NUMU-1;i++)
       {
        for(j=0;j<=NUMU-1;j++)
           {
            printf("\nKi[%d][%d]: ",i+1,j+1);
         /* READ Ki MATRIX */
```

```
            scanf("%lf",&Ki[i][j]);
           }
       }
  printf(CLEAR);
  printf("\nG Matrix\n");
  for (i=0;i<=NUMU-1;i++)
       {
        printf("\nG[%d]: ",i+1);  /* READ G MATRIX */
        scanf("%lf",&G[i]);
       }
  /*****************************/
  for (i=0;i<=NUMU-1;i++)
       {
        MAT5[i]=0;
        for (j=0;j<=NUMU-1;j++)
         /* MAT5=KiG */
         MAT5[i]=MAT5[i]+Ki[i][j]*G[j];
       }
  for (i=0;i<=NUMVAR-1;i++)
       {
        MAT4[i]=0;
        for (k=0;k<=NUMU-1;k++)
         /* MAT4=BKiG */
         MAT4[i]=MAT4[i]+B[i][k]*MAT5[k];
       }
  ChanOut(tomult1,(void *) MAT4,4*sizeof(double));
  ChanOut(tomult3,(void *) MAT4,4*sizeof(double));
  ChanIn(frommult3,(void *) MAT2,16*sizeof(double));
```

```
/*
for(i=0;i<=NUMVAR-1;i++)
   {
    for(j=0;j<=NUMVAR-1;j++)
       printf("A-BK[%d][%d]=%f ",i+1,j+1,MAT2[i][j]);
    printf("\n");
   }
*/
h=(b-a)/N;
t=a;
 printf(CLEAR);
 printf("t=%4.2f  ",t);
 for (i=0;i<=NUMVAR-1;i++)
     printf("X[%d]=%6.3f  ",i+1,X[i]);
 for (i=0;i<=NUMU-1;i++)
     printf("U[%d]=%6.3f  ",i+1,U[i]);
 printf("\n");
 fprintf(stream,"%.3f,",t);
 for (i=0;i<=NUMVAR-1;i++)
     fprintf(stream,"%.3f,",X[i]);
 for (i=0;i<=NUMU-1;i++)
     fprintf(stream,"%.3f,",U[i]);
 fprintf(stream,"\n");
 for (i=1;i<=N;i++)
     {
      for (j=0;j<=NUMVAR-1;j++)
    K1[j]=func1(j)*h;
```

```
  for (j=0;j<=NUMVAR-1;j++)
K2[j]=func1(j)*h;
  for (j=0;j<=NUMVAR-1;j++)
K3[j]=func1(j)*h;
  for (j=0;j<=NUMVAR-1;j++)
K4[j]=func1(j)*h;
  for (j=0;j<=NUMVAR-1;j++)
X[j]=X[j]+(K1[j]+2*K2[j]+2*K3[j]+K4[j])/6;
  for (j=0;j<=NUMU-1;j++)
{
 MAT6[j]=0;
 for (k=0;k<=NUMVAR-1;k++)
     MAT6[j]=MAT6[j]-K[j][k]*X[k];
      /* MAT6=-KX */
 U[j]=MAT5[j]+MAT6[j];
    /*   U=-KX+KiG */
}
  t=a+i*h;
  printf("t=%4.2f  ",t);
  for (j=0;j<=NUMVAR-1;j++)
printf("X[%d]=%6.3f  ",j+1,X[j]);
  for (j=0;j<=NUMU-1;j++)
printf("U[%d]=%6.3f  ",j+1,U[j]);
  printf("\n");
  fprintf(stream,"%.3f,",t);
  for (j=0;j<=NUMVAR-1;j++)
fprintf(stream,"%.3f,",X[j]);
```

```
      for (j=0;j<=NUMU-1;j++)
    fprintf(stream,"%.3f,",U[j]);
      fprintf(stream,"\n");
     }
  exit_terminate(EXIT_SUCCESS);
}
```

Thes2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <misc.h>
#include <channel.h>
#include <time.h>

double MAT2[4][4], MAT4[4];
int NUMVAR, NUMU;

int main(int argc, char* argv[])
{
Channel *tomaster, *frommaster;
Channel *tomult2, *frommult2;

tomaster= (Channel *) get_param(2);
frommaster= (Channel *) get_param(1);
tomult2= (Channel *) get_param(4);
frommult2= (Channel *) get_param(3);
ChanIn(frommult2, (void *) MAT2, 16*sizeof(double));
```

```
ChanIn(frommaster, (void *) MAT4, 4*sizeof(double));
}
```

Thes3.c

```
#include <stdio.h>
#include <stdlib.h>
#include <misc.h>
#include <channel.h>
#include <time.h>
double MAT2[4][4],MAT4[4];
int NUMVAR, NUMU;
int main(int argc, char* argv[])
{
Channel *tomult1, *frommult1;
Channel *tomult3, *frommult3;
tomult1= (Channel *) get_param(2);
frommult1= (Channel *) get_param(1);
tomult3= (Channel *) get_param(4);
frommult3= (Channel *) get_param(3);
NUMVAR=ChanInInt(frommult3);
NUMU=ChanInInt(frommult3);
ChanIn(frommult3, (void *) MAT2,16*sizeof(double));
ChanOut(tomult1,(void *) MAT2,16*sizeof(double));
ChanIn(frommult3, (void *) MAT4,4*sizeof(double));
}
```

Thes4.c

```
#include <stdio.h>
#include <stdlib.h>
#include <misc.h>
#include <channel.h>
#include <time.h>
double A[4][4],B[4][4],K[4][4],MAT1[4][4],MAT2[4][4],
       MAT4[4];
int    i, j, k, NUMVAR, NUMU;
int main(int argc, char* argv[])
{
Channel *tomult2, *frommult2;
Channel *tomaster, *frommaster;
tomult2= (Channel *) get_param(2);
frommult2= (Channel *) get_param(1);
tomaster= (Channel *) get_param(4);
frommaster= (Channel *) get_param(3);
NUMVAR=ChanInInt(frommaster);
NUMU=ChanInInt(frommaster);
ChanOutInt(tomult2,NUMVAR);
ChanOutInt(tomult2,NUMU);
ChanIn(frommaster, (void *) A,16*sizeof(double));
ChanIn(frommaster, (void *) B,16*sizeof(double));
ChanIn(frommaster, (void *) K,16*sizeof(double));
 for (i=0;i<=NUMVAR-1;i++)
     {
      for (j=0;j<=NUMVAR-1;j++)
```

```
            {
             MAT1[i][j]=0;
            /* MAT2=A-BK */
             for (k=0;k<=NUMU-1;k++)
                 MAT1[i][j]=MAT1[i][j]+B[i][k]*K[k][j];
             MAT2[i][j]=A[i][j]-MAT1[i][j];
            }
      }
ChanOut(tomult2,(void *) MAT2,16*sizeof(double));
ChanIn(frommaster, (void *) MAT4,4*sizeof(double));
ChanOut(tomult2,(void *) MAT4,4*sizeof(double));
ChanOut(tomaster,(void *) MAT2,16*sizeof(double));
}
```

REFERENCES

[1] INMOS Limited, ANSI C Toolset User Manual, 1990.

[2] INMOS Limited, ANSI C Toolset Language Reference, 1990.

[3] INMOS Limited, Transputer Reference Manual, 1988.

[4] G. R. Andrews and F. B. Schneider, Concepts and Notations for Concurrent Programming, ACM Computing Surveys, Vol. 15, 1983.

[5] Katsuhiko Ogata, Modern Control Engineering, Prentice Hall, 1990.