

**IMPLEMENTATION OF REGULAR EXPRESSION
TRANSFORMATION ALGORITHMS
ON THE HYPERCUBE**

BY

SRIDHAR MANDYAM

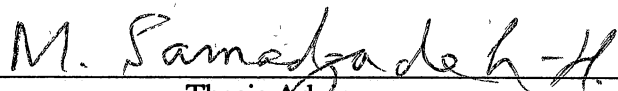
Master of Technology
Karnataka Regional Engineering College
Surathkal, India
1988

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1991

Thesis
1991
M2736
cop. 2

IMPLEMENTATION OF REGULAR EXPRESSION
TRANSFORMATION ALGORITHMS
ON THE HYPERCUBE

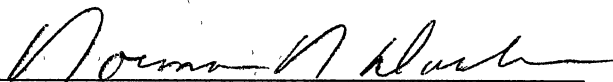
Thesis Approved:



Thesis Adviser







Dean of the Graduate College

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION.....	1
II. PARALLEL PROCESSING.....	4
2.1 Introduction.....	4
2.2 Classification of Computers.....	7
2.2.1 SISD Computer.....	7
2.2.2 SIMD Computer.....	8
2.2.3 MISD Computer.....	8
2.2.4 MIMD Computer.....	8
2.3 Types of Parallel Computers.....	10
2.3.1 Pipelined Processors.....	10
2.3.2 Vector Processors.....	11
2.3.3 Array Processors.....	11
2.3.4 Systolic Processors.....	12
2.3.5 Multiprocessors.....	13
2.4 Performance Measures.....	14
2.5 The iPSC/2 Parallel Computer.....	15
2.5.1 Hypercube and the iPSC/2.....	15
2.5.2 iPSC/2 Node Architecture.....	17
2.5.3 NX/2 Operating System.....	19
III. FUNDAMENTALS OF LANGUAGE THEORY.....	21
3.1 Preliminaries.....	21
3.2 Regular Expressions.....	22
3.3 Finite Automata.....	23
3.4 Transformation Algorithms.....	26
3.4.1 Transformation T1 - RE to NFA.....	26
3.4.2 Transformation T2 - Removing ϵ -moves.....	29
3.4.3 Transformation T3 - NFA to DFA.....	31
3.4.4 Transformation T4 - Minimizing the DFA.....	32
3.4.5 Transformation T5 - DFA to RE.....	34
3.4.6 Transformation T6 - RE Equations for the DFA.....	37
3.4.7 Transformation T7 - Solution of RE Equations.....	37
IV. MULTIPROCESSOR SCHEDULING.....	40
4.1 A Partitioning Approach.....	40

Chapter	Page
4.2 Graphical Representation of the Problem	42
4.2.1 Precedence Graphs	43
4.2.2 Rooted Trees	43
4.3 Scheduling Algorithms	44
4.3.1 Gantt-Chart Representation of a Schedule	45
4.3.2 Scheduling Algorithm A	45
4.3.3 Another Partitioning Approach	48
4.3.4 Scheduling Algorithm B	48
4.3.5 Scheduling Algorithm C	53
4.4 Implementation and Optimization Issues	56
4.4.1 Optimal Number of Processors	56
4.4.2 Suitability Issue	57
4.4.3 Memory Allocation Issues	58
4.4.4 Machine-Independent Communication Issues	59
4.4.5 Machine-Dependent Communication Issues	60
 V. SUMMARY AND FUTURE WORK	 62
5.1 Summary	62
5.2 Future Work	64
 REFERENCES	 65
 APPENDIXES	 69
APPENDIX A - USER MANUAL	70
APPENDIX B - C PROGRAMS	78
APPENDIX C - EXECUTION DETAILS THROUGH A CYCLE OF TRANSFORMATIONS	142
APPENDIX D - SAMPLE RUNS FOR A CONVERGENT CASE	146
APPENDIX E - SAMPLE RUNS FOR A DIVERGENT CASE	151
APPENDIX F - COMPARISON OF PERFORMANCE MEASURES	156

LIST OF TABLES

TABLE		Page
I.	Chronology of Parallel Processing Projects	6
II.	Computer Systems Based on Flynn's Classification.	10

LIST OF FIGURES

Figure	Page
1. Flynn's Classification of Computers.	9
2. A Pipeline Structure.	11
3. Functional Structure of an Array Processor.	12
4. The Concept of a Systolic Array Processor	13
5. Types of Multiprocessors.	14
6. n-dimensional Hypercube for n= 0, 1, 2, and 3	16
7. iPSC/2 Node Block Diagram.	18
8. Cycle of Transformations Performed on a Regular Expression	26
9. Rules for Synthesizing an NFA from Automata M1 and M2	27
10. NFA With ϵ -moves for the RE 0^*1^*	28
11. Algorithm for Removing ϵ -moves in an NFA	30
12. NFA Without ϵ -moves for the RE 0^*1^*	31
13. Algorithm to Construct a DFA from an NFA	31
14. DFA Equivalent to the NFA for the RE 0^*1^*	32
15. Algorithm for Marking Pairs of Inequivalent States in a DFA	33
16. The Minimized DFA for the RE 0^*1^*	34
17. Algorithm to Build an RE Representing an FA	35
18. Two Cases in Deleting a State q_i	35
19. Algorithm for Solving a Set of RE Equations.	38
20. Partitioning an RE into Tasks	42
21. An Example of a Precedence Graph.	43

Figure	Page
22. An Example of a Rooted Tree.	44
23. Algorithm A: Scheduling a Rooted Tree on p Processors.	47
24. Schedule Obtained by Algorithm A on $p=2$ Processors.	47
25. Partitioning an RE with Common Sub-Expressions	49
26. (a) Graph with Tasks of Unequal Node Weights (b) Graph with Tasks of Equal Node Weights.	50
27. Illustration of Converting a DAG into a Rooted Tree	51
28. An Example of a Rooted Tree with Repeated Nodes and Its Label Table	52
29. Algorithm B: Scheduling a Rooted Tree with Repeated Nodes on p Processors	53
30. Schedule Obtained by Algorithm B on $p=2$ Processors	53
31. An Example of a DAG with Some Nodes Having Multiple Successors.	54
32. Algorithm C: Scheduling a DAG on p Processors Directly	55
33. Schedule Obtained by Algorithm C on $p=2$ Processors	56
34. Schedule Obtained by Algorithm A after Communication Optimization	60
35. An Example of a Rooted Tree and a Schedule to Illustrate the Look Ahead Approach.	60

CHAPTER I

INTRODUCTION

Seitz speculated that von Neumann uniprocessor systems' performance was approaching an asymptotic limit of nearly 3×10^9 operations per second [SEIT84]. Even with the ongoing tremendous advances in semiconductor technology, it is becoming increasingly difficult to obtain higher performance from single processor systems. Generally speaking, technology has reached a state that any further development would face certain physical constraints [LEA87]. Moreover, high performance systems like supercomputers have become unaffordable by many research organization due to the high price tags of such systems [KARIN87]. But there still remain several classes of applications for which high speed is crucial and beyond the capabilities of the fastest single processor machines available [HAYES88]. With this trend, there is a general approach to avoid the limitations of uniprocessor systems by using several processors. Parallel processing provides a possible solution in this regard [FOX88]. Other solutions include distributed processing and massively parallel systems.

As part of compiling a program written in a high level language, a phase called *lexical analysis* is performed [AHO86]. In this phase, strings of characters of a language denoting keywords, identifiers, constants, etc. are grouped together into single symbols called "tokens". A program which performs this phase is called a "Lexical Analyzer" or a "Scanner". "Regular Expression Notation" is a formalism which can be used for describing the tokens of a programming language. A "Finite Automaton" is a mathematical model of a recognizer which can be used to recognize the tokens of a programming language specified by a regular expression. These two tools (Regular Expressions and Finite Automata) form

the basis of a Scanner [FIS88]. There is a close relationship between the sets described by regular expressions and the sets identified by finite automata, and there are a set of transformations that can be performed on them. Such transformations have been described in the literature by sequential algorithms [BRZO62, AHO72, HOP79, SUD88]. Two cycles of such transformations were considered in this thesis (as described in section 3.4).

As one objective of this thesis, a regular expression was subjected to the set of transformations along each cycle a number of times. As expected, the form of the regular expression changes through every iteration of each of the cycles. The changes occurring in the form of regular expressions was investigated. One of the cycles of transformations appeared to produce regular expressions that, although not necessarily irredundant and minimized, are in a closed form that can be loosely called a "canonical form". Thus we can say that this cycle "converges". The other cycle yields the canonical form generally after a larger number of iterations than the first one (or it may not even produce a canonical form). Thus we can say that, the latter cycle does not always converge, or it "diverges". This seems to be attributable to a particular transformation in the latter cycle. These details are covered in section 4.4 and Appendixes D and E.

As another objective of this thesis, the parallelism existing in the sequential algorithms for these transformations was exploited to develop parallel algorithms. Subsequently, the parallel algorithms were implemented in the C programming language on a typical parallel processor, namely the Intel's iPSC/2 32-processor, distributed-memory system, with a hypercube interconnection topology between the processors. Some multiprocessor performance measures, such as *speed-up*, *processor efficiency*, and *serial fraction* were evaluated and the results have been discussed. In order to do so, the programs developed were executed on a varying number of processors for different regular expressions of different sizes. The performance measures and results have been summarized (section 5.1).

Implementing parallel algorithms involves such multiprocessor-dependent issues as partitioning and scheduling. The problem of partitioning a problem for the iPSC/2 parallel processor, was studied and implemented. Subsequently, a suitable multiprocessor scheduling algorithm, namely Hu's algorithm [HU61], was used to optimally schedule the tasks in the problem, so as to achieve nearly uniform processor utilization and reduce communication overhead. Moreover, extensions to Hu's scheduling algorithm (namely Algorithms B and C) have been derived to tackle its limitations and they have been realized in developing better schedules for the given problem. An important observation on the number of processors required for scheduling a given problem is also derived from these algorithms. The details are given in various sections of Chapter IV.

The thesis report is divided into various chapters relating to various topics. Initially, related literature work on the subject of language theory and parallel processing is described in chapters I and II. Subsequently, the contributions made by the thesis in relation to the development and implementation of the parallel algorithms for regular expressions, and in relation to the concept of the changing form of a regular expression subjected to a set of transformations, are discussed in Chapter IV. Finally, the report concludes with a discussion of the results and future improvements to this project in Chapter V.

CHAPTER II

PARALLEL PROCESSING

Obtaining more performance from the von Neumann model is becoming increasingly difficult. The task of solving very complex problems within specified time periods continues to surpass the capabilities of the world's fastest and most powerful computers [HWANG89]. Parallel processing holds a good promise to achieve high performance in solving such complex tasks [KUCK78, HOCK81, HWANG84]. This chapter presents an overview of the concepts of parallel processing, including details of the Intel's iPSC/2 Parallel Computer, which is the implementation platform used in this thesis.

2.1 Introduction

The term parallel processor refers to a class of systems that try to increase the computing speed by performing more than one computation concurrently on more than one processor. Connecting a number of powerful processors or Processing Elements (PEs) together into a single system and making them solve a single complex problem through cooperation with each other, is the underlying principle of parallel processing. Parallelism commonly means to do more than one thing at once, which could be interpreted in several ways [DES87]: doing n different activities at once; doing one activity in n simultaneous parts; doing n activities staggered in time; or using k resources for n jobs or k resources for one job. Events occurring on different processors during the same time interval are termed "parallel" events, and those occurring at the same instant are termed "simultaneous" events [HWANG84].

One of the factors for the spread and popularity of parallel processing has been improvements in the hardware technology. Although Grosch's law [DOR85] states that the best price/performance can be obtained with the most powerful uniprocessor, it is no longer true that a system consisting of less powerful processors will have a lower performance than a single large processor of the same total cost [LEA87]. With the important recognition attained by supercomputing and supercomputers among researchers who need more than 100 MFLOPS (Millions of Floating Point Operations Per Second) computing performance, there is a need for supercomputing-class performance that is more affordable [KARIN87]. Summing these issues, conventional architectures are close to their performance limits due to physical effects like the speed of light, supercomputing resources are generally unaffordable due to their high price tags, and researchers' quest for solving computationally intensive problems has been ever increasing. With these trends, research has opened gateways to the field of parallel processing.

Concurrent or parallel architectures are not a new idea. As early as 1945, Vannevar Bush described some proposals along these lines. John von Neumann also preferred the parallel approach [HOCK81], but dropped the idea due to the unreliability and bulkiness of vacuum tubes. In the 1950s, Slotnick and his collaborators at IBM proposed some parallel architectures like Solomon [SLOT62] and Illiac IV [BARN68]. But the first general-purpose computer commercially available, which could perform several operations concurrently, was the Heterogeneous Element Processor (HEP) [JORD83]. A partial list of the many parallel processing projects that have been completed or currently under progress is illustrated in TABLE I. We can see that some parallel computers have been proposed earlier but without achieving success. This could be attributed to the technology that was inadequate at that time and the general preference to the the conceptual simplicity of the sequential stored-program computer [DEN85].

Currently, parallel processing is seen as having the potential to improve such factors as, cost/performance, productivity, and reliability. Some applications suitable for

TABLE I
CHRONOLOGY OF PARALLEL PROCESSING PROJECTS

1960-69	1970-79	1980-84	1985-87	1988-90	1990-
<p style="text-align: center;">Solomon STARAN CDC 6600 IBM 360/91 UNIVAC1108 CDC 7600</p>	<p style="text-align: center;">STAR-100 AP-120b IBM 360/195 IBM 370/165 IBM 370/168 ILLIAC IV PEPE Cray-1 HEP-1 Pluribus Cm* Tandem C.mmp</p>	<p style="text-align: center;">Cyber 203 Cyber 205 IBM 3033 BSP Cray X-MP Hitachi S-810 Fujitsu VP-100 Hitachi S-820 Fujitsu VP-200 NEC SX-1 DAP MPP</p>	<p style="text-align: center;">Cray-2 Cray X-MP/4 NCube NEC SX-2 Convex XP iPSC/1 BBN Butterfly CM-1 FPS 164 Ametek S-14 Fujitsu VP-400 ETA 10 Sequent 8000 HEP-2 Loral Flex</p>	<p style="text-align: center;">NEC IPP Alliant FX/8 Cray Y-MP iPSC/2 FPST Encore Elxsi 6400 Ametek 2010 Hitec-10 IBM 3090/400 IBM GF11</p>	<p style="text-align: center;">Cray-3 ETA-30 IBM RP3</p>

parallel processing, where high speed is crucial, include scientific calculations [WIL87], 3-dimensional partial differential equations solution ([PETER85] as cited in [HWANG89]), monte-carlo techniques in physics and chemistry [KALO87], signal processing of sampled data [HWANG89], graph problems [HIRS82], and weather modelling.

2.2 Classification of Computers

There are many ways of classifying computer systems based on their structure and/or behavior. Flynn's classification is based on multiplicity of instruction streams and data streams in a computer system [FLYN72], Feng's classification is based on the degree of parallelism [FENG77], and Handler's classification is based on the degree of parallelism and pipelining in various subsystems [HAND77]. Kuck's classification [KUCK78], which replaced the data streams with execution streams in Flynn's classification, gives more detail at the hardware level. Other classification schemes have been presented by Treleaven [TREL82], Gajski and Peir [GAJ85], etc.

Flynn's classification is simple and also the most widely used. In this classification, the flow of instructions fetched by the CPU from the memory forms an "instruction stream" (IS), and the flow of operands between the CPU and the memory forms a "data stream" (DS). The four machine organizations based on this classification are described in the following subsections.

2.2.1 SISD Computer

Single Instruction stream Single Data stream (SISD) organization consists of one processing element (PE) and one control unit (CU), and it represents the class of sequential computers. The general architecture of an SISD Computer is shown in Figure 1(a). In SISD computers, instructions are executed sequentially but may be overlapped in their execution stages (a technique called pipelining).

2.2.2 SIMD Computer

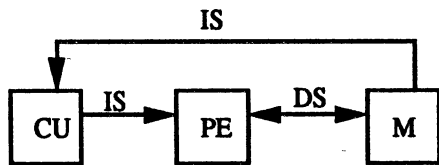
Single Instruction stream Multiple Data stream (SIMD) organization represents the class of machines consisting of multiple processing elements, which are controlled by a single control unit. The general structure of an SIMD machine is shown in Figure 1(b). The control unit sends the same instructions to all the PEs which operate on different data sets from distinct data streams. Some examples of systems belonging to this class are Illiac IV Array Processor, the Distributed Array Processor (DAP) [HOCK81], Associative Processors like the Massively Parallel Processor (MPP) [POTT86], and Connection Machine CM-1 [HILL85].

2.2.3 MISD Computer

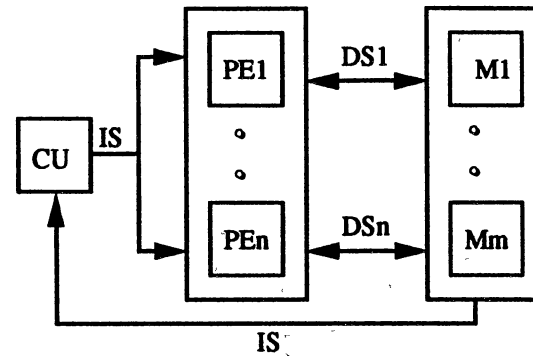
Multiple Instruction stream Single Data stream (MISD) organization consists of multiple processing elements and multiple control units. Its structure is illustrated in Figure 1(c). Each PE receives distinct instructions, but all of them operate on the same data set. Not many parallel processors fit into this category, except Fault-tolerant computers where several CPUs process the same data using different programs [HAYES88].

2.2.4 MIMD Computer

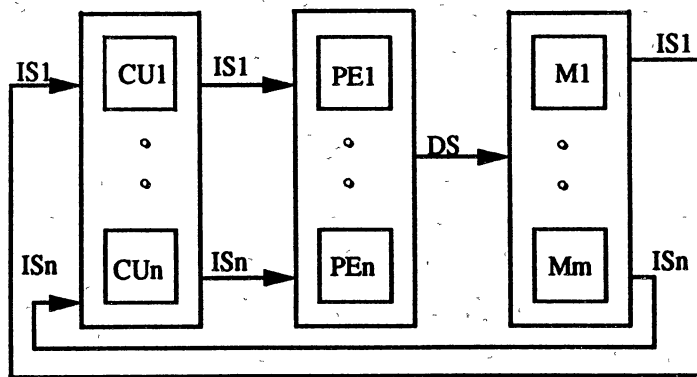
Multiple Instruction stream Multiple Data stream (MIMD) organization represents most multiprocessor systems having the ability to execute several programs simultaneously. Its structure is shown in Figure 1(d). It is almost similar to an MISD system except that each PE operates individually through its own instruction stream on its own data stream [HWANG84]. Since the same data space is shared by all processors, the processors need to interact with each other. If the degree of interactions among the processors is high in an MIMD computer, it is termed as "tightly" coupled. Otherwise, it is "loosely" coupled. Intel's iPSC Hypercube [SULL77, GRAH87], Cm* [GEHR87], NCube [PALM87], Cray X-MP/4, Sequent 8000 [ANI89], etc. belong to the MIMD class.



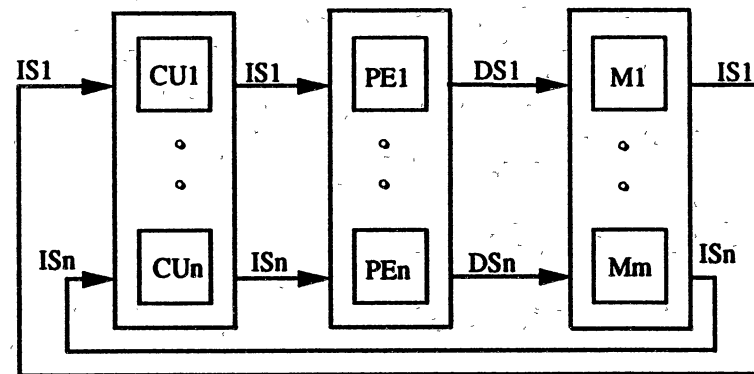
(a) Structure of an SISD Computer.



(b) Structure of an SIMD Computer.



(c) Structure of an MISD Computer.



(d) Structure of an MIMD Computer.

Figure 1. Flynn's Classification of Computers.

(Source: K. Hwang and F.A. Briggs. Computer Architecture and Parallel Processing, McGraw-Hill Book Co., New York, NY, p. 33, 1984.)

Table II [HWANG84] lists several systems under each of the three existing computer organizations (no real systems of MISD class exists).

TABLE II
COMPUTER SYSTEMS BASED ON FLYNN'S CLASSIFICATION

Organization	Computer systems
SISD	IBM 701, IBM 7090, PDP VAX 11/780, IBM 360, Cray-1, CDC Cyber-205, Fujitsu VP-200, FPS-164, TI-ASC
SIMD	Iliac IV, BSP, Staran, MPP, DAP, CM-1
MIMD	IBM 3081/3084, Cm*, Univac 1100/89, C.mmp, Cray-2, Cray X-MP, HEP, iPSC Hypercube, NCube, BBN Butterfly

2.3 Types of Parallel Processors

Parallel processors are categorized under the following architectural configurations.

2.3.1 Pipelined Processors

Pipelined processors are those which perform overlapped computations. In a pipelined processor different parts of a single operation are executed simultaneously in dissimilar modules connected as stages (called pipeline stages) into a cascade chain [KOG81]. The structure of a Pipeline Processor is shown in Figure 2. Each operand passes through several stages in successive time steps before it has been completely processed. The effect of all of the pipeline stages on a data element constitutes an operation. Hence pipeline computers are more tuned for vector processing, where component operations need to be repeated many times [KAIN89]. Typical examples of pipelined computers include Control Data Corporation's Star-100 series [CONT70], TI's Advanced Scientific Computer (ASC), Cray-1, Cray-2, etc.

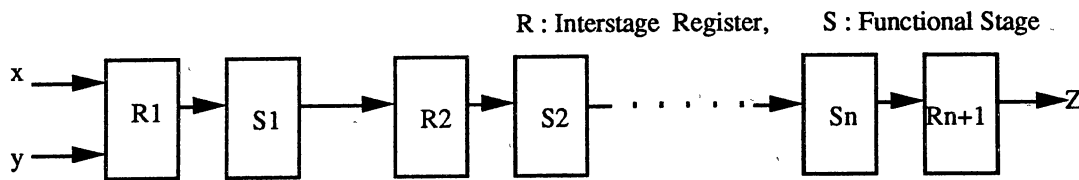


Figure 2. A Pipeline Structure.

(Source: R.Y. Kain. Computer Architecture: Software and Hardware, vol. II, Prentice Hall, Englewood Cliffs, NJ, p. 30, 1989.)

2.3.2 Vector Processors

Vector Processors, as their name implies, are suitable for performing computations on vector data. Vector processing is characterized by the performance of the same operation on all elements of a regular array or a vector simultaneously [ALMA89]. Kogge [KOG81], and Hockney and Jesshope [HOCK81] describe vector processing in detail. The basic idea of vector processing is outlined below. The multiplication of two 100-element vectors on a sequential computer would consist of a loop like, for $I = 1$ to 100 do $A(I) = B(I)*C(I)$. In addition to fetching 100 pairs of operands, the multiplication instruction is also fetched and decoded 100 times, which is a large overhead. Instead, a single vector instruction, indicating that the same operation be performed on all pairs of elements of the two vectors, can be used as $A(1:100) = B(1:100)*C(1:100)$.

2.3.3 Array Processors

According to Karplus [KARP87] "an Array Processor consists of a regularly connected array of processing elements under the supervision of one control unit". All the PEs perform the same function in synchronization with the help of a data-routing mechanism. In general terms, an array processor is seen as a rectangular grid, with each intersection denoting a PE, and the lines between intersections denoting common paths. The architecture of an array processor is shown in Figure 3. The only function of the PEs

is to receive data on the interfaces, operate on the data, and then send data back onto the interfaces. Most of the important functions are performed by the control unit which include specifying each PE's operation, properly routing data among the PEs through the interconnection network, controlling the transfer of data to and from the memory, etc. [HWANG84].

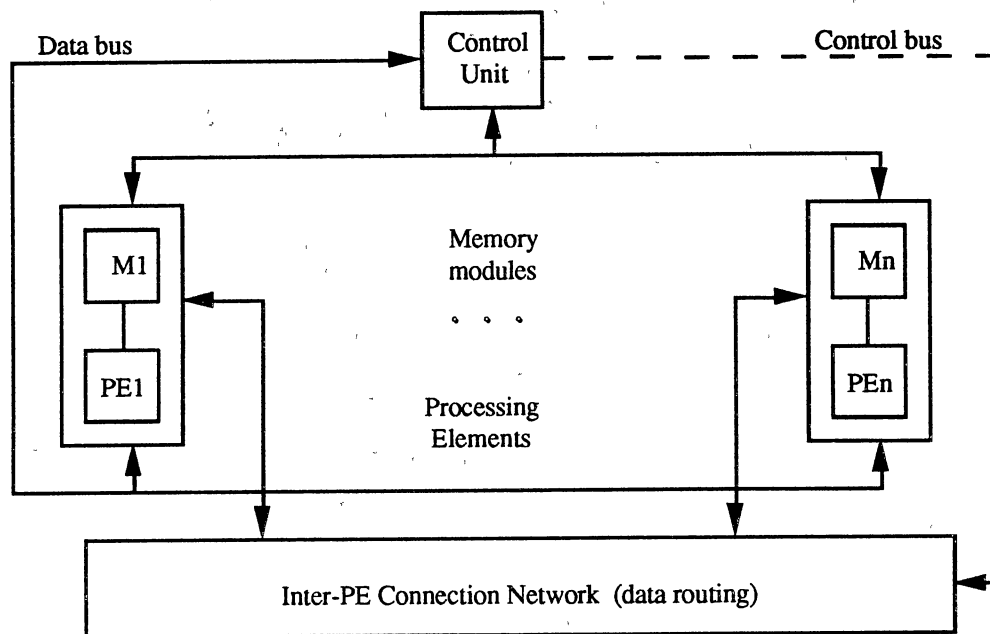


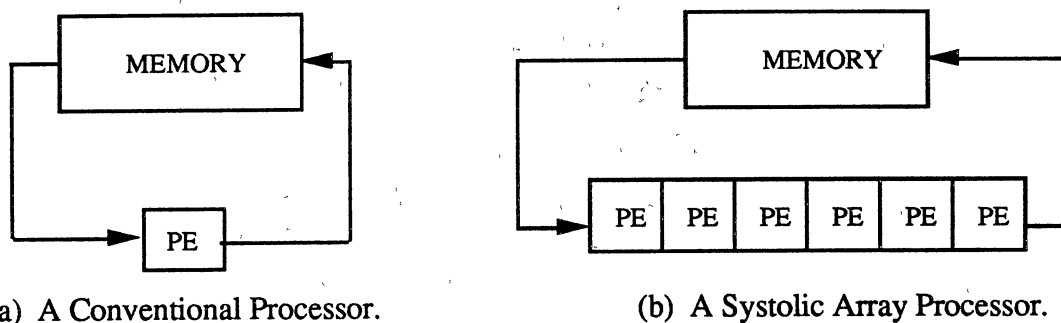
Figure 3. Functional Structure of an Array Processor.

(Source: K. Hwang and F.A. Briggs. Computer Architecture and Parallel Processing, McGraw-Hill Book Co., New York, NY, p. 24, 1984.)

2.3.4 Systolic Processors

A Systolic Processor consists of a set of interconnected PEs each capable of performing some simple operation. The basic principle of a systolic array is illustrated in Figure 4. A single PE in a conventional computer is replaced with an array of PEs, to achieve higher computational throughput [KUNG82]. Once a data item is fetched from the memory, it can be used effectively by each PE it passes through. Thus, systolic systems

are suitable when multiple operations are performed on a data item in a repetitive manner [KAIN89].



(a) A Conventional Processor.

(b) A Systolic Array Processor.

Figure 4. The Concept of a Systolic Array Processor.

(Source: T.Y. Kung. "Why Systolic Architectures?", *IEEE Computer*, vol. 15, no.1, p. 38, January 1982.)

2.3.5 Multiprocessors

The term multiprocessor includes virtually all architectures with more than one processor. The system consists of a number of processors, which are connected through some kind of a communication system to a shared memory, a shared I/O system, and possibly to each other [DES87]. Each processor may have its own local memory and also private devices. A single integrated global operating system provides interactions between processors and their programs. This system can be viewed as a system with n processors and m memory units. If all the m memory units form one single global main memory, which can be accessed by all the PEs, then the system is termed a "shared memory" system, otherwise it is a "distributed memory" system [LAK90]. The structure of both these systems is shown in Figure 5. Communication between processors is required in multiprocessors for coordination purposes, which is employed in the form of "message passing" in distributed memory systems and in the form of "shared variables" in shared memory systems [ALMA89]. The term message passing computer is also used for distributed memory systems.

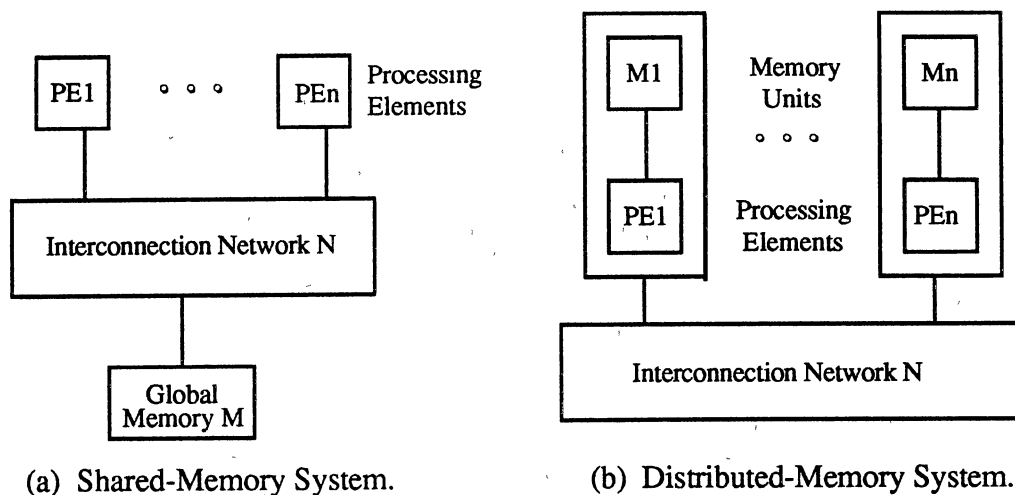


Figure 5. Types of Multiprocessors.

(Source: S. Lakshmivarahan and S.K. Dhall. Analysis and Design of Parallel Algorithms: Arithmetic and Matrix Problems, McGraw-Hill Book Co., New York, NY, p. 6, 1990.)

2.4 Performance Measures

Simple metrics such as clock speed, peak MFLOPS (Millions of Floating Point Operations Per Second) rating, peak MIPS (Millions of Instructions Per Second) rating, memory size and speed, disk size and speed, base system price, price/performance ratio, etc., are available for evaluating parallel processing systems [HWANG89]. A linear combination of these that correspond with the application(s) to be used can help decide to purchase a parallel machine. In practice, the realizable performance from a parallel processing system may be much lower than the peak performance, which could be attributable to the improper match between the parallel algorithm and the architecture [LAK90]. Thus, there are other factors which can help find how effectively the system is being used. Three such measures will be discussed in this section.

The best known measure of the effectiveness of parallel algorithms is the *speed-up ratio* (S_p) [FOX88]. If $T(N)$ is the time required to solve a given problem of size N using the sequential method, and $T_p(N)$ is the time required to solve the same problem using a parallel algorithm with p processors, then speed-up is defined as,

$$S_p = \frac{T(N)}{T_p(N)}$$

Speed-up is normally measured by running the same program on a varying number of processors. Speed-up is greatly influenced by the amount of time the processors spend in communicating with each other. For an application, an approximately linear speed-up with respect to the number of processors is desirable.

Another related measure is *efficiency* E_p [MOIT87] which is the ratio of speed-up S_p to the number of processors p . In other words, E_p is the speed-up achievable per processor. Thus,

$$E_p = \frac{S_p}{p}$$

An efficiency factor close to 1 implies that the resources (the number of processors used for the application) in the system are being used effectively, otherwise they are being under-utilized.

Another important factor is the serial fraction f [KAR90], which is defined as

$$f = \frac{1/s - 1/p}{1 - 1/p}$$

where s is the speed-up on p processors. Serial fraction is used along with speed-up and efficiency to provide useful information on the performance of a system. It is a measure of the rate of change of efficiency. If this rate of change is not linear, then it implies limited parallelism in the application, which can be detected by the serial fraction. Also, this factor can provide information on load imbalances, overhead of synchronization, etc., which cannot be obtained from speed-up and efficiency.

2.5 The iPSC/2 Parallel Computer

2.5.1 Hypercube and the iPSC/2

There are several types of parallel processors in existence like shared memory and

distributed memory, loosely coupled and tightly coupled, packet switching and message passing of data, fine grain and coarse grain, and so on. Among these, one set of choices is the hypercube or the boolean n -cube architecture which is a coarse-grained, MIMD, loosely-coupled, distributed-memory, message-passing, concurrent computer [HWANG89]. The name hypercube originated from the interconnection network used to interconnect its processing elements (PEs) or nodes. There are various types of interconnection network topologies [ALMA89]. The hypercube topology is shown in Figure 6. In this topology the number of nodes is always a power of two (2^n). The value n is called the *dimension* of the hypercube. Each of these nodes is directly connected by fixed communication channels to n other nodes. The nodes in the cube are numbered 0 to $2^n - 1$ and there is an edge between two nodes if their numberings differ by one bit position in their binary representation [HEATH86].

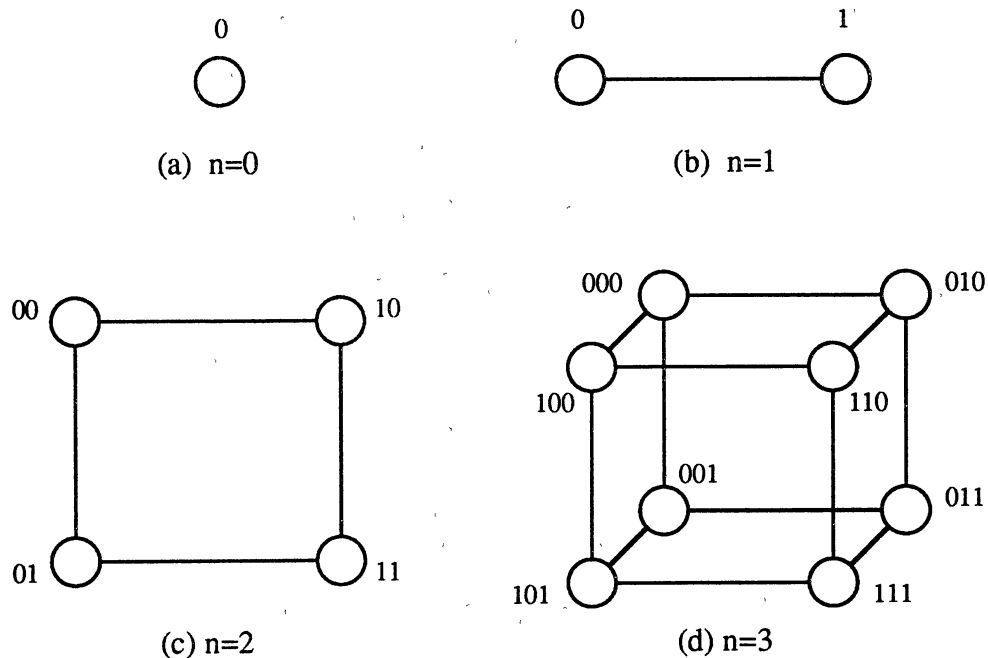


Figure 6. n -dimensional Hypercubes for $n = 0, 1, 2,$ and 3 .

Intel's iPSC/2 Concurrent Supercomputer employs the hypercube topology. An iPSC/2 system consists of compute processors, I/O processors, and a front-end processor.

The front-end processor (generally termed as the iPSC host processor) is called the System Resource Manager (SRM). Each compute processor (generally termed as the iPSC node) is a processor/memory pair, with its own physical memory distinct from that of the host and other nodes. The iPSC computers have support for message passing capabilities so as to communicate with other nodes. I/O processors do not take part in the numerical work of a computation but provide the iPSC/2 system with access to the file system. An iPSC/2 application has a host program that runs on the host processor. A group of iPSC nodes, called a "cube", are allocated for a particular application. A node program runs on this group of allocated nodes. Duties of the host program include initializing the application, providing the necessary human interface, loading the node program on to the nodes, etc. [IPSC89]. Duties of a node program include performing calculations, exchanging messages with other nodes, and sending the data back to the host or other nodes [IPSC89].

2.5.2 iPSC/2 Node Architecture

A block diagram of the iPSC/2 node architecture is depicted in Figure 7. Each of the functional units are discussed in detail in this section.

The Central Processing Unit (CPU) of the iPSC/2 compute node is the Intel's 16 MHz 80386 microprocessor with a rating of 4 MIPS (Million of Instructions Per Second). Like other modern microprocessors, the 80386 also employs pipeline architecture, but unique to the 80386 is the on-chip memory management unit (MMU) which eliminates the serious access delays found in implementations that use off-chip methods.

The iPSC/2 node supports two Numeric Coprocessor options for scalar operations, which reside on the node board itself [CLOSE88]. The first option is the Intel 80387 Numeric Coprocessor which provides floating point, extended integer, and BCD data types. The second option is the Intel's SX Scalar Extension module, which provides two to three times better performance. A third option for vector operations, namely the VP Coprocessor board, can also be attached via the Standard Bus Interface.

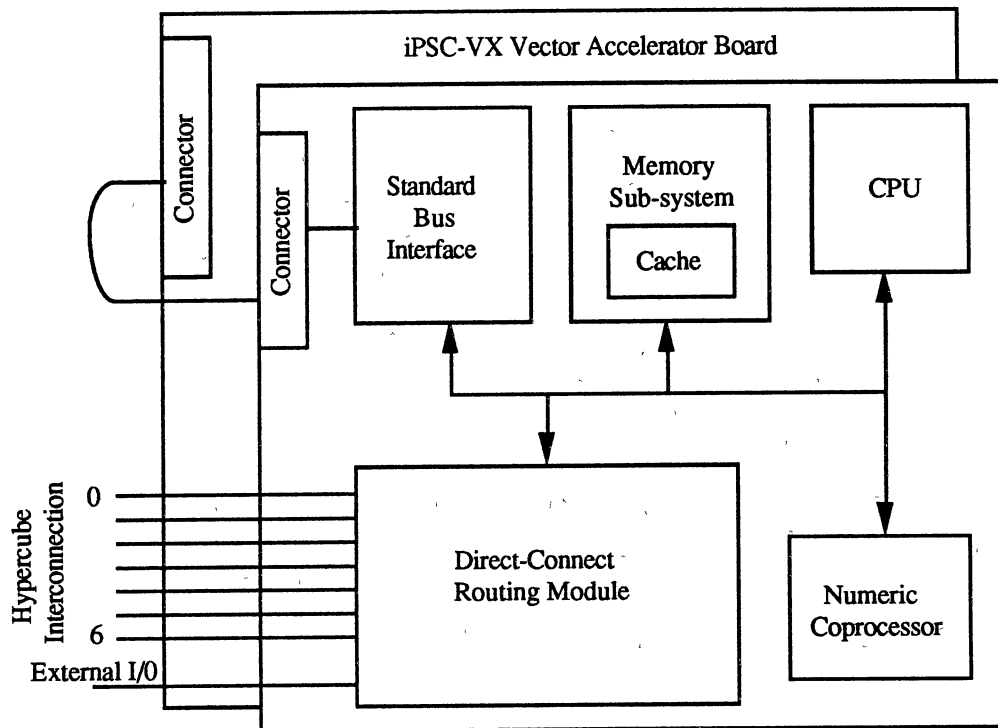


Figure 7. iPSC/2 Node Block Diagram.

(Source: P. Close. "The iPSC/2 Node Architecture", Proc. of the 3rd Conf. on Hypercube Concurrent Computers, and their Applications, p. 44, 1988.)

A Routing Logic Interface called the Direct Connect Module, DCM (which replaces the store-and-forward message passing mechanism used in the original iPSC/1 system) is used in the iPSC/2 system. This module enhances the performance by reducing the message passing latency, increasing the node-to-node channel bandwidth, and allows simultaneous bidirectional message traffic between any two nodes [NUG88]. Routing in this module is based on the e-cube routing algorithm [SULL77], which eliminates deadlock between nodes in the network. Paths (combination of communication channels) between any two nodes are dynamically constructed by this algorithm in a step-by-step process using "routing elements". But the algorithm has a drawback since there is a specific constraint to guarantee deadlock free communication between nodes.

The Memory Subsystem consists of three components: a 16-Megabyte Main Memory, a 64-Kilobyte static RAM cache, and a 64-Kilobyte EPROM containing the boot

loader. Main memory can be in configurations of 1, 4, and 8 Megabyte modules. At most two memory modules can be installed at a time, and hence a maximum of 16 Megabyte configuration with two 8 Megabyte modules can be obtained.

The iPSC/2 also has a Standard Bus Interface tightly coupled to the 386 CPU bus to facilitate the attachment of optional boards of popular buses.

2.5.3 NX/2 Operating System

The NX/2 operating system runs on each node of the Intel iPSC/2 concurrent computer. It provides standard system services such as memory management, multiple process management, message passing capability, intertask protection, and coprocessor support [HWANG89]. There can be up to 20 user processes on each node. All processes have access to 1 Gigabyte of virtual address space (due to 386's paging hardware).

There are two protocols for message passing: a "short messages" (100 bytes or less) 1-trip protocol and a "long messages" (longer than 100 bytes) 3-trip protocol [PIER88]. There are many short message buffers which can be allocated by each node to another node. When a node wants to send a short message to another node and there is a buffer available for it, it simply sends the message. If no buffers are available, it holds the message until buffers are returned by other nodes to the operating system. When a long message is sent by a node, the system initially sends a control message (first trip) to the receiving node. If there is a receive buffer posted for the message on the receiving node, the system sends back a control message (second trip) to the sending node requesting to send the rest of the message (third trip).

The message passing capabilities can be accessed through a nested set of system calls. These calls range from a set of simple synchronous calls, to a set of advanced asynchronous calls that allow overlap of message passing and processing, to interrupt driven message calls [IPSC89]. Synchronous calls include "csend", "crecv", "cprobe" and message "info" calls which block processing till their completion. Asynchronous calls

include "isend", "irecv", "msgwait", "msgdone", and "iprobe" calls, which return as soon as the operation is initiated and do not block. Interrupt driven calls include the "hrecv" and "hsend" calls, which allow more independence between message passing and processing. Moreover, mixing of calls from different levels is permitted. That is, a message can be sent with an asynchronous call and can be received with a simple blocking or interrupt-driven call.

CHAPTER III

FUNDAMENTALS OF LANGUAGE THEORY

Programs written in high-level languages are translated into equivalent machine code programs before they can be executed on a computer. A program which translates a program written in a particular high-level language into an equivalent program in some other language (usually the code for some particular machine) is called a "compiler", "translator", or "interpreter". Compiling a program consists of two stages [FIS88]: an "analysis" stage to recognize the structure and meaning of the program to be compiled (i.e., to determine the intended effect of the program), and a "synthesis" stage to produce the machine or assembly code. In addition, there is an "error correction" stage to detect if the input program is invalid in any sense (i.e., does not belong to the language for which the compiler was written), and if so, return an appropriate message to the programmer.

As far as the compiler is concerned, an initial phase, called *lexical analysis*, normally performs the task of grouping characters together into what are usually referred to as "tokens" (e.g., print, begin, end, read, identifiers, etc.) [AHO86]. A lexical analyzer is sometimes called a scanner. A programming language can be thought of as consisting of a number of strings (sequences of symbols). The definition of a language specifies which strings belong to the language ("syntax" of the language) and the meaning of these strings ("semantics" of the language).

3.1 Preliminaries

The subject of "formal language theory" provides a definition of a most universal language structure by specifying precise rules. The word "formal" refers to the fact that all

the rules for the language are explicitly stated in terms of what strings of symbols can occur. With this general understanding we can now state some abstract definitions.

A *symbol* is loosely defined as any representable character. The terms letter, character, and symbol are used interchangeably.

An *alphabet*, denoted by Σ , is any set of symbols. An alphabet will be considered a finite set for all practical purposes. Examples of alphabets include the set of 26 uppercase and 26 lowercase roman letters called the roman alphabet, the set of numbers 0,1,2,...,9 called the decimal alphabet, and the set $\{0,1\}$ consisting of only 0 and 1 called the binary alphabet.

A *string* is a sequence of symbols juxtaposed or put side by side. The terms string, word, and sentence are used interchangeably. Examples of strings are 0011 over the binary alphabet, and *abba* over $\{a,b\}$. There exists a special string which is allowed to have no symbols. This string, called the "empty string" or "null string", is denoted by ϵ . ϵ belongs to any language.

A *language* is always defined over an alphabet Σ . A language over Σ is defined as a set of strings obtained from Σ . Some examples of languages are

- the empty set ϕ ,
- the set $\{\epsilon\}$ containing only the empty string, and
- the set $\{1^i 0 \mid i \geq 0\}$ consisting of all strings of zero or more 1's followed by a 0.

There is a special language in which any sequence of symbols from an alphabet Σ is a valid string including the null string. This is denoted by Σ^* . For the binary alphabet $\{0,1\}$,

$$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$$

Note that every language over Σ is a subset of Σ^* . Thus Σ^* is considered the universal language.

3.2 Regular Expressions

The process of forming tokens is often driven by token descriptions. "Regular Expression Notation" is a formalism used to describe the various tokens required by

programming languages. According to Fischer and Leblanc, Jr. [FIS88], "the interpretation of regular expressions is the basis of *scanner generators*, programs that actually produce a working scanner given only a specification of the tokens they are to recognize". Such a program will be a valuable compiler-building tool. The sets of strings defined by regular expressions are termed "regular sets", which are a class of languages central to much of language theory.

One of the operations defined on regular expressions is the closure operation. Closure of a language L , denoted by L^* , is the set of strings formed by concatenating any number of strings from L . Formally, L^* is defined as,

$$L^* = \sum_{i=0}^{\infty} L^i$$

The other operations are union and concatenation and they apply the same way as in sets. When interpreting a regular expression that contains several operators, the closure operator has the highest precedence followed by concatenation and union operators.

Let Σ be an alphabet. The regular expressions over Σ and the sets they denote are defined as follows [HOP79]:

- ϕ is a regular expression and denotes the empty set.
- ϵ is a regular expression and denotes the set $\{\epsilon\}$.
- every symbol, r in the alphabet is a regular expression.
- if r and s are two regular expressions, then their union ($r+s$), their concatenation (rs), and closure (r^*) are regular expressions.

Some examples of regular expressions are 00 , $0+1$, 0^* (strings consisting of any number of 0's), 0^*1^*0 (strings consisting of any number of 0's, followed by any number of 1's, and ending with a 0).

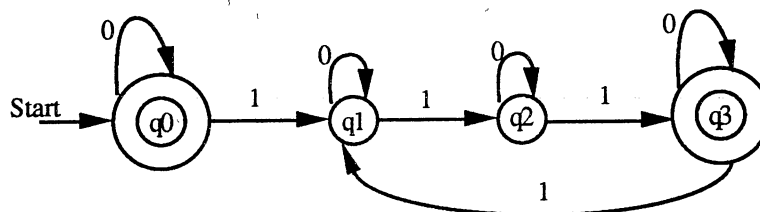
3.3 Finite Automata

Before writing a compiler, it is necessary to have a clear and unambiguous definition of the particular source language. There are two methods of defining languages

[AHO72]: a *generator*, which uses a generative system called a "grammar", and a *recognizer*, which is a highly stylized procedure capable of deciding whether a string belongs to the language or not. A finite automaton (FA) is one of the simplest recognizers. It is a device for recognizing strings of a particular language, in other words, it can be used to recognize the tokens specified by a programming language. Conceptually, a finite automaton is a mathematical model of a system with inputs and outputs. The system can be in any one of a finite number of internal configurations or "states", some of which are termed "final" states. There is a control mechanism which passes control from state to state as each character of the string is read, according to a given set of transitions (or rules). Thus, a finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ [AHO72] where

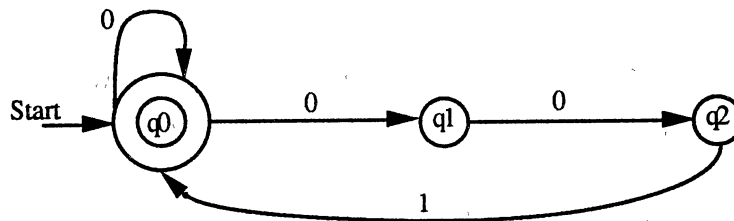
- Q is a finite set of states
- Σ is a finite set of input symbols
- δ is a function which, given a possible combination of the current state and input, takes the automaton to a new state (possibly back to the same state)
- q_0 is the start state
- F is the set of final states

According to Hopcroft and Ullman [HOP79], "every FA is associated with a directed graph, called a *transition graph*, where the vertices correspond to the states of the FA and arcs correspond to the transition(s) that each state can make on an input symbol". If on input a , the FA moves from a state p to a state q , then there is an arc labeled a from vertex p to vertex q in the graph. An example transition graph which accepts all strings of 0's and 1's in which the number of 1's is a multiple of three is given below. The initial



state q_0 , is indicated by the arrow labeled "start". The final states q_0 and q_3 , are indicated by the double circle. The start state and one of the final states can coincide. If all the

transitions from one state lead to at most one single state on an input symbol, the machine is called a Deterministic Finite Automaton (DFA), otherwise it is called a Nondeterministic Finite Automaton (NFA). The automaton shown above is an example of a DFA. An example of an NFA is shown below. There can also be special moves in an NFA called ϵ -moves, which means that the finite automaton makes a transition from state to state on an empty string ϵ .



Now the recognizing power of a finite automaton is illustrated. If after reading the final character of a string, the finite automaton is in one of the final states then the string is "accepted" by the automaton otherwise it is "rejected". A way of representing the transition function δ is by using a 2-dimensional table, as shown below, with rows representing the states and columns representing the input symbols. Each entry in the table is the next state

	Inputs	
States	0	1
q0	q0	q1
q1	q1	q2
q2	q2	q3
q3	q3	q1

q0 - start state
q0,q3 - final states.

that the automaton will move to, on the input symbol given in the column entry, from the original state given in the row entry. From the transition table, the following transitions can be observed.

$$\begin{aligned}\delta(q_0, 1) &= q_1 \\ \delta(q_1, 1) &= q_2\end{aligned}$$

Hence,

$$\begin{aligned}\delta(q_0, 11) &= \delta(\delta(q_0, 1), 1) \\ &= \delta(q_1, 1) \\ &= q_2\end{aligned}$$

Similarly,

$$\delta(q_0, 101001) = q_3$$

which is a final state, and hence the string 101001 is accepted by the DFA.

3.4 Transformation Algorithms

As mentioned earlier, a set of transformations can be performed on a regular expression representing a particular language. A cycle of transformations that are used in this thesis are shown in Figure 8. In the following subsections, an example RE 0^*1^* is

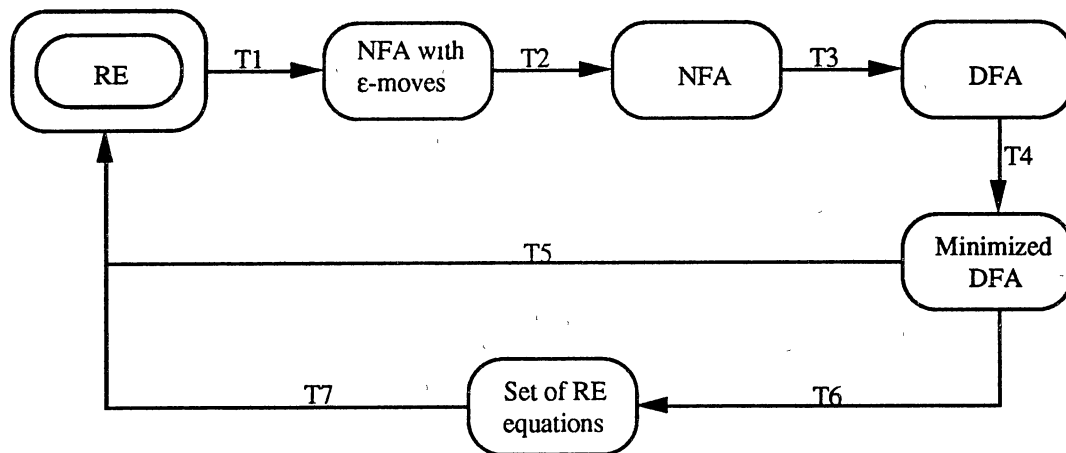


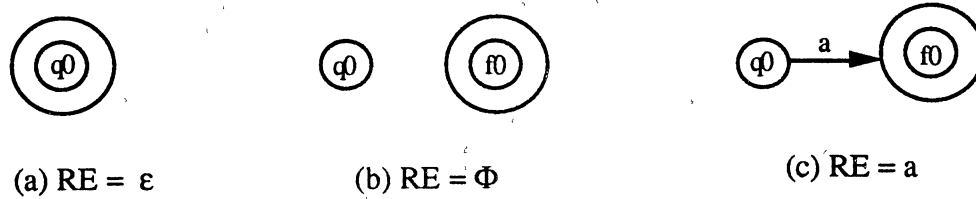
Figure 8. Cycle of Transformations Performed on a Regular Expression.

considered and the various changes occurring during this cycle of transformations are explained in detail. Appendix C gives these details for another example.

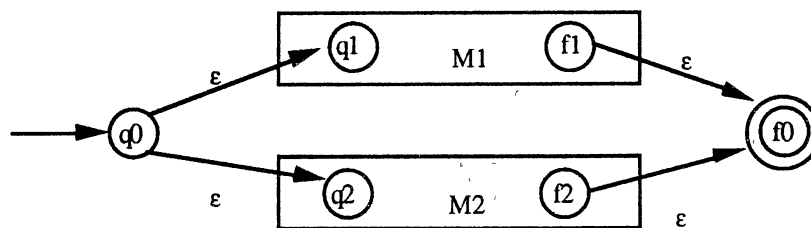
3.4.1 Transformation T1 - RE to NFA

Converting a regular expression into an equivalent finite automaton is the first transformation performed in the cycle. This is the synthesis step as an NFA representing the given regular expression is constructed. The basis for this step is that there is a simple

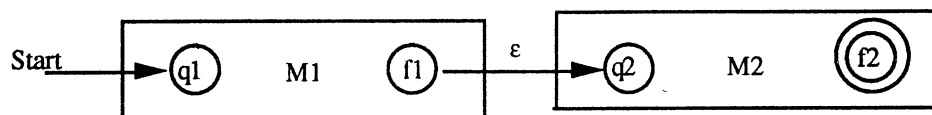
finite automaton for atomic regular expressions, (i.e., regular expressions without any operators) as shown below.



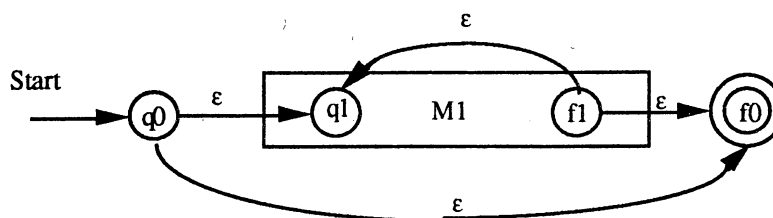
A regular expression, r can be written in the form of (r_1+r_2) or $(r_1.r_2)$ or (r_1^*) where r_1 and r_2 are regular expressions too. Let M_1 and M_2 represent the finite automata for regular expressions r_1 and r_2 , respectively. Then the finite automaton for the three forms of an RE can be constructed using the rules shown in Figure 9. Applying this



(a) Union of M_1 and M_2



(b) Concatenation of M_1 and M_2



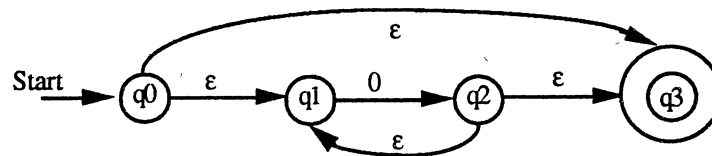
(b) Closure of M_1

Figure 9. Rules for Synthesizing an NFA from Automata M_1 and M_2 .

(Source: J.E. Hopcroft and J.D. Ullman. Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, p. 31, 1979.)

construction technique iteratively yields the automaton for a given regular expression.

The method is illustrated through an example. Consider the regular expression 0^*1^* . The regular expression in complete parenthesized form (considering operator precedence) is $((0)^*(1)^*)$. Note that concatenation is now denoted by the symbol "." instead of juxtaposing. This is in the form of $r_1.r_2$ where $r_1 = (0)^*$ and $r_2 = (1)^*$. Again r_1 is in the form of r_3^* where $r_3 = 0$. The finite automaton for r_3 can be obtained directly from the basis step. Then we use the construction step of Figure 9(c) to obtain the finite automaton for 0^* as follows.



Similarly, the finite automaton for $(1)^*$ can be constructed. Now we use the construction step of Figure 9(b) on the machines M_1 and M_2 to obtain the automaton for 0^*1^* as shown in Figure 10. Note that the automaton obtained by this method is an NFA with some ϵ -moves.

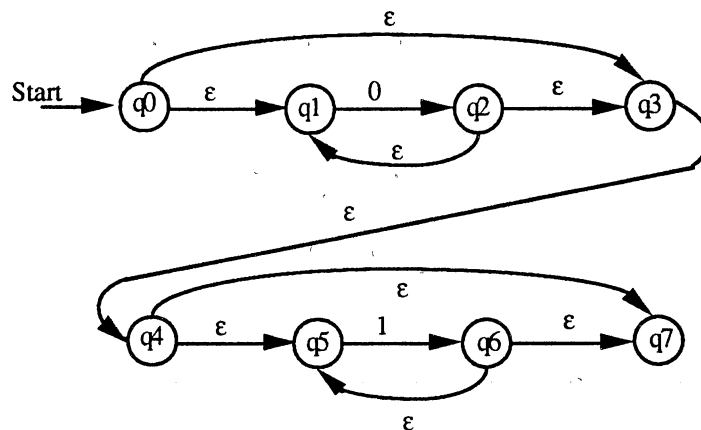
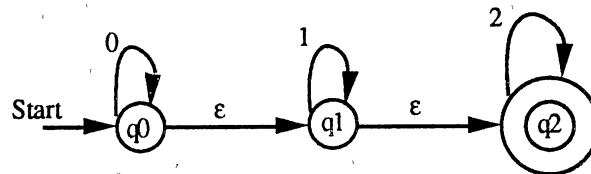


Figure 10. NFA With ϵ -moves for the RE 0^*1^* .

3.4.2 Transformation T2 - Removing ϵ -moves

This transformation removes the ϵ -moves from the NFA obtained in the previous step. The method is illustrated through a simple example. We will come back to the example of 0^*1^* a little later in this section. Consider the NFA with ϵ -moves given below, which accepts the language consisting of any number (including zero) of 0's, followed by any number of 1's followed by any number of . Note that arcs labeled ϵ may be included in any path. Thus the string 012 is accepted by the NFA by following the path with arcs labeled 0, ϵ , 1, ϵ , 2.



ϵ -closure(q) of a state q is the set of all states p such that there is a path from q to p with one or more arcs labeled ϵ . Let us find the ϵ -closure(q_0) for the NFA shown above. There is always a path from q_0 to q_0 with an arc labeled ϵ . There is a path from q_0 to q_1 with an arc labeled ϵ . Also, there is a path from q_0 to q_2 with arcs labeled ϵ, ϵ . Hence ϵ -closure(q_0) = $\{q_0, q_1, q_2\}$. Similarly ϵ -closure(q_1) = $\{q_1, q_2\}$, and ϵ -closure(q_2) = $\{q_2\}$.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be the NFA with ϵ -moves. Then the new NFA M' is constructed by following the algorithm [HOP79] in Figure 11. Let us apply these rules to the NFA considered above. The new machine is given by $M' = (\{q_0, q_1, q_2\}, \{0, 1, 2\}, \delta', q_0, F')$ where

$$\begin{aligned} F' &= F \cup \{q_0\} \text{ as } \epsilon\text{-closure}(q_0) \text{ contains the final state } q_2 \\ &= \{q_0, q_2\} \end{aligned}$$

Then we determine the new transition function, δ' as follows.

$$\begin{aligned} \delta'(q_0, 0) &= \epsilon\text{-closure}(P), \text{ where} \\ P &= \delta(\delta'(q_0, \epsilon), 0) \\ &= \delta(\epsilon\text{-closure}(q_0), 0) \end{aligned}$$

let the NFA with ϵ -moves be $M = \{Q, \Sigma, \delta, q_0, F\}$;

the NFA without ϵ -moves is given by

$$M' = (Q, \Sigma, \delta', q_0, F');$$

where

$$\delta'(q, \epsilon) = \epsilon\text{-closure}(q);$$

if $\epsilon\text{-closure}(q_0)$ contains a state of F then

$$F' = F \cup \{q_0\};$$

else $F' = F$;

for any string w in Σ^* and any symbol a in Σ

$$\delta'(q, wa) = \epsilon\text{-closure}(P) \text{ where,}$$

$$P = \{p \mid \text{for some } r \text{ in } \delta'(q, w), p \text{ is in } \delta(r, a)\};$$

Figure 11. Algorithm for Removing ϵ -moves in an NFA.

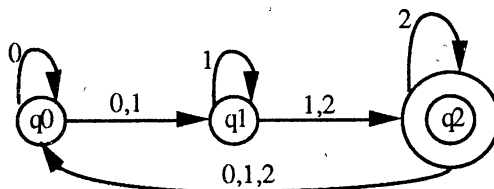
(Source: J.E. Hopcroft and J.D. Ullman. Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, p. 26, 1979.)

$$\begin{aligned} &= \delta(\{q_0, q_1, q_2\}, 0) \\ &= \delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0) \\ &= \{q_0\} \cup \Phi \cup \Phi \\ &= \{q_0\} \end{aligned}$$

Thus,

$$\begin{aligned} \delta'(q_0, 0) &= \epsilon\text{-closure}(q_0) \\ &= \{q_0, q_1, q_2\} \end{aligned}$$

Following this approach, the transition function of M' is determined. Thus the NFA after removing ϵ -moves is shown below.



Using this method on the example of 0^*1^* , whose NFA with ϵ -moves was shown in Figure 10 of section 3.4.1, we obtain the NFA without ϵ -moves as shown in Figure 12.

		0	1
S,F	q0	{q1,q2,q3,q4,q5,q7}	{q5,q6,q7}
	q1	{q1,q2,q3,q4,q5,q7}	Φ
	q2	{q1,q2,q3,q4,q5,q7}	{q5,q6,q7}
	q3	Φ	{q5,q6,q7}
	q4	Φ	{q5,q6,q7}
	q5	Φ	{q5,q6,q7}
	q6	Φ	{q5,q6,q7}
	F	q7	Φ

Figure 12. NFA Without ϵ -moves for the RE 0^*1^* .

3.4.3 Transformation T3 - NFA to DFA

This transformation removes the nondeterminism from the automaton obtained in the previous transformation. That is, a DFA equivalent to the given NFA will be constructed. Let $M = (Q, \Sigma, \delta, q_0, F)$ be the given NFA. Then we can construct the DFA M' by the algorithm [AHO72] presented in Figure 13.

let the NFA be given by $M = (Q, \Sigma, \delta, q_0, F)$;
 the equivalent DFA is given by $M' = (Q', \Sigma, \delta', q_0', F')$ where

Q' is the power set of Q ;
 /* the states of M' are sets of states of M */
 $q_0' = \{q_0\}$;
 F' consists of subsets, S of Q such that $S \cap F \neq \Phi$;
 for all subsets S , and any symbol a in Σ , $\delta'(S, a) = S'$ where
 $S' = \{p \mid \delta(q, a) \text{ contains } p \text{ for some } q \text{ in } S\}$;

Figure 13. Algorithm to Construct a DFA from an NFA.

(Source: A.V. Aho and J.D. Ullman. The Theory of Parsing, Translation, and Compiling, Prentice-Hall, Englewood Cliffs, NJ, p. 117, 1972.)

Consider the NFA M obtained in transformation T2 for the RE 0^*1^* as shown in Figure 12. Since M has 8 states it appears that the DFA M' to be constructed will have

$2^8=256$ states. But, not all of the 256 states will be accessible from the initial state $\{q_0\}$, and hence M' may not contain all the 256 states. Since $\{q_0\}$, the start state, is always accessible the construction begins from this state.

$$\begin{aligned}\delta'(\{q_0\},0) &= \delta(q_0,0) = \{q_1,q_2,q_3,q_4,q_5,q_7\} \text{ and} \\ \delta'(\{q_0\},1) &= \delta(q_0,1) = \{q_5,q_6,q_7\}.\end{aligned}$$

Let $Q_1 = \{q_1,q_2,q_3,q_4,q_5,q_7\}$ and $Q_2 = \{q_5,q_6,q_7\}$, and consider these as the new states obtained. Then,

$$\begin{aligned}\delta'(Q_1,0) &= \delta'(\{q_1,q_2,q_3,q_4,q_5,q_7\},0) \\ &= \delta(q_1,0) \cup \delta(q_2,0) \cup \delta(q_3,0) \cup \delta(q_4,0) \cup \delta(q_5,0) \cup \delta(q_7,0) \\ &= \{q_1,q_2,q_3,q_4,q_5,q_7\} \\ &= Q_1 \\ \delta'(Q_1,1) &= \delta(\{q_1,q_2,q_3,q_4,q_5,q_7\},1) \\ &= \{q_5,q_6,q_7\} \\ &= Q_2\end{aligned}$$

Following this procedure, the new transition function is completely determined when no more new states are encountered. In this example, we see that Q_1 and Q_2 are the only new states obtained. Since Q_1 and Q_2 both contain state q_7 which is a final state of M , both Q_1 and Q_2 are termed as final states of M' along with q_0 . Thus the DFA equivalent to the given NFA is shown in Figure 14.

		0	1
S,F	$Q_0=[q_0]$	Q1	Q2
F	$Q_1=[q_1,q_2,q_3,q_4,q_5,q_7]$	Q1	Q2
F	$Q_2=[q_5,q_6,q_7]$	ϕ	Q2

Figure 14. DFA Equivalent to the NFA for the RE 0^*1^* .

3.4.4 Transformation T4 - Minimizing the DFA

The DFA constructed in transformation T3 may have redundant and inaccessible states which are removed in this transformation. Consider the DFA shown in Figure 14.


```

for  $p$  in  $F$  and  $q$  in  $Q-F$  do
    mark  $(p,q)$  entry in the table;

for each pair of distinct states in  $F \times F$  or  $(Q-F) \times (Q-F)$  do
begin
    if for some input symbol  $a$ ,  $(\delta(p,a),\delta(q,a))$  is marked then
    begin
        mark  $(p,q)$  entry in the table;
        recursively mark all unmarked entries on the list for  $(p,q)$  and
        on the lists of other entries that are marked at this step;
    end; /* then */

    else begin /* no pair  $(\delta(p,a),\delta(q,a))$  is marked */
        for all input symbols  $a$  do
            if  $\delta(p,a) \neq \delta(q,a)$  then
                put  $(p,q)$  on the list for  $(\delta(p,a),\delta(q,a))$ ;
            end; /* end else */
    end /* do */

```

Figure 15. Algorithm for Marking Pairs of Inequivalent States in a DFA.

(Source: J.E. Hopcroft and J.D. Ullman. Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, p. 70, 1979.)

The algorithm for minimizing a DFA [HOP79], outlined in Figure 15, will find the set of states which are equivalent. A table is constructed with an entry for each pair of states as shown below. Each entry corresponding to one final state and one nonfinal state, that is the entries (q_0,q_3) , (q_1,q_3) , and (q_2,q_3) , are marked with an X. For each entry (p,q) that is not yet marked in the table, we consider the pair of states $r=\delta(p,a)$ and $s=\delta(q,a)$ for each input symbol a . If the entry (r,s) is marked for some input symbol a , then the entry (p,q) gets marked. If the (r,s) entry does not get marked for all inputs, then the pair (p,q) is placed on a list associated with the (r,s) entry. If (r,s) entry gets marked in further steps, then each pair on the list associated with the (r,s) entry also gets marked.

In this example, to mark the entry (q_1,q_0) we see that $\delta(q_1,0) = \delta(q_0,0)$ and also $\delta(q_1,1) = \delta(q_0,1)$, that is q_0 and q_1 states go to the same state on both input symbols 0 and 1. Hence (q_0,q_1) entry can neither be marked nor can it be placed on any associated list.

q1			
q2	X	X	
F	X	X	X
	q0	q1	q2

For the entry $(q2, q0)$ we see that $(\delta(q2, 0), \delta(q0, 0)) = (q3, q1)$ has already been marked and hence $(q2, q0)$ entry gets marked. Continuing with these steps we complete marking the table. From the table we see that only $(q0, q1)$ entry is not marked, and hence states $q0$ and $q1$ are considered equivalent. They are merged into a new state $Q1$, and the other state $q2$ is retained as is in the minimized DFA. Moreover, $\delta(Q1, a) = \delta(q0, a) \cup \delta(q1, a)$ for any input a . Now, the minimum state DFA can be easily constructed as shown in Figure 16.

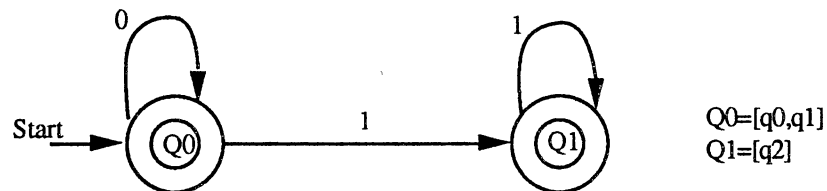


Figure 16. The Minimized DFA for the RE 0^*1^* .

3.4.5 Transformation T5 - DFA to RE

In this transformation the transition graph of a finite automaton is extended to have arcs labeled by regular expressions. The algorithm, shown in Figure 17, builds a regular expression for the set of strings accepted by each individual final state [SUD88]. The language accepted by the DFA is then given by the union of these regular expressions. Let the arc from state q_i to state q_j be denoted by w_{ij} ($w_{ij}=\phi$ if there is no such arc). A set of subgraphs are constructed from the original transition graph G of the DFA such that each subgraph has exactly one final state of G . Each subgraph is processed by deleting particular states as shown in Figure 18. To delete a state q_i , all paths of length two that have

make m copies (G_1, G_2, \dots, G_m) of the transition graph G such that each subgraph has one unique final state of G .

for $t=1$ to m **do**

begin

repeat

a state q_i is chosen in G_t that is neither the start nor the final state;

/ delete this state as follows */*

for every pair of states q_j, q_k not equal to q_i (including $q_j=q_k$) **do**

begin

if $w_{ji} \neq \phi$ and $w_{ik} \neq \phi$ and $w_{ii} = \phi$ **then**

an arc is added from state q_j to state q_k labeled $w_{ji}w_{ik}$;

if $w_{ji} \neq \phi$ and $w_{ik} \neq \phi$ and $w_{ii} \neq \phi$ **then**

an arc is added from state q_j to state q_k labeled $w_{ji}(w_{ii})^*w_{ik}$;

if states q_j, q_k have arcs w_1, w_2, \dots, w_r connecting them **then**

they are replaced by a single arc labeled $w_1 \cup w_2 \cup \dots \cup w_r$;

state q_i and all arcs incident to it are removed from G_t ;

end; */* do */*

until G_t has only the start and final states;

the regular expression RE_t for G_t is determined;

the regular expression for the graph G is accumulated as

$REG = REG + RE_t$;

end; */* do */*

Figure 17. Algorithm to Build an RE Representing an FA.

(Source: T.A. Sudkamp. Languages and Machines: An Introduction to the Theory of Computer Science, Addison-Wesley, Reading, MA, p. 160, 1988.)

state q_i in between state q_j and state q_k are found. Then an arc is added directly from state q_j to state q_k , and labeled $w_{ji}w_{ik}$ if $w_{ii} = \phi$, else labeled $w_{ji}(w_{ii})^*w_{ik}$.

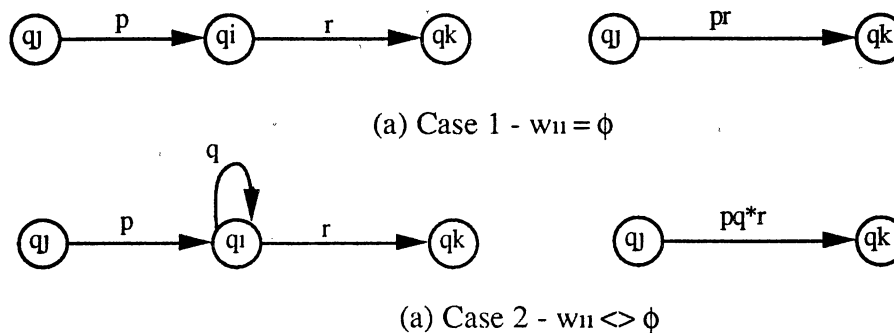
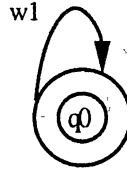
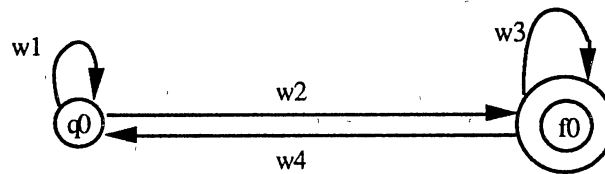


Figure 18. Two Cases in Deleting a State q_i .

After deleting all possible states, the reduced graph has at most two states, a start state and a final state. This graph may have the following form, where the start and final

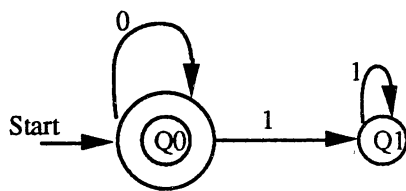


states coincide. The regular expression for this graph is w_1^* . On the other hand, the graph may have the following form.

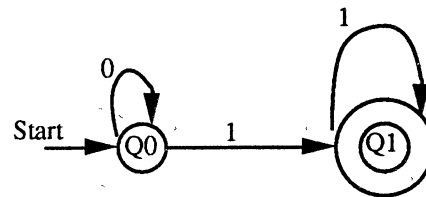


The regular expression for this graph is $w_1^*w_2(w_3 \cup w_4w_1^*w_2)^*$ which may be simplified if any of the arcs in the graph is missing.

Now, consider the minimized DFA shown in Figure 16 of section 3.4.4 for the example RE 0^*1^* . Since this graph has two states, two subgraphs are constructed as:

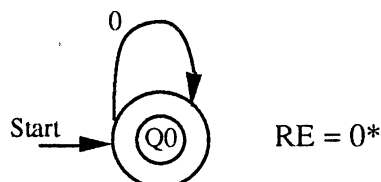


Subgraph G1



Subgraph G2

State Q1 in G1 can be easily deleted since it does not fit into any of the two cases shown in Figure 18. This leaves the reduced version of G1 as follows.

RE = 0^*

G2 cannot be reduced any further since it has exactly one start state and one final state. The RE for G2 is 0^*11^* . Then, the RE for the original graph G is obtained by the union of the REs for G1 and G2 as $0^* + 0^*11^*$.

3.4.6 Transformation T6 - RE equations for the DFA

This transformation determines the set of RE equations which represents a finite automaton. Let m be the number of states in the given automaton. Let a_{ij} denote the set of input symbols such that $\delta(q_i, a_{ij}) = q_j$ for $i=1, \dots, m$ and $j=1, \dots, m$. If there are no such transitions, then $a_{ij} = \epsilon$, since any state can be reached from itself by an arc labeled ϵ . Let x_k represent the RE for the state q_k . Then the following equations can be written for an automaton [ARD60].

$$\begin{aligned}x_1 &= x_1 a_{11} + x_2 a_{21} + \dots + x_m a_{m1} + \epsilon \\x_2 &= x_1 a_{12} + x_2 a_{22} + \dots + x_m a_{m2} \\x_m &= x_1 a_{1m} + x_2 a_{2m} + \dots + x_m a_{mm}\end{aligned}$$

Note that the equation for the start state x_1 has an ϵ added at the end. For the DFA shown in Figure 16 of transformation T4, we can write the set of RE equations as follows.

$$\begin{array}{lcl}X_0 = X_0 0 + \epsilon & \dots & \text{I} \\X_0 = X_0 0 + X_1 1 & \dots & \text{II}\end{array}$$

where X_0 and X_1 represent the expressions for the states Q0 and Q1, respectively.

3.4.7 Transformation T7 - Solution of RE Equations

This method is similar to that of solving a set of linear equations using Gaussian elimination. The algorithm for solving a set of n regular expression equations [AHO72] is presented in Figure 19. The algorithm relies on the fact that there is a simple solution to the equation $X = Xa + b$ where a and b are regular expressions. One of the solutions for this equation is $X = ba^*$.

The method is illustrated through an example. Consider the following set of RE equations.

```

i=1;
while i ≤ n do
begin
  write equation for Xi as Xi = Xi + b where
  a is an RE and b = b0 + Xi+1bi + ... + Xnbn (each bi is an RE)
  for r = i+1 to n do
    In the equation for Xr replace Xi by the RE ba*
  i = i + 1;
end; /* do */

/* At this point, the equation for Xi will have only symbols in Σ and
Xi,...,Xn on the RHS. Thus the equation for Xn will have only Xn and
symbols in Σ on the RHS. */

i = n;
while i ≥ n do
begin
  Solve for Xi = ba*;
  Substitute ba* for Xi in the remaining equations;
  i = i - 1;
end; /* do */

```

Figure 19. Algorithm for Solving a Set of RE Equations.

(Source: A.V. Aho and J.D. Ullman. The Theory of Parsing, Translation, and Compiling. Prentice-Hall, Englewood Cliffs, NJ, p. 106, 1972.)

$$X_1 = X_10 + X_21 + \epsilon \quad \dots \quad (1)$$

$$X_2 = X_20 + X_31 \quad \dots \quad (2)$$

$$X_3 = X_31 + X_21 \quad \dots \quad (3)$$

Equation (1) can be written as

$$X_1 = X_10 + (X_21 + \epsilon)$$

which can be solved as

$$X_1 = (X_21 + \epsilon)0^* \quad \dots \quad (4)$$

Equation (4) can be substituted in Equation (3) to obtain

$$\begin{aligned} X_3 &= (X_21 + \epsilon)0^*1 + X_21 \\ &= X_2(10^*1 + 1) + 0^*1 \quad \dots \end{aligned} \quad (5)$$

Now Equation (2) can be solved as

$$X_2 = (X_31)0^*$$

$$= X_3 10^* \quad \dots \quad (6)$$

Substituting Equation (6) in Equation (5) we obtain

$$\begin{aligned} X_3 &= X_3 10^*(10^*1+1) + 0^*1 \\ &= X_3(10^*10^*1+10^*1) + 0^*1 \quad \dots \quad (7) \end{aligned}$$

We have now reached the last step of the algorithm. We solve for X_3 to obtain,

$$X_3 = 0^*1(10^*10^*1+10^*1)^* \quad \dots \quad (8)$$

X_3 can now be back substituted in Equation (6) to get

$$X_2 = 0^*1(10^*10^*1+10^*1)^*10^* \quad \dots \quad (9)$$

Then X_2 can be substituted in Equation (4) to obtain the solution for X_1 .

Following this procedure, we can obtain the solution for equations I and II obtained in transformation T6 for the DFA in Figure 16, as follows.

$$\begin{aligned} X_0 &= 0^* \\ X_1 &= 0^*11^* \end{aligned}$$

Since both Q_0 and Q_1 are final states, the regular expression for the DFA is obtained by the union of X_0 and X_1 . Thus the final RE for the DFA is given by $0^* + 0^*11^*$.

CHAPTER IV

MULTIPROCESSOR SCHEDULING

As mentioned earlier, parallel processing provides a possible solution in meeting the increasing demand for computational speed in solving very complex problems which would not be possible on sequential computers. Interestingly enough, many physical problems show some sort of inherent parallelism which enables them to be modeled by parallel systems. This factor makes it possible for solving many complex problems on parallel systems. Extracting this inherent parallelism effectively from a given problem leads to the solution of the problem, which could be subsequently implemented on a parallel system. Initially, a given problem is *decomposed* or *partitioned* by identifying the sequential units of computation, called "tasks", in the problem and establishing the interdependencies among them [FOX88]. Subsequently, the tasks are assigned to a set of available processors following a *scheduling* procedure. Partitioning and scheduling are two important multiprocessor-dependent issues in implementing parallel algorithms [POLY86]. In this chapter, we will discuss the partitioning and scheduling approaches used in the implementation of parallel algorithms for the transformations described in section 3.4.

4.1 A Partitioning Approach

The first step in the implementation of parallel algorithms is partitioning the problem. As mentioned above, partitioning a problem consists of identifying the sequential units of computation, called "tasks", in the given problem. The partitioning approach adopted must ensure to make the granularity of the parallel algorithm coarse enough for the

target multiprocessor [SARK89]. The parallelism in the problem is usually specified by the way tasks depend on each other in the partitioned problem. These dependencies should be kept at a minimum by the approach used for partitioning. In this section we will discuss the partitioning approach used in the thesis.

The first transformation T1, in the cycle of transformations shown in Figure 8 of Section 3.4, is to synthesize a finite automaton from a regular expression. The construction method is described in section 3.4.1. It can be easily seen that this process is similar to evaluating an arithmetic expression with various operators and operands. Several methods, both sequential and parallel, for evaluating an expression are available in the literature [AHO72, AHO86, MOIT87, RAM71]. The approach adopted in the thesis follows the general concept of evaluating an expression in Reverse Polish Notation [SOR76].

Initially, a regular expression is converted into a form which consists of only binary operators and the two operands. This requires adding a dummy operand (say 0 or 1) to the right of every closure operator in the RE. For example, 0^* becomes 0^*0 . But this dummy operand will not be used in the evaluation of the RE. Also, the concatenation operation, which is generally represented by juxtaposing its two operands, is now represented by the operator "." between the two operands in the RE. Thus, 00 is represented as 0.0 in the RE. Using these transformations, a given RE, for example,

$$(0^*1+11)1 + (00+0^*1)0 + 0^*1$$

is now represented as,

$$(0^*0.1+1.1).1 + (0.0 + 0^*0.1).0 + 0^*0.1$$

Then this expression is converted to post-fix notation by using the Polish algorithm [SOR76] as follows.

$$00^*1.11.+1.00.00^*1.+0.+00^*1.+$$

Identifying the individual tasks in the RE represented in this form is now straightforward. Each binary operation (the two operands and the binary operator succeeding them) consists

of a single task. Identification of tasks is completed when the last task in the RE is identified. The tasks are identified for the given RE as shown in Figure 20.

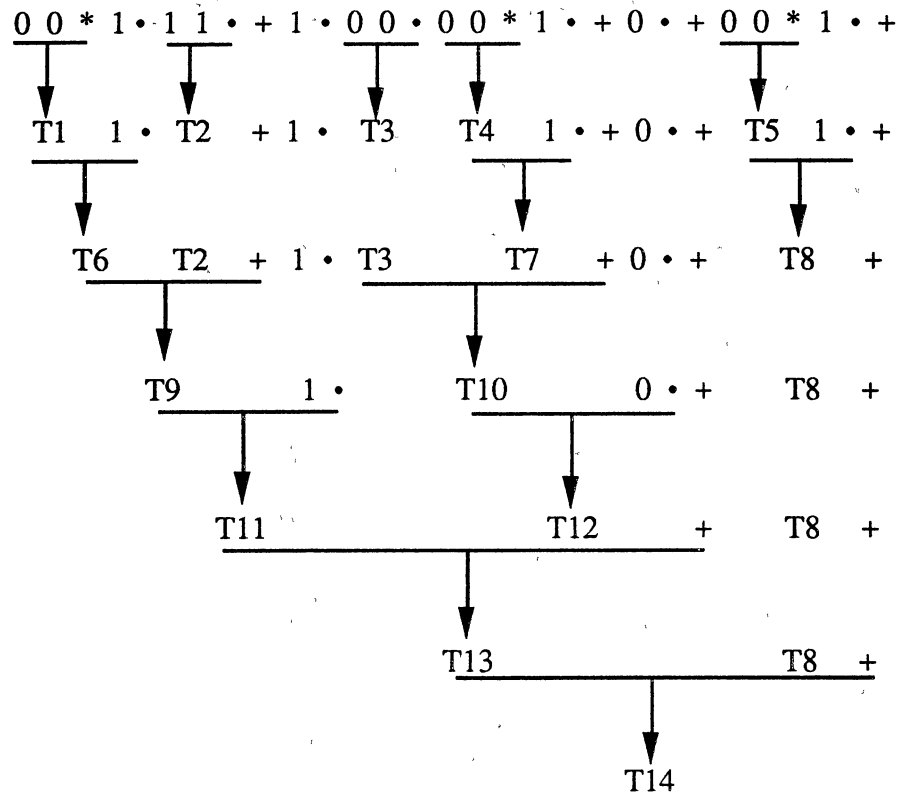


Figure 20. Partitioning an RE into Tasks.

One can directly see that tasks T_1 , T_2 , T_3 , T_4 , and T_5 do not depend on any other tasks (they only need 0 or 1 as their operands), and are thus independent. But task T_6 depends on task T_1 since it needs the result of task T_1 as one of its input. In a similar fashion the other dependencies can be established from the partitioned problem.

4.2 Graphical Representation of the Problem

A problem partitioned into several tasks forms a task system, which can be represented in a form that shows the relationships among the tasks. Various representation

forms for a task system are available [AHO86, GURD85], and two simple forms are discussed in this section.

4.2.1 Precedence Graphs

In general, a task system can be modeled by a precedence graph [COFF76], which is a directed acyclic graph (DAG), G consisting of a set of nodes and a set of edges (directed arcs). An example of a precedence graph is shown in Figure 21. Each node n_i in G represents a task T_i and each arc between nodes n_i and n_j indicates that a precedence relation exists between the tasks, T_i and T_j , on these nodes. This precedence relation specifies which one of the tasks, T_i or T_j , needs to be initiated before the other one. The "predecessors" of a task T_i are the nodes from which arcs are incident to T_i , and the "successors" of T_i are the nodes to which arcs are incident from T_i . Nodes with no predecessors are called *leaf nodes* which represent the initial tasks in the system. A task can be executed or assigned when all of its predecessors have been executed or assigned. Each node is associated with an attribute called *weight* of the node which is the execution time of the task represented by that node (node weight and node execution time are used interchangeably).

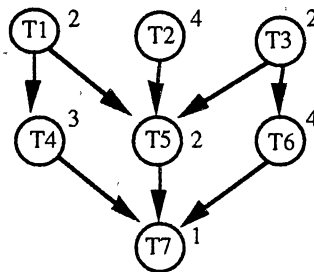


Figure 21. An Example of a Precedence Graph.

4.2.2 Rooted Trees

Another form of representing a task system is by a rooted tree [HU61]. A rooted

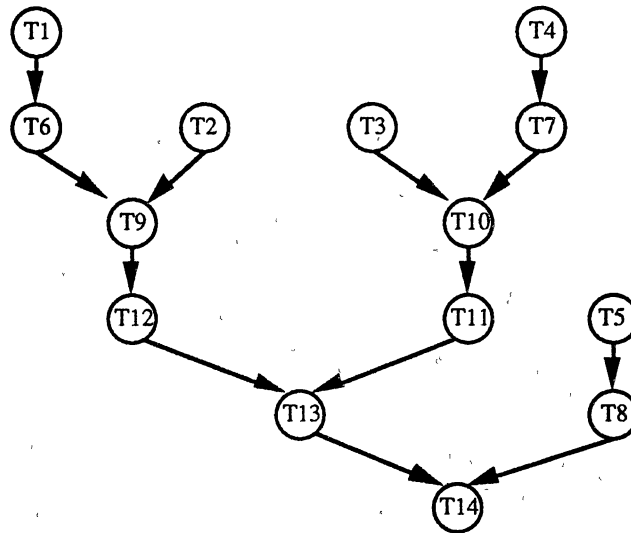


Figure 22. An Example of a Rooted Tree.

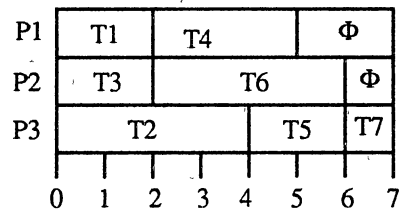
tree is a directed graph similar to a precedence graph in which each node has at most one successor (and any number of predecessors, including zero predecessors). The node which has no successors is called the *root* node. An example of a rooted tree is shown in Figure 22. This corresponds to the RE partitioned into a task system in Figure 20. We can easily see that node T_{14} is the root node and nodes T_1 , T_2 , T_3 , T_4 , and T_5 , which do not have any predecessors in the inverted tree are the leaf nodes.

4.3 Scheduling Algorithms

Once the tasks have been represented in the form of a suitable graph, they can be assigned to a set of processors on the multiprocessing system. A *schedule* or *assignment* for a given precedence graph and a multiprocessor system with p processors, is a complete description of the work to be done by each processor as a function of time [RAM71]. The schedule must not violate any of the precedence relationships in the task graph and it must not allocate more than one processor to a task at any given time. There are several ways of representing schedules for a task systems.

4.3.1 Gantt-Chart Representation of a Schedule

One of the simplest ways to display or specify a schedule is by timing diagrams called *Gantt charts* [CLARK52]. For the precedence graph shown in Section 4.2.1, a schedule on three processors is drawn in the form of a Gantt chart as shown below. An idle period denoted by Φ on this chart is a time interval within which the processor is not



executing any task. The *execution time* or *schedule length* of a schedule is the total time taken to execute all the tasks in the graph as specified by the schedule [COFF76]. The execution time of the schedule shown above is 9. The *Speed-up* ratio S_p obtained by a schedule on p processors is the ratio of the time taken to execute the task system on an uniprocessor system (which is equal to the sum of all node weights) over the execution time obtained by the schedule on p processors [POLY86]. In the example, $S_p = 18/7$ for $p = 3$. The *efficiency (utilization factor)* E_p obtained by a schedule is the ratio of the total busy time of all the processors to the total time during which all the processors were available for execution [HWANG84]. In the example, $E_p = 18/7 * 3 = 18/27$.

4.3.2 Scheduling Algorithm A

In this section we will describe a multiprocessor scheduling algorithm, called Algorithm A, which schedules a task system given in the form of a rooted tree. Algorithm A follows the general approach used in the Hu's algorithm for multiprocessors ([HU61] as cited in [HWANG84]). Each node in the graph is assigned a label as follows.

- The label of the root node is set to 1.
- The label of any other node is set to 1 plus the label of its unique successor node.

For the rooted tree in Figure 22 the label table is shown below.

T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14
6	5	5	6	3	5	5	2	4	4	3	3	2	1

Let L denote the value of the maximum label in the label table, w_i denote the subset of jobs with label i , and $|w_i|$ be the number of tasks in w_i . We define the width, w_G of the graph as,

$$w_G = \max(|w_1|, |w_2|, \dots, |w_L|).$$

Thus, for the graph in Figure 22 we have,

$$w_6 = \{T_1, T_4\}, w_5 = \{T_2, T_3, T_6, T_7\}, w_4 = \{T_9, T_{10}\}, w_3 = \{T_5, T_{11}, T_{12}\}, \\ w_2 = \{T_8, T_{13}\}, \text{ and } w_1 = \{T_{14}\}$$

Hence, the width is

$$w_G = \max(2, 4, 2, 3, 2, 1) = 4.$$

Algorithm A [HU61] for scheduling a rooted tree on p processors is outlined in Figure 23. The trace of this algorithm can be illustrated by scheduling the task graph in Figure 22 on $p=2$ processors. Initially we see that $|w_5| > 2$, and either T_2 or T_3 can be chosen as the victim node since they have no predecessors in w_6 . T_2 is chosen arbitrarily and moved to the set w_6 . At this point the set representation is,

$$w_6 = \{T_1, T_2, T_4\}, w_5 = \{T_3, T_6, T_7\}, w_4 = \{T_9, T_{10}\}, w_3 = \{T_5, T_{11}, T_{12}\}, \\ w_2 = \{T_8, T_{13}\}, \text{ and } w_1 = \{T_{14}\};$$

Now $|w_6| > 2$. Any one of the tasks in w_6 can be chosen as the victim since they are leaf nodes. Again, T_2 is chosen arbitrarily and moved to a new set w_7 . At this point the set representation is,

$$w_7 = \{T_2\}, w_6 = \{T_1, T_4\}, w_5 = \{T_3, T_6, T_7\}, w_4 = \{T_9, T_{10}\}, \\ w_3 = \{T_5, T_{11}, T_{12}\}, w_2 = \{T_8, T_{13}\}, \text{ and } w_1 = \{T_{14}\};$$

Now, since $|w_5| > 2$, T_3 is moved from w_5 to w_6 and then to w_7 . Subsequently, T_5 is moved from w_3 all the way up to w_7 , and then to a new set w_8 . At this point,

```

Label tasks and group them into sets  $w_i$  as described in text.
L1:
  if  $|w_i| \leq p$  for  $i = L, \dots, n, \dots, 1$  then
    Goto L3;
  else if for some  $i$ ,  $|w_i| > p$  then
     $n = i$ ;
L2:
  if  $n \neq L$  then
    find a node from  $w_n$  that does not have any predecessors in  $w_{n+1}$ ;
    /* such a node can always be found in a rooted tree */
    change the node's label from  $n$  to  $n+1$ ;
  end /* then */

  if  $n = L$  then
    select any node from the set  $w_L$  as the victim;
    /* since all are leaf vertices in  $w_L$  */
    change the node's label from  $L$  to  $L+1$ ;
    increment  $L$  by 1;
  end /* then */

  Goto L1;
L3:
  form the schedule as follows:
  for  $i = 1, 2, \dots, L$  do
    execute a task in the set  $w_i$  in the  $(L-i+1)$ th unit of time on one of
    the  $p$  processors;
    /* if less than  $p$  tasks available then the remaining processors idle */
  end /* do */

```

Figure 23. Algorithm A: Scheduling a Rooted Tree on p Processors.

(Source: T.C. Hu. "Parallel Sequencing and Assembly Line Problems", Operations Research, vol. 9, no. 6, pp. 841-848, 1961.)

$w_8 = \{T_5\}$, $w_7 = \{T_2, T_3\}$, $w_6 = \{T_1, T_4\}$, $w_5 = \{T_6, T_7\}$, $w_4 = \{T_9, T_{10}\}$,
 $w_3 = \{T_{11}, T_{12}\}$, $w_2 = \{T_8, T_{13}\}$, and $w_1 = \{T_{14}\}$;

Now, $|w_i| \leq 2$ for $i=1, \dots, 6$. Hence, the schedule can be obtained as shown below.

P1	T5	T3	T4	T6	T9	T11	T8	T14
P2	Φ	T2	T1	T7	T10	T12	T13	Φ
	0	1	2	3	4	5	6	7

Figure 24. Schedule Obtained by Algorithm A on $p=2$ processors.

To schedule the tree on $p=4$ processors, we see that $l_{wi} \leq 4$ for $i=1, \dots, 6$. Hence, the schedule can be readily obtained from the initial set representation of the tasks in the label table as shown below. Note that the width w_G of the tree is also 4. Thus, we can see that when $p=w_G$ (that is, when the number of processors is equal to the width of the graph), the schedule is obtained directly from the initial set representation of the tasks.

P1	T1	T2	T9	T5	T8	T14	
P2	T4	T3	T10	T11	T13	Φ	
P3	Φ	T6	Φ	T12	Φ	Φ	
P4	Φ	T7	Φ	Φ	Φ	Φ	
	0	1	2	3	4	5	6

It should be noted that Algorithm A has certain limitations. Firstly, it is limited to rooted trees and secondly, the rooted tree must have equal weighted nodes.

4.3.3 Another Partitioning Approach

In the RE $(0^*1+11)1 + (00+0^*1)0 + 0^*1$ considered in section 4.1, we see that the sub-expression 0^* and 0^*1 are repeated. In the partitioning approach used in section 4.1 we did not consider this fact, and identified different tasks for the sub-expressions which are repeated in the RE. This approach leads to replication of a task on processors, either different or same, which is a major disadvantage. This can be avoided by identifying a repeated sub-expression with a single task, and then establish the dependencies to other tasks. An RE with common sub-expressions is partitioned as shown in Figure 25. Then the tasks can be represented in the form of a DAG as mentioned earlier.

4.3.4 Scheduling Algorithm B

The resulting DAG obtained by the above mentioned partitioning approach will not be a rooted tree, since it will have nodes having more than one successor. Since Algorithm A is limited to rooted trees, it cannot be used to schedule this DAG. A multiprocessor

RE: $(0*11+0*)0*110 + (0*11+11)0$

Polish notation: $(0 * 0 \cdot 1 * 0 + 0 * 0) \cdot 0 * 0 \cdot 1 * 0 \cdot 0 + (0 * 0 \cdot 1 * 0 + 1 * 0) \cdot 0$

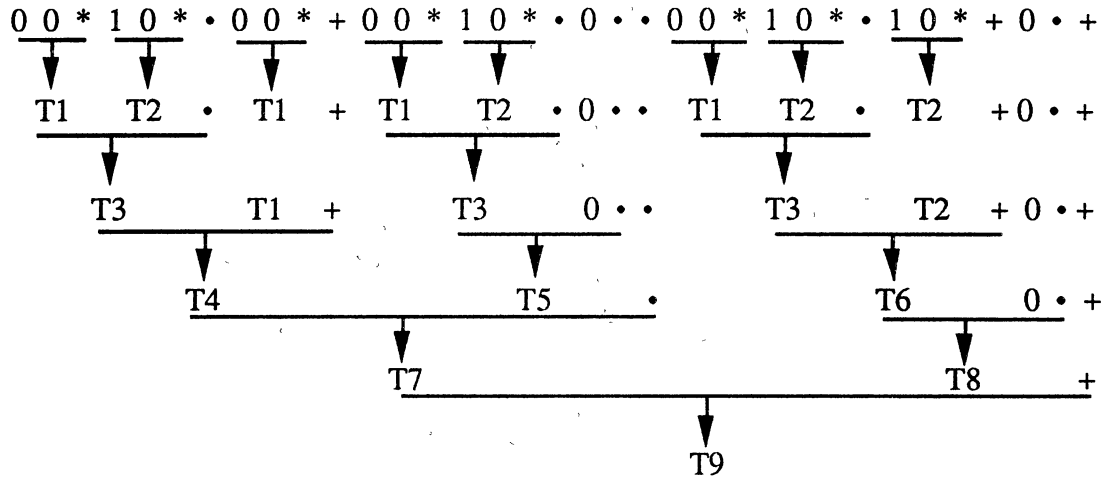


Figure 25. Partitioning an RE with Common Sub-expressions.

scheduling algorithm called Algorithm B has been developed to overcome this problem.

The algorithm has a preprocessing step in converting the given DAG with some nodes having multiple successors into a rooted-tree form with equal node weights.

Removing the constraint of equal node weights

The constraint that all nodes should have equal weights can be eliminated as outlined below. A process of normalization can be defined for general task graphs. Any task graph G can be converted into another graph G' in which all nodes have equal weights [GON77] as shown in Figure 26. A node n_i with weight w_i can be split into a sequence of r nodes (all with execution time t) such that $w_i = rt$. Then the graph G is redrawn to obtain G' by maintaining the integrity of the original graph. The idea is to find the least factor of all the node weights in the original graph G and use this factor as the value for t . Obviously, if this factor is anything other than 1, the value r for each node is not very large. This keeps the exploration of nodes in the new graph G' , which is the cost of the conversion process, to a low value.

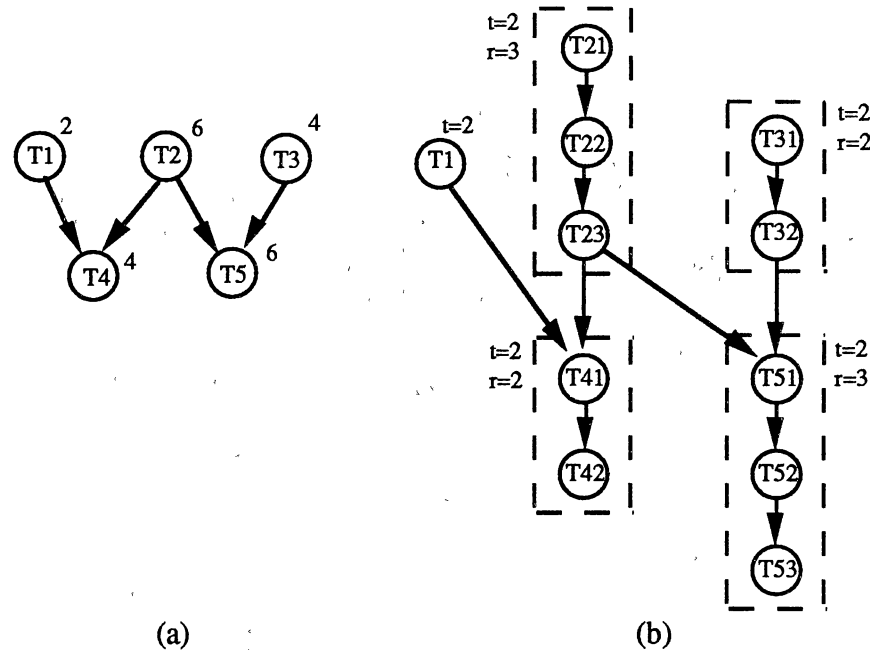


Figure 26. (a) Graph with Tasks of Unequal Node Weights.
(b) Graph with Tasks of Equal Node Weights.

(Source: M.J. Gonzalez. "Deterministic Processor Scheduling", *Computing Surveys*, vol. 9, no. 3, p. 186, September 1977.)

Removing the constraint of *limitation to a rooted tree*

The second constraint of "limitation to a rooted tree" in Algorithm A can be removed as follows. Consider the DAG shown in Figure 27(a). Starting from T_1 , the condition of a rooted tree is violated at node T_1 which has two successors. This violation can be removed by replicating the subtree above (and including) the node T_1 as many times as the number of successors of T_1 . Since T_1 has no predecessors (a source node), T_1 is simply duplicated as T_{11} and T_{12} . The graph is transformed as shown in Figure 27(b) by drawing the directed arcs from T_{11} to T_2 , and from T_{12} to T_3 corresponding to the precedence relations among tasks T_1 , T_2 , and T_3 in the original graph. For the graph in Figure 27(b) the condition of a rooted tree is again violated at node T_3 which has three successors. Hence the subtree above and including T_3 is replicated three times. Subsequently, the precedence relations are established by drawing the directed arcs between T_{31} and T_4 , T_{32} and T_5 , and T_{33} and T_7 . Note that T_{12} in the subtree also gets

replicated as T_{121} , T_{122} , and T_{123} . Now the graph in Figure 27(c) is a complete rooted-tree version of the DAG in Figure 27(a).

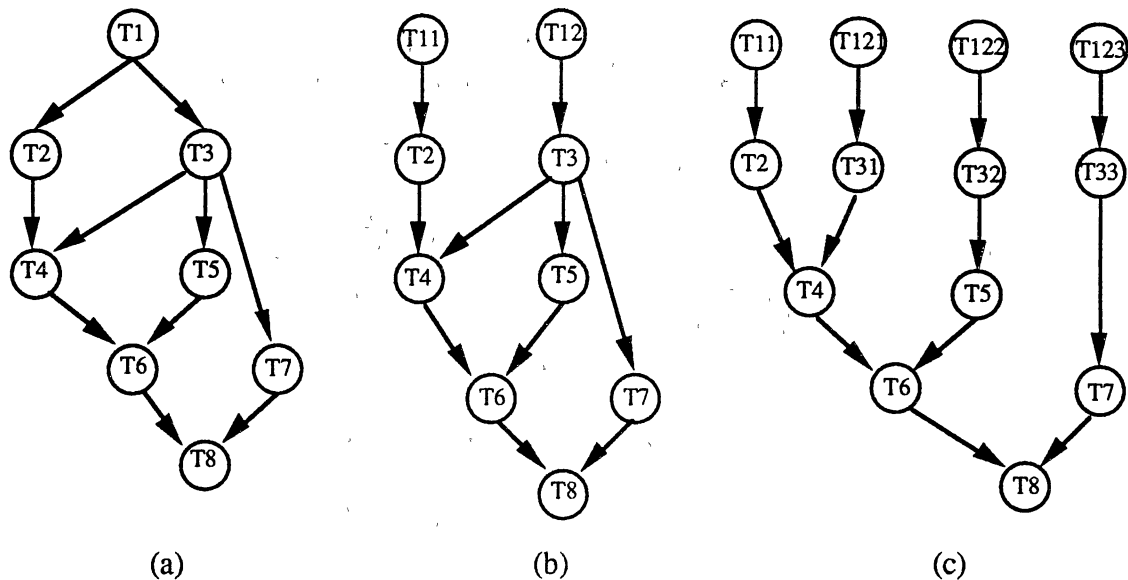


Figure 27. Illustration of Converting a DAG into a Rooted Tree.

Obviously, this method suffers from the disadvantage of the replication of nodes in the resulting rooted tree. For a node at a violation point, let P be the number of all nodes in the subtree above it and let S be the number of its successors. It can be seen easily that the replication factor at each violation point is given by $(P+1)S - (P+1) = (P+1)(S-1) \approx PS$. But, it will be shown in Scheduling Algorithm B that the replication of nodes is only for the purpose of representation and will not be reflected in the schedule to be developed. That is, if a node T_1 is duplicated, it will not be assigned twice as T_{11} and T_{12} to two processors (same or different).

Now we will describe the scheduling Algorithm B which assumes that a given task graph has already been preprocessed so that a corresponding rooted tree with equal node weights is available. Consider the rooted tree with replicated nodes in Figure 28(a) which corresponds to a given DAG that has been preprocessed. Each node in the graph is assigned a label as follows:

- The label of the root node is set to 1.
- The label of any other node (including replicated nodes) is set to 1 plus the label of its unique successor node.

Using this labeling scheme, the label table for the tree is obtained as shown in Figure 28(b). Note that replicated nodes have different labels.

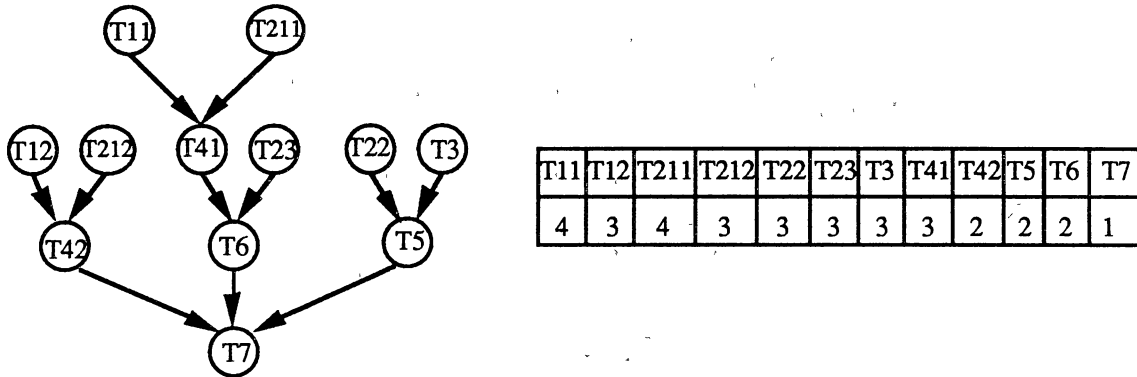


Figure 28. An Example of a Rooted Tree with Repeated Nodes and Its Label Table.

Let L denote the value of the maximum label in the label table, w_i denote the set of tasks with label i , and $|w_i|$ denote the number of tasks in w_i . The width w_G of the graph is defined as in Algorithm A. The initial set representation of the tasks is as follows.

$$w_4 = \{T_{11}, T_{21}\}, w_3 = \{T_{41}, T_{22}, T_{12}, T_{23}, T_{24}, T_3\}, w_2 = \{T_{42}, T_5, T_6\}, w_1 = \{T_7\}$$

Once the tasks have been labeled and grouped into sets Algorithm B, outlined in Figure 29, can be used for scheduling the rooted tree (with repeated nodes) on p processors.

The trace of this algorithm for scheduling the above tree on $p=2$ processors is as follows. Initially, we see that the leaf nodes T_1 and T_2 (actually T_{11} and T_{21} which will be scheduled as T_1 and T_2 respectively) can be scheduled from w_4 during the first time interval. This reduces w_4 to $\{\}$. Subsequently, T_{41} (scheduled as T_4) is selected from w_3 for the second interval (since both its predecessors have been scheduled) and removed from w_3 . Now, T_{22} can be chosen, but being a repeated task one of whose counterparts is already scheduled in the previous time interval, T_{22} is removed from w_3 . At this point,

$$w_4 = \{\}, w_3 = \{T_{12}, T_{23}, T_{24}, T_3\}, w_2 = \{T_4, T_5, T_6\}, \text{ and } w_1 = \{T_7\}$$

```

repeat
  Select at most  $p$  tasks from  $w_i$  such that:
  they are the leaf nodes;
  or
  all their predecessors have been assigned in an interval previous to the
  current time interval;

  if the predecessor of a task is a repeated node then
    any counterpart of a repeated node is considered its predecessor;

  if a repeated node needs to be selected then
    if any of its counterparts has been selected earlier or in the
    current interval then
      discard it from the current set  $w_i$ ;
    else select it for the current time interval;
  end; /* then */

  if all tasks in the set  $w_i$  have been tried for selection then
    goto next set  $w_{i+1}$ ;

  Schedule the  $p$  (or fewer) tasks on  $p$  processors during the current interval;
until (all tasks have been scheduled);

```

Figure 29. Algorithm B: Scheduling a Rooted Tree With Repeated Nodes on p Processors.

Similarly, tasks T_{12} , T_{23} , and T_{24} are tried and removed from w_3 . Then, task T_3 is selected since it satisfies all the conditions. In this manner, the complete schedule is constructed as shown in Figure 30.

P1	T1	T4	T6	T7	
P2	T2	T3	T5	Φ	
	0	1	2	3	4

Figure 30. Schedule Obtained by Algorithm B on $p=2$ Processors.

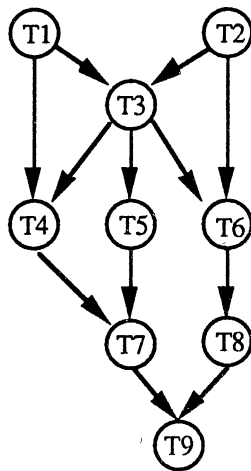
4.3.5 Scheduling Algorithm C

Algorithm B has a disadvantage in the form of replication of nodes during the phase

of preprocessing. A multiprocessor scheduling algorithm called Algorithm C is developed which also schedules a task system given in the form of a DAG on p processors. The advantage of Algorithm C over Algorithm B is that the phase of converting a DAG into a rooted tree is eliminated and the algorithm schedules the DAG directly.

Consider the task graph with some nodes having multiple successors as shown in Figure 31. Each node in it is assigned a label as follows.

- The label of the root node is set to 1.
- The label of any other node is set to 1 more than the label of its successor. If a node has more than 1 successor, then the maximum label is considered.



T9	T8	T7	T6	T5	T4	T3	T2	T1
1	2	2	3	3	3	4	5	5

Label Table

Figure 31. An Example of a DAG with Some Nodes Having Multiple Successors.

Using this labeling scheme the label table for the DAG is obtained as shown in Figure 31. Note that the label of the task T_1 is 4, and not 3.

Again, let L denote the maximum label in the table, w_i denote the set of tasks with label i , and $|w_i|$ denoted the number of tasks in the set w_i . The width w_G of the graph is defined as in Algorithm A. Thus, for the DAG shown above, the initial set representation of the tasks is as follows.

$$w_5 = \{T_1, T_2\}, w_4 = \{T_3\}, w_3 = \{T_4, T_5, T_6\}, w_2 = \{T_7, T_8\}, w_1 = \{T_9\}$$

Once the tasks have been labeled and grouped into sets, they can be scheduled on any number, say p , of processors using Algorithm C outlined in Figure 32.

```

Label nodes and group them into sets  $w_i$  as described in text.
L1:
  if  $|w_i| \leq p$  for  $i = L, \dots, n, \dots, 1$  then goto L3;
  else if for some  $i$ ,  $|w_i| > p$  then  $n = i$ ;
L2:
  if  $n \neq L$  then
    find a node from  $w_n$  that does not have any predecessors in  $w_{n+1}$ ;
    if no such node available in  $w_n$  then
       $n = n + 1$ ;
      goto L2
    end /* then */
    change the node's label from  $n$  to  $n+1$ ;
  end /* then */

  if  $n = L$  then
    select any node from the set  $w_L$  as the victim;
    /* all are leaf nodes in  $w_L$  */
    change the node's label from  $L$  to  $L+1$ ;
    increment  $L$  by 1;
  end /* then */
  goto L1;
L3:
  form the schedule as follows:
  for  $i = 1, 2, \dots, L$  do
    Execute a task in the set  $w_i$  in the  $(L-i+1)$ th unit of time on the
     $p$  processors;
    /* if fewer than  $p$  tasks available then remaining processors idle */
  end /* do */

```

Figure 32. Algorithm C: Scheduling a DAG on p Processors Directly.

The trace of the algorithm for scheduling the DAG in Figure 31 on $p=2$ processors is as follows. Initially, we see that $|w_3| > 2$, but none of T_4 , T_5 , or T_6 can be chosen as the victim node since they have a predecessor in w_4 . So we try to search for the victim in w_4 . Again we fail and finally T_1 in w_5 is chosen as the victim. Then T_1 is removed from w_5 and added to a new set w_6 . At this point,

$$w_6 = \{T_1\}, w_5 = \{T_2\}, w_4 = \{T_3\}, w_3 = \{T_4, T_5, T_6\}, w_2 = \{T_7, T_8\}, w_1 = \{T_9\}$$

Again $|w_3| > 2$, and, following the same argument, T_2 is now removed from w_5 and added to w_6 . Now, we have

$$w_6 = \{T_1, T_2\}, w_5 = \{\}, w_4 = \{T_3\}, w_3 = \{T_4, T_5, T_6\}, w_2 = \{T_7, T_8\}, w_1 = \{T_9\}$$

In the next step, T_3 will be removed from w_4 and added to w_5 . Then one of T_4 or T_5 or T_6 can be moved to w_4 , giving a set representation as follows.

$$w_6 = \{T_1, T_2\}, w_5 = \{T_3\}, w_4 = \{T_4\}, w_3 = \{T_5, T_6\}, w_2 = \{T_7, T_8\}, w_1 = \{T_9\}$$

Now, $|w_i| \leq 2$ for $i=1, \dots, 5$, and hence the schedule is formed as follows.

P0	T1	T3	T4	T5	T7	T9	
P1	T2	Φ	Φ	T6	T8	Φ	
	0	1	2	3	4	5	6

Figure 33. Schedule Obtained by Algorithm C on $p=2$ Processors.

4.4 Implementation and Optimization Issues

Various aspects of implementation of the problem on the iPSC/2 hypercube system, and several optimization issues are considered in the following subsections.

4.4.1 Optimal Number of Processors

From the algorithms described above, it can be seen easily that when $p=w_G$ (that is, when the available number of processors is equal to the width of the graph), the schedule can be directly obtained from the initial set representation of the tasks in the label table. The length of the schedule thus obtained will be equal to L , where L is the number of nodes in the longest path in the DAG. Moreover, it has been shown that, for any precedence graph, L is the minimal schedule length of any optimal schedule [CON67]. Hence, the schedules obtained by either Algorithm A, B, or C have minimum schedule lengths when the available number of processors is equal to the width of the graph.

Thus we conclude with the following lemma which captures this important observation from these scheduling algorithms.

Lemma: Consider a task system of n tasks given in the form of a graph G (either a DAG or a rooted tree) which has a width, $w_G=k$. This system can be executed by Algorithm A, B

or C in L units of time on k processors where L is the number of nodes in the longest path in G .

This is a definite improvement over scheduling the graph G on m processors by any other algorithm, which would also yield a schedule-length of L , where m is the number of initially available tasks (or leaf nodes) of G . Larger number of processors (i.e., more than m in this case) would result in poor processor utilization. Suppose a program can be executed in a given time interval by a lesser number of processors than the available number of processors. Then, the remaining processors could be used as back up processors in critical applications, resulting in increased reliability and efficiency. In non-critical applications, they can be used for doing some important background operations, resulting in better utilization and performance.

Moreover, the schedule obtained on p ($>w_G$) processors will have the same schedule length as the schedule obtained when $p=w_G$. In the case when $p>w_G$, the additional (w_G-p) processors will idle throughout the schedule length. Thus, as an optimization, the user is informed of this optimal number of processors.

4.4.2 Suitability Issue

Another factor, which is the suitability of this application on the hypercube, is discussed in this section. To achieve better speed-up and utilization, each node (individual processor) on the hypercube should have a sufficiently large computation time in comparison to the communication time between the nodes (communication between nodes is required for synchronization purposes). This was realized by feeding a regular expression reasonably large in size (that is, one which contains a large number of terms and numerous operators) as the input to the cycle of transformations in Figure 8. Typically, at the end of every cycle of transformations, there would be an explosion of terms in the new regular expression relative to the most recent one. This could be attributed to transformations T3 (converting an NFA to a DFA) or T6 (solving the set of RE equations)

depending on whether the shorter or longer cycle of transformations is chosen. Hence by feeding a large enough RE, we obtained an RE much larger in size every time through the cycle of transformations. As the size of the RE grows the corresponding finite automaton has a large number of states, and hence the computation needed in synthesizing and subsequently processing such an FA is also large. Thus, each node performs a sufficiently larger number of computations on successive cycles of transformations. Hence, this application is suitable for the hypercube as the computation time is on a larger scale than the communication time.

4.4.3 Memory Allocation Issues

Another implementation issue discussed in this section is the memory allocation for the data structures used in the program. A DFA is represented by a 2-dimensional array and an NFA by a 3-dimensional array (so as to accommodate for its nondeterministic behavior) in the program developed. A major factor which influences the allocation of memory for these structures is the indecisiveness of their sizes. As noted earlier, there is an explosion of states occurring in a finite automaton through every cycle of transformations. This explosion factor is not determinate. Thus, no definite bound can be fixed on the size of the arrays for the NFA and the DFA. One approach is to decide on an arbitrary size and statically allocate memory to these arrays. But, this approach has two major drawbacks. It might lead to an inefficient usage of memory if the automata are not large enough to fill the whole array. On the other hand, the approach might even fail when the explosion of states in the NFA is too large to be accommodated in the fixed size arrays. Such examples have been encountered and can be evidenced in Appendixes C and D.

A better approach that has been adopted in the program is to obtain an approximately close estimate on the number of states in the NFA based on the number of operators in the RE at the beginning of each iteration. Initially, the arrays are sized to this estimate by using dynamic memory allocation features, which facilitate altering the size of

the arrays. Thus if the number of states in the NFA grows larger than the size of the array, more memory is allocated to them. This technique avoids the uncertainty and the problems involved in the previous approach. In addition, the 3-dimensional array used for representing an NFA has been observed to be mostly sparse. That is, only certain states in the NFA exhibit the non-deterministic behavior, which implies that only certain lists in the 3-dimensional array expand in size while others donot. Hence only such lists need to be allocated more memory. This calls for the use of sparse array techniques for the NFA-array. In the program implemented for this thesis project the NFA-array is realized by using a 2-dimensional array of linked lists and the dynamic memory allocation features. These techniques helped reduce the amount of memory used by the program.

4.4.4 Machine-Independent Communication Issues

An optimization issue in reducing the communication overhead is discussed in this section. By arbitrarily assigning a task T_i with a final label i to a processor during a time unit, we obtained the schedule for $p=2$ case as shown in Figure 24 of section 4.3.2. In this schedule, we see that T_6 assigned to processor P_1 depends on T_1 assigned to processor P_2 . Hence processor P_2 has to communicate the result of T_1 to processor P_1 ; until then task T_6 "blocks" on processor P_1 . Similarly, task T_7 blocks until it gets the result of T_4 from processor P_1 . The same situation exists during the 5th time interval, when tasks T_9 and T_{10} have to block for the results of their predecessors T_2 and T_3 , respectively. This communication overhead can be avoided if a task is assigned judiciously to a processor, based on the task's predecessors. For example, when T_6 is to be assigned we check that its predecessor T_1 has been assigned to processor P_2 , and hence we try to assign T_6 to processor P_2 . If a task has more than one predecessor, then we assign it to that processor to which at least any one of its predecessors has been assigned. Following this optimization, the schedule on $p=2$ processors for the tree in Figure 22 is shown in Figure 34. From this schedule, we see that only one communication is required between the two

processors between tasks T_{11} and T_{13} . Since communication involves wasting time, reducing the communication between processors is a significant improvement.

P1	T5	T3	T4	T7	T10	T11	T8	T14
P2	Φ	T2	T1	T6	T9	T12	T13	Φ
	0	1	2	3	4	5	6	7

Figure 34. Schedule Obtained by Algorithm A after Communication Optimization.

4.4.5 Machine-Dependent Communication Optimization

Another implementation as well as optimization issue is discussed in this section. It also concerns the aspect of communication between processors which is important in implementing parallel programs on the hypercube system. It is made feasible by the use of message passing features available on the iPSC/2 hypercube system.

Consider the rooted tree and the corresponding schedule obtained by Algorithm A on two processors as shown in Figure 35. We consider the execution sequence of processor P2. Initially P2 can execute the task T2, and then has to wait for the message to

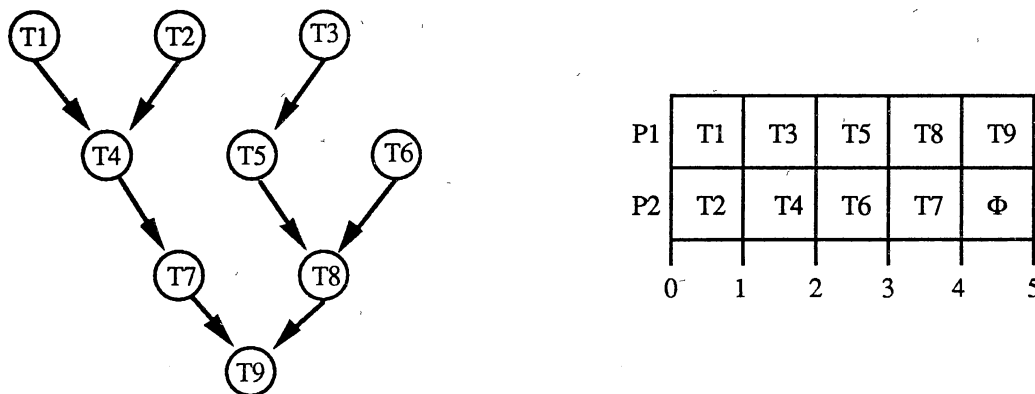


Figure 35. An Example of a Rooted Tree and a Schedule to Illustrate the Look Ahead Approach.

be arrived from processor P1 for the execution of task T4. The conventional approach would be to block execution until the message has arrived by using the synchronous "crecv" call of the desired message type on the iPSC/2 system. This approach is simple and straightforward from the programmers point of view, but results in the loss of processor time (a valuable resource) and a corresponding decrease in the speed-up.

As an alternative, processor P2 posts an asynchronous receive request for the desired message through the "irecv" call on the iPSC/2 system. It then checks if the message has arrived yet or not, through the "cprobe" message information call. If the message has already arrived, task T4 is immediately executed. If it has not yet arrived, then processor P2 adopts the "look-ahead" approach and scans its job queue for the next task that is "ready" to be executed. A "ready" task is one with no predecessors at all or one whose predecessors have all been executed on the same processor or one which has received messages from all its predecessors from all processors. In this case, processor P2 finds task T6 which is an independent task and hence executes it, instead of blocking at T4. Subsequently, it switches context back to task T4, probes again for its requested receive message, and again follows the look-ahead approach, if necessary. It should be noted that multiple probing calls can be safely posted, but posting multiple receive calls for one pending message to be received is a programming error.

Although an asynchronous "crecv" call was posted for receiving a message on processor P2 (receiving processor), the sending processor (processor P1 in this case) has the choice to send the message through any type of call. In this program, messages are always sent by the synchronous "csend" call, since it blocks only for a short period of time until the message leaves the sending processor. Also, a simple "csend" call avoids any further programming complexity.

It is emphasized here that, sending a message synchronously and receiving it asynchronously is made feasible by the flexibility, provided by the iPSC/2 system, of freely mixing different types of message passing calls.

CHAPTER V

SUMMARY AND FUTURE WORK

5.1 Summary

Since parallel processing provides a possible solution to solving computationally intensive problems, the current trend is to avoid the limitations of uniprocessor systems by using several processors. The underlying principle of parallel processing is to connect several powerful processors into a single system and make them solve a complex problem through coordination and cooperation with each other.

The problem that was considered for parallel implementation in this thesis is a set of transformations that can be performed on regular expressions and finite automata both of which form the basis of a "lexical analyzer". The parallelism existing in the sequential algorithms for these transformations was exploited to develop parallel algorithms. Subsequently, the parallel algorithms were implemented in the C programming language on a typical parallel processor, namely Intel's iPSC/2 which is a 32-processor, distributed-memory system with a hypercube interconnection topology between the processors. Certain multiprocessor performance measures, such as *speed-up*, *processor efficiency*, and *serial fraction* were evaluated and the results are discussed.

A given RE was initially partitioned and represented in the form of a task graph by an approach which yields a "rooted tree". Then a multiprocessor scheduling algorithm called Hu's algorithm (referred to as Algorithm A in this report) was used and implemented to schedule the tasks in the rooted tree on a varying number of processors. Realizing the limitations of Algorithm A, a different approach to partitioning the RE was adopted which yields a DAG with nodes having multiple successors (this is a better approach than the

partitioning approach yielding a rooted tree, since it considers all repeated sub-tasks as a single task). Subsequently, two scheduling algorithms were developed to execute such a DAG. One of them, called Algorithm B, was analyzed theoretically and was not implemented due to its cost factor. The other scheduling algorithm, called Algorithm C, was developed as an extension to Hu's algorithm and was implemented. Both theoretical and practical values have been obtained for the performance measures using both Algorithms A and C. Also, the results have been discussed and compared.

An important observation was made from these algorithms regarding the number of processors needed to schedule a task graph in minimum time. This observation, which depends on the "width" of the graph, has been stated as a lemma. In addition, several optimizations have been realized in developing the schedules for varying number of processors which helped reduce the communication between tasks assigned to different processors. These optimizations have been realized in the implementation and schedules obtained with and without these optimizations were compared.

As another objective, the changing form of a regular expression, which has been subjected to a set of transformations a number of times, has been studied. It was observed that one such cycle of transformations appeared to produce regular expressions that are in a closed form which can be loosely called a "canonical form". Another cycle of transformations always yielded the canonical form generally after a larger number of iterations than the first one (or it would not even produce a canonical form). Thus it was decided that the former cycle "converges" and the latter cycle does not always converge, or it "diverges".

Certain limitations of the program are mentioned below. The size of a given RE, and hence the NFA, grows exponentially through each iteration of the "divergent" cycle. Under such situations the program demands too much memory which might fail. Thus, the program is restricted to a certain size of the RE and the NFA. Also, the size of the input symbol set is limited due to memory constraints. Another drawback of the thesis research

is that space and/or time complexity analysis for the scheduling Algorithms A, B, and C is not done in this project.

5.2 Future Work

It should be noted that Algorithms A, B, and C assumed, prior to scheduling, that all the tasks have "approximately" equal execution times. This assumption holds true in the case of evaluation of RE's because in the process of transforming REs, each individual task represents either "concatenation", "union", or "closure" of tasks all of which have the same time complexity in the corresponding algorithm. Other algorithms which do not have such restrictions can be investigated and implemented. Another improvement would be to predetermine the execution times of all tasks, through historical data, heuristics, or some kind of preprocessing, and then use a suitable scheduling algorithm. This would probably lead to stochastic scheduling approaches which are more practical.

In the scheduling algorithms the communication time required to communicate information between tasks on different processors was assumed to be negligible. But in practice, the communication time significantly reflects upon the speed-up and other measures. Thus another improvement, which has practical significance, is to estimate the communication time based upon the amount of information to be communicated between processors and upon the communication scheme used on the implementation platform. Such multiprocessor scheduling algorithms, which utilize the communication times during the scheduling process, could be developed and implemented. Some such algorithms are available in the literature.

Other future work includes investigation of other transformations that can be performed on regular expressions and finite automata, implementation of Algorithm B and comparing the results with that of the other algorithms, implementation of all the algorithms on shared memory parallel processor and comparison of results, and time/space complexity analysis of all scheduling algorithms.

REFERENCES

- [AHO72] A.V. Aho and J.D. Ullman. The Theory of Parsing, Translation, and Compiling, Prentice-Hall, Englewood Cliffs, NJ, 1972.
- [AHO86] A.V. Aho, R. Sethi, and J.D. Ullman. Compilers Principles, Techniques and Tools, Addison-Wesley, Reading, MA, 1986.
- [ALMA89] G.S. Almasi and A. Gottlieb. Highly Parallel Computing, The Benjamin/Cummings Pub. Co., Inc., Redwood City, CA, 1989.
- [ANI89] O. Anita (Ed.) Guide to Parallel Programming on Sequent Computer Systems, Prentice-Hall, Englewood Cliffs, NJ, 1989.
- [ARD60] D.N. Arden. "Delayed Logic and Finite State Machines", in Theory of Computing Machine Design, pp. 1-35, Univ. of Michigan, Ann Arbor, MI, 1960.
- [BARN68] G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.J. Slotnick, and R.A. Stokes. "The ILLIAC IV Computer", IEEE Trans. on Computers, vol. C-17, no. 8, pp. 746-757, August 1968.
- [BRZO62] J.A. Brzozowski. "A Survey of Regular Expressions and Their Applications", IRE Trans. on Electronic Computers, vol. EC-11, no. 3, pp. 324-335, June 1962.
- [CLARK52] W. Clark. The Gantt Chart, 3rd Edition, Sir Isaac Pitman & Sons, Ltd., London, 1952.
- [CLOSE88] P. Close. "The iPSC/2 Node Architecture", Proc. of the 3rd Conference on Hypercube Concurrent Computers and Applications, pp.43-50, 1988.
- [COFF76] E.G. Coffman Jr. (Ed.) Computer and Job-Shop Scheduling Theory, John Wiley, New York, NY, 1976.
- [CON67] R.W. Conway, W.L. Maxwell, and L.W. Miller Theory of Scheduling, Addison-Wesley, Reading, MA, 1967.
- [CONT70] "Control Data STAR-Computer System", Publication 6025600, Hardware Reference Manual, Control Data Corporation, Arden Hills, MN, 1970.
- [DEN85] P. Denning. "The Science of Computing: Parallel Computation", American Scientist, vol. 73, no. 4, pp. 322-323, August 1985.
- [DES87] G.R. Desrochers. Principles of Parallel and Multiprocessing, Intertext Publications, Inc., McGraw-Hill Book Co., New York, NY, 1987.

- [DOR85] P. Ein-Dor. "Grosch's Law Revisited", Comm. ACM, vol. 28, no. 2, pp. 142-151, February 1985.
- [FENG77] T.Y. Feng. (Ed.) "An Overview of Parallel Processors and Processing", ACM Computing Surveys, special issue, vol. 9, no. 1, March 1977.
- [FIS88] N.C. Fischer and J.R. LeBlanc Jr. Crafting a Compiler, The Benjamin/Cummings Pub. Co., Inc., Menlo Park, CA, 1988.
- [FLYN72] M.J. Flynn. "Some Computer Organizations and Their Effectiveness", IEEE Trans. on Computers, vol. C-21, no. 9, pp. 948-960, September 1972.
- [FOX88] G.C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. Solving Problems on Concurrent Processors: General Techniques and Regular Problems, vol. I, Prentice-Hall, Englewood Cliffs, NJ, 1988.
- [GAJ85] D.D. Gajski and J.K. Peir. "Comparison of Five Multiprocessor Systems". Parallel Computing, vol. 2, pp. 265-282, November 1985.
- [GEHR87] E.F. Gehringer, D.P. Siewiorek, and G. Segall. Parallel Processing: The Cm* Experience, Digital Press, Digital Equipment Corporation, USA, 1987.
- [GON77] M.J. Gonzalez. "Deterministic Processor Scheduling", Computing Surveys, vol. 9, no. 3, pp. 173-204, September 1977.
- [GRAH87] J. Graham, and J. Rattner. "Expert Computations on the iPSC Concurrent Computer", Multiprocessors and Array Processors, edited by W.J. Karplus, pp. 167-176, Simulation Councils, Inc., San Diego, CA, January 1987.
- [GURD85] J.R. Gurd, C.C. Kirkham, and I. Watson. "The Manchester Prototype Dataflow Computer", Comm. of the ACM, vol. 28, no. 1, pp. 34-52, January 1985.
- [HAND77] W. Handler. "The Impact of Classification Schemes on Computer Architectures", Proc. of 1977 International Conference on Parallel Processing, edited by J.L. Baer, pp. 7-15, Detroit, MI, 1977.
- [HAYES88] J.P. Hayes. Computer Architecture and Organization, 2nd Edition, McGraw-Hill Book Co., New York, NY, 1988.
- [HILL85] W.D. Hillis. The Connection Machine, MIT Press, Cambridge, MA, 1985.
- [HIRS82] D.S. Hirschberg. "Parallel Graph Algorithms without Memory Conflicts". Proc. of 20th Allerton Conference, pp. 257-263, 1982.
- [HOCK81] R.W. Hockney and C.R. Jesshope. Parallel Computers, Hilger, Bristol, 1981.
- [HOP79] J.E. Hopcroft and J.D. Ullman. Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, 1979.

- [HU61] T.C. Hu. "Parallel Sequencing and Assembly Line Problems", Operations Research, vol. 9, no. 6, pp. 841-848, 1961.
- [HWANG84] K. Hwang and F.A. Briggs. Computer Architecture and Parallel Processing, McGraw-Hill Book Co., New York, NY, 1984.
- [HWANG89] K. Hwang and D. Deroort. Parallel Processing Supercomputer and Artificial Intelligence, McGraw-Hill Series in Supercomputing and Parallel Processing, McGraw-Hill Book Co., New York, NY, 1989.
- [IPSC89] iPSC/2 User's Guide, Intel Scientific Computers, Beaverton, Oregon, October 1989.
- [JORD83] H.F. Jordan. "Performance Measurements of HEP-Pipelined MIMD Computer". Proc. of the 10th Annual Symposium on Computer Architecture, Stockholm, Sweden, pp. 207-212, June 1983.
- [KAIN89] R.Y. Kain. Computer Architecture: Software and Hardware, vol. II, Prentice-Hall, Englewood Cliffs, NJ, 1989.
- [KALO87] M.H. Kalos. "Monte Carlo Methods and the Computers of the Future". Supercomputers-Algorithms, Architectures, and Scientific Computation, Edited by F.A. Matsen and T. Tajima, Univ. of Texas Press, 1987.
- [KAR90] A.H. Karp and H.P. Flatt. "Measuring Parallel Processor Performance", Comm. of the ACM, vol. 33, no. 5, pp. 539-543, May 1990.
- [KARIN87] S. Karin and P.N. Smith. The Supercomputer Era, Harcourt Brace Jovanovich Publishers, Boston, MA, 1987.
- [KARP87] W.J. Karplus. (Ed.) Multiprocessors and Array processors, The Society of Computer Simulation, Simulations Councils, Inc., San Diego, CA, Jan 1987.
- [KOG81] P.M. Kogge. The Architecture of Pipelined Computers, McGraw-Hill Book Co., New York, NY, 1981.
- [KUCK78] D.J. Kuck. The Structure of Computers and Computations, vol. 1, John Wiley, New York, NY, 1978.
- [KUNG82] H.T. Kung. "Why Systolic Architectures?", IEEE Computer, vol. 15, no. 1, pp. 37-46, January, 1982.
- [LAK90] S. Lakshmivarahan and S.K. Dhall. Analysis and Design of Parallel Algorithms: Arithmetic and Matrix problems, McGraw-Hill Book Co., New York, NY, 1990.
- [LEA87] R.M. Lea. "An Overview of the Influence of Technology on Parallelism", Major Advances in Parallel Processing, Edited by C. Jesshope, pp. 3-12, 1987.
- [MOIT87] A. Moitra and S.S. Iyengar. "Parallel Algorithms for some Computational Problems", Advances in Computers, Edited by M.C. Yovits, vol. 26, pp. 94-153, Academic Press, San Diego, CA, 1987.

- [NUG88] S. Nugent. "The iPSC/2 Direct-Connect Technology", Proc. of the 3rd Conference on Hypercube Concurrent Computers and Applications, pp. 59-68, 1988.
- [PALM87] J.F. Palmer. "The NCube Family of Supercomputers", Multiprocessors and Array Processors, Edited by W.J. Karplus, pp. 177-187, Simulation Councils, Inc., San Diego, CA, January 1987.
- [PETER85] V.L. Peterson. "Use of Supercomputers in Computational Aerodynamics", Proc. of the 1985 Science and Energy Symposium, Cray Research Inc., Minneapolis, 1985.
- [POLY86] C.D. Polychronopoulos. "On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems", Centre for Supercomputing Research & Development, Rep. No. 595, August 1986.
- [PIER88] P. Pierce. "The NX/2 Operating System", Proc. of the 3rd Conference on Hypercube Concurrent Computers and Applications, pp. 51-57, 1988.
- [POTT86] J.L. Potter. The Massively Parallel Processor, 2nd Edition, The MIT Press, Cambridge, MA, 1986.
- [RAM71] C.V. Ramamoorthy, K.M. Chandy, and M.J. Gonzalez. "Optimal Scheduling Strategies in Multiprocessor Systems", IEEE Trans. on Computers, vol. C-21, no.2, pp. 137-146, February 1971.
- [SARK89] V. Sarkar. "Partitioning and Scheduling Parallel Programs for Multiprocessors", Research Monographs in Parallel and Distributed Computing, The MIT Press, Cambridge, MA, 1989.
- [SEIT84] C.L. Seitz and J. Matisoo. "Engineering Limits on Computer Performance", Physics Today, vol. 37, no. 5, pp. 38-45, May 1984.
- [SLOT62] D.L. Slotnick, C.W. Borck, and R.C. McReynolds. "The SOLOMON Computer", Proc. of the AFIPS Fall Joint Computer Conference, vol. 22, pp. 97-107, 1962.
- [SOR76] P.G. Sorenson and J.P. Tremblay. An Introduction to Data Structures with Applications, McGraw-Hill Book Co., New York, NY, 1976.
- [SUD88] T.A. Sudkamp. Languages and Machines: An Introduction to the Theory of Computer Science, Addison-Wesley, Reading, MA, 1988.
- [SULL77] H. Sullivan, F.R. Bashkow, D. Klappholz, and L. Cohn. "A Large Scale Homogeneous, Fully Distributed Parallel Machine", Proc. of the 4th Annual Symposium on Computer Architecture, College Park, MD, pp. 105-124, 1977.
- [TREL82] P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins. "Data-Driven and Demand-driven Computer Architectures", ACM Computing Surveys, vol. 14, no. 1, pp. 95-143, March 1982.
- [WIL87] R. Wilhelmson. (Ed.) High Speed Computing: Scientific Applications and Algorithm Design, University of Illinois Press, 1987.

APPENDIXES

APPENDIX A

USER MANUAL

A.1 Introduction

This is the user manual for the parallel implementation of Regular Expression Transformation Algorithms on the Intel's iPSC/2 32-node Hypercube machine. The programs are written in the standard C language (except for extensions supported by the iPSC/2 System for message-passing capabilities). The programs have been compiled on the C-386 compiler and have been executed on the iPSC/2 System with varying number of processors. The programs are compatible with the Green Hill Compilers (except for the extensions supported by the iPSC/2 System).

A major objective of this programming project is to study the performance issues of some multiprocessor scheduling algorithms. The project also attempts to study the changing form of a Regular Expression subjected to a set of transformations. More details on RE transformations, Scheduling Algorithms, etc., can be obtained from the various chapters and appendixes of the report.

This project is divided into two main modules: a host module which is a collection of routines that run on the host computer, and a node module which is a collection of routines that run on all of the nodes (processors) of the iPSC/2 System. The host module and the node module are designed in a way that avoids much of the synchronization between them except for an initial and a final item of information to be communicated between them. On the other hand, the programs in the node module co-ordinate and synchronize their execution through communication with each other appropriately.

A.2 Description of the Host Module

The input to the host module is a regular expression (RE) of arbitrarily long size and its input symbol set (i.e., alphabet). The host module partitions this RE into a set of tasks and represents it as a task system in the form of a graph. Two partitioning approaches are adopted. The first approach partitions the RE such that repeated common sub-expressions are identified as distinct tasks. This approach always yields a representation for the task system in the form of a "rooted tree". The second approach of partitioning considers the repeated common sub-expressions and identifies all of them into a single task. This approach always yields a representation for the task system in the form of a DAG with some nodes possibly having multiple successors.

When the former partitioning approach is chosen (yielding a rooted tree for the partitioned RE), the host module schedules the tree on an arbitrary number, say p , of processors using the scheduling Algorithm A to produce the schedule in the form of a Gantt chart. When the latter approach is chosen (yielding a DAG with nodes having multiple successors), the host module schedules this DAG using Algorithm C to produce the schedule in the form of a Gantt chart.

Once the schedule has been obtained in the form of a Gantt chart, performance measures like "serial execution time", "parallel execution time", "speed-up", "efficiency", and "serial fraction" are evaluated. Subsequently, the host module sends the initial information in the form of a Gantt-chart and the task graph (either rooted tree or DAG) to the node module to initiate its execution. After the node module completes execution, the host module receives information from the node module on the timing parameters.

The host module is divided into sub-modules each corresponding to the functions described above. Each sub-module is contained in a separate file as described below.

File Name	Description
host.c	The driver routine for the host module. Also, calculates and tabulates all the results for Appendixes.
partn.c	Contains the two partitioning approaches, namely, partn_A for Algorithm A and partn_B for Algorithm C.
schedule.c	Obtains the schedule in the form of a Gantt chart using Algorithm A or C, and performs the needed optimizations as described in Section 4.4.
misc.c	Miscellaneous functions.

A.3 Description of the Node Module

The node module does the core computation for this package. The node module comprises a set of nodes (processors) each running the same program. All the nodes work together and synchronize their execution through communication with each other. This module is initiated by the information sent by the host module and subsequently it does not need any more information from either the host module or the user to complete its execution. After completion of its execution, the node module sends information back to the host module for calculating the practical results.

The node module is divided into a set of sub-modules. Each sub-module, contained in a separate file as described below, corresponds to one of the transformation depicted in Figure 8 of Section 3.4.

File Name	Transformation	Description	Nodes
makenfa.c	T1	Converts RE to an NFA with e-moves	All
ecl.c	T2	Removes e-moves in the NFA	All
nfa_dfa.c	T3	Converts the NFA to a DFA	Root
min.c	T4	Minimizing the DFA	Root
dfa_re.c	T5	DFA to RE	All
eqns.c	T6	RE eqns for the DFA	Root

	T7	Solution of RE eqns	Root
other.c	-	Miscellaneous routines	All

Some of the transformations are performed in parallel on all the nodes. transformation T1 is executed by all the nodes, but the final NFA is available only on the the root node which then sends the NFA to all other nodes. Subsequently, transformations T2 and T5 are performed on all nodes, but transformations T3, T4, T6, and T7 are performed only on the root node. At the end of the cycle of transformations, the final RE along with the timing information is sent to the host module by the root node.

A.4 Variables Influencing Execution

Some factors which affect the execution of the program are discussed here. All these factors concern the constant definitions included in the header files (host.h in the host module and node.h in the node module). The user should choose the values of these constants appropriately (depending on the size of RE, size of NFA to be produced, and the implementatin machine) before executing the program.

DEBUG: Flag to obtain the debug output of the program.

ALG_A: Choice of Algorithm A which will be solicited from the user.

ALG_B: Choice of Algorithm B which will be solicited from the user.

ALG_C: Choice of Algorithm C which will be solicited from the user.

MAX_NFA: Maximum number of states in the NFA to be synthesized. This number needs to be chosen so as not to waste memory, since the transtion table for the NFA depends on MAX_NFA.

MAX_COLS: This variable should be set to 1 more than the actual input alphabet used.

MAX_DFA: Maximum number of states in the DFA (analogous to MAX_NFA).

PHI: This variable represents the PHI task in an NFA or a DFA and should be set to the value of the maximum state in the FA, that is, MAX_NFA or MAX_DFA.

MAX_RE: Maximum size of the RE. This size should be long enough to accomodate the divergence of the RE size observed sometimes in the "divergent" cycle.

MAX_PROCS: Maximum number of processors to be used.

MAX_LEVELS: Maximum length of the schedule as given in the Gantt chart.

MAX_TASKS: Maximum number of tasks that can be present in the task graph produced by either of the partitioning approaches.

CONVERG: Choice of Convergence as solicited from the user.

DIVERG: Choice of Divergence as solicited from the user.

A.5 Steps to Execute the program

The sequence of steps to be taken for the execution of the program are as follows.

1. The source code for the program is in the *thesis* directory. Change to the *thesis* directory and list contents to check that the following files are available.

<i>makefile</i>	<i>README</i>			
<i>host.h</i>	<i>host.c</i>	<i>partn.c</i>	<i>schedule.c</i>	<i>misc.c</i>
<i>node.h</i>	<i>node.c</i>	<i>makenfa.c</i>	<i>nfa_dfa.c</i>	<i>min_dfa.c</i>
<i>dfa_re.c</i>	<i>eqns.c</i>	<i>other.c</i>		

2. Issue the command:

make

"make" needs one of the following options:

make cx	to use 386 nodes with 387 coprocessors
make sx	to use 386 nodes with SX coprocessors
make rx	to use i860 nodes
make host	to build only the host executable
make node	to build only the node executable
make clean	to cleanup the directory of unwanted files
make make	to view and/or edit the makefile

3. Issue the command:

make make

to view (or possibly edit) the makefile. The user might want to change the CC, CFLAGS, or LDFLAGS options. For example, if the Green Hill Compiler needs to be invoked:

change CC to CC=gcc

If the user wants to debug the program, to use the profiling information, or to invoke particular compiler optimizations, the CFLAGS and/or LDFLAGS need to be modified.

4. Issue the comand:

make clean

to remove all previous executable files and make a fresh start.

5. Now list the directory contents and ensure that only the files listed in Step 1 are present.

6. Issue the command:

make cx (or sx or vx depending on the type of the node)

The executables "host" and "node" are created. The user need not allocate a cube to execute the program, since a suitable cube is obtained by the HOST MODULE.

7. Before executing the program, enter the RE to be tested in a datafile. Execute the program by issuing the command:

host

The following user interface is produced:

Do you want:

1. Details of the form changes of an RE
2. Details of the scheduling algorithms

Enter your option: 2

Choose option 1 if you need to study the changing form of an RE subjected to the cycle of transformations for a specific number of times. No timing information on scheduling algorithms is produced when this option is chosen. Another submenu will seek the user's choice of convergent or divergent cycle of transformations to be used. The form changes are reported in a file called "app_D" for the convergent cycle, and "app_E" for divergent cycle.

Choose option 2 if you do not want the form changes of an RE, but instead need the timing parameters and the performance measures of the scheduling algorithms. Performance measures are reported for varying number of processors (up to a maximum equal to the width of the graph) in a file called app_F. Note that convergent cycle is always used for studying the scheduling algorithms.

Do you want:

1. Theoretical Results only
2. Practical Results only
3. Both

Enter your option: **1**

Choose option 1 if only theoretical measures (obtained from the Gantt chart) are required. The REs are not actually evaluated through execution on the hypercube. The measures are obtained directly from the schedule

Choose option 2 if only practical measures are needed. The REs are now evaluated through execution on the nodes of the hypercube.

Choose option 3 if both theoretical and practical measures are needed. This option is used to compare the theoretical results with the practical results for either Algorithms A, C, or for both.

Do you want to use:

1. Scheduling Algorithm A
(Note: Partitioning Approach A will be used.)
2. Scheduling algorithm C
(Note: Partitioning Approach B will be used.)
3. Both
(Note: Partitioning Approach A for Algorithm A and Approach B for Algorithm C will be used.)

Enter your option: **1**

This submenu seeks your choice of the scheduling algorithm to be used for scheduling the task graph (the partitioned RE). If option 3 is chosen, each RE is evaluated by Algorithms A and C. Option 3 is used to compare performance measures of Algorithms A and C.

Enter the input alphabet: **01**

This submenu seeks the input alphabet from the user. The user should make sure that the RE(s) fed as input contain only symbols from this alphabet set. The symbol "e" (denoting epsilon) need not be contained in the alphabet as it is assumed to be contained in every alphabet. Also, the user should set the MAX_COLS constant in both the header files to 1 more than the size of the alphabet. For example,

if alphabet is "01"
MAX_COLS should be set to 3 (to include epsilon)

Do you want details for:

1. A single RE
2. Multiple REs

Enter your option: **2**

Enter the name of the datafile: **data.1**

This submenu inquires whether the user wants to execute the program for only one RE or for multiple REs. If option 1 is chosen, the user enters the RE directly on the terminal. If option 2 is chosen, the program reads the REs from a file specified by the user. This reduces the burden of creating a file just for a single RE or entering multiple REs at the terminal.

APPENDIX B

C PROGRAMS

MAKEFILE

```
#
#
# This file is used to compile and link the host_module and
# node_module files for the thesis program of Sridhar Mandyam.
#
CFLAGS=-w -O -B
help:
    @echo
    @echo "You must specify the type of node you wish to build a node"
    @echo "executable for, choose one of the following:"
    @echo
    @echo "    make cx      (for 386 nodes with 387 coprocessors)"
    @echo "    make sx      (for 386 nodes with SX coprocessors)"
    @echo "    make rx      (for i860 nodes)"
    @echo
cx:   host node                #Use default compile and link flags
sx:
    make "CFLAGS=-w -O -B" host
    make "CFLAGS=-w -O -B -sx" "LDFLAGS=-sx" node
rx:
    make "CFLAGS=-w -O -B" host
    make "CFLAGS=-w -O -B -i860" "LDFLAGS=-i860" node
host: host.o partn.o schedule.o misc.o host.h
    cc -o host host.o partn.o schedule.o misc.o -host
host.o partn.o schedule.o misc.o: host.h
node: node.o makenfa.o ecl.o nfa_dfa.o min_dfa.o dfa_re.o eqns.o other.o node.h
    cc -o node node.o makenfa.o ecl.o nfa_dfa.o min_dfa.o dfa_re.o eqns.o other.o
$(LDFLAGS) -node
node.o makenfa.o ecl.o nfa_dfa.o min_dfa.o dfa_re.o eqns.o other.o: node.h
make:
    vi makefile
clean:
    rm host node host.o node.o makenfa.o ecl.o nfa_dfa.o min_dfa.o dfa_re.o
    eqns.o other.o host.o partn.o schedule.o misc.o
```

HOST.H

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

/**      Scheduling Constants      ***/
#define MAX_RE      250 /* Maximum size of RE */
#define MAX_COLS    3  /* Maximum size of the input alphabet */
#define MAX_PROCS   16 /* Maximum number of processors */
#define MAX_LEVELS  100 /* Maximum number of levels in the graph */
#define MAX_TASKS   100 /* Maximum number of tasks in the graph */
#define MAX_SUCC    15 /* Maximum successors of a task */
#define PHI_TASK    ' ' /* Symbol for phi task in the Gantt chart */

/**      General Header Constants      ***/
#define TRUE        1
#define FALSE       0#define SENTINEL -1
```

```

#define          DEBUG    FALSE    /* To print debug information */
/**** Partitioning Constants - DONOT CHANGE THESE CONSTANTS ****/
#define          MIN_SYMB  0        /* Range for number alphabets */
#define          MAX_SYMB  90       /* Range for number of tasks */
#define          BEG_TASK  100      /* Range for number of tasks */
#define          END_TASK 1000
#define          CLOSURE  MAX_SYMB + 1 /* symbol of Closure operation */
#define          CONCAT  MAX_SYMB + 2 /* symbol of Concatenation operation */
#define          UNION   MAX_SYMB + 3 /* symbol of Union operation */
#define          EPSILON MAX_SYMB
/**** Type definitions ****/

typedef int boolean;
typedef unsigned short SHORT;
typedef unsigned char STTYPE;
typedef struct { /* Type for a task in the graph */
    int      label,          /* task's label */
          node,            /* task's processor */
          wt,              /* task's weight */
          status,
          done;            /* if task has completed execution */
    STTYPE  succ[MAX_SUCC], /* task's successors */
          task[3];         /* task's operation */
} Task_tree;
/* Type for the label table of the task graph */
typedef struct Label_struc {
    SHORT value;
    struct Label_struc *next;
} Label_list;

```

HOST.C

```

#include "host.h"
#define DEBUG 1
/*****
This file (host.c) contains the following HOST routines:
    main()           The driver routine for the HOST MODULE
    evaluate_re()    Evaluates an RE by partitioning and scheduling
    send_nodes()     Sends information to the NODE MOUDLE
    measures()       Calculates the theoretical and practical results
    Some printing routines
The external routines called from this file include:
    chk_re()         in      misc.c file
    in_to_post()     in      misc.c file
    partn_A(), partn_C() in    partn.c file
    algorithm_A(), algorithm_C() in  schedule.c file
    Get_schedule()  in      schedule.c file
*****/

#define          CONVERG  1        /* Convergent cycle */
#define          DIVERG  2        /* Divergent cycle */
#define          ALG_A    1        /* option for Algorithm A */
#define          ALG_C    2        /* option for Algorithm C */
#define          TH       1        /* option for theoretical results */
#define          PRACT    2        /* option for practical results */
#define          SINGLE   1        /* Single RE for execution */
#define          MULTIPLE 2        /* REs taken from file for execution */
#define          BOTH     3

#define          HOSTPID  100     /* process id for host process */
#define          NODEPID  0       /* process id for node process */
#define          ALLNODES -1      /* symbol for all nodes */
#define          ALLPIDS  -1      /* symbol for all processes */
#define          INIT_TYP 10      /* type of host to node message */
#define          RE_TYP   100     /* type of RE message to be received */

/**** Declaration of All Global Variables for the host program ****/
int      max_terms, /* Number of terms in the RE = the # of states
                  in the NFA to be synthesized */
    num_tasks, /* number of tasks in the graph */
    num_levels, /* number of levels in the graph */
    num_procs=1, /* number of processors to run the problem */
    iter, /* number of iterations through the cycle of
          transformations */
    main_ch, /* main choice of
            study in the program */

```

```

        res_ch,          /* Need theoretical and/or practical results */
        alg_ch,         /* choice of scheduling Algorithm A or C */
        form_ch,       /* choice of Convergent or Divergent cycle */
        re_ch;         /* single or multiple REs for study */
char   infile[50],     /* string for input file name */
        re[MAX_RE],
        oldre[MAX_RE],
        post[MAX_RE],
        symb_set [MAX_COLS];

Task_tree      tlist [MAX_TASKS];
Label_list     *l_list [MAX_LEVELS];
STTYPE         G_chart [MAX_PROCS] [MAX_LEVELS];

/*****
Function Definition: int main( void )
Description:
This is the driver routine for the HOST MODULE. It obtains an RE
preprocesses it and then partitions it into a suitable task graph. One
of Algorithms A or C is chosen to schedule the task graph. Once the
schedule is obtained in a Gantt chart form it is sent to various nodes
of the cube for their execution. Finally the host program collects
the data from the nodes which include the final RE and some timing
parameters. The RE thus obtained at the end of one cycle of
transformations is subjected again to the cycle as many times as required.
*****/
main()          /* MAIN of the host program */
{
    char       cubetyp[6];
    int        count,          /* current iteration */
            numre=0;         /* number of REs processed */
    boolean    quit = FALSE;

    /* declaration of functions used in main */
    extern int   chk_re();
    extern void  in_to_post();
    void       get_input(), evaluate_re();

FILE   *infp,          /* input file */
        *ofp2,
        *ofp2a,       /* file for Appendix D - Convergence details */
        *ofp2b,       /* file for Appendix E - Divergence details */
        *ofp3a,       /* file for Appendix F - theoretical results */
        *ofp3b;      /* file for Appendix G - practical results */

    get_input();

    if (re_ch != SINGLE)
    if ((infp=fopen(infile,"r")) == NULL) {
        printf("Sorry cannot open %s file \n",infile) ;
        exit(1);
    }
    if ((ofp2a=fopen("app_D","w")) == NULL) {
        printf("Sorry cannot open Appendix DE - Conv/Div file \n") ;
        exit(1);
    }
    if ((ofp2b=fopen("app_E","w")) == NULL) {
        printf("Sorry cannot open Appendix DE - Conv/Div file \n") ;
        exit(1);
    }
    if ((ofp3a=fopen("app_F","w")) == NULL) {
        printf("Sorry cannot open Appendix F - Results file \n") ;
        exit(1);
    }
    if ((ofp3b=fopen("app_G","w")) == NULL) {
        printf("Sorry cannot open Appendix G - Results(P) file \n") ;
        exit(1);
    }
    fprintf(ofp3a, "\nNOTE: All tasks have unit execution times \n");
    fprintf(ofp3a, "          Width specifies the maximum number of processors");
    fprintf(ofp3a, "          that will be used\n\n");
    fprintf(ofp3b, "\nNOTE: All tasks have unit execution times \n");
    fprintf(ofp3b, "          Width specifies the maximum number of processors");
    fprintf(ofp3b, "          that will be used\n\n");
    printf("Input symbol set used: %s \n",symb_set);

```



```

printf("\tNumber of Iterations = %d \n",iter);
printf("\tChoice is ");
if (form_ch == CONVERG)
    printf(" Convergence\n");
if (form_ch == DIVERG)
    printf(" Divergence\n");
if (form_ch == CONVERG) ofp2 = ofp2a;
if (form_ch == DIVERG) ofp2 = ofp2b;
/* print the data into a file */
fprintf(ofp2,"Input symbol set used: %s \n",symb_set);
fprintf(ofp2,"\tNumber of Iterations = %d \n",iter);
fprintf(ofp2,"\tChoice is ");
if (form_ch == CONVERG)
    fprintf(ofp2," Convergence\n");
if (form_ch == DIVERG)
    fprintf(ofp2," Divergence\n");
/* form the cubetyp string depending on num_procs */
cubetyp[0] = '0'+num_procs/10;
cubetyp[1] = '0'+num_procs%10;
cubetyp[2] = '\0';
strcat(cubetyp,"m8");
getcube("susri",cubetyp,NULL,0);
/* get a cube named 'susri' with the requested nodes */
setpid(HOSTPID);
/* set host process id */
load("nodedir/node",ALLNODES,NODEPID);
/* load all nodes with pid NODEPID */

while (!quit) { /* process all REs */
    if (re_ch == MULTIPLE)
        fscanf(infp,"%s\n",re);
    printf("\nRE Number: %d \t****\tRE: %s\n",numre++,re);
    count = 0;
    /* Subjecting the given RE to the cycle of transformations 'iter'
       number of times */
    while (count < iter) {
        fprintf(ofp2,"\n\tITERATION%d\n",count);
        fprintf(ofp2,"RE at the beginning of the cycle: %s\n",re);
        fprintf(ofp3a,"RE = %s\n",re);
        fprintf(ofp3b,"RE = %s\n",re);
        printf("\nThe given regular expression is %s\n",re);
        strcpy(olddre,re);

        /* check RE and get the approximate estimate of the number of
           states in the NFA to be synthesized */
        if ((max_terms=chk_re(re,symb_set)) != 0) {
            in_to_post(post,re); /* convert to post fix notation */
            printf("\nThe post-fix expression is: %s\n",post);
            printf("Estimated # of states in NFA = %d\n",max_terms);

            header(ofp3a);
            header(ofp3b);

            /* partition RE and obtain the schedule */
            if (alg_ch == BOTH) {
                evaluate_re(ofp3a,ofp3b,ALG_A);
                fprintf(ofp3a,"-----");
                fprintf(ofp3a,"-----\n");
                fprintf(ofp3b,"-----");
                fprintf(ofp3b,"-----\n");
                evaluate_re(ofp3a,ofp3b,ALG_C);
            }
            else evaluate_re(ofp3a,ofp3b,alg_ch);

            fprintf(ofp3a,"-----");
            fprintf(ofp3a,"-----\n");
            fprintf(ofp3b,"-----");
            fprintf(ofp3b,"-----\n");
            printf("\n\n");
            fprintf(ofp2,"RE at the end of the cycle: %s \n",re);
        }
        else {
            printf("\nSorry - Invalid RE %s \n",olddre);
            printf("Skipping to next RE \n");
        }
    }
}

```

```

    }
    count++;
} /* inner while */
fprintf(ofp2, "*****\n\n");
if (re_ch == SINGLE)
    quit = TRUE;
if (re_ch == MULTIPLE && feof(infp))
    quit = TRUE;
} /* while !quit */
killcube(ALLNODES, ALLPIDS);
/* kill all processes on all nodes */
relcube("susri");
/* release the allocated cube */
if (re_ch == MULTIPLE)
    fclose(infp);
fclose(ofp2a);
fclose(ofp2b);
fclose(ofp3a);
fclose(ofp3b);
} /* end of host main */

```

```

/*****
Prototype Definition: void evaluate_re(FILE *, FILE *, int )

```

Description:

This routine evaluates an RE in post fix form (similar to evaluating an arithmetic expression). The RE is initially partitioned and represented as a Task Graph. Then, it is scheduled by scheduling Algorithm A or C to obtain the Gantt chart. This schedule is sent to the nodes for subjecting the RE to a cycle of transformations. The final RE and timing parameters are received from the nodes.

```

*****/

```

```

void evaluate_re(ofp3a, ofp3b, alg_ch)
FILE *ofp3a, *ofp3b;
int    alg_ch;      /* choice of Algorithm A or C */
{
    int    nlevels,
           width;      /* width of the task graph */

    /* function prototypes for this routine */
    extern int    init_labels();
    extern void   partn_A(), partn_C(), algorithm_A(), algorithm_C(),
                print_schedule(), free_llist(),
                Get_schedule();
    void    measures(), send_nodes(),
            print_llist();
    if (alg_ch == ALG_A) partn_A(symb_set, post);
    else if (alg_ch == ALG_C) partn_C(symb_set, post);
    /* Get the initial set representation of tasks and the width of the
       graph from the label table. */
    width = init_labels(num_tasks);
    if (DEBUG) {
        printf("\n\n\t INITIAL LABEL TABLE \n\n");
        print_llist();
        /* print the initial set representation */
    }

    if (main_ch == 1) {
        num_procs = width;
    }
    else if (main_ch == 2)
        num_procs = 1;

    nlevels = num_levels;
    while (num_procs <= width) {
        if (alg_ch == ALG_A) algorithm_A(num_procs);
        if (alg_ch == ALG_C) algorithm_C(num_procs);
        if (DEBUG) {
            printf("\n\n\t ADJUSTED LABEL TABLE \n\n");
            print_llist();
        }
        /* get the schedule from the final configuration of label table */
        if (DEBUG) {
            printf("\n\nSchedule Obtained by Algorithm");

```

```

if (alg_ch == ALG_A)
    printf(" A ");
if (alg_ch == ALG_C)
    printf(" C ");
printf("BEFORE Optimization for p=%d Processors\n",num_procs);
}
Get_schedule(G_chart,num_procs);
if (DEBUG) {
    printf("\n\nSchedule Obtained by Algorithm");
    if (alg_ch == ALG_A)
        printf(" A ");
    if (alg_ch == ALG_C)
        printf(" C ");
    printf("AFTER Optimization for p=%d Processors\n",num_procs);
    print_schedule(G_chart,num_procs,num_levels);
}
free_llist();
if (res_ch == TH || res_ch == BOTH ) {
    measures(ofp3a,TH,alg_ch,width);
}
if (main_ch == 1 || res_ch == PRACT || res_ch == BOTH) {
    send_nodes(form_ch);
    /* send required information to all nodes */
    crecv(RE_TYP,re,MAX_RE*sizeof(char));
    /* Block to receive final message from root node. */
}
if (res_ch == PRACT || res_ch == BOTH)
    measures(ofp3b,PRACT,alg_ch,width);
if (main_ch == 1)
    num_procs = width+1;
else if (main_ch == 2) {
    num_levels = nlevels;
    width = init_labels(num_tasks);
    num_procs++;
} /* else if */
} /* while */
} /* end schedule_re */

/*****
Function Definition: void measures( FILE *, int, int, int )
Description:
This function calculates all the theoretical performance measures namely
the Schedule length, Speed-up, Efficiency, and Serial Fraction from the
schedule obtained in G_chart structure.
*****/
void measures(ofp3,typ,choice,width)
FILE *ofp3;
int typ;
int choice;
int width;
{
    int    par_tm,        /* parallel time or schedule length */
          ser_tm,        /* serial time = sum of task weights */
          float sp_up,   /* speed-up factor */
          efficiency,    /* efficiency factor */
          serial;        /* serial fraction factor */

/*
    if (num_procs == 1) {
        ser_tm = final_msg->tm;
        return;
    }
*/
/* since all are unit tasks, serial time = number of tasks */
if (typ == TH)
    ser_tm = num_tasks;

/*
else if (typ == PRACT)
    ser_tm = time->serial;
*/

/* par_tm is same as schedule length which is the number
of levels in the adjusted (stretched) task graph */

```

```

    if (typ == TH)
        par_tm = num_levels;
/*
    else if (typ == PRACT)
        par_tm = time->par;
*/

/* speed-up is the ratio of execution on one processor (sum of all
   task weights) to the execution time on p processors */
sp_up = (float) ser_tm/par_tm;

/* efficiency is the ratio of speed-up to number of processors */
efficiency = sp_up/num_procs;

/* serial fraction is given by, */
serial = (1.0/sp_up - 1.0/num_procs)/(1.0-1.0/num_procs);

/* print all the performance measures */
if (choice == ALG_A) fprintf(ofp3, "    A");
if (choice == ALG_C) fprintf(ofp3, "    C");
fprintf(ofp3, "        %4d    %4d    %2d", width, num_procs, ser_tm);
fprintf(ofp3, "        %2d    %6.3f    %6.3f\n", par_tm, sp_up, efficiency);
fprintf(ofp3, "_____ \n");
}

```

```

/*****
Function Name: void send_nodes( int )

```

Description:

Sending the required information to the nodes for the initiation of their execution. The info sent is the task graph in tlist struc and the schedule in G_chart array. Synchronous send is used here.

```

*****/
void send_nodes(form_ch)
int form_ch;

```

```

{
    int    i,j;
    long   len;
    struct msg_typ {
        int          /* structure for host message */
        work_nodes, /* # of user requested nodes */
        numtasks,   /* # of tasks in task tree */
        numlevels,  /* # of levels in the tree */
        maxterms,   /* max # of states in the NFA */
        form_choice; /* CONVERGENCE or DIVERGENCE ? */
        char         symbset[MAX_COLS]; /* input alphabet set */
        STTYPE       G_chart[MAX_PROCS][MAX_LEVELS];
        Task_tree    tlist[MAX_TASKS]; /* schedule in Gantt chart form */
    } init_msg; /* Task graph structure */

    /* copy all information to be sent into the init_msg structure */
    init_msg.work_nodes = num_procs;
    init_msg.numtasks = num_tasks;
    init_msg.numlevels = num_levels;
    init_msg.maxterms = max_terms;
    init_msg.form_choice = form_ch;
    strcpy(init_msg.symbset, symb_set);

    /* copy the Task tree to the init_msg structure */
    for (i=0; i<num_tasks; i++)
        init_msg.tlist[i] = tlist[i];
    /* copy the Gantt chart to the init_msg structure */
    for (i=0; i<num_procs; i++)
        for (j=0; j<num_levels; j++)
            init_msg.G_chart[i][j] = G_chart[i][j];

    len = sizeof(struct msg_typ);
    printf(" Sending message - Type=%d, Length=%d PID=%d\n",
           INIT_TYP, len, NODEPID);

    /* send init_msg struc to ALLNODES with msg_type INIT_TYP */
    csend(INIT_TYP, &init_msg, len, ALLNODES, NODEPID);
}

```

```

)*****/
Prototype Definition: void get_input( void )

```

Description:

This is the routine which provides the user interaction and obtains all the input required from the user for the execution of the program.

```
void get_input()
{
    main_ch = alg_ch = res_ch = re_ch = form_ch = 0;
    do {
        printf("\nDo you want:\n");
        printf("\t1. Details of Form Changes of an RE\n");
        printf("\t2. Details of Scheduling Algorithms \n");
        printf("\n\t\tEnter your option: ");
        scanf("%d",&main_ch);
        if (main_ch!=1 && main_ch!=2)
            printf("\nIncorrect option - Enter again \n\n");
    } while (main_ch != 1 && main_ch != 2);
    if (main_ch == 1) {
        printf("\nEnter Number of Iterations to be used: ");
        scanf("%d",&iter);
        printf("\n");
    }
    else if (main_ch == 2)
        iter = 1;

    if (main_ch == 1)
    do {
        printf("\nDo you want to use:\n");
        printf("\t1. Convergent Cycle \n");
        printf("\t2. Divergent Cycle \n");
        printf("\n\t\tEnter your option: ");
        scanf("%d",&form_ch);
        if (form_ch!=1 && form_ch!=2)
            printf("\nIncorrect option - Enter again \n\n");
    } while (form_ch != 1 && form_ch != 2);
    else if (main_ch == 2)
        form_ch = CONVERG;

    if (main_ch == 2)
    do {
        printf("\nDo you want:\n");
        printf("\t1. Theoretical Results only \n");
        printf("\t2. Practical Results only\n");
        printf("\t3. Both \n");
        printf("\n\t\tEnter your option: ");
        scanf("%d",&res_ch);
        if (res_ch!=1 && res_ch!=2 && res_ch!=3)
            printf("\nIncorrect option - Enter again \n\n");
    } while (res_ch != 1 && res_ch != 2 && res_ch!=3);
    else if (main_ch == 1)
        res_ch = 0;

    do {
        printf("\nDo you want to choose: \n");
        printf("\n\t1. Scheduling Algorithm A \n");
        printf("\t\t\tNote: Partitioning Approach A will be used \n");
        printf("\n\t2. Scheduling algorithm C \n");
        printf("\t\t\tNote: Partitioning Approach B will be used \n");
        printf("\n\t3. Both \n");
        printf("\t\t\tNote: Partitioning Approach A for Algorithm A \n");
        printf("\t\t\t\t\tand Approach B for Algorithm C will be used \n");
        printf("\n\t\tEnter your option: ");
        scanf("%d",&alg_ch);
        printf("\n");
        if (alg_ch!=1 && alg_ch!=2 && alg_ch!=3)
            printf("\nIncorrect option - Enter again \n\n");
    } while (alg_ch != 1 && alg_ch != 2 && alg_ch != 3);

    /* read the input symbol set from file */
    printf("\nEnter the input alphabet: ");
    scanf("%s", symb_set);
    printf("\nDo you want details for: \n");
    printf("\t1. A single RE \n");
    printf("\t2. Multiple REs\n");
    printf("\n\t\tEnter your option: ");
    scanf("%d",&re_ch);
}
```

```

    if (re_ch == SINGLE) {
        printf("\nEnter the input RE: ");
        scanf("%s",re);
        printf("\n");
    }
    if (re_ch == MULTIPLE) {
        printf("\nEnter the name of the datafile: ");
        scanf("%s",infile);
        printf("\n");
    }
} /* end get_input */

header(ofp3)
FILE *ofp3;
{
    fprintf(ofp3, "_____");
    fprintf(ofp3, "_____ \n");
    fprintf(ofp3, "-----");
    fprintf(ofp3, "----- \n");
    fprintf(ofp3, "Algorithm Width Processors Serial Parallel ");
    fprintf(ofp3, "Speed Efficiency\n");
    fprintf(ofp3, "                time        time        up\n");
    fprintf(ofp3, "_____");
    fprintf(ofp3, "_____ \n");
    fprintf(ofp3, "-----");
    fprintf(ofp3, "----- \n");
}

```

```

/*****
Function Definition: void print_llist( )

```

Description:

This routine prints the label table of the Task Graph in the form of sets of tasks with a particular label (which corresponds to the set representation of tasks).

```

*****/
void print_llist()

```

```

{
    int i;
    Label_list *ptr1;
    printf("\tMax levels: %d\n\n",num_levels);

    printf("----- \n");
    printf("Num Tasks |      Tasks      \n");
    printf("----- \n");

    for (i=0;i<num_levels;i++) {
        ptr1 = l_list[i];
        printf(" %d | ",ptr1->value);
        while (ptr1->next != NULL) {
            ptr1 = ptr1->next;
            printf(" T%d",ptr1->value);
        }
        printf("\n");
    }
    printf("----- \n");
}

```

PARTN.C

```

#include "host.h"
#define DEBUG 1
/*****
This file (partn.c) contains the following HOST routines:
    partn_A      Partitioning approach for Algorithm A
    partn_B      Partitioning approach for Algorithm B
    Some printing routines
*****/

```

```

typedef struct xx {
    unsigned short symb;
    struct xx *next;
} RE_list;

```



```

else if (symb>MAX_SYMB && symb<BEG_TASK) {
/* if the current symbol is an OPERATOR, then we try various cases
depending on the two operands */
if (((symb1>=MIN_SYMB && symb1<=MAX_SYMB) || symb1==EPSILON) &&
((symb2>=MIN_SYMB && symb2<=MAX_SYMB) || symb2==EPSILON)) {
/* if both the operands are ALPHABETS, then we have an
independent task. Store this task in the task table */
tlist[t_num].done = FALSE;
tlist[t_num].node = t_num;
tlist[t_num].wt = 0;
strcpy(tlist[t_num].succ, "");
tlist[t_num].task[0] = n2->symb;
tlist[t_num].task[1] = curr->symb;
tlist[t_num].task[2] = n1->symb;
tlist[t_num].label = num_levels;

curr->symb = t_num++ + BEG_TASK;
curr->next = n2->next;
curr = curr->next;

n2->next = NULL; n1->next = NULL;
junk1 = n1; junk2 = n2;
free(junk1); free(junk2);
}
else if (symb1>MAX_SYMB && symb1<BEG_TASK) {
/* if one of the operands is an operator, move up the list */
curr = n1;
}
else if (symb2>MAX_SYMB && symb2<BEG_TASK) {
/* if the other operand too is an operator, move up the list */
curr = n2;
}
else if (((symb1>=MIN_SYMB && symb1<=MAX_SYMB) &&
(symb2>=BEG_TASK && symb2<=END_TASK)) {
/* first operand is an alphabet and second is a task, then we
have a dependent task. Store the task and all its
information in the task table */
tlist[t_num].done = FALSE;
tlist[t_num].node = t_num;
tlist[t_num].wt = 0; /* set node weightt to 0*/
strcpy(tlist[t_num].succ, "");
strcat(tlist[n2->symb-BEG_TASK].succ, itos(t_num));
tlist[t_num].task[0] = n2->symb;
tlist[t_num].task[1] = curr->symb;
tlist[t_num].task[2] = n1->symb;
tlist[t_num].label = num_levels;

curr->symb = t_num++ + BEG_TASK;
curr->next = n2->next;
curr = curr->next;

n1->next = NULL; n2->next = NULL;
junk1 = n1; junk2 = n2;
free(junk1); free(junk2);
}
else if (((symb2>=MIN_SYMB && symb2<=MAX_SYMB) &&
(symb1>=BEG_TASK && symb1<=END_TASK)) {
/* n2 operand is a symbol and n1 is a node */
tlist[t_num].done = FALSE;
tlist[t_num].node = t_num;
tlist[t_num].wt = 0;
strcpy(tlist[t_num].succ, "");
strcat(tlist[n1->symb-BEG_TASK].succ, itos(t_num));
tlist[t_num].task[0] = n2->symb;
tlist[t_num].task[1] = curr->symb;
tlist[t_num].task[2] = n1->symb;
tlist[t_num].label = num_levels;

curr->symb = t_num++ + BEG_TASK;
curr->next = n2->next;
curr = curr->next; n1->next = NULL; n2->next = NULL;
junk1 = n1; junk2 = n2;
free(junk1); free(junk2);
}
else if (((symb1>=BEG_TASK && symb1<=END_TASK) &&

```



```

        (symb2>=BEG_TASK && symb2<=END_TASK)) {
/* if both operands are nodes */
tlist[t_num].done = FALSE;
tlist[t_num].node = t_num;
tlist[t_num].wt = 0;
strcpy(tlist[t_num].succ,"");
strcat(tlist[n1->symb-BEG_TASK].succ,itos(t_num));
strcat(tlist[n2->symb-BEG_TASK].succ,itos(t_num));
tlist[t_num].task[0] = n2->symb;
tlist[t_num].task[1] = curr->symb;
tlist[t_num].task[2] = n1->symb;
tlist[t_num].label = num_levels;

curr->symb = t_num++ + BEG_TASK;
curr->next = n2->next;
curr = curr->next;

n1->next = NULL; n2->next = NULL;
junk1 = n1; junk2 = n2;
free(junk1); free(junk2);
}
} /* end of else if */

if (curr == NULL || curr->next == NULL ||
curr->next->next == NULL) {
num_levels++;
if (DEBUG) print_RElist(start,num_levels,symb_set);
curr = start;
}
} /* end of while */

if (DEBUG)
printf("\nLast task is T%d which represents the root node\n\n",t_num-1);
/* While partitioning above tasks are assigned labels corresponding to
the level it appeared first in the rooted tree. Now, we adjust the
label such that the label of every task differs exactly by 1 from
its successor */
for (i=0;i<t_num;i++) {
if (no_preds(i)) /* Task i has no predecessors */
continue;
if (tlist[i].task[0]>=BEG_TASK) { /* Has left predecessor */
lpred = tlist[i].task[0] - BEG_TASK;
if (tlist[i].label != tlist[lpred].label+1) { /* Labels donot */
tlist[lpred].label = tlist[i].label-1; /* differ by 1 */
i = -1;
continue;
}
}
if (tlist[i].task[2] >= BEG_TASK) { /* Has right predecessor */
rpred = tlist[i].task[2] - BEG_TASK;
if (tlist[i].label != tlist[rpred].label+1) { /* Labels do not*/
tlist[rpred].label = tlist[i].label-1; /* differ by 1 */
i = -1;
continue;
}
}
} /* of for loop */

num_tasks = t_num;
for (i=0;i<num_tasks; i++)
tlist[i].label = num_levels-tlist[i].label-1;

if (DEBUG) {
printf("\nTask Graph produced by Partitioning Approach A\n");
printf("^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");
print_tlist();
}
} /* end of Partn_RE */

void print_RElist(start,level,symb_set)
RE_list *start;
int level;
char symb_set[MAX_COLS];
{
int symbol,i=0;
RE_list *ptr;

```

```

printf("\nLevel %d ==> \t",level);
ptr = start;
while (ptr != NULL) {
    symbol = ptr->symb;
    if (symbol == EPSILON)
        printf("e ");
    else if (symbol <= MAX_SYMB)
        printf("%c ",symb_set[symbol]);
    else if (symbol < BEG_TASK) {
        if (symbol == CONCAT)
            printf(". ");
        if (symbol == UNION)
            printf("+ ");
        if (symbol == CLOSURE) {
            ptr = ptr->next;
            printf("** ");
        }
    } /* else */
    else if (symbol >= BEG_TASK)
        printf("T%d ",symbol-BEG_TASK);
    ptr = ptr->next;
} /* while */
printf("\n");
}

/*****
Function Name: void partn_C( void )
Description:
    This routine partitions the RE following the improved approach of
    Partitioning and to yield a DAG with nodes having multiples successors.
    Thus the partn_C routine is used along with the Scheduling Algorithm_B for
    scheduling the DAG thus obtained.
*****/
void partn_C(symb_set,post)
char    symb_set[MAX_COLS],
        post[MAX_RE];
{
    int    i,j,pos=0,
           lpred,rpred,      /* predecessors of task */
           t_num=0,         /* task number */
           maxlabel;
    boolean repeated=FALSE;
    char    ctask[5],
           succ[MAX_TASKS];

    RE_list *start, *curr, *n1, *n2, *junk1, *junk2;
           /* pointers to list which contains the postfix RE */

    /* function prototypes */
    extern char *itos();
    void    print_RElist();
    void    print_tlist();
    int     is_present();

    num_levels = 0;
    /* Convert the RE in postfix form to a linked list for ease of
       processing in this routine. Also form this list as the
       reverse of RE, b'cos Polish alorithm evaluates an expression
       from its end */

    start = curr = (RE_list *) NULL;
    i=strlen(post);
    for (--i; i>=0;i--) {
        if (curr == NULL) /* first element to be created */
            start = curr = (RE_list *) malloc(sizeof(RE_list));
        else if (curr != NULL) { /* element created at the end */
            curr->next = (RE_list *) malloc(sizeof(RE_list));
            curr = curr->next;
        }
        if (post[i] == '/')
            curr->symb = CLOSURE;
        else if (post[i] == '.')
            curr->symb = CONCAT;
        else if (post[i] == '+')
            curr->symb = UNION;
        else if (post[i] == 'e')

```



```

    curr = n2;
}
else if ((symb1>=MIN_SYMB && symb1<=MAX_SYMB) &&
        (symb2>=BEG_TASK && symb2<=END_TASK)) {
    /* first operand is an alphabet and second is a task, then we
       have a dependent task. Store it, establishing its dependency
       in the task table. */

    /* extract current task's operation */
    ctask[0] = n2->symb; ctask[1] = curr->symb;
    ctask[2] = n1->symb; ctask[3] = '\0';

    /* first check if current task is repeated or not */
    if ((rtask=is_present(ctask,t_num)) != SENTINEL) {
        repeated = TRUE;
    }
    else    repeated = FALSE;

    if (!repeated) {
        tlist[t_num].done = FALSE;
        tlist[t_num].node = SENTINEL;
        strcpy(tlist[t_num].succ, "");
        strcat(tlist[n2->symb-BEG_TASK].succ, itos(t_num));
        tlist[t_num].task[0] = n2->symb;
        tlist[t_num].task[1] = curr->symb;
        tlist[t_num].task[2] = n1->symb;
        tlist[t_num].label = num_levels;
    }

    /* replace operation by its task number in the RE list */
    /* and move up the list for next operation */
    if (repeated)
        curr->symb = rtask+BEG_TASK;
    else    curr->symb = t_num++ + BEG_TASK;
    curr->next = n2->next;
    curr = curr->next;

    n1->next = NULL; n2->next = NULL;
    junk1 = n1; junk2 = n2;
    free(junk1); free(junk2);
}
else if ((symb2>=MIN_SYMB && symb2<=MAX_SYMB) &&
        (symb1>=BEG_TASK && symb1<=END_TASK)) {
    /* n2 operand is a symbol and n1 is a node */

    /* extract current task's operation */
    ctask[0] = n2->symb; ctask[1] = curr->symb;
    ctask[2] = n1->symb; ctask[3] = '\0';

    /* first check if current task is repeated or not */
    if ((rtask=is_present(ctask,t_num)) != SENTINEL) {
        repeated = TRUE;
    }
    else    repeated = FALSE;

    if (!repeated) { /* store only if not repeated task */
        tlist[t_num].done = FALSE;
        tlist[t_num].node = SENTINEL;
        strcpy(tlist[t_num].succ, "");
        strcat(tlist[n1->symb-BEG_TASK].succ, itos(t_num));
        tlist[t_num].task[0] = n2->symb;
        tlist[t_num].task[1] = curr->symb;
        tlist[t_num].task[2] = n1->symb;
        tlist[t_num].label = num_levels;
    }

    /* replace operation by its task number in the RE list */
    /* and move up the list for next operation */
    if (repeated)
        curr->symb = rtask+BEG_TASK;
    else    curr->symb = t_num++ + BEG_TASK;
    curr->next = n2->next;
    curr = curr->next;
    n1->next = NULL; n2->next = NULL;
    junk1 = n1; junk2 = n2;
    free(junk1); free(junk2);
}
else if ((symb1>=BEG_TASK && symb1<=END_TASK) &&
        (symb2>=BEG_TASK && symb2<=END_TASK)) {
    /* if both operands are nodes */
    /* extract current task's operation */

```

```

ctask[0] = n2->symb; ctask[1] = curr->symb;
ctask[2] = n1->symb; ctask[3] = '\0';
/* first check if current task is repeated or not */
if ((rtask=is_present(ctask,t_num)) != SENTINEL) {
    repeated = TRUE;
}
else repeated = FALSE;
if (!repeated) {
    tlist[t_num].done = FALSE;
    tlist[t_num].node = SENTINEL;
    strcpy(tlist[t_num].succ, "");
    strcat(tlist[n1->symb-BEG_TASK].succ, itos(t_num));
    strcat(tlist[n2->symb-BEG_TASK].succ, itos(t_num));
    tlist[t_num].task[0] = n2->symb;
    tlist[t_num].task[1] = curr->symb;
    tlist[t_num].task[2] = n1->symb;
    tlist[t_num].label = num_levels;
}

/* replace operation by its task number in the RE list */
/* and move up the list for next operation */
if (repeated)
    curr->symb = rtask+BEG_TASK;
else curr->symb = t_num++ + BEG_TASK;
curr->next = n2->next;
curr = curr->next;

n1->next = NULL; n2->next = NULL;
junk1 = n1; junk2 = n2;
free(junk1); free(junk2);
}
} /* end of else if */

if (curr == NULL || curr->next == NULL ||
    curr->next->next == NULL) {
    num_levels++;
    if (DEBUG) print_RElist(start, num_levels, symb_set);
    curr = start;
}
} /* end of while */
num_tasks = t_num;

/* While partitioning above tasks are assigned labels corresponding to
the level it appeared first in the rooted tree. Now, we adjust the
label according to the labeling scheme of Algorithm C */
for (i=0; i<num_tasks; i++)
    tlist[i].label = num_levels-tlist[i].label-1;

for (i=t_num-1; i>=0; i--) {
    strcpy(succ, tlist[i].succ);
    maxlabel = SENTINEL;
    for (j=0; succ[j] != '\0'; j++)
        if (tlist[succ[j]].label > maxlabel)
            maxlabel = tlist[succ[j]].label;
    if (i != t_num-1)
        tlist[i].label = maxlabel + 1;
} /* for i loop */

if (DEBUG) {
    printf("\nTask Graph produced by Partitioning Approach B\n");
    printf("^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");
    print_tlist();
}

} /* end of partn_C */

int is_present(ctask,num)
char ctask[5];
int num;
{
    char temp[5];
    int i,j;
    for (j=0; j<=2; j++)
        ctask[j] += 1;
    for (i=0; i<num; i++) {
        for (j=0; j<=2; j++)
            temp[j] = tlist[i].task[j]+1;
        temp[3] = '\0'; ctask[3] = '\0';
    }
}

```

```

        if (strcmp(temp,ctask) == 0)
            return(i);
    }
    return(SENTINEL);
}

/*****
Function Name: void print_tlist( )
Description:
    This routine prints all the information for all the tasks in the task
    graph in the form of a table. It uses the tlist global structure. The
    routine "print_atask" given below prints all the required information
    of a task.
*****/
void print_tlist()
{
    int i;
    void print_atask();

    printf("\n-----\n");
    printf("Task Number      Level      Predecessors      Successors      Processor\n");
    printf("-----\n");
    for (i=0;i<num_tasks;i++)
        print_atask(i);
    printf("-----\n\n");
} /* of print_tlist */

void print_atask(task)
int task;
{
    int    i,j, pred, succ;

    /* print Task number and its level */
    printf("    T%2d\t      %2d\t      ",task,tlist[task].label);
    /* print L and/or R predecessors, NONE if no predecessors */
    if (no_preds(task))
        printf("NONE ");
    else {
        if ((pred=tlist[task].task[0]) >= BEG_TASK)
            printf("T%d ",pred-BEG_TASK);
        if ((pred=tlist[task].task[2]) >= BEG_TASK)
            printf(" T%d",pred-BEG_TASK);
    }

    /* print all successors of task */
    printf("\t\t");
    for (j=0;(succ=tlist[task].succ[j]) != '\0'; j++)
        printf("T%d ",succ);
    if (j==0) /* no successors for task */
        printf("NONE");

    /* print task's processor */
    if (tlist[task].node != SENTINEL)
        printf("\t%d\n",tlist[task].node);
    else printf("\tNONE\n");
} /* print_atask */

```

SCHEDULE.C

```

#include "host.h"

/*****
This file (schedule.c) contains the following HOST routines:
    Get_schedule()           Gets the schedule
    init_labels()           gets initial label table
    Algorithm_A()           Applies Algorithm A
    algorithm_B()           Applies Algorithm C
*****/
Declaration of external variables */
extern int    num_levels;
extern Task_tree    tlist[MAX_TASKS];
extern Label_list    *l_list[MAX_LEVELS];
/*****
Prototype Definition : Get_Schedule(Label_list l_list[], char **G_chart);
Parameters: l_list - the structure for the label table of the tasks.
            G_chart - the structure (a 2-d array) to represent the schedule in the
            form of a Gantt Chart.

```

Description:

This routine schedules the task graph by either using Algorithm A or Algorithm B depending on the user's choice. It then obtains the schedule in Gantt chart form in the G_chart structure and performs the optimization as mentioned in the document.

```

*****/
void Get_schedule(G_chart,num_procs)
STTYPE      G_chart[MAX_PROCS][MAX_LEVELS];
int         num_procs;
{
    int      i,j,k,
            col,
            row,
            cur_task,      /* current task for execution */
            pred,         /* current task's predecessor */
            pred_node,   /* predecessor task's node */
            that_task,   /* task to swap for optimization */
            that_lpred,  /* swapping task's predecessors */
            that_rpred,
            op1,op2,     /* operands of current task */
            lpred,      /* current task's L and R predecessor */
            rpred;

    boolean swap=FALSE;
    Label_list *ptr;

    /* function prototypes */
    extern void print_schedule();
    int no_preds();

    /* Initially the Gantt chart to PHI tasks for all processors during
       all time intervals */
    for (i=0; i<num_procs; i++)
        for (j=0; j<num_levels; j++)
            G_chart[i][j] = PHI_TASK;

    /* Copying information from l_list (which is the final label table) in
       to the Gantt chart. That is, the tasks with label "i" are scheduled
       arbitrarily on the available processors during ith time interval */
    for (col=num_levels-1; col>=0; col--) {
        ptr = l_list[col]->next;
        for (row=0; row<num_procs; row++) {
            if (ptr != NULL)
                cur_task = ptr->value;
            else
                break;
            G_chart[row][num_levels-col-1] = cur_task;
            /* Update info in the task table as to which node the
               current task is assigned to */
            tlist[cur_task].node = row;
            ptr = ptr->next;
        }
    }
    for (row=0; row<num_procs; row++)
        G_chart[row][num_levels] = 0; /* null terminate each row string */
    if (DEBUG)
        print_schedule(G_chart,num_procs,num_levels);
    if (num_procs == 1) return;
        /* no optimization done if only one processor */

    /* Now, we do the optimization of scheduling the tasks based on their
       predecessors. This is done by checking for the predecessors node
       trying to move the current task to that node */
    for (col=1; col < num_levels; col++) {
        for (row=0; row<num_procs; row++) {
            cur_task = G_chart[row][col];
            if (cur_task == PHI_TASK) continue;
            if (no_preds(cur_task)) {
                tlist[cur_task].node = row;
                continue;
            }
            op1 = tlist[cur_task].task[0];
            if (op1>=BEG_TASK && op1<=END_TASK)
                lpred = op1 - BEG_TASK;
            else
                lpred = 0;
        }
    }
}

```

```

op2 = tlist[cur_task].task[2];
if (op2>=BEG_TASK && op2<=END_TASK)
    rpred = op2 - BEG_TASK;
else
    rpred = 0;

/* Check if one of its predecessor is assigned to the same node,
   then just update the task table and proceed */
if (lpred && tlist[lpred].node == row) {
    continue;
}
else if (rpred && tlist[rpred].node == row) {
    continue;
}
/* Else, check if "that_task" which is in the interval of the
   cur_task's predecessor, but assigned to the current node is
   an independent task or a PHI_TASK
   OR
   if that_task's predecessor is in the current interval,
   then just swap cur_task with that_task */

if (lpred) pred = lpred;
else if (rpred) pred = rpred;

swap = FALSE;
for ( ; ; ) {
    pred_node = tlist[pred].node;
    that_task = G_chart[pred_node][col];
    if (tlist[that_task].task[0]>=BEG_TASK)
        that_lpred = (tlist[that_task].task[0]-BEG_TASK);
    else
        that_lpred = 0;
    if (tlist[that_task].task[0]>=BEG_TASK)
        that_rpred = (tlist[that_task].task[0]-BEG_TASK);
    else
        that_rpred = 0;

    if (that_task == PHI_TASK || no_preds(that_task) ||
        (that_lpred && tlist[that_lpred].node==row) ||
        (that_rpred && tlist[that_rpred].node==row)) {
        swap = TRUE;
        G_chart[row][col] = that_task;
        G_chart[pred_node][col] = cur_task;
        tlist[that_task].node = row;
        tlist[cur_task].node = pred_node;
    }

    if (!swap && pred == lpred) {
        if (rpred) pred = rpred;
        else break;
    }
    else break;
} /* of infinite for loop */
} /* for col loop */
} /* for row loop */
} /* end of Get_schedule() */

/*****
Function Definition: int init_labels(int);
Description:
This function initially assigns labels to tasks following the
Labeling Scheme in Algorithm_A. Then the tasks with label "i" are
grouped into the set "Wi", that is, tasks with label "i" is stored
in the ith list of l_list structure.
*****/
int init_labels(num_tasks)
int num_tasks;
{
    int    level,          /* current level in the task tree */
          cur_task,      /* current task */
          wg;            /* width of the task graph */
    Label_list *ptr1;
    /* Initialize the l_list to have 0 in its first element
       in all the levels */

```



```

for (level=0;level<num_levels;level++) {
    l_list[level] = (Label_list *) malloc(sizeof(Label_list));
    l_list[level]->value = 0;
    l_list[level]->next = NULL;
}

/* Group tasks with label "1" into the set "w1" in the label list
   from the initial label table of tasks */
for (cur_task=0;cur_task<num_tasks;cur_task++) {
    /* Task with label "i", is stored in ith list of l_list */
    ptr1 = l_list[tlist[cur_task].label];
    ptr1->value++;
    while (ptr1->next != NULL)
        ptr1 = ptr1->next;

    ptr1->next = (Label_list *) malloc(sizeof(Label_list));
    ptr1->next->value = cur_task;
    ptr1->next->next = NULL;
} /* of for */

/* Also find out the width of the graph from this initial set
   representation in l_list structure */
wg = 0;
for (level=0;level<num_levels;level++) {
    if (l_list[level]->value > wg)
        wg = l_list[level]->value;
}
return(wg);
}

/*****
Function Definition: void algorithm_A( int )
Description:
This routine is the implementation for Algorithm A (Hu's algorithm) which
is outlined in Section 4.3.1 of the thesis document. The algorithm
schedules a task graph given in 'tlist' structure on an arbitrary number,
say p, of processors by adjusting the label table given in 'l_list'
structure.
*****/
void algorithm_A(num_procs)
int num_procs;
{
    int    i,
           level,           /* current set Wi of tasks */
           victim,         /* victim task to be moved to set Wl+1 */
           lpred, rpred;
    boolean found,         /* tells if a predecessor is found in Wi+1 */
           selected;      /* tells if a victim is found */
    Label_list *ptr1,*backptr, *ptr2, *junk1;

    level = num_levels-1; /* process from leaf tasks */
    while (level >= 0) {
        if (l_list[level]->value > num_procs) {
            if (level == num_levels-1) { /* if level==L */
                /* create a new set WL+1 and increment L */
                ptr1= (Label_list *) malloc(sizeof(Label_list));
                ptr1->value = 0;
                ptr1->next = NULL;
                l_list[num_levels++] = ptr1;
            }

            /* if level <> L */
            backptr = l_list[level];
            ptr1 = backptr->next;
            selected = FALSE;
            /* search until a victim is selected */
            while (!selected && ptr1 != NULL) {
                if (no_preds(ptr1->value)) {
                    selected = TRUE;
                }
                else { /* if the cur_task has predecessors, then */
                    if (tlist[ptr1->value].task[0]>=BEG_TASK)
                        lpred = tlist[ptr1->value].task[0]-BEG_TASK;
                }
            }
        }
    }
}

```

```

else  lpred = 0;
if (tlist[ptr1->value].task[2]>=BEG_TASK)
    rpred = tlist[ptr1->value].task[2]-BEG_TASK;
else  rpred = 0;
/* check if lpred or rpred is in Wn+1 set */
ptr2 = l_list[level+1]->next;
found = FALSE;
while (!found && ptr2 != NULL) {
    if ((lpred && lpred == ptr2->value) ||
        (rpred && rpred == ptr2->value))
        found = TRUE;
    else ptr2 = ptr2->next;
}

if (found) { /* found a predecessor in Wn+1 */
    backptr = ptr1;
    ptr1 = ptr1->next;
} /* if */
else /* no predecessors in Wn+1 */
    selected = TRUE;
}
} /* of inner while */
victim = ptr1->value; /*found a victim so store it */
/* free the victim node from the current row */
backptr->next = ptr1->next;
ptr1->next = NULL;
junk1 = ptr1;
free(junk1);

/* adjust the no. of tasks in the victim's row and its next row */
l_list[level]->value--;
l_list[level+1]->value++;

/* increment the label of the victim node,
   i.e. add it in Wn+1 set */
ptr1 = l_list[level+1];
while (ptr1->next != NULL)
    ptr1 = ptr1->next;
ptr1->next = (Label_list *) malloc(sizeof(Label_list));
ptr1->next->value = victim;
ptr1->next->next = NULL;
level = num_levels - 1; /* start from highest level */
} /* of initial if after while */
else level--;
} /* of while */
}

/*****
Function Definition: void algorithm_C (int )
Description:
This routine is the implementation for Algorithm A (Hu's algorithm) which
is outlined in Section 4.3.1 of the thesis document. The algorithm
schedules a task graph given in 'tlist' structure on an arbitrary number,
say p, of processors by adjusting the label table given in 'l_list'
structure.
*****/
void algorithm_C(num_procs)
int num_procs;
{
    int    i,
           level,          /* current set Wl of tasks */
           victim,        /* victim task to be moved to set Wi+1 */
           lpred, rpred;
    boolean found,        /* tells if a predecessor is found in Wi+1 */
           selected;     /* tells if a victim is found */
    Label_list *ptr1,*backptr, *ptr2, *junk1;

    level = num_levels-1;
    while (level >= 0) {
        /* if |Wl| <= p, then goto next set */
        if (l_list[level]->value <= num_procs) {
            level--;
            continue;
        }
    }
}

```

```

backptr = l_list[level];
ptr1 = backptr->next;
selected = FALSE;
while (!selected && ptr1 != NULL) {
    if (level == num_levels-1) {
        ptr2= (Label_list *) malloc(sizeof(Label_list));
        ptr2->value = 0;
        ptr2->next = NULL;
        l_list[num_levels++] = ptr2;
    }
    if (no_preds(ptr1->value)) {
        selected = TRUE;
    }
    else {
        /* if the cur_task has predecessors, then */
        if (tlist[ptr1->value].task[0]>=BEG_TASK)
            lpred = tlist[ptr1->value].task[0]-BEG_TASK;
        else lpred = 0;
        if (tlist[ptr1->value].task[2]>=BEG_TASK)
            rpred = tlist[ptr1->value].task[2]-BEG_TASK;
        else rpred = 0;
        /* check if lpred or rpred is in Wn+1 set */
        ptr2 = l_list[level+1]->next;
        found = FALSE;
        while (!found && ptr2 != NULL) {
            if ((lpred && lpred == ptr2->value) ||
                (rpred && rpred == ptr2->value))
                found = TRUE;
            else ptr2 = ptr2->next;
        }

        if (found) { /* found a predecessor in Wn+1 */
            backptr = ptr1;
            ptr1 = ptr1->next;
        } /* if */
        else /* no predecessors in Wn+1 */
            selected = TRUE;
    }
    if (ptr1==NULL) { /* could not find a victim Wn */
        backptr = l_list[++level]; /* try in Wn+1 */
        ptr1 = backptr->next;
    }
} /* of immediate while */

/* else found a victim in Wn, so store it */
victim = ptr1->value;
/* free the victim node from the current row */
backptr->next = ptr1->next;
ptr1->next = NULL;
junk1 = ptr1;
free(junk1);

/* adjust the no. of tasks in the victim's row and its next row */
l_list[level]->value--;
l_list[level+1]->value++;

/* add the victim node in the immediately next row */
ptr1 = l_list[level+1];
while (ptr1->next != NULL)
    ptr1 = ptr1->next;
ptr1->next = (Label_list *) malloc(sizeof(Label_list));
ptr1->next->value = victim;
ptr1->next->next = NULL;
level = num_levels - 1;
} /* of while */
}
/*****
Function Name: no_preds();
Function Prototype: no_preds( int);
Description:
    This function returns FALSE if a task has no predecessors, that is, it is
    an independent task. It returns TRUE, if task has atleast one predecessor,
    giving no indication whether it is the R or L one.
*****/
int no_preds(task)
int task;

```

```

{
    int op1,op2;
    op1 = tlist[task].task[0];
    op2 = tlist[task].task[2];
    if ((op1>=MIN_SYMB && op1<=MAX_SYMB) &&
        (op2>=MIN_SYMB && op2<=MAX_SYMB))
        return(TRUE);
    else
        return(FALSE);
}

```

```

void free_llist()
{
    int level;
    Label_list *back_ptr,*frnt_ptr;

    for (level=0;level<num_levels;level++) {
        back_ptr = l_list[level];
        frnt_ptr = back_ptr->next;
        while (frnt_ptr != NULL) {
            free(back_ptr);
            back_ptr = frnt_ptr;
            frnt_ptr = frnt_ptr->next;
        }
    } /* of for */
}

```

MISC.C

```

#include "host.h"
/*****
This file (misc.c) includes the following HOST routines:
    chk_re()           Checks the input RE
    in_to_post()      Converts RE to postfix form
    print_schedule()  Prints the schedule in Gantt chart form
    Miscellaneous routines
*****/

/*****
Function Name : int chk_re( void)
Description:
    Checks whether the given RE is in the right form or not, counting the
    number of right and left parantheses, checking for the right operators
    etc.
*****/
int chk_re(exprn,symb_set)
char exprn[MAX_RE];
char symb_set[MAX_RE];
{
    int    left=0,          /* number of left parantheses */
          right=0,         /* number of right parantheses */
          done = FALSE,
          max,             /* number of terms in the RE */
          i,j;            /* local index variables */

    char    c,ch1,ch2,
           expl[MAX_RE],
           new_exp[MAX_RE];

    for (i=0,j=0;(c=exprn[i]) != '\0';i++) {
        if (c=='(')    left++;
        if (c==')')    right++;

        if (c == '*') {
            expl[j++] = '/';
            /* b'cos - ASCII(/)>ASCII(.) > ASCII(+), which matches the
            precedence vals of *(/) > . > + */
            expl[j++] = '0';
            /* converting to infix notation with num(op)num format */
        }
        else expl[j++] = c;    } /* for */
    expl[j] = '\0';
    while (!done) {
        done = TRUE;

```

```

for (j=0,i=0;(c=exp1[i++]) != '\0'); {
    if (c!='/' && c!='+' && c != '(' && c != '.') {
        ch1 = c;
        ch2 = exp1[i++];
        if (isdigit(ch1) || ch1 == 'e')
            if (isdigit(ch2) || ch2 == '(') {
                new_exp[j++] = ch1;
                new_exp[j++] = '.';
                new_exp[j++] = ch2;
                done = FALSE;
            } /* of isdigit .. */
            else {
                new_exp[j++] = ch1;
                --i;
            }
        else if (ch1 == ')') {
            if(ch2 == '(' || isdigit(ch2)) {
                new_exp[j++] = ch1;
                new_exp[j++] = '.';
                new_exp[j++] = ch2;
                done = FALSE;
            }
            else {
                new_exp[j++] = ch1;
                --i;
            }
        } /* of if ch1 */
        else {
            new_exp[j++] = ch1;
            --i;
        }
    } /* of uppermost if */
    else new_exp[j++] = c;
} /* of for */
new_exp[j] = '\0';
strcpy(exp1,new_exp);
} /* of while */
strcpy(exprn,exp1);

/* estimate number of states in the NFA to be synthesized
from the RE */
max = 0;
for (i=0; (ch1=exprn[i]) != '\0'; i++) {
    if (strchr(symb_set,ch1)) /* if an alphabet */
        max +=2; /* every atomic RE (an alphabet) needs
two states to get its NFA */

    if (ch1 == '+')
        max += 2; /* UNION adds two new states to the NFA */
    if (ch1 == '/')
        max += 2; /* also does CLOSURE */
} /* for */
/* But, CONCAT does not add any new states */
max += 4; /* tolerance on the estimate */
printf("Estimate on states = %d \n",max);

if (left == right) return(max);
else return(0);
}

/*****
Function Name: void in_to_post( char *, char *)
Description:
This routine takes an RE in infix form and converts it into a postfix
form which is returned in exprn array.
*****/
void in_to_post(post,exprn)
char post[MAX_RE], /* input RE in infix form */
exprn[MAX_RE]; /* RE converted to postfix form */
{
    int quit, i,j=0,k=0;
    char c,
str[MAX_RE], /* temp string for the postfix exprn */
stack[MAX_RE]; /* stack for converting infix to postfix form */

```

```

/* Allocate memory to the local strings and initialize them */
strcpy(stack,"");
strcpy(str,"");
for (i=0; (c=exprn[i]) != '\0'; i++) {
    if (c>=48) {
        /* operand encountered */
        str[j++] = exprn[i];
    }
    else if (c == '(')
        stack[k++] = c;
    else if ((c<48) && (c>41))
    {
        /* an operator encountered, popo from stack and add to
        'post' each operator having >= precedence tha present
        operator */

        quit = 0;
        while (k>0 && !quit)
            if (stack[--k] < c) {
                quit = 1; ++k;
            } else str[j++] = stack[k];
        stack[k++] = c;
    }
    else if (c == ')') {
        /* right paran encountered */
        while (stack[--k] != '(')
            str[j++] = stack[k];
    }
} /* of for loop */
for (--k;k>=0;k--)
    str[j++] = stack[k];
str[j] = '\0';
strcpy(post,str);
} /* of in_to_post */

/*****
Description:
This routine converts the given integer 'i' to a string and
returns the string
*****/
char *itos(num)
int num;
{
    int i;
    char temp[5];

    temp[0] = num;
    temp[1] = '\0';
    return(temp);
}

/*****
Description:
This routine searches for a character 'c' in the string 's' and
returns the position where it found 'c'. If 'c' not in the
string 's' it returns a 0.
*****/
int strindex(s,c)
char *s;
int c;
{
    int n;
    for (n=0; ;n++) {
        if (s[n] == c)
            return(n);
        if (s[n] == '\0')
            return(SENTINEL);
    }
}
/*****
Function Definition: print_schedule(char **, int, int);
Description:
This routine prints the Gantt chart as a timing diagram with processor

```

```

axis and a time axis. The schedule is printed in intervals of 13 so as
to accomodate in a line. The routine "p_sch" prints all the requiried
data.
*****/
void print_schedule(G_chart,num_procs,num_levels)
STTYPE G_chart[][MAX_LEVELS];
int num_procs,num_levels;
{
    int nlevels,beg,last;
    void p_sch();
    nlevels = num_levels;
    beg = 0;
    if (nlevels > 13)
        last = 13;
    else last = nlevels;
    while (nlevels>0) {
        p_sch(beg,last,G_chart,num_procs,num_levels);
        nlevels -= 13;
        beg += 13;
        if (nlevels > 13)
            last += 13;
        else last = num_levels;
    }
}

void p_sch(beg,last,G_chart,num_procs,num_levels)
int beg,last;
STTYPE G_chart[][MAX_LEVELS];
int num_procs,num_levels;
{
    int i,j;
    /* printing dividing line of exact length between every
    processor's queue */
    printf("\n ");
    for (j=beg; j<last; j++)
        printf("-----");
    printf("-\n");
    for (i=0; i<num_procs; i++) {
        printf("P%d |",i);
        for (j=beg; j<last; j++) {
            if (G_chart[i][j] != PHI_TASK)
                printf("T%2d |",G_chart[i][j]);
            else
                printf("phi |");
        } /* for j loop */
        /* printing dividing line of exact length between every
        processor's queue */
        printf("\n ");
        for (j=beg; j<last; j++)
            printf("-----");
        printf("-\n");
    } /* for i loop */

    /* printing time ticks at the bottom of schedule */
    printf(" |");
    for (j=beg; j<last; j++)
        printf(" |");
    printf("\n");

    /* printing time intervals at the bottom of schedule */
    if (beg/10)
        printf(" %d",beg);
    else
        printf(" 0");
    for (j=beg; j<last; j++)
        if (j/9) printf(" %d",j+1);
        else printf(" %d",j+1);
    printf("\n");
}

```

NODE.H

```

#include <stdio.h>
#include <ctype.h>
#include <string.h>

```

```

/*****      General Header Constants      *****/
#define      ROOTNODE  0      /* node 0 is set as the controlling node */
#define      TRUE      1
#define      FALSE     0
#define      SENTINEL -1
#define      DEBUG     FALSE /* switch for debug information */

/*****      NFA and DFA header Constants      *****/
#define      MAX_NFA   250 /* Maximum number of states in an NFA */
#define      MAX_DFA   200 /* Maximum number of states in an NFA */
#define      MAX_COLS   3 /* Maximum size of the input alphabet */
#define      MAX_RE     250 /* Maximum size of the input RE */
#define      MAX_TASKS  100 /* Maximum number of tasks in the Task Graph */
#define      MAX_LEVELS 100 /* Maximum number of levels in the Task Graph */
#define      MAX_PROCS  16 /* Maximum number of levels in the Task Graph */
#define      MAX_SUCC   15 /* Maximum number of successors of a task */
#define      PHI      MAX_NFA /* Symbol for phi state in FA's transition table */
#define      PHI_TASK  ',' /* Symbol for phi task in the Gantt chart */

/*****      RE Header Constants      *****/
#define      MIN_SYMB  0 /* Range for number of input symbols */
#define      MAX_SYMB  90
#define      BEG_TASK  100 /* Range for number of tasks */
#define      END_TASK  1000
#define      CLOSURE   MAX_SYMB+1 /* symbol for Closure operation */
#define      CONCAT    MAX_SYMB+2 /* symbol for Concatenation operation */
#define      UNION     MAX_SYMB+3 /* symbol for Union operation */
#define      EPSILON   MAX_SYMB /* symbol for Epsilon */

#define      FREE      0 /* Symbol for a free buffer */
#define      IN_USE    1 /* Symbol for a used buffer */
#define      MARKED    1 /* Symbol for a marked pair in the marking table */
#define      UNMARKED  0 /* Symbol for an unmarked pair in the marking table */
*/

/****      Type Definitions      ****/
typedef unsigned short boolean;
typedef unsigned short SHORT;
typedef unsigned char STTYPE;

/* Type Definition for an NFA */
typedef struct {
    SHORT numst; /* number of states */
    SHORT start; /* start state */
    STTYPE final[20]; /* set of final states */
    STTYPE *table[MAX_NFA][MAX_COLS];
} NFA;

/* Type Definition for a DFA */
typedef struct {
    SHORT numst; /* number of states */
    SHORT start; /* start state */
    STTYPE final[20]; /* set of final states */
    STTYPE table[MAX_DFA][MAX_COLS];
} DFA;

/* Type Definition for a task in the Task Graph */
typedef struct {
    int label, /* task's label */
        node, /* task's processor */
        wt,
        status,
        done; /* if task has completed execution */
    STTYPE succ[MAX_SUCC], /* task's successors */
        task[3]; /* task's operation */
} Task_tree;

/* Type Definition of my node's task queue for execution
by Look Ahead approach */
typedef struct Q_struct {
    int task;
    struct Q_struct *next;
}

```



```

} My_Q;
/* Type Definition for temporary store of NFA while using
Look Ahead approach */
typedef struct BUFTYP {
    int access;
    int task;
    int status;
    NFA *bufnfa;
    struct BUFTYP *next;
} BUFFER;
typedef struct Label_struct {
    SHORT value;
    struct Label_struct * next;
} Label_list;

```

NODE.C

```

#include "node.h"
/*****
This file (node.c) contains the following NODE routines:
main()           The driver routine for the NODE MODULE
get_myQ()       Obtains task queue of my node from schedule
print_myQ()     Prints my node's task queue

The driver routine calls the following external routines:
exec_myQ()      in      makenfa.c file
rm_moves()     in      ecl.c file
nfa_to_dfa()   in      nfa_dfa.c file
min_dfa()      in      min_dfa.c file
dfa_re()       in      dfa_re.c file
solve_eqns()   in      eqns.c file
Miscellaneous routines in other.c file
*****/

/*****
Node Header Constants          *****/
#define HOSTPID      100      /* process id for host process */
#define NODEPID      0        /* process id for node process */
#define ALLNODES     -1      /* symbol for all nodes */
#define ALLPIDS      -1      /* symbol for all processes */
#define INIT_TYP     10      /* type of host to node message */
#define NFA_TYP      20      /* type of node to node NFA message */
#define DFA_TYP      30      /* type of node to node DFA message */
#define RE_TYP       100     /* type of node to host RE message */

/***** All variables global to this and other files are declared *****/
NFA *nfa = NULL; /* Global nfa Structure */
DFA *dfa = NULL; /* Global dfa Structure */

int maxterms, /* max # of states in NFA to be synthesized */
    nnodes, /* # of nodes in the allocated cube */
    work_nodes, /* # of working nodes as requested by the user */
    numtasks, /* # of tasks in the task tree */
    numlevels; /* # of levels in the tree */
char re[MAX_RE], /* Given RE */
    symbset[MAX_COLS];

boolean rootnode = FALSE; /* node which has the last task, i.e the NFA from
                           transformation T1 */

static long
    my_node, /* node id of my node */
    mpid, /* process id of process in my node */
    mhost; /* node id of my host node */
My_Q *HEAD;
Task_tree tlist[MAX_TASKS];
struct msg_typ {
    int work_nodes,
        numtasks,
        numlevels,
        maxterms,
        form_choice;
    char symbset[MAX_COLS];
    STTYPE G_chart[MAX_PROCS][MAX_LEVELS];
    Task_tree tlist[MAX_TASKS];
} init_msg; /* structure for sending initial message to nodes */
/*****
Function Name: int main( void );
Description: This is the main routine of the node program. It initially receives

```

the

information (task graph and the schedule in Gantt chart form obtained either scheduling Algorithm A or Algorithm B) from the host. Then:

1. All nodes participate in synthesizing an NFA following the method outlined in transformation T1 of Section 3.4.1. The NFA thus obtained is sent to all other nodes for the next step.
2. Next, all nodes participate in removing the e-moves from the NFA obtained in T1, following the transformation T2 of Section 3.4.2.
3. Then, only Node 0 participates in converting the NFA to a DFA by following the algorithm in Transformation T3 of Section 3.4.3. Since this algorithm cannot be implemented in parallel, only node 0 participates in this step. Other nodes idle.
4. Again only Node 0 participates in minimizing the DFA of T3 by following the method in Transformation T4 of Section 3.4.4. This procedure cannot be implemented in parallel. The DFA is sent to all other nodes for the next step.
5. If the user's choice corresponds to CONVERGENCE case, all nodes participate in obtaining the RE corresponding to the DFA of step 4, by following the algorithm outlined in Transformation T5 of Section 3.4.5. The RE is sent to the host as the final result from Node 0.
6. If the user's choice corresponds to DIVERGENCE case, then only node 0 obtains the set of RE equations corresponding to the DFA of step 4, by following the procedure outlined in Transformation T6 of Section 3.4.6
7. The set of RE equations of step 6 are solved by using Gaussian Elimination method of Transformation T7 of Section 3.4.7 by Node 0 only. This method cannot be implemented in parallel. The RE for the DFA is subsequently obtained and sent to the host from Node 0.

Then, all nodes wait for more information (another task graph and its corresponding schedule) from the node until a stop signal is encountered from the host.

```

*****/
main()
{
    /* 'main' of node */
    int      i,j,
            choice; /* choice of DIVERGENCE or CONVERGENCE */
    char     mylist[MAX_LEVELS];

    /* declarations of functions used in main */
    extern void  exec_myQ(), rm_emoves(), nfa_to_dfa(), min_dfa(),
                dfa_re(), solve_eqns();
    void       get_myQ(), free_myQ(), print_myQ(),
                print_tlist(), print_schedule(),
                print_nfa(), print_dfa();

    /* node bookkeeping done */
    my_node = mynode();
    mpid = mypid();
    mhost = myhost();

    while (TRUE) { /* node loops until stopped by host */
        /* receive initial message from host into init_msg struc */
        crecv(INIT_TYP, &init_msg, sizeof(struct msg_typ));

        /* unpack information from init_msg received from host */
        work_nodes = init_msg.work_nodes;
        numtasks   = init_msg.numtasks;
        numlevels  = init_msg.numlevels;
        maxterms   = init_msg.maxterms;
        choice     = init_msg.form_choice;
        strcpy(symbset, init_msg.symbset);
        for (i=0; i<numtasks; i++)
            tlist[i] = init_msg.tlist[i];

        /* if i am not a working node just busy loop */
        if (my_node >= work_nodes) continue;

        /* get my node's set of tasks from the Gantt chart into mylist */
        /* form linked list of tasks (task Q) from mylist and print it */
        get_myQ(init_msg.G_chart[my_node]);
        /* initialize nfa */
        if ((nfa=(NFA *) malloc(sizeof(NFA)))==NULL)
            exit(1);
        mem3D(nfa->table, maxterms, MAX_COLS, maxterms);
    }
}

```

```

/* All nodes together synthesize the NFA;
   but only rootnode has the resulting NFA */
exec_myQ();
free_myQ();
if (rootnode) { /* only my node contains the NFA */
    NFA_BUFTYP nfa_buf;
    long node;

    nfa_buf.numst = nfa->numst;
    nfa_buf.start = nfa->start;
    strcpy(nfa_buf.numst,nfa->final);
    for (i=1; i<=nfa->numst; i++)
        for (j=0; j<MAX_COLS; j++)
            strcpy(nfa_buf.table[i][j],nfa->table[i][j]);

    /* CSEND message of TYPE NFA_TYP from static buffer nfa
     * to all other work_nodes */
    for (node=0; node<work_nodes; node++)
        if (node != my_node)
            csend(NFA_TYP, &nfa_buf, sizeof(NFA_BUFTYP), node, NODEPID);
}
else { /* other nodes receive NFA from rootnode */
    NFA_BUFTYP nfa_buf;

    /* CRECV message of type NFA_TYP into static nfa buffer */
    crecv(NFA_TYP, &nfa_buf, sizeof(NFA_BUFTYP));
    nfa->numst = nfa_buf.numst;
    nfa->start = nfa_buf.start;
    strcpy(nfa->final,nfa_buf.final);
    for (i=1; i<=nfa->numst; i++)
        for (j=0; j<MAX_COLS; j++)
            strcpy(nfa->table[i][j],nfa_buf.table[i][j]);
}

if (DEBUG && (my_node == ROOTNODE)) {
    printf("\n\n***** FSA after STEP 1 (NFA) *****\n\n");
    print_nfa(nfa);
}

/* All nodes participate in removing the e-moves from the NFA */
rm_emoves();
if ((dfa=(NFA *) malloc(sizeof(NFA)))==NULL)
    exit(1);
/* only ROOTNODE does transformation T3 */
if (my_node == ROOTNODE) {
    if (DEBUG) {
        printf("\n\n***** FSA after STEP 2 (NFA) *****\n\n");
        print_nfa(nfa);
    }
    nfa_to_dfa(); /* ROOTNODE obtains the DFA from the NFA */
    if (DEBUG) {
        printf("\n\n***** FSA after STEP 3 (DFA) *****\n\n");
        printf("\n\t\t\t Transition Table (dfa) \n\n");
        print_dfa();
    }
}

/* All nodes free their NFA structure */
free3D(nfa->table,maxterms,MAX_COLS);
free((NFA *) nfa);
nfa = (NFA *) NULL;

/* ROOTNODE minimizes the DFA and prints it */
if (my_node == ROOTNODE) {
    min_dfa();
    if (DEBUG) {
        printf("\n\n***** FSA after STEP 4 (DFA) *****\n\n");
        printf("\n\t\t\t Transition Table (dfa) \n\n");
        print_dfa();
    }
}

if (choice == CONVERG) {
    if (my_node == ROOTNODE) {
        long node;
        /* ROOTNODE global sends the min DFA to all other work_nodes */
        for (node=0; node<work_nodes; node++)
            if (node != ROOTNODE)
                csend(DFA_TYP, dfa, sizeof(DFA), node, NODEPID);
    }
}

```

```

    }
    else /* all other nodes receive the new DFA */
        crecv(DFA_TYP, dfa, sizeof(DFA));
    dfa_re(); /* all nodes do transformation T5 */
} /* CONVERGENCE case ends */

/* If Divergence case chosen, only ROOTNODE finds the min DFA and
   also obtains the RE for this DFA */
if (choice == DIVERG) {
    if (my_node == ROOTNODE) {
        solve_eqns();
    } /* if my_node == ROOTNODE */
} /* ends DIVERGENCE case */

if (my_node == ROOTNODE) {
    csend(RE_TYP, re, MAX_RE, mhost, HOSTPID);
    /* send the final RE to the host */
}

/* free DFA space */
free((DFA *) dfa);
dfa = (DFA *) NULL;

/* now loop back to receive another task graph from the host
   or the stop signal */
} /* while */
} /* end of node main */

```

```

/*****
Function Name: get_myQ( char * )

```

Description:

This routine forms the task queue for mynode in the form of a linked list from the G-chart. PHI_TASKS in the schedule are ignored.

```

*****/

```

```

void get_myQ(mylist)
char mylist[MAX_LEVELS];
{
    int    col, task;
    My_Q   *cur;
    HEAD = NULL;
    for (col=0; col<numlevels; col++) {
        if ((task=mylist[col]) == PHI_TASK) continue;
        if (HEAD == NULL) {
            HEAD = (My_Q *) malloc(1*sizeof(My_Q));
            cur = HEAD;
            cur->task = task;
        } /* if */
        else {
            cur->next = (My_Q *) malloc(1*sizeof(My_Q));
            cur = cur->next;
            cur->task = task;
        } /* else */
    } /* for */
    cur->next = NULL;

    /* check if this task does not have any successors, i.e. it is
       last task. Then make this node as the "rootnode" */
    if (tlist[cur->task].succ[0] == '\0') {
        rootnode = TRUE;
    }
    else    rootnode = FALSE;
} /* of get_myQ */

```

```

/*****
Function Name: void free_myQ( void )

```

Description:

This routine releases the memory for the entire task Q of my node

```

*****/

```

```

void free_myQ()
{
    My_Q   *qptr, *freeptr;
    qptr = HEAD;
    while (qptr != NULL) {
        freeptr = qptr;
        qptr = qptr->next;
    }
}

```



```

/* print whether the current state is START, FINAL or START-FINAL
   state */
printf("\t");
if (i == dfa->start) {
    if (strchr(dfa->final,1))
        printf("S-F");
    else
        printf(" S ");
}
else if (strchr(dfa->final,1))
    printf(" F ");
else printf(" ");
printf(" | q%2d |",i);

/* now print the transition table entries */
for (j=0; j<MAX_COLS-1; j++)
    if ((st=dfa->table[i][j]) != PHI)
        printf("\tq%2d\t |",st);
    else printf("\tphi\t |");
printf("\n\t -----\n");
} /* 1st for */
printf("\n\n");
} /* of print_dfa */

no_preds(task)
int task;
{
    if ((tlist[task].task[0]>=MIN_SYMB &&
        tlist[task].task[0]<=MAX_SYMB)
        && (tlist[task].task[2]>=MIN_SYMB &&
            tlist[task].task[2]<=MAX_SYMB))
        return(TRUE);
    else return(FALSE);
}

```

MAKENFA.C

```

#include "node.h"

/*****
This file (makenfa.c) contains the following NODE routines:
    exec_myQ()      execute my Q using Look Ahead approach
    task_to_nfa()  synthesizes NFA for a task
    rules_nfa()    apply rules for making NFA of a task
    printing routines
*****/

/***/ Declaration of external functions and variables *****/
extern void mem3D(), free3D();
extern char *itos();

extern NFA *nfa; /* nfa structure */
extern int maxterms;
extern char symbset[MAX_COLS];
extern Task_tree tlist[MAX_TASKS];
extern My_Q *HEAD; /* Pointer to head of task queue */

/* Declaration of global variables for this file */
BUFFER *buf_hd, /* ptr to the head of the free buffer list */
*bufptr; /* a ptr to the free buffer list */
NFA *old1=NULL, *old2=NULL;

/*****
Function Definition: void exec_myQ(void);
Description :
    This routine synthesizes the NFA from the Schedule given in Gantt chart
    form by the host program. The method in Section 3.4.1 is followed
*****/
void exec_myQ()
{
    My_Q *cur, /* other pointers to task queue */
*back;
    int i,j,col,
node,
my_node,
task, /* current task */
pred,
lpred, /* L and R predecessors of a task */

```

```

    char      rpred;
    char      succ;
    boolean Go; /* flag for Look-Ahead technique */

/* function prototypes */
void      task_to_nfa(), store_in_buffer(),
          free_bufferlist(), rm_task();

/* allocate old1 and old2 structures and memory to its table */
old1 = (NFA *) malloc(1*sizeof(NFA));
mem3D(old1->table,maxterms,MAX_COLS,maxterms);
old2 = (NFA *) malloc(1*sizeof(NFA));
mem3D(old2->table,maxterms,MAX_COLS,maxterms);

cur = HEAD; /* HEAD is the beginning of the task list of mynode() */
while (cur != NULL) { /* till end of list */
    tlist[cur->task].done = FALSE; /* no task done yet */
    cur = cur->next;
} /* of while */

/*****
Look-ahead technique implemented by checking:
* Has the task no predecessor?
* Has the task a (L or R) predecessor?
** Is the predecessor on same node?
**** Has the predecessor(s) completed execution?
** Is the predecessor on a different node?
*** If so, has the message been received?
*****/
my_node = mynode();
cur = HEAD; back = cur;
while (HEAD != NULL) { /* not end of list */
    task = cur->task;
    if (tlist[task].task[0]>BEG_TASK)
        lpred = (tlist[task].task[0]-BEG_TASK);
    else lpred = 0;
    if (tlist[task].task[2]>BEG_TASK)
        rpred = (tlist[task].task[2]-BEG_TASK);
    else rpred = 0;

    Go = FALSE;
    if (!lpred && !rpred) /* no predecessors - so execute */
        Go = TRUE;
    else {
        if (lpred) pred = lpred; /* set pred to L or R pred */
        else if (rpred) pred = rpred;
        while (TRUE) {
            if (tlist[pred].node == my_node) { /* pred on same node */
                if (tlist[pred].done) /* pred completed execution */
                    Go = TRUE; /* so task can execute */
            }
            else if (iprobe(rpred)) Go = TRUE;
            /*
            * if message of TYPE 'rpred' waiting to be received
            * then task can execute
            */

            if (!Go) break;
            else if (!lpred || !rpred) break;
            else { /* if both preds exist, then check */
                Go = FALSE; /* for the other pred too */
                pred = rpred; lpred = 0;
            }
        } /* while */
    } /* else before while */

    if (Go) { /* task READY to be executed */
        tlist[task].done = TRUE;
        task_to_nfa(task);
        /* storing or sending result for successors of the task */
        for (i=0;(succ=tlist[cur->task].succ[i]) != '\0';i++) {
            if (tlist[succ].node != my_node) {
                /* if successor on another node send result to that node */
                NFA_BUFTYP nfa_buf;
                int i,j;
                nfa_buf.numst = nfa->numst;
                nfa_buf.start = nfa->start;
            }
        }
    }
}

```

```

strcpy(nfa_buf.final,nfa->final);
    for (i=1; i<=nfa->numst; i++)
        for (j=0; j<MAX_COLS; j++)
            strcpy(nfa_buf.table[i][j],nfa->table[i][j]);
    /* Synchronously send message of TYPE cur->task, from
    * static buffer nfa to 'succ' node */
csend(cur->task, &nfa_buf, sizeof(NFA_BUFTYP), tlist[succ].node, NODEPID);
    } /* if */
    else { /* successor on same node - so store result in buffer */
        store_in_buffer();
        bufptr->status = IN_USE;
        bufptr->access++;
        bufptr->task = cur->task;
    } /* else */
} /* for */

rm_task(cur,back); /* remove the task from the task Q */
cur = back = HEAD;
/* search for next READY task from head of the task-Q */
}
else { /* Look ahead for next READY task */
    if (cur != back) /* by moving cur and back pointers */
        back = cur; /* in the list */
    cur = cur->next;
}
} /* of upper while */

/* free old1 and old2 structures */
free3D(old1->table,maxterms,MAX_COLS);
free((NFA *) old1);
free3D(old2->table,maxterms,MAX_COLS);
free((NFA *) old2);
free_bufferlist(); /* free all the buffer space */
}

```

```

/*****
Prototype Definition: void task_to_nfa( int)

```

Description:

This routine gets the NFA corresponding to the current task's operation by using the routine rules_nfa(). It uses "old1" and "old2" as the NFA for the two operands of its operation. old1 and old2 could be NFA of atomic RES or NFA of previous result.

```

*****/

```

```

void task_to_nfa(task)
int task;
{
    int    pred,
          my_node;
    char   op1,op2, /* the two operands for the task */
          op;      /* infix operator for the task */

    /* function prototypes */
    void   rules_nfa(), assign(), get_from_buffer();

    my_node = mynode();
    op1 = tlist[task].task[0];
    op2 = tlist[task].task[2];
    op = tlist[task].task[1];

    if (op1>=MIN_SYMB && op1<=MAX_SYMB)
        assign(old1,op1);
    if (op1>=BEG_TASK && op1<=END_TASK) {
        pred = op1 - BEG_TASK;
        if (tlist[pred].node == my_node) /* pred on same node */
            get_from_buffer(pred,old1);
            /* get result from buffer into old1 */
        else if (tlist[pred].node != my_node) { /* pred on different node */
            NFA_BUFTYP nfa_buf;
            /* Synchronous receive message of TYPE pred, into buffer */
            crecv(pred, &nfa_buf, sizeof(NFA_BUFTYP));
            old1->numst = nfa_buf.numst;
            old1->start = nfa_buf.start;
            strcpy(old1->final,nfa_buf.final);
            for (i=1; i<=old1->numst; i++)
                for (j=0; j<MAX_COLS; j++)
                    strcpy(old1->table[i][j],nfa_buf.table[i][j]);
        }
    }
}

```



```

if (op2>=MIN_SYMB && op2<=MAX_SYMB)
    assign(old2,op2);
if (op2>=BEG_TASK && op2<=END_TASK) {
    pred = op2 - BEG_TASK;
    if (tlist[pred].node == my_node) /* pred on same node */
        get_from_buffer(pred,old2);
        /* get result from corresponding buffer into old2 */
    else if (tlist[pred].node != my_node) { /* pred on different node */
        NFA_BUFTYP nfa_buf;

        /* Synchronous receive message of TYPE pred, into buffer */
        crecv(pred, &nfa_buf, sizeof(NFA_BUFTYP));
        old2->numst = nfa_buf.numst;
        old2->start = nfa_buf.start;
        strcpy(old2->final,nfa_buf.final);
        for (i=1; i<=old2->numst; i++)
            for (j=0; j<MAX_COLS; j++)
                strcpy(old2->table[i][j],nfa_buf.table[i][j]);
    }
}
rules_nfa(op);
/* make the nfa from old1 and old2 depending on 'op' */
}

/*****
Function Name: rm_task.
Prototype Definition: rm_task(My_Q *, My_Q *, My_Q *);
Description:
    Removes the currently executed task (pointed to by curr) from mynode's
    task queue, and adjusts the curr and back pointers appropriately.
*****/
void rm_task(curr,back)
My_Q *curr, /* pointers to mynode's task queue */
    *back;
{
    My_Q *old=NULL;
    old = curr; /* temp pointer to the node to be deleted */
    if (curr == back) {
        /* task to be deleted is the first node in the task list */
        HEAD = HEAD->next;
    }
    else { /* task to be deleted in the middle of the list */
        back -> next = curr;
    }

    old->next = NULL;
    free(old);
}

/*****
Function Name: get_from_buffer()
Function Prototype: void gt_from_buffer(BUFFER *, int, NFA *, int)
Description:
    Get the corresponding buffer from the buffer list which corresponds
    to the 'pred' task. Then copy all the information from this buffer's nfa
    to the 'old' nfa passed as parameter
*****/
void get_from_buffer(pred,old)
int pred;
NFA *old;
{
    int i,j;

    bufptr = buf_hd;
    /* Get the right buffer - which has the result of pred */
    while (bufptr->task != pred)
        bufptr = bufptr->next;
    /* Copy nfa from that buffer to old */
    /* copying the state information of the nfa */
    old->numst = bufptr->bufnfa->numst;
    old->start = bufptr->bufnfa->start;
}

```

```

strcpy(old->final, bufptr->bufnfa->final);
/* copying the transition table */
for (i=1; i<=bufptr->bufnfa->numst; i++)
    for (j=0; j<MAX_COLS; j++) {
        strcpy(old->table[i][j],bufptr->bufnfa->table[i][j]);
    } /* for j */

/* Since one task has retrieved info from this buffer, decrement
its # of accesses */
bufptr->access--;
if (bufptr->access == 0) { /* if # access becomes 0 */
    bufptr->status = FREE; /* mark the buffer FREE and also
                           free its nfa */
    free3D(bufptr->bufnfa->table,maxterms,MAX_COLS);
    free((NFA *) bufptr->bufnfa);
}
}

/*****
Function Name: get_free_buffer()
Prototype Definition: BUFFER *get_free_buffer(BUFFER *, int)
Description:
This function searches the buffer list for a free buffer. If it finds
an existing buffer which is FREE, it initializes this buffer's nfa.
If no FREE buffer available in the list, then it creates one at the
beginning or at the end of the list and initializes it appropriately.
The free buffer is then returned.
*****/
void get_free_buffer()
{
    /* Function prototypes */
    BUFFER *create_buffer();

    bufptr = buf_hd; /* bufptr points to the head of the buffer list */
    if (buf_hd == NULL) { /* buffer list is empty */
        buf_hd=create_buffer(); /* Create buffer at beginning */
        bufptr = buf_hd;
    }
    else { /* Buffer list is not empty now */
        while (bufptr != NULL) { /* find a FREE buffer */
            if (bufptr->status == IN_USE)
                bufptr = bufptr->next;
            else
                break;
        }

        if (bufptr == NULL) { /* No FREE buffer available in list */
            bufptr = buf_hd;
            while (bufptr->next != NULL)
                bufptr = bufptr->next;
            bufptr->next=create_buffer(); /* Create buffer at the end*/
            bufptr = bufptr->next;
        }
        else { /* a FREE buffer available, so allocate memory
                and initialize its nfa */
            bufptr->bufnfa = (NFA *) malloc(sizeof(NFA));
            mem3D(bufptr->bufnfa->table,maxterms,MAX_COLS,maxterms);
        }
    } /* else */
}

/*****
Function Name: create_buffer()
Prototype Definition: BUFFER *create_buffer( int )
Description:
This function creates a structure of type BUFFER and initializes the
structure members, including allocating and initializing the buffer's
nfa. The created buffer is returned.
*****/
BUFFER *create_buffer()
{
    BUFFER *new_buffer;
    new_buffer = (BUFFER *) malloc(1*sizeof(BUFFER));
    new_buffer->next = (BUFFER *) NULL;          new_buffer->status = FREE; /*

```

```

set status to FREE buffer */
    new_buffer->access = 0;          /* set # of accesses to 0 */
    new_buffer->bufnfa = (NFA *) malloc(sizeof(NFA));
    /* Allocate small amount of memory and initialize the
       transition table of the buffer nfa */
    mem3D(new_buffer->bufnfa->table,maxterms,MAX_COLS,maxterms);
    return(new_buffer);
}

/*****
Function Name : free_bufferlist()
Function Prototype : void free_bufferlist(BUFFER *)
Description:
    Removes the complete buffer list occupied by all the buffers used
    in this file. Note that the 'bufnfa' members of each of the buffers
    have been assumed to be freed as and when a buffer becomes FREE during
    execution of the program. This routine only frees all the buffers and
    removes the links between them.
*****/
void free_bufferlist()
{
    BUFFER *bufptr = buf_hd, *freeptr;
    while (bufptr != NULL) {
        freeptr = bufptr;
        bufptr = bufptr->next;
        freeptr->next = NULL;
        free((BUFFER *) freeptr);
    } /*while */
    buf_hd = NULL;
}

/*****
Function Name: store_in_buffer()
Prototype Definition: void store_in_buffer(NFA *, NFA *)
Description:
    This routine copies the nfa structure from the 'from_nfa' structure to
    the 'to_nfa' structure, by copying all the members of the structure
    including the transition table. It is assumed that the 'to_nfa' member
    has been allocated only a small amount of memory when passed as a
    parameter. So this routine checks if additionally memory is required and
    allocates the sufficient memory whenever needed.
*****/
void store_in_buffer()
{
    int i,j,sz1;

    get_free_buffer();
    /* copying the state information of the nfa */
    bufptr->bufnfa->numst = nfa->numst;
    bufptr->bufnfa->start = nfa->start;
    strcpy(bufptr->bufnfa->final, nfa->final);

    /* copying the transition table */
    for (i=1; i<=nfa->numst; i++)
        for (j=0; j<MAX_COLS; j++) {
            strcpy(bufptr->bufnfa->table[i][j],nfa->table[i][j]);
        } /* for j */
}

/*****
Function Definition: void assign(NFA *, STTYPE)
Description:
    Gets the NFA for an atomic RE, i.e. either a "e" or "phi" or "a".
    Note, the states in the NFA start from q1 instead of q0.
*****/
void assign(fa,op1)
NFA *fa;
STTYPE op1;
{
    STTYPE i,j;
    if (op1 == EPSILON) { /* NFA for EPSILON */
        fa->numst = 1;

```

```

fa->start = 1; /* states always start from q1 */
strcpy(fa->final,itos(1));
/* start and only state in the NFA moves to the PHI state on
any input symbol */
for (j=0; j<MAX_COLS-1; j++)
    strcpy(fa->table[1][j],itos(PHI));
/* Except on input "e", when it moves to the start state itself */
strcpy(fa->table[1][MAX_COLS-1],itos(1));
}

else { /* NFA for any symbol other than EPSILON */
fa->numst = 2;
fa->start = 1;
strcpy(fa->final,itos(2));
for (j=0; j<MAX_COLS; j++) {
    if (j==op1)
        strcpy(fa->table[1][j],itos(2));
    else
        strcpy(fa->table[1][j],itos(PHI));
        strcpy(fa->table[2][j],itos(PHI));
    } /* for */
} /* of else */
} /* end of assign */

/*****
Function Definition: void rules_nfa(char)
Description:
    Applies the rules of "concatenation", "union", or "closure" on the
    two automata "old1" and "old2" to get the new nfa. The method is
    illustrated in section 3.4.1 of the thesis document.
*****/
void rules_nfa(op)
STTYPE op; /* operation between machines M1 and m2 */
{
    int i=0,j=0,k=0,p=0,t; /* local index variables */
    int shift; /* shift position of states */
    STTYPE *offset();

    if (op == CONCAT) { /* CONCAT of M1 and M2 */
        nfa->start = old1->start; /* start state of M' is that of M1 */
        nfa->numst = old1->numst + old2->numst;
            /* number of states in M' = M1 + M2 states */

        /* M1's transition table is added to M' without any change;
        e-moves from the final state of M1 is added later */
        for (i=1; (i<=old1->numst); i++)
            for (j=0;j<=MAX_COLS-1;j++) {
                strcpy(nfa->table[i][j],old1->table[i][j]);
            }
        p = i;
        shift = old1->numst;
        /* M2's transition table is added to M' without any change;
        e-moves from the final state of M1 is added later */
        for (i=1; (i <= old2->numst); i++,p++)
            for (j=0;j<=MAX_COLS-1;j++) {
                strcpy(nfa->table[p][j],offset(old2->table[i][j],shift));
            }

        /* set the final in the new nfa */
        strcpy(nfa->final,itos(--p));

        /* Adding e-moves from the final states of M1 to the start
        state of M2 in the new machine M' */
        for (i=0; (t=old1->final[i]) != '\0'; i++)
            strcat(nfa->table[t][MAX_COLS-1],itos(old2->start+shift));
    } /* end of CONCAT operation */

    else if (op == UNION) { /* begin of UNION operation */
        /* set the number of states in M' */
        nfa->numst = old1->numst + old2->numst + 2;
        nfa->start = 1; /* set the start state in M' */
        for (j=0;j<MAX_COLS; j++) /* and its transitions to PHI */
            strcpy(nfa->table[1][j],itos(PHI));
        p =2; /* e-move from q1 of M' to start state of M1 */
    }
}

```

```

strcpy(nfa->table[1][MAX_COLS-1],itos(p));
/* Add the transition table of M1 to M' without any change;
   e-moves from the final state of M1 is added later */
for (i=1; (i<=old1->numst); i++,p++)
  for (j=0;j<=MAX_COLS-1;j++) {
    strcpy(nfa->table[p][j],offset(old1->table[i][j],1));
  }
shift = p-1;
/* Adding the e-move from q1 of M' to start state of M2 */
strcat(nfa->table[1][MAX_COLS-1],itos(p));
/* Add the transition table of M1 to M' without any change;
   e-moves from the final state of M1 is added later */
for (i=1; (i<=old2->numst); i++,p++)
  for (j=0;j<=MAX_COLS-1;j++) {
    strcpy(nfa->table[p][j],offset(old2->table[i][j],shift));
  }
/* Now 'p' points to the last state of M', which we
   set as the final state of M' */
strcpy(nfa->final,itos(p));
for (j=0;j<MAX_COLS; j++) /* set its transitions to PHI */
  strcpy(nfa->table[p][j],itos(PHI));
/* Now we add all the e-moves from all final states of
   M1 to the above final state of M' */
for (i=0; (t=old1->final[i]) != '\0'; i++)
  strcat(nfa->table[t+1][MAX_COLS-1],itos(p));
/* Now we add all the e-moves from all final states of
   M2 to the above final state of M' */
for (i=0; (t=old2->final[i]) != '\0'; i++)
  strcat(nfa->table[t+shift][MAX_COLS-1],itos(p));
} /* end of UNION operation */
else if (op == CLOSURE) { /* begin of CLOSURE operation */
  /* Set the number of states in M' */
  nfa->numst = old1->numst + 2;
  /* Set the start state in M' */
  nfa->start = 1;
  for (j=0;j<MAX_COLS; j++) /* set its transitions to PHI */
    strcpy(nfa->table[1][j],itos(PHI));
  /* e-move from q0 of M' to start state of M1 */
  strcpy(nfa->table[1][MAX_COLS-1],itos(old1->start+1));
  /* Adding transition table of M1 to M' without any change;
     e-moves from the final state of M1 is added later */
  p =2;
  for (i=1; (i<=old1->numst); i++,p++)
    for (j=0;j<=MAX_COLS-1;j++) {
      strcpy(nfa->table[p][j],offset(old1->table[i][j],1));
    }
  /* Now index 'p' points to last state of M', which we
     mark as a final state */
  strcpy(nfa->final,itos(p));
  for (j=0;j<MAX_COLS; j++) /* set its transitions to PHI */
    strcpy(nfa->table[p][j],itos(PHI));
  /* adding e-move from start to final state of M' */
  strcat(nfa->table[1][MAX_COLS-1],itos(p));
  /* adding all the e-moves from all final states of
     M1 to the final state of M' indicated by index 'p' */
  for (i=0; (t=old1->final[i]) != '\0'; i++)
    strcat(nfa->table[t+1][MAX_COLS-1],itos(p));
  /* adding all the e-moves from all final states of
     M1 to the start state of M1 */
  for (i=0; (t=old1->final[i]) != '\0'; i++)
    strcat(nfa->table[t+1][MAX_COLS-1],itos(2));
} /* end of CLOSURE operation */
} /* end of rules_nfa */

```

Function Name : offset()

Prototype Definition: char *offset(char *, int)

Description:

This routine takes a string, adds an offset value to each character

```

of the string, and returns the new string.
*****/
STTYPE *offset(oldstr,val)
STTYPE *oldstr;
char val;
{
    int i;
    STTYPE newstr[MAX_RE];
    strcpy(newstr,oldstr);
    for (i=0; oldstr[i]!=0; i++)
        if (oldstr[i] != PHI)
            newstr[i] += val;
    return(newstr);
}

```

ECL.C

```

#include "node.h"
/*****
This file (ecl_p.c) contains the following NODE routines:
    rm_moves      Removes the e-moves from the NFA
    get_ecl       Obtains the e-closure for every NFA state
    print_ecl     Prints the e-closure of every NFA state
*****/

/* All external declarations of functions and variables */
extern char *itos();
extern void rm_repeat();

extern NFA *nfa;
extern int work_nodes,maxterms;

/* Declaration of local variables to the file */
static long
    my_node,
    basic_states, /* basic number of states for each node */
    extra_states, /* additional number of states */
    my_states, /* total number of states for my node */
    my_beg,my_end, /* range of states for my node */
    xlens[MAX_PROCS];

/*****
Function Definition: rm_moves( void )
Description:
    This routine removes the e-moves in the NFA by following the method
    outlined in transformation T2 of Section 3.4.2
*****/
void rm_moves()
{
    boolean both = FALSE; /* flag for to tell if q0 belongs to F */
    int pos,
        i,j,k,p,t1; /* temporary index vars */
    STTYPE *tot_ecl[MAX_NFA], /* array for e_closure of all states */
        *tot_fn[MAX_NFA][MAX_COLS],
        *my_fn[MAX_NFA][MAX_COLS], /* total transition table of the new NFA */
        temp[MAX_NFA]; /* transition table for my states only */
    extern void mem3D(), free3D();
    void get_ecl(), print_ecl();

    /* calculating parameters for my node operation */
    my_node = mynode();
    basic_states = nfa->numst/work_nodes;
    extra_states = nfa->numst*work_nodes;
    my_states = basic_states+extra_states;
    if (my_node < extra_states) {
        my_states = basic_states + 1;
        my_beg = my_node*my_states+1;
    }
    else {
        my_states = basic_states;
        my_beg = (my_node*my_states) + extra_states+1;
    }
    my_end = my_beg + my_states-1;
}

```

```

rm_repeat(nfa->final);
/* allocate memory only to the required number of row pointers
   in tot_ecl and assign to null */
for (i=0; i<=maxterms; i++) {
    if ((tot_ecl[i] = (STTYPE *)
        malloc(maxterms*sizeof(STTYPE))) == NULL)
        exit(1);
    strcpy(tot_ecl[i], "");
}
/* Obtain the e_closure of all states and print it */
get_ecl(tot_ecl); /* all nodes execute this */
if (DEBUG && mynode() == ROOTNODE)
    print_ecl(tot_ecl, nfa->numst); /* only one node prints */
/* if e_closure(q1) contains a state of F, i.e. the set of final
   states, then F' = F U {q0} */
for (i=0, both=FALSE; !both && ((t1=nfa->final[i])!=0); i++)
    if (strchr(tot_ecl[1], t1))
        both = TRUE;

/* Allocate memory dynamically to the my_fn and tot_fn arrays
   which act as the temporary NFA after removing e-moves */
mem3D(my_fn, maxterms, MAX_COLS, maxterms);
mem3D(tot_fn, maxterms, MAX_COLS, maxterms);

/* This part forms the transfer function of the new NFA.
   Algorithm in Transformation T2 of Section 3.4.2 of the
   thesis report followed. But parallel implementation done */
for (i=my_beg; i<=my_end; i++) {
    for (j=0; j<MAX_COLS-1; j++) {
        strcpy(temp, "");
        for (p=0; (pos=tot_ecl[i][p]) != 0; p++)
            if (pos != PHI) {
                strcat(temp, nfa->table[pos][j]);
                rm_repeat(temp);
            }
        strcpy(my_fn[i][j], "");
        for (p=0; (pos=temp[p]) != 0; p++)
            if (pos != PHI) {
                strcat(my_fn[i][j], tot_ecl[pos]);
                rm_repeat(my_fn[i][j]);
            }
    } /* for j loop */
} /* for i loop */

if (my_node != 0) {
    for (i=my_beg; i<=my_end; i++)
        for (j=0; j<MAX_COLS-1; j++)
            strcpy(my_fn[i][j], my_fn[i+my_beg][j]);
}

xlens[0] = (my_states+1)*MAX_COLS*maxterms*sizeof(STTYPE);
for (i=1; i<work_nodes; i++)
    xlens[i] = my_states*MAX_COLS*maxterms*sizeof(STTYPE);
/* get length of contribution of each node into xlens */
gcolx(my_fn, xlens, tot_fn);
/* collect tot_fn using "gcolx" */

/* Assign the new transition matrix to the structure 'nfa'
   Note to blank the e-move column in the newly formed 'nfa' */
for (i=1; i<=nfa->numst; i++) {
    for (k=0; k<MAX_COLS-1; k++)
        strcpy(nfa->table[i][k], tot_fn[i][k]);
    strcpy(nfa->table[i][MAX_COLS-1], "");
}

/* Check if F' = F U {q0} or not */
if (both)
    strcat(nfa->final, itos(nfa->start));
/* free memory for all local dynamic structures */
free3D(my_fn, maxterms, MAX_COLS);
free3D(tot_fn, maxterms, MAX_COLS);
for (i=0; i<=maxterms; i++)
    free(tot_ecl[i]);
}
/*****
Function Definition: void get_ecl(STTYPE ** )

```

Description:

This part forms the my_ecl table, which consists of e_closure for my states. Then, my_ecl from all nodes is collected to give the e_closure of all states in tot_ecl, by using the "gcolx" global operation. E_closure of a state q0 denoted by e_closure(q0) is defined as the set of all the states which can be reached from state q0 with one or more arcs labeled with "e".

```

*****
void get_ecl(tot_ecl)
STTYPE *tot_ecl[MAX_NFA];
{
    boolean done;
    STTYPE *my_ecl[MAX_NFA], /* e_closure of my states only */
           temp[MAX_NFA],
           i,k,st,pos,t1;

    /* allocating memory to my_ecl */
    for (i=0; i<=maxterms; i++) {
        if ((my_ecl[i] = (STTYPE *)
            malloc(maxterms*sizeof(STTYPE))) == NULL)
            strcpy(my_ecl[i],"");
    }

    for (st=my_beg; st<=my_end; st++) {
        k=0;
        strcpy(temp,"");
        done = FALSE;
        pos = st;
        strcpy(my_ecl[st],itos(st));
        while (!done) {
            for (i=0; (t1=nfa->table[pos][MAX_COLS-1][i]) != 0; i++) {
                if (t1 != pos && t1 != PHI && !strchr(my_ecl[st],t1)) {
                    strcat(my_ecl[st],itos(t1));
                    strcat(temp,itos(t1));
                    /* storing future states in a temp array */
                }
            }
            /* for */
            if ((pos=temp[k++]) == 0)
                done = TRUE;
        } /* while */
    } /* outer for */

    if (my_node != 0) {
        for (i=0; i<my_states; i++)
            strcpy(my_ecl[i],my_ecl[i+my_beg]);
    }

    xlens[0] = (my_states+1)*maxterms*sizeof(STTYPE);
    for (i=1; i<work_nodes; i++)
        xlens[i] = my_states*maxterms*sizeof(STTYPE);
    /* get length of contribution of each node into xlens */
    gcolx(my_ecl,xlens,tot_ecl);
    /* collect vector using "gcolx" */

    /* deallocate memory for my_ecl */
    for (i=0; i<=maxterms; i++) {
        free(my_ecl[i]);
    }
}

```

```

/*****
Function Definition: void print_ecl( char **, int);

```

Description:

This function prints the E-cl table which contains the e-closure of each state of the nfa with e-moves

```

*****
void print_ecl(tot_ecl,num)
STTYPE *tot_ecl[MAX_NFA];
int num;
{
    int i,j,t;
    for (i=1; i<=num; i++) {
        printf("\nE_CLOSURE OF q%d : {" ,i);
        for (j=0; (t=tot_ecl[i][j]) != 0; j++)

```



```

        printf("q%d ",t);          printf(" )\n");
    } /* for i */
}
NFA_DFA.C
#include "node.h"
/*****
This file (nfa_dfa.c) contains the following NODE routines:
    nfa_to_dfa()          Converts an NFA to a DFA
    set_dfa_final()      Sets the final states in the DFA
    exists()             Checks if a new DFA state is encountered
Some printing routines
*****/

/* Declaration of all external functions and variables */
extern char    *itos();
extern void    rm_reeat();
extern NFA     *nfa;
extern DFA     *dfa;
extern int     maxterms;

/* Declaration of All global variables */
unsigned char  *nfa_st[MAX_NFA];

/*****
Function Definition: void nfa_to_dfa( void )
Description:
This routine converts the given NFA to a DFA by following the
transformation T3 in Section 3.4.3 of the thesis document. The
structure "pairs" containing the new DFA state and the corresponding
set of NFA states is used.
*****/
void nfa_to_dfa()
{
    STTYPE    str1[MAX_NFA], str2[MAX_NFA];
    int        index1=0, index2=0, /* indexes into pairs */
              index=0,
              i,j,k,st,
              inp; /* input symbol */

    /* function prototypes */
    void      set_dfa_final(), print_pairs();
    int       exists();

    mem2D(nfa_st,MAX_NFA,maxterms);
    strcpy(nfa_st[index1++],itos(nfa->start));
    rm_repeat(nfa->final);
    while (index1 != index2) {
        for (inp=0; inp<=MAX_COLS-2; inp++) {
            strcpy(str2,"");
            for (i=0; (st=nfa_st[index2][i]) != 0; i++) {
                strcpy(str1,nfa->table[st][inp]);
                if (i==0) {
                    strcpy(str2,str1);
                    continue;
                }
                for (j=0; str1[j]!=0; j++) {
                    if (str1[j] == PHI) continue;
                    if (!strchr(str2,str1[j]))
                        strcat(str2,itos(str1[j]));
                } /* for "j" loop */
            } /* for "i" loop */

            if (strlen(str2)==0) {
                dfa->table[index2+1][inp] = PHI;
                continue;
            }
            if (index=exists(str2,index1))
                /* existing set of NFA states */
                dfa->table[index2+1][inp] = index+1;
            else { /* a new set of NFA states => new DFA state */
                dfa->table[index2+1][inp] = index1+1;
                strcpy(nfa_st[index1++],str2); } /* else */
        }
    }
}

```

```

        } /* for "inp" loop */
        index2++;
    } /* while */
    dfa->start = 1;
    dfa->numst = index2;
    strcpy(dfa->final,"");

    if (DEBUG && mynode() == ROOTNODE)
        print_pairs(index2);

    /* find the final states of DFA from the 'nfa_st' structure */
    set_dfa_final(index2);
} /* end nfa_to_dfa */

/*****
Function Definition: int exists( char *, int )
Description:
    This routine checks for every set of NFA states, if this set corresponds
    to a new DFA state and returns the new DFA state.
*****/
int exists(set,last)
unsigned char set[];
int last;
{
    int i,j;
    for (i=0; i<=last; i++) {
        if (strcmp(nfa_st[i],set) == 0)
            return(i);
    }
    return(0);
}

/*****
Function Definition: void set_dfa_final( char *)
Description:
    Given a set of NFA states which corresponds to a DFA state, this routine
    determines if the DFA state is a final state or not, provided one of the
    NFA states in the set is a final state.
*****/
void set_dfa_final(last)
int last;
{
    int i,j,st;
    for (i=0; (st=nfa->final[i]) != 0; i++)
        for (j=0; j<last; j++) {
            if (strchr(nfa_st[j],st))
                strcat(dfa->final,itos(j+1));
        }
} /* end set_dfa_final */

/*****
Function Definition: void print_pairs( int )
Description:
    This routine prints the set of NFA states and the corresponding DFA
    state obtained in the new DFA.
*****/
void print_pairs(last)
int last;
{
    int i,j,st;
    printf("\n\n\n\t*** STEP 3 - Correspondence between DFA and NFA");
    printf(" States ***\n\n");
    printf(" DFA State \tNFA State set \n");
    printf("-----\n");
    for (i=0; i<last; i++) {
        printf("\tQ%2d <==> \t{",i+1);
        for (j=0; (st=nfa_st[i][j]) != 0; j++) {
            printf(" q%2d ",st);
            if ((j%20)==0) printf("\n");
        }
    }
}

```

```

        printf("\n");
    }
    printf("-----\n");
}

```

MIN_DFA.C

```

#include "node.h"
/*****
This file (min_dfa.c) contains the following NODE routines:
    get_newst()    To get the new DFA state
    mark_others() To mark pairs of states in the pending list
    put_pend()     To put in the pending list of the current pair
    append()       To put in a queue for recursively marking a
                  pending list
Some printing routines
*****/

typedef      struct rq {      /* type for recursive pairs to be marked */
            SHORT      p,q;
            struct rq  *next;
            } r_Q;

/* Declartion of all external functions and variables */
extern void  rm_repeat();
extern char  *itos();
extern DFA   *dfa;

/* Declaration of all variables global to this file */
STTYPE pend[100][70],
        newfinal[MAX_DFA];      /* final states of the new DFA */
STTYPE  *marks[MAX_DFA],        /* marking table */
        *new_fn[MAX_DFA],      /* tr table for the minimized DFA */
        *new_st[MAX_DFA];      /* states in the minimized DFA */
r_Q      *recursive_Q;          /* head of recursive list of pairs */

/*****
                                MININIZATION ALGORITHM
Function Definition: void min_dfa( void )
Description:
    This function takes a dfa and removes the inaccessible and irredundant
    states by following the minimization algorithm of transformation T4
    outlined in Section 3.4.4. The minimized dfa is returned back in the
    same 'dfa' structure.
*****/
void min_dfa()
{
    boolean marked = FALSE;
    SHORT   i,j,m,n,        /* local index variables */
            p,q,           /* states p and q in the algorithm */
            r,s,           /* r = delta(p,input) and s = delta(q,input) */
            a,             /* inut symbol */
            num_eq,        /* # of equivalent states */
            num_st=0,      /* # of states in DFA */
            st,
            newstart;      /* start of the new DFA */

    /* function prototypes */
    void put_pend(), mark_others(), print_marks(), print_pend();
    STTYPE get_newst();

    num_st = ++dfa->numst;
    /* allocate only required memory dynamically to all arrays */
    for (i=0; i<= num_st+5; i++) {
        if ((new_st[i]=(STTYPE *)
            malloc((num_st+5)*sizeof(STTYPE)))==NULL)
            exit(1);
        strcpy(new_st[i],"");
        if ((marks[i]=(STTYPE *)
            malloc((num_st+5)*sizeof(STTYPE)))==NULL)
            exit(1);
        strcpy(marks[i],"");
    }
}

```

```

if ((new_fn[i]=(STTYPE *)
    malloc((num_st+5)*sizeof(STTYPE)))==NULL)
    exit(1);
strcpy(new_fn[i],"");
} /* for i */
/* Make the last state as PHI-STATE and adjust the transitions
accordingly */
for (j=0; j<=MAX_COLS-2; j++)
    dfa->table[num_st][j] = num_st;
for (i=1; i<num_st; i++)
    for (j=0; j<=MAX_COLS-2; j++)
        if (dfa->table[i][j] == PHI)
            dfa->table[i][j] = num_st;
for (i=0; i<100; i++)
    strcpy(pend[i],"");
for (i=0; i<=num_st; i++)
    new_st[i][0] = 0;
for (i=0; i<=num_st; i++)
    for (j=0; j<=num_st; j++) /* initialize marks to UNMARKED */
        marks[i][j] = UNMARKED;

/* STEP 1: Mark (p,q) for all p in F and q in (Q-F) */
for (q=1; q<=num_st; q++) {
    if (!strchr(dfa->final,q)) { /* q in (Q-F) */
        for (i=0; (p=dfa->final[i]) != 0; i++) {
            /* p in (F) */
            if (p<q) marks[q][p] = MARKED;
            else marks[p][q] = MARKED;
        } /* for i loop */
    } /* if */
} /* for q */

for (p=2; p<=num_st; p++) {
    for (q=1; q<p; q++) {
        if (marks[p][q] == MARKED) continue;
        marked = FALSE;
        for (a=0; a<=MAX_COLS-2; a++) { /* for all input symbols */
            r = dfa->table[p][a]; /* r = d(p,a) */
            s = dfa->table[q][a]; /* s = d(q,a) */

            if (r==s) continue;
            if (r<s) {
                SHORT temp = r;
                r = s; /* swap r and s */
                s = temp;
            }
            if (marks[r][s]) { /* if (r,s) entry marked */
                marks[p][q] = MARKED; /* mark (p,q) entry also */
                marked = TRUE;
                mark_others(p,q); /* mark all unmarked pairs on the
list for (p,q) */
                break; /* go for next (p,q) pair */
            }
        } /* for a loop */
        if (!marked) { /* no pair (d(p,a),d(q,a)) is marked */
            for (a=0; a<=MAX_COLS-2; a++) { /* for all input symbols */
                r = dfa->table[p][a]; /* r = d(p,a) */
                s = dfa->table[q][a]; /* s = d(q,a) */

                if (r==s) continue;
                if (r<s) {
                    SHORT temp = r;
                    r = s; /* swap r and s */
                    s = temp;
                }
                put_pend(p,q,r,s); /* put (p,q) pair on the list for (r,s) */
            } /* for a loop */
        }
    }
}

```

```

    } /* of if */
  } /* for q loop */
} /* for p loop */
if (DEBUG && my_node == ROOTNODE)
    print_marks(num_st);

num_eq = 0;
for (i=0,p=2; p<=num_st; p++)
    for (q=1; q<p; q++)
        if (marks[p][q] != MARKED)
            num_eq++;
if (num_eq == 0) return;
for (p=2; p<=num_st; p++) {
    for (q=1; q<p; q++) {
        if (marks[p][q] != MARKED) {
            boolean found = FALSE;
            for (i=1; new_st[i][0] != 0; i++) {
                if (strchr(new_st[i],p) || strchr(new_st[i],q)) {
                    strcat(new_st[i],itos(p));
                    strcat(new_st[i],itos(q));
                    rm_repeat(new_st[i]);
                    found = TRUE;
                    break;
                } /* if */
            } /* for i loop */
            if (!found) {
                strcat(new_st[i],itos(p));
                strcat(new_st[i],itos(q));
            }
        } /* if !MARKED */
    } /* for q loop */
} /* for p loop */

for (p=0;p<=num_st;p++) {
    boolean found = FALSE;
    for (i=1; new_st[i][0] != 0; i++) {
        if (strchr(new_st[i],p)) {
            found = TRUE;
            break;
        }
    } /* for i loop */
    if (!found)
        strcat(new_st[i],itos(p));
} /* for p loop */

dfa->numst = --i;

for(i=1;new_st[i][0]!=0;i++)
    for(a=0; a<=MAX_COLS-2; a++) {
        st = dfa->table[new_st[i][0]][a];
        new_fn[i][a] = get_newst(st);
    }

strcpy(newfinal,"");
rm_repeat(dfa->final);
for(i=1; new_st[i][0] != 0; i++) {
    for(j=0;(st=new_st[i][j]) !=0;j++) {
        if (st == dfa->start)
            newstart = i;
        if (strchr(dfa->final,st))
            strcat(newfinal,itos(i));
    }
}

for (i=1;new_st[i][0] !=0;i++)
    for(j=0;j<=MAX_COLS-2;j++)
        dfa->table[i][j] = new_fn[i][j];
dfa->numst = --i;
strcpy(dfa->final,newfinal);
dfa->start = newstart;

/* free memory for all dynamic arrays */
for (i=0; i<=num_st+5; i++) {
    free(marks[i]);
    free(new_st[i]);
    free(new_fn[i]);
}

```

```

} /* of min_dfa */

/*****
Function Name: get_newst()
Prototype Definition: STTYPE get_newst( SHORT );
Description:
    If the given state oldst is in the set of states (new_st) of the new DFA,
    this function returns the corresponding new state.
*****/
STTYPE get_newst(oldst)
SHORT oldst;
{
    SHORT p,q;

    for(p=1;new_st[p][0] != 0;p++)
        if (strchr(new_st[p],oldst))
            return((STTYPE) p);
}

void put_pend(p,q,r,s)
SHORT p,q;
{
    STTYPE put_str[3],
           srch_str[3],
           str1[MAX_DFA];
    SHORT i,j;

    put_str[0] = p; put_str[1] = q; put_str[2] = 0;
    srch_str[0] = r; srch_str[1] = s; srch_str[2] = 0;
    for (i=0; pend[i][0] != 0; i++) {
        strcpy(str1,pend[i]);
        str1[2] = 0;
        if (strcmp(srch_str,str1) == 0) {
            strcat(pend[i],put_str);
            return;
        }
    } /* for i loop */

    strcat(pend[i],srch_str);
    strcat(pend[i],put_str);
}

/*****
Function Name: void mark_others(SHORT, SHORT)
Description:
    When the pair (p,q) gets marked, then recursively mark all pairs on the
    list for (p,q) and also on the lists of other pairs that gets marked in
    this list.
*****/
void mark_others(p,q)
SHORT p,q;
{
    SHORT i,j,
           mark_p,mark_q;
    STTYPE srch_str[3],
           str1[100];
    r_Q *freeptr;

    recursive_Q = NULL;
    append(p,q);
    while (recursive_Q != NULL) {
        p = recursive_Q->p;
        q = recursive_Q->q;
        srch_str[0] = p; srch_str[1] = q; srch_str[2] = 0;
        for (i=0; pend[i][0] != 0; i++) {
            strcpy(str1,pend[i]);
            str1[2] = 0;
            if (strcmp(srch_str,str1) == 0) {
                for (j=2; pend[i][j] != 0; j += 2) {
                    mark_p = pend[i][j];
                    mark_q = pend[i][j+1];
                    marks[mark_p][mark_q] = MARKED;
                    append(mark_p,mark_q);
                } /* for j loop */
            }
        }
    }
}

```

```

        pend[i][2] = 0;
    } /* if */
} /* for i loop */
freeptr = recursive_Q;
recursive_Q = recursive_Q->next;
freeptr->next = NULL;
free(freeptr);
} /* while */
} /* end of mark_others */

append(p,q)
SHORT p,q;
{
    r_Q    *ptr;
    if (recursive_Q == NULL) {
        recursive_Q = (r_Q *) malloc(1*sizeof(r_Q));
        ptr = recursive_Q;
    }
    else {
        /* if (p,q) already exists in the queue then return */
        ptr = recursive_Q;
        while (ptr != NULL) {
            if (ptr->p == p && ptr->q == q)
                return;
            ptr = ptr->next;
        }
        /* now (p,q) does not exist - so append it at the end */
        ptr = recursive_Q;
        while (ptr->next != NULL)
            ptr = ptr->next;
        ptr->next = (r_Q *) malloc(1*sizeof(r_Q));
        ptr = ptr->next;
    }
    ptr->p = p;
    ptr->q = q;
    ptr->next = NULL;
}

/*****
Function Definition: void print_marks( int )
Description:
This routine prints the Marking Table of transformation T5. Each
marked pair is represented by an "X" and an unmarked pair by an "U".
*****/
void print_marks(num)
SHORT num;
{
    SHORT    i,j;
    printf("\n\t Snap shot of Marking table \n\n");
    for (i=2; i<=num; i++) {
        for (j=1; j<i; j++) {
            printf(" (%d,%d)-",i,j);
            if (marks[i][j])
                printf("X");
            else printf("U");
        }
        printf("\n");
    }
} /* end of print_marks */

DFA_RE.C

#include "node.h"

/*****
This file (dfa_re.c) contains the following NODE routines:
    dfa_re()      Driver routine to obtain the RE for the DFA
    rm_redun()   Removing redundancies in the RE
    Miscellaneous linked list routines
*****/
extern void    rm_redun(), rm_repeat();
extern char    *itos();
extern DFA    *dfa;

```

```

extern int      work_nodes;
extern char     re[MAX_RE],
               symbset [MAX_COLS];

typedef struct g1 {
    SHORT      row,col;
    char       re[MAX_RE];
    struct g1 *next;
} Gtype;

Gtype *G1=NULL;          /* G_1 array for sub-graph G1 */

/*****
Function Definition: void dfa_re( void )
Description:
This function takes a dfa and constructs the regular expression in 're'
corresponding to the given dfa, by following the algorithm outlined in
transformation T5 of Section 3.4.5. It constructs several sub-graphs from
the transition graph of the given DFA, determines the RE for each
sub-graph, and finally obtains the RE for the DFA by the union of the
RES of all the sub-graphs.
*****/
void dfa_re()
{
    SHORT      i,j,k,p,x,y,      /* temporary variables */
              accept,          /* only final state of each subgraph */
              start,          /* start state of DFA */
              node_i,          /* node to be deleted in reducing subgraph */
              num_st,         /* # of dfa states */
              inp;            /* input symbol */

    long       my_node;

    Gtype      *w1,*w2,*w3,*w4, /* represent arcs in reduced subgraph */
              *ptr1, *ji_ptr, *ik_ptr, *ii_ptr, *jk_ptr;

    STTYPE     temp_re[MAX_RE], /* temporary store for RE */
              str1[MAX_RE],    /* temporary string variables */
              Wjk[MAX_RE],
              delnodes[MAX_DFA];

    /* function prototypes */
    void print_Gt(), free_Gt();
    void listcat(), listcpy(), delete();
    Gtype *inlist(), *create();

    strcpy(re,"");
    num_st = dfa->numst;
    rm_repeat(dfa->final);

    /* if only one state in DFA, find its RE and quit */
    if (num_st == 1) {
        strcpy(re,"");
        for (j=0;j<MAX_COLS-1; j++)
            if (dfa->table[1][j] != PHI) {
                strcat(re,itos(symbset[j]));
                strcat(re,"+");
            } /* if */
        re[strlen(re)-1] = '\0';          /* removing last '+' in re */
        strcat(re,"**");
        return;
    }

    my_node = mynode();
    num_final = strlen(dfa->final);
    /* Start processing each subgraph Gt by selecting */
    /* only one final state of DFA for each subgraph */
    for (p=my_node; p<num_final; p += work_nodes) {
        accept=dfa->final[p];
        /* Reinitialize G1 for every subgraph by constructing a node-node
           transition graph from the dfa transition table, with entries
           being RES instead of states */
        free_Gt();
        for (i=1; (i<=num_st); i++)          for (inp=0; inp < MAX_COLS-1; inp++) {
            x=dfa->table[i][inp];
            strcpy(str1,itos(symbset[inp]));

```



```

        strcat(str1,"+");
        listcat(i,x,str1);
    } /* for inp loop */
for (i=1; i<=num_st; i++)
    for (j=1; j<=num_st; j++)
        if ((ptr1=inlist(i,j)) != NULL) {
            strcpy(str1,"(");
            k = strlen(ptr1->re) - 1;
            ptr1->re[k] = 0; /* removing the '+' at end */
            if (k>1) {
                strcat(str1,ptr1->re);
                strcat(str1,")");
                strcpy(ptr1->re,str1);
            } /* of if k */
        } /* if ptr1 */
strcpy(temp_re,"");
strcpy(delnodes,"");
/* Choosing a node "i" in G1, that is neither the start
   nor the accepting node */
for (node_i=1; node_i<=num_st; node_i++) {
    if (node_i == dfa->start || node_i == accept) continue;
    /* Delete node i as follows */
    /* Initially select every pair [j,k], neither equal to
       i, including j=k */
    strcat(delnodes,itos(node_i));
    for (j=1;j<=num_st;j++) {
        if (j == node_i || strchr(delnodes,j)) continue;
        for (k=1;k<=num_st;k++) {
            if (k == node_i || strchr(delnodes,k)) continue;
            /* Now test for the TWO cases in the algorithm for
               deleting a node */
            strcpy(Wjk,"");
            /* Test of CASE 1 */
            if (((ji_ptr=inlist(j,node_i)) != NULL) &&
                ((ik_ptr=inlist(node_i,k)) != NULL) &&
                ((ii_ptr=inlist(node_i,node_i)) == NULL)) {
                /* find the new arc Wjk */
                strcpy(Wjk,ji_ptr->re);
                strcat(Wjk,ik_ptr->re);
            } /* end of CASE 1 */
            /* Test of CASE 2 */
            if (((ji_ptr=inlist(j,node_i)) != NULL) &&
                ((ik_ptr=inlist(node_i,k)) != NULL) &&
                ((ii_ptr=inlist(node_i,node_i)) != NULL)) {
                /* find the new arc Wjk = Wji(Wii)*Wik */
                strcpy(Wjk,ji_ptr->re);
                strcat(Wjk,ii_ptr->re);
                strcat(Wjk,"");
                strcat(Wjk,ik_ptr->re);
            } /* end of CASE 2 */
            /* Last step of replacing all arcs between node j and node k
               with a single arc which is the union of all the arcs
               including the new Wjk arc */
            if (Wjk[0] == 0) continue;
            if ((jk_ptr=inlist(j,k)) == NULL)
                /* assigning only the new arc */
                listcpy(j,k,Wjk);
            else if ((jk_ptr=inlist(j,k)) != NULL) {
                strcpy(str1,"(");
                strcat(str1,jk_ptr->re);
                strcat(str1,"+");
                strcat(str1,Wjk);
                strcat(str1,")");
                strcpy(jk_ptr->re,str1);
            } /* if */
        } /* of for k loop */
    } /* of for j loop */
} /* Removing all arcs incident onto
node_i in Gt */
for (i=1;i<=num_st;i++)
    delete(i,node_i);
for (j=1;j<=num_st;j++)
    delete(node_i,j); } /* for node_i loop */
/* At this point only start and accept nodes are

```

```

left in G1. Use them to find the RE for the
duced graph of G_t */
strcpy(temp_re, "");
start = dfa->start;
w1=w2=w3=w4=(Gtype *) NULL;
w1=inlist(start, start);
w2=inlist(start, accept);
w3=inlist(accept, accept);
w4=inlist(accept, start);
/* Copy w1* to temp_re */
if (w1 != NULL) {
    if (strlen(w1) > 1) {
        strcat(temp_re, "(");
        strcat(temp_re, w1->re);
        strcat(temp_re, ")");
    }
    else strcat(temp_re, w1->re);
    strcat(temp_re, "*");
}

/* if only one state in Gt then temp_re has the RE for Gt */
if (start == accept) {
    if (w1 == NULL)
        strcat(re, "e + ");
    else {
        strcat(re, temp_re);
        strcat(re, " + ");
    }
    continue;
}
/* Then Concatenate w2 to temp_re */
if (strchr(w2->re, '+') != NULL) {
    strcat(temp_re, "(");
    strcat(temp_re, w2->re);
    strcat(temp_re, ")");
}
else strcat(temp_re, w2->re);
strcat(temp_re, "(");
/* Then Concatenate w3 to temp_re */
if (w3 != NULL) {
    strcat(temp_re, w3->re);
}
if (w3 != NULL && w4 != NULL && w2 != NULL)
    strcat(temp_re, "+");
/* Then add w4 to temp_re, if it is not NULL.
if w4 is NULL, temp_re is ready */
if (w4 == NULL) {
    strcat(re, temp_re);
    strcat(re, ") * + ");
    continue;
}
else if (w4 != NULL)
    if (strchr(w4->re, '+') != NULL) {
        strcat(temp_re, "(");
        strcat(temp_re, w4->re);
        strcat(temp_re, ")");
    }
    else strcat(temp_re, w4->re);
/* Then Concatenate w1* to temp_re */
if (w1 != NULL)
    strcat(temp_re, w1->re);
/* Then Concatenate w2 to temp_re */
if (strchr(w2->re, '+') != NULL) {
    strcat(temp_re, "(");
    strcat(temp_re, w2->re);
    strcat(temp_re, ")");
}
else strcat(temp_re, w2->re);
/* finally append temp_re to re with a "+" */
strcat(temp_re, ") * + ");
strcat(re, temp_re);
} /* for p=0 ... */

```

```

    free_Gt();
    rm_redun(re);
    /* Each node has a part of the final RE. All these parts are
       collected by using global concatenation system call gcolx() */
    for (i=0; i<work_nodes; i++)
        relens[i] = sizeof(char)*strlen(re);
        /* getting lengths of all node's RE contributions */
    gcolx(re,relens,temp_re);
        /* Global collect RE vector */
    strcpy(re,temp_re);
    re[strlen(re)-2] = '\0'; /* remove extra '+' at end of re */
    rm_redun(re);
} /* of dfa_re */

/*****
Function Definition: Gtype *inlist( SHORT, SHORT)
Description:
    This routine checks if the i,j entry is in the list. If so, it returns
    a pointer to that entry in the list, else a null pointer.
*****/
Gtype *inlist(i,j)
SHORT i,j;
{
    Gtype *ptr1;
    ptr1 = G1;
    while (ptr1 != NULL) {
        if ((ptr1->row == i) && (ptr1->col == j))
            /* check for i,j entry */
            break;
        else ptr1 = ptr1->next;
    }
    return(ptr1);
}

/*****
Function Definition: void listcat(SHORT, SHORT, char *)
Description:
    This routine checks if the i,j entry exists in the list. If so, it
    concatenates 'str' to the re of the i,j entry. If not, it creates
    a new i,j entry at the end of the list, and concatenates 'str' to the
    re of this new entry.
*****/
void listcat(i,j,str)
SHORT i,j;
char str[];
{
    Gtype *ptr1;
    if ((ptr1=inlist(i,j)) != NULL) /* i,j entry exists in list */
        strcat(ptr1->re,str);
    else { /* i,j entry does not exist in list */
        ptr1 = create(i,j);
        strcat(ptr1->re,str);
    }
}

/*****
Function Definition: void listcpy(SHORT, SHORT, char *)
Description:
    This routine checks if the i,j entry exists in the list. If so, it
    copies 'str' to the re of the i,j entry. If not, it creates
    a new i,j entry at the end of the list, and copies 'str' to the
    re of this new entry.
*****/
void listcpy(i,j,str)
SHORT i,j;
char str[];
{
    Gtype *ptr1;
    if ((ptr1=inlist(i,j)) != NULL) /* i,j entry exists in list */
        strcpy(ptr1->re,str);
    else { /* i,j entry does not exist in list */
        ptr1 = create(i,j);
        strcpy(ptr1->re,str);
    }
}

```

```

    }
}
/*****
Function Definition: Gtype *create(SHORT, SHORT)
Description:
    This routine creates the i,j entry at the end of the G1 list, and returns
    a pointer to this newly created entry.
*****/
Gtype *create(i,j)
SHORT i,j;
{
    Gtype *ptr1;
    if (G1 == NULL) {          /* if list empty, create new */
        G1 = (Gtype *) malloc(sizeof(Gtype));
        ptr1 = G1;
    }
    else {                    /* if list not empty, create at end */
        ptr1 = G1;
        while (ptr1->next != NULL)
            ptr1 = ptr1->next;
        ptr1->next = (Gtype *) malloc(sizeof(Gtype));
        ptr1 = ptr1->next;
    } /* else */

    ptr1->row = i;
    ptr1->col = j;
    ptr1->next = NULL;
    strcpy(ptr1->re, "");
    return(ptr1);
}

/*****
Function Definition: void delete(SHORT, SHORT);
Description:
    This routine deletes the i,j entry if it exists from the G1 list.
*****/
void delete(i,j)
SHORT i,j;
{
    Gtype *front,*back;
    front = G1;
    while (front != NULL) {
        if ((front->row == i) && (front->col == j)) {
            if (front == G1)
                G1 = G1->next;
            else
                back->next = front->next;
            front->next = NULL;
            free(front);
            front = NULL;
        } /* if */
        else {
            back = front;
            front = front->next;
        }
    } /* while */
}

void free_Gt()
{
    Gtype *ptr1, *freeptr;
    if (G1 != NULL) {
        ptr1 = G1;
        while (ptr1 != NULL) {
            freeptr = ptr1;
            ptr1 = ptr1->next;
            freeptr->next = NULL;
            free(freeptr);
        } /* while */
    }
}

```

```

        G1 = NULL;
    } /* if */
}

EQNS.C
#include "node.h"
/*****
This file (eqns.c) contains the following NODE routines:
    solve_eqns()    driver routine for solving RE equations
    get_eqns()     obtains the RE equations for the DFA
    sub_eqns()     forward substitution of equations
*****/

/* Declaration of external functions and variables */
extern void    rm_redun();
extern DFA    *dfa;
extern char    re[MAX_RE], symbset[MAX_COLS];

typedef struct eq {
    SHORT row,col;
    char eqn[MAX_RE];
    struct eq *next;
} Eqtype;
Eqtype *eqnmat = NULL;          /* list to store RE equations */
/*****
Function Name: solve_eqns()
Prototype Definition: void solve_eqns(DFA *, char *, char *)
Description:
    This routine initially forms the set of RE equations in the 'eqnmat'
    array for the given DFA by the method in Section 3.4.6. Then, the equations
    in 'eqnmat' are solved by the method in Section 3.4.7 to obtain the solution
    for the state variables of each of the final states of the DFA. Finally,
    the RE is obtained by the union of the REs for all the final states.
*****/
void solve_eqns()
{
    SHORT    i,j,k,
            numvar,          /* # of state variables */
            num_st;         /* # of DFA states */

    boolean flg = FALSE;
    char    str1[MAX_RE];
    Eqtype *ptr1, *ptr2;

    /* function prototypes */
    void    init_eqns(), get_eqns(),
            sub_eqns(), free_eqnmat(),
            eqlistcat(), eqlistcpy(), eqdelete();
    Eqtype *eqcreate(), *ineqlist();

    num_st = dfa->numst-1;
    numvar = dfa->numst-1; /* # of state variables = # of states */

    for (i=1; i<=num_st; i++)
        for (j=0; j<=MAX_COLS-2; j++)
            if (dfa->table[i][j] == (num_st+1))
                dfa->table[i][j] = PHI;

    get_eqns(numvar);

    /* Forward iteration of solving equations using Gaussian
       Elimination method as outlined in transformation T7 of
       section 3.4.7 */
    for (i=1; i<=numvar; i++) {
        if ((ptr1=ineqlist(i,i)) != NULL) {
            strcpy(str1, "(");
            strcat(str1, ptr1->eqn);
            strcat(str1, ")*");
            strcpy(ptr1->eqn, "");
            eqdelete(i,i);
            flg = TRUE;
            for (j=1; j<=numvar+1; j++)
                if ((ptr1=ineqlist(i,j)) != NULL) {
                    if (strcmp(ptr1->eqn, "e") != 0)
                        strcat(ptr1->eqn, str1);
                }
        }
    }
}

```

```

        else      strcpy(ptr1->eqn, str1);          flg = FALSE;
    }
    else if (flg) flg = TRUE;
    if (flg)
        eqlistcpy(i, numvar+1, str1);
} /* of upper if */
sub_eqns(i, numvar); /* forward substitution of the
                    equatin just solved */
} /* of uppermost for i loop */

/* Back substitution process - This part of the code substitutes the
solutions for all variables starting from the last variable (which
is already solved) into the rest of the equations */
for (i=numvar-1; i>=1; i--)
    for (j=1; j<=numvar; j++) {
        strcpy(str1, "");
        if ((ptr1=ineqlist(i, j)) != NULL) {
            if ((ptr2=ineqlist(j, numvar+1)) != NULL) {
                if (strchr(ptr2->eqn, '+') != NULL) {
                    strcat(str1, "(");
                    strcat(str1, ptr2->eqn);
                    strcat(str1, ")");
                }
                else
                    strcat(str1, ptr2->eqn);
            }
            else
                eqlistcpy(j, numvar+1, str1);
            if (strchr(ptr1->eqn, '+') != NULL) {
                strcat(str1, "(");
                strcat(str1, ptr1->eqn);
                strcat(str1, ")");
            }
            else
                strcat(str1, ptr1->eqn);
            eqdelete(i, j);
        }
        else continue;
        if ((ptr1=ineqlist(i, numvar+1)) != NULL) {
            strcat(ptr1->eqn, "+");
            strcat(ptr1->eqn, str1);
        }
        else eqlistcat(i, numvar+1, str1);
    } /* of inner for */

strcpy(re, "");
for (i=1; i<=numvar; i++)
    if (strchr(dfa->final, i))
        if ((ptr1=ineqlist(i, numvar+1)) != NULL) {
            strcat(re, ptr1->eqn);
            strcat(re, " + ");
        }

re[strlen(re)-2] = 0; /* removing the '+' at the end */
rm_redun(re); /* removing the redundancies in the RE */

free_eqnmat();
} /* of solve_eqns */

```

```

/*****
Function Name: void sub_eqns()

```

Description:

This routine performs the forward substitution of the equation which has been currently solved (indicated by the 'ind' parameter) into the rest of the bottom equations until the last equation. The procedure followed for this forward substitution process is that used in Gaussian Elimination method of transformation T7 of Section 3.4.7

```

*****/

```

```

void sub_eqns(ind, numvar)
SHORT ind, numvar;
{
    SHORT i, j;
    char str1[MAX_RE], str2[MAX_RE];
    Eqtype *ptr1, *ptr2;

```

```

if (ind != numvar)
  for (i=ind+1;i<=numvar;i++) {
    strcpy(str1,"");
    if ((ptr1=ineqlist(i,ind)) != NULL) {
      if (strcmp(ptr1->eqn,"e") != 0)
        if (strchr(ptr1->eqn,'+') != NULL) {
          strcpy(str1,"(");
          strcat(str1,ptr1->eqn);
          strcat(str1,")");
        }
        else
          strcpy(str1,ptr1->eqn);
      strcpy(ptr1->eqn,"");
      eqdelete(i,ind);
      for (j=1;j<=numvar+1;j++) {
        strcpy(str2,"");
        if ((ptr1=ineqlist(ind,j)) != NULL) {
          if (strcmp(ptr1->eqn,"e") != 0)
            if (strchr(ptr1->eqn,'+') != NULL) {
              strcpy(str2,"(");
              strcat(str2,ptr1->eqn);
              strcat(str2,")");
            }
            else
              strcpy(str2,ptr1->eqn);
          strcat(str2,str1);
          if ((ptr1=ineqlist(i,j)) != NULL) {
            strcat(ptr1->eqn,"+");
            strcat(ptr1->eqn,str2);
          }
          else
            eqlistcat(i,j,str2);
        }
      } /* of for j loop */
    } /* of if strlen loop */
  } /* of for i loop */
} /* of sub_eqns* */

/*****
Function Name : get_eqns()
Prototype Definition: void gt_eqns(DFA *, int, int, char ***)
Description:
  This routine forms the equation for each state of the DFA by using the
  method in transformation T6 of Section 3.4.6.

  Initially, the dfa state-input 2D transition table is transformed into a
  state-state 2D transition table. Each table entry is an RE obtained by the
  union of all input symbols on which the DFA moves from the state in the row
  entry to the state in the column entry. Then the equations matrix is
  constructed by transposing the above state-to-state matrix.
*****/
void get_eqns(num)
SHORT num;
{
  SHORT i,j,k,x;
  char str1[MAX_RE];
  Eqtype *ptr1, *ptr2;

  /* Obtain each entry of the state-stae matrix from the DFA transition
  table */
  for(i=1;i<=num;i++)
    for(j=0;j<=MAX_COLS-2;j++) {
      if ((x=dfa->table[i][j]) == PHI) continue;
      strcpy(str1,itos(symbset[j]));
      strcat(str1,"+");
      eqlistcat(x,i,str1);
    } /* for j */
  for(i=1;i<=num;i++)
    for(j=1;j<=num;j++)
      if ((ptr1=ineqlist(i,j)) != NULL) {
        strcpy(str1,"(");
        k = strlen(ptr1->eqn)-1;
        ptr1->eqn[k] = 0;
        /* removing the '+' at the end */
        if (k>1) {

```

```

                strcat(str1,ptr1->eqn);
                strcat(str1,"");
                strcpy(ptr1->eqn,str1);
            } /* if k */
        } /* if */
    for (i=1;i<=num;i++)
        if (i==dfa->start) {
            eqlistcpy(i,num+1,"e");
            break;
        }
} /* of get_eqns */

/*****
Function Definition: Eqtype *ineqlist( SHORT, SHORT)
Description:
    This routine checks if the i,j entry is in the eqlist. If so, it returns
    a pointer to that entry in the eqlist, else a null pointer.
*****/
Eqtype *ineqlist(i,j)
SHORT i,j;
{
    Eqtype *ptr1;
    ptr1 = eqnmat;
    while (ptr1 != NULL) {
        if ((ptr1->row == i) && (ptr1->col == j))
            /* check for i,j entry */
            break;
        else ptr1 = ptr1->next;
    }
    return(ptr1);
}

/*****
Function Definition: void eqlistcat(SHORT, SHORT, char *)
Description:
    This routine checks if the i,j entry exists in the eqlist. If so, it
    concatenates 'str' to the re of the i,j entry. If not, it creates
    a new i,j entry at the end of the eqlist, and concatenates 'str' to the
    re of this new entry.
*****/
void eqlistcat(i,j,str)
SHORT i,j;
char str[];
{
    Eqtype *ptr1;
    if ((ptr1=ineqlist(i,j)) != NULL) /* i,j entry exists in eqlist */
        strcat(ptr1->eqn,str);
    else { /* i,j entry does not exist in eqlist */
        ptr1 = eqcreate(i,j);
        strcat(ptr1->eqn,str);
    }
}

/*****
Function Definition: void eqlistcpy(SHORT, SHORT, char *)
Description:
    This routine checks if the i,j entry exists in the eqlist. If so, it
    copies 'str' to the re of the i,j entry. If not, it creates
    a new i,j entry at the end of the eqlist, and copies 'str' to the
    re of this new entry.
*****/
void eqlistcpy(i,j,str)
SHORT i,j;
char str[];
{
    Eqtype *ptr1;
    if ((ptr1=ineqlist(i,j)) != NULL) /* i,j entry exists in eqlist */
        strcpy(ptr1->eqn,str);
    else { /* i,j entry does not exist in eqlist */
        ptr1 = eqcreate(i,j);
        strcpy(ptr1->eqn,str);
    }
}

```



```

}
/*****
Function Definition: Eqtype *eqcreate(SHORT, SHORT)
Description:
  This routine creates the i,j entry at the end of the eqnmat eqlist, and
  returns a pointer to this newly created entry.
*****/
Eqtype *eqcreate(i,j)
SHORT i,j;
{
  Eqtype *ptr1;
  if (eqnmat == NULL) { /* if eqlist empty, create new */
    eqnmat = (Eqtype *) malloc(sizeof(Eqtype));
    ptr1 = eqnmat;
  }
  else { /* if eqlist not empty, create at end */
    ptr1 = eqnmat;
    while (ptr1->next != NULL)
      ptr1 = ptr1->next;
    ptr1->next = (Eqtype *) malloc(sizeof(Eqtype));
    ptr1 = ptr1->next;
  } /* else */

  ptr1->row = i;
  ptr1->col = j;
  ptr1->next = NULL;
  strcpy(ptr1->eqn, "");
  return(ptr1);
}
/*****
Function Definition: void eqdelete(SHORT, SHORT);
Description:
  This routine deletes the i,j entry if it exists from the eqnmat eqlist.
*****/
void eqdelete(i,j)
SHORT i,j;
{
  Eqtype *front,*back;
  front = eqnmat;
  while (front != NULL) {
    if ((front->row == i) && (front->col == j)) {
      if (front == eqnmat)
        eqnmat = front->next;
      else
        back->next = front->next;
      front->next = NULL;
      free(front);
      front = NULL;
    } /* if */
    else {
      back = front;
      front = front->next;
    }
  } /* while */
}
void free_eqnmat()
{
  Eqtype *ptr1, *freeptr;
  if (eqnmat != NULL) {
    ptr1 = eqnmat;
    while (ptr1 != NULL) {
      freeptr = ptr1;
      ptr1 = ptr1->next;
      freeptr->next = NULL;
      free(freeptr);
    } /* while */
    eqnmat = NULL;
  } /* if */
}

```

```

OTHER.C
#include "node.h"
/*****
This file (other.c) contains the following NODE routines:
    mem2D(),mem3D()      Dynamic Memory Allocation routines
    free2D(),free3D()   Dynamic Memory Deallocation routines
    rm_redun()         Remove redundancies in the RE
*****/
/* Dynamic Memory Allocation routines for 2-D and 3-D arrays used
   in the program */

void mem2D(array,n1,n2)
STTYPE *array[MAX_COLS];
int n1,n2;
{
    int i,j;
    for (i=0; i<n1; i++) {
        if ((array[i] = (STTYPE *) malloc(n2 * sizeof(STTYPE))) == NULL)
            exit(1);
        strcpy(array[i],"");
    }
}

void mem3D(array,n1,n2,n3)
STTYPE *array[][MAX_COLS];
int n1,n2,n3;
{
    int i,j;
    for (i=0; i<n1; i++)
        for (j=0; j<n2; j++) {
            if ((array[i][j] = (STTYPE *) malloc(n3 * sizeof(STTYPE))) == NULL)
                exit(1);
            strcpy(array[i][j],"");
        }
}

void free3D(array,n1,n2)
STTYPE *array[][MAX_COLS];
int n1,n2;
{
    int i,j;
    STTYPE *temp;
    for (i=0; i<n1; i++)
        for (j=0; j<n2; j++) {
            temp = array[i][j];
            free(temp);
        }
}

void free2D(array,n1)
STTYPE *array[MAX_COLS];
int n1;
{
    int i;
    STTYPE *temp;
    for (i=0; i<n1; i++) {
        temp = array[i];
        free(temp);
    }
}

strindex(s,c)
char *s,c;
{
    int i;
    for(i=0; ; i++) {
        if (s[i] == c) return(i);
        if (s[i] == '\0') return(0);
    }
}

char *itos(num)
int num;
{
    char str[5];
    str[0] = num;
}

```

```

        str[1] = '\0';
        return(str);
    }

/*****
Function Definition; void rm_repeat(str)
Description:
    Removes all repeated characters in the given string.
*****/
void rm_repeat(str)
char    str[];
{
    char    str1[MAX_NFA],
           str2[MAX_NFA],
           ch;
    int     i;

    strcpy(str1,str);
    strcpy(str2,"");
    for (i=0; (ch=str1[i]) != 0; i++) {
        if (!strchr(str2,ch))
            strcat(str2,itos(ch));
    }
    strcpy(str,str2);
}

/*****
Function Definition: char *strstr( char *, char * )
Description:
    Finds out if a string is contained in another string and returns a
    pointer to the string found
*****/
char *strstr(cs,ct)
char *cs,*ct;
{
    char *ptr;
    int len;

    ptr = cs;
    len = strlen(ct);

    while (*ptr != '\0') {
        if (strncmp(ptr,ct,len) == 0)
            return(ptr);
        else ptr++;
    }

    return(NULL);
}

/*****
Function Definition: void rm_redun(char *)
Description:
    Remove redundant terms like ()* (+)* (0)*, (1)*, 00*, 11*, etc.
    and also redundant parantheses
*****/
void rm_redun(re)
char re[MAX_RE];
{
    int     i=0,p=0,
           done=FALSE,plus=FALSE,
           pos1=0,pos2=0;

    char    stack[MAX_RE],
           temp_re[MAX_RE],
           *ptr1;

    done = FALSE;
    while (!done) {
        if ((ptr1=strstr(re,"()*")) != NULL) {
            strcpy(temp_re,ptr1+3);
            *ptr1 = '\0';
            strcat(re,temp_re);
        }
    }
}

```

```

else if ((ptr1=strstr(re,"(+)")) != NULL) {
    strcpy(temp_re,ptr1+3);
    *ptr1 = '\0';
    strcat(re,temp_re);
}
else if ((ptr1=strstr(re,"(+)**")) != NULL) {
    strcpy(temp_re,ptr1+4);
    *ptr1 = '\0';
    strcat(re,temp_re);
}
else if ((ptr1=strstr(re,"+") != NULL) {
    strcpy(temp_re,ptr1+1);
    *ptr1 = '\0';
    strcat(re,temp_re);
}
else if ((ptr1=strstr(re,"(+)") != NULL) {
    strcpy(temp_re,ptr1+2);
    *(ptr1+1) = '\0';
    strcat(re,temp_re);
}
else if ((ptr1=strstr(re,"e0")) != NULL) {
    strcpy(temp_re,ptr1+1);
    *(ptr1) = '\0';
    strcat(re,temp_re);
}
else if ((ptr1=strstr(re,"e1")) != NULL) {
    strcpy(temp_re,ptr1+1);
    *(ptr1) = '\0';
    strcat(re,temp_re);
}
else if ((ptr1=strstr(re,"0e")) != NULL) {
    strcpy(temp_re,ptr1+2);
    *(ptr1+1) = '\0';
    strcat(re,temp_re);
}
else if ((ptr1=strstr(re,"1e")) != NULL) {
    strcpy(temp_re,ptr1+2);
    *(ptr1+1) = '\0';
    strcat(re,temp_re);
}
else if ((ptr1=strstr(re,"(0)**")) != NULL) {
    strcpy(temp_re,ptr1+4);
    *(ptr1) = '\0';
    strcat(re,"0**");
    strcat(re,temp_re);
}
else if ((ptr1=strstr(re,"(1)**")) != NULL) {
    strcpy(temp_re,ptr1+4);
    *ptr1 = '\0';
    strcat(re,"1**");
    strcat(re,temp_re);
}
else done = TRUE;
} /* of while */

/*
if (DEBUG) printf("\n ** DEBUG RE = %s\n",re);
*/
while (re[pos1] != '\0') {
    if (re[pos1] == '(') {
        stack[p++] = pos1;
        re[pos1] = ' ';
    }

    if (re[pos1] == ')') {
        plus = FALSE;
        pos2 = stack[--p];
        for (i=0;i<pos1-pos2;i++)
            if (re[pos2+i] == '+') {
                plus = TRUE;
                re[pos2+i] = 'P';
            }
        if (re[pos1+1] == '*') plus = TRUE;
    }
}

```

```
        if (plus)
            re[pos2] = '(';
        else re[pos1] = ',';
    } /* of if ptr1 = ' */
    pos1++;
} /* of while */
for (pos1=0;re[pos1] != '\0';pos1++)
    if (re[pos1] == 'P') re[pos1] = '+';

while ((ptr1=strstr(re, " ")) != NULL) {
    strcpy(temp_re,ptr1+1);
    *ptr1 = '\0';
    strcat(re,temp_re);
}

/*
    if (DEBUG) printf("\n ** DEBUG RE = %s\n",re);
*/
} /* of rm_redun */
```

APPENDIX C

EXECUTION DETAILS THROUGH A CYCLE OF TRANSFORMATION

The given RE is (0*+1*)(01)
 The post-fix expression is: 0*1*+01..
 Estimate on states = 22

Details of Partitioning Approach A

NOTE: "*" has only one operand

Level 0 ==> . . 1 0 + * 1 * 0
 Level 1 ==> . T0 + T1 T2
 Level 2 ==> . T0 T3
 Level 3 ==> T4

Last task is T4 which represents the root node

Task Graph produced by Partitioning Approach A

Number of Levels: 3 Number of tasks: 5

Task	Operation	Label	Predecessors	Successors	Processor
T0	0.1	1	NONE	T4	0
T1	1*0	2	NONE	T3	1
T2	0*0	2	NONE	T3	2
T3	T2+T1	1	T2 T1	T4	3
T4	T3.T0	0	T3 T0	NONE	4

INITIAL LABEL TABLE

Num Processors: 2 Max levels: 3

Num Tasks	Tasks
1	T4
2	T0 T3
2	T1 T2

ADJUSTED LABEL TABLE

Num Processors: 2 Max levels: 3

Num Tasks	Tasks
1	T4
2	T0 T3
2	T1 T2

Schedule Obtained by Algorithm A BEFORE and AFTER Optimization

P0	T 1	T 0	T 4
P1	T 2	T 3	phi
	0	1	2

P0	T 1	T 0	phi
P1	T 2	T 3	T 4
	0	1	2

***** FSA after STEP 1 (NFA) *****
Inputs

	0	1	e
S	q 1 { }	{ }	{q2, q6, }
	q 2 { }	{ }	{q3, q5, }
	q 3 {q4, }	{ }	{ }
	q 4 { }	{ }	{q5, q3, }
	q 5 { }	{ }	{q10, }
	q 6 { }	{ }	{q7, q9, }
	q 7 { }	{q8, }	{ }
	q 8 { }	{ }	{q9, q7, }
	q 9 { }	{ }	{q10, }
	q10 { }	{ }	{q11, }
	q11 {q12, }	{ }	{ }
	q12 { }	{ }	{q13, }
	q13 { }	{q14, }	{ }
F	q14 { }	{ }	{ }

*** Transformation T2 - Removing e-moves *****

```

e_CLOSURE( q1 ) : {q1 q2 q6 q3 q5 q7 q9 q10 q11 }
e_CLOSURE( q2 ) : {q2 q3 q5 q10 q11 }
e_CLOSURE( q3 ) : {q3 }
e_CLOSURE( q4 ) : {q4 q5 q3 q10 q11 }
e_CLOSURE( q5 ) : {q5 q10 q11 }
e_CLOSURE( q6 ) : {q6 q7 q9 q10 q11 }
e_CLOSURE( q7 ) : {q7 }
e_CLOSURE( q8 ) : {q8 q9 q7 q10 q11 }
e_CLOSURE( q9 ) : {q9 q10 q11 }
e_CLOSURE( q10 ) : {q10 q11 }
e_CLOSURE( q11 ) : {q11 }
e_CLOSURE( q12 ) : {q12 q13 }
e_CLOSURE( q13 ) : {q13 }
e_CLOSURE( q14 ) : {q14 }

```

***** NFA after Removing e-moves *****
 Inputs

		0	1	e
S	q 1	{q4, q5, q3, q10, q11, q12, q13, }	{q8, q9, q7, q10, q11}	{}
	q 2	{q4, q5, q3, q10, q11, q12, q13}	{}	{}
	q 3	{q4, q5, q3, q10, q11}	{}	{}
	q 4	{q4, q5, q3, q10, q11, q12, q13}	{}	{}
	q 5	{q12, q13}	{}	{}
	q 6	{q12, q13}	{q8, q9, q7, q10, q11}	{}
	q 7	{}	{q8, q9, q7, q10, q11}	{}
	q 8	{q12, q13}	{q8, q9, q7, q10, q11}	{}
	q 9	{q12, q13}	{}	{}
	q10	{q12, q13}	{}	{}
	q11	{q12, q13}	{}	{}
	q12	{}	{q14}	{}
	q13	{}	{q14}	{}
F	q14	{}	{}	{}

*** STEP 3 - Correspondence between DFA and NFA States ***

DFA State		NFA State set
Q 1	<==>	{ q 1 }
Q 2	<==>	{ q 4 q 5 q 3 q10 q11 q12 q13 }
Q 3	<==>	{ q 8 q 9 q 7 q10 q11 }
Q 4	<==>	{ q14 }
Q 5	<==>	{ q12 q13 }

***** FSA after STEP 3 (DFA) *****

	States	Inputs	
		0	1
S	q 1	q 2	q 3
	q 2	q 2	q 4
	q 3	q 5	q 3
F	q 4	phi	phi
	q 5	phi	q 4

*** STEP 4 - Snap shot of Marking table ***

(2,1)-X
 (3,1)-X (3,2)-X
 (4,1)-X (4,2)-X (4,3)-X
 (5,1)-X (5,2)-X (5,3)-X (5,4)-X
 (6,1)-X (6,2)-X (6,3)-X (6,4)-X (6,5)-X

***** Minimized DFA after STEP 4 *****

States		Inputs	
		0	1
S	q 1	q 2	q 3
	q 2	q 2	q 4
	q 3	q 5	q 3
F	q 4	q 6	q 6
	q 5	q 6	q 4
	q 6	q 6	q 6

***** STEP 5 - DFA to RE *****

	Initial G4 Subgraph					
State	1	2	3	4	5	6
State 1 :		0	1			
State 2 :		0		1		
State 3 :			1		0	
State 4 :						(0+1)
State 5 :				1		0
State 6 :						(0+1)

	G4 Subgraph after deleting internal states					
State	1	2	3	4	5	6
State 1 :				(00*1+11*01)		
State 2 :						
State 3 :						
State 4 :						
State 5 :						
State 6 :						

Final RE: (00*1+11*01)

APPENDIX D

SAMPLE RUNS FOR A CONVERGENT CASE

(Note: Both Algorithms A and C produce the same results)

ITERATION0
Given RE: 0^*+1^*
Final RE: 00^*+11^*+e
ITERATION1
Given RE: 00^*+11^*+e
Final RE: 00^*+11^*+e
ITERATION2
Given RE: 00^*+11^*+e
Final RE: 00^*+11^*+e

ITERATION0
Given RE: 0^*1^*
Final RE: $0^*+0^*11^*$
ITERATION1
Given RE: $0^*+0^*11^*$
Final RE: $0^*+0^*11^*$
ITERATION2
Given RE: $0^*+0^*11^*$
Final RE: $0^*+0^*11^*$

ITERATION0
Given RE: $(00+01+10+11)^*$
Final RE: $(0+1)(0+1)^*$
ITERATION1
Given RE: $(0+1)(0+1)^*$
Final RE: $(0+1)((0+1))^*$
ITERATION2
Given RE: $(0+1)((0+1))^*$
Final RE: $(0+1)((0+1))^*$

ITERATION0
Given RE: $(01+11)00(1+10)$
Final RE: $(0+1)1001+(0+1)10010$
ITERATION1
Given RE: $(0+1)1001+(0+1)10010$

Given RE: $(0+1)1001+(0+1)10010$

Final RE: $(0+1)1001+(0+1)10010$

ITERATION2

Given RE: $(0+1)1001+(0+1)10010$

Final RE: $(0+1)1001+(0+1)10010$

ITERATION0

Given RE: $(00+11)(01+10)(11+01)$

Final RE: $((000+110)1(0+1)1+(001+111)0(0+1)1)$

ITERATION1

Given RE: $((000+110)1(0+1)1+(001+111)0(0+1)1)$

Final RE: $(0(001+010)(0+1)1+1(101+110)(0+1)1)$

ITERATION2

Given RE: $(0(001+010)(0+1)1+1(101+110)(0+1)1)$

Final RE: $(0(001+010)(0+1)1+1(101+110)(0+1)1)$

ITERATION0

Given RE: $(0111+000*11+11)+011$

Final RE: $((11+000*11)+0111)+011$

ITERATION1

Given RE: $((11+000*11)+0111)+011$

Final RE: $((11+000*11)+0111)+011$

ITERATION2

Given RE: $((11+000*11)+0111)+011$

Final RE: $((11+000*11)+0111)+011$

ITERATION0

Given RE: $101+1*10+010$

Final RE: $((010+101)+111*0)+10$

ITERATION1

Given RE: $((010+101)+111*0)+10$

Final RE: $((111*0+010)+101)+10$

ITERATION2

Given RE: $((111*0+010)+101)+10$

Final RE: $((111*0+010)+101)+10$

ITERATION0

Given RE: $010+101+010+1*10+10$

Final RE: $((010+101)+111*0)+10$

ITERATION1

Given RE: $((010+101)+111*0)+10$

Final RE: $((111*0+010)+101)+10$

ITERATION2

Given RE: $((111*0+010)+101)+10$

Final RE: $((111*0+010)+101)+10$

ITERATION0

Given RE: $(0*1*)(1+0)$

Final RE: $00*+(1+00*1)1*+(1+00*1)1*0$

ITERATION1

Given RE: $00^*+(1+00^*1)1^*+(1+00^*1)1^*0$ Final RE: $00^*+(1+00^*1)1^*+(1+00^*1)1^*0$

ITERATION2

Given RE: $00^*+(1+00^*1)1^*+(1+00^*1)1^*0$ Final RE: $00^*+(1+00^*1)1^*+(1+00^*1)1^*0$

ITERATION0

Given RE: 0^*1^*00 Final RE: $000^*+((1+01)1^*0+000^*11^*0)0$

ITERATION1

Given RE: $000^*+((1+01)1^*0+000^*11^*0)0$ Final RE: $000^*+(11^*00+0(11^*0+00^*11^*0)0)$

ITERATION2

Given RE: $000^*+(11^*00+0(11^*0+00^*11^*0)0)$ Final RE: $000^*+(11^*00+0(11^*0+00^*11^*0)0)$

ITERATION0

Given RE: $1^*(0+1)0^*$ Final RE: $(0+11^*0)0^*+11^*$

ITERATION1

Given RE: $(0+11^*0)0^*+11^*$ Final RE: $(0+11^*0)0^*+11^*$

ITERATION2

Given RE: $(0+11^*0)0^*+11^*$ Final RE: $(0+11^*0)0^*+11^*$

ITERATION0

Given RE: $(0^*1^*)^*$ Final RE: $(0+1)^*$

ITERATION1

Given RE: $(0+1)^*$ Final RE: $(0+1)^*$

ITERATION2

Given RE: $(0+1)^*$ Final RE: $(0+1)^*$

ITERATION0

Given RE: $(0^*1^*(0^*+1^*))$ Final RE: $0^*+0^*11^*+0^*11^*00^*$

ITERATION1

Given RE: $0^*+0^*11^*+0^*11^*00^*$ Final RE: $0^*+0^*11^*+0^*11^*00^*$

ITERATION2

Given RE: $0^*+0^*11^*+0^*11^*00^*$ Final RE: $0^*+0^*11^*+0^*11^*00^*$

ITERATION0

Given RE: $(0^*+1^*)^*$

Final RE: $(0+1)^*$
 ITERATION1
 Given RE: $(0+1)^*$
 Final RE: $(0+1)^*$
 ITERATION2
 Given RE: $(0+1)^*$
 Final RE: $(0+1)^*$

ITERATION0
 Given RE: $(0^*+1^*+(00)^*+(11)^*)$
 Final RE: 00^*+11^*+e
 ITERATION1
 Given RE: 00^*+11^*+e
 Final RE: 00^*+11^*+e
 ITERATION2
 Given RE: 00^*+11^*+e
 Final RE: 00^*+11^*+e

ITERATION0
 Given RE: $(00+11)^*$
 Final RE: $(00+11)^*$
 ITERATION1
 Given RE: $(00+11)^*$
 Final RE: $(00+11)^*$
 ITERATION2
 Given RE: $(00+11)^*$
 Final RE: $(00+11)^*$

ITERATION0
 Given RE: $(0+00+11)^*$
 Final RE: $(0+11)^*$
 ITERATION1
 Given RE: $(0+11)^*$
 Final RE: $(0+11)^*$
 ITERATION2
 Given RE: $(0+11)^*$
 Final RE: $(0+11)^*$

ITERATION0
 Given RE: $(0(00+11))^*$
 Final RE: $(000+011)^*$
 ITERATION1
 Given RE: $(000+011)^*$
 Final RE: $(000+011)^*$
 ITERATION2
 Given RE: $(000+011)^*$
 Final RE: $(000+011)^*$

ITERATION0

Given RE: $(0+1+00+11)^*$

Final RE: $(0+1)^*$

ITERATION1

Given RE: $(0+1)^*$

Final RE: $(0+1)^*$

ITERATION2

Given RE: $(0+1)^*$

Final RE: $(0+1)^*$

ITERATION0

Given RE: $(01+(00+11))^*$

Final RE: $(0(0+1)+11)^*$

ITERATION1

Given RE: $(0(0+1)+11)^*$

Final RE: $(0(0+1)+11)^*$

ITERATION2

Given RE: $(0(0+1)+11)^*$

Final RE: $(0(0+1)+11)^*$

ITERATION0

Given RE: $(000+11)^*$

Final RE: $(11+000)^*$

ITERATION1

Given RE: $(11+000)^*$

Final RE: $(11+000)^*$

ITERATION2

Given RE: $(11+000)^*$

Final RE: $(11+000)^*$

ITERATION0

Given RE: $(000+111)^*$

Final RE: $(000+111)^*$

ITERATION1

Given RE: $(000+111)^*$

Final RE: $(000+111)^*$

ITERATION2

Given RE: $(000+111)^*$

Final RE: $(000+111)^*$

APPENDIX E

SAMPLE RUNS FOR A DIVERGENT CASE

(Note: Both Algorithms A and C produce the same results)

ITERATION0
Given RE: 0^*+1^*
Final RE: $e+00^*+11^*$
ITERATION1
Given RE: $e+00^*+11^*$
Final RE: 00^*+11^*+e
ITERATION2
Given RE: 00^*+11^*+e
Final RE: 00^*+11^*+e

ITERATION0
Given RE: 0^*1^*
Final RE: $0^*+0^*11^*$
ITERATION1
Given RE: $0^*+0^*11^*$
Final RE: $0^*+0^*11^*$
ITERATION2
Given RE: $0^*+0^*11^*$
Final RE: $0^*+0^*11^*$

ITERATION0
Given RE: $(00+01+10+11)^*$
Final RE: $((0+1)(0+1))^*$
ITERATION1
Given RE: $((0+1)(0+1))^*$
Final RE: $((0+1)(0+1))^*$
ITERATION2
Given RE: $((0+1)(0+1))^*$
Final RE: $((0+1)(0+1))^*$

ITERATION0
Given RE: $(01+11)00(1+10)$
Final RE: $(0+1)1001+(0+1)10010$
ITERATION1
Given RE: $(0+1)1001+(0+1)10010$
Final RE: $(0+1)1001+(0+1)10010$

Final RE: $(0+1)1001+(0+1)10010$

ITERATION2

Given RE: $(0+1)1001+(0+1)10010$

Final RE: $(0+1)1001+(0+1)10010$

ITERATION0

Given RE: $(00+11)(01+10)(11+01)$

Final RE: $(000+110)1(0+1)1+(001+111)0(0+1)1$

ITERATION1

Given RE: $(000+110)1(0+1)1+(001+111)0(0+1)1$

Final RE: $0(001+010)(0+1)1+1(101+110)(0+1)1$

ITERATION2

Given RE: $0(001+010)(0+1)1+1(101+110)(0+1)1$

Final RE: $0(001+010)(0+1)1+1(101+110)(0+1)1$

ITERATION0

Given RE: $10+(0+11)0*11$

Final RE: $10+(00*1+110*1)1$

ITERATION1

Given RE: $10+(00*1+110*1)1$

Final RE: $(00*1+110*1)1+10$

ITERATION2

Given RE: $(00*1+110*1)1+10$

Final RE: $(00*1+110*1)1+10$

ITERATION0

Given RE: $(0111+000*11+11)+011$

Final RE: $11+000*11+0111+011$

ITERATION1

Given RE: $11+000*11+0111+011$

Final RE: $11+000*11+0111+011$

ITERATION2

Given RE: $11+000*11+0111+011$

Final RE: $11+000*11+0111+011$

ITERATION0

Given RE: $00+11(0+1)*11+00$

Final RE: $110*1(00*1)*1(1+00*1(00*1)*1)*+00$

ITERATION1

Given RE: $110*1(00*1)*1(1+00*1(00*1)*1)*+00$

Final RE: $110*1(00*1)*1(1+00*1(00*1)*1)*+00$

ITERATION2

Given RE: $110*1(00*1)*1(1+00*1(00*1)*1)*+00$

Final RE: $110*1(00*1)*1(1+00*1(00*1)*1)*+00$

ITERATION0

Given RE: $101+1*10+010$

Final RE: $010+101+111*0+10$

ITERATION1

Given RE: $010+101+111*0+10$

Final RE: $111*0+010+101+10$

ITERATION2

Given RE: $111*0+010+101+10$

Final RE: $111*0+010+101+10$

ITERATION0

Given RE: $010+101+010+1*10+10$

Final RE: $010+101+111*0+10$

ITERATION1

Given RE: $010+101+111*0+10$

Final RE: $111*0+010+101+10$

ITERATION2

Given RE: $111*0+010+101+10$

Final RE: $111*0+010+101+10$

ITERATION0

Given RE: $010+1*10+010+101+10+101$

Final RE: $010+101+111*0+10$

ITERATION1

Given RE: $010+101+111*0+10$

Final RE: $111*0+010+101+10$

ITERATION2

Given RE: $111*0+010+101+10$

Final RE: $111*0+010+101+10$

ITERATION0

Given RE: $0*1*00$

Final RE: $000*+((1+011*)0+000*11*0)0$

ITERATION1

Given RE: $000*+((1+011*)0+000*11*0)0$

Final RE: $000*+0(1+00*11*)00+100$

ITERATION2

Given RE: $000*+0(1+00*11*)00+100$

Final RE: $000*+100+0(100+00*11*00)$

ITERATION0

Given RE: $(0*1*0*1*)*$

Final RE: $((0+1))^*$

ITERATION1

Given RE: $((0+1))^*$

Final RE: $((0+1))^*$

ITERATION2

Given RE: $((0+1))^*$

Final RE: $((0+1))^*$

ITERATION0

Given RE: $(0*1*(0*+1*))$

Final RE: $0*+0*11*+0*11*00*$

ITERATION1

Given RE: $0^*+0^*11^*+0^*11^*00^*$ Final RE: $0^*+0^*11^*+0^*11^*00^*$

ITERATION2

Given RE: $0^*+0^*11^*+0^*11^*00^*$ Final RE: $0^*+0^*11^*+0^*11^*00^*$

ITERATION0

Given RE: $(0^*+1^*+(00)^*+(11)^*)$ Final RE: 00^*+11^*+e

ITERATION1

Given RE: 00^*+11^*+e Final RE: 00^*+11^*+e

ITERATION2

Given RE: 00^*+11^*+e Final RE: 00^*+11^*+e

ITERATION0

Given RE: $(00+11)^*$ Final RE: $e+(0(00)^*+(1+0(00)^*01(11+10(00)^*01)^*)10(00)^*0+(1+0(00)^*01(11+10(00)^*01)^*)1$

ITERATION1

Given RE: $e+(0(00)^*+(1+0(00)^*01(11+10(00)^*01)^*)10(00)^*0+(1+0(00)^*01(11+10(00)^*01)^*)1$ Final RE: $00(00)^*1(1(00)^*1)^*1(00)^*+00(00)^*+110(00)^*0+11+e$

ITERATION2

Given RE: $00(00)^*1(1(00)^*1)^*1(00)^*+00(00)^*+110(00)^*0+11+e$ Final RE: $00(00)^*1(1(00)^*1)^*1(00)^*+00(00)^*+110(00)^*0+11+e$

ITERATION0

Given RE: $(0(00+11))^*$ Final RE: $e+(00(000)^*+(01+00(000)^*001(101+100(000)^*001)^*)100(000)^*0+(01+00(000)^*001(101+100(000)^*001)^*)1$

ITERATION1

Given RE: $e+(00(000)^*+(01+00(000)^*001(101+100(000)^*001)^*)100(000)^*0+(01+00(000)^*001(101+100(000)^*001)^*)1$ Final RE: $(0000(000)^*1(10(000)^*1)^*10(000)^*+0000(000)^*)00+0000(000)^*1(10(000)^*1)^*1+000+01100(000)^*0+011+e$

ITERATION2

Given RE: $(0000(000)^*1(10(000)^*1)^*10(000)^*+0000(000)^*)00+0000(000)^*1(10(000)^*1)^*1+000+01100(000)^*0+011+e$ Final RE: $(0000(000)^*1(10(000)^*1)^*10(000)^*+0000(000)^*)00+0000(000)^*1(10(000)^*1)^*1+000+01100(000)^*0+011+e$

ITERATION0

Given RE: $(01(00+11))^*$ Final RE: $e+(010(0010)^*+(011+010(0010)^*0011(1011+1010(0010)^*0011)^*)1010(0010)^*0+(011+010(0010)^*0011(1011+1010(0010)^*0011)^*)1$

ITERATION1

Given RE: $e+(010(0010)^*+(011+010(0010)^*0011(1011+1010(0010)^*0011)^*)$

1010(0010)*0+(011+010(0010)*0011(1011+1010(0010)*0011)*1
 Final RE: (010001(0001)*1(101(0001)*1)*101(0001)*+010001(0001)*00+
 010001(0001)*1(101(0001)*1)*1+0100+0111010(0010)*0+0111+e

ITERATION2

Given RE: (010001(0001)*1(101(0001)*1)*101(0001)*+010001(0001)*00+
 010001(0001)*1(101(0001)*1)*1+0100+0111010(0010)*0+0111+e

Final RE: (010001(0001)*1(101(0001)*1)*101(0001)*+010001(0001)*00+
 010001(0001)*1(101(0001)*1)*1+0100+0111010(0010)*0+0111+e

ITERATION0

Given RE: (01+(00+11))*

Final RE: e+(0((0+1)0)*+(1+0((0+1)0)*(0+1)1(11+10((0+1)0)*(0+1)1)*
 10((0+1)0)*(0+1)+1+0((0+1)0)*(0+1)1(11+100+1)0)*0+1)1)*1

ITERATION1

Given RE: e+(0((0+1)0)*+(1+0((0+1)0)*(0+1)1(11+10((0+1)0)*(0+1)1)*
 10((0+1)0)*(0+1)+1+0((0+1)0)*(0+1)1(11+100+1)0)*0+1)1)*1

Final RE: e+(0((0+1)0)*+(1+0((0+1)0)*(0+1)1(11+10((0+1)0)*(0+1)1)*
 10((0+1)0)*(0+1)+1+0((0+1)0)*(0+1)1(11+100+1)0)*0+1)1)*1

ITERATION0

Given RE: (000+11)*

Final RE: e+(1(11)*+(00+1(11)*100(000+01(11)*100)*01(11)*1+
 (00+1(11)*100(000+01(11)*100)*0

ITERATION1

Given RE: e+(1(11)*+(00+1(11)*100(000+01(11)*100)*01(11)*1+
 (00+1(11)*100(000+01(11)*100)*0

Final RE: 0001(11)*1+000+11(11)*00(0(11)*00)*0(11)*+11(11)*+e

ITERATION2

Given RE: 0001(11)*1+000+11(11)*00(0(11)*00)*0(11)*+11(11)*+e

Final RE: 11(11)*00(0(11)*00)*0(11)*+11(11)*+0001(11)*1+000+e

ITERATION0

Given RE: (000+111)*

Final RE: e+(00(000)*+(11+00(000)*011(111+100(000)*011)*100(000)*
 0+(11+00(000)*011(111+100(000)*011)*1

ITERATION1

Given RE: e+(00(000)*+(11+00(000)*011(111+100(000)*011)*100(000)*
 0+(11+00(000)*011(111+100(000)*011)*1

Final RE: 000(000)*11(1(000)*11)*1(000)*+000(000)*+11100(000)*0+111+e

ITERATION2

Given RE: 000(000)*11(1(000)*11)*1(000)*+000(000)*+11100(000)*0+111+e

Final RE: 000(000)*11(1(000)*11)*1(000)*+000(000)*+11100(000)*0+111+e

APPENDIX F

COMPARISON OF PERFORMANCE MEASURES

NOTE: All tasks have unit execution times.
Width specifies the maximum number of processors that can be used.

Current RE: $((0^*+1^*)0+(0^*+1^*)1^*)((0^*+1^*)0^*+0)$

Algorithm	Width	Processors	Serial time	Parallel time	Speed up	Efficiency
A	6	2	17	9	1.889	0.944
A	6	3	17	7	2.429	0.810
A	6	4	17	6	2.833	0.708
A	6	5	17	6	2.833	0.567
A	6	6	17	5	3.400	0.567

C	3	2	9	6	1.500	0.750
C	3	3	9	5	1.800	0.600

Current RE: $(00+11)(01)+(00+11)(10)$

Algorithm	Width	Processors	Serial time	Parallel time	Speed up	Efficiency
A	4	2	11	6	1.833	0.917
A	4	3	11	5	2.200	0.733
A	4	4	11	4	2.750	0.688

C	3	2	8	5	1.600	0.800

C	3	3	8	4	2.000	0.667
---	---	---	---	---	-------	-------

Current RE: (01+10)(0+1*+00)00+(00+11)(10+0011+1*)

Algorithm	Width	Processors	Serial time	Parallel time	Speed up	Efficiency
A	5	2	22	12	1.833	0.917
A	5	3	22	9	2.444	0.815
A	5	4	22	7	3.143	0.786
A	5	5	22	7	3.143	0.629
C	4	2	18	10	1.800	0.900
C	4	3	18	7	2.571	0.857
C	4	4	18	7	2.571	0.643

Current RE: (010+100)(0+1*+00)00+(00+11)(10+0011+1*)

Algorithm	Width	Processors	Serial time	Parallel time	Speed up	Efficiency
A	5	2	24	13	1.846	0.923
A	5	3	24	9	2.667	0.889
A	5	4	24	8	3.000	0.750
A	5	5	24	7	3.429	0.686
C	4	2	20	11	1.818	0.909
C	4	3	20	8	2.500	0.833
C	4	4	20	7	2.857	0.714

VITA

Sridhar Mandyam

Candidate for the Degree of

Master of Science

Thesis: **IMPLEMENTATION OF REGULAR EXPRESSION TRANSFORMATION
ALGORITHMS ON THE HYPERCUBE**

Major Field: Computer Science

Biographical:

Personal Data: Born in Bangalore, India, June 11, 1965, the son of Krishna Swami and Vatsala.

Education: Graduated from V.R. College, Nellore, India, in May, 1982; received Master of Science (Tech.) degree in Electronics and Instrumentation from Birla Institute of Technology and Science, Pilani, India in July, 1986; received Master of Technology degree in Industrial Electronics from Karnataka Regional Engineering College, Surathkal, India in March, 1988; completed requirements for the Master of Science degree at Oklahoma State University in July, 1991.

Professional Experience: Teaching Assistant, Department of Computer Science, Oklahoma State University, August, 1988 to December, 1990; Research Assistant, ORIGINS, Business School, Oklahoma State University, June, 1989 to July, 1989.