SOFTWARE INTERFACE FOR MULTIPROCESSOR SIMULATION

Ali Makoui

December 1986
CSD-860014

UNIVERSITY OF CALIFORNIA

Los Angeles

Computer Science Department

Software Interface for Multiprocessor Simulation

Engineering Report UCLA-CSD-860014

by

Ali Makoui

Date Published - December 1986

PREFACE

The research described in this report, "Software Interface for Multiprocessor Simulation," UCLA-CSD-860014, by Ali Makoui, was carried out under the direction of Professor Walter J. Karplus in the Computer Science Department, School of Engineering and Applied Science, University of California, Los Angeles.

This project was partially sponsored by the NASA Lewis Research Center under grant NASA NAG 3-132, Professor Milos D. Ercegovac Principal Investigator; and Professor Walter J.Karplus, Co-Principal Investigator. Support was also provided by the National Science foundation under the CER grant.

This report is based on doctoral research performed by the author, with the guidance of his Doctoral Committee: Walter J. Karplus, chairman, and Milos D Ercegovac, Jack W. Carlyle, Edward C. Deland, R. Clay Sprowls.

TABLE OF CONTENTS

page

LIST OF FIGURES

page

ACKNOWLEDGMENTS

I will always feel the deepest gratitude to my advisor, Professor Walter J. Karplus, for his constant guidance, encouragement, and above all, his invaluable friendship during the course of this research.

The general subject of this dissertation originated from the initial guidance of Professor Karplus and Professor Milos D. Ercegovac, who also provided many suggestions and ideas for its development and have extensively contributed with ideas, criticisms, and attention.

Many thanks to the members of my doctoral committee, Professors R. Clay Sprowls, Edward C. Deland, Milos D. Ercegovac, Jack W. Carlyle, and Walter J. Karplus who have always shown interest and willingness to participate.

This dissertation was partially sponsored by the NASA Lewis Research Center under grant NASA NAG 3-132. I would like to especially thank Edward Milner, John Szuch, and Clint Hart for providing valuable information and support. Support was also provided by the National Science Foundation under the CER grant to the UCLA Computer Science Department. I also extend my special thanks to Teledyne Controls executive management for their support during the course of this research.

The Computer Science department of UCLA provided an ideal work environment. Its staff was always a source of help and enthusiasm. In particular, by their direct connection to this work, June Mayers, Merilyn Kell, and Vera Morgan

deserve special mention. I also wish to thank my friends Shun Cheung, Socrates Dimitriadis, and David Yeh for their help and suggestions.

Most important of all, I wish to thank my parents, Maryam and Reza, who provided loving encouragement, support, and examples of perseverance which have helped me to reach each of my past goals, academic and otherwise. I am most grateful to my lovely wife, Bita, for her continuous love, encouragement, and support. Many thanks to my sister, Susan, for her intellectual stimulation and encouragement. Finally, I thank all my friends and colleagues for being so understanding.

Ali Makoui

November 1986

ABSTRACT OF THE DISSERTATION

Software Interface for Multiprocessor Simulation

by

Ali Makoui

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1986

Professor Walter J. Karplus, Chair

The simulation of complex dynamic systems on digital computers is frequently a highly computation-intensive activity. The mathematical models to be implemented on the simulator often contain a large number of simultaneous ordinary differential equations involving the generation of nonlinear functions of the independent variable. Networks of microprocessors constitute one promising approach to obtain sufficient computational speeds, particularly where real-time operation is required. A principal obstacle to the realization of high-speed multiprocessor simulations is the absence of effective software. The research described in this dissertation is directed to the development of a methodology for the design of software systems to fulfill this objective. This work culminated in the design of a user-friendly software interface. This software package permits the user to express the mathematical simulation model in a higher-level simulation language and to execute it on a network of microprocessors. The partition of the source program among the processors is accomplished automatically. To this end, the source code is converted into a data flow graph, analyzed and divided among the processors in such a way as to minimize the overall execution time in the presence of interprocessor communication delays.

In most simulation problems, all program segments are known in advance. It is, therefore, expedient to employ a static scheduling scheme. This entails extensive preprocessing to determine an optimum scheduling strategy and to allocate all program modules to the individual processing elements prior to execution. Based upon an extensive investigation of previously proposed scheduling algorithms, a suitable algorithmic approach has been developed in the present research. This algorithm analyzes the data flow graph and determines the most appropriate sequence of the execution of the nodes comprising the graph. A heuristic is employed to allocate the balanced graph among available processors so as to minimize communication delays.

The software package permits the user to specify the characteristics of the multiprocessor network and to determine an optimum number of microprocessors for a specific problem. The effectiveness of the allocation algorithm is investigated using a number of benchmarks. A self contained user manual is provided as an appendix.

CHAPTER 1

INTRODUCTION

1.1 High-Speed Simulation

Ever expanding computation requirements have made the construction of high-speed computer systems a major challenge for both hardware and software designers. Demands for higher speed and performance continue to exceed the capabilities of technology. As new fast and inexpensive systems become available, they are used to solve new problems which were not considered or formulated for computers before. New VLSI technologies have provided inexpensive microprocessors, support chips, and custom and semicustom logics(such as gate arrays and standard cells) which allow us to build networks of microprocessors that are many times more powerful and yet less expensive than single processor mainframe systems.

There are many large real-time problems that require high-speed execution. The simulation of large systems, modeled by a set of nonlinear ordinary differential equations is one of these applications.

A wide variety of applications exist where an exact solution for a system is not possible due to cost, complexity, or size. In these cases a model that takes into

account many, but presumably not all, of the features of the system is built. This model is then analyzed and programmed on a computer and experiments are run for different initial conditions and parameter values. This is called simulation. Simulation provides the means for observing the behavior of a system when direct measurements of the system parameters are inconvenient, hazardous, or expensive.

Real-time simulation consists of obtaining the simulated parameters in the same time frame that the actual variables evolve in the system being simulated. Data must be sampled from input signals and incorporated into the numerical integration algorithm to evaluate the derivatives. Also, the result of the integration must be available for external use with minimum delay. Therefore, a simulator working in real time must be fast enough to respond immediately to signals sent from the physical devices, and generate signals at specific points in time. This requires computing speeds not attainable with conventional sequential computers.

A variety of approaches has been used to simulate large systems in real time. Analog and hybrid computers as well as digital computers have been used for this purpose. A complete and consistent classification of all these systems is rather difficult. Available classifications are loose, overlapping, and subject to debate. One approach is to use Flynn's widely used framework[FLYN 72], which classifies digital computers into four broad categories — essentially, all combinations of one to many control units and one to many sets of data.

(1) "Single-instruction single-data-stream"(SISD) systems that have one control unit working on one set of data. This organization represents most

conventional one-CPU computers available today.

(2) "Single-instruction multiple-data-stream"(SIMD) systems that consist of an array of processors executing the same instruction(having one control unit), each on a different set of data. ILLIAC IV and SOLOMON are examples of this type of systems.

(3) "Multiple-instruction Single-data-stream"(MISD) systems that consist of a number of processors each executing a certain instruction on a stream of data that flows through the system. Pipelines are examples of MISD systems.

(4) "Multiple-instruction multiple-data-stream"(MIMD) systems that consist of network of processors each executing a different sequence of instructions. MIMD systems include a wide variety of parallel and distributed systems.

Figure 1.1 shows the different types of computer organizations. Note that few architectures could be described as pure. Therefore, the classifications shown here are relatively crude. Some of the most important of these systems are discussed in the following sections.

1.2 Analog and Hybrid Computers in Simulation

Before the digital computers became popular, analog and hybrid computers were employed to attain the computing speeds needed for the real-time simulation of large systems. In analog computers, each operation is done by a separate computational unit, and all operations are performed in parallel. This means that

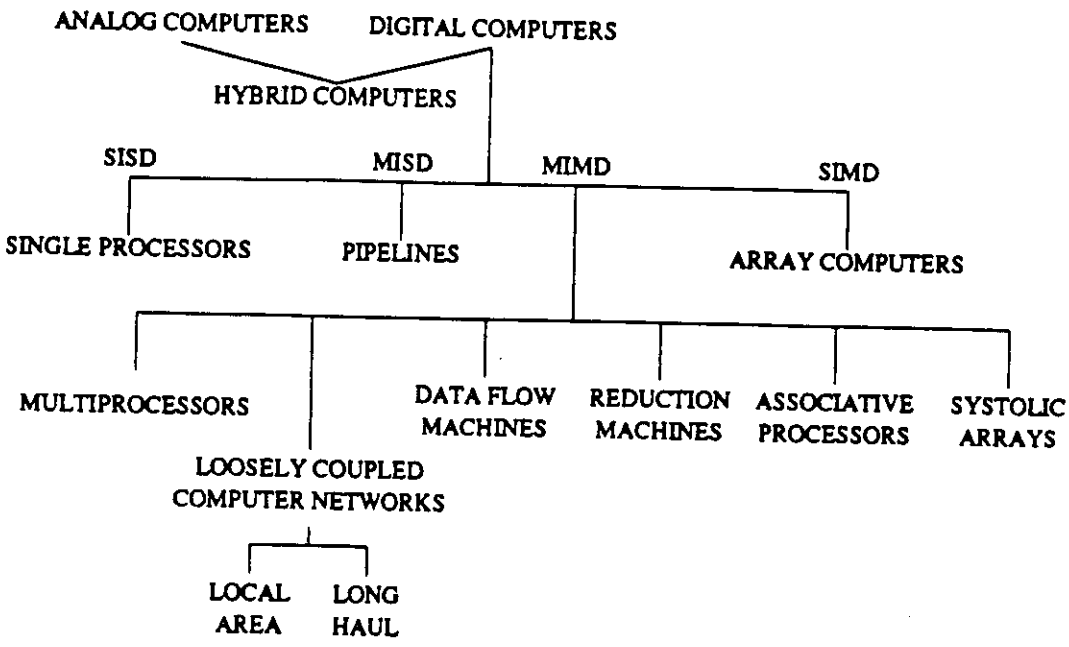


Figure 1.1 Different types of computer organizations

adding to problem complexity requires more operational units, but does not increase the execution time. In other words, the processing speed is independent of the problem complexity. This is in contrast to digital computers, where larger problems are still executed on the same hardware but require more time to solve.

Accuracy in analog computers depends on the accuracy of the components performing the computations. Nonlinear equations require nonlinear components that usually have low accuracy. Scale factors are used to represent each dependent variable by a voltage level and the independent variable by the time variable. Each operational unit is considered as a black box, capable of generating an output voltage which is a function of its input voltage. These units are wired together to form the proper equation. Program storage is more difficult than digital computers. A separate patch board has to be used for each problem.

Hybrid computers are configured by interconnecting digital and analog computers in a variety of different ways[BEKE 68]. These computers are generally used to combine the speed of an analog computer with the accuracy and flexibility of a digital computer. Analog computers can achieve desired speeds for most simulation problems. However, they have problems including maintaining many computational units, low accuracy due to nonideal behavior of the computing elements, difficulty in program and data storage, and realization of nonlinear components. Hybrid computers have additional problems of requiring the mastering of both analog and digital techniques as well as A/D and D/A conversion methods.

1.3 Digital Computers in Simulation

Recent advances in digital computer technology have almost completely phased out the use of analog and hybrid computers and have made it possible to build simulators using digital computers. In these computers, the same hardware executes programs of different complexity. The accuracy of results depends on the number of bits in registers rather than component tolerances. Program and data are easily stored in storage media. The problem with digital computers is the sequentiality of the execution. Since a limited number of operational units are used to carry out the computations, the operations are serialized. Hence, the speed is decreased as the complexity of the problem increases.

Many methods have been developed to increase the speed of computation in digital computers. One method is to increase the speed of a uniprocessor by designing high-speed circuits, reducing the number of logic levels, reducing the

number of cycles per operation, exploiting architectural features such as cache memory and RISC(reduced instruction set computers), and so on. However, the physical limitation of the circuits will eventually prevent further gains in speed. One such limitation is the distance between the CPU components. The time that is needed for electrical signal to flow between components, and hence carry the information, will eventually become greater than the time needed by these components to process the information.

Another method is to use fast coprocessors that work in conjunction with a host such as a minicomputer or a mainframe, executing complex or repetitive functions that are off-loaded from the host. These special-purpose digital devices, usually referred to as peripheral array processors, attain very high processing speeds for specialized numerical computations through extensive internal parallelism and pipelining[KARP 84a].

Yet another method is to introduce parallelism at different levels of the system hardware and software. The term parallel processing, in a very general sense, covers methods that attempt to increase speed by performing computations simultaneously. All modern computers involve some parallelism. Whether or not a computer is termed parallel is a matter of degree only.

In hardware, parallelism can be introduced at several levels. At the gate level, it involves computing all bits of a number simultaneously instead of one bit at a time. At the register level, it involves computing several words in parallel, instead of one word at a time. Finally, at the processor level, blocks of information can be

processed simultaneously.

In software, program allocation techniques are used to reduce the number of data exchanges between modules executed in parallel. Also, adaptable architectures are developed that are capable of adapting to requirements of the program via software. Some of the most popular parallel techniques are discussed below.

1.3.1 Look-Ahead Computers

This kind of machine uses several computational units and overlap the execution of several instructions. Examples of this type of machine are the IBM 360/91 and the CDC 6600. Instructions on the CDC 6600 are executed by 10 separate functional units. Up to 32 consecutive instructions are maintained in an instruction stack. A special control unit called the "scoreboard" is responsible for the selection of the registers and functional units to be used in the execution of an instruction. These computers are general purpose computers and have been used for simulation as well as other applications.

1.3.2 Peripheral Array processors

Although lower in cost than mainframe computers, minicomputers have limited processing speed for intensive calculations. This has led to the design of the peripheral array processors which are employed in host-plus-peripheral systems to enhance the capability of host machines[KARP 77]. Acting as fast coprocessors, these machines are able to perform the complex calculations needed in scientific

applications. Array processors have a unique bus structure that allows them to perform simultaneous fetch, addition, and multiplication operations. The host provides the overall system control and performs the I/O operations, while the array processor performs high-speed complex calculations.

One type of array processors, the AD-10, is especially suited for simulation applications. This array processor has an integration module, fast temporary register files and fast arithmetic units that employ pipelining techniques. One major difficulty in using the AD-10 is the lack of a floating point unit, so that it is necessary to scale all variables to avoid overflow. More recently, however, Applied Dynamics International has introduced floating point capabilities. This new system is designated the AD-100.

Host-plus-peripheral systems are widely used in simulation applications that justify a price range of $100,000 to $500,000; yet they are very inexpensive compared to mainframes with the same capabilities.

1.3.3 Array Computers

Machines of this type(for example ILLIAC IV)have a single control unit and several synchronous processing units(SIMD architecture), performing the same operation on different data streams. In the ILLIAC IV, the processing units(PEs) form an array in which each processing unit has a direct data path to four neighboring units. The control unit broadcasts an instruction to all PEs

simultaneously, where each PE executes this instruction on a different data item. An array organization of this kind is very useful in computations involving the calculation of a function defined on a mesh or grid of points, where the value of the function at each point is influenced by the value of its neighbors. This is typical of systems modeled by partial differential equations. In fact the ILLIAC IV was originally designed to do simulations for such applications as meteorology, heat transfer, and fluid dynamics. While SIMD machines work well where data is structured in dense arrays, they are poor general-purpose machines.

1.3.4 Systolic Arrays

Systolic arrays are network of special-purpose cells designed to execute a particular algorithm as efficiently as possible. A systolic system consists of a set of interconnected cells, each capable of performing some simple operation[KUNG 82]. By tailoring the system to a specific problem, a very efficient communication structure can be achieved.

Systolic arrays can be used in matrix arithmetic, two-dimensional convolution and correlation, discrete Fourier transform, and any other problem where repetitive computations are performed on a large set of data.

1.3.5 Pipeline and Vector Computers

Machines of this kind use pipelines to achieve high execution speeds. A pipeline consists of a sequence of processing segments, through which a data stream

passes. Each segment performs partial processing on data and passes the results to the next segment. The final result is obtained after the data have passed through the last segment. When the pipeline is full, each segment is operating on different data, providing parallelism. Because each processing element works on a different step in the longer sequence of instructions, a pipeline is usually referred to as an MISD system.

Vector instructions apply a single pipelined operation to sets of vector operands. Computers such as the Cray-1, the Burroughs Scientific Processor(BSP), the CDC Cyber 205, and the Texas Instruments ASC combine array, pipeline, and specialized hardware techniques. These computers are usually referred to as "vector supercomputers" and are today's fastest computers. Supercomputers are used for number-crunching applications and are very suitable for the simulation of partial differential equations and are fast enough to perform real-time simulation of ordinary differential equations. They are, however, too expensive for performing dedicated scientific simulations.

1.3.6 Multiprocessors

Uniprocessor systems follow the famous von Neumann model of computation, in which, a single processing element executes instructions of a program one word at a time, and frequently modifies the contents of a main memory. Improvements in performance can be achieved by fashioning a system with several processing units, sharing the same address space, but running independently. Multiprocessors have been designed with processing elements ranging from

mainframe computers to microprocessors.

Examples of mainframe multiprocessors are the IBM 3033 MP, consisting of two IBM 3033, the IBM system/370 model 168 MP, and the Cray X-MP. These multiprocessor systems generally achieve better performance that their corresponding uniprocessors. For example, study shows that the performance of the IBM 3033 MP is between 1.2 to 1.8 times that of the uniprocessor[CONN 79]. Since the cost of mainframe computers is high, minicomputers and microprocessors have been used in multiprocessor systems. An example of a multi-minicomputer system is Carnegie-Mellon Cmmp, consisting of several PDP-11 minicomputers sharing the same address space.

The development of inexpensive microprocessors has made it possible to interconnect a large number of microprocessors to perform high-speed computation. An example of such a system is Carnegie-Mellon Cm* . This system, is a modular multi-microprocessor in which all processors have immediate access of all memory, although, the system works faster if most of the code and data references made by a processor are held locally to that processor. The processing unit, a Computer module or Cm, is a processor-memory pair. Computer modules are grouped to form a cluster. The system can be expanded to arbitrary size by interconnecting clusters via intercluster busses[SWAN 77, JONE 77].

Multiprocessors are MIMD machines. Scheduling of the tasks and minimizing the communication delays between processors are key issues in designing multiprocessors, as the progress of each processor depends on what other

processors have achieved.

1.3.7 Loosely Coupled Computer Networks

Loosely coupled networks consist of a number of traditional computers that are linked together. Examples of this type of networks are ARPANET[TANE 81] and ETHERNET[SHOC 82]. The ARPANET is a so-called long haul network which connects large computers in many universities and other organizations in the United States and Europe. This network is mainly used for passing electronic mail and information among different sites.

The ETHERNET is a local computer network architecture which can be used to interconnect small personal computers within an organization. This type of network permits the users to access different resources in the same facility and to exchange files and messages. While these networks are ideal for exchanging information; software and hardware incompatibilities and long communication delays between physically separate computers does not let them work together efficiently on the same problem.

1.3.8 Instruction Sequencing Methods

Instruction execution methods in digital computers can be classified into four broad categories — Control driven , data driven , demand driven, and pattern driven. Control-driven machines are driven by one or more sequential instruction streams. A program counter in each stream shows the address of the next instruction to be

executed. Program and data are stored in a global addressable memory, and program instructions frequently modify the contents of the memory. Data-driven or data flow computers, on the other hand, do not have a program counter and are driven by the availability of data. Any instruction whose data is available is ready for execution. Demand-driven or reduction machines execute an instruction only when the result it generates is needed by another already active instruction. Finally, pattern-driven systems execute their instructions when some enabling pattern or condition is matched.

Systems discussed so far were all designed based on the popular control-driven concept. In the following sections, we will discuss the other three possible methods.

1.3.9 Data Flow Machines

In pure data flow, as defined by Dennis[DENN 80] and many others, a program is shown by a graph in which all operations are functions without far reaching side effects. The only sequentiality is the partial order of the operations required by data dependencies. The computation is performed by sending tokens down the arcs. A token is a logical entity that contains a value and the address of the destination node. A node is ready for execution when all input arcs to it have a token. Since passing large structures from node to node is not efficient, large data structures such as arrays are implemented by tokens with references to those structures. Data flow machines have received a lot of attention in recent years, with a few experimental systems implemented so far.

1.3.10 Demand-Driven Machines

Demand-driven or reduction machines process the program graph in the opposite direction from the data-driven machines. In these machines a demand for a result activates a node which in turn will activate its arguments until constants are encountered which will then return a value to the demanding node. Functional languages as suggested by Backus[BACK 78] are suitable for this type of machine. A functional programming(FP) specifies computation as application of a combination of functions to a given object[ERCE 84a]. When a functional program is applied to an input object, it produces an output object. There are no variables, no states, and consequently no side effects. Mago's functional programming machine is an example of this type of machines which executes a purely functional programming language[MAGO 79a, MAGO 79b]. Reduction machines show great promise but have certain difficulties — If a machine does not have parallel computing resources large enough to store all the required data at once, the problem must be partitioned into computation blocks that fit into machine and then the partial results must be combined[HAYN 82]. Furthermore, an applicative programming language may not appeal to programmers without sufficient mathematical background.

1.3.11 Pattern-Driven Machines

Pattern-driven or associative processors such as STARAN[BATC 74] have been designed around the concept of an associative memory. The contents of an associative memory can be accessed when they match the string being sought.

Associative processors have arithmetic elements as well as data searching circuitry for each memory word or group of words[HAYN 82]. These computers are very efficient in two-dimensional image processing. They are, however, expensive and difficult to program.

1.3.12 Adaptable Architectures

Adaptable systems are capable of adjusting to computing requirements via software. One example of adaptable architectures is the microprogrammable computer, in which the interconnections between different devices are reconfigured by software. Another type of adaptable system, the "reconfigurable" system [VICK 80], is capable of reconfiguring the interconnection between different functional units. Yet another type of adaptable system, the "dynamic" system, redistributes resources among programs so as to increase hardware utilization [VICK 79]. Adaptable architectures are now under development to enhance the throughput of supersystems[KART 80]. These systems are very useful when the appropriate structure is not known, or the system must handle a wide variety of programs with unknown structure.

1.4 Simulation of Lumped Parameter Systems

Many physical systems, such as electrical systems and aerospace dynamic systems, can be modeled by a set of interacting elements in which the behavior of each element is specified completely in terms of the excitation-response relationship at its external terminals. These systems are usually referred to as lumped systems

and are defined by a set of ordinary differential equations.

A differential equation is an equation involving a function and its derivatives. Ordinary differential equations are differential equations that involve unknown functions of only one independent variable, where partial differential equations involve unknown functions of two or more independent variables. Ordinary differential equations, when expressed mathematically, take the form

$$f(x,\ y,\ y',\ ...,\ y^n) = 0$$

which specifies a relation between an independent variable, x, a dependent variable, y, and derivatives of this dependent variable.

In order to build a simulator and connect it to real-world hardware, these equations need to be integrated in real time[KARP 82]. Often, engineers rather than programming specialists are involved in the simulation project, and frequently they have to change the mathematical model and its computer implementation "on the spot". This requires a friendly and easy to use software interface[MAKO 83].

1.5 Continuous System Simulation Languages

Many simulation-oriented languages and software packages have been designed to implement simulation models on computers. These special-purpose languages, do not have the programming flexibility and portability of higher-level languages such as FORTRAN and C. Instead, they offer special functions, data structures, and other features that facilitate model definition and information

collection. Using high-level simulation-oriented languages alleviates the need for extensive programming and simplifies the use of the system. These languages are designed to model discrete, continuous, or combined systems.

Discrete system simulation languages are specially designed for simulating systems whose states change discretely at given points in time. Continuous system simulation languages(CSSLs), on the other hand, are designed to handle models described by a set of differential equations. Finally, combined continuous-discrete languages are well suited for simulation of systems which are not satisfactorily simulated by either type of languages.

GPSS [GORD 75], SIMULA [BIRT 73], and SIMSCRIPT [KIVI 75] are examples of powerful discrete simulation languages, and GASP [PRIT 74] is a combined continuous-discrete simulation language. A great deal of difficulty exists in design of languages for general class of partial differential equations. A few languages, such as PDEL [CARD 72] and LEANS [SCHI 73], have been designed. On the other hand, a variety of powerful CSSLs for solution of problems modeled by ordinary differential equations exist[KARP 74]. We are specially interested in the later group of languages, and we will discuss them in more detail.

1.5.1 Block Form CSSLs

Two type of CSSLs have been developed, block form and expression based. Block form languages work as analog or hybrid computer emulators and allow the model to be programmed in essentially the same way as it is solved on an analog

computer. A model is defined as a block diagram and is implemented by a set of macro instructions that define blocks the same way that they may have been connected on an analog computer patch board. Today by decreasing popularity of analog simulation, languages of this type such as, DAS [GASK 63] and MIDAS [HARN 64], are not widely used any more.

1.5.2 Expression Based CSSLs

In expression based languages, the problems are programmed from the differential and algebraic equations which express the model, rather than breaking them into functional elements. Dynamic Models(DYNAMO), Continuous System Modeling Program, Version III(CSMP-III), Continuous System Simulation Language, Version IV(CSSL-IV), and Advance Continuous Simulation Language(ACSL) are some of these languages that will be discussed here.

The DYNAMO language[PUGH 70] uses first-order difference equations to model continuous systems. This language is suitable for system dynamic studies were the rate of change of system components are studied to determine their influence on the stability or growth of the system. The results are usually used to suggest reorganization or produce early warnings from an undesired direction. In DYNAMO, the model is defined in terms of varying rates of flow and the corresponding changes in level of the state variables.

The CSMP-III is a nonprocedural language with many simulation functions and macros. It has several integration algorithms and very good diagnostics and

debugging facilities. A CSMP-III program has three types of statements: structural statements, which define the model; data statements, which assign initial values to variables and define constants; and control statements, which are commands to exercise the model.

CSSL-IV and ACSL are very similar. Similar to CSMP-III, they are nonprocedural languages designed in accordance with the CSSL standard[STRA 67] developed by the Society for Computer Simulation. A model can be defined in either an explicit or an implicit mode. An explicitly structured program in divided into three regions: INITIAL, DYNAMIC, and TERMINAL regions which contain equations for initialization, execution of dynamics, and post-processing respectively. These languages provide a wide variety of simulation operators which facilitates the simulation effort and have enjoyed wide use over a number of years.

1.6 Objectives

Among all the systems mentioned in the previous sections, the network of microprocessors offers the potentials of tremendous execution power with very low cost. Other systems, such as supercomputers, array computers, and peripheral array processors can achieve the desired speeds. However, they all involve higher costs that can not be justified for many dedicated scientific applications. Furthermore, a network of microprocessors has several advantages over the other systems, including the ability to trade off the number of processors against the time required for simulation and the graceful degradation of the network by retiring failed processors. Utilization of off-the-shelf components for the network eliminates the

cost of designing special purpose hardware.

In order to take advantage of the parallelism offered by a network of microprocessors, programs must be reorganized at the level of the algorithm to detect any concurrency that enables one to take advantage of parallel architecture. This means that the programmer must have support tools and techniques to do this. Design of a proper software interface that facilitates the use of a microprocessor network is of great importance. Programmers insist on high level, nonprocedural languages. Making this available to the microprocessor network requires a software interface between the user and the underlying parallel hardware. This is the vital link in having effective network simulators.

One objective of the present research is to design a software interface for a network of microprocessors fashioned to simulate systems modeled by ordinary differential equations.

Most physical systems modeled by ordinary differential equations are comprised of loosely coupled components. The simulation of these systems naturally decomposes into a set of concurrent processes, with each process simulating a part of the effort completed at each step to find the solution for the next step. As long as each process needs to interact with others only at the end of each cycle, the processes can run in parallel on a multiprocessor system.

In order to maximize network utilization, each processor must have a balanced load with minimum interaction with the others. This means that the parallelism in the program must be revealed. The traditional von Neumann concept

of programming is not suitable for a network of processors.

One promising approach is the data flow method where the execution is driven by the need for data values that are produced and consumed. Implementation of a pure data flow machine, however, requires the design of special purpose hardware. In this project, with the intention of using off-the-shelf microprocessors, the data flow concept is used only as a parallel sequencing tool and is mixed with other techniques to get the best of both worlds — Utilizing the concepts of data flow such as single assignment rule, locality of effect and so on to reveal parallelism; and implementing them on a network of off-the-shelf von Neumann microprocessors to lower the cost.

1.6.1 Design Considerations

The goal of the design effort was to provide A Language Interface (abbreviated to ALI) that is simple to use but fast enough to provide the response time required for the real-time simulation of complex systems. The system was designed and analyzed with respect to the following factors:

(1) Fast response time — The system shall be fast enough to perform all real-time calculations within a given deadline.

(2) Ease of programming — An existing and easy-to-learn programming language shall be used as the source language, so as to enable a user unskilled in programming to learn the programming language and to implement the simulation model in a short time.

(3) Standard hardware — The system shall be made with off-the-shelf hardware modules with no need to design special-purpose processors or control units.

(4) Cost-effectiveness — The system shall be in a lower price range than a mainframe computer of comparable speed.

(5) Modularity and expandability — The software system shall be capable of operating with different numbers of processors, and be general enough to be used on different types of microprocessors.

1.7 System Overview

Figure 1.2 shows the block diagram of ALI. Either CSSL-IV or ACSL can be used as the high-level language. The model is expressed in one of these two languages and is translated into a data flow graph through several intermediate steps. Although CSSL-IV and ACSL compilers are commercially available, they generates FORTRAN code which is unsuitable for that purpose. Instead, these compilers are used only for detecting syntax and semantics errors.

The user is also urged to run the program on a sequential machine and debug the run-time errors and user mistakes before using the data flow system. Run-time errors are errors, such as divide by zero and arithmetic overflow or underflow, that can only be detected during run time. User mistakes are typographical errors and design mistakes that are syntactically and semantically correct but do not generate the desired results. Finding these errors is much easier on sequential machines and

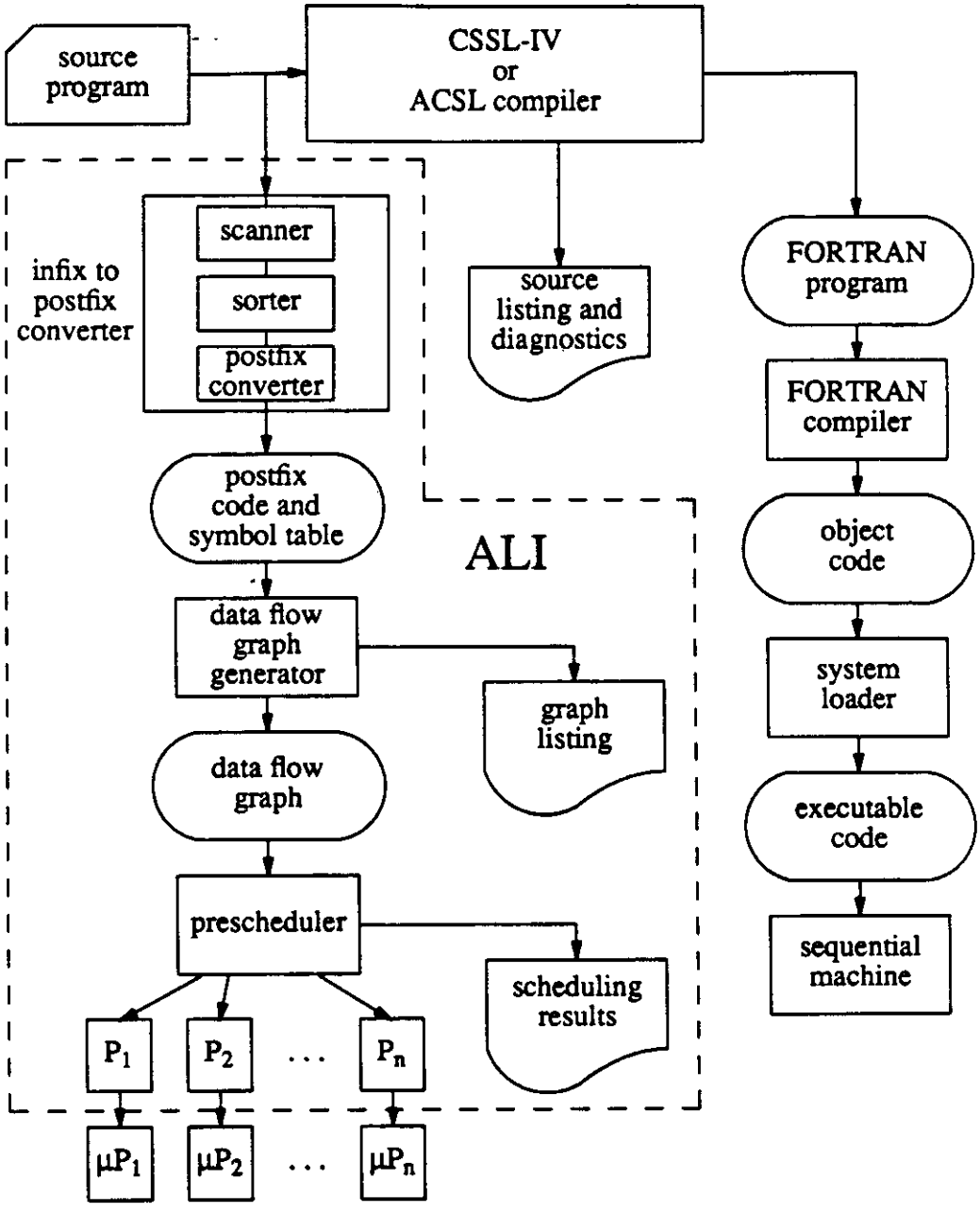


Figure 1.2 System block diagram

the user can take advantage of the debugging facilities of the source language. When all errors are detected, the error free CSSL source program is used to generate the data flow graph. The data flow graph is then analyzed and allocated among the processors.

Instead of breaking the graph into single operations and allocating them to processors, groups of related serial operations are taken together to form packets of executable code. Each packet needs to interact with the others only at the beginning and at the end of its execution to exchange the results. Execution of each packet is assigned to one processor that performs it sequentially. An allocation heuristic is developed to analyze a data flow graph and to divide it into loosely coupled executable packets.

Figure 1.3 shows the different software modules of the system. Each module accepts one or more input files, processes them, and generates one or more output files. The functions of each of these modules are described in this dissertation. The source codes and a user manual are included in the appendices.

The software system is suitable for a hardware system which is being designed in an ongoing project at UCLA[ERCE 84a]. The hardware is organized into several clusters, each containing one or more microprocessors, memory modules, and interface modules[ERCE 84b]. Figure 1.4 shows this configuration. Intercluster communication is done through a global broadcast multi-bus with one unidirectional bus dedicated to each cluster[ERCE 84c]. Each result token carries a unique source tag. Clusters accept or ignore a data token based on a local filter table,

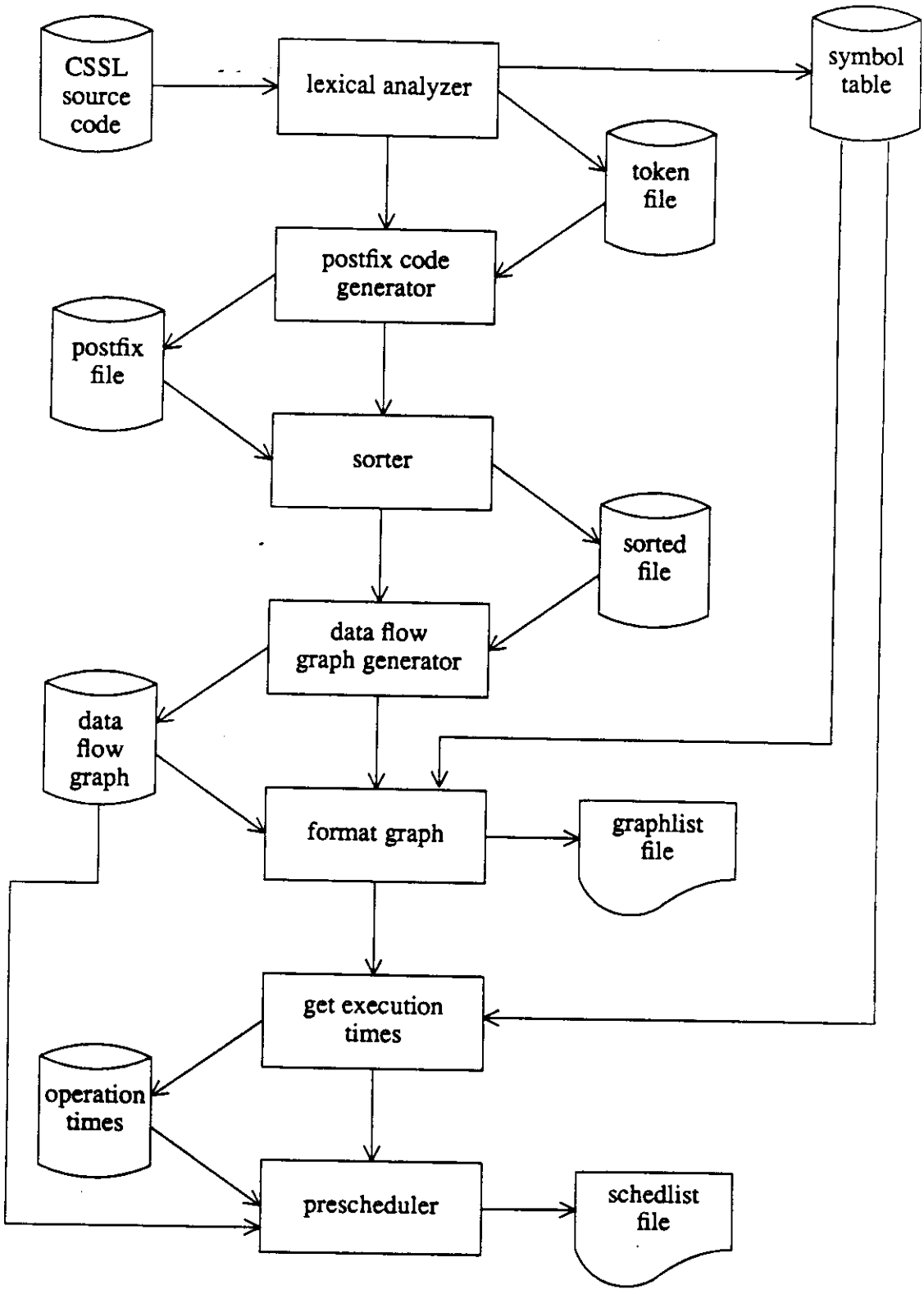
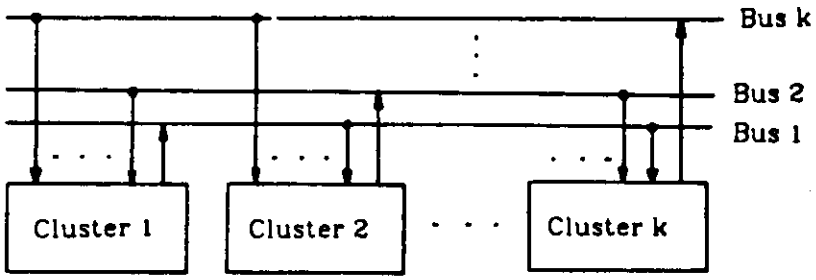
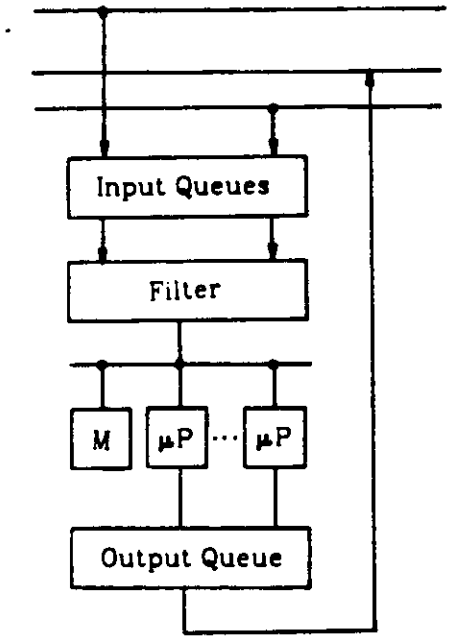


Figure 1.3 Software block diagram

generated at compile time. An accepted token is transferred to a predetermined region of local memory. When all arguments of a task are available, the local scheduler invokes the task.



(a) Data flow multi-microprocessor organization



(b) Cluster organization

Figure 1.4 Hardware block diagram

1.8 Structure of the Dissertation

This dissertation is organized into nine chapters and two appendices:

Chapter 2 dealing with ordinary differential equations and high-level languages emphasizes the role of a suitable source language in facilitating programming and detection of parallelism. It discusses the class of Continuous System Simulation Languages(CSSL), especially the languages CSSL-IV and ACSL that were selected as the source languages for ALI.

Chapter 3 explains the reasons that the FORTRAN intermediate code and the object code generated by a commercial CSSL compiler is not easy to use on a network of microprocessors. It points out the need to design a translator to convert a CSSL source program into a data flow graph without generating the FORTRAN intermediate code. It presents the lexical analyzer and the postfix code generator of this translator. It also describes the sort algorithm that reorders the nonprocedural part of a CSSL program, such that each variable receives a value before it is used in any statement.

Chapter 4 presents the concept of data flow. It explains why in this project, instead of using data flow in its pure sense, it is utilized only as a parallel sequencing tool. The method of representing a data flow graph is also described. The chapter concludes with a discussion of the algorithm that generates the data flow graph.

Chapter 5 on scheduling is concerned with the strategy to be used to allocate executable tasks to each processor and the heuristics in defining it. This strategy must distribute a balanced load to each processor and at the same time must minimize interprocessor communications. Chapter 5 also shows that the original data flow graph can be transformed into another graph, whose number of active nodes at any point of time does not exceed the maximum number of available processors. It concludes by presenting a method of allocating this new graph among the processors so as to minimize communication delays.

Chapter 6 describes methods of performing integration, table look up, and function generation. The proper performance of these operations greatly contributes to attaining high speed.

Chapter 7 is devoted to the benchmark — the model of a helicopter jet engine provided by NASA. It also describes the operations of jet engines in general.

Chapter 8 analyzes the results of the benchmark and compares ALIs performance with different numbers of processors.

Chapter 9 summarizes the results of the research.

Appendix A is the self-contained user manual providing all necessary information for using the system.

Appendix B contains all source programs, which are written in PASCAL and are fully commentized for future modifications and enhancements.

CHAPTER 2

DIFFERENTIAL EQUATIONS AND HIGH-LEVEL LANGUAGES

2.1 Introduction

The motivation of this research was to provide designers with a software tool, so as to enable them to simulate systems of ordinary differential equations on a network of microprocessors. The choice of a programming language can have a major impact on the effectiveness of constructing programs which are reliable and reasonably easy to understand, modify, and maintain. A model can be easy or difficult to program in a given language, depending on how many of the language constructs match the structures needed to build the model. In addition, a programming language influences the way that its users think about programming.

The primary purpose of a programming language is to provide the programmer with the necessary tools to develop reliable and cost effective programs. Furthermore, a programming language suitable for a multiprocessor system being used by nonprogrammers must have, among others, two important properties: On one hand, it must be user friendly, and on the other hand, it must be able to preserve the natural parallelism of the model, so that it can easily be translated into machine code and allocated to individual processors.

In this chapter, the source languages selected for the system are described. First, ordinary differential equations and the criteria for selecting a suitable language for solving them are discussed. Then the class of Continuous System Simulation Languages(CSSL) is introduced, followed by a discussion of the CSSL-IV and ACSL languages, which are selected as the high-level languages of the system.

2.2 Ordinary Differential Equations in Simulatin

System simulation is a useful tool for designing and studying the behavior of physical systems. Many properties of these systems are formulated quantitatively by mathematical relations involving certain functions of space and time and possibly the derivatives of these functions.

There are at least two aspects to study of physical systems. One of them is to describe the system in terms of mathematical equations, the other is to find the solution to those equations. A system is first divided into several subsystems. A mathematical model is then derived by defining inputs and outputs for each subsystem and some, but presumably not all, relations between various subsystems. In order to study the important features of the system, the model must be simplified, as much as possible, by eliminating unimportant features. This must be done carefully by considering experimental data and past experience with similar problems.

Mathematical models can be classified into three broad categories, depending on the nature of the problem, the amount of insight into the system, and the

attributes of the system that are being studied: Discrete event, lumped parameter, and distributed parameter models. These models are defined by algebraic, ordinary differential, and partial differential equations respectively(Figure 2.1).

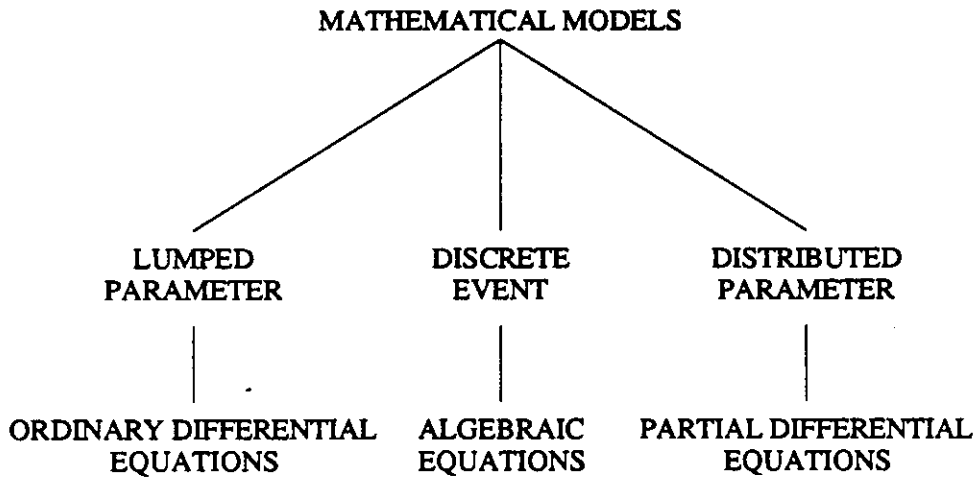


Figure 2.1 Different mathematical models for physical systems

An ordinary differential equation is an equation that involves a single unknown function of a single variable and some of its derivatives. The simulation of systems modeled by ordinary differential equations is the topic studyed in this research. The choice of a suitable language plays an important role in designing powerful simulators. A simulation-oriented language can aid the users in describing the model and provide useful simulation operators such as integration, hysteresis, and lead-lag to perform various simulation operations.

2.3 Choice of a Suitable Language

The main considerations used in selecting a language for ALI were:

(1) Simplicity and power — The language shall be relatively simple and easy to use. More complex features must be available to the expert, but not a matter of concern for the novice. Thus, a language is desirable that will permit the novice to learn a minimum subset and then advance, if he wishes, to the use of a complex and powerful simulation language.

(2) Transparency — The underlying characteristics of the system shall be hidden from the users so that they do not have to know about data flow or multiprocessor systems in order to write their programs.

(3) User friendliness — The language shall provide meaningful error messages and debugging facilities, and shall enable the user to work with familiar terms and concepts.

(4) Simulation Features — Basic simulation tools, such as integration, limited integration, backlash, etc., must be provided for the user. A library of subroutines to perform trigonometric and logical operations is very desirable.

(5) Data flow convertibility — The language must be easy to convert to a graph form suitable for data flow and multiprocessor systems, thereby avoiding a complicated compiler design.

Several alternatives can be considered for selecting a programming language. The first option is to use the assembly language of the microprocessors. This option has all the advantages and disadvantages of writing code in assembly language. Writing in assembly language eliminates the need for a compiler, and the code will run faster. But on the other hand, it defies our goal of having a user friendly and easy to use system. Programmers prefer high-level languages and nonprogrammers will probably switch their system rather than learning to write in assembly language. Portability and transparency will be lost and the code will be difficult and expensive to develop and maintain.

The second option is to use a data flow language. This has the advantage of helping the user to write concurrent code. But the users usually do not want to learn a new programming language. The concept of a data flow programming language may be especially difficult for nonprogrammer users. Furthermore, so far there has been no data flow language with sufficient simulation tools.

The third option is to choose an existing language and to write a compiler to generate a data flow graph for it. This provides the maximum transparency for the end users and puts the burden of dealing with parallel processing totally on the shoulder of the software system designer. The selection of a suitable language from a wide variety of available languages is very important and must be addressed carefully.

General purpose languages such as FORTRAN, PASCAL, or other procedural languages are not designed to meet the specific requirements of

simulation problems. Physical systems to be modeled are usually composed of several loosely coupled subsystems running in parallel. Writing the model in a procedural language does not take into account the parallelism that is inherent in the system. The user, when expressing the model, has to write the statements in exactly the order that they are to be executed. Parallel equations cannot be distinguished and are written sequentially. A complicated optimizing complier is therefore needed to extract the parallelism. Furthermore, general purpose languages do not provide useful operators, such as the integration operator, as a part of the language. Such operators must be defined by the user as functions and put in a user library.

Special purpose simulation languages provide a variety of facilities to help simulation programmers. A family of languages called the Continuous System Simulation Languages(CSSL) are specially designed for the simulation of systems that are described by systems of ordinary differential equations. Continuous System Simulation Language - version four(CSSL-IV), Advanced Continuous Simulation Language(ACSL), and Continuous System Modeling Program - version three(CSMP-III) are some of these languages. The design of all of these languages is based upon recommendations of Simulation Council's Committee on Continuous System Simulation Language published in SIMULATION[STRA 67]. Either CSSL-IV or ACSL, which are the most popular ones, can be used in ALI and we will use the term "CSSL language" to refer to both languages.

2.4 CSSL-IV and ACSL Languages

The syntax and semantics of ACSL[MITC 75] and CSSL-IV[NILS 76, NILS 84] are virtually identical and correspond very closely to the requirements of dynamic systems simulation. Both languages provide the user with a variety of simulation operators and functions. Of particular significance is the fact that these languages are nonprocedural, so that the model can be expressed by a series of equations, not necessarily in the same order that they are to be executed. These statements are sorted by the language processor.

2.5 Structure of a CSSL Program

A CSSL program has two distinct parts:

Part I : Model Definition
This part contains those statements concerned with defining the model or the structure of the system being simulated.

Part II : Run-Time commands
This part contains the sequence of commands that exercise the model, i.e. change parameters, start runs, control the output, etc.

The model definition part of a program can be written in either an explicit or an implicit mode. An explicitly structured program is divided into three regions : INITIAL, DYNAMIC, and TERMINAL regions(Figure 2.2). Each one of these regions corresponds to a separate phase of the simulation. The INITIAL region contains the equations required prior to the execution of the dynamics, the

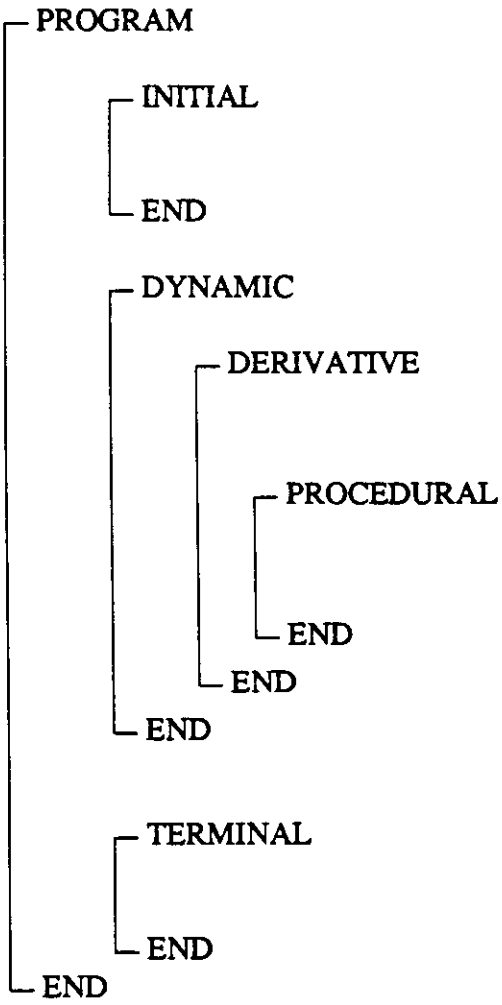


Figure 2.2 Structure of a CSSL program

DYNAMIC region contains the dynamics of the system and the TERMINAL region contains equations that are exercised to do post-processing on the solution.

In the INITIAL region, arrays and constants are defined, and initial values are assigned to variables. In the DYNAMIC region the model is exercised. In the TERMINAL region, calculations needed after the end of the simulation are performed. The DYNAMIC region contains one or more DERIVATIVE sections. Each one of these sections contains a set of differential equations. The

DERIVATIVE section is nonprocedural. If procedural code such as repetition loops or conditional statements are required, they are written in a PROCEDURAL block in which the statements are executed in the same sequence that they are written. An example of a CSSL-IV program that simulates the ejection of a pilot seat from a jet fighter is shown in Figure 2.3.

Note that SWIN function acts as a switch. If the first argument is greater than zero, the output will be equal to the second argument. Otherwise the output will be equal to the third argument. In fact,

YGE 1 = SWIN(Y1 - Y, 0.0, 1.0)

is equal to:

```
IF (Y1 - Y > 0) THEN
                  YGE1 = 0.0
                ELSE
                  YGE1 = 1.0
```

The run-time commands part of a CSSL program is a set of commands that exercise the model. Some of these commands specify, for example, the name of variables whose values must be stored, or printed, or plotted. Other commands signal the executive to start and stop the execution. These commands are, in fact, the way a CSSL programmer communicates with the executive. In the example in Figure 2.3, the run-time commands specify the title of the program to be pilot ejection and request that both a tabular output and a plot to be printed.

```
program pilot ejection
 initial
  constant thedeg = 15.0, degrad = 57.3, ...
    mass = 7.0, cd = 1.0, s = 10.0, y1 = 4.0, ...
    g = 32.2, ve = 40.0, r0 = 0.0023769, ...
    va = 900., xmn = -60.0, ymx = 30.0, ...
    tmx = 40.0
  cinterval cint = 0.01
  the = thedeg/degrad
  comment  seat initial velocity
  vx = va - ve * sin(the)
  vy = ve * cos(the)
  vic = sqrt(vx ** 2 + vy ** 2)
  thic = atan2(vy, vx)
 end initial
 dynamic
  derivative eject
    comment  relative positions
    x = integ(v * cos(th) - va, 0.0)
    y = integ(v * sin(th), 0.0)
    comment  space velocity and flight path angle
    v = integ(yge1 * (-d / mass - g * sin(th)), vic)
    th = integ(yge1 * (-g * cos(th) / v), thic)
    comment  compute drag
    d = 0.5 * r0 * cd * s * v ** 2
    yge1 = swin(y1 - y, 0.0, 1.0)
  end derivative
  termt(x .le. xmn .or. y .ge. ymx .or. t .ge. tmx)
 end dynamic
 terminal
 end terminal
end program
comment run-time commands
 hdr pilot ejection
 prepar t,th,v,x,y
 start
  print t,th,v,x,y
  plot t, th, v, x, y
 stop
```

Figure 2.3 Pilot ejection program in CSSL-IV

2.6 Implementation

CSSL languages act as an adjunct to an established procedural language. The commercially available CSSL compilers translate the source code into FORTRAN, which then, together with PROCEDURAL blocks and FORTRAN subroutines, is converted into the machine code of a sequential computer. As mentioned before, for a multiprocessor system, the FORTRAN language is not desirable, because it is difficult to divide the code into parallel blocks. Therefore, in this project, the CSSL and the FORTRAN compilers are only used to detect syntax and semantics errors. Once it is ascertained that there are no errors, the object code is disregarded, and the now error free source code is used to generate code for the microprocessor system, using the lexical analyzer, the postfix code generator, etc. which are part of the system.

The real-time simulation starts when the code in the DYNAMIC region is first executed and ends when the DYNAMIC region is terminated. Since the INITIAL and the TERMINAL regions are executed before and after the DYNAMIC region respectively, they have no effect on the speed of the real-time simulation. Therefore, all the effort is made to speed up the execution of the DERIVATIVE section where the state variables are calculated for each time level.

Each DERIVATIVE section is a parallel process. It has its own unique name, its own unique independent variable, and its own unique set of integration control parameters. Statements in the DERIVATIVE section obey the "single assignment" rule. Each variable can appear at the left hand side of a statement only

once. There are no far-reaching data dependencies in a CSSL program and, as is discussed in the following chapters , a CSSL program can therefore be efficiently translated into a data flow graph.

CHAPTER 3

INTERMEDIATE CODE GENERATION

3.1 Introduction

Once a suitable high-level language is selected, the next step is to write a translator to convert the source programs into machine executable code. As mentioned in the previous chapter, the commercially available CSSL and FORTRAN compilers are only used to detect syntax and semantics errors. When all the errors are corrected, the object code is disregarded, and the error free source code is used to generate the data flow graph.

Before the CSSL source code is translated into a data flow graph, it is converted into intermediate code in several steps. First, the source code is converted into tokens which are easier to store and to analyze. Next, it is converted from infix notation to postfix notation, which makes the code more suitable for processing by computer algorithms. Then the nonprocedural part of the code, namely the statements of the DERIVATIVE section, are sorted. After that the postfix code is translated into a data flow graph to reveal the parallelism. Finally, this data flow graph is analyzed and divided into executable packets which are allocated to processors. In this chapter the lexical analyzer, the postfix code generator, and the sorter are discussed.

3.2 Lexical Analyzer

The lexical analyzer converts the long strings of characters into simpler forms. Each string of characters in the source program that identify a single entity is converted into a single token. Each token can be either one or two words. Keywords and operators are shown by single word tokens. Variables, constants, and array references are shown by double word tokens. The first word shows the entity type, and the second one shows the entity's relative position in the symbol table.

Functions are also shown by double word tokens. The first word shows the function's unique identification, and the second word shows the number of arguments(This number is found by the postfix code generator by counting the number of operands between the two matching parentheses following the function name). This makes it possible to implement easily functions, such as MAX and MIN, that have a variable number of arguments. The structure of the symbol table is shown in Figure 3.1. As example, a part of a symbol table is shown in Figure 3.2.

3.2.1 Symbol Table

Each entity in the symbol table has several words. The first word is the total length of the entry. The second word is the form of the entity(constant, simple variable, array, etc.). The third word is the type(integer, real, etc.). The fourth word is the length of the string of characters that denotes the entity, and the fifth word is a pointer to the start of the string. Array and table entries have four more words, a word to show the number of dimensions(up to three dimensions are allowed), and

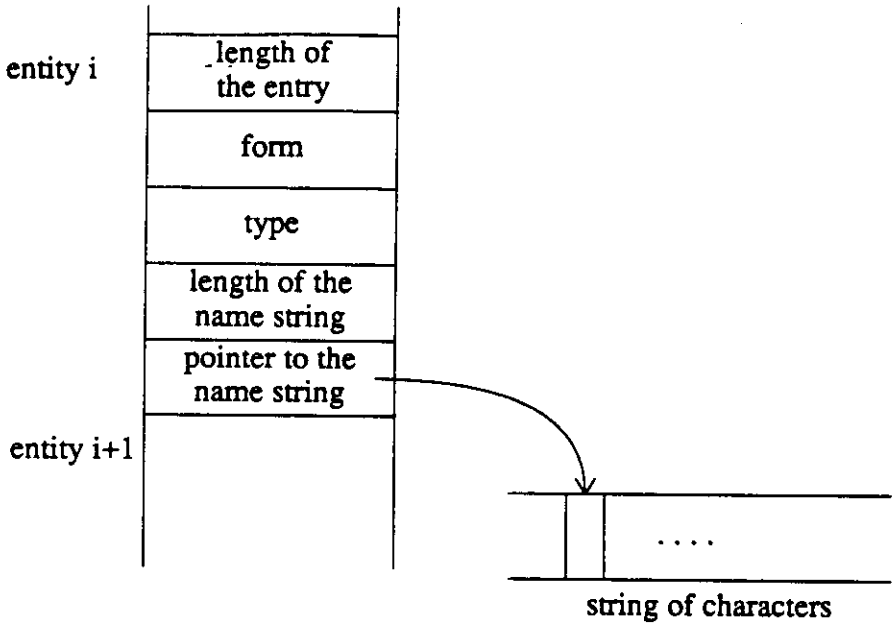


Figure 3.1 Structure of symbol table

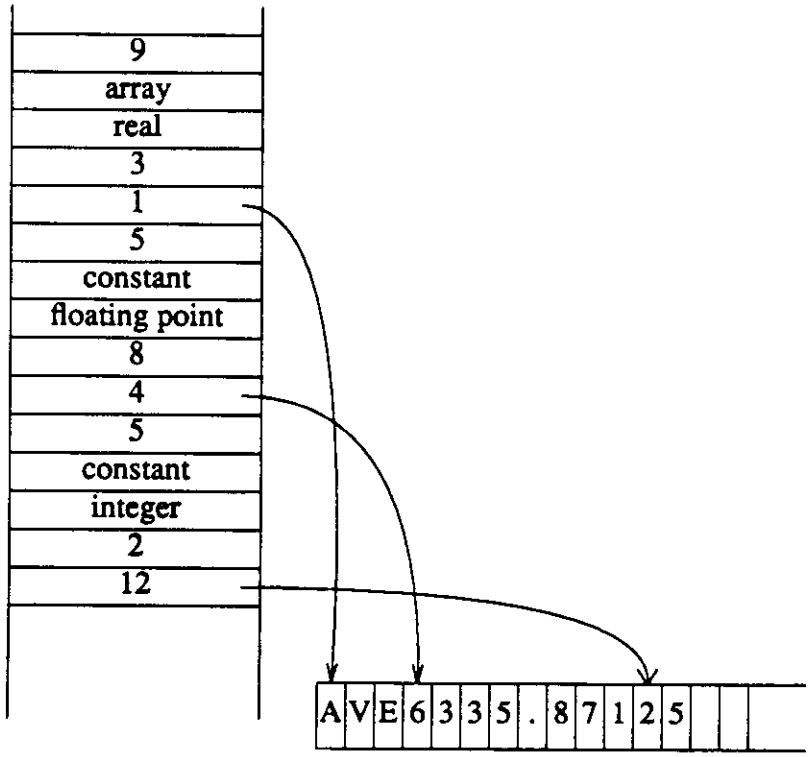


Figure 3.2 An example of a symbol table

three words to show the size of each dimension.

The string of characters that denotes a name or a constant is not included in the symbol table. A separate data structure is used to store those strings. Only the length and a pointer to the start of the string is embedded in the symbol table. This saves space(since each name may have different number of characters) and makes easier to scan the symbol table.

3.2.2 Reserved Word Table

Reserved words are stored in a separate table called the "reserved word table"(Figure 3.3). A nonzero value in the second entry in the token table means that a token is not generated upon scanning that reserved word. For example, the "COMMENT" directive is a reserved word which is recognized by the system, but no token is generated for it.

In order to speed up the scanner, an index method based on the length of identifiers is used to search the reserved word table. The names of identifiers are compared with only those entries in the reserved table that have the same length.

3.2.3 Operator Table

Operators are stored in another table called the "operator table"(Figure 3.4). All arithmetic, logical, and relational operators used in FORTRAN and CSSL are supported.

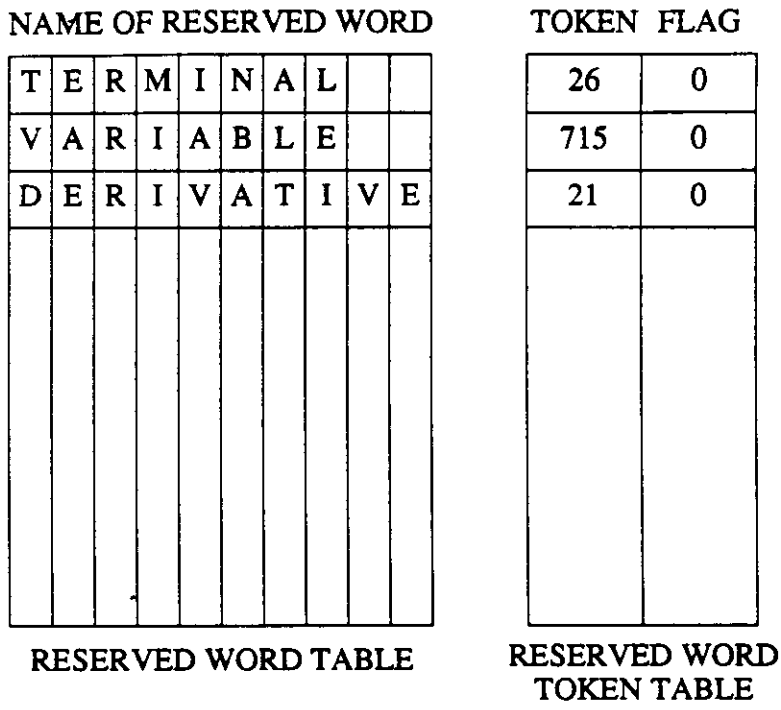


Figure 3.3 Tables for reserved words

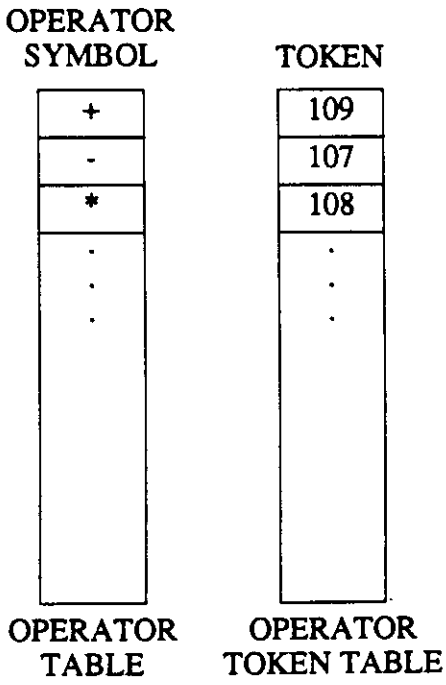


Figure 3.4 Tables for operators

3.2.4 Scanning Identifiers

The lexical scanner reads characters of the source code one by one and builds up strings of characters that make a single entity. For example, the string of characters "DERIVATIVE" is one entity. The first character is an alphabetic character so the lexical analyzer starts building an identifier. It stores the characters in a temporary buffer and keeps on reading new characters. As long as new characters are either alphabetic or numeric, they are added to the buffer, and a pointer that marks the end of the buffer is incremented. This continues until a nonalphanumeric character is read. The lexical analyzer assumes that the end of the identifier string is reached and does not add the new character to the buffer. Instead, it searches the reserved word table to see if the identifier is a reserved word. If it is found there, the corresponding token is written in the output file.

If the identifier is not a reserved word, the symbol table is searched to see if it was previously defined. When searching the symbol table, as in the case of the reserved word table, only those entities that have the same string length are considered to match the scanned string. If the lengths are the same, the two strings are compared to find a match. If a match is found, a two word token is generated. The first one shows the form, and the second one shows the relative position of the identifier in the symbol table. If the identifier is not found, it is entered in the symbol table, the character string is added to the end of the "character string table", and the pointers are adjusted accordingly.

3.2.5 Scanning Operators

If the string starts with a numerical character, i.e., it is a constant such as 6335.871, the lexical analyzer only searches the symbol table to find it. For numerical values a two word token is generated. The first word shows the form, and the second word shows the relative position of the constant in the symbol table.

If the string starts with an operator, such as + , * , etc., the operator table is searched to find the proper token. A string that does not start with an alphanumeric character and is not found in the operator table is flagged as unrecognizable, and a diagnostic message is generated.

3.2.6 Generating the Token File

When the lexical analyzer generates a token, it enters it in the output file and starts examining the next character from the source file. In this way it builds the next lexical entity. This continues until all characters in the source file have been examined. At that point the source code is completely converted to tokens and is passed to the postfix code generator to be translated to postfix notation. Figure 3.5 shows the calling sequence of the lexical analyzer modules. The symbol table is saved on the disk to be used by subsequent software modules(Figure 1.3).

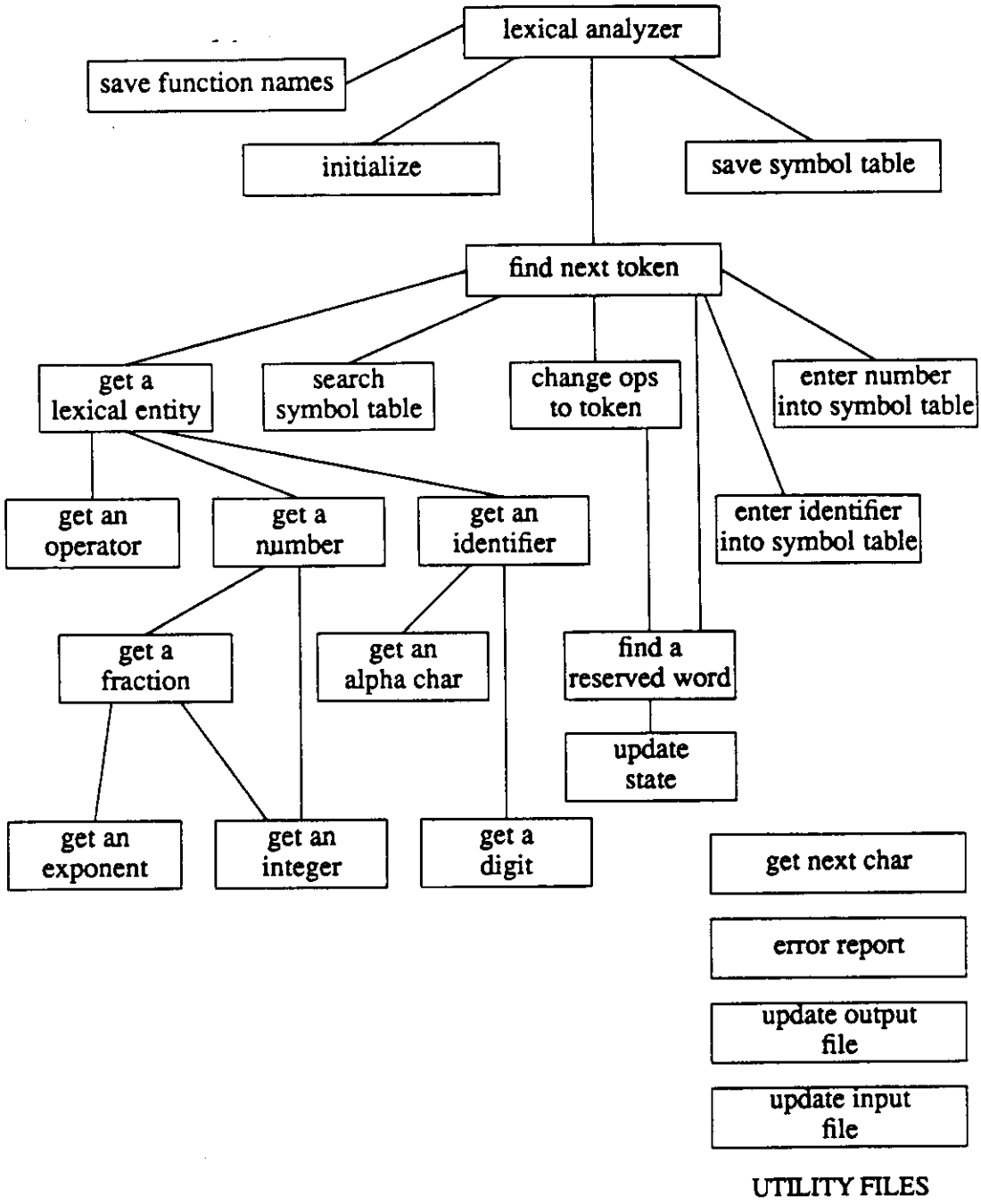


Figure 3.5 Lexical analyzer calling sequence

3.3 Postfix Code Generator

The source code is written in infix notation, in which binary operators are inserted between their two operands. This notation is well suited for humans. However, computer algorithms are performed more easily if the code is converted to postfix notation in which an operator always immediately follows its operands, and the operands are written in the sequence in which they are to be executed. For example, in the expression A = B + C * D if multiplication has precedence over addition(as in FORTRAN), it should be executed as A = B + (C * D). On the other hand, if addition and multiplication have the same precedence and the statement is scanned from left to right, it will be evaluated as A = (B + C) * D. In postfix notation the first expression is written as ABCD*+=, and the second one is written as ABC+D*= . In either case there is no confusion, each operator always follows its immediate operands, and there is no need for parentheses.

The postfix generator uses a stack to store operators, functions and, array references. Operators are assigned relative precedences that show which operation is performed first in any expression. The precedences of the operators are given in Figure 3.6. As can be seen, the arithmetic operators have a higher priority than the relational operators which in turn have a higher priority than the logical operators.

	operator
highest priority	- (unary minus)
	** (power)
	* /
	+ -
	.GT. .GE. .LT. .LE. .EQ. .NE.
	.NOT.
	.AND.
lowest priority	.OR.

Figure 3.6 Precedence of the operators

The source code, that has been converted to tokens, is scanned by the postfix generator. Operands and reserved words are written into the output file. Whenever a function or an array reference is scanned, it is pushed onto the stack and is popped whenever all its arguments have been scanned. The number of arguments that each function has is found by counting the number of operands between the two matching parentheses immediately following the function name. This number is written into the output file after the function token.

Whenever an operator is scanned, its precedence is compared with the precedence of the operator at top of the stack. If the operator at top of the stack has a higher precedence, meaning that it is to be executed before the new operator, the operator at the top of the stack is popped and is written into the output file. The precedence of the scanned operator is then compared with the precedence of the new

operator at the top of the stack, and so on. If the stack is empty or the precedence of the operator at the top is less than the precedence of the scanned operator, the scanned operator is pushed onto the stack.

Parentheses have a special meaning in arithmetic. Any expression that is inside the parentheses is performed first. Therefore, a left parenthesis is always pushed onto the stack. When a right parenthesis is scanned, the stack is popped until the matching left parenthesis is reached. This assures that the operations inside the parentheses are performed together. There is no need to copy left and right parentheses to the output file.

When the end of the statement is reached, all remaining operators in the stack are popped into the output file. This continues until all statements in the input file have been translated to postfix notation. The postfix file is then passed to the sorter to sort the statements of the DERIVATIVE sections.

3.4 Sorter

CSSL languages are nonprocedural, meaning that the user can express the model of the system without being concerned about the sequence which the program statements are executed. The statements of the DERIVATIVE section can therefore be written in arbitrary order. The sorter sorts these statements so that the use of each variable is preceded by its definition.

3.5 Sort Algorithm

The sort algorithm [MITC 75] is processed in two passes. In the first pass all variables on the left hand side of the assignment are marked as undefined. The state variables are excluded, because for each time level the value of the state variable at the past time level is used to do the calculations. Therefore, the state variables are known at the beginning of each time level and are not marked as undefined. Pass one also generates a list of input variables for all statements(a list of the variables used on the right hand side of assignment statements).

Pass two examines the statements one by one. If any of the variables in the input list is marked as undefined, the statement is saved, and the next statement is examined. If none of the variables in the input list is marked as undefined, it is inferred that all of them have been previously defined. The output variable(the variable on the left of assignment) is marked as defined, and the entire statement is written into the output file. Whenever one variable is marked as defined, all statements that were saved previously are reexamined to see if any of them now have all their input variables defined. By continuing this procedure, all statements are eventually sorted, and any algebraic loop of the type

```
A = B
B = C
C = A
```

is detected. When an algebraic loop is detected, it is flagged as unsortable and a diagnostic message is generated.

The following functions involving memory operators(operators which require past history) can break algebraic loops:

BCKLSH	(backlash or hysteresis) (ASCL only)
CMPXPL	(complex pole)
DELAY	(delay)
DERIVT	(numerical differentiation)
HSTRSS	(backlash or hysteresis) (CSSL-IV only)
IMPL	(implicit equation solver)
INTEG	(integration)
LEDLAG	(lead-lag transfer function)
LIMINT	(double limited integration)
LOGIC	(flip flop) (CSSL-IV only)
MODINT	(moded integrator)
REALPL	(first order lag)
ZHOLD	(zero hold)

Variables using these functions on the right hand side of their assignment statements are assumed to have known values at the beginning of each time level. Consequently they are marked as being defined by the sort algorithm.

3.6 Example

The familiar pilot ejection program[STRA 67] is used as an example. Figure 3.7 shows the variables of the DERIVATIVE section after pass one of the sorter is completed.

SYMBOL	UNDEFINED FLAG	LIST OF INPUT VARIABLES					
X	OFF	V	TH	VA			
Y	OFF	V	TH				
V	OFF	YGE1	D	MASS	G	TH	VIC
TH	OFF	YGE1	G	TH	V	THIC	
D	ON	R0	CD	S	V		
YGE1	ON	Y1	Y				

Figure 3.7 Sort table at the end of pass one

Note that all variables having the INTEG function on the right hand side of their assignment statements are marked as defined. Figure 3.8 shows the statements of the DERIVATIVE section after pass two is completed.

```
X = INTEG(V * COS(TH) - VA, 0.0)
Y = INTEG(V * SIN(TH), 0.0)
D = 0.5 * R0 * CD * S * V ** 2
YGE1 = SWIN(Y1 - Y, 0.0, 1.0)
V = INTEG(YGE1 * (-D / MASS - G * SIN(TH), VIC)
TH = INTEG(YGE1 * (-G * COS(TH) / V), THIC)
```

Figure 3.8 The DERIVATIVE section after sort is finished

Note that the statements are internally converted to postfix notation before the sorting begins. Here, they are shown in the infix notation for clarity. After sorting is completed, the output file is passed to the data flow generator to generate the data flow graph.

CHAPTER 4

DATA FLOW

4.1 Introduction

In the past, computer system designers have traditionally assumed the von Neumann model of computation. In this model, the computer has a main memory that holds the data and the instructions, and a central processing unit(CPU) that executes these instructions. Instructions are fetched from the memory and are executed one at a time. The results, if any, are sent back to the main memory. This model of execution has an implicit draw back. The sequential access to the main memory constitutes a "bottle neck".

To overcome this problem, advances in technology have made possible radically different computational models. One of these is the data flow model. In data flow, a program is represented by a directed graph. The nodes of the graph represent operations, and the arcs represent data paths. Data items appear as tokens on the arcs. A node can be fired(executed) if all its input arcs have received their data tokens, and there is no token on the output arcs. Figure 4.1 shows a simple graph node before and after it is fired.

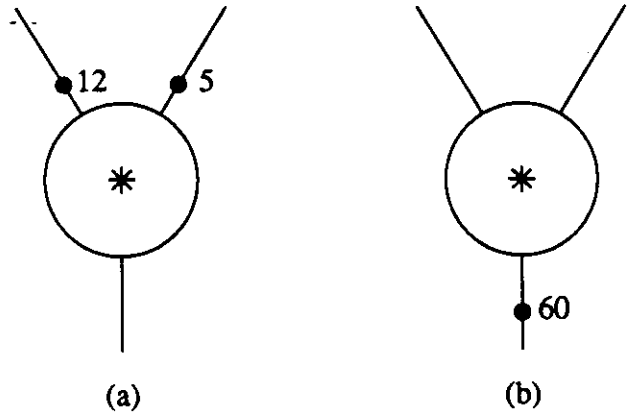


Figure 4.1 A data flow node (a) before and (b) after firing

In data flow the availability of data determines the sequence of execution. Each node acts as a function. There are no side effects or far-reaching data dependencies. A program represented by a data flow graph therefore reveals all its inherent parallelism.

4.2 Data Flow Machines

As Dennis defines them, data flow machines are, in fact, a form of language-based architectures, in which the base language is the program graph[DENN 80]. The source program is converted to a graph, and the hardware implements the formal behavior of the program graph. So far, several data flow languages have been defined. These include ID, developed at the University of California at Irvine[ARVI 76, GOST 79]; VAL, developed at MIT[McGR 80]; and SIMPLE designed at the University of Manchester[GLAU 78].

The design and implementation of data flow machines has been carried at several locations. Probably, the first operating machine based on data flow concepts was DDM1, designed by Burroughs. DDM1 is a recursively structured data driven machine in which concurrent tasks are dynamically allocated to available processors. Another operating data flow machine was developed at Texas Instruments. This is an experimental multiprocessor system capable of accepting a program written in a conventional language, compile it, link it and then partition it across any number of processors[JOHN 80]. The test bed hardware is capable of executing program graphs. The software consists of a FORTRAN compiler, a link editor, an allocator, and a loader. Yet another operating data flow machine is the LAU system designed at CERT, Toulouse, France. In this machine, the source programs are written in a single assignment language, which is translated to instructions directly executable by machine circuits[COMT 79].

A data flow machine based on token-labeling has been designed at the University of Manchester. This machine uses a dynamic tagging model in which each token carries a label that identifies the context of that particular token[WATS 82]. In this way more than one token can be active at each arc of the graph at the same time. This model is very effective in executing iteration loops where many instructions belonging to different iterations of the loop may be active at the same time. At MIT, a cell block architecture has been proposed, which has a large set of instruction cells. These cells are grouped into blocks. Through a distribution network, cell blocks send operation packets to operational units and receive the result packets[DENN 80].

4.3 Data Flow as a Sequencing Tool

In a pure data flow machine, when the program graph is executed, any node whose input data have arrived can be executed by any operational unit. In the simulation of dynamic systems, there are very few iterations or conditional branches. It is in fact possible to analyze the graph prior to the start of the simulation and to preallocate it among the processing units. In ALI, the program graph is divided into packets of sequential code. Each packet is permanently allocated to a specific processor and stored there before starting the simulation. The pure concept of data flow which is based on total asynchrony is, therefore, not followed. Rather, the data flow concept is used as a parallel sequencing tool and is mixed with other methods to meet the system speed requirements.

4.4 Data Flow Graph Representation

In ALI, the data flow graph is a directed graph whose nodes can have a variable number of sources and destinations. Each node represents a function that can be of any complexity; as simple as an addition, or as complex as a two dimensional table lookup. Each data flow node is internally represented as shown in Figure 4.2.

operation	number of sources	s_1	...	s_n	number of dests	d_1	loc_1	...	d_m	loc_m

Figure 4.2 Representation of a data flow node in ALI

Nodes can have a variable number of sources and destinations, but the number of required sources for each operator is known. Nodes of the graph are numbered and stored sequentially. Therefore , the address of each node is its relative number from the top of the graph. In Figure 4.2, s_i is the ith source and d_i is the ith destination. Since the destination node may have many inputs, loc_i indicates the operand number in the destination node where the result is send to. In Chapter 5, the method of allocating the graph among different processors is described. There are some items of information, such as the earliest and the latest execution times of each node and the new added arcs for introducing artificial dependencies, that are needed for doing this allocation. This information is stored in separate data structures.

4.5 Data Flow Graph Generation

In a CSSL program, the INITIAL and the TERMINAL regions are executed only once. Therefore, the main effort to speed up the system is spent in executing the DYNAMIC region and especially the DERIVATIVE section within the DYNAMIC region. Consequently, the DERIVATIVE section is converted into a data flow graph, is extensively analyzed, and is divided among different processors so as to be executed as rapidly as possible.

In order to generate the data flow graph, a stack is maintained to store the operands. The postfix code is scanned from left to right. Whenever an operand is encountered, it is pushed into the stack. Whenever an operator is scanned, a new node is generated for it, and depending upon the type of the operator(unary, binary, etc.), the necessary number of operands is popped from the stack, with the top-most

operand used as the right-mostly operand of the node. The current node number is then pushed into the stack and is used as an operand for the next operation. Whenever a node is popped from the stack to be used as an operand for the current node, the current node is written as the destination for that node. When an assignment operator is reached, the operand at the top of the stack becomes the variable that receives the result of that node. Figure 4.3 shows the algorithm. The result of applying this algorithm to the pilot ejection problem is shown in Figure 4.4. Note that the graph is internally stored in numerical form. Here, it is shown in graphical form for clarity.

Another module, "Format Graph," formats this internally stored information in a printable form. The result is stored in a file called "graphlist" which is in ASCII form and is printable on any hard copy printers. The graphlist file generated for the pilot ejection problem is shown in Figure 4.5. For example, the first three lines indicate that the operation performed at node 1 is a cosine, the input is the identifier which is called "th" in the source program, and the output is send to node 2 where it is used as the second operand. The graphlist file and the method of drawing a data flow graph from the information given in this file is described in detail in Appendix A.

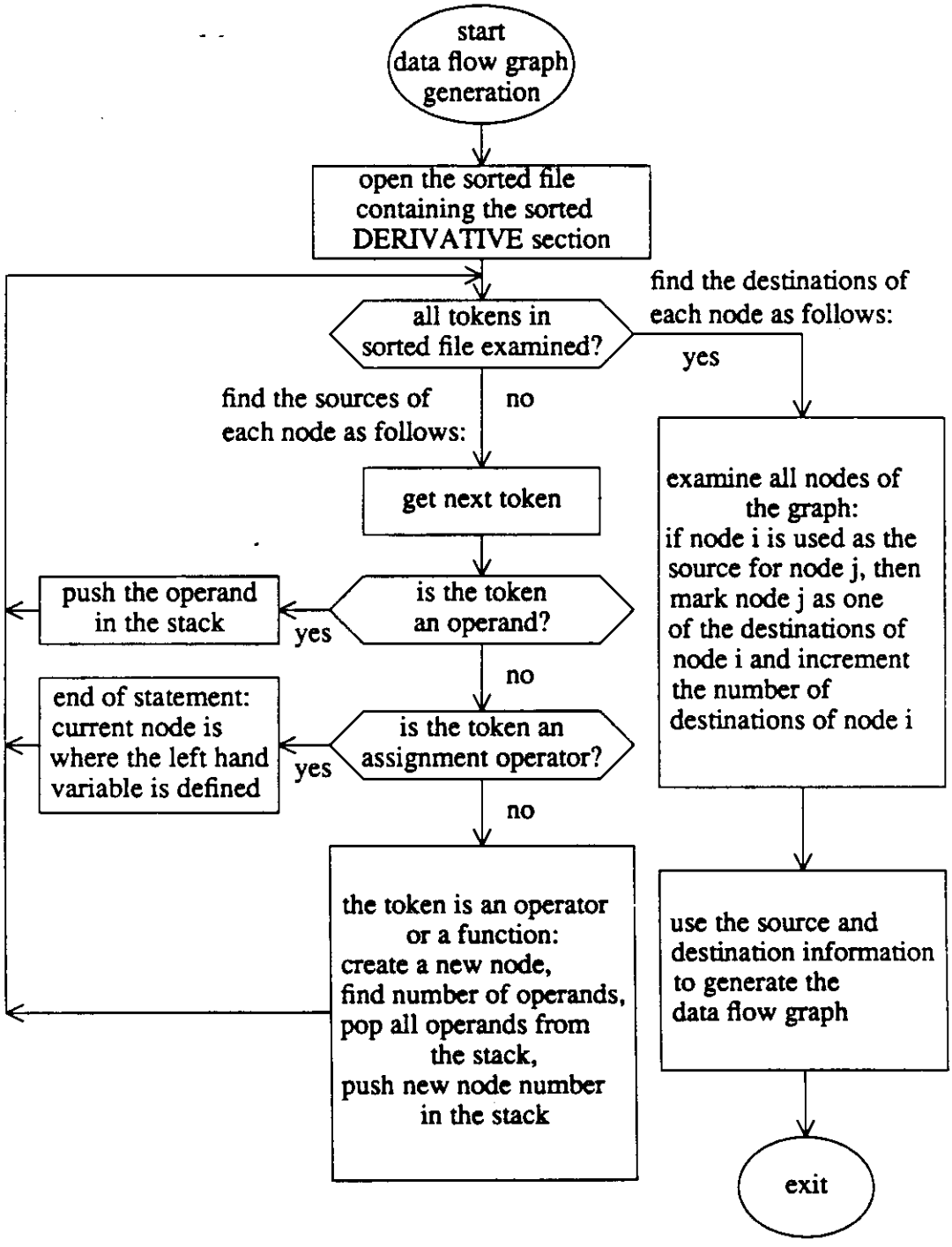


Figure 4.3 Data flow graph generation algorithm

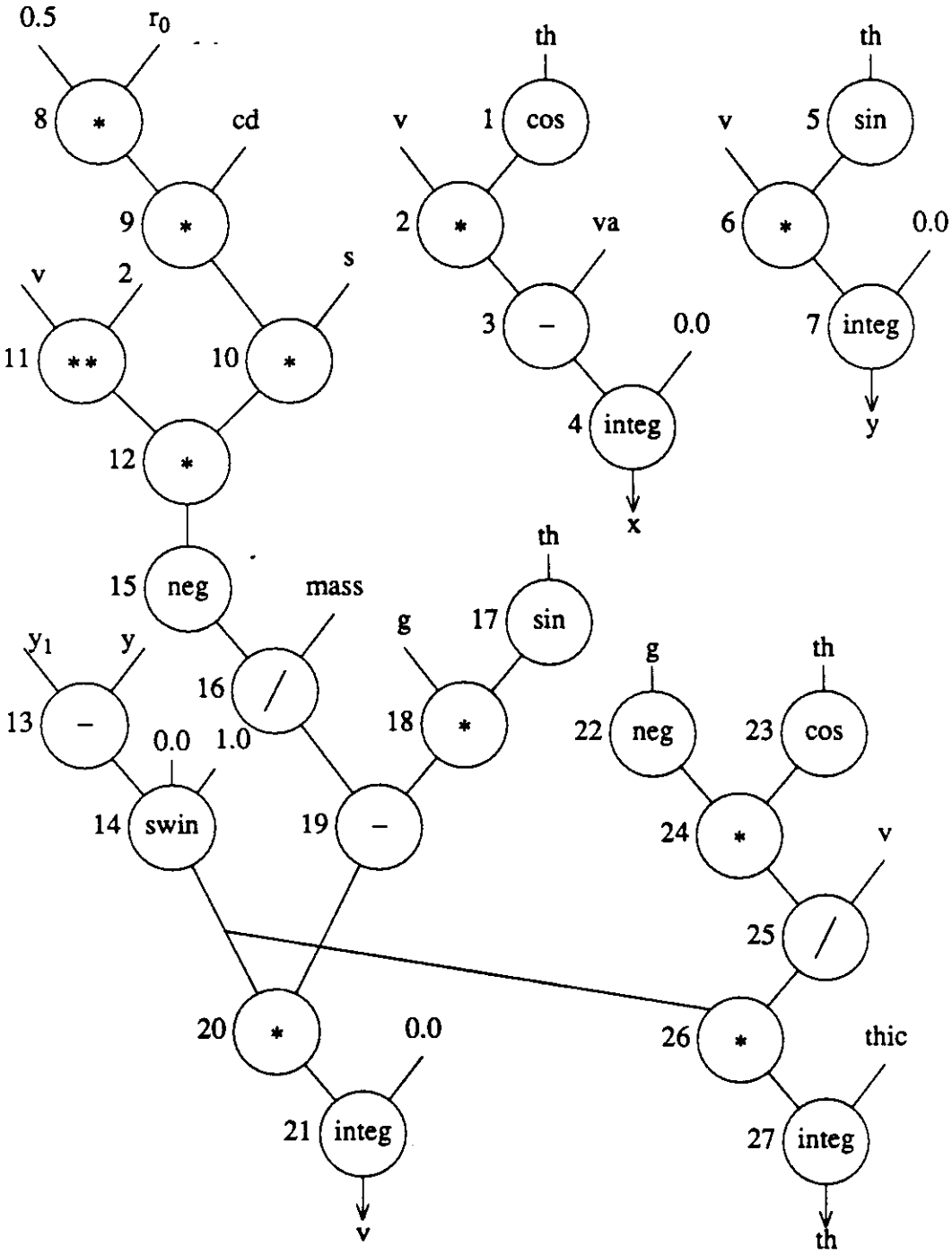


Figure 4.4 Data flow graph of the pilot ejection problem

```
[1]  cos
     1 sources ...      th
     1 destinations ...  node 2 @loc2
[2]  mult
     2 sources ...      v    node 1
     1 destinations ...  node 3 @loc1
[3]  sub
     2 sources ...      node 2    va
     1 destinations ...  node 4 @loc1
[4]  integ
     2 sources ...      node 3    0.0
     1 destinations ...  this is an output node
     var defined ...    x
[5]  sin
     1 sources ...      th
     1 destinations ...  node 6 @loc2
[6]  mult
     2 sources ...      v    node 5
     1 destinations ...  node 7 @loc1
[7]  integ
     2 sources ...      node 6    0.0
     1 destinations ...  this is an output node
     var defined ...    y
[8]  mult
     2 sources ...      0.5    r0
     1 destinations ...  node 9 @loc1
[9]  mult
     2 sources ...      node 8    cd
     1 destinations ...  node 10 @loc1
[10] mult
     2 sources ...      node 9    s
     1 destinations ...  node 12 @loc1
[11] **
     2 sources ...      v    2
     1 destinations ...  node 12 @loc2
[12] mult
     2 sources ...      node 10    node 11
     1 destinations ...  node 15 @loc1
     var defined ...    d
[13] sub
     2 sources ...      y1    y
     1 destinations ...  node 14 @loc1
```

Figure 4.5 The graphlist file for the pilot ejection problem
(continued on the next page)

```
[14]  swin
      3 sources ...      node 13    0.0    1.0
      2 destinations ...  node 20 @loc1    node 26 @loc1
      var defined ...     yge1
[15]  negate
      1 sources ...      d
      1 destinations ...  node 16 @loc1
[16]  div
      2 sources ...      node 15    mass
      1 destinations ...  node 19 @loc1
[17]  sin
      1 sources ...      th
      1 destinations ...  node 18 @loc2
[18]  mult
      2 sources ...      g    node 17
      1 destinations ...  node 19 @loc2
[19]  sub
      2 sources ...      node 16    node 18
      1 destinations ...  node 20 @loc2
[20]  mult
      2 sources ...      yge1    node 19
      1 destinations ...  node 21 @loc1
[21]  integ
      2 sources ...      node 20    vic
      1 destinations ...  this is an output node
      var defined ...     v
[22]  negate
      1 sources ...      g
      1 destinations ...  node 24 @loc1
[23]  cos
      1 sources ...      th
      1 destinations ...  node 24 @loc2
[24]  mult
      2 sources ...      node 22    node 23
      1 destinations ...  node 25 @loc1
[25]  div
      2 sources ...      node 24    v
      1 destinations ...  node 26 @loc2
[26]  mult
      2 sources ...      yge1    node 25
      1 destinations ...  node 27 @loc1
[27]  integ
      2 sources ...      node 26    thic
      1 destinations ...  this is an output node
      var defined ...     th
```

Figure 4.5 (Continued)

CHAPTER 5

PRESCHEDULER

5.1 Introduction

The use of a network of microprocessors raises two basic problems. First, in order to take advantage of multiple processors, the execution load of each processor must be carefully determined and balanced. Second, communication delays between processors must be minimized. This involves determining data dependencies in the program and allocating the highly communicating tasks to the same processor which may create load imbalance. Therefore, the optimal solution for both problems may be contradictory. Finding the best allocation strategy requires designing an efficient scheduler.

A scheduler is a program which allocates the resource of processor time. More specifically, it determines which task a given processor must be executing at each moment in time.

A significant portion of scheduling concepts was first developed in the field of operations research for job-shop or assembly-line applications. These concepts were later on used and adapted for computer resource scheduling. In the following sections, different concepts of scheduling theory , as tailored for computer

applications, are discussed. In each case, the type of problems supported by ALI are identified. Followed by description of the scheduling algorithm used in ALI.

5.2 Scheduling Strategies

Scheduling problems can be studied in two levels of abstraction: user-management level and resource-management level(Figure 5.1). In the user-management level, jobs are independent programs submitted by a large population of independent users. Schedulers dealing with this level are job schedulers designed to improve the overall system performance. These schedulers are aimed to minimize the user waiting times, to maximize resource utilization, and to maximize the system throughput.

In the resource-management level, tasks which are usually parts of a larger program compete for resources. Schedulers dealing with this level are task schedulers which allocate physical resources to tasks. These schedulers are aimed to execute the tasks within certain execution deadlines, to minimize the execution length of individual programs, and to maximize resource utilization.

In each level of abstraction, jobs or tasks are represented by a set of attributes such as arrival times, durations, and dependencies. If these attributes are known prior to run time, a deterministic model for the problem can be made. If any one of these attributes are not known, the problem can only be modeled by stochastic methods. Problem models found by either method are used to find the best allocation strategy to meet the given performance criteria.

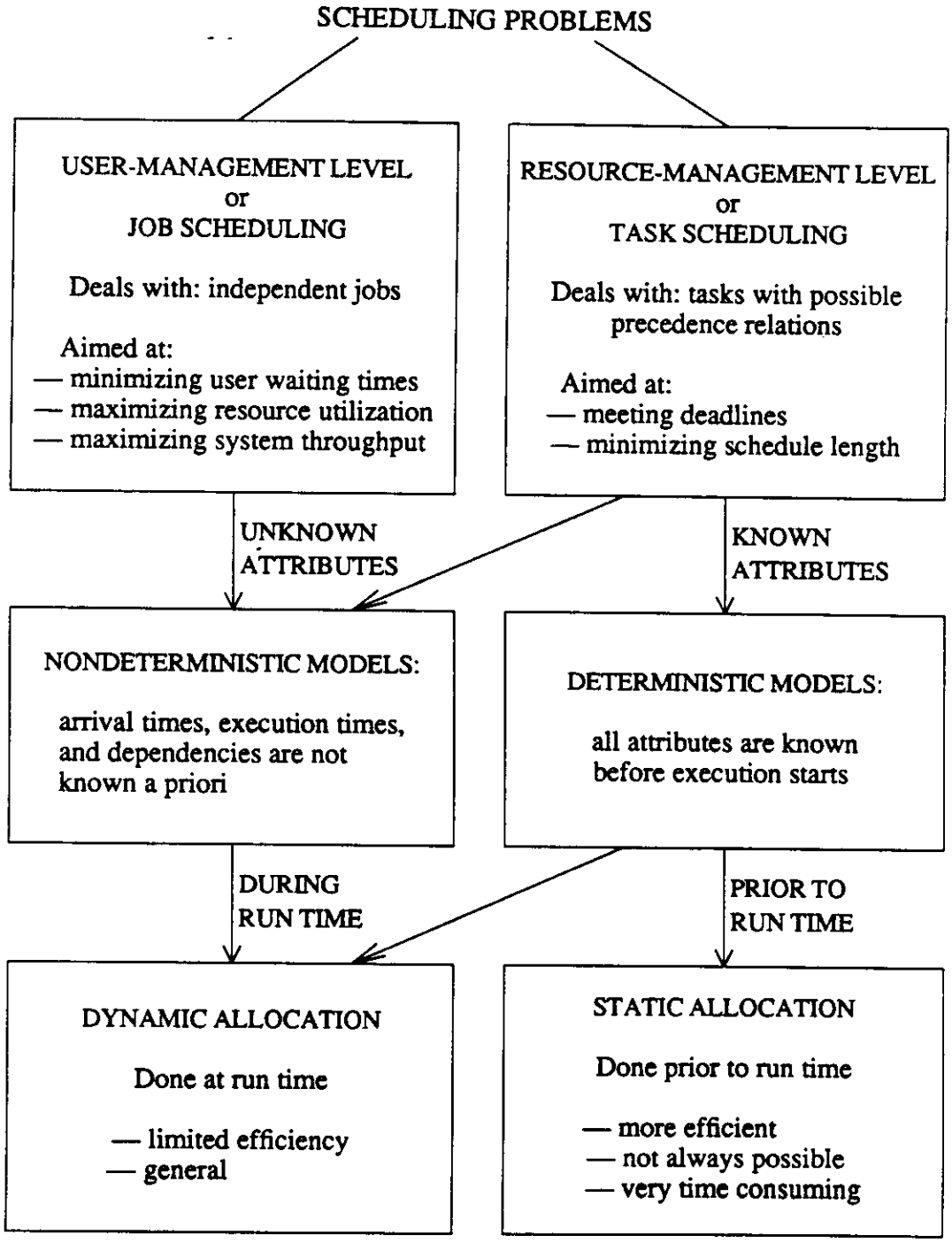


Figure 5.1 Scheduling problems

Allocation strategies can be viewed as either static(done prior to run time) or dynamic(done during rum time). Problems with nondeterministic models can only be scheduled in run time; while deterministic problems can be scheduled either prior to or during execution time.

Job schedulers improve the overall system throughput without favoring a special program. For example, they usually delay longer tasks in favor of smaller ones to minimize user waiting times. In real-time applications, on the other hand, meeting deadlines is the most important goal of the system, and the system is dedicated to the simulation problem on hand. Therefore, a task scheduler that optimizes execution of a single program is more suitable for these applications.

In ALI, the whole multiprocessing system runs a single simulation problem at a time; and a task scheduler, called the prescheduler, analyzes the program to find the best allocation in order to meet real-time deadline requirements. Therefore, job schedulers are not discussed any further, and the rest of the chapter discusses task schedulers in more detail.

5.3 Dynamic Allocation

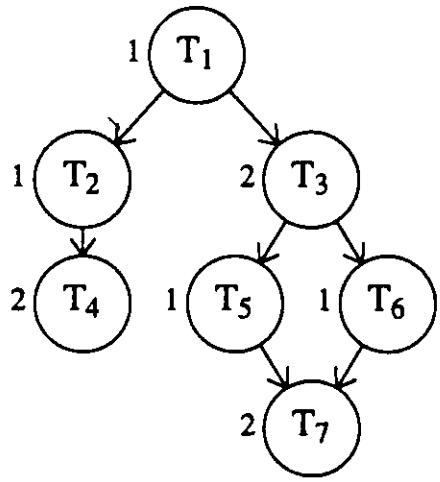
If the nature of the problem is such that any of the model attributes such as arrival times, execution times, or precedence dependencies are not known in advance, scheduling must be done during run time. When a processor becomes idle, it notifies the scheduler to get new work load. The scheduler selects a task that has received all its inputs and is ready to be executed. However, assigning the first

task to the first available processor, without analyzing the rest of the program in detail, may not be the best scheduling strategy, and it may increase the overall delay time.

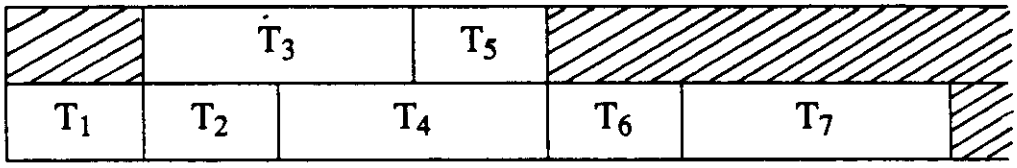
To understand this, consider the simple case of two tasks T_i and T_j. T_i becomes ready for execution first. However, it results are not needed in near future. T_j becomes ready for execution just a few moments later, but its results are needed right away. If the scheduler is smart and can predict that T_j will be ready shortly, it can delay execution of T_i and give more priority to T_j.

For example, suppose the simple program shown in Figure 5.2(a) is to be executed on two processors. The allocation is shown in the form of a Gantt chart[CLAR 52], which consists of a time axis for each processor with intervals marked off and labeled with the name of the tasks being executed. In Figure 5.2(b), at time t equal to 3, processor P_1 becomes available and immediately starts executing the only ready task, T_3. The program is executed in seven time units. In Figure 5.2(c), however, processor P_1 is left idle for one time unit to let task T_2 , whose results are needed by T_6, to be executed. As a result the program execution is done in only six time units.

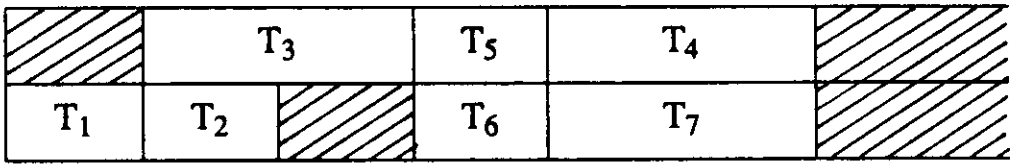
Moreover, when several tasks are ready for execution, the scheduler must find the best candidate for execution. The more intelligent a scheduler is, the more time it needs to analyze the interaction between the tasks to find the best candidate for execution. However, since dynamic scheduling is done in real time, a lot of analysis may overshadow the gain of using a multiprocessor system. Therefore,



(a)



(b)



(c)

Figure 5.2 A two processor allocation

dynamic scheduling, although general, is suboptimal.

Dynamic scheduling is best suited for systems that have structures that must be resolved in run time. These structures, such as iteration loops without a predetermined number of iterations, make impossible to analyze the program prior to run time.

5.4 Static Allocation

If all attributes of tasks are known a priori, an extensive analysis can be performed prior to run time to find the best allocation strategy. In the case that all tasks belong to a single program, it is desirable to further analyze the program and to group the tasks into execution packets.

Each packet is a group of closely related tasks with minimum interaction with other parts of the program; so that when all inputs are ready, it can be executed to completion without any further inputs from the outside. In this way, the whole packet can be assigned to one processor. Thereby, minimizing the interprocessor communication delays. Packets can be sent to each processor before the actual run begins. At run time, all each processor has to do is to wait for the inputs of each packet to arrive. The packet is then ready for execution immediately.

5.5 Classification of Scheduling Problems

A scheduling problem can be described by three attributes: resources, tasks, and performance criteria. Various classifications for scheduling problems are possible, depending on what subset of each attribute they cover. In the following sections, each of these attributes are discussed, identifying features supported by ALI.

5.6 Resources

Each computing system has several resources that the executing tasks compete for them. Processor time, memory space, and input/output devices are some of these resources. Processor time is allocated by the scheduler, while memory space is allocated by the memory management system, and input/output operations are controlled by I/O handlers.

The main goal of a multiprocessor system is to gain speed. This makes the resource of processor time the most important of all resources. The availability of inexpensive memory has led to the assumption that all processors in ALI can keep their executable code and the related data in memory during execution. Furthermore, solution of ordinary differential equations is not an I/O bound problem. Therefore, the allocation of processing time is the only resource allocation problem considered here.

5.6.1 Number of Processors

The number of processors determines what kind of actions need to be taken by the scheduler. In single processor systems, the scheduler acts as a sequencer that determines what task is to be executed by the processor, and how long the execution of that task must continue before another task is executed. In multiprocessor systems, the scheduler is to determine which task is to be executed on each processor at each moment; problems such as load balancing and minimizing communication delays must be carefully considered. ALI is designed for multiprocessor systems. The number of processors is a tunable parameter, therefore, any number of processors can be used in the system.

5.6.2 Type of Processors

In multiprocessor systems, processors can either be identical(homogeneous) or different(heterogeneous). Advantage of heterogeneous systems is that they can be upgraded by new and more powerful processors as they become available. However, the task of a scheduler will be more difficult if some processors are capable of performing some operations faster and more efficiently than the others. This is because the scheduler has to decide which processor is best suited to perform each task and to decide whether to delay the allocation of a task until the suitable processor is available, or whether to allocate it to a processor that is available now but executes the task less efficiently. Only homogeneous systems are handled by ALI.

5.7 Tasks

The nature of tasks executed on the system, such as their arrival rates, execution times, interruptibility, and dependencies are very important in selection of an allocation strategy best suited for each class of applications. Here, a task is considered to be a unit of computation that can be executed to completion without any interaction with other tasks.

If all tasks have known attributes before execution starts, the allocation problem is deterministic and as was discused before, a static scheduling algorithm can be used for it. If any of these attributes are unknown, the scheduling problem is nondeterministic and is solved by stochastic approaches.

5.7.1 Arrivals and Durations

When the arrival time and duration of tasks are not known a priori, a probability distribution function can be used to describe the time between successive arrivals and duration of tasks. In many scheduling strategies that deal with scheduling jobs in an operating system, job arrivals are regarded as independent, random events. In other cases, a worst-case analysis can be made by using the maximum possible values of task execution times.

In deterministic problems, further restrictions may be enforced. For example, algorithms are developed that assume all tasks have unit execution times[HU 61]. Other algorithms, as is the case in ALI, may assume that all execution times are mutually commensurable[MUNT 69]. This requires that all

execution times are divisible by a common integer an integral number of times. This assumption is not very restrictive since general execution times can be approximated arbitrarily closely by a set of mutually commensurable values.

5.7.2 Dependencies

The individual tasks in a scheduling problem may have constraints on each other requiring some tasks be completed before others can be started. This is because of inherent sequentiality of the problem which requires a task to wait for the result of computation done by another task. This is usually referred to as a partial order[CONW 67] and is represented by a precedence graph. In a precedence graph, nodes represent tasks and edges represent precedence relations.

Various degrees of precedences may apply to a given set of tasks. At one extreme, all tasks may be independent. This is usually the case where all tasks are different jobs submitted by independent users. In other cases, precedences are sequentiality constraints among different tasks.

Some algorithms may require further restrictions. For example, they may only apply to rooted trees where each node has only one successor. ALI, as will be described in detail, is designed to accept arbitrary precedence graphs with known dependencies.

5.7.3 Interruptibility

In some scheduling strategies, once a task is started on a processor, it must execute to completion. These are nonpreemptive strategies. In preemptive strategies, on the other hand, execution of tasks can be interrupted to let other tasks execute. Preempted tasks are resumed later on and are eventually given enough time to complete. Preemptive strategies involve some overhead for context switching and will result in better performance only if preemption does not occur frequently.

Nonpreemptive strategies have the virtue of simplicity of implementation and good utilization of machinery. The scheduling algorithm used in ALI is nonpreemptive; once a task is started on a processor, it must execute to completion.

5.8 Performance Criteria

The goals of using a multiprocessor system are different from system to system. Hence, a suitable allocation strategy must be selected accordingly. In some applications, a multiprocessor system is used to increase the overall throughput of the system. In these systems, minimizing the processor idleness and mean flow time is desirable.

In other applications, such as real-time simulation, a multiprocessor system is used to meet a certain fixed deadline not achievable by using a single processor in the same cost range. In these systems, minimizing the execution time to reach the deadline is desirable. Some of these criteria are discussed in the following sections.

5.8.1 Meeting Deadlines

In some scheduling problems,such as those controlling or simulating a real-time process, tasks must be completed before a given deadline. Here, the problem is not only to minimize the scheduling completion time, but the deadline must be met in order to interact with external hardware.

5.8.2 Minimizing Completion Time

In many scheduling algorithms, the goal is to minimize the completion time of programs, although there is not a deadline imposed on the execution. Besides the obvious reason of finishing a program faster, these schedulers are indirectly optimizing the processor utilization and system throughput.

If f_i denotes the finishing time of task i, then the completion time of a schedule with n tasks is defined as[COFF 76]:

$$T = \max_{1 \le i \le n} \left\{ f_i \right\}$$

On the other hand, the processor utilization is defined as the total time that processors were busy doing execution, divided by the total time they were available. Therefore, in a system with m processor, the processor utilization is defined as[COFF 73]:

$$U = \frac{\sum_{i=1}^{n} d_i}{m\,T}$$

Where d_i is the duration of task i

As can be seen, reducing the completion time, T, maximizes the processor utilization.

Furthermore, if T_i is the work load on processor i, then the total processor idleness can be defined as:

$$I = (T - T_1) + (T - T_2) + \cdots + (T - T_n)$$

$$= mT - \sum_{i=1}^{m} T_i = mT - \sum_{i=1}^{n} d_i$$

Since the total work load on all processors is equal to total program execution time(which is a fixed value), reducing T minimizes the total processor idleness.

5.8.3 Minimizing Number of Processors

This goal, through reducing the number of processors, reduces the hardware costs, indirectly reduces interprocessor communications(because execution is done on fewer processors), and enables a system with a limited number of processor to execute a large problem and meet its deadlines.

5.8.4 Minimizing Mean Flow Time

Flow time(or time in system) of a task is its completion time. The mean flow time of a schedule is defined as[COFF 73]:

$$\bar{t} = \frac{\sum_{i=1}^{n} f_i}{n}$$

where f_i is the completion time of task i and n is the total number of tasks.

This measure is usually used for operating systems that deal with independent jobs submitted by independent users. Reducing the mean flow time, reduces the user waiting times; and at the same time, releases system resources, such as memory, to be used by other jobs.

5.9 Scheduling Algorithms

Many algorithms have been devised for scheduling a set of tasks with given precedence relations. Among these algorithms, only those that can result in a desirable allocation without trying all possible solutions are of practical use. The execution of the known algorithms for general scheduling problems(problems with a set of tasks with arbitrary precedences) are not, however, bounded by a polynomial of the number of the tasks.

Instead, their running times are exponential in the length of their input and as the number of tasks increases, the time to find the best allocation becomes intolerable. In fact, it is shown that if there is a solution for the general scheduling problem in polynomial time, then a large group of other problems, such as the classic traveling salesman problem(find the shortest route for a salesman who must visit n cities), can also be solved in polynomial time[COOK 71, KARP 72]. All these problems are in the class of the so-called NP-complete problems which can be

solved in polynomial time only by a nondeterministic machine(a machine that can guess a solution). All NP-complete problems are reducible to each other in the sense that if at some time in the future one of them is found to have an efficient method of solution, that method can be modified to apply to all the others.

Presently, there are two approaches to find the best solution for general scheduling problems: dynamic programming approach and critical-path approach. The dynamic programming method[RAMA 72] groups tasks into subsets, called precedence partitions, to indicate the earliest and latest times during which tasks can be started and still guarantee minimum execution time for the graph. Using these partitions, algorithms are developed to determine the minimum number of processors required to process a graph in the smallest possible time and to determine the minimum time to execute a graph on a given number of processors.

The critical-path method is used by Barskiy to find the minimum number of processors to execute a graph within its critical path[BARS 68]. This method involves adding additional precedence relations to reduce the number of simultaneously active nodes to less than or equal to the number of processors. By doing this, it is guaranteed that there is always enough computing power to execute all nodes of the graph as soon as they become ready for execution. Fernandez has enhanced Barskiy's algorithm to make it more efficient[FERN 72]. Algorithms are designed that use the critical-path approach for finding the best allocation for a general graph[BUSS 74, LEVY 73]. These algorithms use heuristics to introduce additional precedence relations and to do backtracking if the time of the longest path in the graph is increased.

The problem with both dynamic programming and the critical path approach is their time complexity. When the number of nodes in the graph grows, the time required to execute the graph becomes insurmountable. With the worst-case solution growing exponentially with the number of tasks, the next options are either to restrict the precedence relations or to use approximate(or suboptimal) solutions.

The extreme case for restricting the precedence relations is to assume that all tasks are independent. Algorithms exist to find the best allocation for a set of independent tasks.[MCNA 59]. The next level of complexity is when the precedence graph is a rooted tree(a graph which each node has at most one successor). There are efficient algorithms to find the shortest execution time of a rooted tree when the task durations are all the same and preemption is not allowed[HU 61], and when the task durations are mutually commensurable and preemption is allowed[MUNT 70]. Other special cases involve finding the minimum execution time of a general precedence graph with mutually commensurable execution times on two processors for both preemptive schedules[MUNT 69], and nonpreemptive schedules[FUJI 69, COFF 72]. Suboptimal algorithms are designed for the dynamic programming approach which use heuristic alternatives[RAMA 72], and for the critical-path approach[LEVY 73] which does not backtrack if the length of the critical path is increased.

None of the mentioned algorithms consider the penalty of interprocessor communication delays due to arbitrary division of the graph among the processors. In practice, these delays can result in sever degradation of system performance if too much communication traffic is required.

5.10 Scheduling Strategy in ALI

In order to be able to analyze a program completely, it is necessary to estimate the approximate time spent in each part of the program. Study shows that a large number of subsystems can be modeled without using structures of unpredictable length. In many simulation problems, all program segments are known in advance, as is the time required for their execution. Table lookups are exceptions and require some iteration. The whole table lookup procedure is considered as one operation, and an average time is used for its execution.

In a simulation program written in CSSL-IV or ACSL, most of the execution time is spent in the DERIVATIVE section, in which, for each time step, the value of all state variables are calculated independently of each other, and the results are exchanged at the end of the time step. In order to simulate the system in real-time, a deadline must be met in completing the execution of each time step. The INITIAL and the TERMINAL regions are executed once at the beginning and at the end of each run respectively and do not have a deadline. Therefore, it is sufficient to find a good scheduling algorithm for the DERIVATIVE section.

One way of analyzing programs to find independent packets is to use a data flow graph. The "Data Flow Graph Generation" stage in ALI converts the statements of the DERIVATIVE section into a data flow graph, making it possible to analyze the program and allocate independent packets to different processors prior to run time.

Among all methods mentioned in the previous section, the critical-path approach is the most practical and straight forward. The algorithm used in ALI is very similar to the suboptimal algorithm defined by LEVY[LEVY 73]. Although, many modifications are done to make it faster and more efficient. Several versions of the algorithm with different heuristics are examined.

5.10.1 Definitions

Before describing the algorithm, a few definitions are introduced. For simplicity, it can be assumed that each graph has only one entry and one exit node. This does not reduce the generality, because any arbitrary graph can be converted to single entry, single exit graph by adding dummy nodes to the top and the bottom of the graph.

Definition 1: A "basic scheduling" is one that can assign each ready task to the first available processor, without artificially delaying the processors.
As mentioned before, a basic scheduling may not be the best strategy on an arbitrary graph. However it is possible to transform a graph to another graph, by adding artificial precedences, so that basic scheduling on the transformed graph can result in an optimal schedule on the original graph.

Definition 2: A "critical path" in a graph is the longest path from its entry node to its exit node.
The time that takes to execute the critical path is denoted by tcp and is the lower bound on the execution of the total graph. The length of the critical path is

determined by the sequential precedences of the graph and therefor cannot be reduced by adding more parallelism.

Definition 3: The "earliest completion time" of a node is the earliest time the node can be finished without violating the graph precedences. It is defined as:

$$e_c = \max (t (\pi_k)) + t_j \qquad (j = 1, 2, ..., n)$$

where π_k is the kth path the entry node through node j

Definition 4: The "latest completion time" of a node is the latest time by which the node can be executed without increasing the length of the critical path. It is defined as:

$$l_c = \min (t_{cp} - t (\hat{\pi}_k)) \qquad (j = 1, 2, ..., n)$$

where $\hat{\pi}_k$ is the kth path the exit node node j

Definition 5: "Activity of a node" is defined as a function whose value is one in the interval where the node is active and zero otherwise[BARS 68].

$$a (C_j , t) = \begin{cases} 1 & \text{for } t \in [C_j - t_j , C_j] \\ 0 & \text{otherwise} \end{cases}$$

where C_j is completion time of node j

Definition 6: The "load density function" is defined by :

$$F (C , t) = \sum_{j=1}^{n} a (C_j , t)$$

where C is either the earliest or the latest completion time

The value of the load density function in each time step shows how many nodes are active in that interval[BARS 68]. If there are more active nodes than processors,

then some of the nodes should be executed at a later time to reduce the active nodes .

Definition 7: The "maximum height of the load density function" is the maximum number of processors that are required to execute the graph without increasing the length of the critical path.

$$m_{max} = \left\lceil \max_{t=1,\,t_{cp}} \left| F(C, t) \right| \right\rceil$$

This is usually higher than the number of processors needed to execute the graph within its critical path length.By moving the execution of the nodes that are not needed immediately to other time steps, the height of the load density function can be reduced, thereby reducing the number of the required processors.

There is also a minimum number of processors that is required to execute a graph within the tcp or any other deadline.

Lemma : The minimum number of processors is found by dividing the total execution time of the graph by the length of the critical path(or any other deadline).

$$m_{min} = \frac{\sum_{i=1}^{n} t_i}{t_{cp}}$$

This is just an estimate, and since the work load is not uniformly divided through the time, the actual minimum number may be more than this lower bound. There are other formulas that give sharper bounds[FERN 73]. However, they require more calculation time . In ALI, the simple bound given above is used, assuming that if the number were not sharp enough the number of processors will be increased.

Another question that may arise is: What is the minimum time required to execute a given program with exactly m processors?

Lemma : A lower bound for the execution time is found by dividing the total execution time of the program(in data flow, the total execution time of all nodes of the graph) by the total number of the processors. And since the lower bound can not be less than the length of the critical path, the lower bound will be:

$$t_{min} = \max\left\{ t_{cp} , \frac{\sum_{i=1}^{n} t_i}{m} \right\}$$

Again there are other formulas that give sharper bounds but take more time to calculate[FERN 73], [LANG 77].

These definitions will be clarified by an example. The graph in Figure 5.3 has ten nodes. Two dummy nodes are added to make it a single entry, single exit graph. Node numbers are shown inside each node, and the execution times of the nodes are written next to them. There are two critical paths in the graph and are shown with double lines. The length of the critical path is four time units.

Figure 5.4(a) shows the nodes when they are at their earliest completion time. This is the earliest time that each node can be executed without violating the given precedences. For example, node 4 can only be executed after node 1 is executed, and so on.

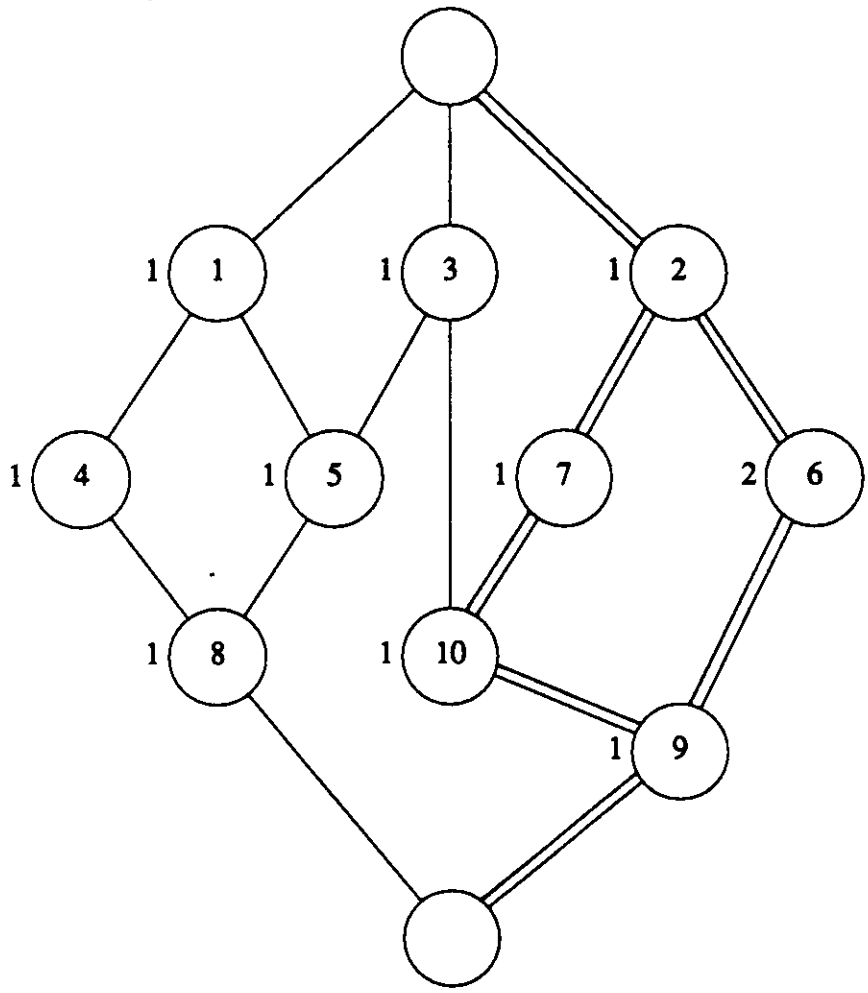
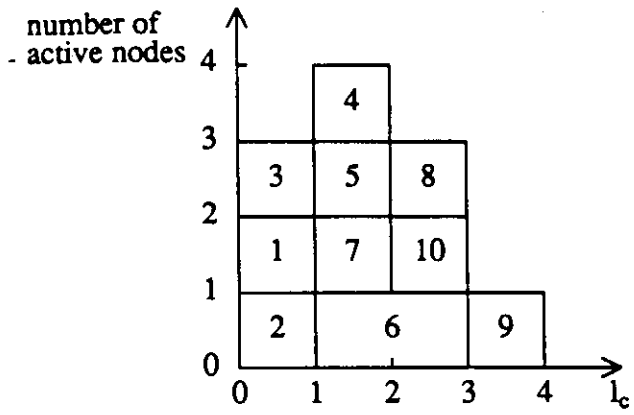
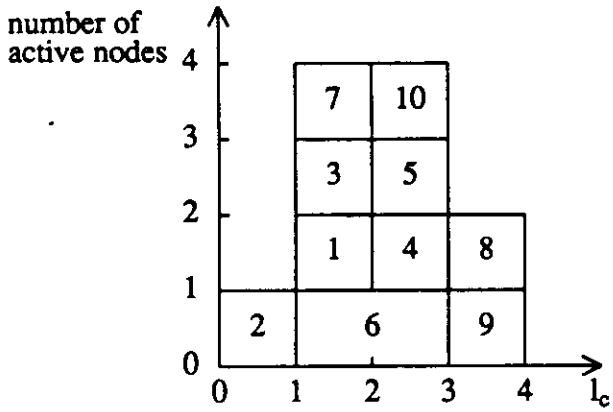


Figure 5.3 A sample graph

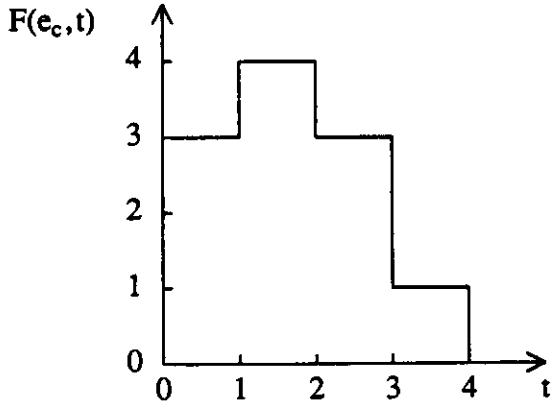
Figure 5.4(b) shows the nodes when they are at their latest completion time. This is the latest time that each node must be executed without increasing the length of the critical path. Comparing Figures 5.4(a) and 5.4(b), it can be seen that the nodes on the critical path have zero slack time and are started at the same time in both charts. Nodes that are not on the critical path,however, have some slack time. For example node 4 can be delayed one time step without increasing the length of



(a) The earliest completion time



(b) The latest completion time



(c) The load density function

Figure 5.4 Load density and completion functions of the sample graph

the critical path.

Figure 5.4(c) shows the load density function for the earliest completion time, which is different from the load density function for the latest completion time. The height of this function at each time step is the number of the active nodes at that time step. This is the maximum number of processors required if all nodes are to be executed at their earliest possible time. This upper bound in Figure 5.4(c) is four. The graph ,however, can be executed with a smaller number of processors without increasing the length of the critical path. This requires delaying some of the nodes to later time steps where less than four nodes are active.

This is the main concept of our allocation strategy. By delaying those nodes that have some slack time, the graph is executed with fewer processors while the deadline is still met. For example this graph can be executed by three homogeneous CPU if nodes 2, 6 and 9 are assigned to CPU 1; nodes 1, 4, 5 and 8 are assigned to CPU 2; and nodes 3, 7, and 10 are assigned to CPU 3.

5.11 Prescheduling Algorithm

Assumptions : The algorithm derived here assumes that all processors are of the same type, and tables are not very large and can be kept in any processor local memory. When a packet is being executed, the CPU is noninterruptible and execution continues to completion.

The algorithm is very similar to the near optimal algorithm defined in [LEVY 73]. It uses heuristic methods for finding the best allocation. But it does not backtrack to check if the length of the critical path is increased. More rigorous algorithms are defined in [BUSS 74], which are very time-consuming and are not bounded by a polynomial of the number of the graph nodes.

The first step is to calculate the earliest and the latest completion times of all nodes and the time of the original critical path. Starting from the top of the graph, the earliest completion times are calculated first. The time of the critical path is the largest of the earliest completion times of the nodes in the graph. Starting from the bottom of the graph, the latest completion times are calculated next.

For each time step for which the number of active nodes ,r , is greater than the total number of processors,m , r-m dependencies are added between active nodes. So that r-m nodes are delayed for later time steps, and only m nodes remain active for this time step to be executed by m processors. It is important to decide which dependencies to add.

Each dependency corresponds to an arc which is added to the graph. For each time step a list of all active nodes that can be used as the head of these arcs and a list of all active nodes that can be used as the tail of these arcs are created. These are called head and tail lists respectively[LEVY 73]. When a node is selected as the head of an arc, this node will be delayed until the execution of the node at the tail is completed. This suggests that it is better to add an arc between a tail node with the smallest earliest completion time and a head node with the largest latest initiation

time, so that by the time the tail node is completed, there still remains sufficient time to schedule the head node. For this reason, the nodes in the tail list are sorted in ascending order of the earliest completion time. Similarly, the nodes in the head list are sorted in descending order of their latest initiation time.

Bussel, et al.[BUSS 74] and Levy[LEVY 73] do not include nodes which are on the critical path in the head list, because delaying these nodes will increase the length of the critical path. For a suboptimal scheduler, however, this is not a good strategy. A suboptimal scheduler does not backtrack if the length of the critical path is exceeded. Therefore, delaying the nodes on the critical path may result in smaller increases in the execution time than other nodes. This is confirmed by the results of different benchmark problems given in Chapter 8.

Another difference between the algorithm in ALI and Levy's algorithm is treatment of redundant arcs. When a new arc is added, some already existing arcs may become redundant. For example, in Figure 5.5 after addition of the arc between node 3 and node 2, the arc between node 1 and node 2 becomes redundant. Levy's algorithm deletes redundant arcs by an algorithm that has complexity of $O(n^2)$. In ALI, procedures for calculation of the earliest and latest completion times are designed in a way that redundant arcs do not effect these calculations. Therefore, there is no need to make special effort to delete the redundant arcs.

After the head and tail lists are ready, for each time step where there are more active nodes than m(the number of the processors), enough arcs are added to reduce the active nodes to m.

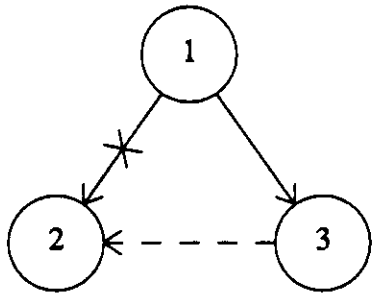


Figure 5.5 A redundant arc

The first nodes from the head and the tail lists are chosen. If both nodes are the same or if the added arc creates a cycle, that arc is not acceptable and another arc is tried. The next candidate arc is selected such that the difference between the latest initiation time of the head and the earliest completion time of the tail is maximum. When an arc is found; the earliest and latest completion times and the length of the longest path are updated and another arc is added until the number of active nodes for this time step is reduced to m. Then the next time step is processed.

Two versions of the algorithm, hereby referred as A and B, are developed. Algorithm A does not allow processor idle times. Only those nodes which become active at each time step are used in the head list and therefore be delayed. Other active nodes which are started in previous time steps are not included in the head list.

Algorithm B, however, allows processor idle time. Therefore, it includes all active nodes in the head list. The comparison between the two algorithms and their performance evaluation are done in Chapter 8. The algorithm is formally defined in the next section.

5.12 Formal Definition of the Algorithm

This algorithm(Figure 5.6) transforms a graph to another graph that could be allocated to m processors using a basic scheduling method:

1) $t \leftarrow 0$
   compute e_c, l_c and t_{cp}

2) $t = t + 1$
   if $t > t_{cp}$ stop. The graph transformation is complete.

3) if $F(e_c,t) = r < m$ go to 2

4.1) Build the head list from the active nodes(for Algorithm A, only include those active nodes which start at time step t).
   Sort the head list according to descending order of the latest initiation time.

4.2) Build the tail list from the active nodes
   Sort the tail list according to ascending order of the earliest completion time.

5) DO FOR arccount = 1 to r - m
5.1) choose the first node in the tail list as T.
5.2) choose the first node in the head list as H.
5.3) if $T \neq H$ and if arc(T, H) does not create a cycle, go to 5.
5.4) choose the next available node in the tail list as NEWT.
5.5) choose the next available node in the head list as NEWH.
5.6) if $l_{c_H} - t_H - e_{c_{NEWT}} > l_{c_{NEWH}} - t_{NEWH} - e_{c_T}$ then
   choose NEWT as T
   else
   choose NEWH as H
   go to 5.3
5.7) add arc(T, H) to the graph,
   delete T from the tail list,
   delete H from the head list, and
   update e_c, l_c, and find the new critical path (if it has changed)
   arccount $\leftarrow$ arccount + 1

6) go to 2

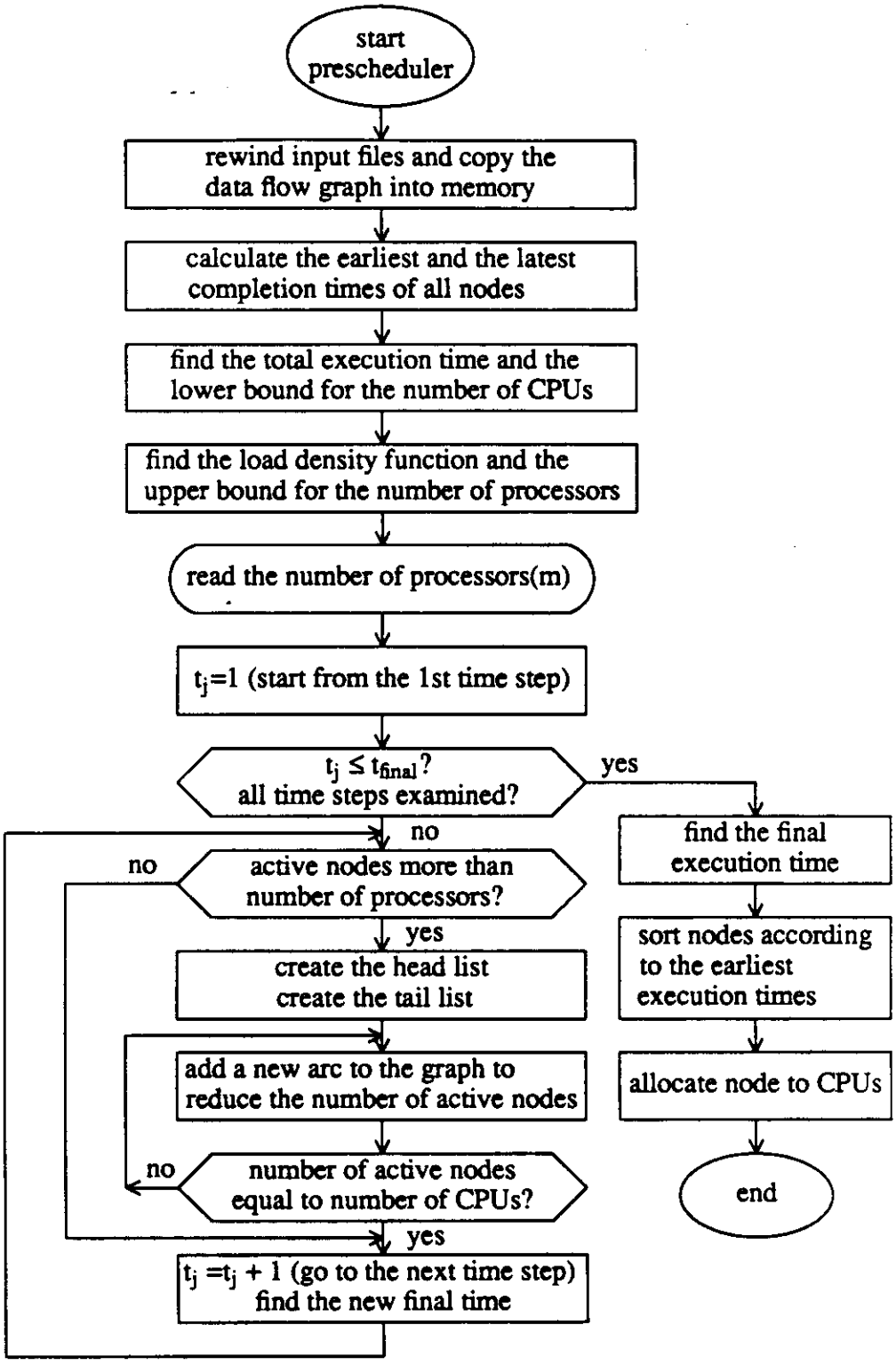


Figure 5.6 The prescheduling algorithm

5.13 Assigning the Work Load to Processors

By adding the precedences to the graph, the number of active nodes at any time step is reduced to less than or equal to the number of processors. This guaranties that, at any time step, there are enough processors to execute all active nodes.

Therefore, basic scheduling can be applied to the graph to allocate it among the processors. However, basic scheduling alone does not take into account the interprocessor communication delays. Interprocessor communication is needed when a node and one of its successors are executed by two different processors. However, this does not necessarily introduce a delay. For example, in the simple example in Figure 5.7, node 2 and its predecessor, node 3, are executed by two different processor. Node 2 is done at time t equal to 1. Node 3, however, can not be started before time t equal to 10. A communication delay is, therefore, introduced only if this delay is grater than 8 time units.

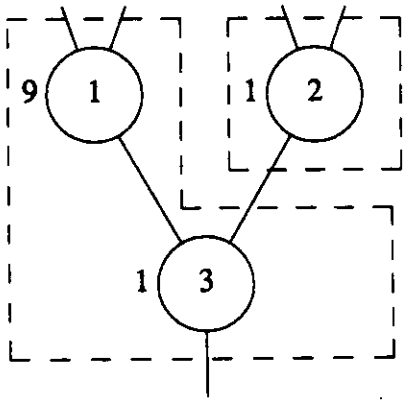


Figure 5.7 Interprocessor communication example

The basic scheduling strategy with a heuristic that minimizes the interprocessor communications is applied to the graph to allocate it among the processors:

1) The longest path of the graph after all artificial precedences are added is equal to execution time of the graph. Therefore, each disjoint critical path is assigned to one CPU which is dedicated to execution of that critical path.

2) If a node has only one predecessor which is ready for execution immediately after its father node, the predecessor is assigned to the same CPU.

3) If the predecessor is not available for execution right away(because it needs more inputs from other parts of the graph), the CPU is added to the pool of the idle processors.

4) If a node has more than one predecessor which becomes ready for execution immediately after the father node, one of the predecessors is assigned to the same CPU, the rest is assigned to processors from the idle processor pool. (Since the number of active nodes was reduced to less or equal the number of CPUs by the previous algorithm, it is guaranteed that there will be enough idle CPU to execute all available nodes immediately). This is the only case that requires interprocessor communication at the run time.

CHAPTER 6

JET ENGINE SIMULATION

6.1 Introduction

Real-time simulation is an effective tool for analyzing the behavior and interaction between components of a jet engine. It is useful in all different phases of design, test, and evaluation of a jet engine.

Analog computers were previously used for performing real-time simulation. They, however, suffer from drawbacks such as low accuracy, difficulty in generating multivariable functions, and large amount of equipment needed.

Hybrid computers divide the simulation task between their digital and analog portions. While the digital portion performs multivariable function generations, the analog portion performs other calculations such as integration. This method, although very effective to operate in real time, is still not totally desirable. Difficulty in handling analog components and several sources of errors including low accuracy of analog components and the fact that the digital update time appears as a time delay to the analog portion and can cause dynamic errors[SZUC 78] make all-digital simulators an attractive alternative.

The problem with all digital approaches is the limited speed of the processor. For real-time simulations that involve hardware-in-the-loop interactions, the frame time of computer must be short enough to interact with outside world and to be able to perform all necessary calculations in time. This requires high-speed computing capabilities. Usually a frequency range of up to 10 Hz is studies. Hence, the frame time should be less than 100 milliseconds.

With the advent of fast microprocessors, it is now possible to perform the simulation on a network of microprocessors and achieve high speeds required for real-time interaction with the outside hardware. A jet engine is usually modeled by several loosely coupled components. Each component is modeled separately, and they exchange the computational results at the end of each time frame.

The simulation of a jet engine is selected as the benchmark for ALI. In this chapter, jet engines are studied. Sections 6.2 and 6.3 review the basic concepts of jet engines. The rest of the chapter discusses the jet engine model provided by the NASA/LEWIS Research Center which is used as the benchmark.

6.2 Jet Propulsion

The basic principle of jet propulsion is neither new nor complicated. It is, in effect, the identical elementary force which imparts the energy to a toy balloon when it escapes one's fingers and flies off while deflating through its open stem. Jet engines develop thrust by accelerating a mass of gases produced by burning fuel with air or some other oxidizer.

The basic jet propulsion equation is obtained from the second law of motion stated by Newton in 1680. This law states that "the resultant force acting on a body is equal to the product of the mass times the acceleration of the body". This is represented by the following formula:

$$F = M*a$$

Where(in appropriate units):
F is force
M is mass
a is acceleration

When this formula is applied to a jet engine, it becomes:

$$F = \frac{w}{g} *(V_2 - V_1)$$

Where(in appropriate units):
F is force
w is flow rate of air, gas or fuel
V_1 is initial velocity of a mass of air, gas or fuel
V_2 is final velocity of a mass of air, gas or fuel
g is gravitational acceleration

The above formula, although simplified to a great extent, defines the main source of thrust in a jet engine. Another source of thrust is the pressure difference between the jet nozzle, where the gases are exhausted, and the ambient air. In practice, many other terms such as rates of air and fuel, leaks in the engine, temperature and density of air, etc. must also be considered.

6.3 Basic Components of a Jet Engine

A jet engine is like a large stovepipe(Figure 6.1). Huge quantities of air enter the engine at the air inlet in the front. This incoming air is compressed by a

compressor and is passed to a combustion chamber(combustor). The fuel is sprayed through nozzles into the front of the combustion chamber. The resulting mixture of fuel and air is burned to produce hot, expanding gases that enter a turbine. The high velocity of the gases entering the turbine causes the turbine to rotate. The power the turbine extracts from the gases is used to drive the compressor which is mounted on the same shaft. Roughly 75% of the power generated inside a jet engine is used to drive the compressor. Only what is left over is available to produce the thrust needed to propel the airplane. Finally, the exhaust gases are carried rearward through a short duct, called engine tail pipe, and are discharged through the jet nozzle which is the opening at the rear of this pipe.

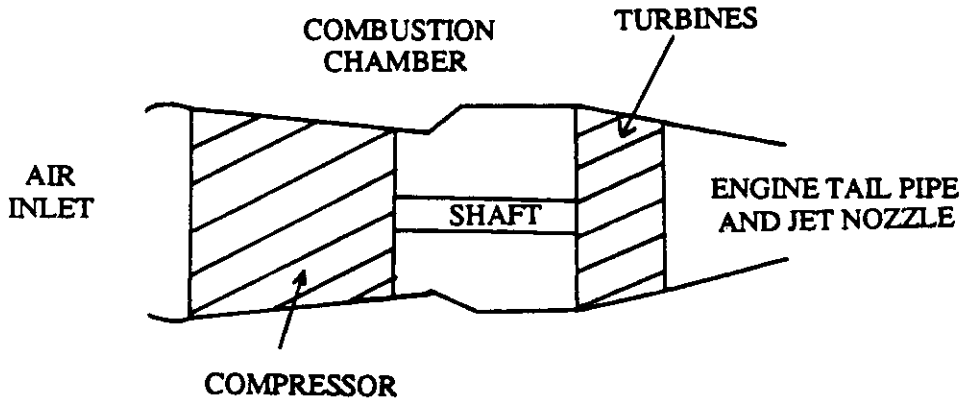


Figure 6.1 A simple turbojet engine

The jet engine shown in Figure 6.1 is that of a turbojet engine. There are other types of jet engine too. The propjet, or turbopropjet, is a version of the basic turbojet unit in which a conventional aircraft propeller is mounted on the central turbine shaft. The turbine actuates the compressor and, in addition, rotates the propeller blades. This arrangement is an attempt to combine the desirable features of

the turbojet power plant and the aircraft propeller(Figure 6.2). In this type of engine, most of the propulsive thrust comes from the propeller, and only a small portion comes from the exhaust nozzle.

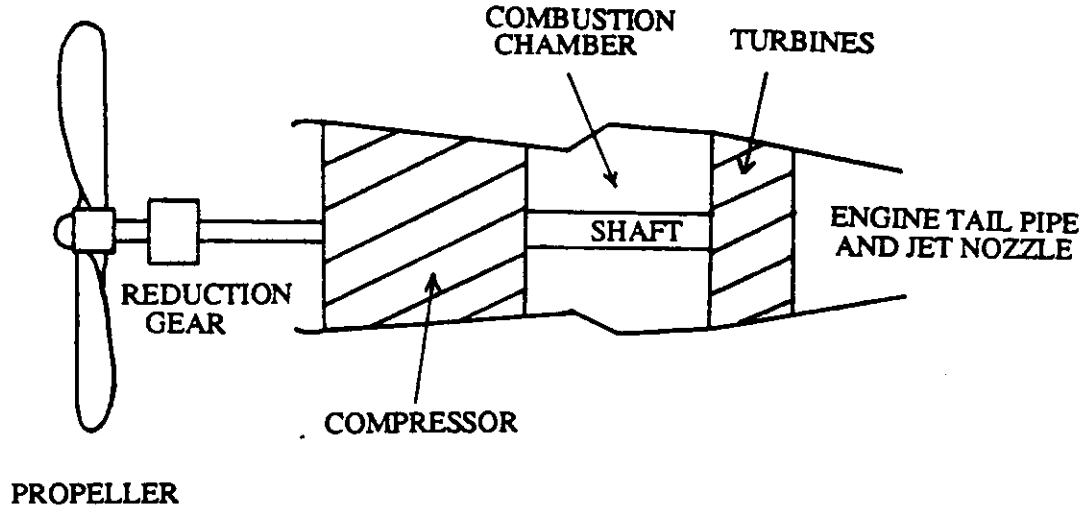


Figure 6.2 A turboprop engine

A variation of the turboprop engine, known as the turboshaft engine, is a gas turbine power plant in which all delivered power is in the form of shaft power that is used, through a transmission system, to operate something other than an aircraft propeller. Turboshaft engines are currently used to power helicopters and for various marine and land applications such as generating electricity and pumping natural gas through cross-country pipelines. The simulation of a turboshaft jet engine is used as the benchmark for ALI.

6.4 Engine Description

The model is provided by the NASA/LEWIS Research Center. The engine is a small, lightweight turboshaft helicopter engine of 1500-horsepower class[HART 84]. The engine has two turbines and one compressor. A gas generator turbine drives the compressor and a free-spinning power turbine delivers the power to the engine output shaft. The compressor has two parts. A five-stage axial part and a single-stage centrifugal part. In the axial part, a series of rotating blades and stationary vanes compress the air as it flows through the compressor in an axial direction. In the centrifugal part, the air entering through the center is rotated with an impeller. The air is carried to the perimeter of the impeller by centrifugal force. The air pressure is increased in a diffuser and is delivered to the combustor.

6.5 Model Description

The engine model is a sixth-order nonlinear system represented by five 1st-order ordinary differential equations and one first-order lag. The computational flow diagram of this engine model is shown in Figure 6.3. Major components of the engine such as the compressor, combustor, turbines, and rotors are shown. Arrows show the flow of information from one component to another. Intercomponent volumes are assumed at locations where gas dynamics are required. In these volumes, the storage of mass and energy occur.

The CSSL-IV program showing the model is shown in Figure 6.4. Initial conditions for state variables and other parameters are given as constants in the

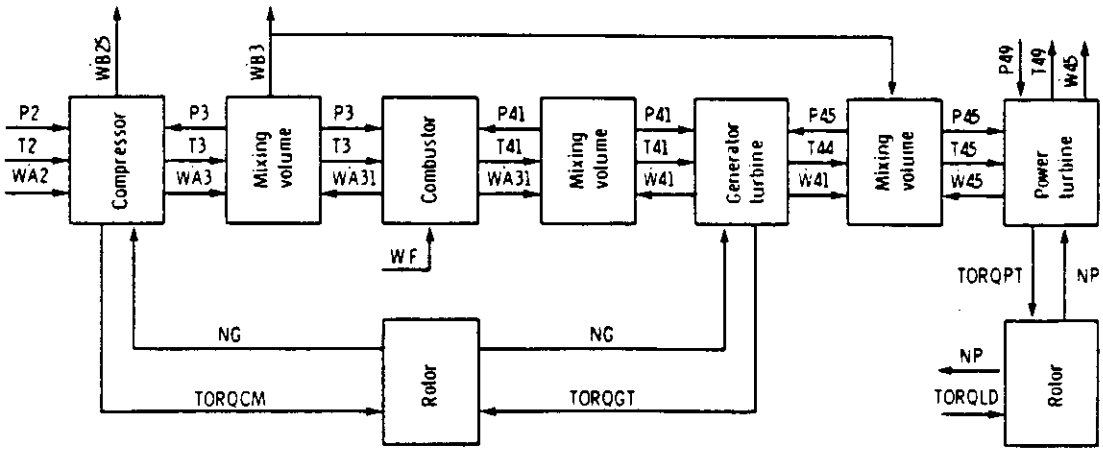


Figure 6.3 Block diagram of small turboshaft engine

INITIAL region. The DERIVATIVE section embedded in the DYNAMIC region contains the model definition statements. The data flow graph for this program is shown in Figure 6.5.

The compressor, combustor, and turbines are static elements and are modeled by a combination of algebraic and tabular data. There are seven functions of one variable and one function of two variables expressed in tabular form. These tables are build using the manufacturer's engine performance data developed during the engine design phase, as well as engine experimental test data.

Rotors and intercomponent volumes are dynamic elements. Six state variables are associated to dynamic components. Rotor dynamics are represented by the equations of conservation of angular momentum. There are two rotors: one between the compressor and the gas generator turbine, and the other between the power turbine and the load. Two state variables represent the rotational speeds of

these rotors.

Volume dynamics are represented by the equations of conservation of mass and energy. Three state variables represent the total stored air mass at the compressor outlet, the pressure at the combustor outlet, and the interturbine pressure. The temperature at the compressor outlet is represented by a first-order lag equation.

```
program small turbo-shaft engine simulation
constant igt =.0445, ipt=.417, kv41=6.17, kv45=13.63, kwgtrb=.0876
constant hvf=18300., ra=640.2,t3lg=.02, v3=658.82, p2=14.620, t2=538.32
constant p49=15.539, torqld=26.26, wfph=200.06, icng=36155., icnp=20000.
constant icp41=81.31, icp45=21.27,ict3=992.5,icws3=.0898, kdpb=.034
constant effb=.985,chf1=184.3
table ypngc 1, 11, 65.,80.,82.,85.,87.,89.,92.,94.,96.,98.,100.
table xprc 1, 77, 1.0,3.15,3.25,3.40,3.52,3.65,3.75,  1.0,5.61,5.83,...
            1.0,7.4,7.74,8.0,8.25,8.7, 1.0,8.4,8.9,9.3,9.6,9.9,...
            10.1, 1.0,9.5,10.1,10.6,11.0,11.3,11.5, 1.0,11.4,11.9,...
            12.3,12.7,13.0,13.2, 1.0,12.8,13.3,13.7,14.1,14.4,14.6,...
            1.0,14.4,14.9,15.3,15.7,16.0,16.2, 1.0,15.8,16.3,16.7,17.1,...
            17.4,17.6, 1.0,16.7,17.2,17.6,18.0,18.3,18.5
table zw2c 1, 77, 3.14,3.14,3.13,3.12,3.10,3.05,2.97, 4.76,4.76,...
            4.75,4.74,4.7,4.64,4.49, 5.28,5.28,5.26,5.22,5.16,5.06,4.92,...
            5.87,5.87,5.84,5.80,5.74,5.64,5.50, 6.28,6.28,6.26,6.23,6.17,...
            6.09,5.99,  6.73,6.73,6.70,6.63,6.54,6.42,6.31,  7.65,7.65,...
            7.63,7.58,7.51,7.42,7.30, 8.28,8.28,8.26,8.22,8.15,8.06,7.95,...
            8.99,8.99,8.97,8.93,8.87,8.67,  9.65,9.65,9.63,9.59,9.53,...
            9.45,9.32,  10.1,10.1,10.08,10.04,9.98,9.90,9.80,
table xpngc 1, 13, 65.,78.6,80.,81.,82.,83.,84.,85.,86.,87.,88.,88.4,100.
table zb1 1,13,.109,.109,.1062,.1025,.0978,.0895,.074,.05,.031,.0146,...
0.0038,0.0,0.0
table xprc1 1, 9, 1.,5.,8.,10.,12.,14.,16.,18.,20.
table ztrc 1, 9, 1.3,1.8,2.01,2.15,2.26,2.38,2.49,2.65,2.80
table xw2c, 1, 4, 3.0,8.3,10.0,12.0
table zwxq2 1, 4, .0851,.0846,.0815,.779
table xw2cb 1, 4, 0.0,3.0,4.98,12.0
table zb2 1, 4, .01057, .01057,.0090,.0090
table xprgt 1, 10, .210,.214,.2141,.215,.2195,.224,.23,.25,.28,.35
table zdhgtq 1, 10, 43.,41.,40.8,40.42,39.24,38.52,37.6,35.5,32.3,24.6
table xprpt 1, 10, .30,.35,.40,.45,.50,.55,.60,.65,.70,.80
table zdhptq 1, 10, 34.0,30.3,26.7,23.2,20.0,17.2,14.3,11.3,8.3,2.0
table zw45c 1, 10, .372,.372,.3705,.367,.3625,.357,.351,.340,.325,.285
 initial
  f1 = funset(40)
  del2 = p2/14.696
  kdhb = chfl + effb * hvf
  rtth2 = sqrt(t2/518.67)
  khpt = 30./3.1415926*550.
  wf=wfph/3600.
  algorithm istart=6, irun=7
  cinterval = .20
  nisteps nist = 3; nsteps nst = 1000
 end initial
```

Figure 6.4 CSSL-IV program to simulate a turboshaft jet engine
(continued on the next page)

```
dynamic
 derivative
   pcngc = ngc / 447.
   ps3 = .956 * p3
   ps3q2 = ps3 / p2
   wa2c = mapfun(1, 7, 11, xprc, ypngc, zw2c, ps3q2, pcngc)
   wa2 = wa2c * del2 / rtth2
   b1 = fun1(2, 13, xpngc, zb1, pcngc)
   b2 = fun1(8, 4, xw2cb, zb2, wa2c)
   wb25 = (b1 + b2) * wa2
   wa3 = wa2 - wb25
   t3q2 = fun1(3, 9, xprc1, ztrc, ps3q2)
   t3c = t3q2 * t2
   t25q2 = 1.15 + .039 * ps3q2
   t25 = t25q2 * t2
   t3 = realpl(ict3, t3lg, t3c)
   wxq2 = fun1(4, 4, xw2c, zwxq2, wa2c)
   wb3 = (wxq2 + .0025) * wa2
   ws3dt = wa3 - wb3 - wa31
   ws3 = integ(icws3, ws3dt)
   krwqv3 = ra * ws3 / v3
   wa31 = sqrt(krwqv3 /kdpb * (p3 - p41))
   h2 = .239 * t2
   h25 = .240 * t25
   h3 = .2496 * t3 - 8.4
   p3 = krwqv3 * t3
comment combustor outlet pressure and temperature
   p41dt = kv41 * t41 * (wa31 + wf - w41)
   p41 = integ(icp41, p41dt)
   far41 = wf / wa31
   h41 = (h3 + kdhb * far41) / (1. + far41)
   t41 = 3.298 * h41 + 308.
comment gas generator turbine enthalpy drop and flow
   thta41 = 1.8326e-3 * t41 + .0856
   pr45q1 = p45 / p41
   dhqth4 = fun1(5, 10, xprgt, zdhgtq, pr45q1)
   dh41 = dhqth4 * thta41
   h44 = h41 - dh41
   w41 = kwgtrb * p41 / sqrt(thta41)
```

Figure 6.4 (Continued)

```
comment compressor and g.g. turbine torques and speed
    torqc = 7429.35 / ng * (h3 * wa3 + h25 * wb25 - h2 * wa2)
    torq41 = 7429.35 * w41 / ng * dh41
    ngdt = 9.5493 / igt * (torq41 - torqc)
    ng = integ(icng, ngdt)
    ngc = ng / rtth2
comment power turbine enthalpy drop and flow
    h45 = .9623 * h44
    t45 = 3.537 * h45 + 174.5
    thta45 = 1.8326e-3 * t45 + .0856
    dhqth5 = fun1(6, 10, xprpt, zdhptq, pr49q4)
    pr49q4 = p49 / p45
    dh45 = dhqth5 * thta45
    h49 = h45 - dh45
    t49 = 3.549 * h49 + 160.1
    w45c = fun1(7, 10, xprpt, zw45c, pr49q4)
    w45 = w45c * p45/sqrt(thta45)
comment inter-turbine pressure
    p45dt = kv45 * t45 * (w41 - w45 + .7826 * wxq2 * wa2)
    p45 = integ(icp45, p45dt)
comment power turbine torque and speed
    torq45 = 7429.35 * w45 / np * dh45
    npdt = 9.5493 / ipt * (torq45 - torqld)
    np = integ(icnp, npdt)
    hpout = np / khpt * torq45
    wfqps3 = wfph / ps3
  end derivative
  termt(t .gt. 10.0)
 end dynamic
 terminal
 end terminal

end program
```

Figure 6.4 (Continued)

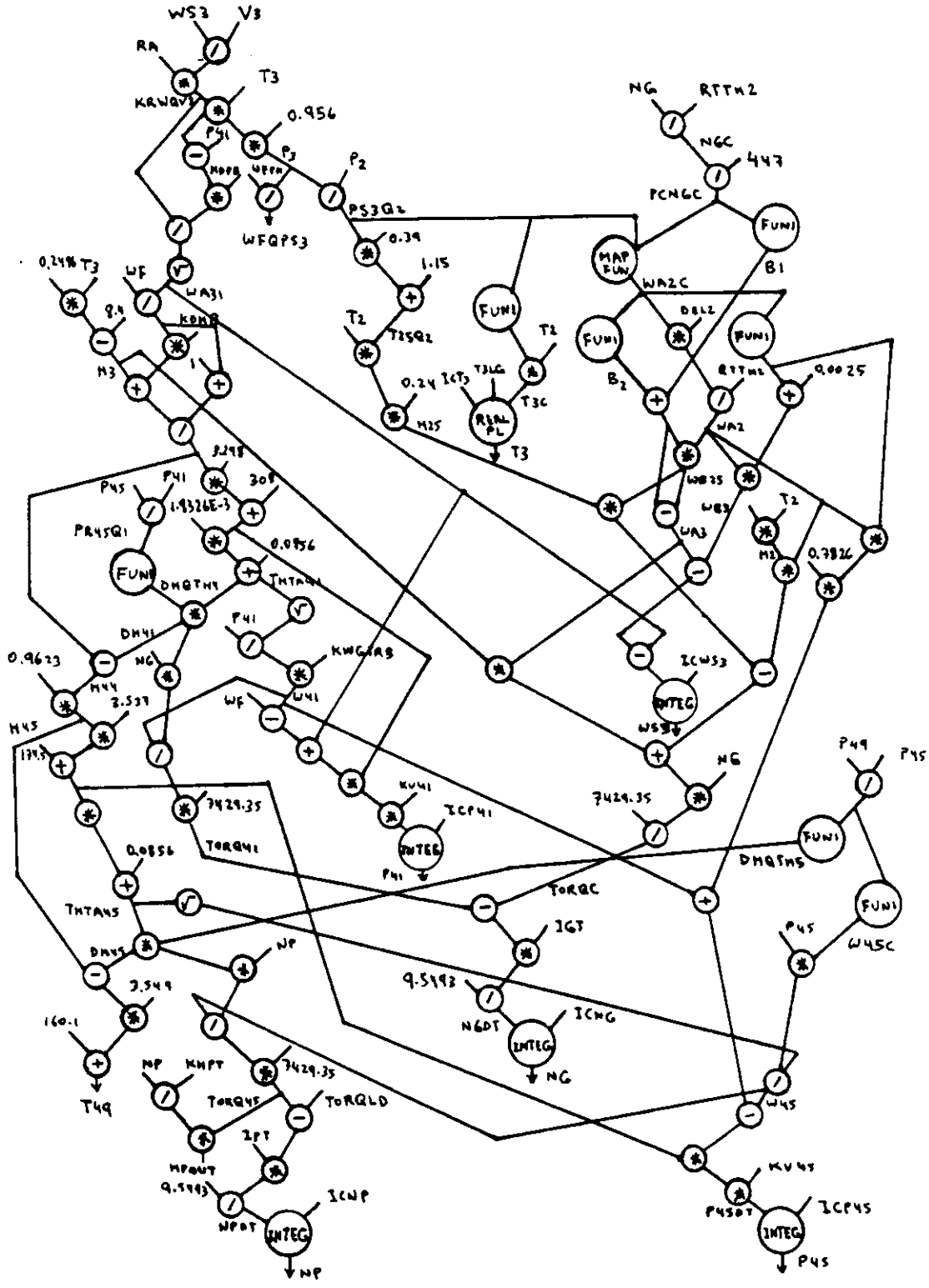


Figure 6.5 Data flow graph of the turboshaft jet engine

CHAPTER 7

FUNCTION GENERATION

7.1 Introduction

In order for the prescheduler to analyze a program prior to run, the initiation and duration of all tasks must be known a priori. Determining these values becomes very difficult if the program contains iteration loops or conditional branching. There are methods for unfolding loops with known number of iterations[RUSS 69]. However, if the number of iterations must be resolved at run time or if the program contains conditional branches of different lengths, it will not be possible to exactly determine the completion time of the program. To estimate the time, one must assign probabilities to each branch of a conditional statement[MART 69], as well as to the number of iterations of a loop.

As mentioned before, ALI does not support stochastic models. Therefore, if there is a loop or a conditional branch in the program, the user must define a function to contain them. When the simulation starts, the user is prompted to enter the estimated time of those functions as well as all other CSSL or user defined functions. For some functions, however, a default execution time is provided, and the users have the choice of either using those defaults or use their own estimated values. These functions are described in the rest of this chapter. The way the users can

accept the defaults or use their own values is described in the User Manual in Appendix A.

7.2 Function Generation Methods

There are two widely-used ways in which a function can be generated: the Taylor series approximations and table lookup methods. In Taylor series approximations, if a function f(x) possesses continuous derivatives of all orders around a point x_0, then the Taylor series expansion for the function around the point x_0 is expressed by the following power series:

$$f(x) = \sum_{n=1}^{i=\infty} \frac{f^{(n)}(x_0)}{n!} (x - x_0)^n$$

The value of the function at a neighboring point,x_1, can be found by substituting the value of x_1 for x, calculating the first n terms of the Taylor series, and ignoring all other higher-order terms. These ignored higher-order terms determine the truncation error. The Taylor series approximations is straightforward and provides an easy way of controlling the truncation error. However, for all but very simple functions, calculation of the higher-order derivatives is very difficult and sometimes impossible.

Besides, in many practical problems, functions can not be expressed by a formula and only their experimental values are available in tabular forms. Therefore, table lookup methods are universally used for function generation on digital computers.

In these methods, a function is represented in a tabular form, listing the values of the function for a range of input values. Whenever the value of the function is needed for a specific input value, the table is searched and if an exact match is not found, interpolation or extrapolation methods are used to find an approximate value for the function. In simulation problems, most of the execution time is spent for function generation. Table lookup methods, therefore, play an important role in the execution of these problems.

7.3 Table Lookup Methods

The functions used in simulation problems are mostly in one- or two-dimensional forms. In one-dimensional form, for specific values of x, the corresponding values of f(x) are tabulated. The values of the function can be found by a simple interpolation(Figure 7.1).

This interpolation can be either linear or of a higher-order schema. The linear interpolation of f(x) for the interval x_i to x_{i+1} is:

$$\hat{f}(x) = f(x_i) + \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$$

In two-dimensional form, a function is represented by a family of parallel curves. If the values of the family of curves is expressed for the same set of values of x, a two-dimensional interpolation can be made in the directions parallel to the coordination axes(Figure 7.2).

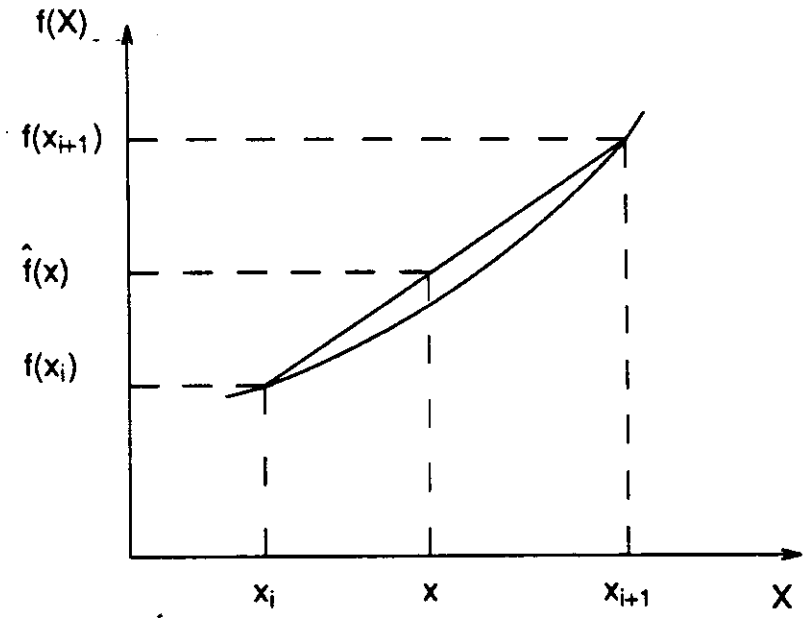


Figure 7.1 Function of one variable

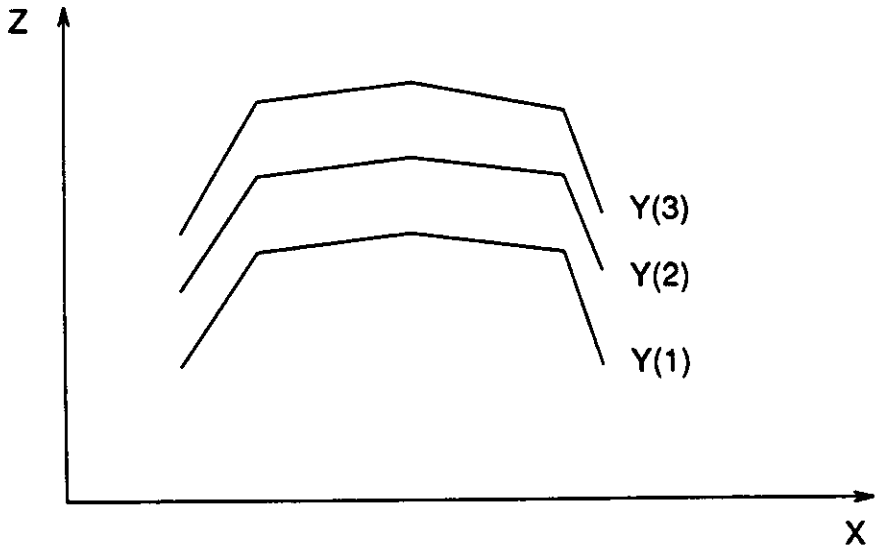


Figure 7.2 Regular type function of two variable

If the values of the family of curves are expressed for different set of values of x, a two-dimensional interpolation in directions not parallel to the coordination axes is required(Figure 7.3). This type of two-dimensional function is frequently used to represent the component performance data of aeronautical systems such as jet engines and is referred to as a map function.

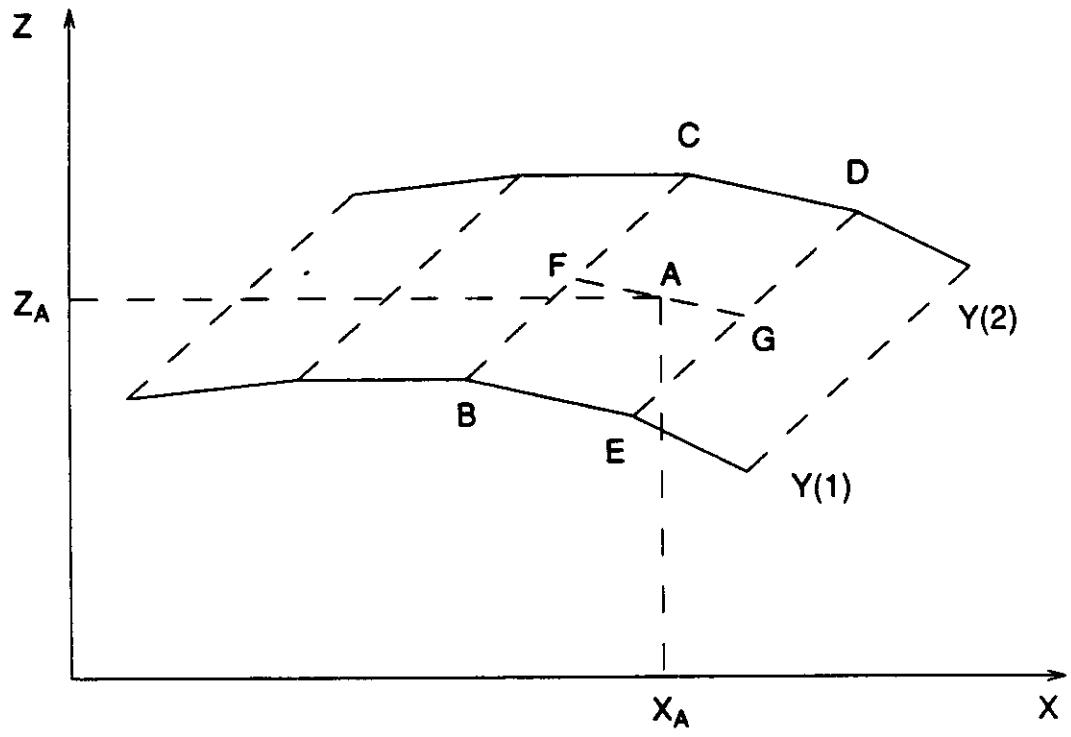


Figure 7.3 Interpolation method for map-type function of two variables

A majority of functions used in simulation can be generated by either one-dimensional or map-type two-dimensional table lookups. Therefore, default values for these two type of table lookups are provided in ALI.

7.4 Default Execution Times

ALI is capable of accepting all standard CSSL operators as well as any user defined functions. However, since the system is designed to work for any microprocessor, the user must know the execution times of all simple operators and will be prompted to enter the estimated times of these operators. Furthermore, execution times of more complex operators and functions, such as integration or table lookups, depends on the method used to perform them. Therefore, an advanced user must know the execution times of all simple and complex operators.

However, for the casual user, default execution times of some widely used operators and functions are provided. The execution times of simple operators are found from the Motorola MC68000 User's Manual[MOTO 82]. Note that even for the same type of microprocessor, the execution times depend on the external clock rate. Therefore, execution times are usually given in terms of external clock periods rather than the actual millisecond figures. Furthermore, the number of clock periods for each instruction depends on the operand size(byte, word, long) and the addressing mode(register direct, memory direct, etc.). The execution times used in ALI are found by averaging the values for different operand sizes and operation modes.

These execution times are then normalized by dividing all execution times to the execution time of the addition operation and are rounded to the closest whole integer. This makes all nodes of the graph mutually commensurable as required by the prescheduler.

The algorithms to calculate the default values for more complex operators are given in the following sections. The actual values used for those defaults are given in Appendix A.

7.4.1 One-Dimensional Table Lookup

In simulation problems, successive function references are usually made for consecutive values of the independent variable. An easy way of speeding up the search is, therefore, to save a pointer to the last entry referenced in the table, so that the next table lookup can start from that point rather than from the beginning of the table. Each table is assigned a number and a separate table is used to store the search pointers for all tables(Figure 7.4).

A new search starts from the scan interval were the previous value was found. Usually the new value is either in the same interval(no need to search up or down) or in the next few higher or lower intervals. In ALI, it is assumed that table searches need an average of one interval change(either search up or search down).

The one-dimensional table lookup operator used in ALI is called FUN1[HART 78]. This operator is defined as:

FUN1(tblnum, maxpt, xx, zz, xin)

Where
tblnum is the table number
maxpt is the total number of points
xx and zz are arrays containing x and f(x) values respectively
xin is the input value

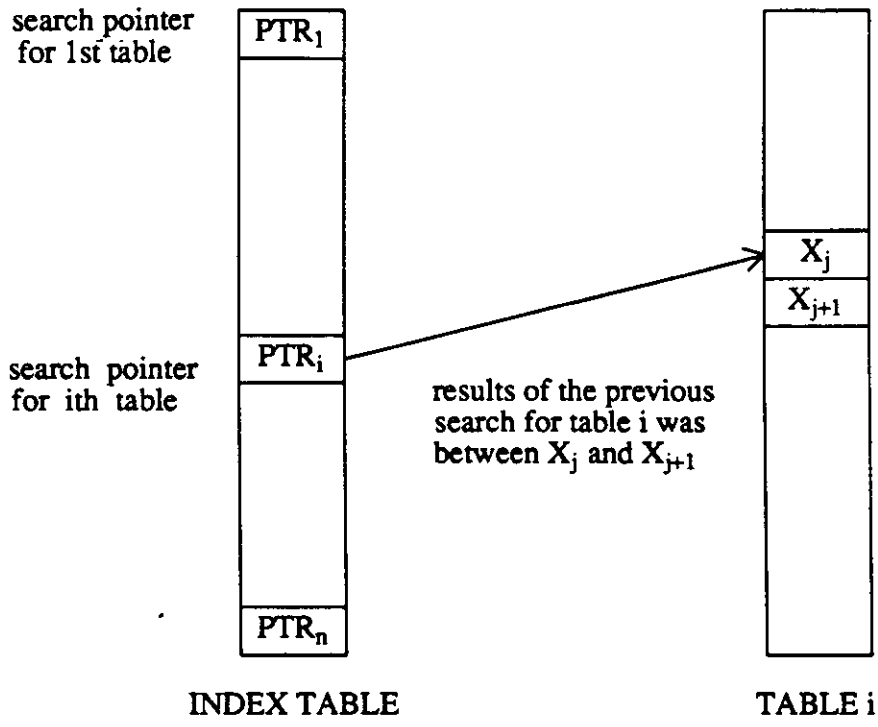


Figure 7.4 Index table

Note that in order to support unequally spaced data points, both x and f(x) values are stored. The algorithm for implementing this operator is shown in Figure 7.5. The function contains 1 multiplication, 1 division, and some 20 to 25 add/subtract/index operations. The default execution time for this function is given in Appendix A.

7.4.2 Map-Type Two-Dimensional Table Lookup

Similar to the FUN1 operator, the two-dimensional table lookup operator for the map-type functions, MAPFUN[HART 78], uses search pointers to speedup the search. This operator supports unequally spaced data points. Therefore, the values

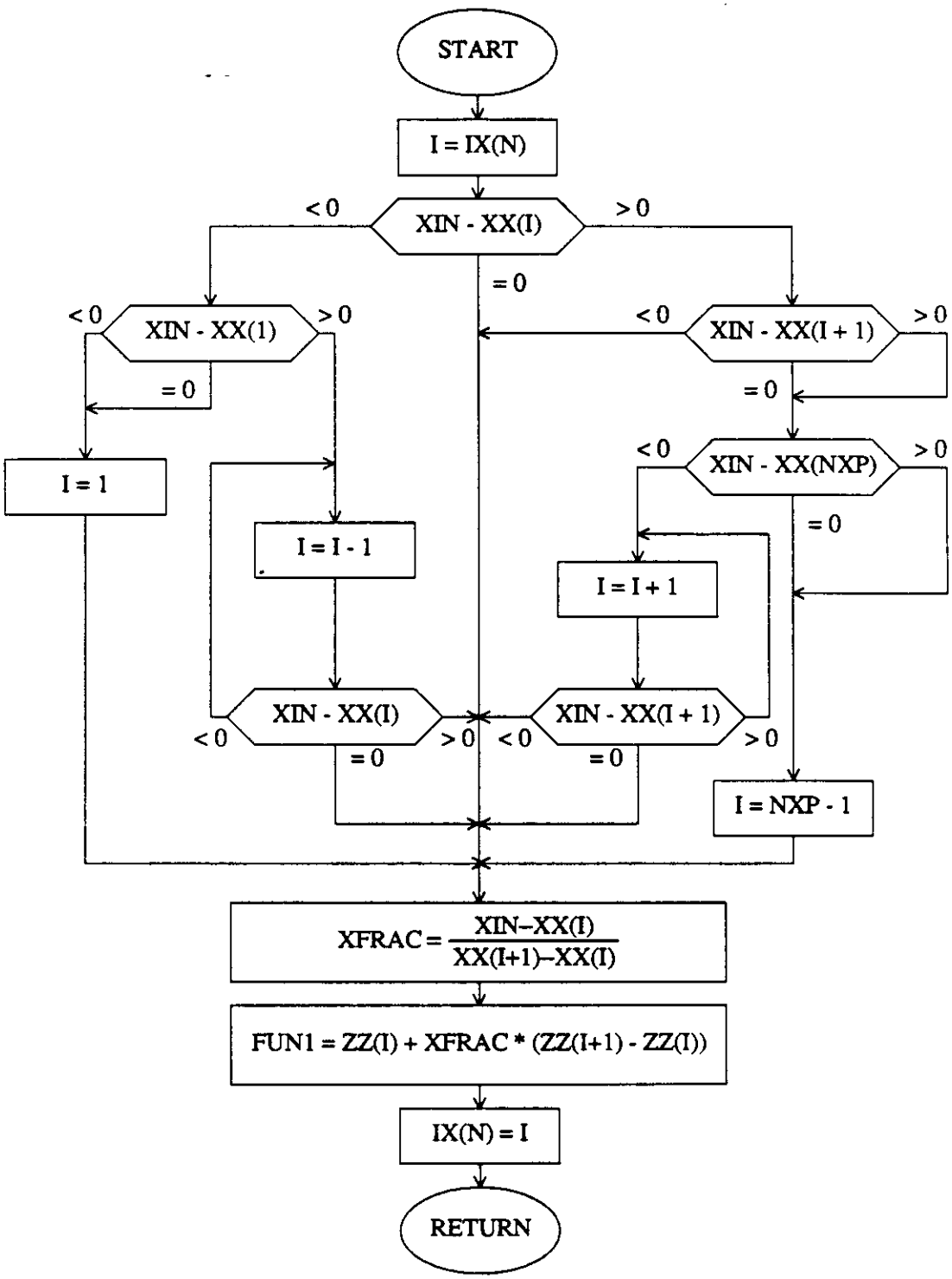


Figure 7.5 FUN1 algorithm

for x, y, and f(x,y) are stored. In order to make the search faster, three separate one-dimensional tables are used to store these data. The MAPFUN operator is defined as:

MAPFUN(tblnum, maxpt, ncrv, xx, yy, zz, xin, yin)

Where
tblnum is the table number
maxpt is the total number of points
ncrv is the number of curves
xx, yy, and zz are arrays containing x, y,
and f(x,y) values respectively
xin and yin are the input values

In order to find the value of the function f(x,y), a search is first made in the y direction to find the two curves between which the point lies(similar to a one-dimensional search). Next, the value of the x input is compared with table entries to find the interval in which the x input lies[HART 78]. Finally, by interpolating along the sides of the parallelogram made by the two x and y intervals, the value of the function for the desired point is found(Figure 7.3).

The function includes 2 divisions, 7 multiplications, and some 60 to 75 add/subtract/index operations. The default execution time for this operator is given in Appendix A.

7.4.3 FORTRAN Library Functions

The default execution times for the following FORTRAN library functions are provided in ALI:

Function	Description
ACOS	arc cosine
ALOG	natural logarithm
ALOG10	common logarithm
ASIN	arc sine
ATAN	arc tangent
COS	cosine
EXP	exponential
SIN	sine
SQRT	square root
TAN	tangent
TANH	hyperbolic tangent

All these functions are implemented by a one-dimensional table lookup. Therefore, the default execution times for all these functions are the same as the FUN1 operator.

7.4.4 Exponentiation

The exponentiation operator, **, can be implemented by 1 multiplication and 2 one-dimensional table lookups as follows: The equation

$$y = x^n$$

can be rewritten as

$$\log(y) = \log(x^n) = n \log(x)$$

Therefore,

$$y = \log^{-1}(n \log(x))$$

The default execution time of the exponentiation operator is given in Appendix A.

7.5 Integration Algorithm

There are several operators, such as integration(INTEG) and first-order lag(REALPL), that require calculation of the state variables. Each state variable is defined by an ordinary differential equation(ODE) with given initial conditions. An integration algorithm is, therefore, required to find the value of the state variable at each time step. In order to explain the criteria for selecting a suitable integration algorithm, some background in ODEs and different methods of their solution is useful.

7.5.1 Basics

An ODE, when expressed mathematically, takes the form

$$f(x, y, y^{(1)}, \cdots, y^{(n)}) = 0$$

which specifies a relation between an independent variable, x, a dependent variable, y, and the derivatives of this dependent variable.

In order to find a unique solution to this equation, a set of initial conditions is also necessary. If the initial conditions are specified at a single point, the problem is called an initial value problem. On the other hand, if the initial conditions are shared between two or more points, the problem is referred to as a boundary value problem.

The order of a differential equation is the order of the highest order derivative entering into the equation. Much work has been done in solving first-order ODEs. Therefore, it is a common practice to convert a set of higher-order ODEs to a set of

first-order ODEs. This conversion can be done by a simple procedure[BEKE 68].

A system of ODEs can either be stiff or nonstiff. Stiff systems contain both rapidly and slowly varying components with eigenvalues differing by at least 2 orders of magnitude. Stiff problems are generally treated differently and are not considered here.

Furthermore, initial value problems are more popular and widely used. Therefore, only the solution of a set of first-order nonstiff initial value ODEs is considered in ALI.

This type of equation can be expressed as:

$$y' = f(x, y)$$
$$y(0) = Y_0$$

Where
x is the independent variable
y is the dependent variable
Y_0 is the initial conditions
note that y and Y_0 can be vectors

The solution for this equation can be found either by analytical or by numerical techniques. Analytical methods are not powerful enough to solve general ODE problems and their usefulness is limited to very simple cases. Therefore, in almost all practical situations, numerical methods are employed to find approximate solutions to the ODEs.

In these methods, the range of the independent variable is divided into a set of separate points. Depending on whether the distance between those points is fixed

or not, the problem is referred to as having a fixed or a variable step size.

The solution at each point is found by using the known values of the function and its derivatives at some previous points. If only the known values from the last point is needed, the algorithm is known as self-starting. On the other hand, if the results from several previous points are needed, the algorithm is known as non-self-starting. At the beginning, only the initial conditions are specified. Therefore, non-self-starting algorithms, as their name imply, require a self-starting method to obtain the results for the first few steps.

7.5.2 Self-starting Methods

The method of Taylor series expansion is a good example of a self-starting method. In this method, if all higher-order derivatives of y exist at a point x_i, the function y(x) can be approximated near x_i by the first few terms of the Taylor series:

$$y(x) = y(x_i) + y^{(1)}(x_i)(x-x_i) + \cdots + \frac{y^{(n)}(x_i)}{n!}(x-x_i)^{(n)} + \cdots$$

This method is self-starting and is excellent when feasible. However, since it requires calculation of the higher-order derivatives of y, it has not benn used as a general purpose algorithm. Halin[HALI 83] has recently developed a very efficient Taylor series method using symbolic method for determining the higher derivatives. In this way the problems of finding numerical differentiation are avoided, and relatively large step sizes can be used without incurring large truncation errors. The problem with Halin's method is its large storage requirements.

Another self-starting method, known as the Euler's method, ignores second- and higher-order terms of the Taylor series. The solution at each step is:

$$y_{n+1} = y_n + h\, y'_n$$

Where
h is the step size
y_{n+1} is the solution at step n+1

This method is simple but introduces large truncation errors. In order to reduce the truncation error, the step size must be kept small which, in turn, increases the computation time and the round-off errors. The Euler's method is limited to problems with low accuracy requirements in which the solution can be found with a few number of steps.

The most widely used self-starting methods are the Runge-Kutta formulas. These formulas, instead of evaluating the higher-order derivatives, evaluate the first derivative of the function at several points in the vicinity of a given point. A weighted sum of these values is then used to approximate the dependent variable at that point. Thereby, matching the first few terms of the Taylor series. These methods have the advantage of being self-starting. On the other hand they do not provide a simple measure of the truncation error. Furthermore, if the derivative function is complicated, several evaluation of it can be very time consuming.

7.5.3 Non-self-starting Methods

Non-self-starting methods are based on the fact that any continuous function can be arbitrarily closely approximated by a polynomial of a sufficiently large

degree. The value of the independent variable at each step is found by an equation of the type

$$y_{n+1} = a_1\, y_n + a_n\, y_{n-1} + \cdots + a_k\, y_{n+1-k}$$

$$+ h\,(b_0\, y'_{n+1} + b_1\, y'_n + \cdots + b_k\, y'_{n+1-k})$$

where
h is the step size
a and b are constants

Note that except for the derivative at step n+1, all values on the right hand side of the equation are known at step n+1. If the coefficients are selected in a way that b0 is set to zero, the resulting equation is called an explicit or closed-type formula. If, on the other hand, b0 is nonzero, the resulting formula is called an implicit or open-type formula. Since the value of the derivative is not known for step n+1, implicit formulas require iterative solutions.

In order to determine the values of the coefficients, a finite polynomial with sufficient degree is embedded in the equation to yield an exact solution to the equation. Thereby, exact values for the coefficients are found. In the method of undetermined coefficients, a lower order polynomial is used such that some of the coefficients are found and some others are left free or unspecified. These free coefficients can then be selected in a way to make the calculations easier or to reduce the truncation errors.

7.5.4 Noniterative Multistep Methods

One way of selecting the free coefficients is to set

$$a_2 = \cdots = a_n = 0$$

This results in the widely used Adams formulas. If in addition b0 is set to zero, the result will be the famous Adams-Bashforth(A-B) equations. Therefore, in A-B method, the independent variable at step n+1 is defined as:

$$y_{n+1} = a_1\, y_n + h\,(b_1\, y'_n + \cdots + b_k\, y'_{n+1-k})$$

K is usually called the order of the formula.

The A-B formulas of the first- to fifth-order are:

$$y_{n+1} = y_n + h\, y'_n$$

$$y_{n+1} = y_n + h/2\,(3y'_n - y'_{n-1})$$

$$y_{n+1} = y_n + h/12\,(23y'_n - 16y'_{n-1} + 5y'_{n-2})$$

$$y_{n+1} = y_n + h/24\,(55y'_n - 59y'_{n-1} + 37y'_{n-2} - 9y'_{n-3})$$

$$y_{n+1} = y_n + h/720\,(1901y'_n - 2774y'_{n-1} + 2616y'_{n-2} - 1274y'_{n-3} + 251y'_{n-4})$$

If the coefficients are selected such that

$$a_1 = a_3 = \cdots = a_n = 0 \quad \text{and} \quad a_2 \neq 0$$

the result will be the Noystrom formula. The Noystrom formulas for the first- to fifth-order are given below.

$$y_{n+1} = y_{n-1} + 2h\, y'_n$$

$$y_{n+1} = y_{n-1} + 2h\, y'_n$$

$$y_{n+1} = y_{n-1} + h/3\,(7y'_n - 2y'_{n-1} + y'_{n-2})$$

$$y_{n+1} = y_{n-1} + h/3\,(8y'_n - 5y'_{n-1} + 4y'_{n-2} - y'_{n-3})$$

$$y_{n+1} = y_{n-1} + h/90\,(269y'_n - 266y'_{n-1} + 294y'_{n-2} - 146y'_{n-3} + 29y'_{n-4})$$

Another widely used family of formulas is the Newton-Cotes open end formulas. The first- , third- , and fifth-order formulas are given below.

$$y_{n+1} = y_{n-1} + (2h)\, y'_n$$

$$y_{n+1} = y_{n-3} + (4h/3)\,(2y'_n - y'_{n-1} + 2y'_{n-2})$$

$$y_{n+1} = y_{n-5} + (3h/10)\,(11y'_n - 14y'_{n-1} + 26y'_{n-2} - 14y'_{i-3} + 11y'_{i-4})$$

7.5.5 Iterative Multistep Methods

If the coefficient b0 is nonzero, the resulting formula will require an iterative solution but will be more stable. If

$$a_2 = \cdots = a_n = 0 \quad \text{and} \quad b_0 \neq 0$$

the results will be the Adams-Moulton(A-M) formulas. For example, the third-order A-M formula is:

$$y_{n+1} = y_n + h/12(5y'_{n+1} + 8y'_n - y'_{n-1})$$

In order to find the value of the independent variable at step n+1, the value of its derivative for that step must be found first. A variety of predictor-corrector and predictor-modifier-corrector methods can be employed. Bekey and Karplus discuss the algorithms for implementing these methods and present the methods for controlling the truncation error and changing the step size[BEKE 68].

7.5.6 Selection of the Default Algorithm

The selection of a suitable algorithm to solve a given initial value problem depends, to a large extent, on the nature of the problem. Therefore, it largely depends on the user to decide which method to choose. Usually a Runge-Kutta method is selected as the default algorithm.

However, for real-time applications, several calculations of the derivative function at each step are not desirable. The same argument is true for the implicit non-self-starting methods, because these methods require iterations. Therefore, noniterative methods are more appropriate for ALI.

Noniterative multistep formulas are very popular and offer reasonably reliable results. The coefficients of these formulas are integers that grow larger as the order of the formula increases. Since a large coefficient contributes to the round-off errors, formulas of larger orders than five is seldom used. Furthermore, formulas of even-order are not more accurate than the lower odd-order formulas[HILD 56].

Therefore, the odd-order formulas of fifth order or less are widely used.

7.5.7 Implementation

The integration effort at each time step involves the calculation of the derivative function, the calculation of the state variable at that time step, and the overhead for saving the results for later steps. In ALI, the calculation of the derivative function is included in the program graph; al other operations are lumped together and shown by the INTEG operator. Therefore, the default execution time of the INTEG operator given in Appendix A does not include the calculation of the derivative function which is treated the same as the rest of the graph. Since the INTEG operator in ALI does not represent the derivative calculation, those user defined INTEG algorithms are supported that require only one derivative calculation. When the user is prompted to enter the execution time for the INTEG operator, only the time to calculate the state variable and the overhead time to store the results must be entered.

The overhead time for the third order formulas includes calculation of y_{n+1} at each step which is generally in the form of

$$y_{n+1} = y_m + C_1\, h\, (C_2 y'_n + C_3\, y'_{n-1} + C_4\, y'_{n-2})$$

since c_1 to c_4 are all known parameters, the formula can be simplified as:

$$y_{n+1} = y_m + a_1 y'_n + a_2\, y'_{n-1} + a_3 y'_{n-2}$$

Calculation of this formula requires three multiplications, three addition/subtractions, and storing and updating past values. In a similar way, the fifth-order formulas require five multiplications and five addition/subtractions. The third-order Newton-Cotes formula, however, has an interesting property: The term inside the parentheses involves two multiplication by two for y'_n and y'_{n-2} which can be implemented by a shift-left operation which is much faster than multiplication. And y'_{n-1} does not involve a multiplication.

Therefore, the term inside the parentheses can be found without multiplications. In this way y can be found by only one multiplication and five addition/subtraction/shift operations. The third-order Newton-Cotes formula is, therefore, the fastest formula and is selected as the default algorithm for ALI.

The formula requires seven storage locations for saving the results of previous steps. Saving and updating these results at the end of each time step is implemented by first moving the results of each step to the next previous one and then saving the results of the present step(Figure 7.6). The value of the default execution time for the Newton-Cotes formula is given in Appendix A.

7.5.8 REALPL Operator

The first-order lead-lag, REALPL, is implemented as:

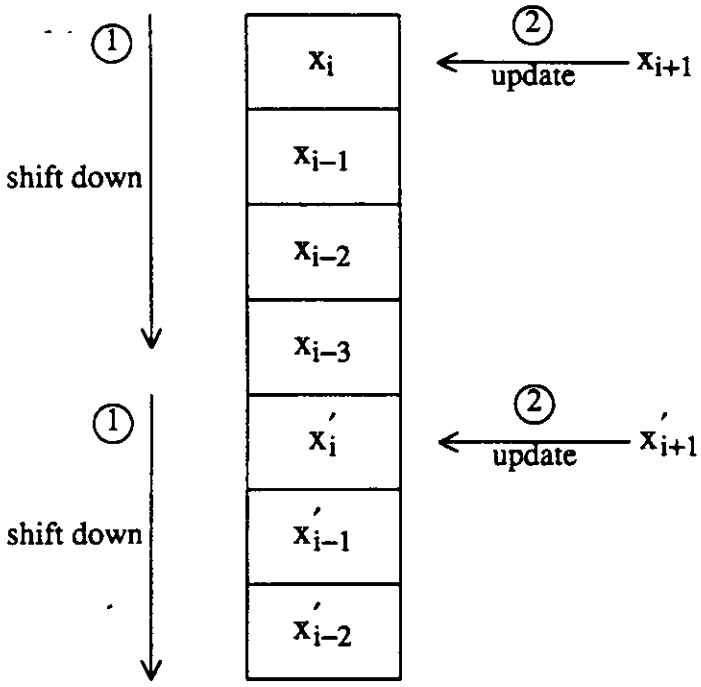


Figure 7.6 Updating stored values for the integration algorithm

$$y = \text{REALPL}(p, x, ic) = \text{INTEG}(\frac{x-y}{p}, ic)$$

Therefore, its execution time equals to the execution time of one INTEG and one division operations.

7.6 Iteration Loops and Conditional Branches

ALI does not support stochastic models. Therefore, if there is a loop or a conditional branch in the program, the user must define a function to contain them. However, as mentioned in Chapter 2, some CSSL operators, such as SWIN, act as a conditional branch. If such operators are used in the program, both branches of the conditional statement are included in the graph and both branches will be executed.

By the time the SWIN operator is reached, the results of both branches are available and the results of the selected branch will be send to the destination node. The SWIN operator, therefore, acts as a simple move operation and its default value is assumed to be equal to one addition operation.

7.7 User Defined Operators and Functions

User defined integration algorithms that do not require more than one calculation of the derivative function at each step and any user defined functions are supported by ALI. However, the users must know the execution time of those functions and will be prompted to enter them. The way the users can enter these values is described in the User Manual in Appendix A.

CHAPTER 8

PERFORMANCE EVALUATION

8.1 Introduction

In order to evaluate the usefulness of ALI, two measures of performance are considered. First, it is desirable to determine how much time is needed to convert a CSSL program into a data flow graph and to allocate it among the processors. The other performance measure is the efficiency of the prescheduling algorithm. It is also desirable to determine the execution of the program on the multiprocessor system and compare it with the execution time on a single processor system.

The conversion of the source program into the data flow graph and the preallocation of the graph are not done in real time. As long as the program and the execution times of individual operations are not changed, the data flow graph can be generated only once. Furthermore, the prescheduler, as its name implys, analyzes the graph prior to actual execution.

In most simulation applications, efficient execution of the program in real time is of main interest, and is quite tolerable to spend several minutes or even several hours for preallocation. The time spent for graph generation and preallocation varies with the program size and number of processors. The fewer the

number of processors, the more artificial dependencies must be added to the graph and, therefore, the longer the preallocation time. The jet engine benchmark(Figure 6.4), a program of 110 lines of code, was converted to a 102 nodes graph in 11 CPU-seconds on a VAX 11/750 host. The prescheduling time varied from 19 CPU-seconds for a 2-CPU configuration(the worst-case) to less than 1 CPU-second for an 11-CPU configuration. This time is well below the tolerable threshold for many applications. Therefore, in the rest of this chapter, only the second performance measure, i.e., the efficiency of the preallocation algorithm and the real-time execution of the program graph is considered.

In ALI, the execution times of all operations are known a priori, therefore, in order to evaluate the performance, there is no need to use the actual hardware; once the graph is preallocated among the processors, it is possible to precisely determine the graph execution time. Other performance evaluation methods, including the use of the Network II.5 [KARP 84], are also possible.

Three benchmarks are used to evaluate the usefulness of the system. These problems are the turboshaft jet engine(described in Chapter 6), the pilot ejection problem[STRA 67], and the nuclear power plant model[YEH 86]. The intent was to investigate whether the preallocation strategy actually provides for increased speed of the execution, and how much the execution time improves as more processors are added to the pool. The programs are written in CSSL-IV and the results are examined for various numbers of processors. The default execution times has been used for the jet engine and the power plant simulation problems. For the pilot ejection problem a fifth-order A-B integration algorithm with 92 time units and an

exponentiation algorithm with 200 time units were used.

8.2 Results of the Benchmarks

General information about the benchmarks is shown in Table 8.1. The first column shows the total number of nodes in the graph. Each node can be as simple as an addition or as complex as a two-dimensional table lookup. The second column shows the total execution times of all nodes of each graph, which is the execution time of the graph when using a single processor. The third column shows the critical paths of each graph and, therefore, the minimum time to execute them. The fourth column shows the lower bound for the number of processors below which it is not possible to execute the graph within the time of its critical path. And the last column shows the upper bound for the number of processors above which the additional processors will simply remain idle all the time.

Table 8.1 Benchmarks General Information

	number of nodes	total execution time	time of the critical path	minimum number of CPUs	maximum number of CPUs
Jet Eng.	102	2071	465	5	11
Pilot Ejec.	27	998	344	4	8
Power Pl.	98	1323	223	6	19

Each benchmark is executed with different numbers of processors. Figure 8.1 shows the execution times of each benchmark(y-axis) versus the number of processors(x-axis). Note that the y-axis shows the normalized execution times(execution times are normalized by dividing them by the execution time of addition/subtraction). These curves demonstrate two important points:

1) The total execution time does not improve beyond a certain threshold for the number of processors because the sequentiality constraints of each problem does not allow the execution times faster than the length of the critical path.

2) This threshold for the number of processors is reached by adding only a few processors and is well below the upper bound given in Table 8.1 which indicates the efficiency of the allocation algorithm.

As mentioned in Chapter 5, two different versions of the preallocation algorithm have been developed. Algorithm A does not allow processor idle times while algorithm B does. The results of the benchmarks show equivalent or better execution times for algorithm A. Furthermore, the preallocation time for algorithm A is faster because it does not add any more artificial dependencies on the nodes that are already active. Therefore, algorithm A is selected as the prescheduling algorithm for ALI(All curves in Figure 8.1 are algorithm A results). Figure 8.2 shows the difference between the results of the two algorithms for the jet engine problem. The results for the other benchmarks were closer to each other and are not repeated again.

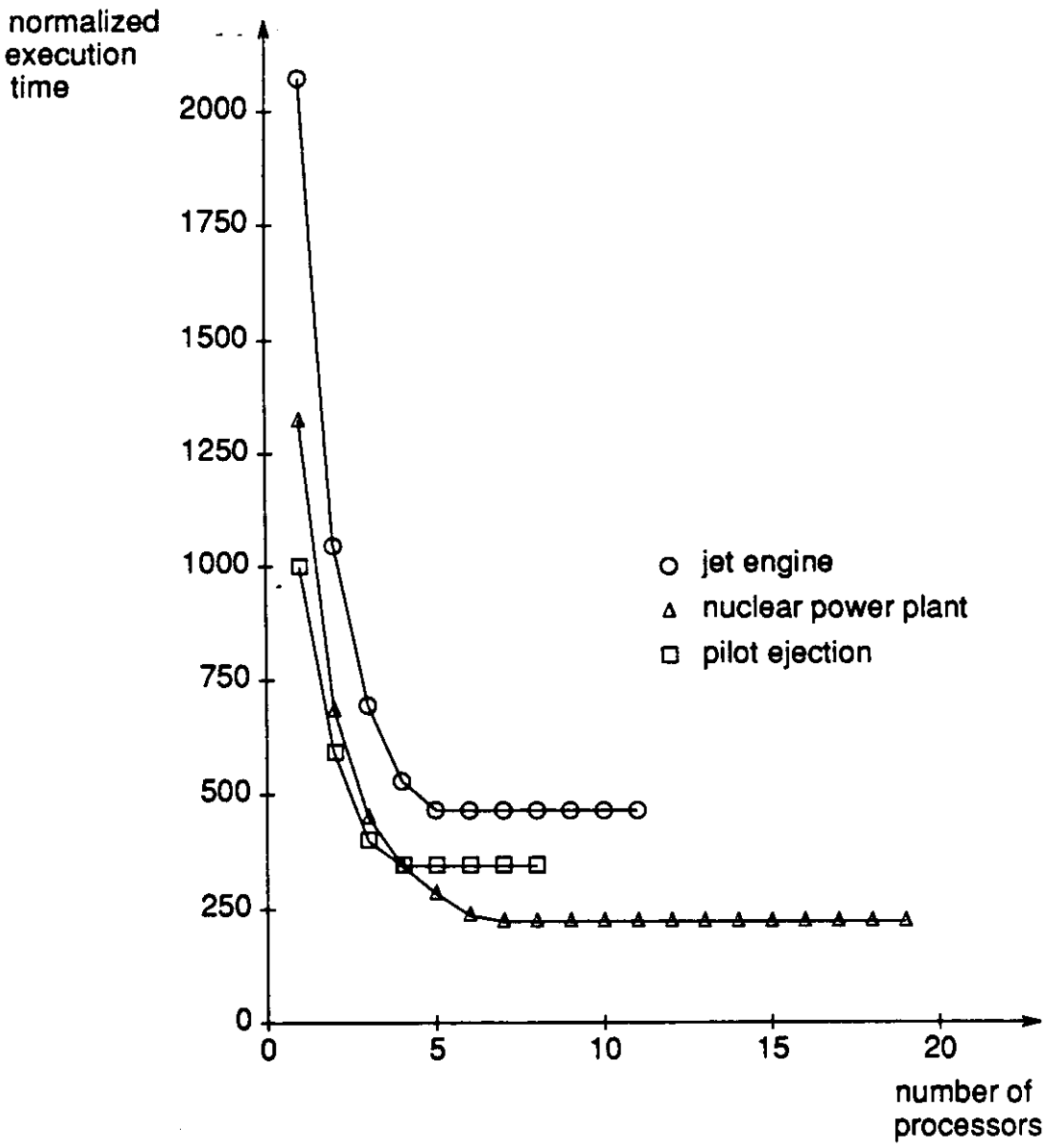


Figure 8.1 Execution times of the benchmarks

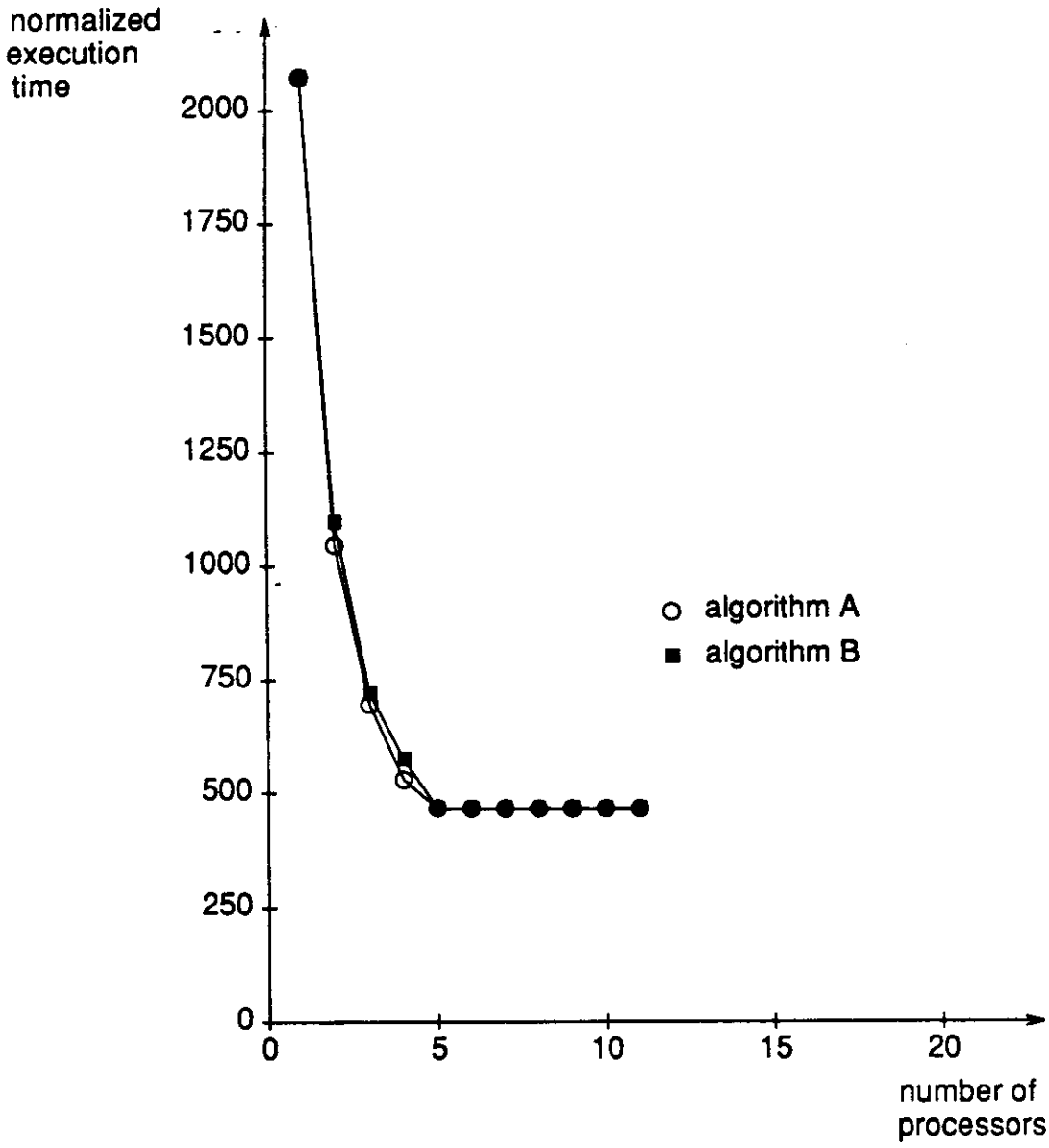


Figure 8.2 Comparison of the two algorithms for the jet engine problem

Among the important performance measures of multiprocessor systems are the speedup and efficiency of these systems compared to a single processor system[KUCK 78].

Speedup is defined as:

$$S_P = \frac{T_1}{T_P}$$

where T_1 is the execution time on a single processor

T_P is the execution time on P processors

Efficiency is defined as:

$$E_P = \frac{S_P}{P}\% = \frac{T_1}{T_P * P}\%$$

The efficiency is in fact the percent of the total time that CPUs perform useful calculations divided by the total time that they are available. The speedup curves and the efficiency curves for the benchmarks are shown in Figures 8.3 and 8.4 respectively. The speedup for all benchmarks increases by adding more processors until the execution time equals the execution time of the critical path. After that the speedup curves level off. The efficiency remains high as long as the execution time has not reached the length of the critical path. After that the efficiency decreases sharply due to the fact that the additional processors can not improve the execution time anymore and the added processing power is wasted.

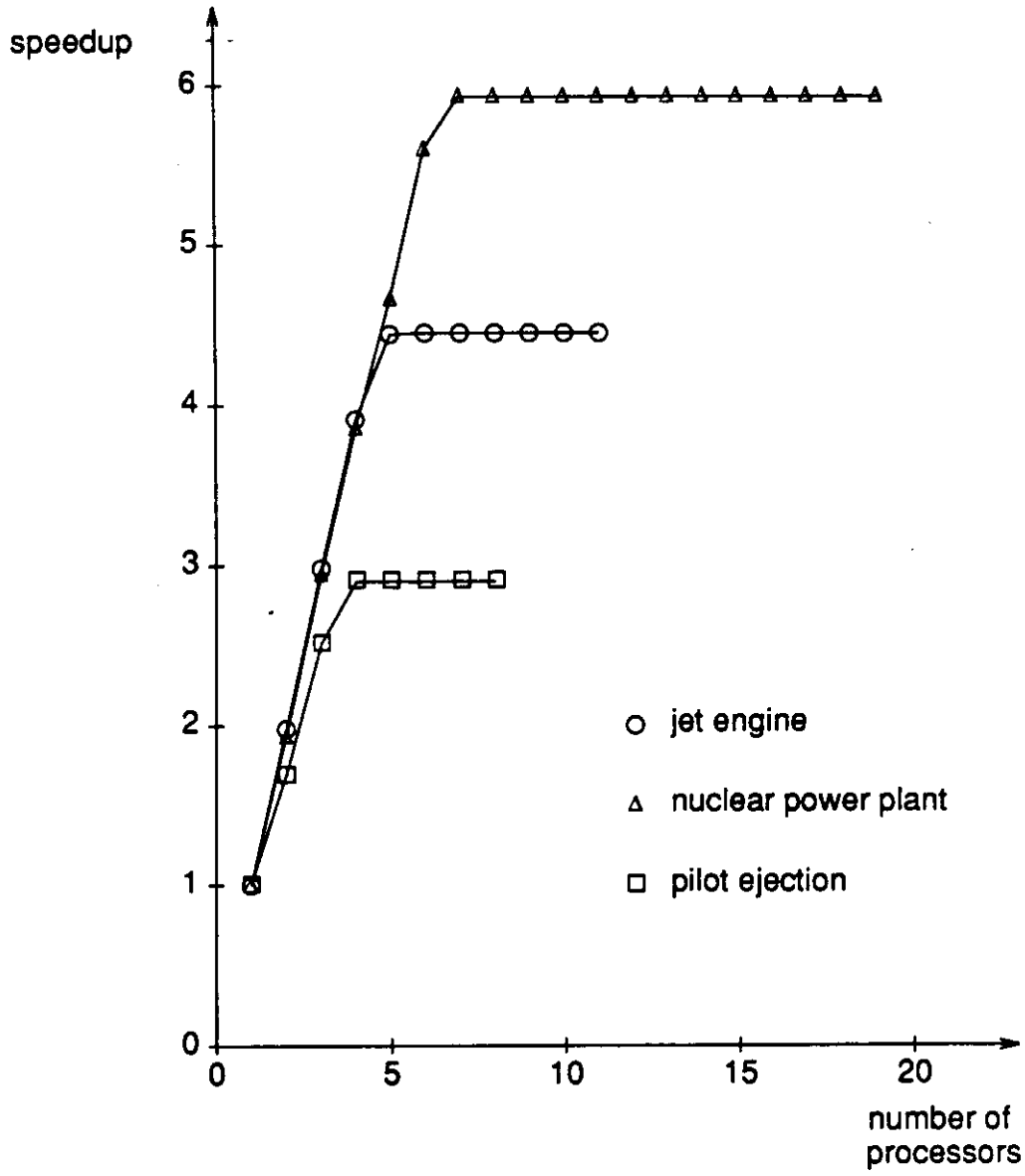


Figure 8.3 Speedup curves

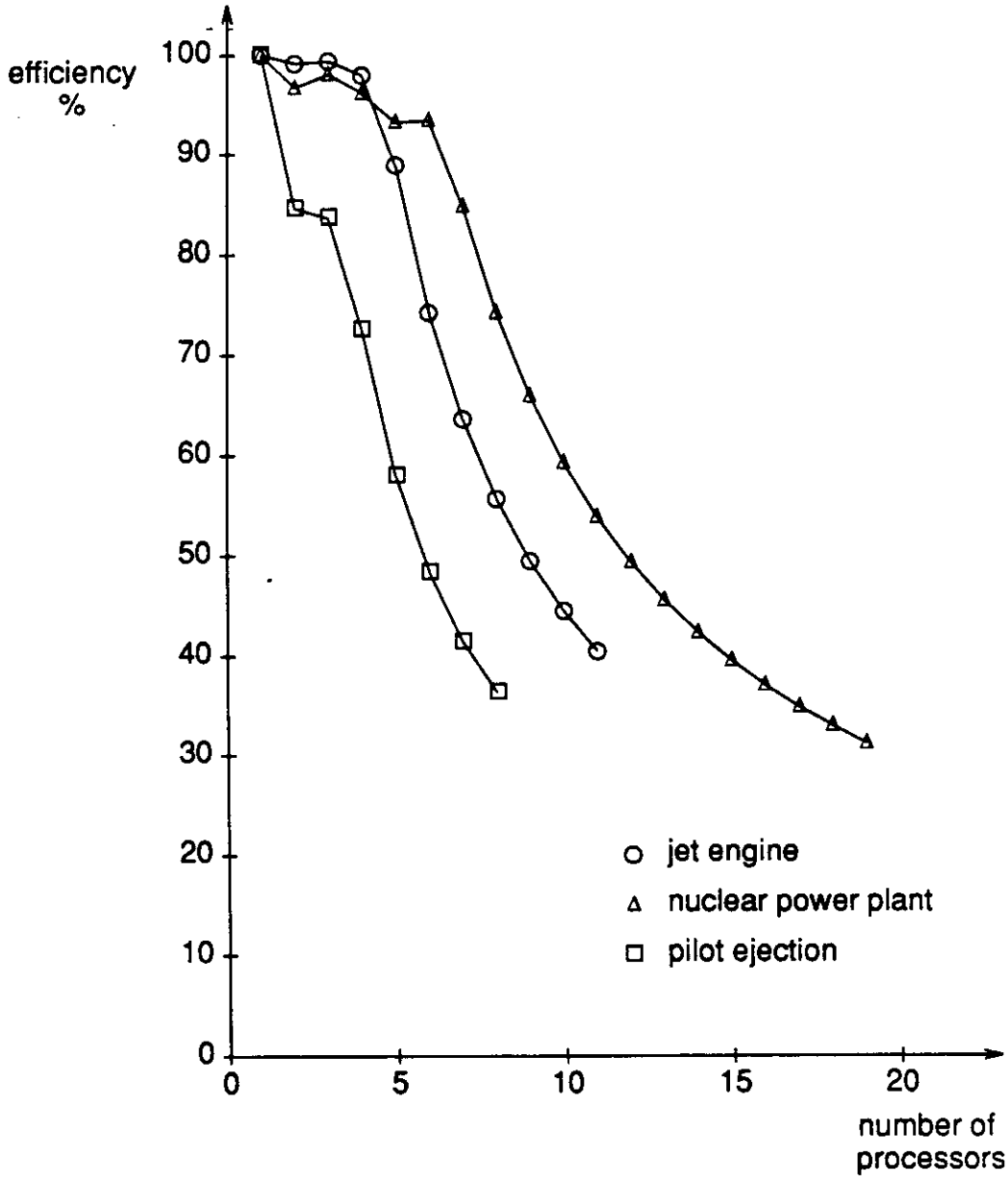


Figure 8.4 Efficiency curves

The results of the benchmarks clearly indicate the advantages of using a multiprocessor system over a single processor system. By adding only a few processors, a graph can be executed within the time of its critical path. Figure 8.3 shows that in all benchmarks, adding 2, 3, or 4 processors results in almost the same factor of increase in the speed of the execution. The choice of the number of processors depends on the trade-offs that the user has between the speedup, efficiency, and hardware costs. The efficiency curves can be viewed as normalized speedup curves(speedup divided by the number of processors). Therefore, the best combination of speedup and cost can be selected from these curves. For example, Figures 8.3 and 8.4 show that for the nuclear power plant problem, a 6-CPU configuration executes the graph within the time of its critical path and at the same time has more than 90% efficiency. This configuration, therefore, can be selected as the optimum configuration. On the other hand, for the jet engine problem, the 6-CPU configuration is less than 80% efficient and the 4-CPU configuration, which is more than 90% efficient, executes the graph slower than the time of the critical path. The user, must, therefore, make a trade-off between the speed of execution and the efficiency.

8.3 Effects of Communication Delays

As mentioned in Chapter 5, ALI uses a heuristic, which minimizes the effects of communication delays, in dividing the graph among the processors. ALI is also capable of analyzing the effects of different values of communication delays on the total execution time. Figure 8.5 shows the effects of communication delays on the

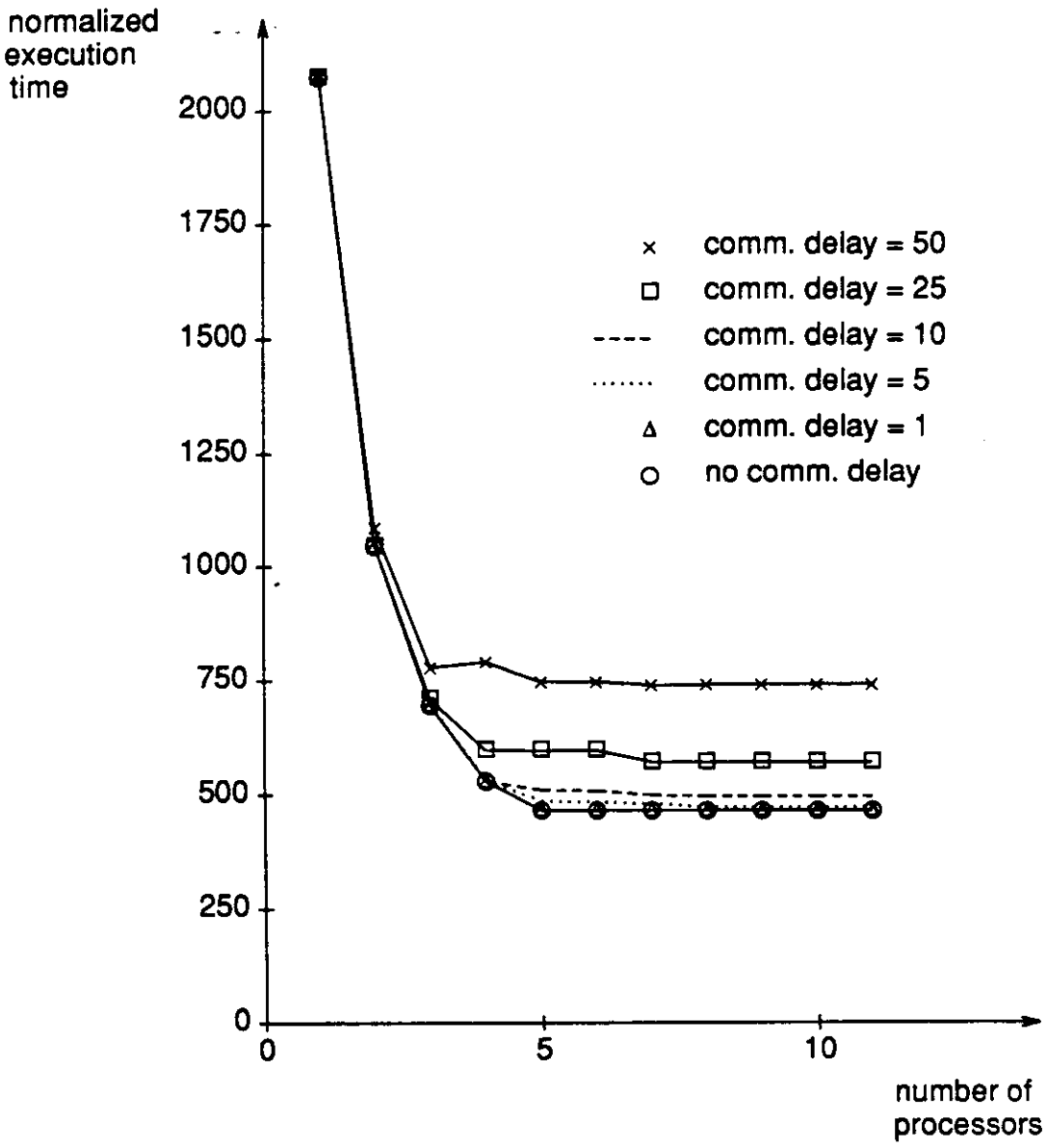


Figure 8.5 Effects of communication delays on the jet engine problem

jet engine problem. The communication time is normalized by dividing it by the execution time of the addition operation(as all other execution times in ALI are). The following points can be observed.

1) When the ratio of the communication time to the addition time is 1, there is no effect on the total execution time.

2) When the ratio is 5, the increase in the total execution time is always less then 4% for all different numbers of processors.

3) In most other multiprocessor systems, after the number of processors reaches a certain threshold, the total execution time, due to excessive interprocessor communication increases. In ALI, however, if the number of processors is more than the maximum number needed, the extra processors simply remain idle all the time. For example, for the jet engine problem, this maximum number is 11(Table 8.1). Therefore, in the 13-CPU configuration, two processors are always idle and there is no difference in the total execution time of the 13-CPU and 11-CPU configurations.

The nuclear power plant and the pilot ejection problems are even less sensitive to communication delays(Figures 8.6 and 8.7). A communication time to addition time ratio of 75 does not have any effect on the execution time of either benchmarks with any number of processors. A ratio of 100 only increases the execution times by a few percents. These results show the effectiveness of the heuristic used for reducing communication delays. The capability of analyzing the effects of communication delays allows the user to fine tune the system by selecting

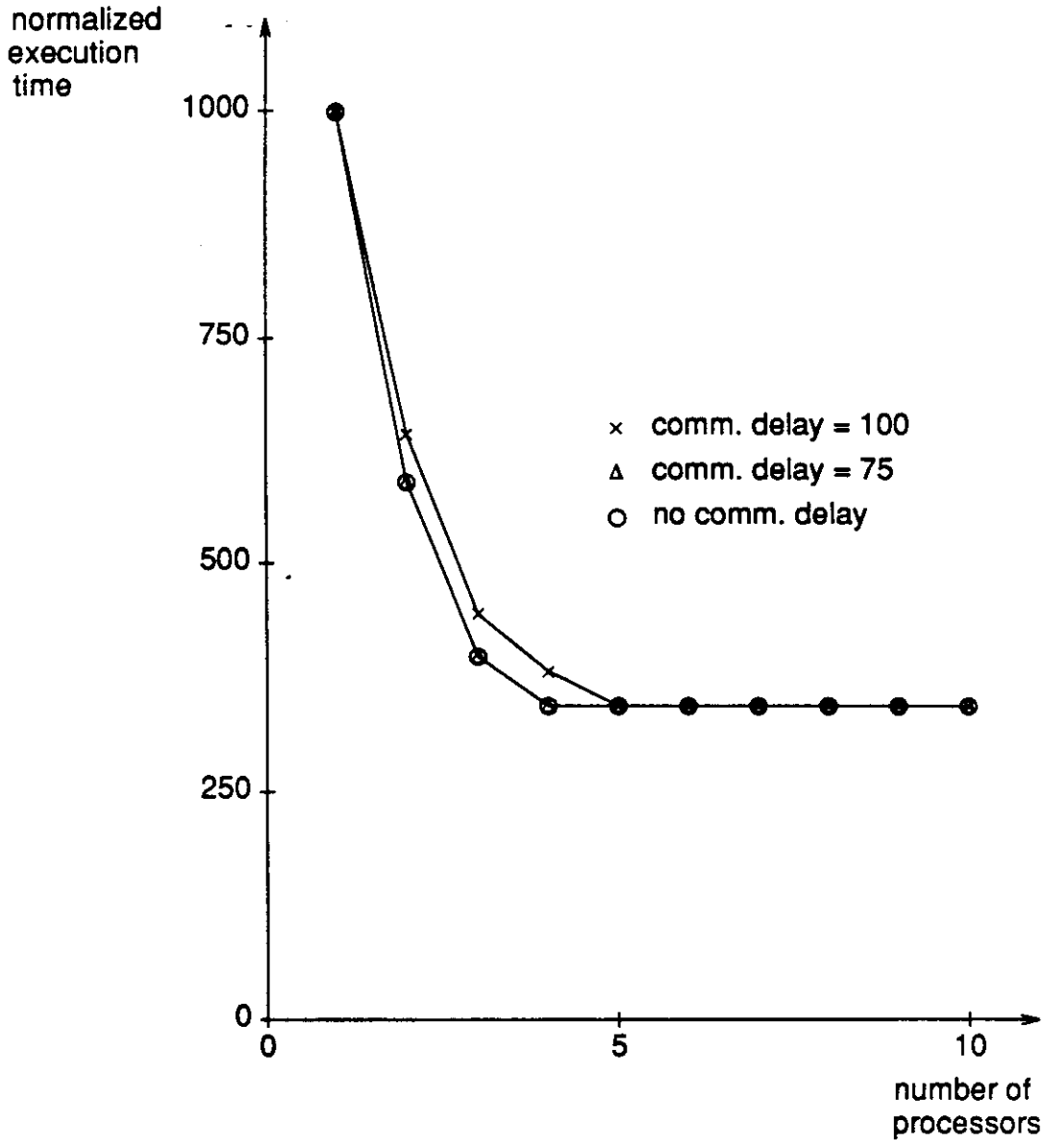


Figure 8.6 Effects of communication delays on the pilot ejection problem

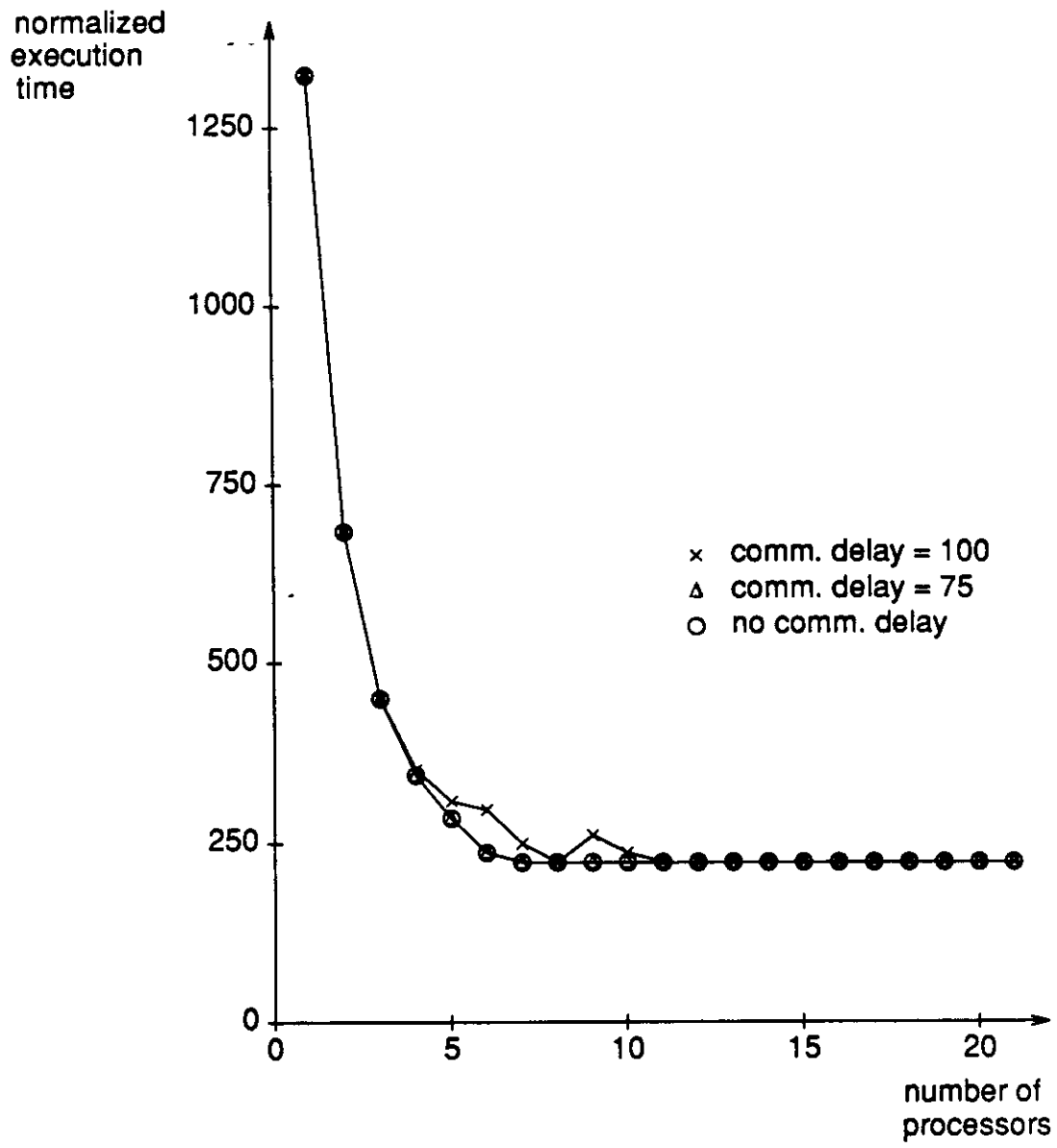


Figure 8.7 Effects of comm. delays on the nuclear power plant problem

the most suitable number of processors when the actual value of the interprocessor communication time is known.

CHAPTER 9

CONCLUSIONS

9.1 Conclusions

The methodology for designing a user friendly software interface for multiprocessor simulation has been presented. This software interface is especially important for the real-time simulation of systems modeled by ordinary differential equations and allows the user to use a multiprocessor system for simulation without having to partition the source program among the processors. A variety of general-purpose and simulation-oriented high-level languages has been carefully studied and two popular simulation languages, CSSL-IV and ACSL, have been selected as the high-level languages. The software translator accepts a source program written in either CSSL-IV or ACSL and automatically converts this program into a data flow graph and preallocates the graph among the processors.

The source code is first converted into tokens by the lexical analyzer. The token file is then converted from the infix notation into the postfix notation by the postfix code generator. Since the model definition part of a CSSL program is nonprocedural, the statements of the model definition part are sorted to ensure that the usage of a variable is preceded by its definition. The sorted postfix file is converted into an internally represented data flow graph. This internally represented

graph is converted to a printable file.

Provisions have been made for the user to either accept a set of default execution times based on the MC68000 microprocessor execution times or to enter user defined execution times and ,thereby, targeting the system on any type of microprocessor. The user is interactively directed to either accept the default execution times or to enter new values.

The existing algorithms for the allocation of graphs to multiprocessors have been extensively studied and the most suitable algorithm has been modified and enhanced to preallocate the data flow graph among the network of microprocessors. This algorithm balances the graph by delaying the execution of some nodes in such a way that at no time does the number of active nodes exceed the number of available processors. A heuristic is developed which allocates the balanced graph among the processors in a way that minimizes the interprocessor communication delays. The user can analyze the effects of different values of interprocessor communication time on the total execution time of the graph.

In order to evaluate the usefulness of the software interface, three benchmarks have been used. The results of the benchmarks demonstrate the advantages of using a multiprocessor system over a uniprocessor system. These results show that as long as the time of the critical path of a graph is not reached, an m-CPU configuration results in an almost m-times speedup over the single processor configuration. Furthermore, the analysis of the effects of communication delays show that if the ratio of the communication time to the addition time is not more

than two orders of magnitude, the total execution time remains close to the ideal execution time.

Finally, a self-contained user manual with all necessary information for using the system has been provided. In summary, this dissertation has demonstrated the feasibility of a user friendly software interface which facilitates the use of multiprocessor systems for simulation applications and provides the implementation of such an interface.

REFERENCES

[ARVI 76] Arvind, K. P. Gostelow, and W. Plouffe, *Programming in a Viable Data Flow Language,* Technical Report #89, Aug. 1976, Department of Information and Computer Science, University of California at Irvine.

[BACK 78] J. Backus, "Can Programming Be Liberated From the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM,* Vol. 21, No. 8, Aug. 1978, pp. 613-641.

[BARS 68] A. B. Barskiy, "Minimizing the Number of Computing Devices Needed to Realize a computational process within a specified time," *Engineering Cybernetics(USSR),* Vol. 6, Jan. 1968, pp. 59-63.

[BATC 74] K. E. Batcher, "STARAN Parallel Processor System Hardware," *AFIPS Conference Proceedings,* Vol. 43, 1974, National Computer Conference, pp. 405-410.

[BEKE 68] G. A. Bekey, and W. J. Karplus, *Hybrid Computation,* John Wiely & Sons, Inc., New York, 1968.

[BIRT 73] G. M. Birtwistle, O.J. Dahl, B. Myrhaug, and K. Nygaard, *SIMULA BEGIN,* Auerbach Publishers, Philadelphia, PA, 1973.

[BUSS 74] B. Bussell, E. B. Fernández, and H. O. Levy, "Optimal Scheduling for Homogeneous Multiprocessors," *Information Processing 74,* North Holland Publishing Co., Amsterdam, 1974, pp. 286-290.

[CARD 72] A. F. Cardenas, *The PDEL Language and Its Use to Solve Partial Differential Equation Problems,* University of California, Los Angeles, Computer Science Department Report, June 1972.

[CLAR 52] W. Clark, *The Gantt Chart(3rd ed.),* Sir Isaac Pitman and Sons Ltd., London, 1952.

[COFF 72] E. G. Coffman Jr., and R. L. Graham, "Optimal Scheduling for Two Processor Systems," *Acta Informatica,* Vol. 1, 1972, pp. 200-213.

[COFF 73] E. G. Coffman Jr., and P. J. denning, *Operating Systems Theory,* Prentice-Hall Inc., Englewood Cliffs, N. J., 1973.

[COFF 76] E. G. Coffman Jr., (Ed.), *Computer and Job-Shop Scheduling Theory,* John Wiley and Sons, N. Y., 1976.

[COMT 79] D. Comte, N. Hifdi, "LAU Multiprocessor: Microfunctional Description of Technological Choices," 1st European Conference on Parallel and Distributed Processing, Toulouse, France, Feb. 1979, pp. 8-15.

[CONN 79] W. D. Connors, J. H. Florkowski, and S. K. Patton, "The IBM 3033: An Inside Look," *Datamation,* May 1979, pp. 198-218.

[COOK 71] S. A. Cook, "The Complexity Of Theorem-Proving Procedures," *Proceedings of 3rd ACM Symposium in Theory of Computing,* ACM, N. Y., 1971, pp. 151-158.

[CONW 67] R. W. Conway, W. L. Maxwell, and L. W. Miller, *Theory of Scheduling,* Addison-Wesley Publishing Co. Inc., Reading, Mass., 1976.

[DENN 80] J. B. Dennis, "Data Flow Supercomputers," *Computer,* Vol. 13, No. 11, Nov. 1980, pp. 48-56.

[ERCE 84a] M. D. Ercegovac, and W. J. Karplus, "On a Data Flow Approach in High-Speed Simulation of Continuous Systems," *Proceedings of the International Workshop on High-Level Computer Architecture 84,* Los Angeles, 1984, pp. 2.1-2.8.

[ERCE 84b] M. D. Ercegovac, et al., "Task Partitioning and Simulation For a Data Flow Multimicroprocessor System," *The Proceedings of the 1984 Summer Simulation Computer Conference,* July 23-25, 1984, Boston, Ma, pp. 326-331.

[ERCE 84c] M. D. Ercegovac, P. K. Chan, and T. M. Ravi, "A Data Flow Multiprocessor Architecture for High-Speed Simulation of Continuous Systems," *Proceedings of the International Workshop on High-Level Computer Architecture 84,* Los Angeles, 1984, pp. 2.9-2.17.

[FERN 72] E. B. Fernández, "Activity Transformation on Graph Models of Parallel Computation," *Report No. UCLA-ENG-7278,* Computer Science Department, University of California, Los Angeles, Oct. 1972.

[FERN 73] E. B. Fernández, and B. Bussell, "Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules, *IEEE Transactions on Computers,* Vol. C-22, No. 8, Aug. 1973, pp. 745-751.

[FLYN 72] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers,* Vol. C-21, No. 9, Sep. 1972, pp. 948-960.

[FUGI 69] M. Fuji, T. Kasami, and K. Ninomiya, "Optimal Sequencing of Two Equivalent Processors," *SIAM Journal of Applied Mathematics,* Vol. 17, No. 4, Jul. 1969, pp. 784-789.

[GASK 63] R. A. Gaskill, J. W. Harris, and A. L. McKnight, "DAS - A Digital Analog Simulator," *AFIPS Conference Proceedings,* Vol. 23, 1963, Spring Joint Computer Conference, p.83.

[GLAU 78] J. R. W. Glauert, *A Single Assignment Language for Data Flow Computing,* MSc thesis, University of Manchester, 1978.

[GORD 75] G. Gordon, *The Application of GPSS V to Discrete System Simulation,* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1975.

[GOST 79] K. P. Gostelow, and R. E. Thomas, "A View of Data Flow," *AFIPS Conference Proceedings,* Vol. 48, 1979, National Computer Conference, pp. 629-636.

[HALI 83] H. I. Halin, "The Applicability of Taylor Series Methods in Simulation," *The Proceedings of the 1983 Summer Computer Simulation Conference,* July 11-13, 1983, Vancouver, B.C., Canada, pp. 1032-1076.

[HARN 64] R. T. Harnett, F. J. Sanson, and L. M. Warshawsky, "MIDAS ... An Analog Approach to Digital Computation," *Simulation,* Vol. 3, No. 3, Sept. 1964, pp. 17-43.

[HART 78] C. E. Hart, *Function Generation Subprograms for Use in Digital Simulations,* Technical Memorandum No. X-71526, NASA, 1974.

[HART 84] C. E. Hart, and L. M. Wenzel, Real-Time Hybrid Computer *Real-Time Hybrid Computer Simulation of a Small Turboshaft Engine and Control System,* Technical Memorandum No. 83579, NASA, Feb. 1984.

[HAYN 82] L. S. Haynes, R. L. Lau, D. P. Siewiorek, and D. W. Mizell, "A Survey of Highly Parallel Computing," *Computer*, Vol. 15, No. 1, Jan. 1982, pp. 9-24.

[HILD 52] B. Hildebrand, *Introduction to Numerical Analysis*, McGraw-Hill, N. Y., 1956.

[HU 61] T. C. Hu, "Parallel sequencing and assembly line problems," *Operations Research, Vol. 9, N0. 6, Nov. 1961, pp. 841-848.*

[JOHN 80] D. Johnson, et al., "Automatic Partitioning of Programs in Multiprocessor Systems," *Proceedings of Compcon Spring 80*, Feb. 1980, pp. 175-178.

[JONE 77] A. K. Jones, et al., "Software Management of Cm* - A Distributed Multiprocessor," *AFIPS Conference Proceedings*, Vol. 46, 1977, National Computer Conference, pp. 657-663.

[KARP 72] R. M. Karp, "Reducibility Among Combinatorial Problems," *Complexity of Computer Computation*, R. E. Miller, and J. W. Thatcher (Eds.), Plinum Press, N. Y., 1972, pp. 85-104.

[KARP 74] W. J. Karplus, and R. N. Nilsen, "Continuous System Simulation Languages: A State-of-the-ART Survey," Annales de l'Association Internationale pour le Calcul Analogique, No. 1, Jan. 17, 1974.

[KARP 77] W. J. Karplus, "Peripheral Processors for High-Speed Simulation," *Simulation*, Vol. 29, No. 5, Nov. 1977, pp. 143-153.

[KARP 82] W. J. Karplus, and A. Makoui, "The Role of Data Flow Methods in Continuous Systems Simulation," *The Proceedings of the 1982 Summer Computer Simulation Conference*, July 19-21, 1982, Denver, Co., pp. 13-16.

[KARP 84a] W. J. Karplus, "The Changing Role of Peripheral Array Processors," *The Proceedings of the Conference on Peripheral Array Processors*, Oct. 11-12, 1984, Boston, Ma, pp. 1-13.

[KARP 84b] W. J. Karplus, and S. Cheung, "Performance Evaluation Tools for Simulators Consisting of Networks of Microcomputers," *The Proceedings of the 1984 Summer Computer Simulation Conference*, July 23-25, 1982, Boston, Ma., pp. 317-325.

[KART 80] S. p. Kartashev, and S. I. Kartashev, "Supersystems for the 80's," *Computer*, Vol. 13, No. 11, Nov. 1980, pp. 11-14.

[KIVI 75] P. J. Kiviat, R. Villanueva, and H. M. Markowitz, (Ed. E. C. Russell), *SIMSCRIPT II.5 Programming Language,* CACI, Inc., Los Angeles, CA., 1975.

[KUNG 82] H. T. Kung, "Why Systolic Architectures?", *Computer,* Vol. 15, No. 1, Jan. 1982, pp. 37-46.

[LANG 77] T. Lang, and E. B. Fernández, "Improving the Computation of Lower Bounds for Optimal Schedules," *IBM Journal of Research and Development,* Vol. 21, May 1977, pp. 273-280.

[LEVY 73] H. O. Levy, *Application of Graph Transformations to Scheduling* MS Thesis, School of Engineering and Applied Science, University of California, Los Angeles, 1973.

[MAGO 79a] G. A. Mago, "A Network of Microprocessors to Execute Reduction Languages, Part I," *International Journal of Computer and Information Services,* Vol. 8, No. 5, 1979, pp. 349-385.

[MAGO 79b] G. A. Mago, "A Network of Microprocessors to Execute Reduction Languages, Part II," *International Journal of Computer and Information Services,* Vol. 8, No. 6, 1979, pp. 435-471.

[MAKO 83] A. Makoui, and W. J. Karplus, "Data Flow Methods for Dynamic System Simulation: A CSSL-IV Microcomputer Network Interface," *The Proceedings of the 1983 Summer Computer Simulation Conference,* July 11-13, 1983, Vancouver, B.C., Canada, pp. 376-382.

[MART 69] D. F. Martin, and G. Estrin, "Path Length Computation on Graph Models of Computations," *IEEE Transactions on Computers,* Vol. C-18, No. 6, June 1969, pp. 530-536.

[McGR 80] J. R. McGraw, "Data Flow Computing - Software Development," *IEEE Transactions on Computers,* Vol. C-29, No. 12, Dec. 1980, pp. 1095-1103.

[MCNA 59] R. M. McNaughton, "Scheduling With Deadlines and Loss Functions," *Management Science,* Vol. 6, No. 1, Oct. 1969, pp. 1-12.

[MITC 75] Mitchell and Gauthier, Assoc., Inc., ACSL, *Advanced Continuous Simulation Language, User Guide/Reference Manual,* 1975, Concord, MA.

[MOTO 82] Motorola, *MOTOROLA MC68000 16-BIT Microprocessor User's Manual, 3rd edition,* Prentice Hall, Inc., Englewood Cliffs, N. J.

[MUNT 69] R. R. Muntz, and E. G. Coffman, Jr., "Optimal Preemptive Scheduling on Two-Processor systems," *IEEE Transactions on Computers*, Vol. C-18, No. 11, Nov. 1969, pp. 1014-1020.

[MUNT 70] R. R. Muntz, and E. G. Coffman Jr., "Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems," *Journal of ACM*, Vol. 17, No. 2, Apr. 1970, pp. 324-338.

[NILS 76] Nilsen Associates, *CSSL-IV User's Guide and Reference Manual*, Chatsworth, Ca., 1976.

[NILS 84] Nilsen Associates, Simulation Services Div., *CSSL-IV Reference Manual*, Chatsworth, Ca., 1984.

[PRIT 74] A. Alan B. Pritsker, *The GASP IV Simulation Language*, John Wiley and Sons, Inc., New York, 1974.

[PUGH 70] A. L. Pugh, *DYNAMO II User's Manual*, MIT Press, Cambridge, Mass., 1970.

[RAMA 72] C. V. Ramamoorthy, K. M. Chandy, and Mario J. Gonzalez, "Optimal Scheduling Strategies in a Multiprocessor System," *IEEE Transactions on Computers*, Vol. C-21, Feb. 1972, pp. 137-146.

[RUSS 69] E. C. Russell, *Automatic Program Analysis*, Doctoral Dissertation, Department of Electrical Engineering, University of California, Los Angeles, 1969.

[SCHI 73] W. E. Schiesser, *An Introduction to LEANS-III Lehigh Analog Simulator Version III and DDS Distributed System Simulator*, Lehigh University, Bethlehem, PA, Computer Center Report, 1973.

[SHOC 82] J. F. Shoch, Y. K. Dalal, D. D. Redell, and R. C. Crane, "Evaluation of the Ethernet Local Computer Network," *Computer*, Vol. 15, No. 8, Aug. 1982, pp. 10-27.

[STRA 67] J. C. Straus, et al., "The SCI Continuous System Simulation Language(CSSL)," *Simulation*, Vol. 9, No. 6, Dec. 1967, pp. 281-303.

[SWAN 77] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm*: A Modular Multi-Processor," *AFIPS Conference Proceedings*, Vol. 46, 1977 National Computer Conference, pp. 637-644.

[SZUC 78] J. R. Szuch, "Models for Jet Engine Systems, Part I Techniques for Jet Engine Systems Modeling," *Control and Dynamics Systems*, Vol. 14, 1978, pp. 213-257.

[TANE 81] A. S. Tanenbaum, *Computer Networks,* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.

[VICK 79] C. R. Vick, *A Dynamically Reconfigurable Distributed Computing System,* Doctoral Dissertation, Auburn University, Auburn, Ala., 1979.

[VICK 80] C. R. Vick, S. P. Kartashev, and S. I. Kartashev, "Adaptable Architectures for Supersystems," *Computer,* Vol. 13, No. 11, Nov. 1980, pp. 17-35.

[WATS 82] I. Watson, and J. Gurd, "A Practical Data Flow Computer," *Computer,* Vol. 15, No. 2, Feb. 1982, pp. 51-57.

[YEH 86] H. C. Yeh, W. E. Kastenberg, and I. Catton, "Nuclear Power Station Simulation," *The Proceedings of the 1986 Summer Computer Simulation Conference,* July 28-30, 1986, Reno, Nevada, pp. 1064-1069.

APPENDIX A

USER MANUAL

A-1 Introduction

This appendix is a self-contained user manual providing all necessary information for using ALI(A Language Interface). In the following sections, an overview of the system is given. This is followed by instructions to run the system, to enter the values of the tunable parameters, and to interpret the results.

A-2 System Overview

The user writes the source program in either CSSL-IV or ACSL. Both languages belong to the class of Continuous System Simulation Languages(CSSL) and are especially designed for the simulation of systems that are described by systems of ordinary differential equations.

ALI does not perform syntax analysis on the source code. The user ,therefore, must use the proper CSSL compiler to find compilation errors. The program should also be executed on the sequential host machine to find programming errors and run-time errors, such as divide by zero, etc. ALI converts an error free program into a graph and divides it among the processors of the parallel target system.

The software in ALI is organized in seven files(Table A-1). Each major function except the data flow graph generation is placed in a separate file. The data flow graph generation, due to its complexity, is placed in three separate files: the "dataflowgen" file generates the graph in an internal form usable by other modules, the "getexectimes" file interactively sets the execution times of each operator and function used in the program, and the "formatgraph" file reformats the internal graph into a printable form.

Table A-1 Source Files of ALI

File Name	Description
scanner.p	Lexical Analyzer
postfixgen.p	Postfix Code Generator
sorter.p	Sorter
dataflowgen.p	Internal Graph Generator
getexectimes.p	Get Execution Times
formatgraph.p	Format Graph
scheduler.p	Prescheduler

The source files are written in Berkeley PASCAL and their names are appended by ".p" extension. The executable object files have the same name without any extension for the UNIX operating system or with .EXE extension for the VMS operating system. These object files can be executed separately or one after the other using a command file.

A-3 Command Files

A command file is an executable file that contains one or more commands to the operating system. If these commands are for executing object files, then activating the command file will execute each individual object file in turn.

The software written in ALI is operating system independent. Therefore, depending on the operating system of the host computer, a command file can be made to run the simulation. As example, command files for UNIX and VMS operating systems are included(Figures A-1 and A-2). A command file in UNIX is activated by typing its name at the terminal. A command file in VMS is activated by typing character @ followed by the command file name at the terminal.

In UNIX, individual executable object files are executed by typing their name at the terminal. For example, typing scanner will start the Lexical Analyzer. Therefore, the command file must contain the name of all executable object files. In VMS, typing "run" followed by the name of the object file does the same. Therefore, the command file must contain the "run" command for each executable module. Both the UNIX command file(Figure A-1) and the VMS command file(Figure A-2) contain requests for the user to enter the name of the source program and then start the simulation by executing each module in turn.

```
echo -n 'enter the program name>'
read filename
cp $filename sourceprog
scanner
postfixgen
sorter
dataflowgen
getexectimes
formatgraph
scheduler
```

Figure A-1 Command file to run ALI in UNIX

```
inquire filename "enter the program name>"
$copy 'filename' sourceprog
run scanner
run postfixgen
run sorter
run dataflowgen
run getexectimes
run formatgraph
run scheduler
```

Figure A-2 Command file to run ALI in VMS

Note that in order to allocate the same program among different numbers of processors, there is no need to convert the source code into the data flow graph each time. When a program is converted into a data flow graph for the first time, the results are stored in intermediate files. And as long as the source program is not changed, the same intermediate results can be used for prescheduling. In other words, all steps except the prescheduler need to be executed only once for each program(Figure A-3). After that, the prescheduler can be executed as many times as the user wants(by typing scheduler.obj at the terminal) to examine the program allocation on different numbers of processors. Normally the user should try to reduce the number of processors as much as possible, in order to reduce communication times.

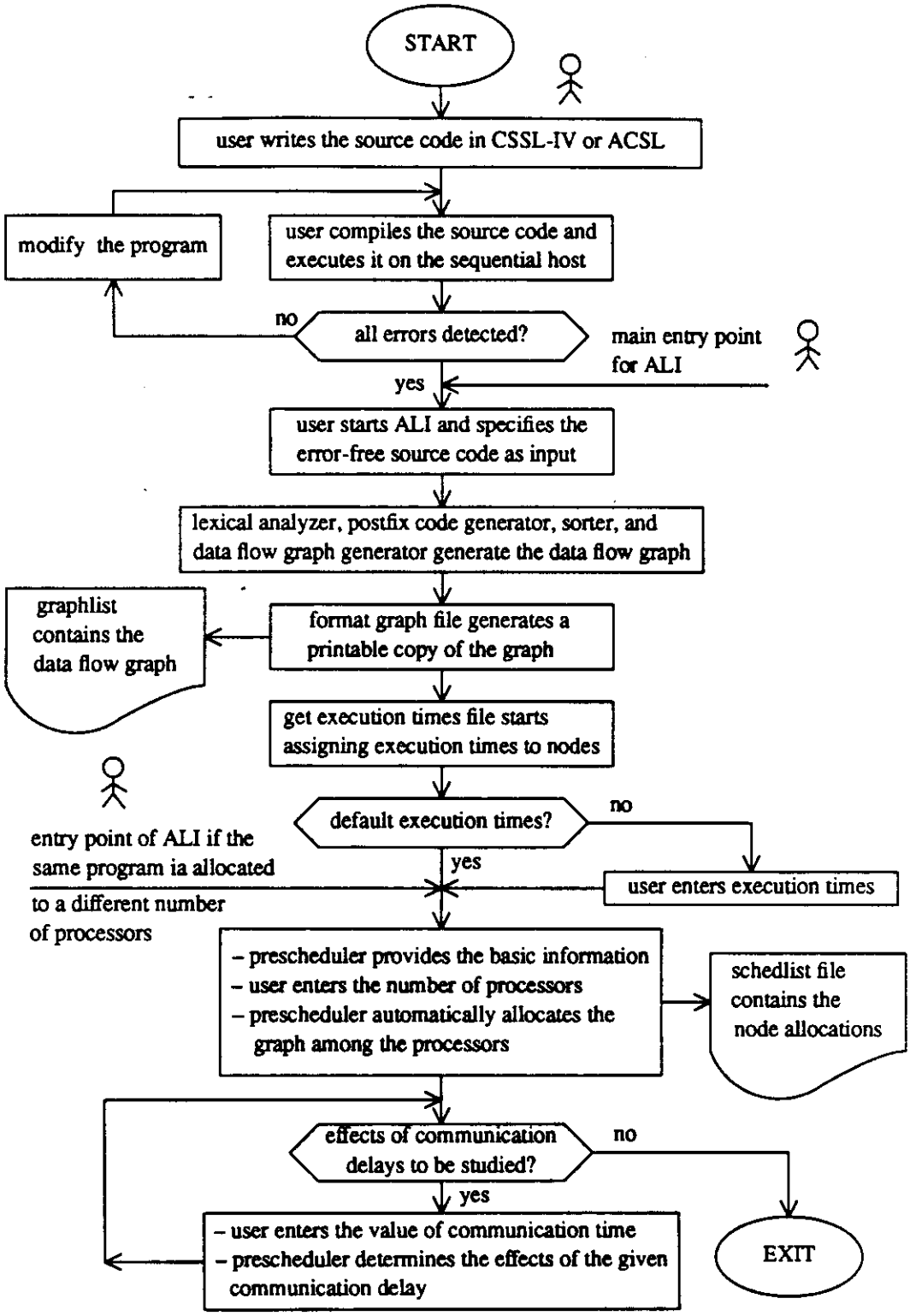


Figure A-3 Different steps in using ALI

A-4 Interactive Simulation

In order to make the modeling as flexible as possible, some system parameters are tunable, i.e., the user can change them for each simulation run. These parameters are the execution times of simple operators,such as addition and multiplications, which depend on the specific processor; the execution times of functions and more complex operators,such as integration and table lookup, which depend on both the specific algorithm to implement them and the execution times of the simple operators; and the number of processors which must be minimized to reduce communication delays.

ALI is capable of accepting all standard CSSL operators as well as any user defined functions. However, since the system is designed to work for any type of microprocessor, the user must know the execution times of all simple operators, and therefore the execution times of functions and complex operators, and will be prompted to enter the estimated times of the operators and functions used in the program. Use of some CSSL operators, such as IMPL(implicit loop), which involve iteration loops are discouraged because there are no default values for these operators and the user must estimate their execution times.

For the casual user, default execution times for simple operators and some widely used functions are provided. The execution times of simple operators are found from the MOTOROLA MC68000 User's Manual[MOTO 82]. The execution times are found by averaging the values for different operand sizes and operation modes. Since the execution times depend on the external clock rate of the

microprocessor, the execution times are given in terms of the external clock cycles rather than the actual microsecond figures(Table A-2). The execution times are normalized by dividing them by the execution time of the addition operator and rounding them to the closest integer. The default execution times of more complex operators are found from the default algorithms explained in Chapter 7 and are shown in Table A-3.

Table A-2 Default Execution Times of Simple Operators

function/ operator	execution time
addition(+)	1
division(/)	26
multiplication(*)	12
negation(-)	1
subtraction(-)	1

Table A-3 Default Execution Times of Functions

function/ operator	execution time	function/ operator	execution time
acos	63	integ	32
alog	63	mapfun	211
alog10	63	realpl	59
asin	63	sin	63
atan	63	sqrt	63
cos	63	swin	1
exp	63	tan	63
**	138	tanh	63
fun1	63		

The user has the choice of using the default execution times given in Tables A-2 and A-3 or to use any other execution times suitable for each specific application. When the user is prompted to enter the execution times, a help

command is available that explains the procedure. After the execution times are entered, the user can review them and correct them again if necessary.

The simulation is then continued by starting the prescheduler, which analyzes the graph, provides preliminary information, and prompts the user to enter the number of processors for this simulation run. When the user enters the number of processors, the simulation continues and the graph is automatically divided among the processors.

The pilot ejection problem is used as an example to show how the system can be used. The purpose of the pilot ejection problem is to determine the trajectory of a pilot ejected from a fighter aircraft in order to find out whether he will strike the vertical stabilizer of the aircraft[STRA 67]. The pilot and seat travel along rails and are disengaged from the rails at $Y = Y_1$, at velocity V_E, and at an angle θ_E backward from vertical(Figure A-4).

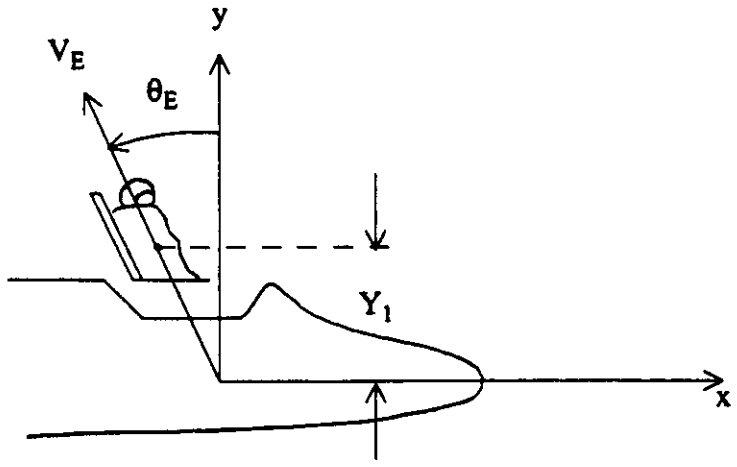


Figure A-4 Initial conditions for the pilot ejection problem

The initial conditions are given by:

$$V(0) = \sqrt{(V_A - V_E \sin\theta_E)^2 + (V_E \cos\theta_E)^2}$$

$$\theta(0) = \tan^{-1} \frac{V_E \cos\theta_E}{V_A - V_E \sin\theta_E}$$

$$X(0) = Y(0) = 0$$

Once the pilot and seat are ejected, they follow a ballistic trajectory. Since it is the relative motion of the pilot with respect to the aircraft that is important, it makes sense to assume that the coordinate axes are fixed on the aircraft and to formulate the problem so as to obtain the relative motion of the pilot with respect to the aircraft directly(Figure A-5)

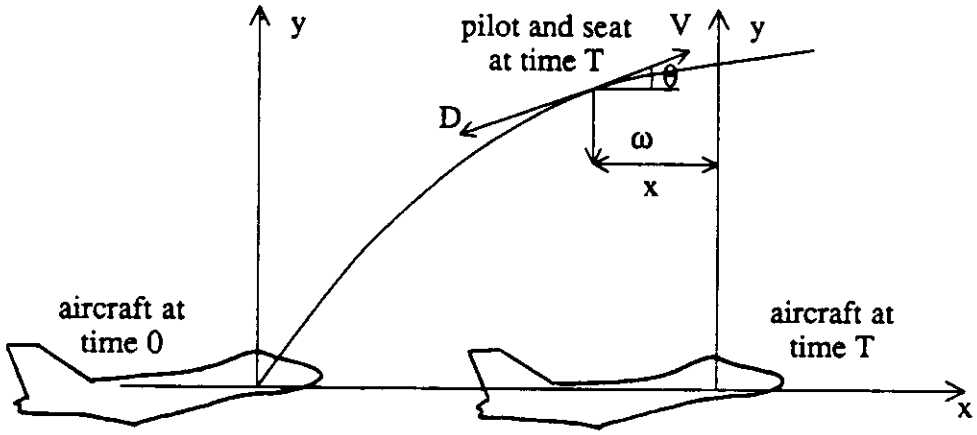


Figure A-5 Trajectory of the pilot after ejection

The governing equations are:

$$\dot{X} = V\cos\theta - V_A$$

$$\dot{Y} = V\sin\theta$$

$$\dot{V} = \begin{cases} 0 & \text{if } 0 \le Y < Y_1 \\ \dfrac{-D}{M} - g\sin\theta & \text{if } Y \ge Y_1 \end{cases}$$

$$\dot{\theta} = \begin{cases} 0 & \text{if } 0 \le Y < Y_1 \\ \dfrac{-(g\cos\theta)}{V} & \text{if } Y \ge Y_1 \end{cases}$$

$$D = \frac{1}{2}\rho C_D S V^2$$

The CSSL-IV program for this problem is shown in Figure A-6. The real-time simulation starts when the code in the DYNAMIC region is first executed and ends when the DYNAMIC region is terminated. The statements defining the model are contained in the DERIVATIVE section within the DYNAMIC region. In a CSSL program, the INITIAL and TERMINAL regions are executed only once. Furthermore, the run-time commands part at the end of the program are commands to the executive to exercise the model and are not executed during the real-time execution of the program.

Therefore, the main effort to speed up the system is spent in executing the DYNAMIC region and especially the DERIVATIVE section within it. Consequently, ALI converts the DERIVATIVE section into a data flow graph, extensively analyzes it, and divides it among different processors in a way to be executed as rapidly as possible. Before using ALI, however, the user should compile this program with a CSSL-IV compiler and should run the object code on the host computer(not necessarily in real-time) until all compilation errors and run time errors are detected and the user is satisfied that the program does what it is supposed to do. At this point the user starts using ALI to allocate this program on a network of microprocessors.

```
program pilot ejection
 initial
  constant thedeg = 15.0, degrad = 57.3, ...
   mass = 7.0, cd = 1.0, s = 10.0, y1 = 4.0, ...
   g = 32.2, ve = 40.0, r0 = 0.0023769, ...
   va = 900., xmn = -60.0, ymx = 30.0, ...
   tmx = 40.0
  cinterval cint = 0.01
  the = thedeg/degrad
  comment  seat initial velocity
  vx = va - ve * sin(the)
  vy = ve * cos(the)
  vic = sqrt(vx ** 2 + vy ** 2)
  thic = atan2(vy, vx)
 end initial
 dynamic
  derivative eject
   comment  relative positions
   x = integ(v * cos(th) - va, 0.0)
   y = integ(v * sin(th), 0.0)
   comment  space velocity and flight path angle
   v = integ(yge1 * (-d / mass - g * sin(th)), vic)
   th = integ(yge1 * (-g * cos(th) / v), thic)
   comment  compute drag
   d = 0.5 * r0 * cd * s * v ** 2
   yge1 = swin(y1 - y, 0.0, 1.0)
  end derivative
  termt(x .le. xmn .or. y .ge. ymx .or. t .ge. tmx)
 end dynamic
 terminal
 end terminal
end program
comment run-time commands
 hdr pilot ejection
 prepar t,th,v,x,y
 start
  print t,th,v,x,y
  plot t, th, v, x, y
 stop
```

Figure A-6 Pilot ejection program in CSSL-IV

An example of a session to do this allocation is given below. For clarity, the inputs typed by the user are written in boldface. It is assumed that the commands to run the system are in a UNIX command file called ALI. The numbers on the left

hand side of each line are not a part of the system output and are added here in order to refer to them in the following discussion.

(1) **ALI**
(2) enter the name of the program > **pilotejection**
(3) lexical scanner started
(4) postfix code generation started
(5) sorter started
(6) data flow graph generation started
(7) do you want default operation times? y=yes, n=no, h=help>y
(8) prescheduler started
(9) total execution time : 696
(10) time of critical path : 222
(11) number of nodes : 27
(12) minimum number of processors needed : 4
(13) maximum number of processors needed : 8
(14) enter number of processors for this run >**6**
(15) ---------- time-step 1
(16) ---------- time step 13
(17) ---------- time step 25
(18) ---------- time step 27
(19) do you want to analyze the effects of comm. delays? y=yes, n=no, h=help>y
(20) enter the ratio of communication time to addition time>**10**
(21) do you want to try another value?>**y**
(22) enter the ratio of communication time to addition time>**100**
(23) do you want to try another value?>**n**
(24) file graphlist contains the program graph
(25) file schedlist contains the results of prescheduling
(26) end of simulation

The user starts the system by executing command file ALI which contains the commands to run the system(line 1). The user is prompted to enter the name of the program and the user specifies pilotejection as the name of the source program(line 2). This file is used as the input for the lexical analyzer. The lexical analyzer and other system modules(postfix code generator, sorter, and data flow graph generator) are activated by the command file one after the other. Each module writes a message on the terminal to indicate that it has started(lines 3 to 6). During the data flow graph generation, the user must decide whether to use the default

execution times for each graph operation or to enter user defined operation times. In this example, the user decides to use the default operation times(line 7).

Next, the prescheduler is activated(line 8). The prescheduler provides some essential information that helps the user to decide how many processors to choose. The total execution time of the graph(line 9) is the total execution power needed to execute all operations of the graph which is obviously the time needed for a single processor to execute the graph. The time of the critical path(line 10) is the length of the longest path through the graph and is, therefore, the minimum time required to execute the graph with any number of processors. Number of nodes(line 11) is an indication of the size of the graph.

The minimum and maximum number of processors(lines 12 and 13) are estimates for the lower bound and the upper bound for the number of processors to execute the graph within the time of its critical path(the fastest possible time). The minimum number of processors(line 12) is found by dividing the total execution time of the graph by the length of the critical path. This number, however, is just an estimate, and since the work load is not uniformly divided through the time, the actual minimum number may be more than this lower bound. The maximum number of processors(line 13) is the maximum number of nodes that can be executed simultaneously without violating the sequentiality constraints of the graph. In fact, if the user specifies a number larger than this maximum, the additional processors will simply be left idle at all time.

After providing the user with essential graph data, the prescheduler prompts the user to enter the number of processors for this run(line 14). In this example, the user selects 6 processors for the system. In order to allocate the graph among the specified number of processors, the prescheduler analyzes the graph and by adding artificial data dependencies allocates the nodes among different processors. Whenever a dependency is added, the corresponding time step is written on the terminal(lines 15 to 18) which is an indication of how much effort is spent in prescheduling with the given number of processors and also assures the user that the program is running.

After the preallocation is done, the user can study the effects of the communication delays on the total execution time(line 19). In this example, the user enters a communication time to addition time ratio of 10(line 20). The user then enters another value for the ratio(lines 21 and 22) and indicates that no more values for the communication time is to be studyed(line 23).

The final printable results are stored into two ASCII file: the graphlist file contains a description of the graph(line 24), and the schedlist file contains the results of the prescheduling(line 25). These two files are described in more detail in the following sections. Finally, the end of simulation is indicated by the sound of the keyboard bell and writing a message at the user terminal(line 26).

In the next example, the user does not want to use the default execution times and changes them. The use of the help command is also shown in this example.

ALI
enter the name of the program > **pilotejection**
lexical scanner started
postfix code generation started
sorter started
data flow graph generation started
do you want default operation times? y=yes, n=no, h=help>**h**
The default execution times are calculated from the Motorola MC68000 microprocessor user's manual. Execution times are normalized by dividing all execution times by the execution time of the addition operation.
These are the default operation times :

function/ operator	execution time	function/ operator	execution time
addition	1	multiplication	12
division	26	subtraction	1
acos	63	fun1	63
alog	63	integ	32
alog10	63	mapfun	211
asin	63	realpl	59
atan	63	sin	63
cos	63	sqrt	63
exp	63	tan	63
**	138	tanh	63
negation	1	swin	1

If you have used any other functions besides those mentioned above or if you do not want to use the default execution times, you must enter the execution times manually. Again you have to normalize everything to make the addition time equal to one.
Example :
If addition takes 6 CPU cycles and multiplication takes 72 CPU cycles, you have to enter 72/6 = 12 units for multiplication.
All execution times must be rounded to an integer value.
For example, if addition takes 6 CPU cycles and division takes 159 CPU cycles, you can either enter 26 or 27 time units for division operation, but not 6.5

do you want default operation times? y=yes, n=no, h=help>**n**
enter the execution time for cos > **12**
enter the execution time for sin > **12**
enter the execution time for swin > **5**
enter the execution time for integ > **93**
enter the execution time for div > **36**

```
 enter the execution time for **       >    193
 enter the execution time for mult     >     15

 do you want to make any corrections? y=yes, n=no, r=review, h=help> r
 these are your selected execution times :

 function/     execution
 operator       time
   cos            12
   sin            12
   swin            5
   integ          93
   div            36
   **            193
   mult           15
   add             1
   sub             1
   negate          1

 do you want to make any corrections? y=yes, n=no, r=review, h=help> n
prescheduler started
  total execution time    :    845
  time of critical path   :    354
  number of nodes         :     27
  minimum number of processors needed    :    3
  maximum number of processors needed    :    8
  enter number of processors for this run >6
  ---------- time step 1
  ---------- time step 3
  ---------- time step 13
  ---------- time step 15
  ---------- time step 16
  do you want to analyze the effects of comm. delays? y=yes, n=no, h=help>h
    The effects of interprocessor communication delays on the execution
    time can be analyzed for different values of communication
    time. The value must be normalized by dividing it by the execution
    time of the addition operation. For example, if in your system,
    addition takes 6 clock cycles and the interprocessor communication
    time takes 27 clock cycles, a value of 4 or 5 must be entered for
    the communication time to addition time ratio, but not 4.5.
  enter the ratio of communication time to addition time>50
  do you want to analyze the effects of comm. delays? y=yes, n=no, h=help>y
  do you want to try another value?>n
  file graphlist contains the program graph
  file schedlist contains the results of prescheduling
end of simulation
```

Note that the user is prompted to enter the execution times of those operators that are used in the program(seven in this example). Since the execution times must be normalized by dividing them to the execution time of addition operation, addition/subtraction/negation operations are assumed to have unit execution times. The review command, "r", displays the current values of the execution times as modified by the user.

As mentioned before, in order to allocate the same program among different numbers of processors, there is no need to convert the source code into the data flow graph each time. As long as the source program and the execution times are not changed, the user can run the prescheduler alone. This is shown in the next example.

```
scheduler
prescheduler started
  total execution time    :    845
  time of critical path   :    354
  number of nodes         :     27
  minimum number of processors needed     :    3
  maximum number of processors needed     :    8
  enter number of processors for this run >5
  ---------- time step 1
  ---------- time step 3
  ---------- time step 13
  ---------- time step 15
  ---------- time step 16
  ---------- time step 27
  ---------- time step 31
  do you want to analyze the effects of comm. delays? y=yes, n=no, h=help>n
  file graphlist contains the program graph
  file schedlist contains the results of prescheduling
end of simulation
```

In this example, it is assumed that the data flow graph is already generated by executing ALI, and the user wants to allocate the same graph among a different number of processors(five processors in this example).

A-5 Interpreting the Results

The output of the system is in the form of an internally represented graph divided among the processors and two printable files. One of the printable files, graphlist, contains the data flow graph in a printable form. The graphlist file for the pilotejection problem is shown in Figure A-7.

```
[1]  cos
     1 sources ...      th
     1 destinations ...  node 2 @loc2
[2]  mult
     2 sources ...      v    node 1
     1 destinations ...  node 3 @loc1
[3]  sub
     2 sources ...      node 2    va
     1 destinations ...  node 4 @loc1
[4]  integ
     2 sources ...      node 3    0.0
     1 destinations ...  this is an output node
     var defined ...    x
[5]  sin
     1 sources ...      th
     1 destinations ...  node 6 @loc2
[6]  mult
     2 sources ...      v    node 5
     1 destinations ...  node 7 @loc1
[7]  integ
     2 sources ...      node 6    0.0
     1 destinations ...  this is an output node
     var defined ...    y
[8]  mult
     2 sources ...      0.5    r0
     1 destinations ...  node 9 @loc1
[9]  mult
     2 sources ...      node 8    cd
     1 destinations ...  node 10 @loc1
[10] mult
     2 sources ...      node 9    s
     1 destinations ...  node 12 @loc1
[11] **
     2 sources ...      v    2
     1 destinations ...  node 12 @loc2
[12] mult
     2 sources ...      node 10    node 11
     1 destinations ...  node 15 @loc1
     var defined ...    d
[13] sub
     2 sources ...      y1    y
     1 destinations ...  node 14 @loc1
```

Figure A-7 The graphlist file for the pilot ejection problem
(continued on the next page)

```
[14]  swin
      3 sources ...      node 13    0.0    1.0
      2 destinations ...  node 20 @loc1    node 26 @loc1
      var defined ...     yge1
[15]  negate
      1 sources ...      d
      1 destinations ...  node 16 @loc1
[16]  div
      2 sources ...      node 15    mass
      1 destinations ...  node 19 @loc1
[17]  sin
      1 sources ...      th
      1 destinations ...  node 18 @loc2
[18]  mult
      2 sources ...      g    node 17
      1 destinations ...  node 19 @loc2
[19]  sub
      2 sources ...      node 16    node 18
      1 destinations ...  node 20 @loc2
[20]  mult
      2 sources ...      yge1    node 19
      1 destinations ...  node 21 @loc1
[21]  integ
      2 sources ...      node 20    vic
      1 destinations ...  this is an output node
      var defined ...     v
[22]  negate
      1 sources ...      g
      1 destinations ...  node 24 @loc1
[23]  cos
      1 sources ...      th
      1 destinations ...  node 24 @loc2
[24]  mult
      2 sources ...      node 22    node 23
      1 destinations ...  node 25 @loc1
[25]  div
      2 sources ...      node 24    v
      1 destinations ...  node 26 @loc2
[26]  mult
      2 sources ...      yge1    node 25
      1 destinations ...  node 27 @loc1
[27]  integ
      2 sources ...      node 26    thic
      1 destinations ...  this is an output node
      var defined ...     th
```

Figure A-7 (Continued)

All nodes of the graph are shown with their node numbers, their operations, and a list of their source and destination nodes. If the sources of a node are constant values or variables,they are identified by their values and names respectively. If a source is the result of an intermediate operation, the node number where that intermediate result is generated is given. The destinations of each node are identified by the node numbers where the result is send to and the location of the input(i.e. whether it is used as the first or nth operand in that node). Nodes whose results are not used by any other node are identified as the output nodes.

The information in the graphlist file can be used to draw the program graph. For example, in Figure A-7, the first line indicates that the operation in node 1 is cosine. the second line indicates that the node has one input and that input is called "th" in the source program. The third line indicates that the result of this operation is send to node 2 where it is used as the second operand. With this information, the user can draw the first node of the graph(Figure A-8(a)). In the same way, the next three lines in Figure A-7(lines 4 to 6) indicate that node 2 performs a multiplication; one of the inputs is the identifier called "v" in the source program and the other is the result of the operation from node 1; and the result is sent to the first input of node 3(Figure A-8(b)). The next three lines in Figure A-7(lines 7 to 9) indicate that node 3 subtracts the identifier "va" from the output of node 2 and sends the result to the first input of node 4(Figure A-8 (c)).

The next two lines(Figure A-7 lines 10 and 11) indicate that node 4 performs an integration; the first input is the results of the operation performed at node 3 and the second input(the initial condition) is constant 0.0. The next line(line 12) indicates

that the results of node 4 is an output of the graph. the next line(line 13) indicates that this output corresponds to the identifier "x" in the source program (Figure A-8(d)). In fact, these four nodes correspond to the statement

x = integ(v * cos(th) - va, 0.0)

in the source program(Figure A-7 line 19). Following this method, the user can easyly draw the program graph. Figure A-9 shows the complete graph drawn from the information provided by the graphlist file.

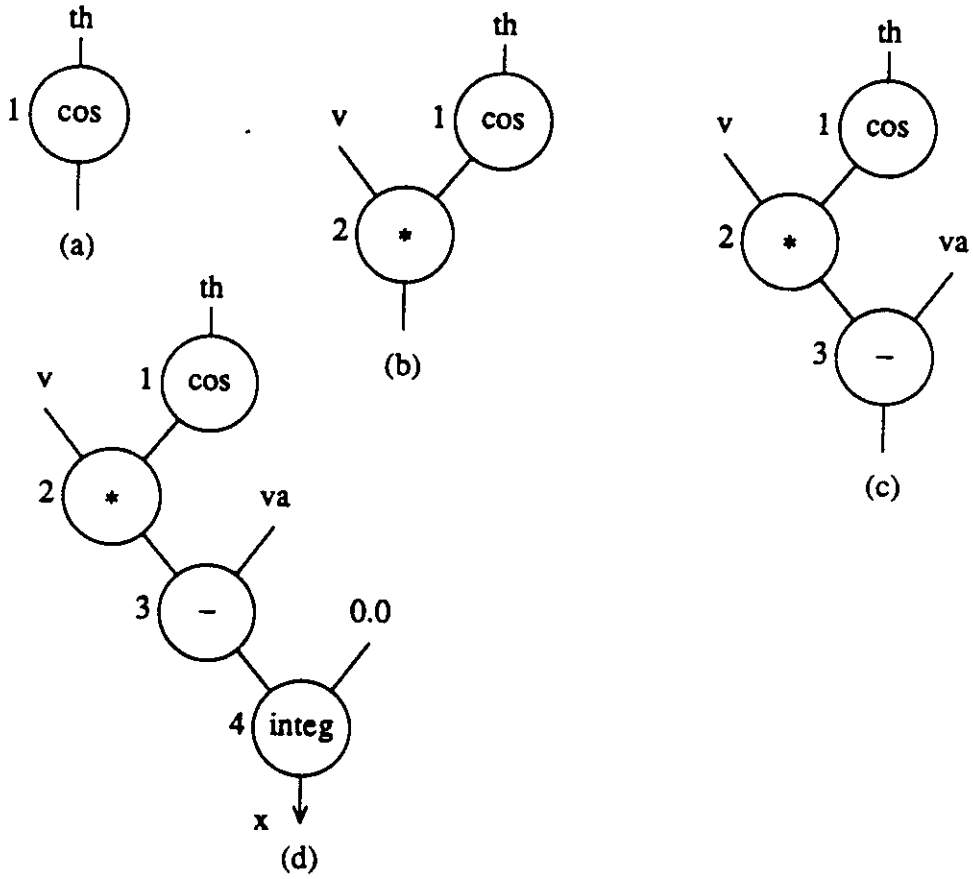


Figure A-8 Drawing the data flow graph from the graphlist file

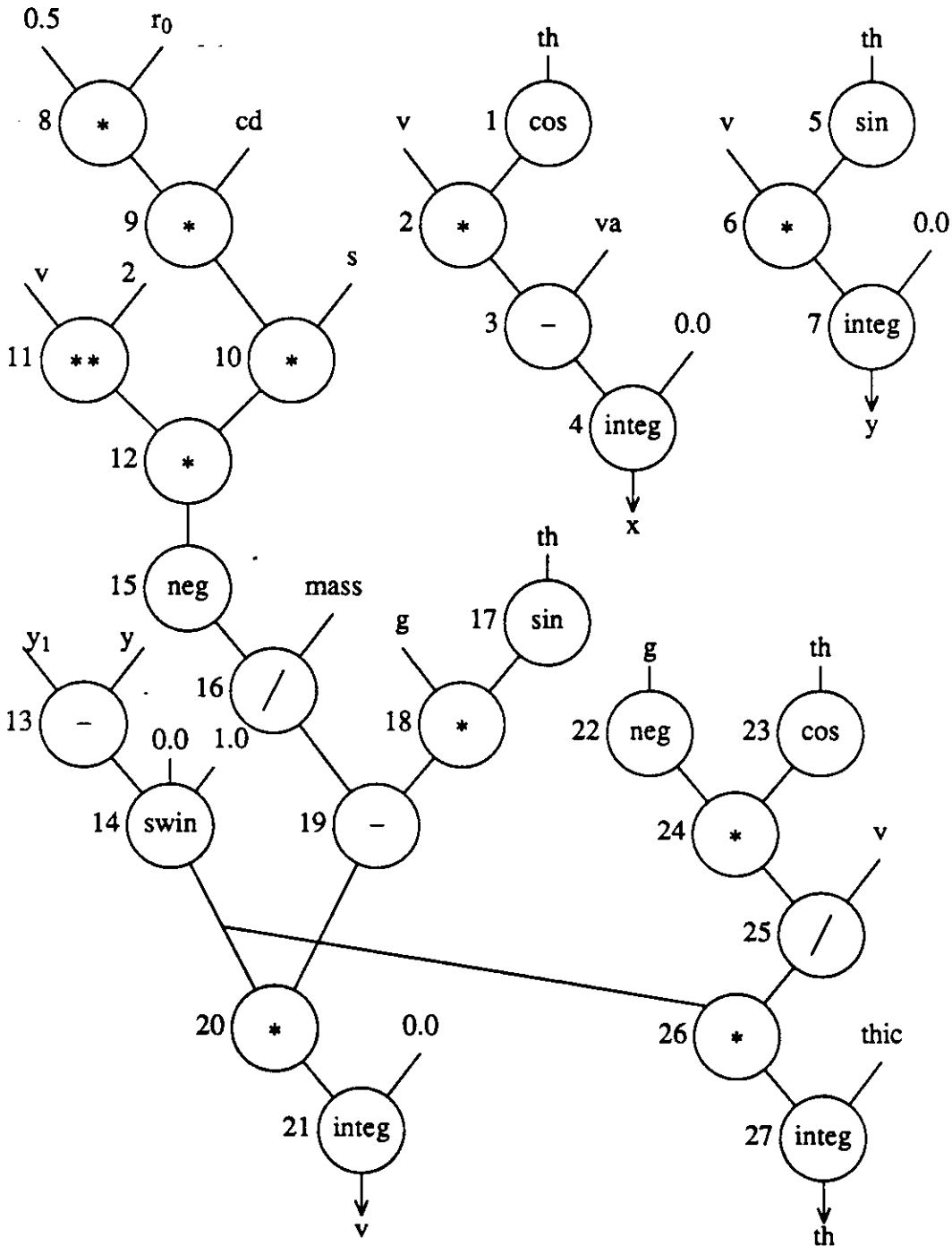


Figure A-9 Data flow graph of the pilot ejection problem

The other printable file, schedlist, contains the results of the scheduling algorithm. Figure A-10 shows the schedlist file for the pilot ejection problem. The first four lines are general information about the graph. For the example in Figure A-10, this information indicates that the graph has 27 nodes; the total execution time of the graph, i.e. the execution time of the graph on one CPU is 696 time units; and the length of the original critical path, i.e. the fastest possible execution time of the graph is 222 time units.

The next 5 lines are specific information for the allocation. For the example in Figure A-10, this information indicates that the number of processors selected for this allocation is 3; the length of the final longest path, i.e. the execution time of the graph on 3 processors is 240 time units; the execution time on 3 processors is 8.11 percent longer than the fastest possible execution time; and in order to do this allocation, 29 artificial data dependencies are added to the graph.

The next 4 lines show the processor loads in time units and percent of time that each processor is idle. The schedlist file also contains two lists that show node to processor assignments. One list is sorted according to the earliest start times of the nodes, and the other is a list of node assignments of each processor. If the user had requested to study the effects of the communication delays, the length of the longest path with the given communication delay(s) and the percent of increase in the total execution time is given at the end of the schedlist file.

General graph information:
number of nodes : 27
total execution time : 696
length of the original critical path : 222

Specific allocation information:
number of processors : 3
length of the final longest path : 240
increase in the length of the longest path : 8.11 percent
to execute the graph with 3 processors 29 data dependencies are added

Processor loads:
cpu 1 total load = 240 percent of idleness = 0.00%
cpu 2 total load = 234 percent of idleness = 2.50%
cpu 3 total load = 222 percent of idleness = 7.50%

Node to processor assignments:

node	cpu	execution time	earliest start time	earliest compl. time	latest exec. time	longest path
11	1	138	0	138	138	*
17	2	63	0	63	89	
23	3	26	0	26	26	*
1	3	63	26	89	89	*
24	2	63	63	126	152	
8	3	12	89	101	101	*
22	3	1	101	102	102	*
2	3	12	102	114	114	*
9	3	12	114	126	126	*
10	3	12	126	138	138	*
13	2	1	126	127	133	
5	2	63	127	190	196	
3	3	1	138	139	151	
12	1	12	138	150	150	*
18	3	12	139	151	195	
15	1	1	150	151	151	*
14	1	1	151	152	152	*
16	3	26	151	177	195	
25	1	12	152	164	164	*
26	1	12	164	176	176	*
27	1	32	176	208	208	*
19	3	1	177	178	196	
20	3	12	178	190	208	
6	2	12	190	202	208	
21	3	32	190	222	240	
7	2	32	202	234	240	
4	1	32	208	240	240	*

Figure A-10 The schedlist file for the pilot ejection problem
(continued on the next page)

Individual CPU allocations:

node	cpu	execution time	earliest start time	earliest compl. time	latest exec. time	longest path
11	1	138	0	138	138	*
12	1	12	138	150	150	*
15	1	1	150	151	151	*
14	1	1	151	152	152	*
25	1	12	152	164	164	*
26	1	12	164	176	176	*
27	1	32	176	208	208	*
4	1	32	208	240	240	*
17	2	63	0	63	89	
24	2	63	63	126	152	
13	2	1	126	127	133	
5	2	63	127	190	196	
6	2	12	190	202	208	
7	2	32	202	234	240	
23	3	26	0	26	26	*
1	3	63	26	89	89	*
8	3	12	89	101	101	*
22	3	1	101	102	102	*
2	3	12	102	114	114	*
9	3	12	114	126	126	*
10	3	12	126	138	138	*
3	3	1	138	139	151	
18	3	12	139	151	195	
16	3	26	151	177	195	
19	3	1	177	178	196	
20	3	12	178	190	208	
21	3	32	190	222	240	

Effects of communication delays:
communication time to addition time ratio: 10
length of the longest path with communication delays: 240
increase in the length of the longest path : 0.0%

communication time to addition time ratio: 100
length of the longest path with communication delays: 249
increase in the length of the longest path : 3.7%

Figure A-10 (Continued)