MULTIPROCESSOR OPTIMIZATIONS : INTERCONNECTION AND TASK ASSIGNMENT

by

Füsun Erdim

B.Sc. in Electrical Eng., Boğaziçi University, 1974
M.Sc. in Electrical Eng., Boğaziçi University, 1975

Submitted to the Institute for Graduate Studies in Science and Engineering in partial fulfillment of the requirements for the degree of

Doctor
of
Philosophy
in
Electrical Engineering

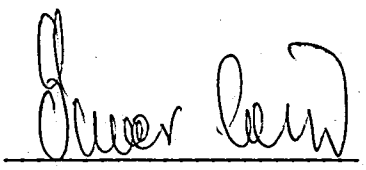
BOĞAZİÇİ UNIVERSITY
1985

MULTIPROCESSOR OPTIMIZATIONS :
INTERCONNECTION
AND
TASK ASSIGNMENT

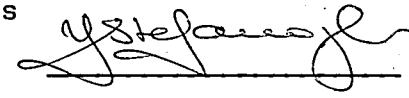
by
Füsun Erdim

APPROVED BY

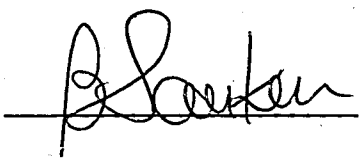
Y.Doç.Dr. Ömer Cerid
(Thesis Supervisor)



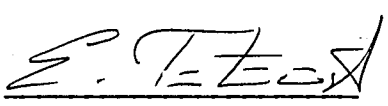
Doç.Dr. Yorgo İstefanopulos
(Co-advisor)



Doç.Dr. Bülent Sankur



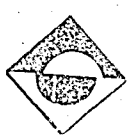
Prof.Dr. Ergür Tütüncüoğlu



DATE OF APPROVAL : June,24 1985

Dedicated To My Parents :

Sabahat and Ali Rıza Erdim



ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to all the past and present members of the Electrical Engineering Department for their valuable contributions during my undergraduate and graduate study.

Particular gratitude is due my advisor Y.Doç.Dr. Ömer Cerid for his continuous support and valuable comments. Also, I would like to acknowledge the support provided by my co-advisor Doç.Dr. Yorgo İstefanopulos who enabled me to use a grant (NATO Grant No.0460/82) for information exchange between Boğaziçi University and Polytechnic of Central London, in relation to this study. Thanks are also due Doç.Dr. Bülent Sankur and Prof.Dr. Ergür Tütüncüoğlu for their valuable suggestions and for participation as jury members in the doctoral committee.

ABSTRACT

MULTIPROCESSOR OPTIMIZATIONS :
INTERCONNECTION
AND
TASK ASSIGNMENT

Effective spreading of the use of multiprocessors, -or distributed processing in general-, and achieving the potential advantages of this new design option require various hardware and software-related problems to be solved.

This study is a research on two basic problem areas, namely the Interconnection and the Task Assignment in Multiprocessors.

Any multiprocessor system that employs more than one processor for a single job must be designed to allow efficient communication between processors, so that the advantages of multiprocessing is not negated by inefficient communication. As the number of processors grows, the interconnection design becomes more crucial as crossbar or fully-connected schemes become impractical. Thus, from a realizability point of view a partially-connected structure is desirable, which, however, in turn, introduces the problem of variable interprocessor distances, complicating the task assignment process. In the first part of this study, PON (Processor Omega Network), a partially-connected, multistage processor network with desirable implementation and communication properties is proposed and evaluated.

In any distributed processing environment, except for identical processors forming a fully-connected network of uniform interprocessor distances, optimal assignment of software modules comprising a task to processors of the network is essential for minimum-time completion of the task and this can be achieved by balancing two conflicting factors ; minimization of interprocessor communication and maximization of load balance of processors.

In addition to the complexities of the previously studied resource limited task assignment environments, partially-connectedness introduces the new interrelated problems of indirect data transfers, availability of intermediate processors, and data routing when more than one path is available between non-adjacent pairs.

Two different performance measures are proposed for the two operation environments considered. The minimum port-to-port time (PTP) criterion produces optimal assignments in single-run environments, whereas the optimum performance in a multi-run operation mode is achieved by minimizing the least re-initiation period (LIP), which is equivalent to maximizing the overlap between successive task executions. The characteristics of the objective functions, the number of constraints, and the precedence relations dictated an algorithmic solution to the assignment problem.

An analytical model is developed to describe the task assignment environment considered in this study, and based on the model components and the proposed objectives, the optimization problems for both environments are formulated. Some possible methods for storage-and-processing efficient representations of hardware and software are investigated and the task assignment algorithm for partially-connected networks (PCTAA) is presented and the methods and modifications to reduce its computational complexity -related to the structure of networks and tasks- are discussed in order to extend its use to analysis of larger systems.

ÖZETÇE

Çoklu-işlemci, ya da daha genel olarak dağıtık bilgiişlem, kullanımının yaygınlaşabilmesi ve bu yeni tasarım seçeneğinin getirdiği olanaklardan tam anlamıyla yararlanılabilmesi için donanım ve yazılıma ilişkin çeşitli sorunların çözümü gerekmektedir.

Bu araştırmada, "Çoklu-işlemcilerde Arabağlantılama ve Görev Atanması" gibi iki temel sorun ele alınmıştır.

Birbiriyle ilişkili yazılım parçacıklarının (modül) oluşturduğu bir görev için birden fazla işlemci kullanan her yapı, işlemciler arası iletişimin yol açabileceği darboğazların çoklu-işlemcililiğin getireceği üstünlükleri yok etmemesi için bilinçli bir biçimde tasarlanmalıdır. Kullanılan işlemci sayısı arttıkça tam-bağlantılı veya "crossbar" türü arabağlantı ağlarının kullanımı mümkün olamadığından, gerçekleştirilebilirlik açısından kısmi-bağlantılı bir yapı istenmekte, bu ise işlemciler arasındaki uzaklıkların farklı olmasına neden olduğundan görev atama işlemini güçleştirmektedir. Bu çalışmanın ilk kısmında kısaca PON (Processor Omega Network) olarak isimlendirilen kısmi-bağlantılı, çok-katlı (multistage) bir işlemci bağlantı ağı önerilmekte ve irdelenmektedir.

Birbirlerine eş uzaklıkla bağlı, tek tür işlemcilerin oluşturduğu tam-bağlantılı dizgelerin dışında kalan bütün dağıtık işlem ortamlarında, görevi oluşturan yazılım modüllerinin işlemcilere en iyi biçimde atanması, görevin en kısa sürede tamamlanması için gereklidir. Bu ise, işlemciler arası iletişimin en aza indirgenmesi ve de iş dağılım dengesinin en üst düzeye çıkarılması gibi birbirleriyle çelişen iki faktörün dengelenmesini zorunlu kılmaktadır.

Kısmi-bağlantılılık, görev atama sorununa doğrudan bağlı olmayan işlemciler arasında dolaylı veri iletimi, ara işlemcilerin iletim için serbest ele geçirilmesi ve birden fazla en kısa yol durumunda veri yönlendirme gibi birbirleriyle ilişkili ek yeni sorunları getirmektedir.

Ele alınan iki ayrı çalışma ortamı için iki ayrı başarım ölçütü önerilmektedir. Tek-seferli (single-run) ortamlarda kısaca PTP (Port-To-Port time) olarak isimlendirilen görev tamamlama süresinin en aza indirilmesiyle en iyi görev ataması sağlanmakta, çok-seferli (multi-run) ortamlarda ise en üstün başarım, görevin ardarda tekrarı sırasında en küçük yeniden başlatma süresi LIP (Least re-Initiation Period) nin en aza indirilmesiyle elde edilmektedir. Amaç işlevlerinin özellikleri, sınırlamaların çokluğu ve modüller arası ilişkiler görev atama probleminin çözümüne algoritmik bir yaklaşım gerektirmektedir.

Burada, bir model geliştirilerek her iki ortam için ilgili eniyileme problemleri tanımlanmakta, donanım ve yazılım gösterimi için çeşitli tanımlama yöntemleri irdelenerek algoritma için en uygun gösterim belirlenmekte ve "kısmi-bağlantılı dizgelerde görev atama algoritması" (PCTAA) sunulmaktadır. Ayrıca, algoritmanın işlemsel karmaşıklığını azaltarak daha büyük dizgelerin analizinde kullanılmasını sağlamak amacıyla arabağlantı ağ yapısı ve görev çizgesinin özelliklerine ilişik gerekli yöntem ve değişiklikler önerilmektedir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF TABLES

1.0
INTRODUCTION

1.1 Distributed Processing : Promises and Problems

The growing need for high-performance, low-cost computing structures in order to match the requirements of compute-bound problems in various fields of application, and the enhancements due to the advent of VLSI technology have given rise to a wide area of research : the architectural and programming issues in distributed processing. The progress in technology permits the availability of low cost and small size processors, making distributed processing economically feasible, and distributed processing brings the well-known advantages such as higher speeds, exploiting parallelism and concurrency in the algorithms, low initial system costs, incremental growth, flexibility and reliability.

Effective spreading of the use of multiprocessors, or distributed processing in general, and achieving the above mentioned potential advantages require two basic sets of problems to be solved : problems related to hardware and problems related to software.

The hardware problem may be viewed at various levels:

1- The processing element level : This is related to the structure and execution mechanism of the processor.

2- The network level : Since the processors will be connected to each other in the form of a network, the interconnection structure must be designed such that:-

   i) each processor should be able to communicate with any other in the network.

   ii) degree of connectedness imposed on the processors, which determines the number of I/O ports, should be kept low and fixed.

   iii) it must be possible to expand the network by adding new processors.

   iv) interprocessor distances, which affect the communication delay, should be kept as low as possible.

v) from a realizability point of view, assuming a large number of processors, the network may not be fully-connected.

vi) for reliability and improved communication density, the availability of alternate paths is preferable.

vii) a regular topology is desirable for ease of implementation.

The software problem may be treated at different levels as well:

1- Language and algorithm design level : Since one of the most important factors for higher speed is the exploitation of parallelism and concurrency in the algorithms, the design of algorithms and languages that reveal explicit parallelism presents an important research issue.

2- Program partitioning level : Development of efficient methods, to analyze and partition a computation task into modules such that minimum dependency is exhibited between the segments in different modules, is essential.

3- Assignment level : Assigning modules that comprise a single task onto processors of a distributed system so as to cooperatively execute the task in minimum time is one of the major concerns of distributed processing.

Undoubtedly, a large amount of research effort is devoted to solving the problems of both categories, which are actually interrelated. The choice of a processing element and the execution mechanism of processors, or the interconnection network, is closely related to module-to-processor assignment, which is affected as well by the effectiveness of program partitioning, the algorithm under consideration and the programming language used to implement the algorithm. Thus, all the elements of hardware and software need to be optimized for efficient utilization of distributed processors.

1.2 General Statement of the Problem

In this study, we will give brief information on the hardware and software problems related to distributed processing, and concentrate on the partially-connected (P-C) interconnection networks of processors and on the optimal assignment of software modules comprising a single program onto processors in a partially-connected network.

Both of the topics of interconnection networks and task assignment have received great interest in the last decade and quite a number of topologies and assignment methods have been proposed. However, the interconnection networks have been studied mainly for SIMD (single instruction multiple data) environments such as that of array processors, where they are used as permutation networks for permuting the data moved between processors and memories; in these networks data alignment in the memories, such as the skewed representation of matrix elements, has been the primary concern. Similarly, for systolic arrays the problem has been that of designing the underlying hardware to match the characteristics of the specific software algorithm. Our interest in interconnection networks is in their use in MIMD (multiple instruction multiple data) environments such as that of true multiprocessors for general-purpose computing systems.

The task assignment problem for multiprocessors has been attacked by some researchers employing various methods for optimal or suboptimal solutions. However, in all the previous work on task assignment, the underlying processor network has been assumed to be fully-connected (F-C) (or bus-connected in some cases), and this is an unrealistic assumption, particularly for networks of large number of processors. Moreover, in most of these studies, except for a few recent ones, the effect of interprocessor communication is disregarded in the efforts to minimize the total run-time of the program. It has been observed that the interprocessor communication due to data passing between non-coresident software modules is responsible for the so-called "saturation effect", which is the degradation in system throughput for increased number of processors, and can only be alleviated using appropriate task assignment strategies. The efficiency of task assignment is important especially in real-time environments, where the task has to be completed within a given deadline.

Even for the fully-connected networks, the task assignment process is highly combinatoric and thus is in the class of NP-complete problems.

Task assignment on partially-connected networks has not been considered previously. Partially-connectedness adds new interrelated problems to the task assignment process:

1- Problem of indirect data transfers between non-adjacent processor pairs, involving intermediate processors,
2- Problem of the availability of intermediate processors on-route that act as repeaters from source to destination, and
3- Problem of data routing when more than one path is available between two indirectly communicating processors.

These problems in addition to the previously studied task assignment environment with constraints such as data dependences (precedence constraints) in the algorithm, non-time dependent constraints such as the number and memory capacity of processors, and real-time constraints such as the input data rate or maximum finish time, create a highly constrained optimization problem.

In this study, we propose a model based on graphical and array representation of the problem, formulate the related discrete optimization problem and present an algorithmic solution for this real-world scenario.

We distinguish between two different environments, the single-run environment, and the multi-run environment where periodic execution of a single task or successive execution of many tasks is considered, and accordingly, propose two different objective functions to be minimized : namely, the port-to-port time (PTP), which is the maximum completion time of the task for the single-run case, and the least re-initiation period (LIP), -which is denoted symbolically as Λ -, for the multi-run environment.

The inherent combinatorial behaviour of the optimal assignment algorithm limits its use in large systems. Methods and modifications related to the structure of networks and tasks are discussed with the aim of reducing the complexity of the proposed algorithm.

1.3 Contributions of this Research

The major contribution of this research is the solution of the task assignment problem in partially-connected processor networks in the presence of real-time constraints.

The task assignment problem involves both hardware and software components. A model is developed to describe the task assignment environment. The software component of the model is assumed to be represented by a single-entry, directed, acyclic graph (the process graph), which exhibits the precedence relations between the modules of a single task. The hardware component, also represented by a graph (the processor graph), is assumed to be a partially-connected network of identical processors with unit distance between adjacent pairs. The possibility of alternate shortest paths between indirectly connected processors and the related problem of optimal path selection are considered.

Two different performance measures are proposed depending on the problem statement. The minimum port-to-port time criterion produces optimal assignments for the single-run or non-loaded operation environment, whereas the optimum performance in a multi-run environment is achieved by minimizing the least re-initiation time, which is equivalent to maximizing the overlap between succesive task executions. Based on the model components and the proposed objectives, the optimization problems for both environments are formulated. An algorithmic solution is presented and methods to reduce its computational complexity are discussed.

Compared to the methods in previous studies, the significant features of the proposed algorithm are: the efficiency of LDF (Load Density Function) generation, which is achieved just by a single scan of the module list, relaxation of the simplifying assumption that the modules receiving data from a common source can start execution simultaneously only after all have received their data, and better feasibility check of the generated assignment which takes into consideration not only direct but indirect precedence relations in the process graph, as well.

Another contribution of this research is in the area of multiprocessor interconnection strategies. A multistage processor network, PON (Processor Omega Network), with regular interstage connections is proposed and evaluated mainly with respect to its communication properties. PON has low average path length, resonable processor reachability and linear implementation costs compared to multistage switching networks and other cube-type multistage processor networks. It provides various row-column alignment patterns for the same size, is incrementally expandable, homogeneous, and requires a fixed number of I/O ports per processor regardless of the size of the network. It also improves reliability and work distribution due to the presence of alternate paths.

1.4 Outline of the Dissertation

The subject material of this dissertation is treated in eight chapters.

Chapter 1 introduces the subject of the study by presenting the promises and problems of distributed processing environment.

Chapter 2 is devoted to the hardware environment of distributed processing, where the possibilities regarding the structural complexity and execution mechanisms of processing elements, the organizations of distributed systems, and the interconnection networks are briefly surveyed. The multistage switching networks (MSN) are presented as an introduction to processor multistage networks (PMN). A regular configuration, the Processor Omega Network (PON), is introduced and evaluated against some other structures.

The software environment of distributed processing is introduced in Chapter 3. The task assignment problem is stated in general, and in partially-connected processor networks in particular. The related research on task assignment, file allocation and scheduling, and the solution techniques are presented. The proposed solution procedure is outlined.

Chapter 4 develops an analytic model for the task assignment environment, introduces the performance measures for single-run and multi-run environments, and presents a mathematical formulation, both for fully-connected and partially-connected networks, as a discrete optimization problem.

In Chapter 5, the methods for storage-and-processing-efficient representations of software and hardware components of the assignment process are investigated and the actual storage representations are determined in preparation for the algorithmic solution presented in Chapter 6.

Chapter 6 presents PCTAA (task assignment algorithm for partially-connected networks), and discusses each of its steps, the formal algorithms of which are provided in the Appendices. The use of the proposed algorithm is demonstrated by examples and its performance characteristics are evaluated.

In Chapter 7, some methods and possible modifications in the algorithm for reducing the computational complexity of the assignment process, in order to enhance its use in the analysis of larger systems are discussed.

Chapter 8 concludes the subject of the dissertation by summarizing the achievements and some possibilities for further research in distributed processing.

2.0
THE HARDWARE ENVIRONMENT FOR DISTRIBUTED PROCESSING

In this chapter, we briefly review the hardware issues related to the design and efficient utilization of distributed processing systems.

Any multiprocessor system that employs more than one processor for a single job must be designed to allow efficient communication between processors, or between processors and memories, so that the advantages of multiprocessing are not negated by inefficient communication. As the number of processors grows, the interconnection design becomes more crucial as crossbar or fully-connected schemes become impractical.

Another point of interest is the organization of the computing system for which the interconnection problem is considered. There are basically two major computer organizations for distributed processing, namely SIMD and MIMD, [FLYN72] , and depending on the desired parallelism in data and instruction handling, either one is employed with its particular expectations for the performance of an interconnection network.

The structural complexity of processing elements varies according to the application environment of high-performance computing systems. In addition, for higher degrees of parallelism and concurrency, new execution mechanisms have emerged as opposed to that of conventional processors and this forms the basis for the research on the so-called non-von-Neumann architectures and languages [BACK78] .

In the following sections, the structure and execution mechanisms of processing elements, the organization of computing systems, the topic of interconnection networks and specifically the Multistage Switching Interconnection Networks will be treated separately. In Sections 2.5 and 2.6 we consider processor interconnection strategies, introduce Processor Multistage Networks, specifically PON (Processor Omega Network), and finish the discussion on hardware issues with the proposed hardware configuration.

2.1 The Structure and Execution Principle of Processing Elements

Depending on the desired application environment, the processing elements (PE) used in distributed processing may consist of :

i) Simple arithmetic-logic units (ALU), equipped with a data transfer register (DTR) and with no control capability, or

ii) Complete central-processing-units (CPU), possibly with some local memory, or

iii) Processor-Local memory-I/O ports ensemble, or

iv) Integrated computing and I/O processors.

For example, type (i) may form the basic PE in an array processor while a processor network will employ types (ii)-(iv) as the basic unit. In the latter sections of this study, we are concerned with PE's of the (iii)rd category and in this case the PE will simply be referred to as the "processor".

Most of the conventional computers are based on the von-Neumann principle, where the CPU - connected to the memory via the so-called "von-Neumann bottleneck" - sequentially executes the program instructions stored in the memory, the operation sequence being determined by the contents of a program counter. Thus, at any one time only one instruction is being executed : this execution mechanism is known as the "control-flow". The important point is that program execution is based on sequenced memory updates causing an enormous traffic of information through the bottleneck, much of which is not actually significant data, but addresses used to locate the data.

In search of increased parallelism and concurrency in program execution, recently computer designs based on non-von-Neumann principles are attracting increasing interest as an alternative to conventional architectures. The basis of such designs are the "data-flow" and "demand-flow" (or "demand-driven") execution mechanisms.

In a data flow computer [DENN79] , an instruction is ready for execution, or "fired" , when all its required operands are available. There is no concept of control flow and thus there is no program counter. A data flow program is represented by a graph where the instruction nodes (or "actors") are connected by arcs along which data tokens are passed between actors. The instructions themselves are represented by "activity templates" (FIGURE 2.1.1) which are used in forming "operation packets" for execution of the form

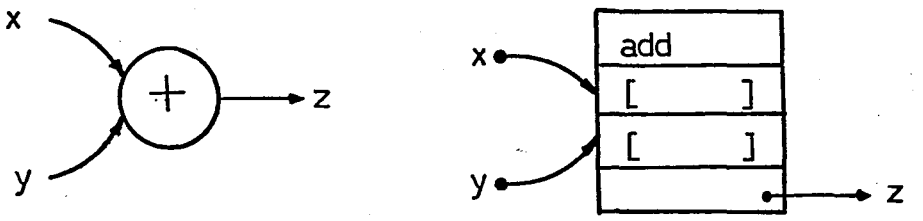
operation packet: <opcode, operands, destinations>

and a "result packet",

result packet: <value, destination>

for each destination field of the template. FIGURE 2.1.2 shows the basic execution mechanism of data flow principle aptly called a "circular pipeline" where the activity is controlled by the flow of information packets traversing the ring in counterclockwise direction. When an instruction is "ready", having received all operand and acknowledge packets, the Update Unit which has updated the corresponding template in the Activity Store upon arrival of each result packet enters its address in the Ready Instruction Queue, - a FIFO stack - , and the Fetch Unit, which scans the queue, fetches the next template in line from the store, forms it into an operation packet and passes it on to the Operation Unit. The Operation Unit performs the operation specified by the operation code and generates and forwards result packets to the Update Unit. Thus, a number of packets may be flowing simultaneously in different parts of the ring such that at any one time every active unit may be handling a different instruction and this brings the concurrency advantage of data flow principle. The configuration in FIGURE 2.1.2 might actually be considered as a data-flow PE for a data-flow multiprocessor if many such PE's are interfaced to a connection network through their Update Units.

In demand-driven execution, the requirement for a result triggers the operation that will generate it and the sequence of instruction execution is determined by the flow of demand [TREL82]. A program is represented as an expression consisting of nested applications, each composed of an



a) A Data Flow Instruction (Actor)

b) An Activity Template

FIGURE 2.1.1 Elements of Data Flow Execution

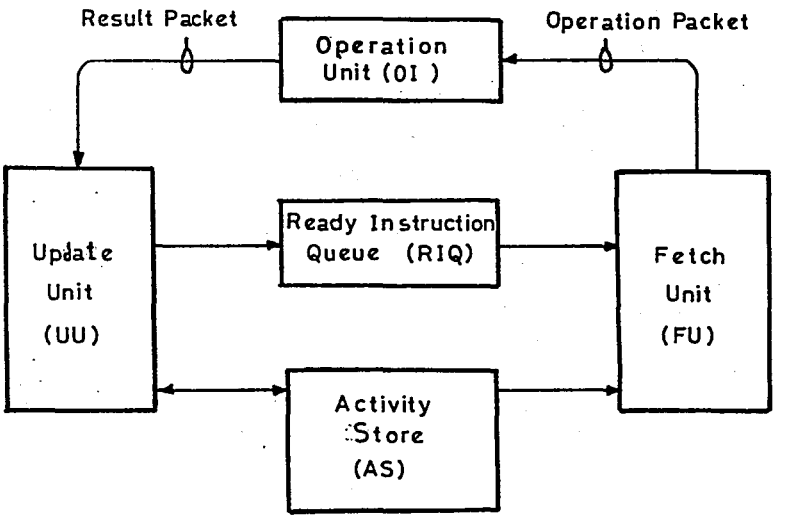


FIGURE 2.1.2 Data Flow Circular Pipeline

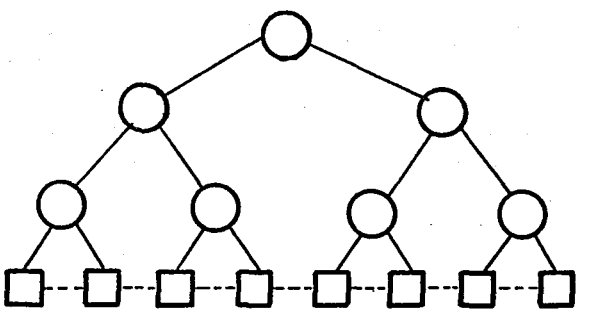


FIGURE 2.1.3 Basic Tree Structure of a Reduction Machine

operator and an operand, and sequences. As an example, the expression <6,(+:<2,4>)> is a sequence of two elements: number 6 and an innermost application composed of + operator and a sequence of numbers 2,4 as the operand. Since the first requirement is for the result of the program, demand flows through subexpressions until subexpressions consisting of primitive operations are encountered and then reduction proceeds in the opposite direction as the successive replacement of reducible subexpressions, until the result value of the original expression is reached. This sequence of reductions is referred to as the "outermost" or "lazy evaluation" rule.

Computer desingns based on reduction are called "reduction machines" and are most conveniently configured as tree structures (FIGURE 2.1.3), where the processing elements, or cells, are close to type (i), i.e. with limited processing and memory capability. There also exist reduction computers which are based on a data-driven mechanism and employ the "innermost" or "eager evaluation" rule [MAG079].

The control, data, and demand driven execution principles are not distinct and it is possible that different combinations of the three mechanisms are employed within one system.

In principle, we are concerned with the conventional (control-flow) processors, although actually a data-flow concept is inherent in the execution of a single program by multiprocessors where each dependent module of the program is executed only when the required data is available from its predecessors residing on other processors.

2.2 The Organization of Distributed Systems

According to Flynn [FLYN72] who tried to classify computing systems in terms of parallelism within the instruction stream and the data stream, four broad classifications of machine organizations are possible, listed as follows :

1- SISD (single-instruction stream-single-data stream) organization which represents most conventional computing equipment available today.

2- SIMD (single-instruction stream-multiple-data stream) organization where single instruction stream causes parallel execution of incoming data to the system, which includes most array processors, systolic arrays and pipelined processors.

3- MISD (multiple-instruction stream single-data stream) organization which represents some specialized systems.

4- MIMD (multiple-instruction stream multiple-data stream) organization referred to as "multiprocessors", including true multiprocessors where several autonomous processors cooperate in the execution of a program, and shared resource multiprocessors composed of skeleton processors sharing the resources.

Naturally, the work done in distributed processing involves either the SIMD or MIMD type of organizations. Although our main concern is multiprocessors, and not simply an array of processors, nevertheless we will glance briefly at SIMD structures as well, since most of the research on interconnection networks has originally emerged and been carried out for SIMD systems.

The basic machine organization for an SIMD computer is shown in FIGURE 2.2.1 [KUCK77]. Here, a control processor decodes instructions, executes sequential parts of the program and for parallel executable program segments, it controls the ALU's and routes the parallel data between ALU's and memories by controlling the switches in two interconnection networks, called the alignment networks. The alignment networks must be able to

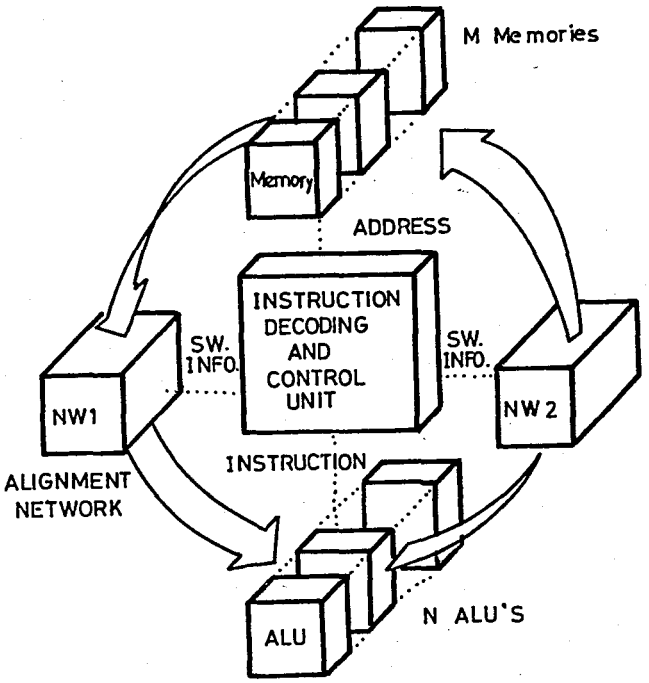


FIGURE 2.2.1 Basic SIMD Machine Organization

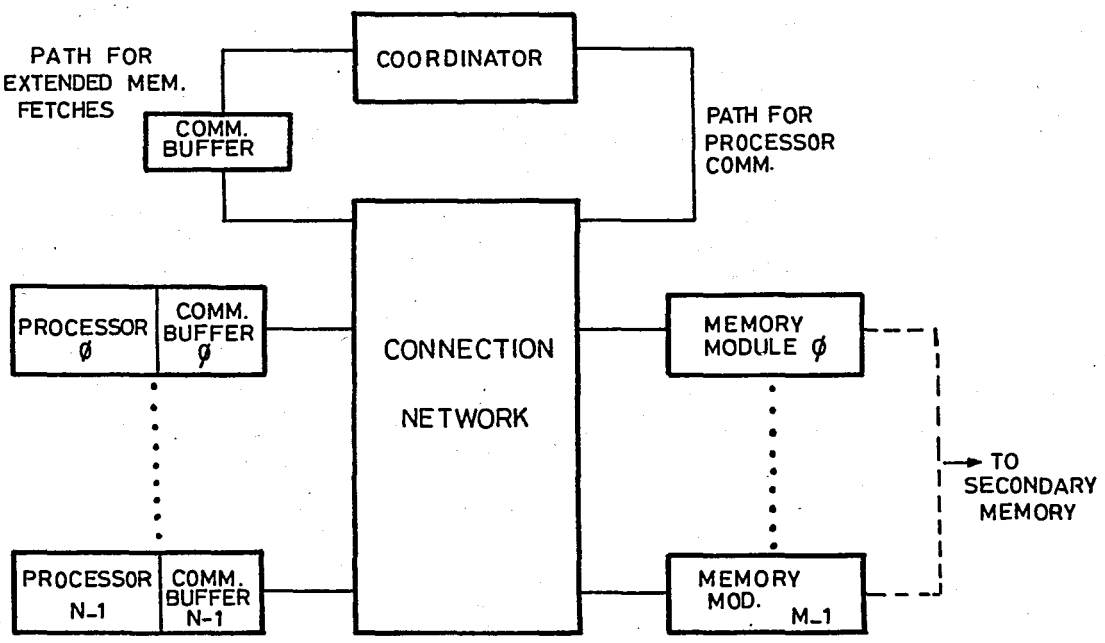


FIGURE 2.2.2 A Typical Configuration of MIMD Architecture (P-M)

handle the indexing patterns found in programs, for example, the uniform shift of 3 necessary in A(I)+A(I+3), and scramble/unscramble the data for memory access. Data alignment in memories, efficiency of the alignment networks, and the efficiency of the control algorithm on these networks to reduce the reconfiguration overhead are the major problems in the design of an SIMD machine.

A typical configuration of an MIMD computer is shown in FIGURE 2.2.2. [LUND80]. Here, the processors are autonomous with individual processing and memory access capability and execute segments of a computation task. A coordinator implements the synchronization of processes and smooths out the execution sequence. The design of an efficient interconnection network seems to be the major problem. This configuration, where a bidirectional network is positioned between the processors and memory modules, is referred to as the processor-to-memory (P-M) approach and provides to the processors the ability to share large blocks of data and to vary the amount of memory used.

An alternative MIMD structure is to equip each processor with local memory in order to achieve fast memory access and let the processors communicate with each other via a unidirectional interconnection network positioned between the processors (FIGURE 2.2.3.). This is known as the processor-to-processor (P-P) approach. Here, the processors cooperatively execute the partitioned and assigned segments of a computation task, and thus, program partitioning and assignment to processors as well as the choice of an efficient interconnection network present the major problems to be solved.

A survey on multiprocessor organizations appears in [ENSL77].

Let us call the organizations used in non-von-Neumann machines as CICD (concurrent-instruction-concurrent-data) to differentiate from the control flow multiprocessors.

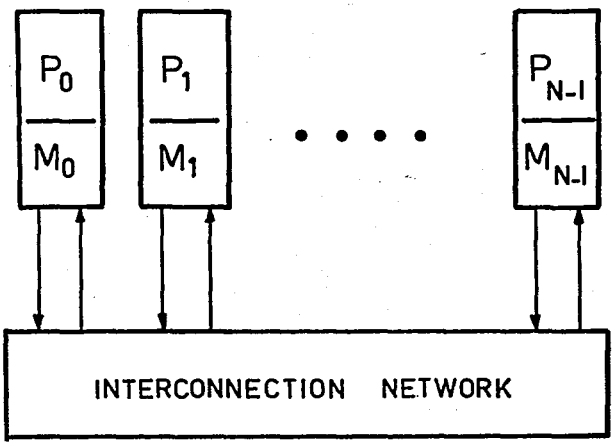


FIGURE 2.2.3 An Alternative MIMD Structure (P-P)

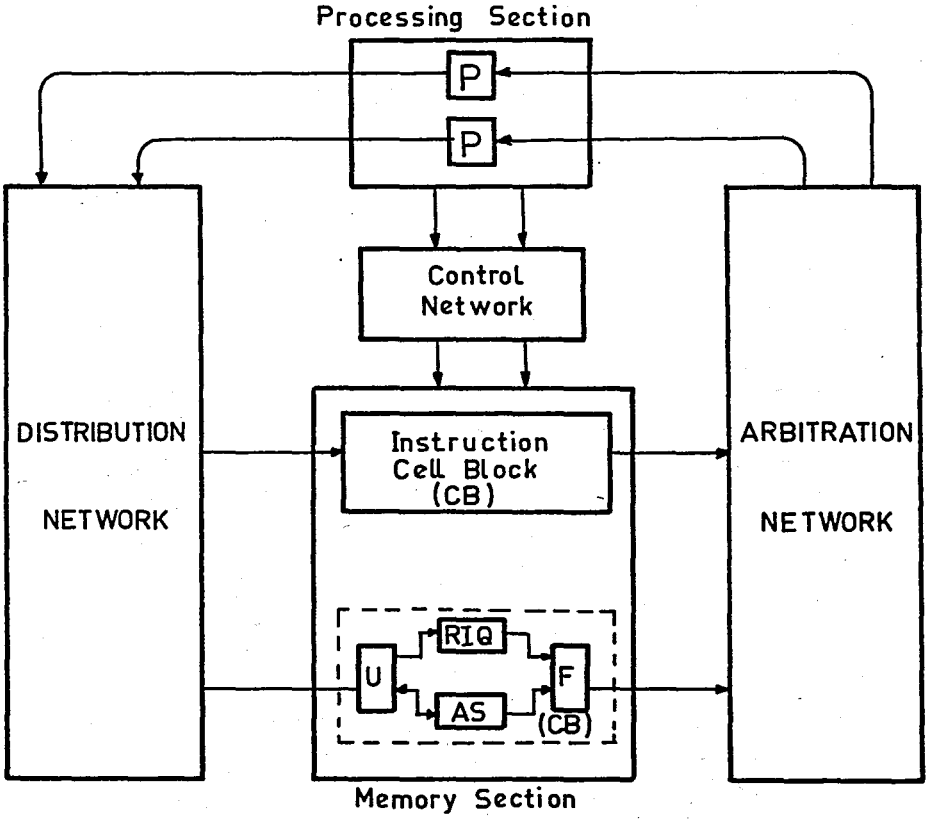


FIGURE 2.2.4 MIT Data flow Computer

As we have mentioned previously, the basic structure for reduction computers is in the form of a tree where, possibly, the expressions stored in the leaf cells will be reduced as they move up the tree.

For a data flow computer, one possible organization might be to form the cells shown in FIGURE 2.1.2. into a network, as previously stated. Such a multiprocessor network obviously posseses the problems of efficient interconnection and software assignment, i.e. partitioning of the task into segments to be stored in the local Activity Store as activity templates. Two other basic approaches to a data flow multiprocessor in experiment stage are the Manchester Ring [WATS82], which in very simple terms is the basic mechanism (FIGURE 2.1.2.) enriched with multiple Operation Units, and the MIT architecture [DENN79], illustrated in the simplified diagram of FIGURE 2.2.4. Here, the distribution network distributes result packets to Cell-blocks (CB) according to template addresses and an arbitration network routes operation packets to processors according to the availability of processors and if the processors are not identical according to the operation-codes as well. These configurations are proposed in order to avoid the task assignment process since all processors are equally apart from the store. However, we are doubtful about the performance and feasibility of such networks due to increased interconnection costs and port-to-port distances. Also, having studied the task assignment problem, we note that the ratio of module processing time to communication time should be high to achieve the speed benefits of multiprocessors and therefore, it is questionable if the concurrency advantage will compensate for the time lost in interprocessor communication, with communication involving complete operation/result packets and the module processing time being that of instructions. A constructive criticism on data flow computers appears in [GAJS82].

2.3 Interconnection Networks

The selection of an interconnection network depends on the organization of the distributed system under consideration. In an MIMD environment, the function of an interconnection network is to provide direct or indirect links between processing elements for the sole purpose of interprocessor communication, whereas in an SIMD environment, data permuting capability of an interconnection network is desired as well. Thus, the application required determines the parameters of the network.

From a practical design viewpoint,four basic parameters are identified in selecting the architecture of an interconnection network (FIGURE 2.3.1); namely the communication mode, the control strategy, the switching methodology and the network topology. If we view a typical interconnection network as consisting of a number of switching elements and interconnecting links, the control strategy determines whether the switching elements are set by a common control unit or by the individual switching elements. Circuit switching, packet switching or integration of the two can be selected depending on the transmitted data volumes in the application. A dynamic topology permits reconfiguration of interprocessor links by controlling the switching elements, whereas the links in a static topology remain passive and dedicated.

SIMD organization is best suited to centrally controlled circuit-switched synchronous networks of dynamic topology and MIMD organization mostly favours decentrally controlled packet-switched asynchronous networks of either topological category.

A detailed treatment of interconnection networks appears in [ANDE75] and [FENG81].

We want to mention one important class of the dynamic topology, the Multistage Switching Networks, that are widely used in both SIMD and MIMD environments. They are discussed in the next section.

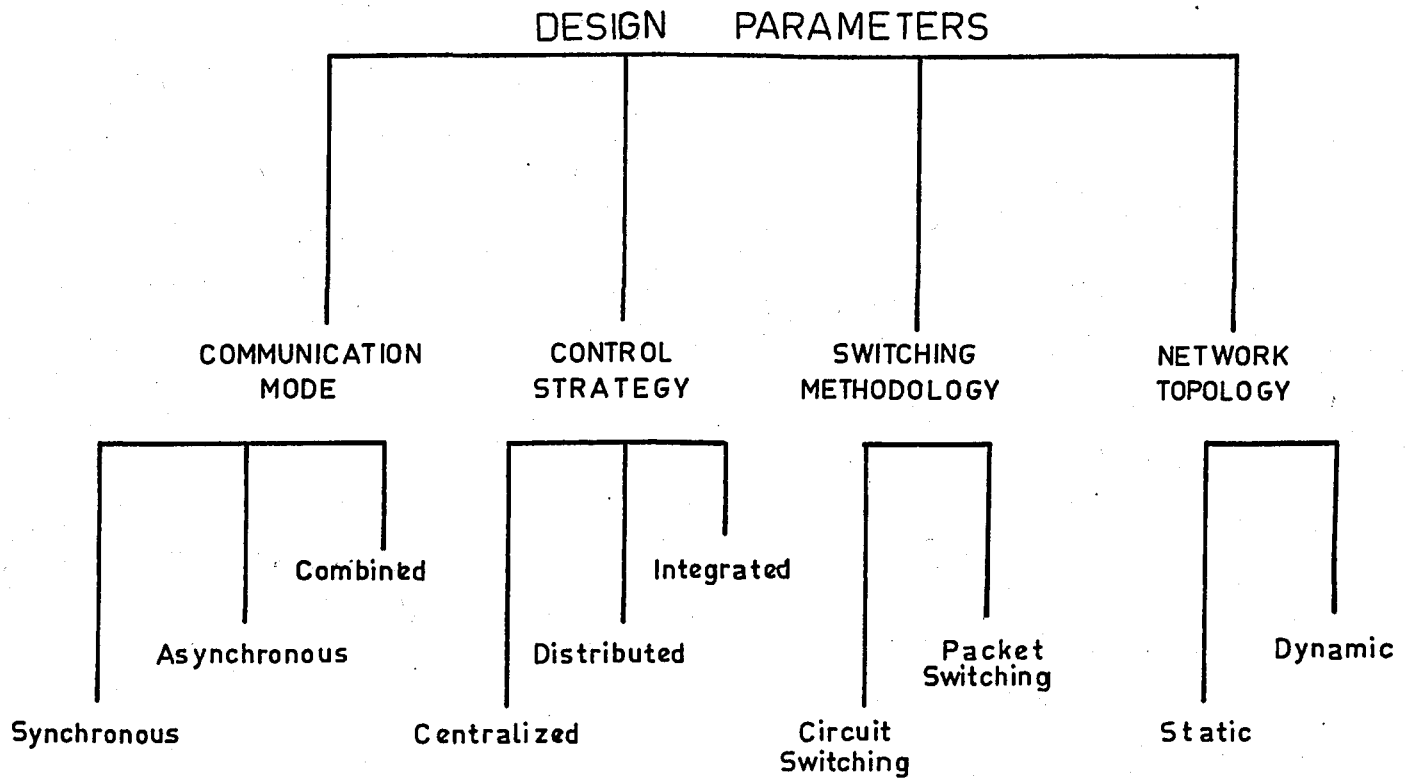


FIGURE 2.3.1 Basic Design Parameters of Interconnection Networks

2.4 Multistage Switching Networks (MSN)

This class of dynamic networks ,developed following the work of Benes [BENE65] on telephone switching, have found wide application in distributed processing, especially in SIMD environments.

The binary n-cube multistage switching network (MSN) [SIEG81] with $N=2^n$ inputs and N outputs to connect N x N elements (processor/memory) is defined to consist of $\log_2 N$ switching stages, with N/2 switches per stage (FIGURE 2.4.1, [PEAS77]) where, each 2x2 switching element has two states, straight or exchange, although some systems permit a broadcast state as well. Thus, it has a cost of $(N/2)\log_2 N$ switches and $o(\log_2 N)$ end-to-end communication delay.

The connections between stages are based on the n cube interconnection functions [SIEG77] defined by

$$C_i(P_{n-1}..P_{i+1}P_iP_{i-1}..P_o)=P_{n-1}..P_{i+1}\bar{P}_iP_{i-1}..P_o$$

where $P_{n-1}...P_o$ is the binary representation of element addresses (or equivalently labels of input/output lines) and $\bar{P}_i$ denotes complement of P_i for $0\leq i<n$. That is, C_i applied to i th stage pairs input/output lines that differ in i th bit position and if the element addresses are considered as the corners of an n-dimensional cube, this network connects each element to its n neighbours.

The importance of MSN's for SIMD computers is that they can be used as permutation networks operating on the input data, by controlling the switch settings either in stages or individually. However, being unable to realize arbitrary permutations, such as the bit-reverse permutation, in a single pass through the network, multiple passes are permitted, where 2 - 3 passes are found to be necessary and sufficient to realize any permutation and 6 passes sufficient to generate any connection of the input to the output lines [PEAS77], [PARK80] in these blocking type networks, meaning that in simultaneous connection requests to a common output some inputs need to be deferred.

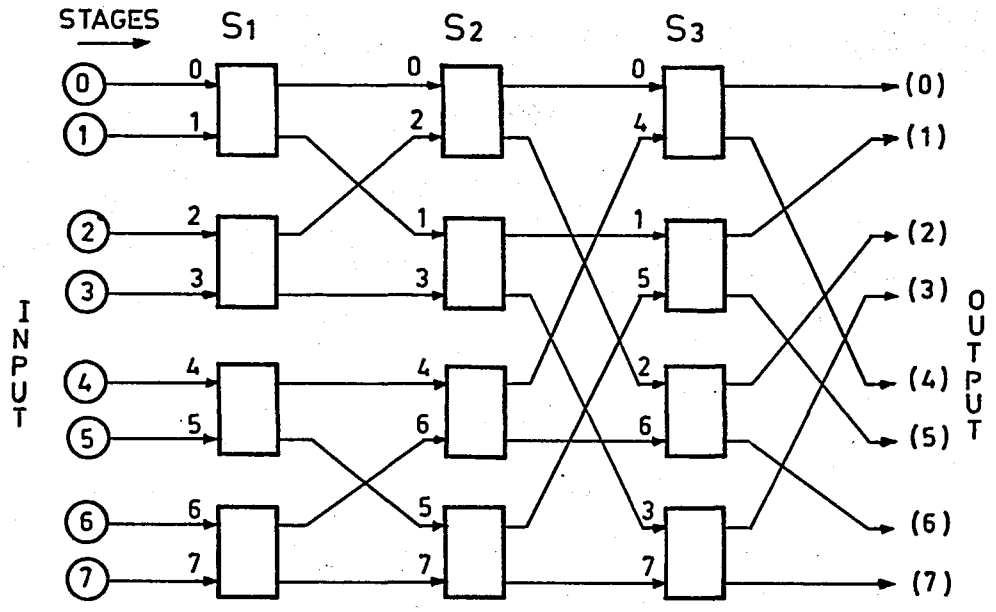


FIGURE 2.4.1 A Multistage Switching Network (Indirect Binary 3-Cube)

Various cube-type MSN's proposed for SIMD interconnections, such as the Indirect Binary n-cube network [PEAS77], the Omega Network [LAWR75], the Baseline network [WU 80], the Flip network [BATC76] and the regular SW Banyan network (S=F=2) [LIPO77] actually are all topologically equivalent, but not necessarily equally efficient, the difference between them lying mainly in the hardwired connections between the stages [SIEG77], [SIEG79], [PARK80], [WU 80].

MSN's are used in MIMD environments as well, where, depending on the configuration, they are positioned between processors or between processors and memories and provide a uniform path length of $o(\log_2 N)$.

In the next section, we propose a class of processor multistage networks (PMN) which have a close topological resemblance to multistage switching networks.

2.5 Processor Interconnection Networks

There is a vast amount of literature on processor interconnection strategies and very frequently many new schemes are proposed. We confine our discussion to three basic schemes of interest. These are the ring, the tree and multistage connection networks, and parameters of interest are expandability of network, local degree of connectedness and regularity of connections, path length between processors, reliability and physical realizability.

Ring structures permit expansion, provide fixed local degree per processor and are physically realizable, but path length between processors increases linearly with every new processor introduced and there is no mechanism for fault-tolerance, except for the modified versions such as the chordal ring [ARDE81].

Tree networks [HORO81] provide fixed number of processor connections, are physically realizable, reliable if augmented using extra ring connections connecting the nodes at the same level [GOOD81], and the path length depends on the depth of the tree which is related to the number of processors and branching degree employed. The disadvantage of tree networks seems to be the high traffic load concentrating at the root.

The topology of MSN's, mentioned in the previous section, provides fixed and regular connections, realizability, reliability due to alternate paths between the nodes and permits expansion at the cost of increasing the uniform path length between the processors attached to the two ends.

The processor multistage networks (PMN) are based on the topology of MSN's such that the switching elements are replaced by processors and the two ends of the network are connected to form a cylindical structure. Each PMN is actually a virtual tree network that rolls around the cylinder at endless depth. PMN's are less costly than MSN's in the sense that the switching elements are eliminated, but at the expense of variable interprocessor distances, which necessitate proper task assignment, a topic to be discussed in the next chapter.

PMN's have attracted a number of researchers, but so far the interstage connection pattern chosen is that of Indirect r-ary n-cube [BURT81], which forces the number of processors in the network to "facets" of $N=nr^n$ for regularity of connections, where $n=2^i$ for some i [WINT83]. Then, for a 2-ary n-cube the increments on network size grow as 2, 8, 64, 2048,.. etc., and when n is any integer, incrementing the network size can be achieved by doubling the height of the cylinder and increasing the number of stages by one. The replication of facets is possible for some intermediate N, for example N=8 can be duplicated for a network of N=16, but with an increase in the average path length.

In Section 2.6 we introduce another processor multistage network with better properties compared to the previously studied multistage networks of processors.

2.6 Processor Omega Networks (PON)

We now propose a new class of PMN's that permits reasonable incremental expandability (in increments as low as 4) and employs a fixed pattern for interstage connections regardless of the size and alignment of the network. We called this network the Processor Omega Network (PON) due to its resemblance to one of the multistage switching networks, the Omega Network of Lawrie [LAWR75] which is illustrated in FIGURE 2.6.1 for interconnection of 8 processors. Here, the interstage connections are based on the shuffle function defined by

$$S(x) = (2x + \lfloor 2x/N \rfloor) \bmod N$$

where x is the binary representation of index of an input line, $N=2^n$ for some n, and $\lfloor x \rfloor \leq x$. Thus the shuffle permutation corresponds to left-rotate of the index bits.

We represent an N-element PON as N-processors arranged in a matrix (i,j) of r rows and c columns connected by the shuffle interconnection as illustrated in FIGURE 2.6.2 and the index of each processor is given by the single index

$$P = i + jr$$

with $0 \leq i \leq r-1$, $0 \leq j \leq c-1$ and $0 \leq P \leq N-1$. The last column of the figure coincides with the first. Each processor in j th column is connected to two processors in column (j+1)mod c on its right, and to two processors in column (c+j-1)mod c on its left. Each processor in the i th row is connected to two processors in rows

$$r/2\ (i \bmod r/2)\ ;\ \text{up}$$

and

$$r/2\ (i \bmod r/2)+1\ ;\ \text{down}$$

on its right and to two processors on its left in rows

$$\lfloor i/(r/2) \rfloor\ ;\ \text{up}$$

and

$$\lfloor i/(r/2) \rfloor + r/2\ ;\ \text{down}$$

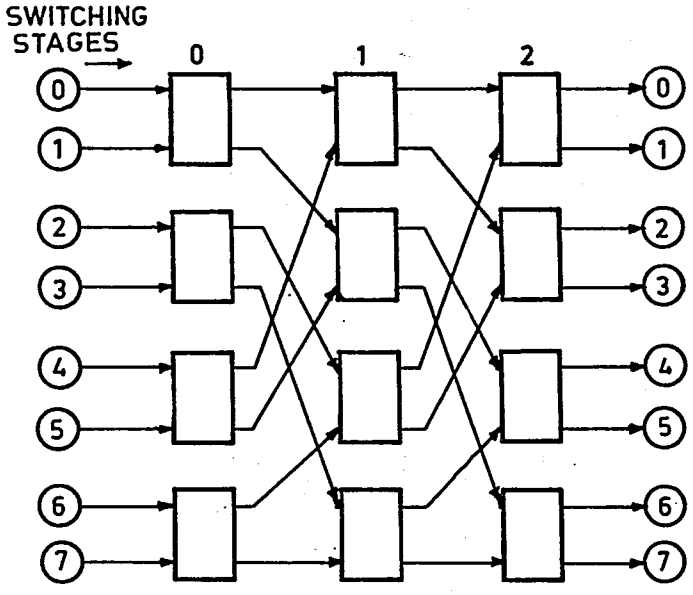


FIGURE 2.6.1 An (MSN) Omega Network (N = 8)

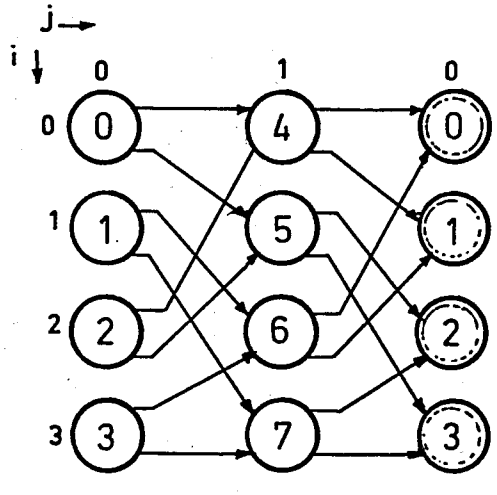


FIGURE 2.6.2 A (PON) Processor Omega Network (N = 8)

The connection only requires that r=2k for some k. However, we can assume that $r=2^k$ at most so that $N=c2^k=2^n$ for some n. The main advantage of this configuration is homogeneity, that is, the view of any processor of the rest of the network is the same, a fact that can easily be proved by manipulating row and column indexes, and will permit a single type of system software for all the processors in a multiprocessor environment.

To have a better appreciation for PON, we derive expressions for some of its deterministic properties and compare with those of Omega MSN.

First, the average path length l in an N=rc unidirectional PON is derived as

$$l_{PON} = \frac{1}{N} \sum_{i=0}^{d_{max}} n_i \cdot i$$

$$= \frac{1}{N} \left[\sum_{i=0}^{\log_2 r - 1} 2^i \cdot i + \sum_{i=\log_2 r}^{c-1} r \cdot i + \sum_{i=0}^{\log_2 r - 1} (r - 2^i)(c+i) \right]$$

and,

$$l_{PON} = (c-3)/2 + \log_2 r + 1/r \qquad (2.6.1)$$

where n_i is the number of processors reached at a path length of i, d_{max} is the maximum path traversed from any one node to reach all the others, and use is made of the identities

$$\sum_{i=0}^{M} 2^i = 2^{M+1} - 1 \quad , \text{ and } \quad \sum_{i=0}^{M} i = M(M-1)/2$$

For the special case, where $r= 2^c$ and $N= c\ 2^c$,

$$d_{max} = 2c-1$$

and l_{PON} simplifies to

$$l_{PON} = 3/2\ (c-1) + 2^{-c}$$

The path length in MSN with $N= 2^n$ is fixed and is given by,

$$l_{MSN} = \log_2 N - 1 = n-1 \qquad (2.6.2)$$

TABLE 2.6.1 PON versus MSN for $1 \leqslant n \leqslant 6$, $N = 2^n = rc$

N	n	k	$r=2^k$	$c=2^{n-k}$	n_{SW}	l_{MSN}	l_{PON}
2	1	1	2	1	1	0	1
4	2	1	2	2	4	1	1
8	3	1	2	4	12	2	2
8	3	2	4	2	12	2	1.75
16	4	1	2	8	32	3	4
16	4	2	4	4	32	3	2.75
16	4	3	8	2	32	3	2.62
32	5	1	2	16	80	4	8
32	5	2	4	8	80	4	4.75
32	5	3	8	4	80	4	3.62
32	5	4	16	2	80	4	3.56
64	6	5	32	2	192	5	4.53

If we relate MSN and PON for the same $N = 2^n = rc = 2^k 2^{n-k}$ (2.6.1) can be rewritten as

$$l_{PON} = 2^{-k}(2^{n-1} + 1) + k - 3/2 \qquad (2.6.3)$$

TABLE 2.6.1 gives l_{PON}, l_{MSN} and n_{SW}, the number of switching elements in MSN, for some parameters including N=64, where

$$n_{SW} = N/2 \log_2 N = n\, 2^{n-1}$$

We note a few points in TABLE 2.6.1. First, the increments on N are smaller than the increments of binary n-cube PMN's. Second, we have a wider choice of network alignments indicated by the column for k (note that for N=64, just one representative is shown). For example, for N=16 the alignment r=c=4 corresponds to a replicated n-cube while the alignment r=8 and c=2, only permitted in PON, provides lowest average path lengths.

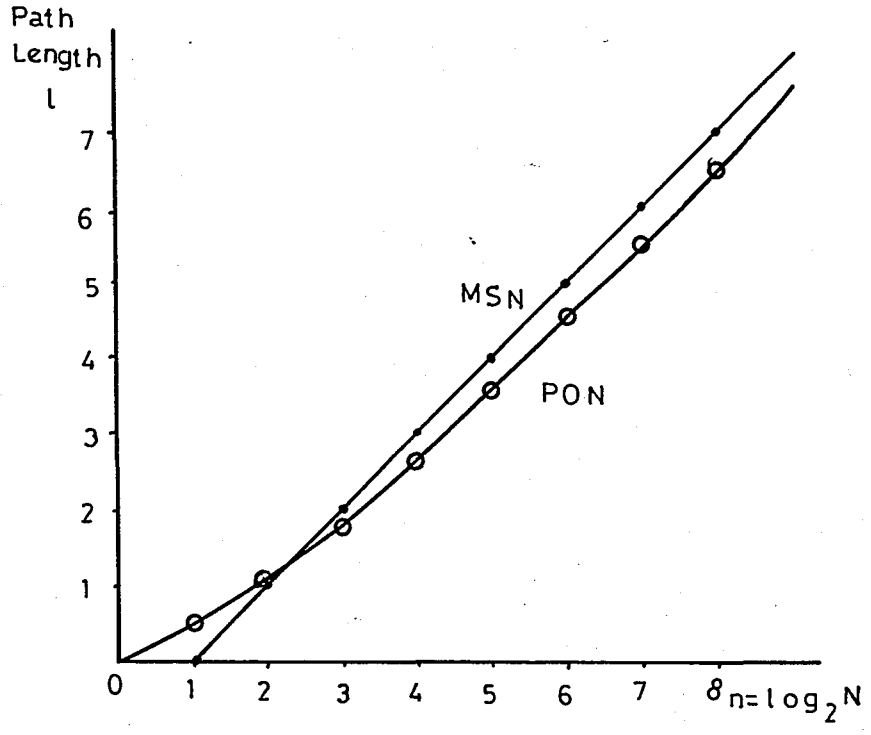


FIGURE 2.6.3 Average Shortest Path Lengths in MSN and PON

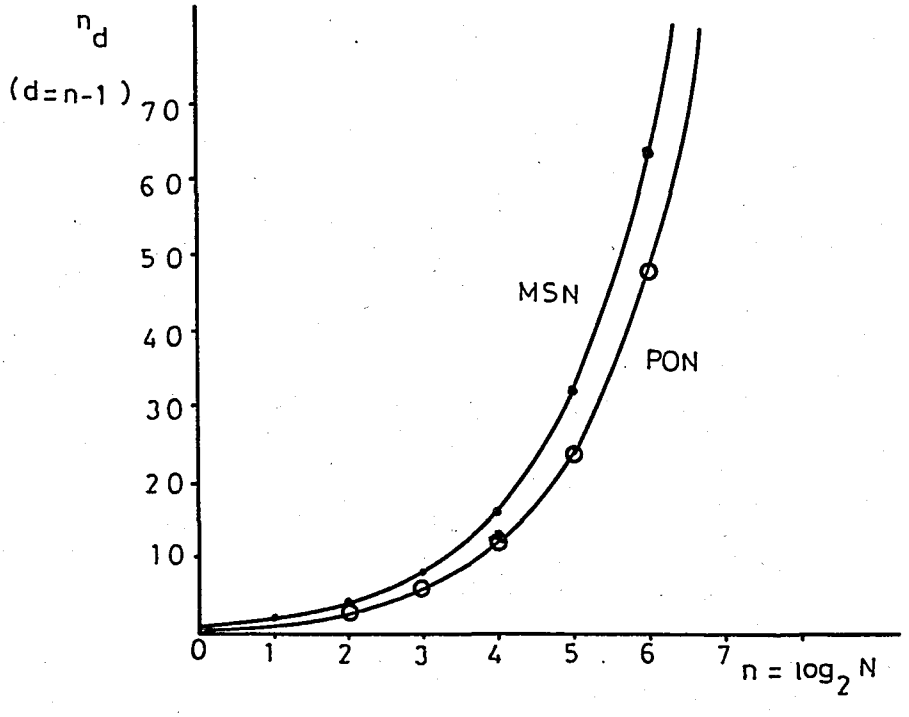


FIGURE 2.6.4 Processor Reachability in MSN and PON

l_{PON} (with c=1,2) and l_{MSN} are plotted in FIGURE 2.6.3. For $n > 2$, PON always outperforms MSN, with a difference of 1/2.

Next, we want to investigate the reachability property in PON. Let n_d denote the number of distinct processors reachable from any node within a path length of d. For $N = 2^n$ processors, in MSN's, all 2^n processors are reachable within

$$d = l_{MSN} = n - 1 \qquad (2.6.4)$$

In PON's, considering the configuration with c = 2 in order to achieve the shortest l_{PON}, we have $r = 2^{n-1}$ and

$$\log_2 r = d = n - 1$$

The number of distinct processors n_d reachable from any node within the distance d for some r-c alignments with $N \leqslant 32$ are tabulated in TABLE 2.6.2. As expected, c = 2 provides the highest processor reachability for the same N.

N	r	c	d	n_d
4	2	2	1	3
8	2	4	2	5
	4	2		6
16	2	8	3	7
	4	4		11
	8	2		12
32	2	16	4	9
	4	8		15
	8	4		22
	16	2		24

TABLE 2.6.2 Processor Reachability in PON for $N \leqslant 32$

We consider three cases :

Case 1 : $\log_2 r = d < c=2$

$$n_d = \sum_{i=0}^{d} 2^i = 2^{d+1} - 1$$

Substituting the value for d from (2.6.4)

$$n_d = 2^n - 1 , \quad d < c=2 \qquad (2.6.5)$$

Case 2 : $\log_2 r = d = c=2$

$$n_d = \sum_{i=1}^{d} 2^i = 2^{d+1} - 2$$

Substituting for d ,

$$n_d = 2^n - 2 , \quad d = c=2 \qquad (2.6.6)$$

Case 3 : $\log_2 r = d > c=2$

$$n_d = \sum_{i=0}^{d} 2^i - \sum_{i=0}^{d-c} 2^i$$

$$= 2^{d+1} [1 - 2^{-c}]$$

$$= (3/4)\ 2^{d+1} \quad , \text{ since } \quad c=2.$$

Substituting for d ,

$$n_d = (3/4)\ 2^n , \ d > c=2 \qquad (2.6.7)$$

Processor reachability of both networks is plotted in FIGURE 2.6.4 for $d = l_{MSN}$. It is seen that for this c=2 configuration of PON's, 75% of the processors compared to those in MSN are reachable in PON within the same distance, d.

TABLE 2.6.3 l and d_{max} of some unidirectional/bidirectional PON's

N	r	c	d_{max} (uni)	d_{max} (bi)	l_{uni}	l_{bi}
8	4	2	3	3	1.75	1.5
12	4	3	4	3	2.25	1.66
12	6	2	4	4	2.33	2.00
16	4	4	5	4	2.75	2.00
16	8	2	4	4	2.62	2.25
24	8	3	5	4	3.12	2.29
32	16	2	5	5	3.56	3.06
64	16	4	7	5	4.56	3.36

Moreover, we can make a rough cost comparison of the two if we assume that in PON each processor is connected to the rest of the network with a 3x3 switch (two for external connections and one for the internal connection) and we denote the cost of a K x K switch with $o(K^2)$. The cost of an MSN with n_{SW} 2x2 switches will be $(n2^{n+1})$, where for the PON with $N=2^n$ processors it will be $(9\ 2^n)$. Thus for $n>5$, i.e. $N>32$, PON will be less costly than MSN Omega.

So far we have assumed that PON is unidirectional in order to be able to compare it to the unidirectional MSN. Bidirectional PON's offer lower average shortest path lengths, than unidirectional ones, as a result of their higher processor reachability. Moreover, we relax the $r=2^k$ restriction and let r=2k. TABLE 2.6.3 illustrates average shortest path length l and maximum path length d_{max} for some unidirectional and bidirectional PON's. We see that average path lengths for bidirectional networks are an order better than for unidirectional ones and higher c is preferable in contrast to higher r alignment of unidirectional networks.

Reachability, the average shortest path lengths and maximum path lengths provide some measure for the expected performance of these partially-connected networks. Actually, in the remaining sections of this dissertation our task will be to assign software modules of a computation task to processors in such a partially-connected network so as to minimize the interprocessor communication during execution to be able to minimize the completion time of the task. Then, bidirectional PON will serve as a feasible model for the hardware component of the task assignment model to be developed in Chapter 4.

For the rest of the work will assume that interconnection of adjacent processors can be achieved using intermediate dual-ported memory units. Then, unit data transfer between a processor pair will take a time proportional to the execution of a store-load instruction sequence and some memory management. This time will be associated with an interprocessor distance of unity, as the hardware cost of a unit transfer.

3.0
THE SOFTWARE ENVIRONMENT FOR DISTRIBUTED PROCESSING

3.1 General

The research on the software problems of distributed computing can be broadly classified into three groups :

1- Languages and algorithms suitable for distributed processing

2- Program analysis, transformations and task partitioning

3- Task assignment

As already mentioned, these groups are interrelated : a proper language that enables the software designer to indicate parallelism explicitly for an algorithm that lends itself well to parallel execution eases the program analysis and partitioning phase, and naturally improves the performance of the task assignment phase.

As far as the programming languages are concerned, basically two categories can be identified :

1- Conventional languages

2- Non-conventional languages

The disadvantage of conventional languages in distributed processing is a result of their underlying machine architecture, that is, the sequential execution mechanism of von-Neumann machines. Assignment statements and unstructured constructs, like GOTO statement, seem to prevent their efficient use in distributed computers. Moreover, they have no mechanism to indicate the operation parallelism explicitly.

The non-conventional languages are basically the dataflow [ACKE82] and functional languages. The work done in non-conventional languages is highly stimulated by Backus [BACK78], who also proposed a functional language, FPP, with a desire to make programming a mathematical science rather than an art, so that programs can be generated and verified mathematically.

Functional languages are well suited to reduction machines, since they represent programs as function applications. Dataflow languages have explicit constructs like "forall", they are free from side-effects and are based on the single-assignment rule, and the variables, not their addresses, are manipulated. VAL and ID are two examples among the proposed dataflow languages. The developments in both categories are promising for applications in a distributed processing environment, although still some problems remain to be solved, such as the debugging of distributed software [MCGR80].

On the other hand, work in translating conventional languages into non-conventional ones has one sound objective; the desire to exploit the experience and vast body of software in existing languages [VEEN81]. This has led to work on program flow analysis and transformation techniques to reduce dependences in the programs in order to permit parallel execution. Related work is reported in [BANE79], [PADU80], and [ALLA80].

In [BANE79], the concept of Π-blocks is introduced, which, simply stated, corresponds either to an independent partition of a program or an indivisible block containing maximally dependent statements. Within a Π-block, data flow between operations may be represented by bidirectional arcs, whereas between Π-blocks the flow is unidirectional representing the precedence relations between the blocks. Π-blocks of a sample program graph are illustrated in FIGURE 3.1.1, where directed arcs indicate dependences between the nodes which correspond to assignment statements $S_1,\ldots,S_7$. Program partitioning corresponds to extracting Π-blocks of a task, which we will call software "modules".

The last step in developing software for a distributed environment is the assignment of these dependent modules to processors so as to minimize the execution time. This is a complicated problem of combinatoric nature and will be treated in depth in the remaining sections.

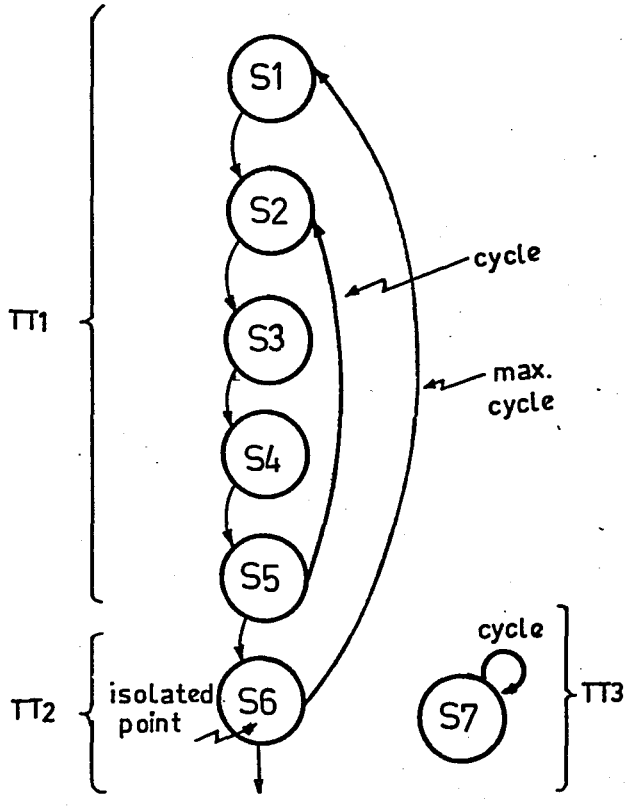


FIGURE 3.1.1 Extraction of ΠΙ-Blocks for Program Partitioning

3.2 The Task Assignment Problem

The relation of task assignment problem to other phases of software and hardware design of distributed systems is illustrated symbolically in FIGURE 3.2.1.

The task assignment problem implies that segments of a task are to be assigned to particular processors for execution at a particular time and is closely related to problems that occur in scheduling.

The simplest problem in scheduling theory [CONW67] is the scheduling of a set of independent tasks on a single machine so that some objective is achieved, such as the minimum completion time for the tasks. The constrained version of this problem, where the tasks are no longer independent, poses a little harder problem to deal with. As we move away from the simplest problems, the next class is where we have more than one machine, that is, the problems for scheduling on two identical machines which have a deterministic solution, the most well-known being the Johnson's Problem. For the case when we have more than two machines, whether in a flow-shop environment, where tasks are pipelined over a non-identical set of machines, or, in job-shop environments where independent jobs composed of a number of ordered tasks requiring different machines are scheduled, no deterministic solutions could be found so far [COFF76]. For these types of problems, enumerative procedures for optimal solutions or heuristic procedures for suboptimal solutions are chosen depending on the size and tolerance nature of the applications.

In multiprocessor scheduling, where independent or related tasks are to be assigned to processors, as well as in the general scheduling problems mentioned above, the primary concern has been to make an assignment to processors in order to minimize the total or mean completion time of tasks by considering only the processing time requirements of tasks and the precedence relations among tasks if they are not independent.

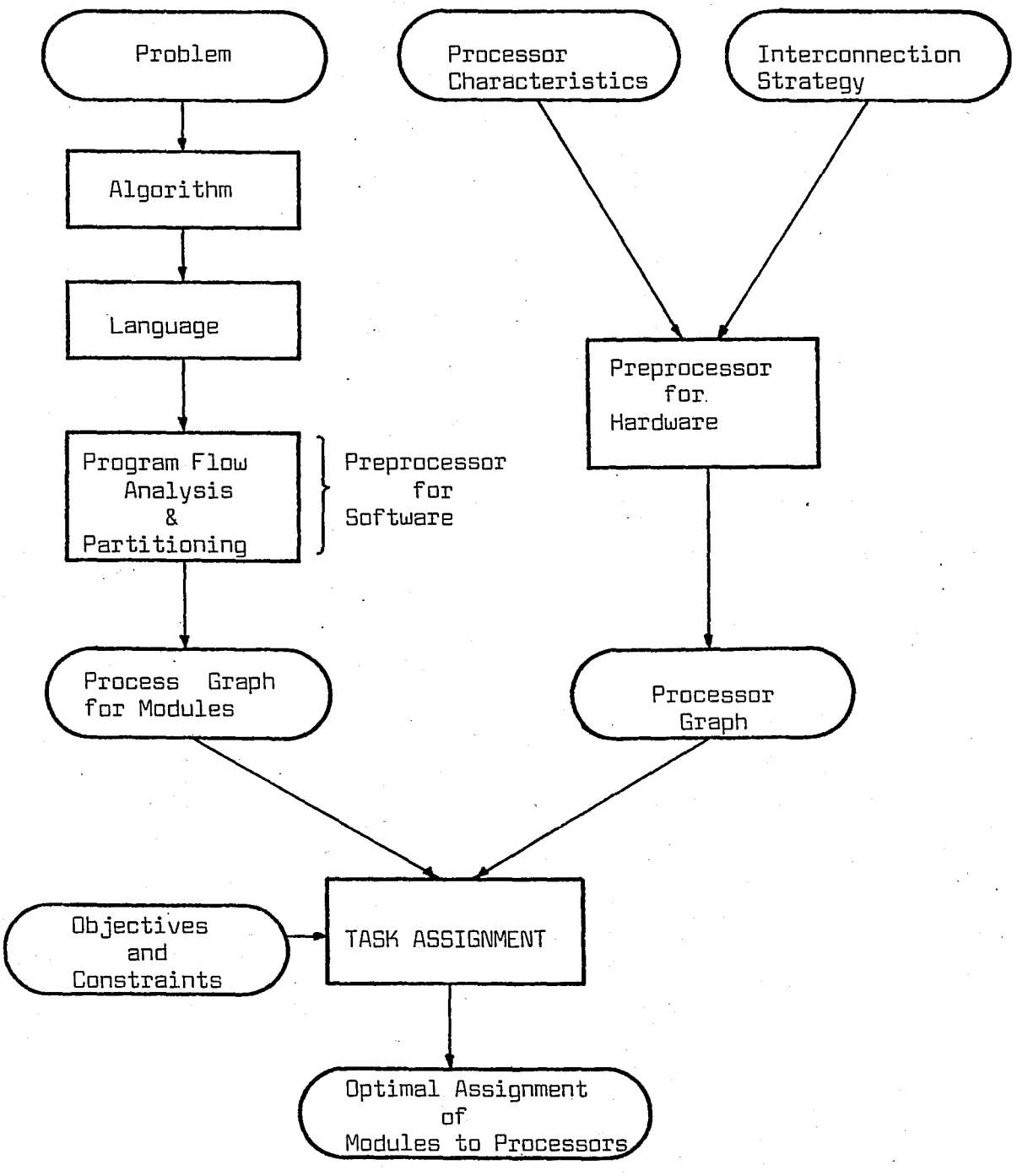


FIGURE 3.2.1 Relation of Task Assignment Problem to Other Phases of Software and Hardware Design

Ideally, in a multiprocessor system one would have expected an ideal increase in throughput that is linearly proportional to the number of processors used, when compared to the single processor results. However, in practice, the throughput increases for the first few additional processors only, and after a certain point it begins to decrease with every new processor added to the system. This phenomenon of decrease in throughput for increased number of processors is called the " saturation effect " , [FLYN72], [CHU 80], and is caused by the excessive interprocessor communication (IPC) in the system, an issue not considered in multiprocessor scheduling where communication in the system is totally ignored. This is where the task assignment problem differs from that of scheduling.

In order to avoid the saturation effect we must investigate the nature of IPC. The overhead in processing time due to IPC may occur for various reasons. In some systems it is only due to the actual data passing between dependent software modules that reside on different processors. In some other systems it may also include the time required to satisfy communication protocols and for the management of store and external resources. There may also be delays due to queueing at both ends of a communication path that add up to IPC overhead and altogether degrade the system performance.

Given a software problem, which we will call a "task", assume that in order to be executed in a multiprocessor system it is partitioned into a number of software modules such that the amount of data that needs to be transferred between dependent modules, i.e. the amount of intermodule communication (IMC), is minimal. This means that blocks of instructions of the program with strong data dependences among them are clustered in forming the modules. Then, the amount of IMC in a partitioned task is strictly a function of the software characteristics of the problem and the partitioning procedure employed. The next step in distributed processing of the task is to assign these modules to processors. When any two modules with IMC between them are assigned to different processors in the system,

they will cause IPC, in an amount proportional to IMC between the modules and the cost of unit transfer between the processors, which may conveniently be associated with the distance between them. If the modules are coresident, we assume that the overhead is negligible and that there is no IPC cost. IPC is, therefore, a function of IMC, the distance between processors and the module-to-processor assignment, and in order to minimize the overhead due to IPC, we require proper task partitioning and task assignment. Here we assume that there is a software preprocessor which performs the partitioning phase and thus the modules with their related IMC values are available for the assignment phase.

In the remaining sections of this study, our aim is to optimally assign modules to processors so as to achieve minimum finish time or maximum throughput. In this respect, we need to tackle with two conflicting problems. In any multiprocessor system the obvious tendency is to distribute the work-load to all the processors as evenly as possible in order to reduce the overall processing time and improve system efficiency. This corresponds to the concept of load balancing. On the other hand, we have seen that a processing overhead due to IPC is incurred whenever dependent software modules are assigned to different processors, and to reduce IPC, we have to cluster the communicating modules to as few processors as possible. Then, an optimal assignment strategy should balance these two competing factors for maximum system performance.

In any real application, we have other complications due to constraints on the assignment, related to the limitations on resources. These limited resources may include the number of processors, processor characteristics such as speed, memory capacity and peripherals, and real-time constraints such as the maximum finish time of the task or the task frequency. An efficient assignment procedure must be able to incorporate such constraints in the optimization model and tune the system to satisfy the constraints.

So far nothing has been said on the interconnection strategy of processors. For fully-connected processors, i.e. when there is a direct path connecting any processor pair in the network, it is sufficient to consider the interprocessor distances and processor speeds in the assignment process. In the case of identical processors with uniform interprocessor distance -a highly valid assumption for closely-coupled processors- , the contribution of any assigned processor to IPC cost due to IMC will be equal.

Task assignment for partially-connected processor networks presents additional problems. The first difficulty arises in the communication of non-adjacent processors. Any pair of processors that are not directly connected have to communicate over some intermediate processors and the success of communication naturally depends on the availability of these processors during the exact transfer intervals. Assuming that they are available, these processors-on-route will have additional communication loads. This is the situation with networks of unique shortest paths between each pair. When the interconnection strategy allows more than one path of shortest length between processor pairs, we have the additional problem of alternate routes between processors. Now apart from the availability of processors we have to select some path among the alternatives i.e. route the data over some selected processors based on the preferable satisfaction of our objectives.

In this dissertation, our emphasis is on the optimal module-to-processor assignment in partially-connected homogeneous processor networks with alternate routes under the limitations imposed by the number of processors and the real-time constraints.

We will distinguish between two environments for the task assignment, one being called the non-loaded or single-run environment where tasks are expected to be repeated at irregular intervals over a long period of time. The second environment we consider is the loaded or multi-run environment

where successive, periodic execution of a task is of concern. Accordingly, we will propose two objective functions to be minimized in order to achieve load balancing and minimization of IPC. The first objective function is the so-called port-to-port time (PTP) used in a recent research for fully-connected networks in non-loaded environments [HOL082] . PTP is defined to be the elapsed time from the first start time of any module until the finish time of the last module to finish, i.e. the maximum completion time of the distributed task, and is composed of the processing time, the time spent for IPC, and the idle time on processors, corresponding to the waiting time of modules due to precedence constraints. Assignments that minimize PTP will obviously minimize IPC and balance load distribution, especially in non-loaded environments. The performance of PTP criterion in loaded environments depends on the task frequency and may degrade at higher frequencies.

The second objective function that we introduce produces assignments with most well-balanced load (including processing and communication) and suboptimal PTP in non-loaded environments. We call it the least re-initiation period (LIP), related to multiple, periodic execution of a task. LIP corresponds to the maximum of the reserved-times of the processors, determines maximum input data rate to the system, i.e. the task frequency, and also is a measure of the overlap between successive task executions. Minimum LIP is a robust performance criterion maximizing the overlap and suboptimizing PTP, such that in multi-run environments, assignments with low LIP and suboptimal PTP outperform assignments with higher LIP and optimal PTP, this being possible with as few as two repetitions of the task set. Then, in a multi-run environment we speak of the overall completion time and minLIP dominates minPTP criterion.

Related to scheduling terminology, our assignment strategy will be based on nonpreemptive (or basic) scheduling, meaning that interruption of a module is not permitted before its completion. Although, in general, preemptive disciplines generate better schedules than nonpreemptive ones, the context-switching overhead of preemption will cause further performance degradation and is unacceptable for our task assignment environment.

In the next section, we present a brief review of the related research and solution techniques for the task assignment problem and Section 3.4 outlines the proposed method of attack, which is treated in detail in Chapters 4 to 6.

Any solution to the task assignment problem exhibits combinatoric complexity due to the inherent combinatorial nature of the problem. Indeed, any assignment or scheduling problem apart from the simple ones we have mentioned, is in the class of the so-called NP-complete (or NP-hard) problems indicating that they possess no deterministic solution computable in polynomial time with respect to the dimension of the input [ULLM76] . This means that for very large systems, we might have to be content with suboptimal solutions and the task assignment procedure should be flexible enough to incorporate heuristics in order to find acceptable solutions for problems of higher dimensions. Chapter 7 of the dissertation presents a discussion of some methods to reduce the complexity of enumerations.

3.3 Related Research and Solution Techniques

As previously mentioned, task partitioning and task assignment are the two essential phases in the optimal utilization of a distributed system.

Task partitioning is purely a software design issue, related to program analysis techniques and compiler generation, [KUCK72], [JENN77], [ALLA 80], [PADU80], [JOHN80], [VEEN81] , and its importance is in providing the software component of the input of the task assignment process.

The task assignment problem and related problems of processor and job scheduling, as well as the file allocation problem, have been studied for many years, most of the techniques used being adaptations from older, well-established results developed in management science and operations research, i.e. techniques from graph theory, optimization theory,queueing theory, mathematical programming and various algorithmic or heuristic methods. The cost function in these studies is usually formulated so as to minimize either the maximum finish(flow) time or the mean flow time of the generated assignment(schedule) [CONW76] .

A graph theoretic approach is one of the most commonly used techniques by researchers both in task assignment [STON77], [JENN77], [RAO 79], and in scheduling fields [CONW67], [COFF76] . It is based on a graph representation of the task where the modules are represented as nodes in the graph and the dependence between modules by arcs connecting the associated nodes. With this graphical representation of the problem, in scheduling, the nonpreemptive schedules generated are "list schedules", so that the problem is reduced to that of finding an optimal list of tasks and whenever a processor is available it is assigned a task from the ordered list [COFF76] .

We want to mention two important issues from processor scheduling. One of them is the work of Hu [HU 61] in operations research, which is next to Johnson's results for two-machine flow-shop problems is probably

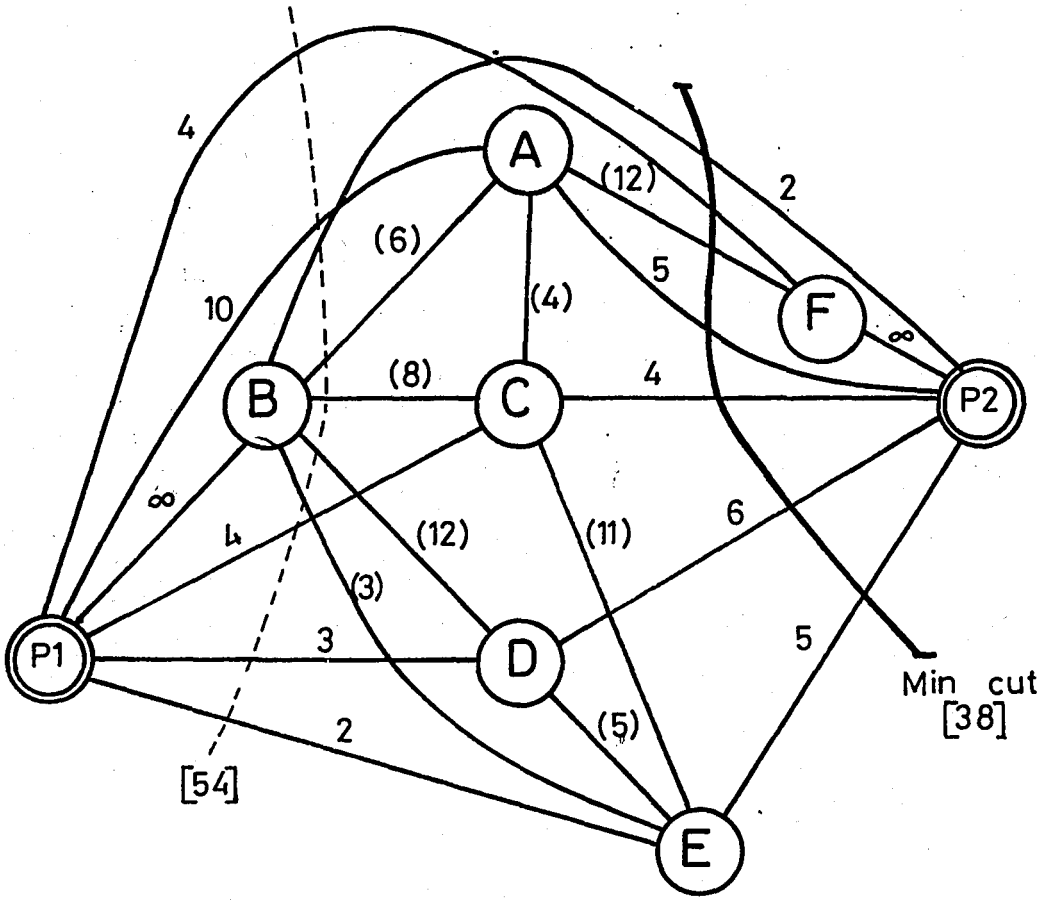
the most frequently cited reference in multiprocessor scheduling, as commented in [GONZ77] . Hu addressed the problem of bounds -assuming unit duration tasks- , on 1) minimum completion time of a task graph, given limited number of processors, and 2) minimum number of processors to process a graph, given limited completion time. Approximately, his bounds relate the required minimum time with the levels of the graph and the minimum number of processors with the number of nodes at each level.

A different approach, in computer science, to time and parallel processor bounds in terms of the number of operands in arithmetic expressions for compilation of high level languages is reported in [BANE79] .

The reason we emphasized Hu's work is due to the so-called "multiprocessor anomalies" problem first addressed by Graham. In [GRAH66], Graham shows that the maximum finish time of a schedule may actually 'increase' with relaxations on the constraints, the most important ones for us being the decreased processing times and precedence relations in the task graph, and the increased number of processors. These anomalies are studied in a scheduling environment where the communication costs have even not been considered. The first two problems mentioned are related to both partitioning and assignment, and the last has strong .implications for the task assignment. Usually in the assignment problem, the number of processors in the system is assumed to be a fixed input parameter, but because of the communication overhead it may well be that a given task will finish execution earlier if the number of processors used is less than the available. This is true especially in partially-connected networks where the average interprocessor distance increases with an increased number of processors in the system. If we could have achieved bounds on the optimal number of processors, given the task graph, the time limit and the processor graph, we could have used optimal number of processors in the assignment : either by selecting a closely-coupled subset of the network in a normal operation mode where the network is fixed, or by designing accordingly if in the design phase. Unfortunately, although Graham's

results carry over to the task assignment environment, Hu's bounds on unit-execution task graphs with communication costs excluded are not applicable and determination of bounds for the general task assignment environment that we consider poses a very challenging problem to be solved due to all the additional complications. [RAMA72] gives an example of processor scheduling using Hu's bounds, where the concept of E (earliest precedence) and L (latest precedence) partitions is used on the graph to determine the critical paths and to select dominating nodes. A good survey on deterministic processor scheduling is presented in [GONZ77] .

One method employed to solve the task assignment problem in presence of communication costs is borrowed from the work on flows in networks and makes use of the well-known max-flow min-cut theorem [FORD64] . Here, it is assumed that IPC costs between non-coresident modules are known apriori and assigned as weights to arcs in the task graph. The processing cost of each module is assumed to be given as well, so that if a module cannot be executed on a processor it is assigned an infinite processing cost for that particular processor. With this setup, Stone [STON77] has shown that (FIGURE 3.3.1) : for two processors (P1,P2), two nodes (P1,P2) can be added to the graph and connected to modules with arcs that are assigned a weight as the processing cost of the module on the opposite processor and by treating the resulting graph as a network, with P1 as the source and P2 as the sink, the max-flow min-cut theorem can be applied for a min-cut on the network, to partition the modules into two disjoint sets and thus to distribute the modules to two processors for minimum cost assignment. Although it seems to be simple and favourable in complexity (with a time upper bound of N^3 for a network with N nodes [RAO 79]), this method of module assignment is infeasible since it provides no information on sequencing of modules on processors, resource constraints and load balancing cannot be incorporated and it becomes unmanageable for N>2. As reported in [RAO 79], even in the two processor case, where one of them has limited memory, the problem is in the NP-complete class and the complexity advantage is lost.



P1 : {A,B,C, D,E }

P2 : { F }

FIGURE 3.3.1 Graph Showing IPC {(.)} and Processing Costs for the Min-Cut Example

The other basic method applied in file allocation and task assignment is to formulate the problem as a discrete optimization problem with constraints which then can be solved using mathematical programming techniques such as integer programming, dynamic programming and branch-and-bound. Because of the large number of constraint the resulting problem is non-linear as an integer programming problem and has to be linearized by additional constraints [CHU 69], [GYLY76], at the expense of increased problem size. Apart from the increase in size and solution time of the problems, integer programming methods are not reliable due to the possibility of not converging to a solution at all.

Due to the NP-complete nature of the optimal assignment any solution method depends on enumeration techniques that examine all feasible alternatives. Ignoring integer programming for the reasons stated above, dynamic programming [BELL62] and branch-and-bound [KOHL76] are the two well-known methods used to reduce enumeration.

If we model the search space as a finite tree of partial solutions, dynamic programming is a breadth-first search method that uses dominance rules to prune the tree. Being a breadth-first search method, demand on the memory capacity is high to be able to save the entire solution tree and since only partial solutions are generated at every step, it cannot be interrupted during execution before the final stage, with the hope of finding suboptimal solutions.

Branch-and-bound, on the other hand, is a depth-first search method where the most recently computed best solution is always available and the process can be interrupted before the end for acceptable solutions. In [MA 82], this technique is used for assignment considering IPC cost as the only objective to be minimized.

An interesting approach to deal with enumerations as depth-first search methods might be the distributed execution of the task assignment problem itself. Such an effort on network computes is reported in [EL-D80] .

Apart from the module assignment problem for multiprocessors, data mapping problems to minimize communication costs have been extensively studied for array processors, i.e. in SIMD environments. Some examples are given in [BOKH81], [IRAN82] and [MOLD83]. In [BOKH81], Bokhari considers the mapping problem and shows its relation to graph isomorphism problem, bandwidth reduction problem for sparse matrices and to quadratic assignment problem. He presents a heuristic method based on graph theory, where using adjacency matrices to represent the process and processor graphs, he tries to achieve maximum matching of the two. The algorithm complexity is reported as $o(N^2)$ for an N x N array of processors. Obviously, such heuristic methods can as well be employed for the task assignment problem to obtain suboptimal solutions.

An algorithmic approach to optimal task assignment is employed in [HOLL82]. Algorithmic solution procedures may be considered as depth-first search methods, similar to the branch-and-bound technique, where increased number of problem constraints helps to reduce the search space by efficient pruning, in contrast to complicating the solution process in other methods. They also allow generation of suboptimal solutions in order to reduce the complexity. In [HOLL82], an optimal solution to the task assignment problem under real-time constraints in non-loaded environments and a suboptimal solution in loaded environments are presented. Her algorithm can be used for task assignment in fully-connected processor networks of uniform or variable interprocessor distance to provide a safe upper bound on PTP, since she makes the approximation that when a module has to pass data to more than one successor, the successors are assumed to start execution simultaneously only after the last transmission is complete, although the modules whose data are sent earlier, already have their data available and

can start execution. This way of computing PTP might lead to a situation where an assignment is rejected as not satisfying the real-time constraints even though it actually meets the deadline. Moreover, her method cannot be used for partially-connected networks where, intermediate processors are used in data transfers and the IPC cost which is a function of the distance should not all be associated with the source (transmitting) module. Another point is that, minPTP is used as the main objective in both loaded and non-loaded environments, whereas we will show that the proposed minLIP is a better performance measure in loaded environments.

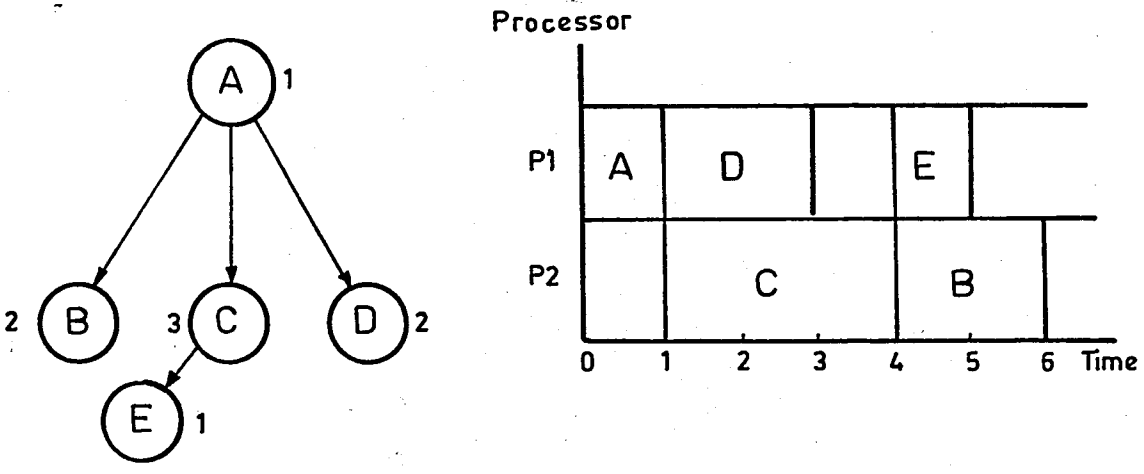
3.4 Proposed Method of Attack

We have defined our task assignment environment as one with a single task of dependent modules, a partially-connected network of identical processors and with alternate routes, and limited time which is a function of module processing times, IPC time and idle time due to precedences.

The presence of precedence relations makes it necessary to know the location and the time sequence of every other module in the system to make an assignment. Moreover, because of the partially-connectedness of the network, distances and alternate paths between processors and the availability of intermediate processors at required instants also need to be known. Closed form mathematical optimization techniques are not able to represent all the required information and are therefore inappropiate for the solution of the task assignment problem as developed here, which requires some form of enumeration.

We propose an algorithmic solution procedure to solve the task assignment problem. It is based on a depth-first search technique and constraints are imposed to reduce the solution space. Every feasible assignment that satisfies the constraints will be evaluated and if its performance is better than the previous ones, it will override the formers as the so-far-best assignment. With the algorithmic approach, it is also possible to specify bounds on the number of generated assignments or "acceptable" values for the desired performance and interrupt the algorithm execution before termination.

In representing the generated assignments, it is common to use a graphical representation where it is easy to see the load on each processor and sequencing among the modules. We will refer to such a chart as the load density function (LDF), following the terminology in [HOLL82], -just as a matter of preference-, which is the same as the "Gannt chart" used in



a) Sample Task Graph

b) The Load Density Function

FIGURE 3.4.1 A Sample Graph and Load Density Function

scheduling theory [CONW67]. As an example, the load density function corresponding to an arbitrary assignment (assuming IMC = 0) of a sample task graph is illustrated in FIGURE 3.4.1. The vertical axis represents the processors and the time used by the modules is shown along the horizontal time axis.

In the next chapter, we will develop a model, state our objectives and present a mathematical formulation of the problem both for fully-connected and partially-connected processor networks.

In Chapter 5, the methods of actual storage representations for the hardware and sofware components of the problem will be discussed and the most efficient ones will be determined in preparation for the algorithmic solution presented in Chapter 6.

4.0
THE TASK ASSIGNMENT MODEL

4.1 Description of the Model Components

A model developed to describe the task assignment environment must represent both hardware and software components.

In our model, we define a software task as a collection of cooperating modules obtained after task partitioning, such that no two modules have maximally connected components between them and the precedences are unidirectional. In a distributed processing system of multiprocessors the modules comprising a task will be executed in different processors and the completion time of the task is considered to be the completion time of the last module to finish.

A module is described by a number of attributes related to its processing requirements: mainly, the number of instructions to execute the module which when divided by the speed of a processor gives the processing time for that module, the amount of storage required by the instructions and data of the module, and the amount of data to be sent to other succeeding modules or to be received from the predecessors, called the intermodule communication (IMC).

The hardware environment of the task assignment model can be described by the characteristics of the processors and the interconnection strategy between the processors.

The processors may be characterized by their speed, their memory capacity and I/O capability. For a network of non-identical processors these attributes have to be supplied for each processor. Here, we confine ourselves to networks of identical processors and will need to specify just a single set of processor characteristics.

The interconnection strategy describes the connection pattern of processors and can be characterized by giving the pairs of directly connected processors and the interprocessor distances. For a fully-connected interconnection strategy, it is sufficient to provide distances between the processor pairs. For a partially-connected network, full information as to the adjacent processors of each processor and for indirectly connected pairs, the identities of processors on the paths and the distance information must be supplied.

In this study, we treat the more general and realizable case of partially-connected interconnection strategy of identical processors.

4.2 Performance Measures

Having mentioned the components of the model, task assignment is the process of assigning modules comprising a single software task to processors in a network such that some objective is optimized. For this study, the initial objective is to minimize the maximum completion time of the task, measured from initiation to completion time of the task set, and this time will be called the port-to-port time (PTP) similar to the terminology in [HOLL82], an appropiate term borrowed from avionics. Thus, if we consider the processor network as a system and the task as a whole, PTP is the time elapsed from entry of the task to the exit from the system, and hence is given the name port-to-port time.

PTP consists of processing time of modules, interprocessor communication time (IPC) caused by modules sending data to dependent modules resident on other processors, and the idle time on processors due to precedence relations, such that, a module assigned to a processor cannot be executed although the processor has finished work on the previously assigned modules, because its predecessors on another processor has not yet completed execution. IPC depends both on the IMC between two modules and the distance between the processors in which the modules reside. For coresident modules with IMC between them IPC is taken to be zero. Thus, each component of PTP is a function of the software characteristics, the hardware characteristics and the module-to-processor assignment, and our task is to find optimal assignment given the software and the hardware characteristics.

PTP is a good performance measure of load-balancing and reduced IPC in single-run or non-loaded environments where a new task is not instantiated before the PTP of current task is reached. However, one might be interested in periodically repeated instantiations of a task or overlapped execution of successive tasks so as to improve utilization and the throughput of the system. Here, we introduce such a performance measure, LIP, the least re-initiation period, and denote it with the symbol (Λ).

Let R correspond to the reserved time of a processor from initiation of its first module until completion of the last module assigned to it. The least re-initiation period LIP, is the maximum R over all the processors and is a very good performance measure for load-balancing, such that, in fully-connected networks, minLIP is achieved at the most load balanced assignments. It gives an indication of the overlapping of successive task executions such that a small LIP value implies a high degree of overlapping and permits higher initial data rate. MinLIP value gives the highest data rate allowable before queues start to build up in the system. This is an important issue in performance prediction in loaded environments. In [HOLL82], Holloway has noted that assignments generated for fully-connected networks using minPTP criterion in a non-loaded environment, might perform very badly in loaded environments and tried to predict behaviour of the system by taking minPTP non-loaded assignment as the starting point, generating curves of PTP for increased loading for two extreme cases -a set of independent modules (there called min.precedence) and a task chain of modules (there called max.precedence)-, and, interpolate between the two curves. Since minPTP assignment does not yield good performance in loaded systems, the use of minLIP as the objective in generating assignments for loaded environments is expected to improve the situation and produce superior results.

Thus, we distinguish between the two environments in task assignment, the single-run environment and the multi-run environment, and recommend the use of two different objectives, minPTP and minLIP, respectively.

In the following sections, we develop a mathematical formulation of the task assignment problem, based on the model and the objectives presented so far.

4.3 Mathematical Formulation

The software component of the task assignment model is described by a process graph, defined to be a finite, directed acyclic graph (DAG) , where the modules are represented as nodes in the graph and, data transfer and precedence relations between the modules are represented by directed arcs between the nodes in the graph [HARA69], [COFF76], [GONZ77] . This means , the directed arcs between the nodes imply that a partial ordering or precedence relation exists between the nodes. Knuth [KNUT73], defines a partial ordering as a relation among the objects of a set satisfying the following properties. For any elements i, j, k in S :

i) Transitivity : If $i \prec j$ and $j \prec k$, then $i \prec k$.

ii) Asymmetry : If $i \prec j$, then $j \not\prec i$.

iii) Irreflexivity: $i \not\prec i$.

where the relation $i \prec j$ implies that i precedes j . Apart from the globally assigned module number given to the nodes, a second number is associated with each node which refers to the execution time of the module. Associated with each arc there is a number corresponding to the amount of IMC between modules, associated with end nodes of the arc. The process graph can then be represented by a quadruplet $(\mathcal{M}, \mathcal{T}, I, \prec)$, corresponding to module set, processing times, IMC and precedences, respectively. An IMC value of zero implies that there is no data transfer between the considered modules and $\prec = 0$ implies independent modules. Here we consider single-entry or single-entry-single-exit connected (SEC) graphs [RAMA72] and $\prec \neq 0$.

The hardware component of the task assignment model is also represented by a graph, a processor graph, defined to be a non-directed graph if the interconnecting links are bidirectional or a directed graph in case of unidirectional links. The processors are represented as the nodes of the graph, with connecting arcs corresponding to the interprocessor links. Distances (or other link costs) between processors are represented as weights associated with arcs in the graph.

In this study, we assume identical processors of known characteristics (such as the speed, the memory capacity,.etc..) and that the processing times of modules are given after being normalized with respect to the processor speed. When considering partially-connected networks, we also assume a bidirectional network and the value given for the distance between two nodes corresponds to the number of interprocessor links traced from the source to the destination node involved in the communication. For fully-connected networks, distance corresponds to the length of a path connecting a processor pair.

In formulating the problem, we will first handle the case for fully-connected networks since it is easier to grasp and then deal with the problem of partially-connected networks based on the formulation of the former case.

For mathematical formulation, we will use the following notation to describe the software and the hardware components. The actual storage representations will be derived in the next chapter.

For a process graph of M modules and a processor graph of N processors we define (Capital characters within brackets denote array dimensions):

Software :

PROC(M)	An M-vector describing processing time requirements of each module. Convenient unit is seconds .
ND	A scalar. The number of arcs in the process graph, i.e., the number of dependent module pairs in the task.
DEP(ND,2)	A matrix giving list of dependent pairs in the process graph such that for the pair given in each row, former module precedes the latter in the graph.
IMC(ND)	A vector consisting of IMC values between each pair of dependent modules given in DEP . Convenient unit is bytes or words .

Hardware :

DIST(N,N)	An NxN matrix describing the 'distance' between processors.
PROUT(.)	List of processors-on-route between communicating processors that are not directly connected .
ROUT(N,N)	An NxN matrix for routing in partially-connected networks. Entries are pointers to PROUT .

After the software and the hardware have been specified, we next define an assignment matrix, X(M,N), such that

$$x(i,k) = \begin{cases} 1 \text{ , if } i \text{ th module is assigned to k th processor} \\ 0 \text{ , otherwise} \end{cases}$$

That is , we assume that an assignment has been generated and X(M,N) has been constructed accordingly .

Now we may start computing the components of PTP corresponding to an assignment, namely the processing time, the IPC time, and the idle time.

Processing time is represented by an M-vector, where

$$PROCT(i) = PROC(i) * x(i,k) \qquad (4.3.1)$$

That is, PROCT(i) is the processing time of module i on processor k to which it has been assigned .

The time spent in IPC is a function of IMC of modules, the distance between processors, and the module-to-processor assignment. Then IPCT is represented by an M-vector, where

$$IPCT(i) = \sum_{j \epsilon R_i} IMC(i,j) * x(i,k) * x(j,l) * DIST(k,l) \qquad (4.3.2)$$

R_i is the set of modules receiving data from module i, i.e., $R_i = \{j | IMC(i,j)>0\}$. This means that, the time spent in IPC, for module i, is given by summing over all modules j to which module i sends data, with the summands consisting of the IMC from module i to module j times the distance between the processors to which modules i and j have been assigned.

In order to specify the idle time on a processor, we make use of start and finish times of modules on processors. As we have mentioned earlier, idle time on a processor is caused when the processor, having completed execution of a previous module, is free but cannot start execution of the next module in its assigned work list since that module is not yet "ready".

This situation is due to the precedence relations among the modules. A module is "ready" for execution when all its predecessors in the process graph are completed and provided it with the data to operate on. This is same as the "firing" concept of data-flow machines - an instruction is "fired" when all its operands are available- , although the contents of a module here is assumed to be much more than a single instruction. Thus, a module whose predecessors reside on other processors must wait until it receives the required data.

Then, a convenient way to describe the idle time of some processor k before executing certain module i is to treat it as the delay between the start time of module i and the finish time of some module (i_{-1}) assigned to precede i on k, and associate it with the waiting time of module i . We denote this waiting period of process modules by an M-vector, WAITP, where

$$WAITP = [\ START(i,k) - FINISH(i_{-1},k)\ ] * x(i,k) \qquad (4.3.3)$$

We had defined PTP as the maximum finish time over all the modules, which is equivalent to maximum finish time among all the processors to

which the modules have been assigned. Then, in terms of the three components, the finish time F(k) of processor k is given by

$$F(k) = \sum_{i \in A} [\ PROCT(i) + IPCT(i) + WAITP(i)\ ] \qquad (4.3.4)$$

where $A = \{j \mid x(j,k)=1\}$.

Because of the nature of its components, F(k) is a function of module-to-processor assignment. The maximum finish time over all the processors is PTP, so that

$$PTP = \max_{i \leq k \leq N} \{\ F(k)\ \} \qquad (4.3.5)$$

Our aim is to make assignments such that PTP is minimized. Then, minimum PTP is given by

$$PTP_{min} = \min_{X} \{\ PTP\ \} \qquad (4.3.6)$$

that is, by minimizing PTP over all possible assignments.

In a more representative form, this is equivalent to

$$PTP_{min} = \min_{X} \{\ \max_{1 \leq k \leq N} [\ \sum_{1 \leq i \leq M} (\ PROCT(i)+IPCT(i)+WAITP(i)\ )\ *\ x(i,k)\ ]\ \} \qquad (4.3.7)$$

Then the optimization problem given in (4.3.7) produces a minimum finish time task assignment. The data required to solve the objective function are a measure of processing time, IPC time, and precedence constraints , and are obtained from the software and hardware specifications and the task assignment .

Having finished formulation for PTP, we next discuss formulation of the optimization problem for LIP, which determines the degree of overlap between successive task executions in a multi-run environment.

We have defined R for each processor as its reserved time. We notice that the difference between R(k) and F(k) for some processor k is the start time of the first module assigned to k . Then R(k) of processor k is given by,

$$R(k) = F(k) - START(\ 1(k),\ k\ ) \qquad (4.3.8)$$

where F(k) is given in (4.3.4) and START(1(k), k) denotes the start time of first module on processor k. The maximum reserved time over all the processors gives LIP for the assignment, i.e.,

$$LIP = \max_{1 \leqslant k \leqslant N} \{\ R(k)\ \} \qquad (4.3.9)$$

In order to make minLIP assignments, we have to find minimum LIP over all the possible assignments, so that

$$LIP_{min} = \min_{X} \{\ LIP\ \} \qquad (4.3.10)$$

or equivalently in a form similar to (4.3.7) ,

$$LIP_{min} = \min_{X} \{\ \max_{1 \leqslant k \leqslant N} \{\ \sum_{1 \leqslant i \leqslant M} [\ PROCT(i)+IPCT(i)\ ] * x(i,k) + \sum_{\substack{1 \leqslant i \leqslant M \\ i \neq 1(k)}} WAITP(i) * x(i,k)\ \}\ \} \qquad (4.3.11)$$

Comparing (4.3.7) and (4.3.11) , it is easily seen that minimizing LIP helps to minimize PTP as well .

The overlap in successive task executions is given by,

$$OVLP = PTP - LIP \qquad (4.3.12)$$

Thus, decreasing LIP helps to increase the overlap. Actually there might be many assignments with the same PTP but different LIP values, such that the one with minimum LIP gives the maximum overlap.

If a task is to be executed K times (K-run), then the total completion time after K iterations, denoted by KPTP, is given by ,

$$KPTP = (K-1)\ LIP + PTP \qquad (4.3.13)$$

Thus for a K-run environment we can select an assignment based on KPTP, related to LIP and PTP. To see this, consider two assignments X_1 and X_2 characterized by PTP and LIP values as

X_1 : { PTP_1, LIP_1 } where $PTP_1 > PTP_2$

X_2 : { PTP_2, LIP_2 } $LIP_1 < LIP_2$

In a single-run environment, the choice of X_2 is preferable since one aims for lower PTP. The situation might alter, however, in a multi-run environment. To determine when X_1 with higher PTP is preferable, we compute KPTP for each assignment. We require that $KPTP_1 \leqslant KPTP_2$, or

$$(K-1)\ LIP_1 + PTP_1 \leqslant (K-1)\ LIP_2 + PTP_2$$

Then the number of iterations K, after which X_1 supercedes X_2 is given by ,

$$K \geq \frac{(PTP_2 - LIP_2) - (PTP_1 - LIP_1)}{LIP_1 - LIP_2} = \frac{OVLP_2 - OVLP_1}{LIP_1 - LIP_2} = 1 + \frac{\Delta PTP}{\Delta LIP} \qquad (4.3.14)$$

That is, when K exceeds a certain value (which may be as low as 2) given by (4.3.14), X_1 outperforms X_2 , and the selection is governed more by the lower LIP criterion than by the lower PTP .

The points mentioned so far will be used to determine the selection of optimal assignments depending on the problem environment. Before proceeding with the task assignment in partially-connected networks, we want to mention one more point related to LIP and also give an example for the task assignment on fully-connected processors.

When comparing the performance of single processors and multiprocessors one commonly used measure is the speed-up S_N [PADU80] achieved by using N processors, such that

$$S_N = T_1 \; / \; T_N$$

where T_1 is the time required to execute the task sequentially on a single processor, and T_N by using N processors. Ideally $S_N = N$ which is never achievable in practice.

We define K-run speed-up S_{NK} , for the iterative execution of a task on N processors, by

$$S_{NK} = \frac{K * T_1}{KPTP} = \frac{T_1}{LIP + (PTP - LIP) \; / \; K} \qquad (4.3.15)$$

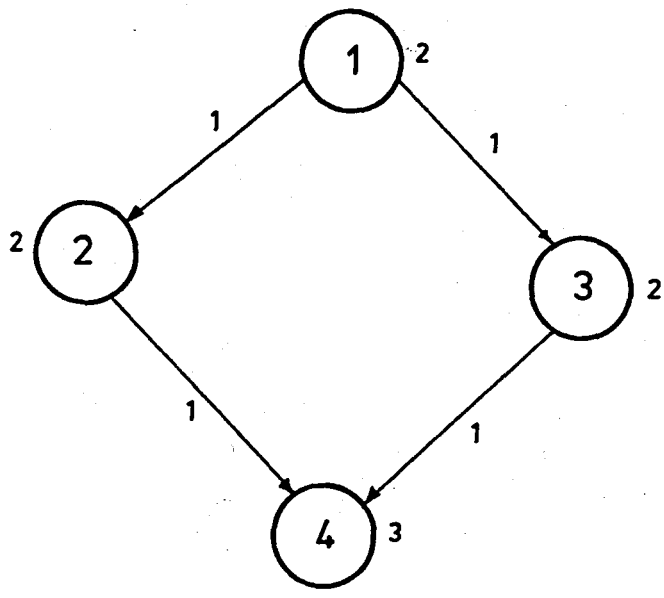


FIGURE 4.3.1 Example Process Graph

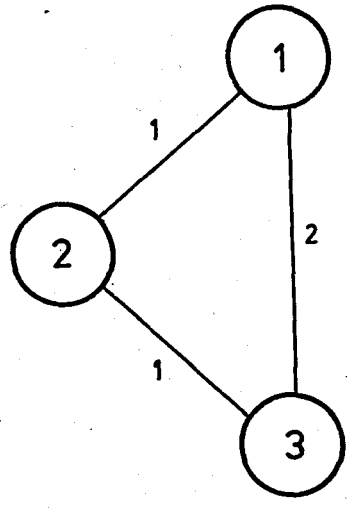


FIGURE 4.3.2 Example Processor Graph (F-C)

and ,

$$\lim_{K\to\infty} S_{NK} = T_1 / LIP \qquad (4.3.16)$$

Expression (4.3.16) shows very clearly the influence of LIP on speed-up, as is to be expected.

Now, we give an example of task assignment. The process graph shown in FIGURE 4.3.1 corresponds to a task consisting of four modules. The processing time for modules and IMC values are given next to nodes and arcs, respectively. We want to make an arbitrary assignment of four modules to three processors shown by the graph in FIGURE 4.3.2. The distances between processors are given as weights on non-directed arcs. The two graphs are represented in our notation by the following items :

$M = 4 , \quad N = 3$

$ND = 4$

$$DEP = \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 4 \end{bmatrix} \qquad IMC = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \qquad DIST = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{bmatrix}$$

Let the arbitrary assignment be given in our notation by the assignment matrix X as

$$X = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

The processing vector and the time spent in IPC of modules are given by

PROCT = (2,2,2,3)

and

IPCT = (3,1,0,0)

There are four dependence pairs, same as the number of arcs in the process graph and only one of these, (3,4), is coresident for the task assignment chosen, giving zero IPCT(3) by definition. Module 1 sends data to modules 2 and 3 at the same IMC value of 1. But, since the distance between processors 1 and 3 , where modules 1 and 3 reside , is 2, IPCT caused by module 3 is 2 whereas it is 1 for module 2, and, their sum gives IPCT(1) = 3.

FIGURE 4.3.3 shows the load density function constructed for this example assignment, where the vertical axis represents the processors and, the time occupied by processing time, IPC time and idle time due to modules is shown along the horizontal time axis. It must be noted that precedence relations given in DEP have governed the construction of the load density function once the assignment is made.

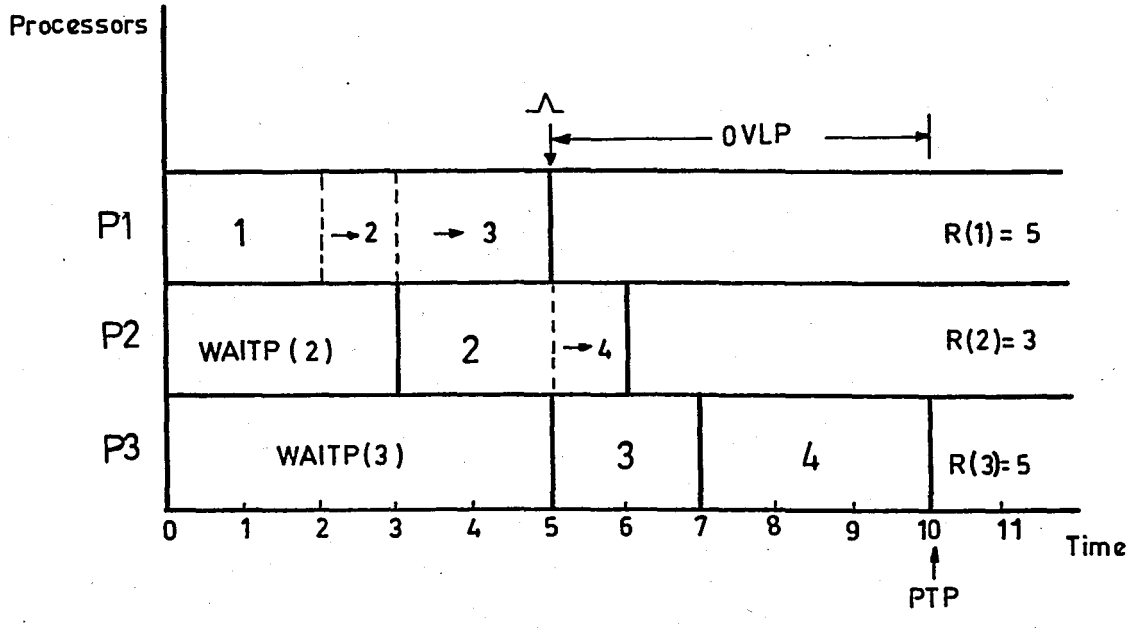


FIGURE 4.3.3 Load Density Function for F-C Example
(M = 4 , N = 3)

The values for PTP and LIP are computed from the start and finish times of the load on processors :

$$PTP = F(3) = FINISH(4,3) = 10$$

and

$$LIP = \max_{k} \{ R(k) \} = \max \{ 5,3,5 \} = 5$$

Then OVLP = 5 and if we repeat the task for five times (K = 5), KPTP = 30 whereas for a single processor T_1 = 9 and K * T_1 = 45. The speed-up is not high, S_{NK} = 1.5 , but the multiprocessor system permits data at a rate 1 / LIP = 0.2 /sec , whereas for the uniprocessor case the rate is 1 / T_1 = 0.1 /sec, twice slower than the former.

4.4 Extension to Partially-Connected Networks

Task assignment for partially-connected networks has complicating differences compared to the fully-connected case. One of the basic differences is that any pair of processors that are not connected directly, has to communicate with the aid of intermediate processors and this naturally depends on the availability of those processors. If available, those processors-on-route will have additional transfer duties apart from their assigned processing and IPC duties. In addition to distances between processors, now a list PROUT containing intermediate processors between each indirectly connected processor pair must be supplied. This increases the complexity of the representation of the hardware component of the model and of the processing for the assignment. This is the situation when there is only one shortest path of intermediate processors between a source and a destination pair.

When the interconnection pattern of the processor network is such that there is more than one path of shortest length connecting a pair, for various reasons such as reliability and better work distribution, we face the problem of alternate routes and the related problem of routing. Routing, in this case, refers to a decision making process in selecting one of the equivalent (in length) alternate paths depending on the availability of processors on route and minimization of objectives. For this case, IMC cost is fixed only for the source and the distance used in IPC computation is the distance to the first intermediate processor although the source-destination pair is farther apart. For convenience, the distance between adjacent processors is taken as unity and for the PMN networks we consider, this distance is uniform for any adjacent pair.

Now, let us assume that an initial assignment of modules to processors is made, alternate routes for non-adjacent communicating processors are analyzed and optimal paths are selected such that the selected processors are assigned transfer duties in addition to module processing and IPC. We may now start computing the components of PTP and accordingly of LIP .

As it is clear from the previous discussion, PROCT and WAITP vectors for the fully-connected formulation remain the same (Equation (4.3.1) and (4.3.3), respectively), but we have to change the IPCT vector, where

$$IPCT(i) = \sum_{j \in R_i} IMC(i,j) * x(i,k) * x(j,l) \qquad (4.4.1)$$

We have to define as well two new vectors, XFER for the transfer operations and WAITX for idle time on processors before the transfer operations. The dimension of both vectors is the same and depends on the number of transfer operations assigned to particular processor during the process of load density function generation.

If we consider transfer operations as assignment-created modules, we can associate a processing time and a wait time with each, similar to the PROCT and WAITP of original modules. XFER denotes the "processing time" of transfer modules, where

$$XFER(i') = IMC(i,j) * x(i,k) * x(j,l) \;;$$

$$i' \in T_{k'} \;;\; k' \in \{1,..,N\} \qquad (4.4.2)$$

for IMC from module i on processor k to module j on processor l, where k' is the intermediate processor which will "execute" the transfer module i' . $T_{k'}$ is the set of all transfer operations (modules) assigned to processor k' .

WAITX defines the waiting time for transfer modules, where

$$WAITX(i') = START(i',k') - FINISH(i'_{-1},k') \qquad (4.4.3)$$

is the time difference between the start time of transfer module i' and finish time of some module (i'_{-1}) assigned to precede i' on processor k', and during which k' is left idle.

We may now compute the finish time on all processors. Finish time F(k) for processor k is given by

$$F(k) = \sum_{i \in A} [PROCT(i)+IPCT(i)+WAITP(i)] + \sum_{i \in T_k} [XFER(i)+WAITX(i)] \qquad (4.4.4)$$

Let us augment the set A corresponding to process modules assigned to k to include elements of T_k (the transfer set of k) as well and name it A^*, augment M - the number of original modules - to M^* to include transfer modules as well, and assume that X is augmented to X^* to enable representation of transfer module assignments such as x(i',k') but X^* still represents the same number of possible assignments as X. Then we can easily obtain equations for PTP_{min} and LIP_{min} as we have done for the assignment problem on fully-connected processors.

For minimum PTP we have,

$$PTP_{min} = \min_{X^*} \{\max_{1 \leq k \leq N} \{\sum_{1 \leq i \leq M^*} [PROCT(i)+IPCT(i)+WAITP(i)+XFER(i)+WAITX(i)]*x(i,k)\}\} \qquad (4.4.5)$$

For LIP, we recall that the difference between F(k) and reserved time R(k) is the start time of first module, some module 1(k) on k, which is equivalent to neglecting the wait time for that module in the summation for the total load on processor k. Here, we have two types of modules that can be 1(k) and two types of waiting times must be considered,accordingly. Then, the optimization problem corresponding to the objective function stated as minimum LIP is given by,

$$LIP_{min} = \min_{X^*} \{\max_{1 \leq k \leq N} \{\sum_{1 \leq i \leq M^*} [PROCT(i)+IPCT(i)+XFER(i)]*x(i,k) + \sum_{\substack{1 \leq i \leq M^* \\ i \neq 1(k)}} [WAITP(i)+WAITX(i)]*x(i,k)\}\} \qquad (4.4.6)$$

Both problems as formulated in (4.4.5) and (4.4.6) are functions of module processing times, IMC between the modules and distance between the processors, module precedence relations, processor availability and initial module-to-processor assignments.

We will present a simple example of task assignment on partially-connected processors. The next chapter on storage representations will prepare us to the algorithmic solution of (4.4.5) and (4.4.6) to be discussed in Chapter 6.

The example process and processor graphs are shown in FIGURE 4.4.1 and we want to assign four modules to three partially-connected processors.

The initial data are given below.

$M = 4$, $N = 3$

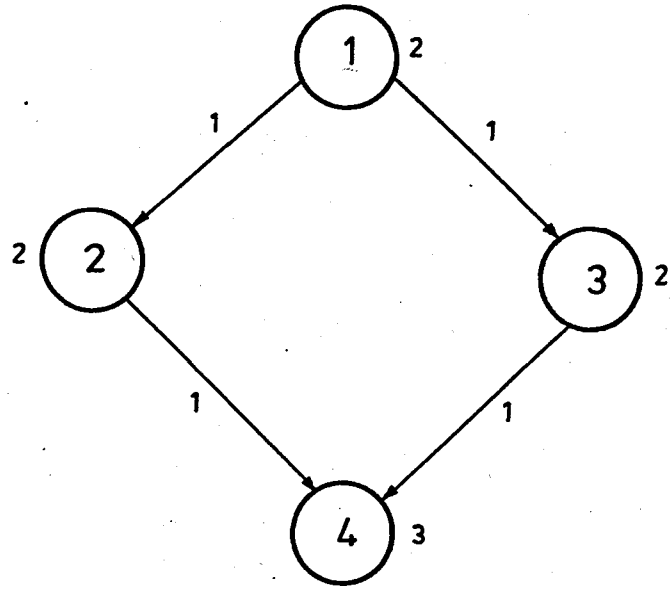
$ND = 4$

$$DEP = \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 4 \\ 3 & 4 \end{bmatrix} \qquad IMC = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \qquad DIST = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{bmatrix}$$

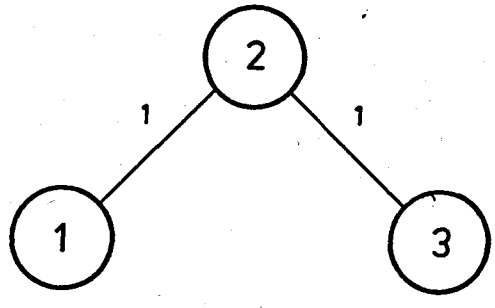
$$ROUT = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \qquad PROUT = 2$$

Let the assignment be given by the matrix

$$X = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$



a)Process Graph



b)Processor Graph

FIGURE 4.4.1 Example Process and Processor Graphs (P-C)

The processing time and IPC vectors are given by,

PROCT = (2,2,2,3)

IPCT = (2,1,0,0)

There is one transfer operation for processor 2, transferring data from module 1 on processor 1 to module 2 on processor 3. For convenience, let us label it as module 5.

Then, XFER(5) = IMC(1,2) = 1

FIGURE 4.4.2 gives the load density function for the example.

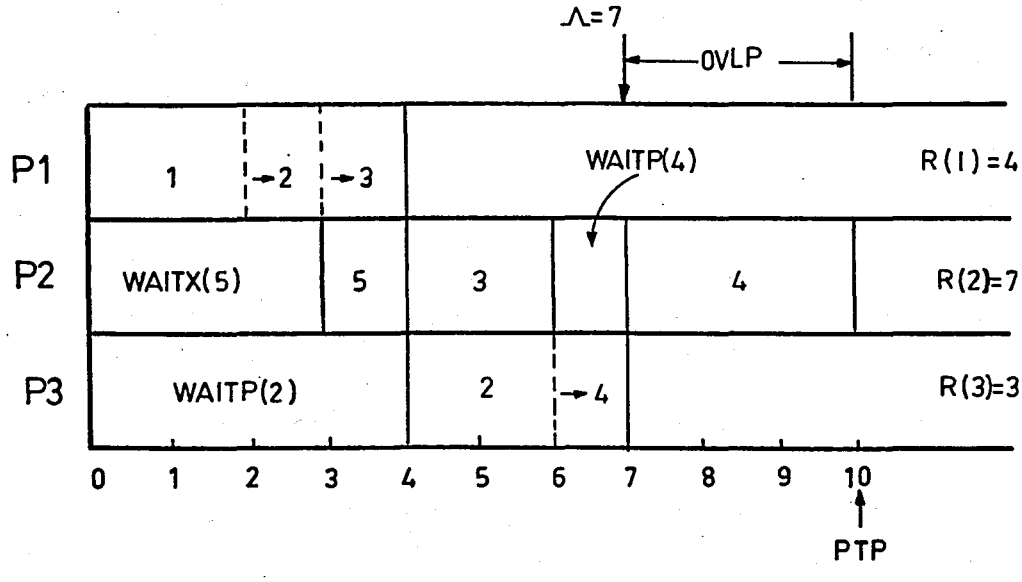


FIGURE 4.4.2 Load Density Function for P-C Example
(M = 4 , N = 3)

5.0
STORAGE REPRESENTATIONS FOR HARDWARE AND SOFTWARE

5.1 Storage Representations for Hardware

Various representation methods can be employed to represent the topology of the processor network. However, one must choose a representation based essentially on the following three criteria:

i) Storage efficiency

ii) Processing efficiency

iii) Characteristics of networks under consideration

There is no priority among the three criteria. We need to find a compromise to satisfy all the three.

For a fully-connected network, only the values corresponding to the distances between the processors in the network suffice to represent the topology. These values may be supplied by a distance matrix $\{ D_{k,l} \}$, where the entries represent the lengths of the links between processors k and l, or by making use of the symmetry, some form of pointer mechanism or linked-list structures may be employed to reduce storage requirements.

For a partially-connected network, however, like the ones we consider in this study, information regarding the processors on-route and the number of alternate routes between any pair of processors that are not directly connected must be supplied in addition to the distance information. Thus, the storage requirements are inversely proportional with the connectedness of the network and we need to find a clever way to represent all the information we require with as little overhead as possible regarding the storage capacity and the processing time.

We now introduce three models capable to represent partially-connected networks, namely the Matrix-Pointer, the Pointer and the Modified Matrix-Pointer models. We present each using an example along which we explain the method and discuss its efficiency.

5.2 Matrix-Pointer Representation

Let us consider an example graph to represent a four-processor partially-connected network. The distance, i.e. the length of a link, between two adjacent processors is taken as unity. The values we must represent are : the interprocessor distances, { D }; the number of alternate shortest paths for each D>1, {NROUT}; and the list of processors on-route, excluding the source, for each alternate route. The network is labeled as shown in FIGURE 5.2.1.

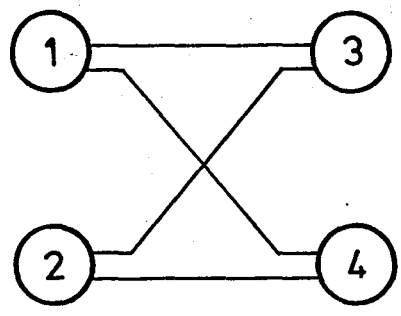


FIGURE 5.2.1 Four-processor Partially-Connected Network

First, we form the distance matrix $\{D_{k,l}\}$, where

$$D_{k,l} = \begin{cases} 0 & ,k = l \\ l_{k,l} & ,k \neq l \end{cases}$$

and PROUT array, to list the processors on-route from k to l including l. PROUT is arranged as shown below :

$$PROUT_{k,l} : (NROUT_{k,l}; proc.list^{1}_{k,l}; \ldots; proc.list^{NROUT_{k,l}}_{k,l})$$

for all $k,l \in S_2$, where $S_2 = \{k,l | D_{k,l} \geq 2\}$ and each processor list contains $D_{k,l}$ processors. NROUT is the number of alternate shortest paths from k to l.

For the network in FIGURE 5.2.1 , $\{D_{k,l}\}$ and {PROUT} are given as,

$$D(k,l)=\begin{bmatrix} 0 & 2 & 1 & 1 \\ 2 & 0 & 1 & 1 \\ 1 & 1 & 0 & 2 \\ 1 & 1 & 2 & 0 \end{bmatrix}$$

PROUT(m) = (2, 3, 2, 4, 2, $1 \rightarrow 2$
2, 3, 1, 4, 1, $2 \rightarrow 1$
2, 1, 4, 2, 4, $3 \rightarrow 4$
2, 1, 3, 2, 3) $4 \rightarrow 3$

1, 6, 11, 16 = m; $NROUT_m$

For PROUT, m is the starting-entry index for the list of processors on the route from k to l, and we introduce another matrix $\{A_{k,l}\}$ for routing information, where

$$A_{k,l} = \begin{cases} m, & \text{index for PROUT for } k,l \in S_2 \\ 0, & k,l \notin S_2 \end{cases}$$

Then $\{A_{k,l}\}$ for the example is given by

$$A(k,l) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 11 \\ 0 & 0 & 16 & 0 \end{bmatrix}$$

If we need to find the relevant values for transfer from 1 to 2 for example, D(1,2)=2 gives us the distance, A(1,2)=1 gives us the index for PROUT, and starting at PROUT(1), we access the routing information such that, there are two alternate paths from 1 to 2, one over processor 3 to 2 and another over processor 4 to 2.

This representation certainly fits the network, unidirectional or bidirectional, and does not lead to processing overhead since during task assignment we can immediately access {D(k,l)} and {A(k,l)} and recording the index m obtained from {A(k,l)} permits unique access to relevant data

for future references for $k,l \in S_2$. However, the storage requirements are not very favourable. For a network with N processors, we need

N^2 locations for {D(k,l)} ,

N^2 locations for {A(k,l)} ,

$$\sum_{m=1}^{R} [NROUT_m * D_m + 1] \text{ locations for } \{PROUT(m)\} ,$$

where $R = |S_2|$, the number of nodes in S_2. In total, we require $2N^2$ locations for the matrices and $o(N^2)$ locations for {PROUT}.

For the example with N=4, 52 locations are required and for an 8-element PON, the representation occupies 186 locations.

Thus the storage requirement for N processors is in the order of $3N^2$ to N^3 . This is a waste especially for bidirectional networks of higher N, where {D(k,l)} is symmetric and we actually need only N(N-1)/2 entries.

Regarding {A(k,l)}, we notice that the number of non-zero entries is R, the total number of processors not reachable in unit distance from each node. If we consider PON's, where the local degree of nodes is 4, i.e.,each node can access at most 4 neighbours, 4N entries out of N^2 are wasted.

Storing {PROUT} necessitates R groups, each of variable size depending on N and network topology, and the number of locations required increases with N. However, the information contained in {PROUT} is essential for task assignment and no reduction in storage seems to be possible for the moment.

5.3 Pointer Representation

We consider the same network of N=4 shown in FIGURE 5.2.1. {PROUT} is formed as explained for the previous method.

We try to reduce the storage requirements for {D(k,l)} and {A(k,l)} at the expense of a small amount of extra processing.

We form (N-1) groups of processors, where each group contains processor pairs (k,l) such that for k=1,2,...,N-1 the index l goes as l=k+1,...,N and we form a base array, {B(k)}, serving as a group pointer and generated using the recursive definition (5.3.1).

$$B(k+1) = B(k) + N-k \qquad (k=1,2,..,N-2)$$

$$B(1) = 0 \qquad (5.3.1)$$

Then the index i for each given k and l in a group is obtained using

$$i = B(k) + l-k \qquad (5.3.2)$$

Before using (5.3.2), pair (k,l) is checked to see if k<l. If k>l, we exchange k and l to index the base array for determining i and set another index j such that

$$j = \begin{cases} 1, & k<l \\ 2, & k>l \end{cases} \qquad (5.3.3)$$

The index i computed using (5.3.2) is used to access a distance array {D(i)}, and pair (i,j) is used to access a routing matrix {A(i,j)} which gives index m for {PROUT}.

The arrays for FIGURE 5.2.1 are given below.

$B(k) = (0,3,5)$; $D(i) = (2,1,1,1,1,2)$;

$$A(i,j) = \begin{bmatrix} 1 & 6 \\ 11 & 16 \end{bmatrix} ;$$

PROUT(m) = (2,3,2,4,2, 1 → 2
2,3,1,4,1, 2 → 1
2,1,4,2,4, 3 → 4
2,1,3,2,3) 4 → 3

Then, if we need to access the information for the transfer 1 → 2, we are given k=1 < l=2 ; j=1, i=B(k)+l-k=1 and we have D(1)=2 and m=A(1,1)=1 for referencing {PROUT}. For the case of transfer 2 → 1,j=2 since k=2 > l=1, i=B(l)+k-l=1 and we have D(1)=2 and m=A(1,2)=6.

Compared to the previous method, the pointer representation requires a small amount of processing in checking (k,l) and in evaluating expressions (5.3.2) and (5.3.3), in order to access {D(i)} and {PROUT}. As before, m can be recorded for future references to the transfer. With this presentation, the model can be used to represent bidirectional partially connected networks. However, unidirectional networks can be handled by adding an extra column to {D(i)} and accessing the entries in a manner similar to that for {A(i,j)}.

The storage requirements are as follows :

(N-1) locations for {B(k)} ,

N(N-1)/2 locations for {D(i)} ,

R locations for {A(i,j)} ,

$$\sum_{m=1}^{R} [NROUT_m * D_m + 1] \text{ locations for } \{PROUT(m)\}.$$

The last component is the same for both methods. If we consider PON's, $R \cong N^2-4N$. Then, excluding {PROUT}, the method requires $[(3/2)N^2-(7/2)N-1]$ locations, much less than $[2N^2]$ of the matrix method.

In this representation the example network requires 33 locations and for an N=8 PON , 125 locations are required. Thus, the pointer method is more efficient than the matrix method as far as storage is concerned.

5.4 Modified Matrix-Pointer Representation

In this method, we have one matrix {DA(k,l)} which is a combination of {D(k,l)} and {A(k,l)} matrices introduced in the first method, such that ,

$$DA(k,l) = \begin{cases} D(k,l), & \text{if } D(k,l) = 0,1 \\ m \quad , & \text{if } D(k,l) > 1 \end{cases}$$

and m is used to index an array {DP(m)} which is a combination of {PROUT(m)} and distance information for D(k,l) > 1. The smallest value of m is 2 and DP(1) = 0 is dummy.

{DA} and {DP} for the 4-processor network of FIGURE 5.2.1 are as follows.

$$DA(k,l) = \begin{bmatrix} 0 & 2 & 1 & 1 \\ 8 & 0 & 1 & 1 \\ 1 & 1 & 0 & 14 \\ 1 & 1 & 20 & 0 \end{bmatrix}$$

	DIST	NROUT	PR.1	PR.2	
DP(m) = (0,	2	2	3,2	4,2 ,	1 → 2
2	2	2	3,1	4,1 ,	2 → 1
8	2	2	1,4	2,4 ,	3 → 4
14	2	2	1,3	2,3)	4 → 3
20					

Given a pair (k,l), if DA(k,l) > 1, we record it as index m to access {DP(m)}, where m gives the distance, (m+1) gives NROUT and so on.

The storage requirements are N^2 for {DA} and R for distance in {DP}, in addition to that of {PROUT}. Excluding {PROUT}, for PON's we require $[2N^2-4N+1]$ locations for N processors. For N=4 and N=8, 41 and 155 locations are required, respectively.

Comparing with the other two, this method of representation seems to be a compromise in storage and processing overheads among the three. It can be used to model both unidirectional and bidirectional networks.

5.5 Assumptions

A comparison of the three methods with respect to storage requirements for 4 and 8 processor networks is given in TABLE 5.5.1. The difficulty arises mostly with the PROUT array.

At this point, it is worthwhile to recall the task assignment problem and review the requirements related to the representation.

Our aim is to assign software modules represented by a precedence graph onto a partially-connected network of processors so as to optimize the finish time of the task and the workload distribution. We know that IPC time is an important component of finish time and to minimize its effect, we need to respect the principle of "locality of communication", that is, communication should be restricted to nearby processors. Moreover, we are not interested in all the alternate paths between each processor pair, but only in the shortest alternate paths between them. This means that if d_1 is the shortest distance between k th and l th processors, we want to consider only NROUT paths of distance d_1, where NROUT is the number of alternate paths of shortest distance. This is in agreement with the type of interconnection schemes introduced earlier, PMN's, especially with PON as an outstanding regular representative (Chapter 2). These networks provide $NROUT_i \geqslant 2$ alternate paths for connections with interprocessor distance $D_i \geqslant 1$, the exact values depending on the number of processors N and the row-column arrangement of nodes. Here D_i is not a distance actually but refers to the number of inter-processor-links (i.p.l). In order to minimize IPC as we have aimed, we obey the principle of locality within a tolerance to permit alternate paths, and reach our first basic assumption.

Assumption 1 : Task assignment strategy should permit IPC between k th and l th processors if and only if $D(k,l) \leqslant 2$, where D(k,l) is the number of i.p.l.'s between processors k and l.

TABLE 5.5.1 Comparison of Storage Requirements for the Hardware-Representation Methods

(* : with $D(k,l) \leq 2$ assumption)

N	Matrix-Pointer Method	Pointer Method	Modified Matrix-Ptr. Method	Modified Pointer Method*
4	D : 16 A : 16 - PROUT : 20 Total : 52	B : 3 D : 6 A : 4 PROUT : 20 Total : 33	DA : 16 DP : 25 - - Total : 41	B : 3 D : 6 - PROUT : 7 Total : 16
8	D : 64 A : 64 - PROUT : 58 Total :186	B : 7 D : 28 A : 32 PROUT : 58 Total :125	DA : 64 DP : 91 - Total :155	B : 7 D : 28 - PROUT : 37 Total : 72

This means that we do not take the principle of locality of communication in the strict sense, i.e., permit communication of adjacent processors ($D(k,l) \leqslant 1$) only, but allow one degree of freedom which in turn allows alternate paths, and enhances the work distribution and the possibility of finding feasible solutions. Insisting on the locality in the strict sense would be to force almost a perfect match between the process graph and the processor graph and might lead to a situation without a solution in an environment, where the network is given and the assignment of an arbitrary task is desired, as we have here. The principle is valid for special-purpose multiprocessors or arrays which are designed for a specific algorithm in mind.

To complete the justification for Assumption 1, we need to consider one more point and see that $D(k,l) \leqslant 2$ is a good compromise between locality and processor accessing capability regarding a solution. First, we state our second assumption for the hardware.

Assumption 2 : The class of partially-connected networks employed in the task assignment process is assumed to have bidirectional links.

We concentrate on bidirectional PON's since they are easily implementable using dual-ported memories and they provide lower average i.p.l's than unidirectional networks as a result of better processor reachability, and permit alternate routes, an important issue in reliability and task assignment. It must be noted that for unidirectional networks, there exist no alternatives for the shortest path when $D(k,l) \leqslant 2$.

We are interested in n_2, the number of nodes reachable from any one node within $D \leqslant 2$. TABLE 5.5.2 lists n_2 , $\%n_2$ and d_{max} for some networks of varying N ,r and c, where r and c are the number of rows and columns in a PON, respectively, and

$$\%n_2 = \frac{n_2}{N-1} * 100$$

TABLE 5.5.2 Processor Reachability within $D \leqslant 2$ for some $N \leqslant 64$

N	r	c	d_{max}	n_2	% n_2
8	4	2	3	6	85.7
12	4	3	3	10	90.9
16	4	4	4	10	66.7
24	8	3	4	12	52.2
32	8	4	4	13	41.9
64	16	4	5	13	20.6

Analysis of the table reveals that n_2 covers more than 50% of the processors for moderately sized processor networks and Assumption 1 is thus justified.

Now, with the simplifications imposed by Assumptions 1 and 2, we propose the following Modified-Pointer model to represent the hardware component of the task assignment model.

5.6 Modified Pointer Model for the Hardware

We refer to Pointer representation presented in Section 5.3. The base array {B(k)} is formed using Equation (5.3.1) and accessed using Equation (5.3.2) if $k \neq l$, as for the Pointer model. We do not need index j since for D = 2, there is only one intermediate processor whether the transfer is $k \rightarrow l$ or in the reverse direction. {D(i)} is accessed using i computed in (5.3.2).

Modification lies in the distance array {D(i)} which is now conveniently used for two purposes, similar to the function of {DA} matrix in the Modified-Pointer representation.

The entries of {D(i)} possess three meanings as given by (5.6.1).

$$D(i) = \begin{cases} 1\ , & \text{if } D(k,l) = 1 \\ m\ , & \text{if } D(k,l) = 2 \\ 0\ , & \text{if } D(k,l) > 2 \end{cases} \qquad (5.6.1)$$

The minimum value for m is m = 2 and PROUT(1) = 0, i.e. dummy similar to DP(1) in Modified Matrix-Pointer method. Index m is used to access {PROUT} as before. Now {PROUT(m)} lists only NROUT and the intermediate processors for D(k,l) = 2 between k and l. The value of NROUT is mostly 2, but depending on N and the configuration, it may vary and we prefer to keep the NROUT entries in {PROUT}.

Total storage requirement for an N-processor PON is

$(N-1)$ locations for {B(k)} ,

$N(N-1)/2$ locations for {D(i)} ,

$\sum_{m-1}^{n_2-4} [NROUT_m+1]+1$ locations for {PROUT(m)} , and

approximately $o(N^2)$ as a whole.

The arrays for the 4-processor network in FIGURE 5.2.1 are as shown below.

$$B(k) = \begin{bmatrix} 0 \\ 3 \\ 5 \end{bmatrix} \quad ; \quad D(i) = \begin{bmatrix} 2 \\ 1 \\ 1 \\ 1 \\ 1 \\ 5 \end{bmatrix}$$

NROUT

PROUT(m) = (0,2,3,4, 1 ⟷ 2
2 ↗ 2,1,2) 3 ⟷ 4
5 ↗

The storage efficiency of the method is contrasted to other methods for N = 4,8 in the last column of TABLE 5.5.1.

5.7 Storage Representation for the Software

The software component of the task assignment model is a directed acyclic graph representing the algorithm, where the nodes correspond to modules of the task and directed arcs joining the nodes indicate the precedence relations between the modules. In the graph representation, the processing time of each module is written next to its node and the IMC values are written as weights on the arcs. The parameters of interest for the representation of the process graph are the processing time of the modules, and IMC and precedence relations between the modules. As previously has been noted, other module attributes, like memory or I/O restrictions, can be easily incorporated if required.

In order to represent the modules, we first label each node in the graph in increasing numerals 1,2,..,M as we scan the graph from left to right and top to bottom, considering precedences. After the modules are numbered, we can represent the processing time of each using a matrix if non - identical processors are used. That is, {PROC(i,k)}, an M x N matrix for M modules and N processors gives the processing times.

PROC(i,k) = processing time of i th module on k th processor

Preferences may be imposed on assignments if processors are not identical. For example, if the code length of a module exceeds the memory capacity of a processor, the use of a very large processing time for that processor may eliminate such an assignment.

Here we concentrate on networks of identical processors and represent processing time requirements of the modules as an array {PROC(i)} where

PROC(i) = processing time of i th module

on all the processors.

Representation of the precedences and the related IMC values between the modules has two basic alternatives : matrix and array .

In the matrix representation of M modules, the connectivity matrix CON(M-1,M-1), giving the precedences, has entries

$$CON(i,j) = \begin{cases} 1 , & \text{if } i \prec j \\ 0 , & \text{otherwise} \end{cases}$$

where i = 1, ..,M-1 ; j = i+1 .

We might use a shortcut to represent IMC values in the same matrix by altering entries CON(i,j) such that now

$$CON(i,j) = \begin{cases} IMC(i,j) , & \text{if } i \prec j \\ 0 , & \text{otherwise} \end{cases}$$

The storage requirement for this representation will be $o(M^2)$ locations. However, the processing requirements are not favourable since we have to scan M^2 elements for dependences, while the actual number of dependent pairs is much less than that.

In the array representation, one way is to provide a list {DEP(i,j)} of dependent pairs only, giving (i,j) for each $i \prec j$ in each row, and another list {IMC(i,j)} of the associated IMC values. The storage requirement is proportional to the number of dependent pairs and processing is more efficient.

Instead of duplicating the source module i for each $i \prec j$, we can group successors of each source module, order the source modules 1,2,..,M and eliminate the specification of the source. If a module has no successors

its successor list entry is zero. The end of each successor list is also identified by zero. We name this array {DSUC}, corresponding to the direct successors of each module. The format of two sample rows of DSUC is shown below, where i precedes modules j, l and r, and M is the terminal module :

Module	DSUC
.	.
.	.
.	.
i	(j l r 0)
.	.
.	.
.	.
M	(0)

The IMC array follows a similar pattern except that, since we will access it using the information in DSUC, the zeroes to mark the end of rows and the last row are not required. Each entry IMC(i,p) gives the value of IMC from module i to its successor j in p th position in the list of its successors in DSUC.

In the process graph, if nodes corresponding to modules i and j are connected by a directed arc, then $i \prec j$ and this relation is specified in DSUC. On the other hand, if $i \prec j$ and $j \prec k$, then k is an indirect successor of i, i.e. $i \prec\prec k$. One-step precedence pairs in DSUC are not sufficient to fully represent the precedence relations and we propose a second array {ISUC}, for indirect successors. The format of one row of ISUC array is as follows, where 0 denotes the end of a row :

	module	indirect successors
ISUC : (	i	k l ..... 0)

To conclude this section, as an example consider the representation of the process graph in FIGURE 5.7.1. The processing times of modules will be represented by

$$PROC(M) = \begin{bmatrix} 3 \\ 2 \\ 2 \\ 4 \\ 2 \\ 4 \end{bmatrix}$$

To represent precedences and IMC, the first two methods produce the following arrays :

Matrix CON (including IMC values) will be as follows :

$$CON = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

DEP and IMC arrays for the first array representation method will be as follows,

ND = 7 ; the number of dependent pairs

$$DEP = \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 4 \\ 2 & 5 \\ 3 & 5 \\ 4 & 6 \\ 5 & 6 \end{bmatrix} \qquad IMC = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 1 \\ 1 \\ 2 \\ 2 \end{bmatrix}$$

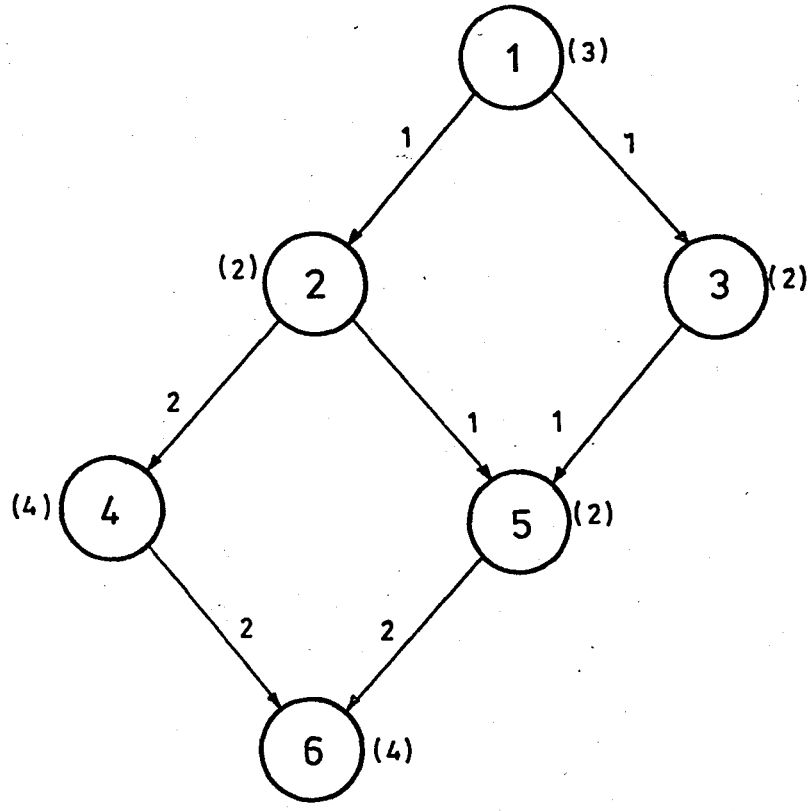


FIGURE 5.7.1 Example Process Graph (M = 6)

The arrays for the proposed representation of the same graph are given below :

$$DSUC = \begin{bmatrix} 2 & 3 & 0 \\ 4 & 5 & 0 \\ 5 & 0 & \\ 6 & 0 & \\ 6 & 0 & \\ 0 & & \end{bmatrix} \qquad IMC = \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 1 & \\ 2 & \\ 2 & \end{bmatrix} \qquad ISUC = \begin{bmatrix} 1 & 4 & 5 & 6 & 0 \\ 2 & 6 & 0 & & \\ 3 & 6 & 0 & & \end{bmatrix}$$

Now, we are ready to explain the task assignment algorithm introduced in Chapter 6, which employes the representations determined in this chapter for the process and processor graphs of the assignment problem.

6.0
TASK ASSIGNMENT ALGORITHM FOR P-C PROCESSOR NETWORKS

6.1 General Description

In this chapter, we present an algorithm to compute the module-to-processor assignment in partially-connected processor networks so as to achieve min.PTP in the single-run environment or min.LIP in the multi-run environment, for a given process and processor graph.

It is an enumerative algorithm based on depth-first search where at each iteration a sample assignment is generated and evaluated to check its performance.

The optimization problems defined by Equation (4.4.5) and Equation (4.4.6) need not have unique solutions. It is possible that multiple assignments yield the same minimal values for PTP or LIP, and the optimal solution space consists of all such alternative assignments. When one is interested in all the optimal solutions, especially for a small system, a possibility is to output every generated so-far-best assignment and let the user pick up the optimal solutions. However, from a practical design viewpoint, just one optimal module-to-processor assignment for each objective is sufficient and this view is adopted in our solution procedure.

For the single-run environment, where the objective is to minimize maximum completion time, we assume that an initial bound, PTP, larger than any possible finish time is given. The first evaluated assignment produces a new value PTPX as the current bound. If PTPX is lower than PTP, the assignment is saved as the so-far-best and PTP:=PTPX. Then,at any time during the execution of the algorithm, PTP has the value corresponding to the assignment that first resulted in this so-far-best PTP value. The algorithm runs until the end, giving one of the assignments that is optimal with respect to PTP.

For the multi-run environment,the objective might be to find assignments that minimize KPTP, given by Equation (4.3.13) as KPTP = (K-1)LIP + PTP , or LIP,if the maximum task repetition frequency is of utmost importance. We are concerned with the latter case. We assume that an initial bound LIP,larger than the expected repetition period is given. This time, any evaluated assignment which yields a current value LIPX that is lower than the last value of LIP is saved, and LIP and PTP bounds are updated with the current values. The final result is an optimal assignment with respect to LIP.

The proposed task assignment process basically involves two phases :

1) Assignment generation phase, and
2) Evaluation phase

For each assignment generated in phase 1 and input to phase 2, an LDF is generated by assigning start and finish times to all the modules on all processors according to the precedence relations between the modules as specified by the directed arcs in the given process graph.

During LDF generation, if there are no indirect transfers requiring the availability of intermediate processors, the generated LDF is complete and we proceed to compute the resultant current values and check the bounds.

If there arise transfer requests, however, we proceed as follows : Since a transfer request arises when the distance D(k,l) between the processors (k,l) of the two communicating modules (i,j) is equal to 2, the source processor is assigned an extra time for IPC, equal to IMC(i,j). The intermediate processor, if available, will also have a transfer duty of the same duration. Then, if all goes well, the destination processor may start executing the dependent module IMC(i,j) time units after the source has finished transmission. This is the transfer interval and we specify it with its start time XS, which is the transmission finish time of the source, and with its finish time XF, which is the earliest time the destination processor may start execution. FIGURE 6.1.1 illustrates parameters of a transfer operation on a partial LDF.

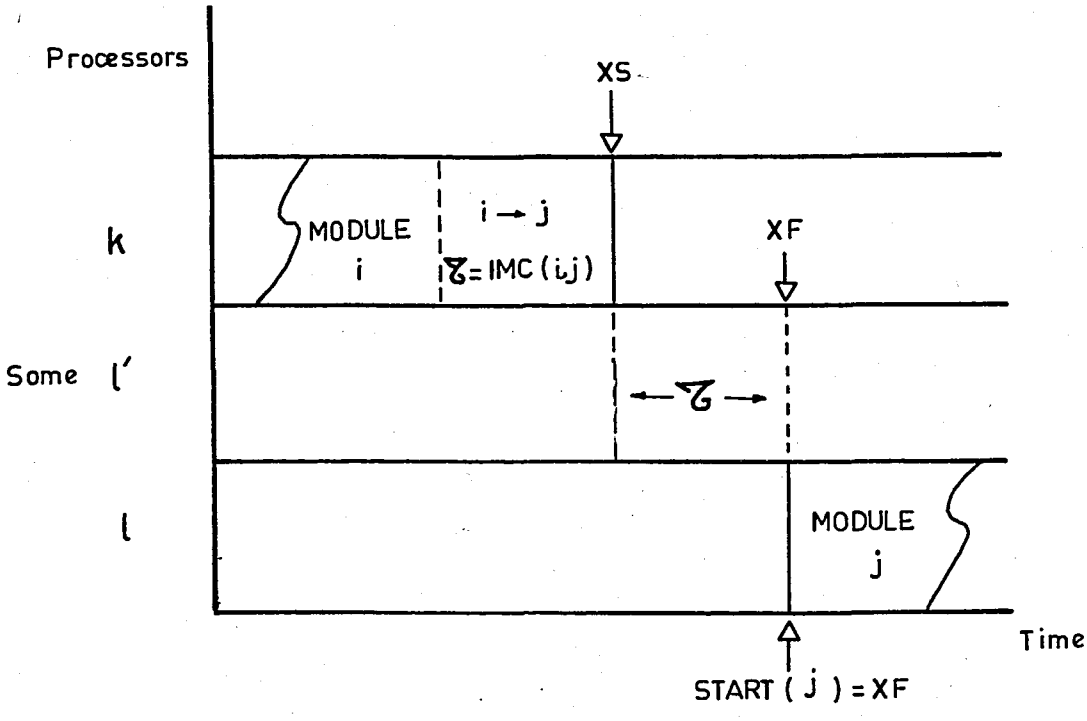


FIGURE 6.1.1 Parameters of Intermediate Transfer

We recall from Section 5.6 that the corresponding entry of the distance array when D(k,l) = 2 gives an index m to address the PROUT array. This means , for example if m = 2, we address the second entry of PROUT arranged as follows.

m	1	2	3	4
PROUT(m)	0	$NROUT_2$	l'_1	l'_2

The number of alternate routes, NROUT, is given by PROUT(m), the first processor on route is given by PROUT(m+1) and the second, alternate processor is given by PROUT(m+2) when NROUT=2. Then, index m, obtained from D(k,l), completely characterizes the processors that may be involved in the transfer. We record m, XS, and XF in a row of a transfer table TX for each transfer request discovered during LDF generation. These values provide all the information required to insert transfer modules on intermediate processors after the partial LDF, i.e., the LDF before the transfer modules are inserted, is complete.

However, when we have a final assignment we will also need to know with which module pair each transfer operation is associated. For this purpose, the pair (i,j) is recorded in a transfer-module array T. Every transfer module is given a label,the first being (M+1). T is part of the arrays required to specify a generated LDF and is saved when the assignment is found to be successful. TX, on the other hand, is not needed after transfer module insertion and can be overwritten.

An LDF is characterized by the sequence of modules on each processor and the start and finish time of each module. If there are indirect transfers, we have additional created-modules assigned to some processors with their sequence and timing information, where their source and destination modules are specified in the T array. This means that, if we want to print out or save an LDF to characterize a successful assignment, a set of arrays have to be considered. To economize on time and space, we assume double workspace for these arrays and use a workspace flag (WSF) such that initially WSF=0 and the assignment is developed in the first workspace. If the bound for this assignment is better than the initial bound, the bounds are updated, WSF is complemented to indicate the second workspace and the assignment is saved until the first assignment, which yields lower bounds. Thus, WSF is complemented after each successful assignment to indicate the alternate workspace for the succeeding iterations and the current LDF is saved. After the final run, the workspace indicated by the complement of WSF contains the optimal assignment, and LDF and PTP/LIP bounds can be printed out.

The algorithm corresponding to this global description consists of seven steps, each of which are described in detail in the following sections.

Sections 6.7 and 6.8 illustrate the use of the algorithm in both environments by examples. In the final sections of this chapter, we discuss the performance characteristics of the task assignment algorithm ; we demonstrate that the algorithm is correct and, discuss its computational requirements.

The iterative structure of the task assignment algorithm is shown in FIGURE 6.1.2 and FIGURE 6.1.3. In Step 0, the process graph, the processor graph and the objectives are defined, and an initial value for PTP/LIP is set. The WSF is initialized.

Steps 1, 2 and 3 generate all sample assignments. Step 1 generates permutations of the order of the modules and for each permutation generated, Step 2 determines the number of modules to be assigned to each processor. Using the number determined for each processor in Step 2 to select the modules from the order generated in Step 1, Step 3 determines which modules are assigned to which processors and initializes the working arrays for the generated assignment.

In Step 4, we determine if the current assignment satisfies precedence and locality constraints. If the constraints are not violated, we have a feasible assignment and we proceed to the next step. Otherwise, the assignment is rejected and we return to Step 2 to generate the next assignment.

In Step 5, we generate the LDF. If there is no indirect communication request, the generated LDF is complete and we compute the bound. If the current bound is better than the former we save the assignment, else reject the assignment and, go to Step 2. If we have transfers recorded in the transfer table, and the computed temporary bound is lower than the former best bound we proceed to Step 6. Else, we reject the assignment and go back to Step 2.

In Step 6, we scan the transfer table and insert the transfer modules on intermediate processors, if available. If a transfer module, either due to nonavailability of a processor or due to no improvement over the last bound, cannot be inserted the assignment is rejected. When all the transfer modules are inserted successfully we compute and check the current bound after insertions. If it is lower, we update the bounds, complement WSF and return to Step 2 for the next assignment.

```
PROCEDURE : BEGIN ;

        STEP0.  Initialize data

                Do ;
        STEP1.  Generate next permutation

                Do ;
        STEP2.  Generate next composition

        STEP3.  Initialize the assignment

        STEP4.  Check constraints

        STEP5.  Generate LDF and check bounds

        STEP6.  Generate final LDF and check bounds

                END ;

                END ;

                Output optimal assignment

                END PROCEDURE ;
```

FIGURE 6.1.2 The P-C Task Assignment Algorithm

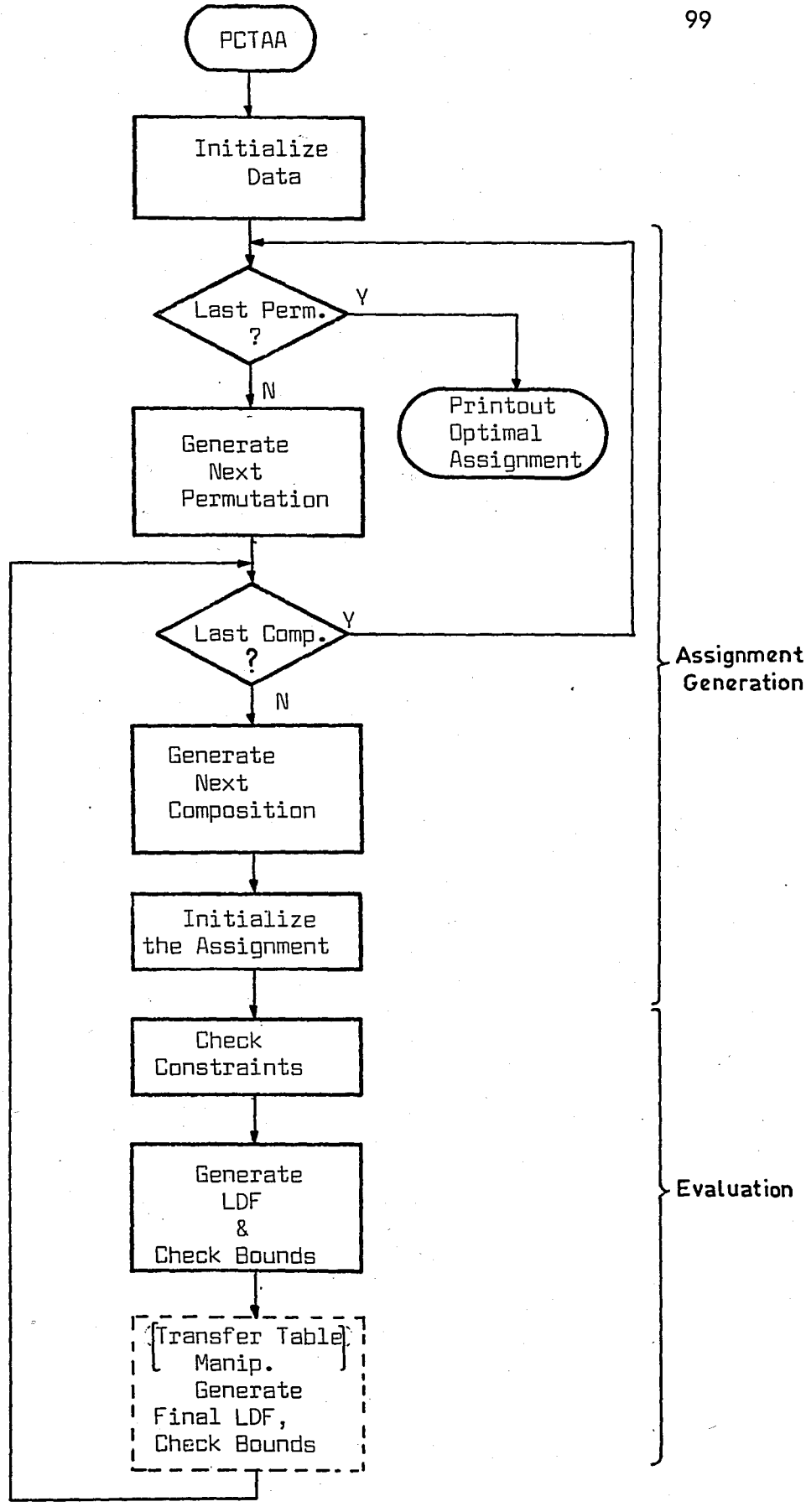


FIGURE 6.1.3 Simplified Flow Diagram of PCTAA

The mechanism of pruning the solution tree is apparent with this description. At all possible points in the algorithm, the process is controlled to see if constraints are violated or if there is no improvement in the bounds, and if so, such assignments are rejected, saving any further computation.

The algorithm terminates after the last composition of the modules for the last permutation, giving the LDF and the bounds for the optimal assignment.

The algorithms for the steps of the task assignment algorithm are provided in the Appendices A to F.

6.2 Initialization

In Step 0, the user enters the system parameters :

For the software description :

M = number of modules

Arrays :

PROC
DSUC
IMC
ISUC

} as defined in Section 5.7

For the hardware description :

N = number of processors

Arrays :

B
D
PROUT

} as defined in Section 5.6

For the objectives :

$$K = \begin{cases} 0 \text{ , for single-run environment} \\ \neq 0 \text{ , for multi-run environment} \end{cases}$$

For K=0 ;

PTP = some number greater than any possible finish time

For K≠0;

LIP = some number larger than the expected repetition period

The workspace flag WSF is set to zero.

(The last permutation flag LASTP is set to zero.)

6.3 Assignment Generation

We use a combination of permutations and compositions to generate every possible assignment.

In Step 1, we generate every permutation of M modules. We assume the processors to be ordered with labels 1,2,..,N and keep the processor order fixed. Then, by permuting the order of the modules and assigning modules to processors in order, such that the first module is assigned to processor 1, the second to processor 1 or 2 depending on the number of modules for each processor to be computed in Step 2, we generate every possible assignment of M modules to N processors.

The permutation step of the algorithm is considered as a subroutine which accepts a value for M and the current permutation, and generates the next permutation of M. The current permutation is stored in a vector A(M). A flag is raised when the last permutation has been generated. When every composition of the last permutation has been generated, then every possible assignment has been enumerated.

In Step 2, we compute the number of modules to be assigned to each processor. This is same as the problem of "distributing M like objects into N unlike cells, with no cell empty" in combinatorial mathematics and it is also called "the composition of M into N parts" [NIJE78].

Given M and N,this step computes a set of integers L(k) where $1<L(k)<M$ and k = 1,2,..,N such that

$$M = \sum_{k=1}^{N} L(k)$$

The value of L(k) corresponds to the number of modules to be assigned to processor k.

The order of compositions is important to ensure that in combination with the permutation of modules, every module is assigned to every processor in every possible order.

For example, assume we have a system of four modules and three processors. The following are the three possible compositions of four things into three parts.

M		L(1)		L(2)		L(3)
4	=	2	+	1	+	1
	=	1	+	2	+	1
	=	1	+	1	+	2

The composition step of the task assignment algorithm is considered as a subroutine which accepts values for M, N and L, and generates next values for L. The first call to the subroutine generates L(1) = M - N + 1 , i.e., the maximum number of modules that can be assigned to a processor, and all the other parts are unity. Subsequent calls change this distribution until the generation of the last composition where L(N) = M - N + 1 and a flag LASTC is raised. After the assignment for the last composition has been generated, the algorithm returns to Step 1 to generate the next permutation and for each permutation, every composition of M into N parts must be generated. When the last composition for the last permutation has been generated, the process is completed.

Step 3 of the algorithm finalizes the module-to-processor assignment generation by assigning L(k) modules from the permuted list A(M) to each processor k in order , k = 1,2,...,N. First, WSF is checked to determine the current workspace and depending on whether it is zero or not, Set0: {C,Y,S,F,T} or Set1: {CC,YY,SS,FF,TT} is used in calling the initialization subroutine.

The following arrays which are used in the allocation procedure are initialized in this step (considering WSF=0) :

C(N) = workspace copy of L(N).

Y(N,C(N)) = It is the assignment array with respect to processors, showing the order of modules assigned to each processor. Each entry Y(k,p(k)) gives the module occupying p(k) th position on k th processor. k= 1,..,N ; p(k)= 1,..,C(k).

X(M) = It is the assignment vector with respect to modules,showing the processor to which the module is assigned. Each entry X(i) gives the processor to which module i is assigned.

O(M) = Each entry O(i) gives the order or position of module i on the processor to which it is assigned. This means that if X(i)= k, then Y(k,O(i))= i.

S(N,C(N)) = Start-time array, aligned same as the Y array. Each entry S(k,O(i)) will correspond to the start time of some module i in O(i) th position on processor k. Initially, the entries are all set to zero.

F(N,C(N)) = Finish-time array, aligned similar to S and Y arrays. Each entry F(k,p(k)), associated with some module i at the same position as in Y, gives finish time of i. Initially, all entries are set to zero.

In the following work, we will interchangeably use S or START for the start-time array and F or FINISH for the finish-time array. When we want to describe the start time of a module i, we will use the notation START(i). Alternatively, we will use START(k,p(k)) to refer to the start time of some module i on processor k at p(k) th position. On the other hand, when we want to refer to start and finish time of processors, we might use START(k), which is actually START(k,1), and FINISH(k) to describe FINISH(k,C(k)). Since most of the time, we use letters i-j to refer to the modules and k-l to refer to the processors, the meaning should be clear.

As an example, consider again M=4 and N=3. The contents of the arrays after Step 3 for the generated sample permutation and composition will be as follows.

A = (1,2,3,4) ; sample permutation
C = (2,1,1) ; sample composition

$$Y = \begin{bmatrix} 1 & 2 \\ 3 & \\ 4 & \end{bmatrix} \qquad S = F = \begin{bmatrix} 0 & 0 \\ 0 & \\ 0 & \end{bmatrix}$$

module i	processor X(i) = k	position O(i)
1	1	1
2	1	2
3	2	1
4	3	1

6.4 Constraint Checking

This part of the algorithm checks feasibility of the generated assignment with respect to the constraints imposed on the problem.

By feasibilty of assignment here we mean and treat two cases :

1- Precedence constraints, i.e. the order of the dependent coresident modules,

2- Locality of communication constraint, i.e. the restriction on the interprocessor distances of the assignment.

Considering case 1 first, we know that if two modules i and j have a precedence relation between them ,i.e. $i \prec j$, and they are also coresident such that X(i)=X(j)=k for some k $\{ 1,..,N \}$, then for a feasible assignment module i should precede module j on processor k. This requires the order of module i to be less than the order of module j, i.e. O(i)<O(j). In the case $X(i) \neq X(j)$, i.e. the modules are not coresident, nothing can be said about the feasibility of orders in obeying the precedence relations.

The precedence relations between two modules can be of two types, as we have mentioned in Section 5.7. If $i \prec j$ this is direct precedence. Since the precedence relation is transitive, i.e. if $i \prec j$ and $j \prec m$ then $i \prec m$, we will call this type of precedence "indirect precedence" and denote as $i \prec\prec m$. The list of direct precedence relations is always supplied as part of the software representation, as in the DSUC array of our model. However, determining indirect precedences from such a list is not very practical, and that is why we have added an ISUC array, giving indirect successors of each module, to our software representation. In a recent study [HOLL82] , precedence check on pairs of indirect precedence is not performed, such an assignment passes to the insertion phase and is rejected after many reinsertions of the pair when the bound is exceeded. We find such a mechanism time consuming and impractical, and thus have included checking the feasibility of indirectly preceding pairs as well.

Once the two arrays DSUC and ISUC are given and the assignment is generated, we check all the pairs in the arrays for coresidence and, if coresident, for their respective positions on the processor; we reject the assignment if any pair $i \prec j$ or $i \prec\prec j$ fails to satisfy the condition $O(i)<O(j)$ on their common processor.

Regarding the feasibility check on interprocessor distances of the assignment, in Assumption 1 (Section 5.5) , we have adopted a rule to satisfy the principle of locality of communication such that the maximum distance allowed between two communicating processors is bounded to the value of 2.

Therefore, if the assignment generated in Step 3, causes dependent modules to be placed on processors that are farther apart than 2 units, we reject the assignment saving further computation.

Then, if we recall that in our representation of the hardware, any $D(k,l)>2$ is represented by $D(k,l) = 0$, where $D(k,l)$ is the distance between k th and l th processors, all we need to do is to scan DSUC array for dependent modules and check if $D(k,l) = 0$ when $k \neq l$, where $k= X(i)$, $l= X(j)$ and $i \prec j$ for pair (i,j).

Generation of assignments and the result of constraint checking for FIGURE 6.7.1 is summarized in TABLE 6.4.1.

It is possible to have constraints other than feasibility of precedence and locality of communication. These might be related to limitations on resources, such as the limited storage capacity of processors or limited I/O capability of processors. Such constraints, if present, can be easily incorporated in the constraint checking algorithm and are very useful in reducing the computational complexity of the task assignment problem since many of the assignments will be rejected at this phase before going into

TABLE 6.4.1 Assignment Generation for FIGURE 6.7.1 (M=4, N=3)

* : infeasible assignment

Perm.n.	Compositions								
	C(1)=2,C(2)=1,C(3)=1			C(1)=1,C(2)=2,C(3)=1			C(1)=1,C(2)=1,C(3)=2		
1234	(12)	(3)	(4)	(1)	(23)	(4)	(1)	(2)	(34)
2134	*(21)	(3)	(4)	(2)	(13)	(4)	(2)	(1)	(34)
3124	*(31)	(2)	(4)	(3)	(12)	(4)	(3)	(1)	(24)
1324	(13)	(2)	(4)	(1)	(32)	(4)	(1)	(3)	(24)
2314	(23)	(1)	(4)	*(2)	(31)	(4)	(2)	(3)	(14)
3214	(32)	(1)	(4)	*(3)	(21)	(4)	(3)	(2)	(14)
4213	*(42)	(1)	(3)	*(4)	(21)	(3)	(4)	(2)	(13)
2413	(24)	(1)	(3)	*(2)	(41)	(3)	(2)	(4)	(13)
1423	(14)	(2)	(3)	*(1)	(42)	(3)	(1)	(4)	(23)
4123	*(41)	(2)	(3)	(4)	(12)	(3)	(4)	(1)	(23)
2143	*(21)	(4)	(3)	(2)	(14)	(3)	*(2)	(1)	(43)
1243	(12)	(4)	(3)	(1)	(24)	(3)	*(1)	(2)	(43)
1342	(13)	(4)	(2)	(1)	(34)	(2)	*(1)	(3)	(42)
3142	*(31)	(4)	(2)	(3)	(14)	(2)	*(3)	(1)	(42)
4132	*(41)	(3)	(2)	(4)	(13)	(2)	(4)	(1)	(32)
1432	(14)	(3)	(2)	*(1)	(43)	(2)	(1)	(4)	(32)
3412	(34)	(1)	(2)	*(3)	(41)	(2)	(3)	(4)	(12)
4312	*(43)	(1)	(2)	*(4)	(31)	(2)	(4)	(3)	(12)
4321	*(43)	(2)	(1)	(4)	(32)	(1)	*(4)	(3)	(21)
3421	(34)	(2)	(1)	*(3)	(42)	(1)	*(3)	(4)	(21)
2431	(24)	(3)	(1)	*(2)	(43)	(1)	*(2)	(4)	(31)
4231	*(42)	(3)	(1)	(4)	(23)	(1)	*(4)	(2)	(31)
3241	(32)	(4)	(1)	(3)	(24)	(1)	*(3)	(2)	(41)
2341	(23)	(4)	(1)	(2)	(34)	(1)	*(2)	(3)	(41)

the allocation phase. For example, given the code lengths of modules, L_i, and the storage capacities of processors, S_k, in identical units, for every assignment it is possible to check if

$$\sum_{j} L_j \leq S_k \; ; \; j \in \{i \mid X(i)=k\} \; ; \; k \in \{1,..,N\}$$

is satisfied and otherwise reject the assignment.

Similarly, for I/O capability of processors, given the I/O attributes of processors as

$$IOP_k = \begin{cases} 1 \; , & \text{if } k \text{ th processor has I/O capability} \\ 0 \; , & \text{otherwise} \end{cases}$$

and the I/O attributes of modules as

$$IOM_i = \begin{cases} 1 \; , & \text{if module } i \text{ requires I/O} \\ 0 \; , & \text{otherwise} \end{cases}$$

the assignments may be rejected at the constraint checking phase if

$$IOP_k * IOM_i = 0 \; ; \; X(i)= k \; ; \; k \in \{1,..,N\} \; ; \; i \in \{1,..,M\}$$

After the constraint checking phase, we have a feasible assignment and we continue to the next step of inserting modules on processors for LDF generation.

6.5 LDF Generation

In the previous step, we have checked that precedence relations on each processor is satisfied. In this step, we insert precedence relations between non-coresident modules and generate an LDF for the assignment by specifying start and finish time of each module on each processor.

In order to satisfy the precedence relations, we scan the DSUC array, which gives successors of each module i (i= 1,..,M) in its i th row.

In the initialization step, the entries in START and FINISH arrays, corresponding to the start and finish times of modules on each processor, have all been initialized to zero. Now, as we scan DSUC we update these entries for each relevant module in order. A flow diagram of GENLDF subroutine for LDF generation is given in Appendix E.

The start time of each module depends on the finish time of its predecessors in the graph and on the communication delay if they are non-coresident, and also on the finish time of the previous module on the same processor if it is not the first module. The finish time of each module is computed as the sum of its start time, processing time, and IPC time, that is,

$$\text{FINISH}(i) = \text{START}(i) + \text{PROC}(i) + \sum_{\substack{i \prec j \\ X(i) \neq X(j)}} \text{IMC}(i,j)$$

and we should have

$$\text{START}(j) \geq \text{FINISH}(i)$$

for any $i \prec j$.

For each module i treated as a source (in communications), we check if it is the first module on its processor k and if so we insert its processing time, that is,

$$\text{FINISH}(i) = \text{START}(i) + \text{PROC}(i)$$

If i is not the first module on processor k, we make sure that

$$START(i) \geq FINISH(i_{-1})$$

where (i_{-1}) denotes the previous module on k, and then compute its finish time.

Processing time of each module is inserted only when it is considered as a source module and after the first time, we just update its finish time by adding the IPC time due to IMC to non-coresident dependent modules.

The modules that are treated as destination modules are assigned appropriate start times, only.

As we scan DSUC, if a pair (i,j) is coresident, i.e. k=l, we simply check if START(j) $\geq$ FINISH(i) to satisfy $i \prec j$. If precedence is not satisfied we update start time of j, i.e. set START(j) = FINISH(i) and proceed to the next successor module in the list.

If j = 0, i.e. either the module has no successor or its successor list is exhausted, we proceed to the next source module.

If a pair (i,j) is not coresident ($k \neq l$), we update the finish time of i by adding IMC(i,j) and check the distance between processors k and l. If the distance m = 1, we proceed as in the coresident case. If the distance is more than one, index m points to PROUT array for the processors on-route between k and l, and a transfer using some intermediate processor is required. We assume at this moment that a processor will be available to carry out the transfer task at the right time. The start time of the transfer is XS = FINISH(i), and the finish time of the transfer is XF = XS + IMC(i,j). If S(j)< XF, then XF is also the start time of module j. We record source-destination pair (i,j) in the transfer-module array T, and the values of m, XS, and XF are saved in the transfer table TX, to be used in the next step of the algorithm.

LDF generation algorithm is very efficient in the sense that just a single scan of the DSUC array is sufficient, in comparison to multiple scans through the modules employed in previous studies.

After all the modules in DSUC are scanned and are assigned start and finish times, using subroutine CBOUND, we compute PTPX and LIPX for this assignment, given by

$$PTPX = \max_{k} \{ FINISH(k,C(k)) \}$$

$$LIPX = \max_{k} \{ FINISH(k,C(k)) - START(k,1) \}$$

where C(k) and 1 denote the last and the first modules, respectively, on processor k.

Then, we check the current bound against the previous bound PTP / LIP.

For K= 0, i.e. single-run environment, if PTPX $\geq$ PTP we reject the assignment. Otherwise, we go to check the transfer table.

For K$\neq$ 0, i.e. multi-run environment, if LIPX$\geq$LIP the assignment will be rejected.

After the bounds are checked and we have a possible assignment, we check the transfer table: if it is empty, we have a final LDF for the assignment; we update PTP and LIP with the current values and save the LDF by complementing WSF, before going back to Step 2 for the generation of the next assignment. If the transfer table is not empty, we have a partial LDF and LIPX is temporary, since we still need to insert the transfer modules indicated in the table. Therefore, we proceed to the next step for the completion of the current assignment.

6.6 Transfer Table Manipulation

After the assignment of process-modules is completed, the entries in the transfer table for transfer-modules have been marked, and current PTPX and LIPX have been computed and checked against PTP and LIP, we start to manipulate the transfer table. For each transfer entry in the table, we will use m to access the PROUT array for the number (NROUT) and the identities (l) of intermediate processors, and XS and XF entries will denote the start and finish times, respectively, for the transfer. For each transfer module, we will try to find an idle slot on its candidate processor(s) matching the transfer interval, insert and name the transfer module, and if we are successful in inserting all the modules listed in the table, the current assignment will be the 'best-so-far'; the bounds for PTP and LIP will be updated and the assignment will be saved.

The insertion of a transfer module involves two basic steps : determining the available processors and among the available processors selecting the one which minimizes the bounds. The first step is the Check-Insertion phase to determine if the intermediate processor is available during the transfer interval, i.e., if the module can be inserted. We consider three cases with respect to XS and XF :

Case 1 : $XF \leqslant START(l,1)$; front-empty.
The module can be inserted before the 1st module on processor l.

Case 2 : $START(l,1) < XF \leqslant FINISH(l,C(l))$; intermediate.
The module can be inserted only if there is an appropriate idle slot among the process-modules.

Case 3 : $XS \geqslant FINISH(l,C(l))$; end-empty.
The module can be inserted after the last module on l.

The three cases are illustrated in FIGURE 6.6.1.

After each processor is checked for insertion, if it is available, it is also checked for the bounds and eliminated at this stage if the bounds are exceeded. Thus "availability" in our terms implies "available and within the bounds" .

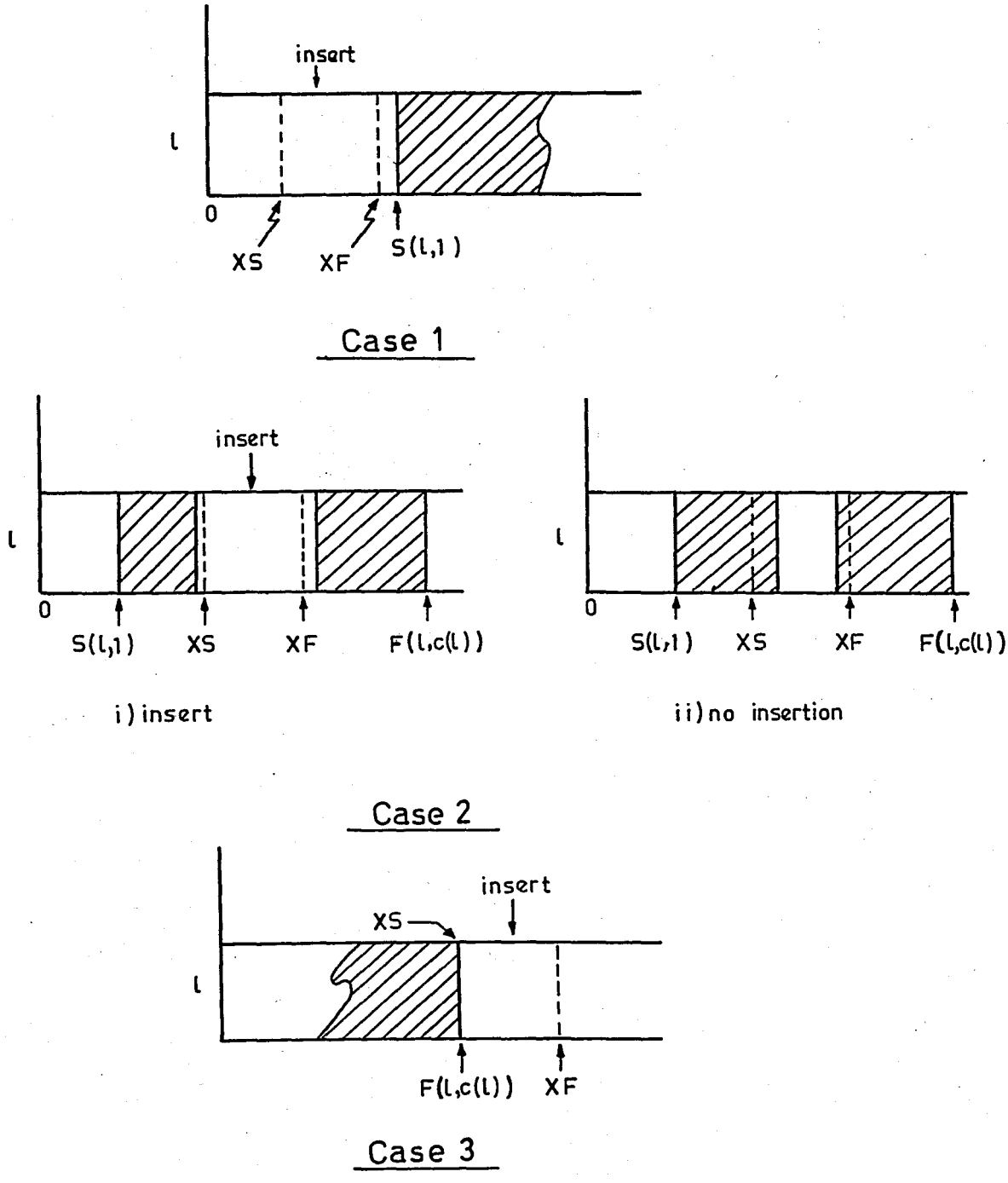


FIGURE 6.6.1 Cases for Transfer Module Insertion

The algorithm for checking the insertion employs a double-purpose flag, INS, to identify successful insertions and to be used in determining the number of available processors as well. At the beginning of each transfer module insertion, INS is initialized to zero, and after each successful check it is incremented by an index (il), which corresponds to the order of intermediate processor l in PROUT. That is, il= 1,2 depending on whether the first or the second processor is checked for availability, for the case NROUT = 2. If there is just one intermediate processor, i.e.,NROUT = 1, then il= 1 for the check. Then, after all the processors have been checked, INS may have the following values with their associated meanings :

$$INS = \begin{cases} 0 \ , & \text{insertion not possible, i.e. invalid assignment} \\ 1 \ , & \text{first processor is available} \\ 2 \ , & \text{second processor is available} \\ 3 \ , & \text{both processors are available} \end{cases}$$

When there is just one processor available, we insert the transfer module on that processor and check the bounds. However, when both processors are available, i.e., have got appropriate idle slots, we have to select one of them based on the problem definition.

We have to notice that, during transfer table manipulation, the insertions do not influence PTPX of the current assignment. For the single-run environment PTPX was checked previously against PTP after process-module assignments such that PTPX < PTP and this PTPX will replace PTP at the end if the insertions are all successful.

However, LIPX for the current assignment may change with the insertions if it happens that the reserved time R(l) on the intermediate processor l increases, such that R(l)>LIPX due to two possibilities :

i) XS < START(l,1) ; transfer module inserted to a slot before the first module.

ii) XF > FINISH(l,C(l)) ; transfer module inserted to a slot at the end

Then, for single-run environment (K= 0) where the objective is to minimize PTP, the maximum completion time, we select the first available processor and update LIPX with the corresponding LIP value. For case i) above R(l) = FINISH(l,C(l)) - XS . For case ii) R(l) = XF - START(l,1) . For either case, if R(l)>LIPX, LIPX is updated. After all the insertions are finished the current PTPX which is less than the previous PTP becomes the new PTP. However, LIPX replacing the previous bound LIP need not be smaller.

In the multi-run environment (K≠ 0), our objective is to find assignments that minimize LIP. For this environment, LIPX was checked against LIP after the assignment of process-modules, such that LIPX<LIP. Then, after we check each processor for insertion, we check if its R(l) has increased LIPX, i.e.,R(l)>LIP. If the bound is exceeded, for NROUT= 2, for the first processor we manipulate the INS flag that was incremented at check-insertion phase and thus indicate that the processor is "unavailable". If the processor is available, i.e., the bound is not exceeded, its R(l) is temporarily saved and the processor which causes a smaller LIPX is selected for insertion. For NROUT= 1 case, if R(l)>LIPX then LIPX is updated and the assignment is rejected if LIPX≥LIP. After all the insertions required by the transfer table entries are completed, LIPX and PTPX become new LIP and PTP values, respectively. On the contrary to the single-run case, here, current LIPX<LIP but PTPX replacing PTP need not be smaller than the previous value.

Every time we insert a transfer-module on the selected processor l, we give to it a module number (i_{max}) where i_{max}= M+1 for the first inserted module and at the completion of an assignment, i_{max}= M^*, i.e., the sum of process and transfer modules. C(l) and l th rows of Y, S and F are updated accordingly.

At the end of any successful assignment, the workspace flag WSF is complemented to point to the alternate workspace and we return to Step 2 for generation of the next assignment.

6.7 Example for Single-Run Environment

In this section, we illustrate the P-C task assignment algorithm by working out a simple example problem. The process and processor graphs for M=4 and N=3 are shown in FIGURE 6.7.1.

First,we initialize the data for the algorithm, corresponding to Step 0.

$$M = 4$$

$$PROC = (1, 1, 1, 1)$$

$$DSUC = \begin{bmatrix} 2 & 3 & 0 \\ 4 & 0 & \\ 4 & 0 & \\ 0 & & \end{bmatrix} \qquad IMC = \begin{bmatrix} 1 & 1 \\ 1 & \\ 1 & \end{bmatrix} \qquad ISUC = \begin{bmatrix} 1 & 4 & 0 \end{bmatrix}$$

$$N = 3$$

$$B = \begin{bmatrix} 0 \\ 2 \end{bmatrix} \qquad D = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \qquad PROUT = (0, \underset{\text{NROUT}}{\overset{\downarrow}{1}}, \underset{\text{L}}{\overset{\uparrow}{2}})$$

$$K = 0 \quad \text{and let} \quad PTP = 8$$

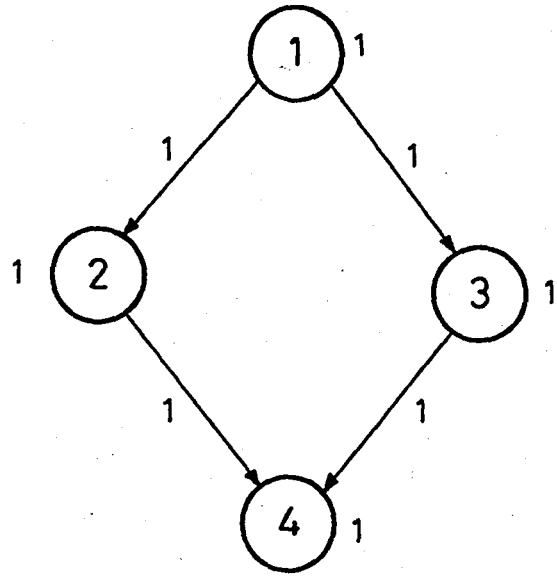
$$WSF = 0 \quad ; \quad LASTP = 0$$

The first permutation (Step 1) of 4 elements is A = (1, 2, 3, 4) and the first composition (Step 2) of 4 modules into 3 parts is L(1) = 2, L(2) = 1, L(3) = 1. We make the assignment by initializing the arrays (Step 3) :

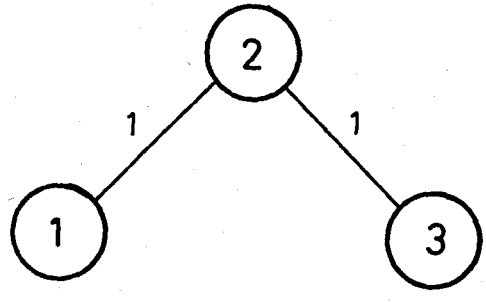
$$C = (2, 1, 1) \qquad X = (1, 1, 2, 3) \qquad O = (1, 2, 1, 1)$$

$$Y = \begin{bmatrix} 1 & 2 \\ 3 & \\ 4 & \end{bmatrix} \qquad S = F = \begin{bmatrix} 0 & 0 \\ 0 & \\ 0 & \end{bmatrix}$$

That is, modules 1 and 2 are assigned to processor 1, module 3 to processor 2 and module 4 to processor 3.



Process Graph (M = 4)



Processor Graph (N = 3)

FIGURE 6.7.1 Example Process and Processor Graphs

We check constraints (Step 4). In this example, interprocessor distance is maximum 2 and the order of coresident pair, (1,2), on processor 1 satisfies precedence. Therefore, we have a feasible assignment and we proceed to Step 5, for generating LDF.

We give a trace of LDF generation for this assignment in TABLE 6.7.1. Initially, it = 1, indicating first index to transfer table.
We scan DSUC :

i=1 ; O(1)=1 so we set F(1)=S(1)+PROC(1)=0+1=1

j=2 ; k=X(i)=1=l=X(j), i.e. coresident, so we check start time of module 2. Since START(2)=0 < FINISH(1)=1, we set S(2)=1. Module 1 has one more successor : module 3 .

j=3 ; l=X(3)=2 and k=1, modules are not coresident. Therefore, we update F(1) for IMC to module 3. IMC(1 -> 3) is given by IMC(1,2) which is 1. Then F(1)= F(1) + IMC(1,2) = 1+1 = 2 . We find the distance for k=1 : B(1)= 0 and l-k = 1 . Then m = D[B(1)+l-k]= D(1)= 1 < 2 . We check start time of 3 : START(3)= 0 < FINISH(1)= 2 and we update start time of module 3 : S(3)= F(1)= 2 . Next entry of DSUC is for module 2.

i=2 ; Module 2 is not the first module on processor 1 and we check its start time against finish time of module 1, which is the preceeding module ; S(2)= 1 < F(1)= 2 and therefore, start and finish times for module 2 are updated : S(2)= F(1)= 2 and F(2)= S(2) + PROC(2)= 2 + 1= 3. Successor of module 2 is 4.

j=4 ; k=1≠l=3 , they are not coresident. We update finish time of 2 for IMC to 4, i.e. F(2)= F(2) + IMC(2,1) = 3+ 1= 4 . We find the distance between processors 1 and 3 which gives m= D(2)= 2. Since m > 1 , we have a transfer module . We mark in the transfer table :

T(1)= [2 4] ; from module 2 to module 4 .
TX(1,1)= 2 ; m
TX(1,2)= 4= XS = F(2)
TX(1,3)= 5= XF = F(2) + IMC(2 → 4)
it := it + 1 = 2

The start time of module 4 is updated since S(4)= 0 :
S(4) = XF = 5 . The next entry in DSUC is for module 3 .

TABLE 6.7.1 Trace of LDF Generation for the Example

MODULE	INITIAL	---------- SCAN OF DSUC ---------->				
		1→2	1→3	2→4	3→4	4→0
1	S(1)=F(1)=0	F(1)=1	F(1)=2			
2	S(2)=F(2)=0	S(2)=1		S(2)=2		
3	S(3)=F(3)=0		S(3)=2	F(2)=3, 4	F(3)=3, 4	
4	S(4)=F(4)=0			S(4)=5		F(4)=6
TRANSFER TABLES: TX T it=1				↓ TX(1,.) =(2,4,5) T(1,.) =(2,4) it=2		

i=3 ; O(3)= 1 therefore we insert its processing time :
F(3)= S(3) + PROC(3) = 2 + 1 =3. We check its successor .

j=4 ; X(4)= 3= l k= X(3)= 2 m= D(k,l)= 1 , so we just update finish time of module 3 , for IMC to module 4 :
F(3)= F(3) + IMC(3,1) = 3 + 1 = 4 . We check start time of module 4 , S(4)= 5 > F(3) .
Last entry in DSUC is for module 4 .

i=4 ; it has no successors and we update its finish time :
F(4) = S(4) + PROC(4) = 5 + 1 = 6 .

Now, since it=2≠1 , we have a transfer entry. We will compute partial bounds and proceed to Step 6 . Partial LDF is shown in FIGURE 6.7.2. with PTPX= 6 and LIPX= 4. For Step 6, i_{max}= 5, first number for a transfer module. Since PTPX < PTP = 8 , we will insert the transfer by a call to XFER : il=1, from PROUT(m)=1 we see that NROUT=1 and l= PROUT(m+1)= 2 . Check-insert algorithm succeeds in XS ≥ F(l,C(l)) test and the insert position is ii= C(l)+1=2 , INS= 1 . Since NROUT= 1 , we compute R for processor 2 and check against LIPX using the UPRL routine. R = 3<LIPX and LIPX remains the same .

We next update the arrays using the routine UPARR . i= 2 > C(2)= 1 . So, simply C(2) = 2

Y(2,2) = i_{max} = 5 ; inserted module

S(2,2) = XS = 4

F(2,2) = XF = 5

There are no other transfer entries in the transfer table. New bounds are PTP=6 and LIP=4 . We complement WSF to save the LDF given by the set { C,Y,S,F,T } and return to Step 2 for the next composition of the same permutation. Final LDF is shown in FIGURE 6.7.3.

After all 24 permutations with 3 compositions are tested, we output the optimal assignment , which has minimum values PTP = 6 and LIP = 4 , for this example .

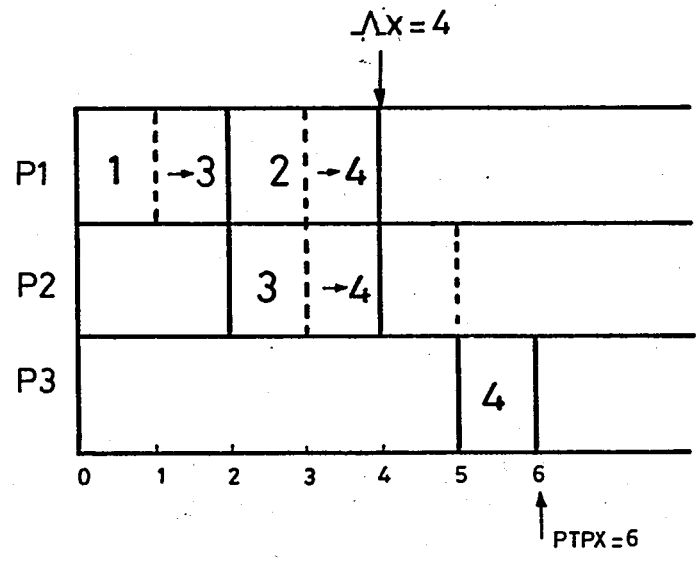


FIGURE 6.7.2 Partial LDF for the Example

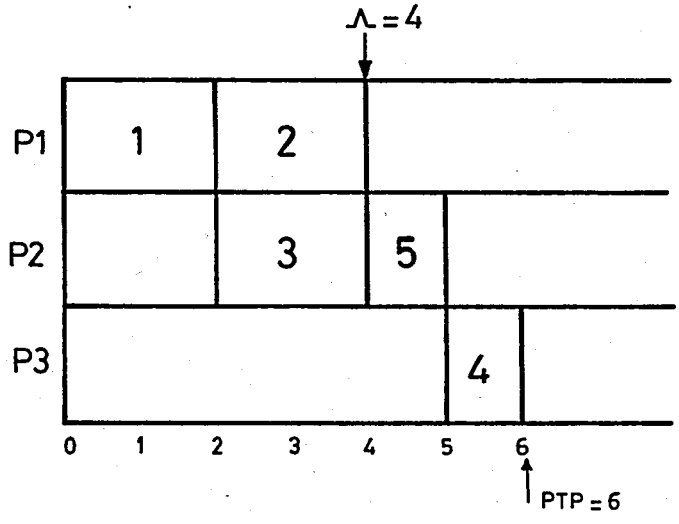


FIGURE 6.7.3 Final LDF for the Example

6.8 Example for the Multi-Run Environment

The example graph we choose is representative of inner-product of two vectors, which when executed iteratively can be used to calculate matrix-vector products, both functions forming the basis for many digital signal processing applications. The example process graph for two 4-vectors and the processor graph of 4 processors are shown in FIGURE 6.8.1.

This time, we will consider two sample assignments only to exemplify the difference between minPTP and minLIP assignments in multi-run or loaded environments.

Assignment X_1 :
Permutation : A = (1, 3, 6, 2, 4, 5, 7, 8)
Composition : C = (1, 2, 1, 4)

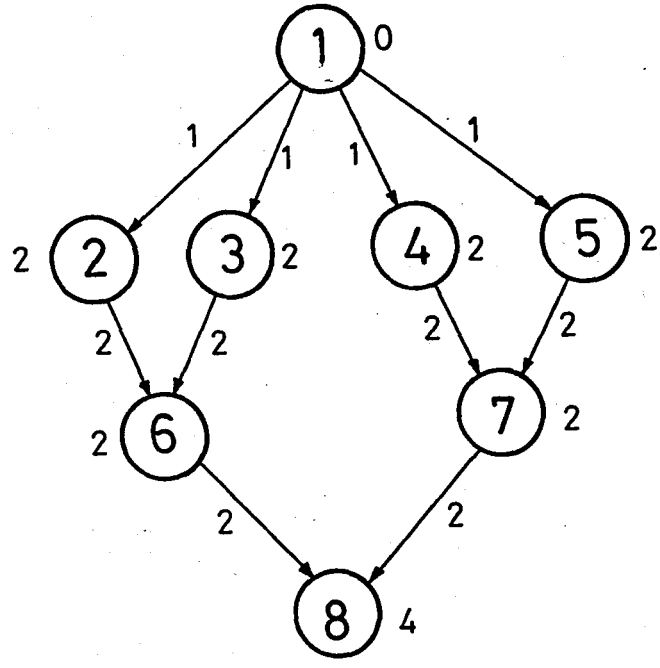
$$Y = \begin{bmatrix} 1 & & & \\ 3 & 6 & & \\ 2 & & & \\ 4 & 5 & 7 & 8 \end{bmatrix}$$

The partial LDF after Step 5 is illustrated in FIGURE 6.8.2. For the transfer 1->3 between the modules from processor 1 to processor 2, NROUT= 2 and l_1= 3 , l_2=4 are the candidate intermediate processors.

The interval is specified by XS=2 and XF=3 . Processor 3 is not available during this interval and CHK-INS routine in XFER returns after first trial since C(3)=1 and, INS=0 . Since NROUT=2 , processor 4 is tested next. CHK-INS routine succeeds in first trial since XF satisfies XF<START(l,1) condition. INS=2 , ii=1 and transfer module 9 is inserted before the first module (module 4) on processor 4, increasing X to 11. The final LDF for X_1 is shown in FIGURE 6.8.3.

Assignment X_2 :
Permutation : A = (1, 8, 4, 5, 7, 2, 3, 6)
Composition : C = (1, 1, 3, 3)

$$Y = \begin{bmatrix} 1 & & \\ 8 & & \\ 4 & 5 & 7 \\ 2 & 3 & 6 \end{bmatrix}$$



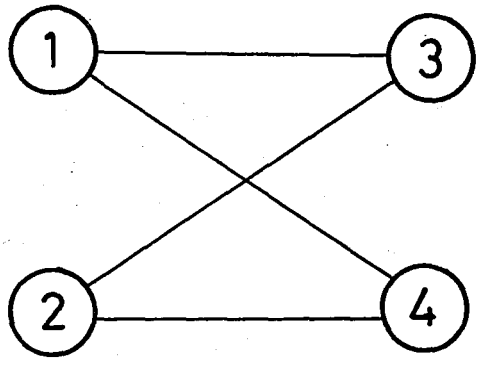


FIGURE 6.8.1 Process and Processor Graphs for the Example

There are no transfers and the LDF generated in Step 5 is final. It is illustrated in FIGURE 6.8.4.

After 4 iterations,

$KPTP_1 = 46$ and,

$KPTP_2 = 39$.

If we specify the problem for minPTP (K=0) , assignment X_1 will be selected as the optimal assignment with PTP = 13 . However, if the problem is specified as a minLIP ($K \neq 0$) assignment, the assignment X_2 will be saved with LIP= 8 . With a repetition period of 8 , X_2 will give an output at constant periods, whereas the port-to-port time of successive iterations for X_1 will continuously increase due to queueing on processors. Even when the repetition rate is higher, requiring LIP < 8 , X_2 will perform better than X_1 . Thus, the use of the minLIP criterion in multi-run or loaded environments, in order to maximize the overlap and minimize the delay due to queueing, is essential and, any performance prediction methodology for analyzing the behaviour of a loaded system should be based on the min assignment of the non-loaded system instead of the minPTP assignment which cannot exploit the benefits due to overlap.

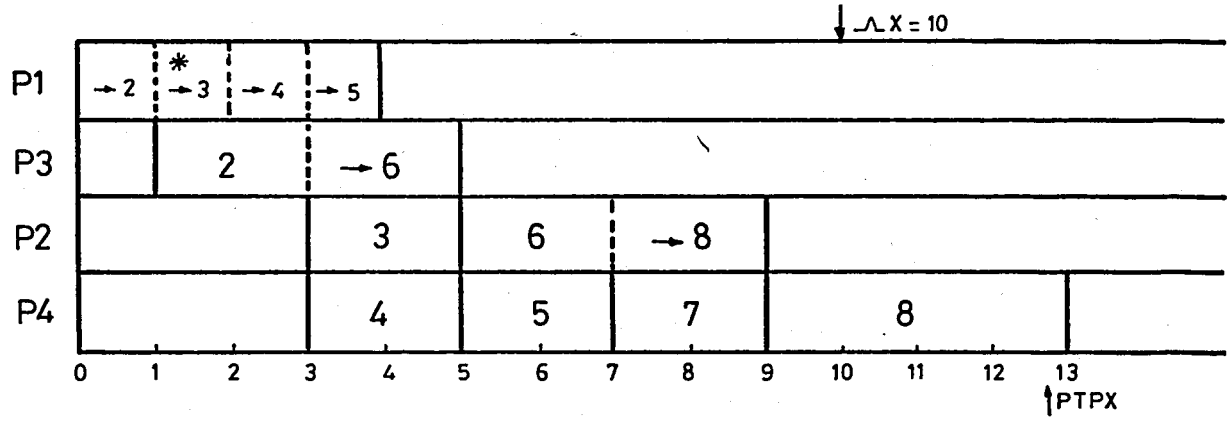


FIGURE 6.8.2 Partial LDF for Assignment X_1

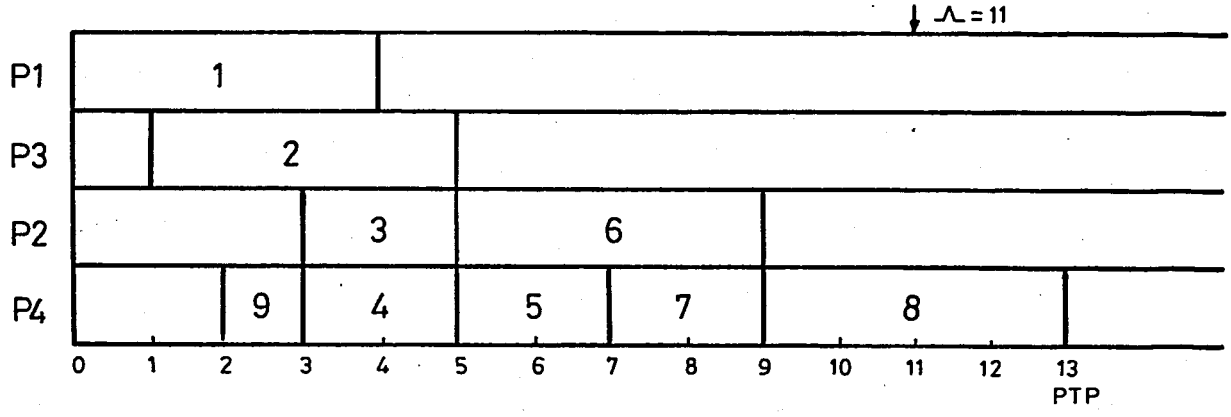


FIGURE 6.8.3 Final LDF for Assignment X_1

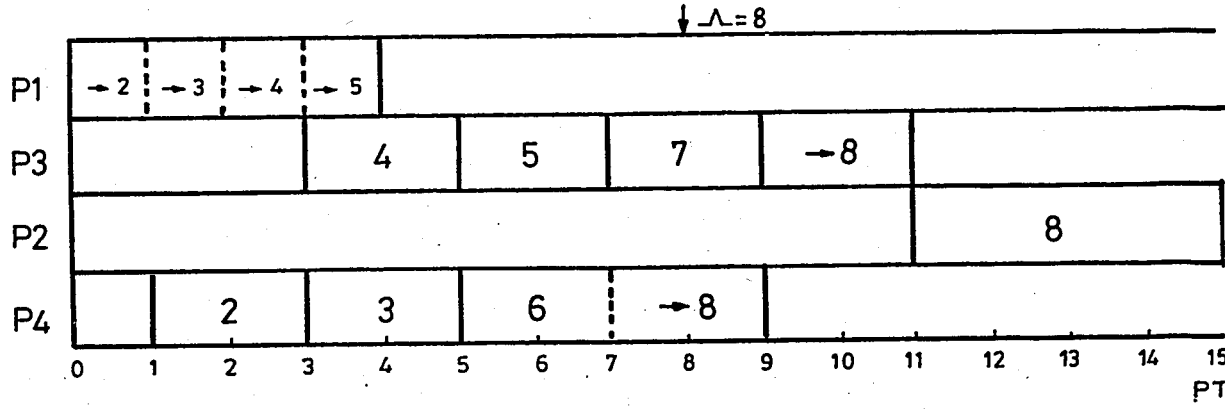


FIGURE 6.8.4 LDF for Assignment X_2

6.9 Verification of PCTAA

In order to verify that the task assignment algorithm for partially-connected networks (PCTAA) is performing correctly we must demonstrate that the algorithm 1) generates every possible assignment, 2) generates LDF for feasible assignments only , 3) terminates , and 4) terminates with the requested minimum PTP or LIP assignment.

It is easy to show that PCTAA generates every possible task assignment. This is handled in Steps 1 and 2 of the algorithm, completed in Step 3. Step 1 (permutations) generates every possible ordering of the M modules. Step 2 (compositions) partitions these ordered modules into N groups for every possible way of grouping M modules into N processors. As we have mentioned previously, we keep the order of N processors fixed while making the assignment. At first sight, since the distances between the processors are different in a partially-connected network, one might suspect that the processors need to be permuted as well. We consider an example with M = 8 and N = 4 to show that this is unnecessary. Using processor permutations, we have the sample assignment :

		Resulting assignment (i) :			
Perm. of modules	: 1,3,6,2,4,5,7,8	Processors :	1	2	3	4
Composition	: 1,2,1,4	Modules :	(36)	(2)	(4578)	(1)
Perm. of processors	: 4,1,2,3					

Using our method with processor order fixed, we have :

		Resulting assignment (ii) :			
Perm. of modules	: 3,6,2,4,5,7,8,1	Processors :	1	2	3	4
Composition	: 2,1,4,1	Modules :	(36)	(2)	(4578)	(1)
Fixed proc. order	: 1,2,3,4					

We see that the resulting assignments (i) and (ii) are the same. Actually, the order in permutations and compositions of case (ii) corresponds to a rotate-left of the order of permutations and compositions of case (i).

Since we generate all permutations of M modules and for each permutation all non-zero compositions of M modules into N processors, permutation of processors is not required explicitly and by keeping the processor order fixed we assign every module to every position on each processor as it travels through all permutations and compositions, using Steps 1,2 and 3 of the algorithm.

Another point of interest is the order of transmissions when a module has to send data to more than one non-coresident module. It is clear that the modules on the critical path should receive their data before the others for timely finish of the task. Recall that we generate LDF in Step 5 by scanning through the DSUC array which has a fixed order. However, owing to the principle of assignment generation, this ordering is taken care of by different permutations, in a manner similar to that of the previous case. Otherwise, we have to treat the modules that receive data in the following dominance order, where (H) and (L) correspond to highest and lowest values, respectively :

1- D_{IN} (H)
2- D_{OUT} (H)
3- IMC_{IN} (L)
4- IMC_{OUT} (H)
5- PROC (H)

which requires reordering the elements of DSUC for every assignment. Fortunately, there is no need to explicitly determine the critical modules and the multiple-transmission ordering.

We now show that the algorithm produces LDF only for feasible assignments, which is equivalent to rejecting all non-feasible assignments. Step 4 of the algorithm checks every generated assignment to see if precedence relations specified in the process graph are satisfied between coresident modules, by a single scan through DSUC and ISUC arrays, which contain all pairs with a direct and indirect precedence relation between

them, respectively. Also, during the scan through DSUC, non-coresident modules are checked for the distance between their processors. To minimize IPC and delays due to nonavailability of intermediate processors, we have imposed the constraint that the interprocessor distance be limited to two, permitting just one intermediate processor for the transfers. The assignments that pass these two tests are feasible within our definition and are forwarded to Step 5 for LDF generation. Any assignment that fails either test is non-feasible and is rejected at this step.

We have shown that PCTAA generates all possible assignments and we get every non-feasible assignment. We now show that the algorithm always terminates. Since the number of permutations and compositions is finite, Steps 1 and 2 terminate after the last composition for the last permutation. Step 3 assigns the modules to processors working on finite arrays. Step 4 performs a single scan of DSUC and ISUC arrays which are finite. Step 5 for LDF generation again performs a single scan of the finite DSUC array to assign start and finish times to modules on processors. If there are any entries in the transfer table TX, Step 6 calls the insertion algorithm XFER a finite number of times for the entries in TX, where one or two intermediate processors, depending on NROUT, are checked for insertion. Thus, the algorithm will always terminate.

At every possible point in the algorithm the current bound, partial or final, is compared to the so-far-best bound and always the assignment which yields a bound lower than the recent-best is saved. Any assignment that is equally well or worse, compared to the recent one for the objective under consideration, is rejected. Therefore, the algorithm terminates with the minPTP or minLIP assignment, as the optimal solution requested.

6.10 Complexity of PCTAA

The computational complexity of the task assignment algorithm is a function of the complexities of each of its steps. Step 0 is the initialization step and involves no computation. The complexity of the remaining six steps is individually discussed in Appendices A - F : Generation of each successive permutation is performed using a single transposition (exchange) of two elements of the previous permutation. Compositions, likewise, increment and decrement only a determined pair of the previous composition. Initialization of the assignment involves a single scan through the modules of the process. The complexity of constraint checking depends on the assignment and possibly many of the generated assignments are rejected at this step. For feasible assignments a complete scan of DSUC is required for constraint checking. LDF generation is very efficient as has already been stressed many times involving just a single scan of DSUC and the process of transfer table manipulation, if required, involves the insertion of just a few transfer modules.

It is clear that although each iteration of the algorithm is efficient, its computational complexity is dominated by the number of iterations of Step 1, the permutations, which exhibits factorial growth on the number of modules. The total number of assignments to be considered for constraint checking in Step 4, for a system of M modules and N processors, is the number of permutations times the number of compositions, given by

$$M! \begin{pmatrix} M-1 \\ M-N \end{pmatrix}$$

This limits the usefulness of the algorithm to problems with a small number of modules.

The next chapter discusses some methods that can be employed to reduce the complexity of the algorithm and extend its use to larger problems.

7.0
SOME METHODS TO REDUCE COMPLEXITY

7.1 Reduction in the Number of Modules

The complexity of the proposed PCTAA increases as a function of the number of modules in the system.

To analyze larger systems, it may be possible to group the modules into clusters to reduce the number of modules for the assignment. An example of clustering is depicted in FIGURE 7.1.1 , where a system of modules is reduced to a system of 4 modules, resulting in 24 permutations for the assignment against 40320 permutations for the original system.

Another possibility is to partition a system and analyze the parts separately or instead of considering all the modules, only time-critical parts may be treated in the assignment process and then integrated into the system. For these special cases the proposed algorithm may be employed without modifications. For other systems, however, the algorithm may be modified as discussed in the next two sections.

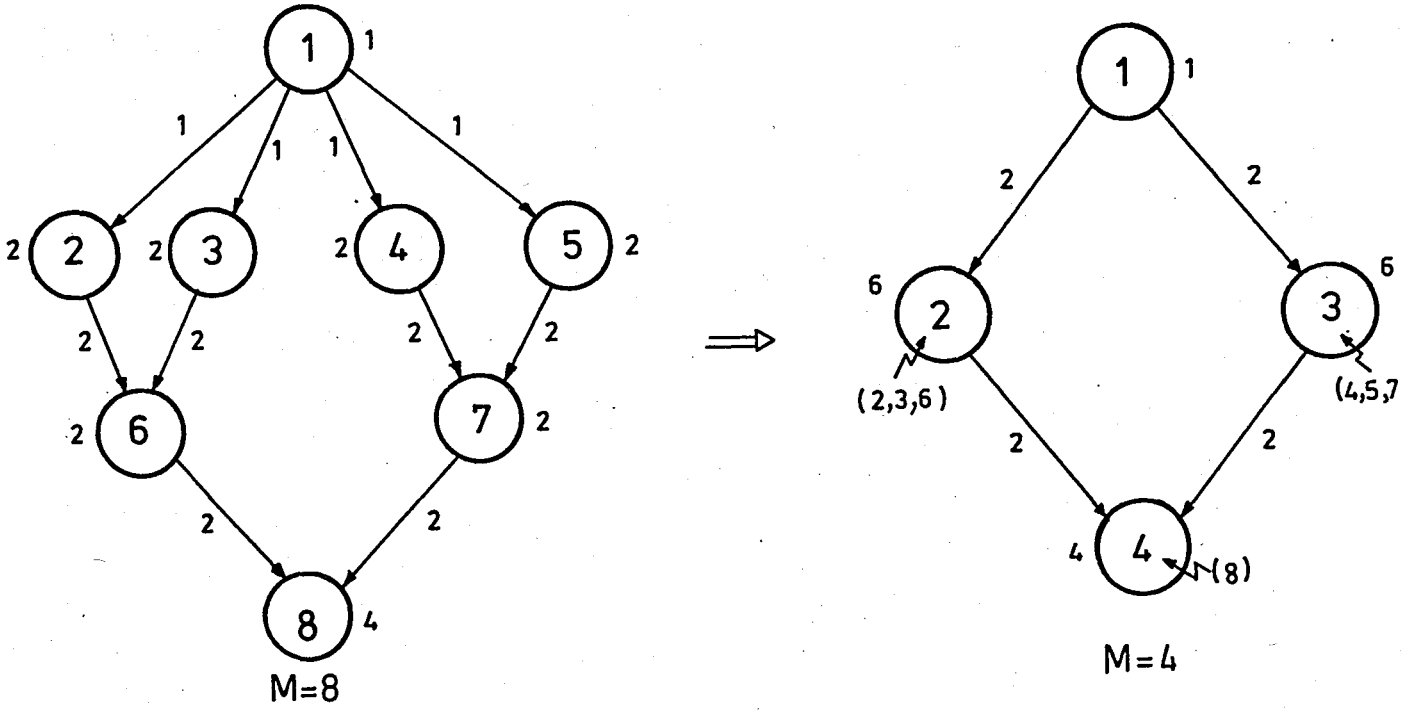


FIGURE 7.1.1 Example of Module Clustering

7.2 Reductions at Constraint Checking Phase

The first reduction technique we proposed is based on the observations of a number of assignments for various task graphs and fully or partially-connected processor networks. As an example we refer to TABLE 7.2.1 corresponding to some feasible assignments for FIGURE 6.8.1.

Observation 1 : The minPTP and minLIP assignments are in the min ΣIMC subset of the feasible assignments, where

$$\Sigma \text{ IMC} = \sum_{\substack{i \prec j \\ X(i) \neq X(j)}} \text{IMC}(i,j) \qquad i = 1,\ldots,M$$

This is an expected situation since minPTP and minLIP are proposed to minimize IPC in the two environments and ΣIMC corresponds to ΣIPC with interprocessor distances of unity. Due to locality restriction, contribution due to distance is at most 2 * IMC's, where ΣIPC is given by

$$\Sigma \text{ IPC} = \sum_{\substack{i \prec j \\ k=X(i) \neq X(j)=l}} \text{IMC}(i,j) * D(k,l)$$

IPC time added to each module processing time influences the waiting period, i.e. the start time, of the successors of the module. Considering that the finish time on each processor is the sum of processing times, IPC times and the idle times depending on the assignment, we see that minimizing IPC time helps to minimize PTP and LIP for balanced assignments since the processing times are fixed.

Observation 2 : The minLIP assignments occur among the min Σ IMC assignments with interprocessor communication being restricted to adjacent processors.

This means that in the multi-run environments we obey the principle of locality of communication in the strict sense. This then results in many

TABLE 7.2.1 Example for the Relation of PTP-LIP and ΣIMC

* :Minimum PTP/LIP among the assignments

Permutation	Composition of Feasible Assign.	ΣIPC	ΣIMC	PTP	LIP
13624578	1, 2, 2, 3	13	10	17	15
	1, 2, 3, 2	17	12	21	19
	2, 1, 2, 3	15	11	17	14
	2, 1, 3, 2	19	13	21	14
	3, 1, 2, 2	18	11	18	12
	3, 2, 1, 2	17	11	Rejected	
	3, 2, 2, 1	17	11	20	14
	1, 2, 1, 4	9	8	* 13	11
	1, 2, 4, 1	13	10	19	17
	2, 1, 1, 4	11	9	17	15
	2, 1, 4, 1	15	11	19	12
	1, 1, 1, 5	11	8	17	16
	3, 1, 1, 3	14	9	16	13
	3, 1, 3, 1	14	9	16	12
	3, 3, 1, 1	20	13	24	18
18457236	1, 1, 3, 3	8	8	15	* 8
	.	.	.	.	.
	.	.	.	.	.
	.	.	.	.	.

simplifications in the algorithm. We have to simplify the hardware representation such that the PROUT array is not required any more and the entries of the distance array are to be modified as given by

$$D(k,l) = \begin{cases} 1 , & \text{if } D(k,l) = 1 \\ 0 , & \text{if } D(k,l) > 1 \end{cases}$$

Also, in GENLDF routine we may omit the D(k,l)>1 test since there can be no transfers, and Step 6 of transfer table manipulation is omitted totally.

We propose a rough initial bound for the sum of IMC's , SIMC , as given by

$$SIMC = \left\lceil \frac{\sum_{i \prec j} IMC(i,j)}{\lfloor M / N \rfloor} \right\rceil$$

A similar initial bound can be used for initializing PTP/LIP , as given by

$$\text{Initial PTP/LIP} = \left\lceil \frac{\sum_{i} PROC(i) + \sum_{i \prec j} IMC(i,j)}{\lfloor M / N \rfloor} \right\rceil$$

Then, at the constraint checking phase, when DSUC is scanned for precedence or locality tests, IMC between non-coresident pairs can be summed and if the sum exceeds SIMC, the assignment can be rejected. Otherwise, SIMC will be updated with the new sum. For the multi-run environment, moreover, locality test will reject any assignment with D(k,l) > 1 .

Considering the number of evaluated assignments at this step, it may be wiser to separate the algorithms for the single-run and the multi-run environments to avoid the environment checking steps which will accumulate

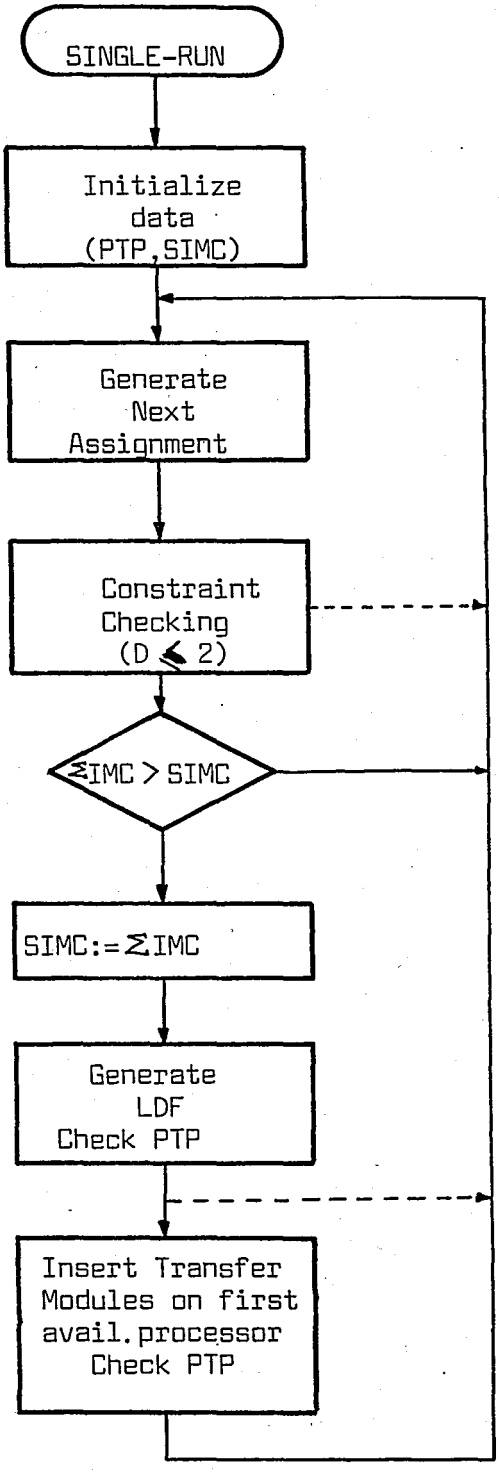


FIGURE 7.2.1 Single-Run PCTAA for Reduction

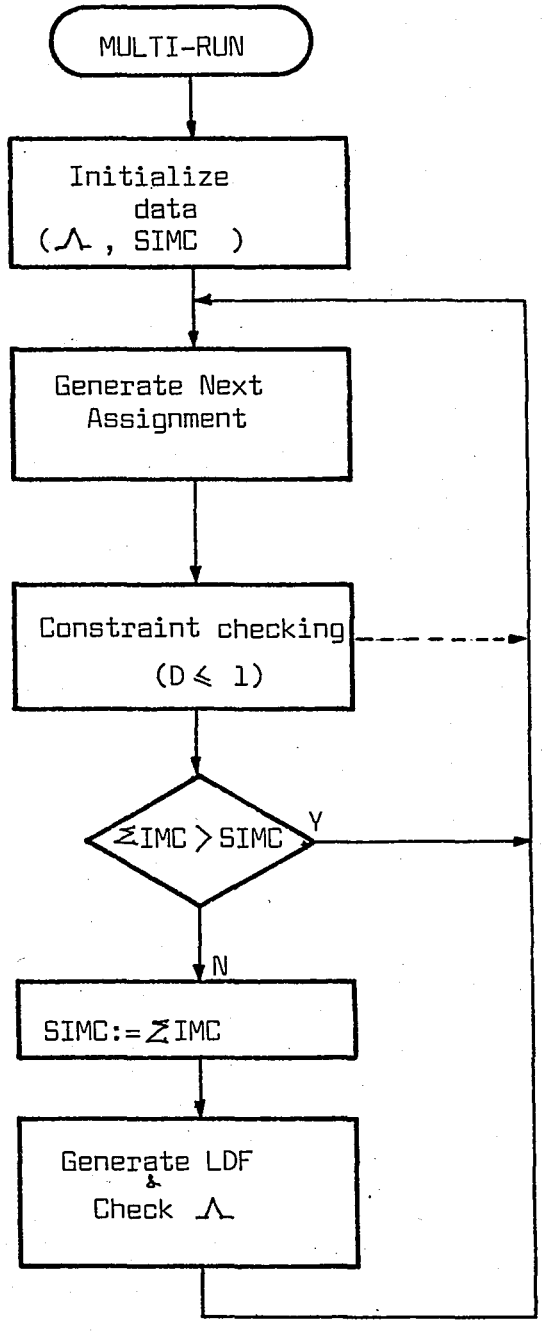


FIGURE 7.2.2 Multi-Run PCTAA for Reduction

to some unnecessary overhead. FIGURE 7.2.1 and FIGURE 7.2.2 outline the algorithms for the two environments.

Other reductions may be possible due to additional constraints. For example, we have mentioned the storage capacities of processors versus the code length, and the I/O problem. If only certain processors have communication capability with the outside world, the start and terminal modules should be assigned to those processors and this can be checked at the constraint checking phase. Also, for SEC graphs in a multi-run environment, we know that the entry and the exit nodes of the process graph should not be assigned to the same processor for minLIP objective.

Many constraints like the ones we have mentioned can be incorporated into the constraint checking phase and help to reduce the number of LDF's generated or transfer tables manipulated (Steps 5 and 6). However, constraint checking is performed for every generated assignment and therefore reducing the number of the generated assignments is highly desirable. This subject is discussed in the next section.

7.3 Reductions in Assignment Generation

In PCTAA we have used permutations and compositions to generate every possible assignment of M modules to N processors. This method of assignment generation is useful in arbitrary partially-connected processor networks. In practice, however, we will need to assign modules of some process graph onto a given network and it may be possible to exploit the topological properties of the network.

When the processor networks under consideration are regular and possess some symmetry property, the task assignment problem can be handled more efficiently. The processor multistage networks (e.g. PON) introduced in Chapter 2 are homogeneous in the sense that each processor has an identical view of the network. Due to this property, for single-entry graphs, the first module can be arbitrarily assigned, for example to processor 1. Then we can ignore module 1 in the permutations and this gives a saving of M iterations. Further, if I/O constraints are imposed on the network, -which is highly probable in practice- , the terminal modules may be left out in the permutations by numbering the processors appropriately. For SEC graphs, for example, fixing the entry in the exit modules gives a total saving of M(M - 1) iterations, which is considerable for large M. For example, for M=8, the number of permutations will reduce from 40320 to 720 with I/O restrictions.

In a homogeneous network, the first processor to be assigned can be any processor of the network, and with our general task assignment procedure, N replications of the same communication pattern will be evaluated, each starting at one of the N processors. The number of times each pattern is replicated corresponds to the number of equivalence classes induced on the set of processors, $D = \{1,2,\ldots,N\}$, by the permutation group of the set, a well-known topic investigated in graph theory, in number theory, or combinatorial analysis in general [BECK64], [LIU68], [BERG71].

Interconnection networks can be considered as functions, each a bijection, i.e. one-to-one and onto mapping, on the set of processor addresses (or numbers). To find the equivalence classes induced by the functions on the set of processors, we can represent the symmetries in the

network by the permutation group of the processors. We will explain using an example. In the 4-processor network in FIGURE 7.3.2 arranged in the form of a square, D = {1,2,3,4} and the permutation group G consists of the following permutations : (Π_1 is the identity element of the group, Π_{2-4} represent rotations and Π_{5-6} represent the symmetry with respect to the diagonals.)

$$\Pi_1 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix} ; \Pi_2 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{pmatrix} ; \Pi_3 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \end{pmatrix} ; \Pi_4 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 2 & 3 \end{pmatrix} ;$$

$$\Pi_5 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 3 & 2 \end{pmatrix} \text{ and } \Pi_6 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 2 & 1 & 4 \end{pmatrix}$$

In terms of cycles, $\Pi_1 \rightarrow (1)^4$; $\Pi_2 \rightarrow (4)^1$; $\Pi_3 \rightarrow (2)^2$; $\Pi_4 \rightarrow (4)^1$; $\Pi_5 \rightarrow (1)^2(2)$ and $\Pi_6 \rightarrow (1)^2(2)$, where the numbers in the brackets represent the cycle lengths.

The cycle index P_G of the group is defined by

$$P_G(x_1, x_2, \ldots, x_k, \ldots) = \frac{1}{|G|} \sum_{\Pi \in G} x_1^{b_1} x_2^{b_2} \ldots x_k^{b_k} \ldots$$

where i denotes the cycle lengths and b_i the multiplicity of each cycle and, for our example, it is given by

$$P_G = \frac{1}{6} \left[x_1^4 + 2x_1^2 x_2 + x_2^2 + 2x_4 \right]$$

Employing Pólya's theory of counting [DEBR64], we obtain the number of equivalence classes of functions from domain D to range R = {1,2,3,4} by evaluating $[P_G]_{x_i=|R|}$, which equals 8 for our example, corresponding to the symmetries of the square. This means that, if we permute the order of the four processors arranged at the vertices of a square, the number of distinct permutations is 24/8 = 3 , for example the set {1234,1243,1324}, and we do not need all of the 24 permutations. Similarly, for an 8 processor PON, where the processors are arranged at the corners of a cube, using the symmetry of the cube with respect to its vertices, the number of identical patterns is 24, requiring 40320/24 = 1680 distinct permutations.

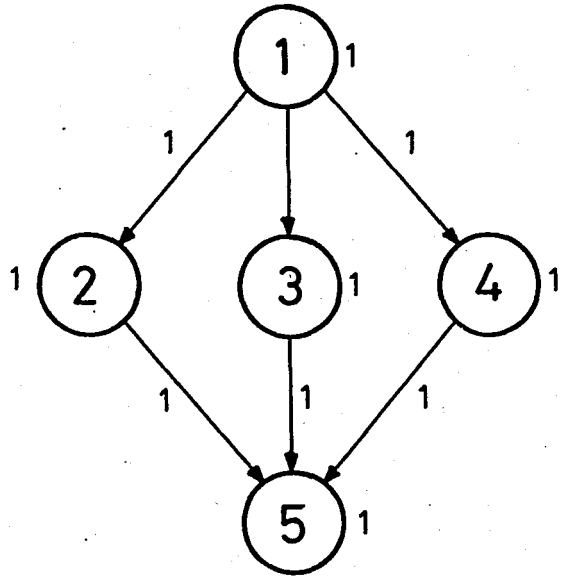


FIGURE 7.3.1 Example Process Graph (M = 5)

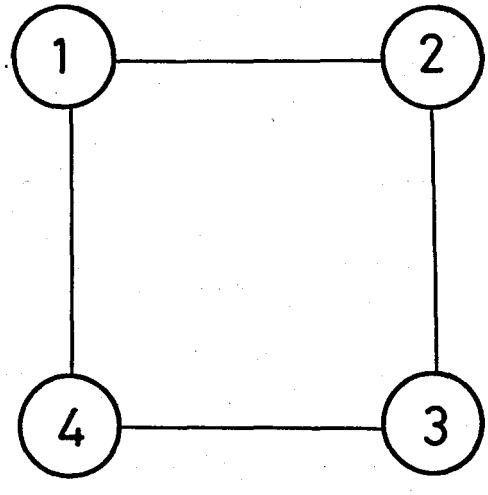


FIGURE 7.3.2 Example Processor Graph (N = 4)

Obviously, extending this method to higher sized networks, determining the required processor permutations themselves and automatization of the permutation generation process require some more work to be done, but these problems will be solved only once for a given network. Then, assuming that this can be done, we can propose the following modified PCTAA for regular networks.

If we would have had N packages of modules to be assigned to N processors, we would have permuted the processors and assigned a package to each. This requires partitioning the set of M modules into N subsets, i.e., the problem of "distributing M distinct objects into N like cells, with no cell empty". The number of k-partitions of an n-set is given by the 'Stirling number'(of the second kind), which satisfies the following recurrence equation

$$S(n,k) = S(n-1,k-1) + k\ S(n-1,k)$$

and identities

$$S(n,1) = S(n,n) = 1$$
$$S(n,k) = 0 \quad \text{for } k > n$$

TABLE 7.3.1 shows S(n,k) up to and including S(9,9).

n / k	1	2	3	4	5	6	7	8	9
1	1								
2	1	1							
3	1	3	1						
4	1	7	6	1					
5	1	15	25	10	1				
6	1	31	90	65	15	1			
7	1	63	301	350	140	21	1		
8	1	127	966	1701	1050	266	28	1	
9	1	255	3025	7770	6951	2646	462	36	1

TABLE 7.3.1 k-partitions of n, S(n,k)

Then, if we consider the module set of FIGURE 7.3.1, where M=5, to be assigned to 4 processors in FIGURE 7.3.2, the number of distinct partitionings is given by,

$$S(5,4) = S(4,3) + 4\ S(4,4) = 6 + 4.1 = 10$$

which corresponds to the 10 partitionings listed in TABLE 7.3.2.

(15) (2) (3) (4)	(1) (24) (3) (5)
(1) (25) (3) (4)	(1) (2) (34) (5)
(1) (2) (35) (4)	(13) (2) (4) (5)
(1) (2) (3) (45)	(1) (23) (4) (5)
(14) (2) (3) (5)	(12) (3) (4) (5)

TABLE 7.3.2 Partitions for the Example
S(5,4) = 10

Let us consider the assignment of the partitioned set {12; 3; 4; 5} to the processors whose order is permuted, keeping the partition including module 1 on processor 1 due to symmetry. Thus we have 3! assignments corresponding to this set.

Processor				
1	2	3	4	
(12)	(3)	(4)	(5)	.....a
(12)	(3)	(5)	(4)	.....b
(12)	(4)	(3)	(5)	c
(12)	(4)	(5)	(3)	.....b
(12)	(5)	(3)	(4)	c
(12)	(5)	(4)	(3)	.....a

Letters a, b, c denote equivalent assignments and we see that indeed we have 3 distinct assignments. Then for the 10 partitionings with 3 processor permutations each, the total number of assignments is 30. With PCTAA

```
PROCEDURE : BEGIN ;

        STEP0.  Initialize data

                Do ;
        STEP1.  Generate next module partition

                Do ;
        STEP2.  Generate next processor permutation

        STEP3.  Initialize the assignment, check constraints

        STEP4.  Generate LDF and check bounds

        STEP5.  Generate final LDF and check bounds

                END ;

                END ;

                Output optimal assignment

                END PROCEDURE ;
```

FIGURE 7.3.3 Modified Task Assignment Algorithm

without any restrictions we will generate $M!\binom{M-1}{M-N} = 480$ assignments. If we restrict module 1 to processor 1, the total number will be 4!(4)= 96. The advantage of the method gained by exploiting the network symmetry is obvious.

Then, if the network topology is known and fixed, distinct permutations on the order of the processors can either be saved in memory for small N or can be generated during the computations using an algorithm. For each given process graph with M nodes, an algorithm to generate S(M,N) partitions of modules in topological order may be employed and the task assignment algorithm may then be modified as shown in FIGURE 7.3.3. It must be noted that, apart from the difference in assignment generation phase,i.e. Steps 1 and 2, the precedence checking part of Step 4 of PCTAA is not required since the ordering of the modules within each partition already obeys the precedence constraints. This brings a reduction due to elimination of duplicate or infeasible assignments. If we employ the method of the previous section, Step 5 will also be eliminated for minLIP assignments. Then such a method will suffice to cover all the required assignments at a lower computational cost, owing to the symmetry of the network.

Since the number of processors is usually much less than the number of modules, even if the network topology cannot be exploited, using I/O constraints, the complexity of permutations can be reduced. The separation of the processes of generating N module packages and the permutation of processors also permits exploitation of the symmetry of the process graph to eliminate equivalences, although the problem seems to be less straightforward than that of the network topology, since the symmetry of a process graph changes dynamically with each assignment, due to the distance-varying contribution of IMC.

Variations or combinations of various methods discussed in this chapter can be applied appropriately in order to reduce the computational complexity and this will allow the analysis and task assignment in larger systems.

8.0
CONCLUSIONS AND RECOMMENDATIONS FOR FURTHER RESEARCH

8.1 Summary and Conclusions

In this study, we have presented a miniaturized image of the macroscopic problems in distributed processing environment, by giving a brief survey of the hardware and software environments.

We have concentrated on two important problem areas :
Interconnection and Task assignment.

For the interconnection of processors, we have introduced PON, a regular, partially-connected, multi-stage processor network which provides:

i) expandability and modularity, requiring a fixed number of connections at each processor,-which is the same as the number of I/O ports-,regardless of the size of the network;

ii) fault-tolerance due to the presence of alternate paths;

iii) homogeneity, which can be exploited in task assignment or in preparing monitor software for the processors;

iv) regular interstage connection pattern, which permits quite a number of row-column alignment patterns and enhances incremental expandability,-as low as 4 processors-, in contrast to other multistage networks with variable interstage patterns, where only particular sizes and alignments are permitted and an increase in the network size can be achieved by doubling the height of the cylinder and by incrementing its circumference by one extra stage;

v) interconnection of each pair of processors in the network without the need for direct paths, a property that makes the network realizable and eases its implementation.

We have derived analytical expressions for some deterministic properties of PON, mainly, the average path length and processor reachability, and compared it to other networks mainly the MSN's and other unidirectional

cube-type interconnection networks. The comparison revealed that for the same number of processors, the average path length of PON is always lower than that of MSN , and reachability is around 75%. A rough comparison of costs also favoured PON for N≥32. Later, while determining the appropriate storage representation of partially-connected networks, higher reachability property of bidirectional PON's, -which permits over 50% reachability within a path length of 2, for moderate sizes-, is exploited to simplify the storage representation which in turn reduced the number of intermediate processors to be tested for availability in the task assignment process. With its simple but powerful structure, PON apparently is a promising candidate for the interconnection of multiprocessors, although further study is required to determine its area complexity if VLSI implementation is of concern.

The second problem we have addressed is the task assignment problem. In any distributed processing environment, with the exception of identical processors forming a fully-connected network of uniform interprocessor distances, proper assignment of the software modules that comprise a task to processors is essential for minimum-time completion of the task, by achieving load balance and minimum interprocessor communication.

The environment we considered is described by a model where the software component, the process graph, is assumed to be a single-entry directed acyclic graph exhibiting the precedence relations between the modules, and the hardware component, the processor graph, is based on the regular interconnection of identical processors that form a partially-connected network with interprocessor distance of unity between adjacent processors.

We distinguished between two operational modes ; single-run where the minimum completion time is of concern, and multi-run where the overlap between successive iterations and the minimum re-initiation time are important, and accordingly, we proposed two different objective functions, minPTP and minLIP, for the two modes, respectively. We have demonstrated that the minLIP criterion introduced in this study is a robust performance measure in the multi-run mode jointly optimizing IPC and load balance, -as

achieved by the minPTP criterion in the single-run mode-, and it outperforms minPTP criterion by maximizing the overlap.

We determined the dominant parameters for the task assignment problem in partially-connected networks to be the precedence relations in the process graph, the interprocessor distances, the number and availability of intermediate processors for indirect transfers, the selection of the proper processor when more than one is available, and the real-time constraints given as minPTP or minLIP.

We formulated the discrete optimization equations for the two environments. The complexity of the problem prevented the use of closed form mathematical optimization techniques and dictated an algorithmic solution, which benefits from additional constraints in reducing the solution space, and can be tailored easily to satisfy varying demands for optimal or suboptimal solutions.

The important steps of the proposed task assignment algorithm are the sample assignment generation, the constraint checking and the LDF generation, which enables description of the generated assignment in graphical form. LDF generation and transfer table manipulation are the unique features of the algorithm and are handled very efficiently using a single scan of the associated list of process modules or the transfer modules. In the constraint checking phase, both direct and indirect precedence relations are checked and communication is restricted to processors with a maximum separation of two links. Any assignment that yields for PTP or LIP a value lower than those of the past assignments is made the new optimal assignment temporarily, and after the final iteration a module-to-processor assignment that is optimal with respect to PTP or is achieved.

We have discussed the performance characteristics of the proposed algorithm : we showed that it generates all possible assignments, generates LDF for feasible assignments only, and it terminates with an optimal assignment. Its computational complexity is mainly a function of the number of modules and hence is useful for small systems.

We have discussed the possibilities for reductions in module number, at the constraint checking phase, and in assignment generation using the symmetries in the network. We have observed that minPTP and minLIP assignments are in the subset of minΣIMC assignments and moreover, minLIP requires strict locality of communication. This simplified LDF generation and eliminated the transfer table manipulation step for minLIP assignments. In order to exploit the symmetry properties of the network in reducing the number of generated assignments, it was necessary to modify the algorithm such that modules and processors are treated separately. This approach also permits reductions due to task symmetry if possible, and enables the analysis of systems with a larger number of modules. The modified task assignment algorithm is also proposed .

In the latter part of this study, we have presented algorithms for various steps of PCTAA and we have not implied any specific language for the actual implementation. We have to note, however, that the computational and coding efficiency of the PCTAA can be greatly improved if many of its segments are implemented in an assembly language. Moreover, the proposed algorithms are based on depth-first search and therefore it might be more efficient to execute them on multiprocessors, the only communication required being the exchange of the most recent best values for PTP or LIP.

In assignment generation, it is assumed that the number of processors, N, is given. However, as it was discussed in the section on related research, this N. might not be optimal. A modification in Step 2 (compositions) of PCTAA in order to permit "empty cells" as well during the distribution of modules to processors will allow generating assignments with $N_{opt} \leqslant N_{avail}$ but at a cost of a factor of N.

8.2 Recommendations for Further Research

Distributed processing is an area of ever-growing interest due to the limited speed achievable with single processors of current semiconductor technology, on one hand, and the increasing demand for higher computational speeds, on the other. However, many issues related to the interconnection and programming of multiprocessors, -each presenting interesting areas for research-, must be treated efficiently in order to realize the potential benefits of distributed processors.

Regarding the topic of processor interconnections, the network presented in this study is regular and easily implementable, but it is expandable at best in increments of four processors. This is acceptable in systems implemented in VLSI, but for distributed systems of multiple microprocessors, increments fewer than those provided with this network might be desirable. Various processor interconnection topologies, for better incremental expandability, or for other possible requirements for specific problems, deserve further study.

Related to the problems in software design for distributed processing, we have mentioned the three interrelated research areas -languages and algorithms, program partitioning, and assignment-, and discussed the assignment problem assuming that the process graph is given. The particular way an algorithm is represented by a process graph and input to the assignment phase, affects the overall performance of the resulting optimal assignment. Development of efficient methods for task partitioning itself in order to achieve better IMC characteristics, or a combined treatment of the partitioning and assignment phases, where the status information of one phase is fed back to the other, might provide better results.

The contribution of IMC to the overall cost is distance-dependent and is determined by each module-to-processor assignment generated. Due to the Saturation effect, the optimal number of processors to be used in a system before the interprocessor communication begins to degrade the system

performance is an important design parameter. Assuming a fully-connected network with unit distance between adjacent processors, it might be possible to compute lower bounds for the completion time, the re-initiation period and the number of processors as a function of the characteristics of a given process graph. However, we have to note that the problem of determining bounds for the general task assignment model is a very difficult one that requires a large amount of experimental work, which is hindered by problem dimensions due to the combinatorial nature of the assignment process.

In our model, it is assumed that each processing element performs both of the tasks of processing and interprocessor communication. When the processing elements are composed of two separate processors, one for each task, LDF generation needs to be modified accordingly, by keeping separate start and finish time arrays for each processor. Also, we have concentrated on conventional structures based on the control flow execution principle. The effect of task assignment in other systems with different execution mechanisms may be investigated with the strong expectation of improved performance.

During LDF generation, we have assumed that the processors operate with a polling mechanism for the input data and each module is assumed to occupy an indivisible time block with its processing and IPC time on its assigned processor. It may be possible to give higher priority to communication such that after the processing time of a module, a transfer operation whose data is available during the module processing time is inserted before the IPC time for that module. Another strategy that can be investigated is the "transmit-first" strategy, where each processor will be assigned IPC times before the processing times, according to the precedences. Many other strategies may be incorporated in the LDF generation phase and this is a useful area to pursue.

The model of the presented algorithm is based on a principle of "tolerant" locality of communication, where the interprocessor communication distance is restricted to two, but, it has been observed that periodic task executions favour "strict" locality, the communication being restricted to adjacent processors. The relation of the concept of tolerant and strict locality of communication to the algorithm-network structure can be investigated further.

For dynamic environments, where the system parameters change sharply over time, efficient dynamic task assignment strategies, which require mechanisms for the measurement of current system state and prediction of future behaviour, and that allow tasks to be re-assigned for optimal performance present another interesting area for research.

Further study in assignment generation methods that exploit the symmetries in the process and the processor graphs in order to avoid duplicate assignment patterns is essential, and this topic seems to present a very interesting research area for the solution of task assignment problem in large systems within a reasonable computational complexity.

BIBLIOGRAPHY

[ACKE82] Ackermann,W.B., " Data Flow Languages ," IEEE Computer, Feb.1982.

[ALLA80] Allan,J.S. and A.E.Oldehoeft, " A Flow Analysis Procedure for the Translation of High-Level Languages to a Data Flow Language ," IEEE Trans.on Comp., Sep.1980.

[ANDE75] Anderson,G.A. and E.D.Jensen, " Computer Interconnection Networks : Taxonomy, Characteristics and Examples ," ACM Computing Surveys, Dec.1975.

[ARDE81] Arden,B.W. and H.Lee, "Analysis of Chordal Ring Network, " IEEE Trans. on Comp., Apr.1981.

[BACK78] Backus,J., " Can Programming Be Liberated From the von-Neumann Style ?! A Functional Style and Its Algebra of Programs ," 1977 ACM Turing Award Lecture, Communications of the ACM, Aug.1978.

[BANE79] Banerjee,U., S.-C.Chen, D.J.Kuck, and R.A.Towle, "Time and Parallel Processor Bounds for Fortran-Like Loops," IEEE Trans. on Comp., Sep.1979.

[BATC76] Batcher,K.E.," The Flip Network in STARAN ," Proc. 1976. Int'l Conf Parallel Processing , Aug.1976.

[BECK64] Beckenbach,E.F.(ed.): Applied Combinatorial Mathematics, John Wiley and Sons, Inc., New York, 1964.

[BELL62] Bellman,R.E. and S.E.Dreyfus, Applied Dynamic Programming, Princeton University Press, 1962.

[BENE65] Benes,V.E., Mathematical Theory of Connecting Networks and Telephone Traffic, Academic Press, 1965.

[BERG71] Berge,C., Principles of Combinatorics, Academic Press, New York, 1971.

[BOKH81] Bokhari,S.H., " On the Mapping Problem ," IEEE Trans. on Comp., Mar.1981.

[BURT81] Burton,F.W. and M.R.Sleep, " Executing Functional Programs on a Virtual Tree of Processors," ACM Conf on Functional P.Lang. and Comp. Architecture, New Hampshire, Oct.1981.

[CHU 69] Chu,W.W., "Optimal File Allocation in a Multiple Computer System," IEEE Trans. on Comp., Oct.1969.

[CHU 80] Chu,W.W., L.J.Holloway, M.T.Lan, and K.Efe, "Task Allocation in Distributed Data Processing," IEEE Trans. on Comp., Nov.1980.

[COFF76] Coffman,E.G.(ed.), Computer and Job-Shop Scheduling Theory, John Wiley and Sons, Inc., 1976.

[CONW67] Conway,R.W., W.L.Maxwell and L.W.Miller, Theory of Scheduling, Addison-Wesley Publ.Co.Inc., 1967.

[DEBR64] de Bruijn,N.G., "Polya's Theory of Counting," in Applied Combinatorial Mathematics, E.F.Beckenbach(ed.), Wiley, New York, 1964.

[DENN79] Dennis,J.B., "The Varieties of Dataflow Computers," Proc 1st Int'l Conf Distributed Computing Systems,Toulouse Oct.1979.

[EL-D80] El-Dessouki,O.I. and W.H.Huen, "Distributed Enumeration on Between Computers," IEEE Trans. on Comp., Sep.1980.

[ENSL77] Enslow,P.H.Jr., "Multiprocessor Organization - A Survey," ACM Computing Surveys, Mar.1977.

[FENG81] Feng,T.-Y., "A Survey of Interconnection Networks," IEEE Computer, Dec.1981.

[FLYN72] Flynn,J.N., "Some Computer Organizations and Their Effectiveness," IEEE Trans. on Comp., Sep.1972.

[FORD64] Ford,L.R. and D.R.Fulkerson, Flows in Networks, Princeton University Press, 1964.

[GAJS82] Gajski,D.D., D.A.Padua, D.J.Kuck and R.H.Kuhn, "A Second Opinion on Data Flow Machines and Languages," IEEE Computer, Feb.1982.

[GONZ77] Gonzalez,M.J.Jr., "Deterministic Processor Scheduling," ACM Computing Surveys, Sep.1977.

[GOOD81] Goodman,J.R. and C.H.Sequin, "Hypertree : A Multiprocessor Interconnection Topology," IEEE Trans. on Comp., Dec.1981.

[GRAH66] Graham,R.L., "Bounds for Certain Multiprocessing Anomalies," Bell Syst. Tech. J. 45, 1966.

[GYLY76] Gylys,V.B. and J.A.Edwards, "Optimal Partitioning of Workload for Distributed Systems," Digest of Papers,COMPCON 76 Fall, 1976.

[HARA69] Harary,F., Graph Theory, Addison-Wesley, 1969.

[HOLL82] Holloway,L.J., "Task Assignment in a Resource Limited Distributed Processing Environment," Ph.D.Dissertation, Computer Science Dept., UCLA, 1982.

[HOR081] Horowitz,E. and A.Zorat, "The Binary Tree as an Interconnection Network : Application to Multiprocessor Systems and VLSI," IEEE Trans. on Comp., Apr.1981.

[HU 61] Hu,T.C., "Parallel Sequencing and Assembly Line Problems," Operations Research, Sep.1961.

[IRAN82] Irani,K.B. and K.-W.Chen, "Minimization of Interprocessor Communication for Parallel Computation," IEEE Trans. on Computers, Nov.1982.

[JENN77] Jenny,C.J., "Process Partitioning in Distributed Systems," Proceedings NTC, 1977.

[JOHN80] Johnson,D., et al., "Automatic Partitioning of Programs in Multiprocessor Systems," COMPCON 80; VLSI; New Horizons, 1980.

[KNUT73] Knuth,D.E., The Art of Computer Programming : Volume I / Fundamental Algorithms, Addison-Wesley, 1973.

[KOHL76] Kohler,W.H. and K.Steiglitz, "Enumerative and Iterative Computational Approaches," in Computer and Job-Shop Scheduling Theory, E.G.Coffman et al., Eds., John Wiley, 1976.

[KUCK72] Kuck,D.J., Y.Muraoka, and S.-Y.Chen, "On the Number of Operations Simultaneously Executable in Fortran-Like Loops and Their Resulting Speed-up," IEEE Trans. on Comp., Dec.1972.

[KUCK77] Kuck,D.J., "A Survey of Parallel Machine Organization and Programming," ACM Computing Surveys, Mar.1977.

[LAWR75] Lawrie,D.K., "Access and Alignment of Data in an Array Processor," IEEE Trans. on Comp., Dec.1975.

[LIU 68] Liu,C.L., Introduction to Combinatorial Mathematics, Mc Graw-Hill, New York, 1968.

[LUND80] Lundstrom,S.F. and G.Barnes, "A Controllable MIMD Architecture," Proc 1980 Int'l Conf Parallel Processing, 1980.

[MA 82] Ma,R.P.-Y., E.Y.S.Lee and M.Tsuchiya, "A Task Allocation Model for Distributed Computing Systems," IEEE Trans. on Computers, Jan.1982.

[MAGO79] Mago,G.A., "A Cellular Computer Architecture for Functional Programming, " Proc IEEE COMPCON 80, New York, Feb.1980.

[MCGR80] Mc Graw,J.R., "Data Flow Computing - Software Development," IEEE Trans. on Comp., Dec.1980.

[MOLD83] Moldovan,D.I., "On the Design of Algorithms for VLSI Systolic Arrays," IEEE Proceedings, Jan.1983.

[NIJE78] Nijenhuis,A. and H.S.Wilf, Combinatorial Algorithms, Academic Press, 1978.

[PADU80] Padua,A.D., D.J.Kuck, and D.H.Lawrie, "High-Speed Multiprocessors and Compilation Techniques," IEEE Trans. on Comp., Sep.1980.

[PARK80] Parker,D.S., "Notes On Shuffle/Exchange-Type Switching Networks," IEEE Trans. on Comp., Sep.1980.

[PEAS77] Pease,M.C., "The Indirect Binary n-Cube Microprocessor Array," IEEE Trans. on Comp., May.1977.

[RAMA72] Ramamoorthy,C.V., K.M.Chandy and M.J.Gonzalez,Jr., "Optimal Scheduling Strategies in a Multiprocessor System," IEEE Trans. on Comp., Feb.1972.

[RAO 79] Rao,G.S., H.S.Stone and T.C.Hu , "Assignment of Tasks in a Distributed Processor System with Limited Memory," IEEE Trans. on Comp.,Apr.1979.

[SEDG77] Sedgewick,R., "Permutation Generation Methods," ACM Computing Surveys, Jun.1977.

[SIEG77] Siegel,H.J., "Analysis Techniques for SIMD Machine Interconnection Networks and the Effect of Processor Address Masks," IEEE Trans. on Comp.,Feb.1977.

[SIEG79] Siegel,H.J., "A Model of SIMD Machines and a Comparison of Various Interconnection Networks," IEEE Trans.on Comp., Dec.1979.

[SIEG81] Siegel,H.J. R.J.Mc Millen, "The Multistage Cube : A Versatile Interconnection Network," IEEE Computer, Dec.1981.

[STON77] Stone,H.S., "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," IEEE Trans.on Software Engineering, Jan.1977.

[TREL82] Treleaven,P.C., D.R.Brownbridge and R.P.Hopkins, "Data-Driven and Demand-Driven Computer Architecture," ACM Computing Surveys, Mar.1982.

[ULLM76] Ullman,J.D., "Complexity of Sequencing Problems," in Computer and Job-Shop Scheduling Theory, E.G.Coffmann et al., Eds.,John Wiley, 1976.

[VEEN81] Veen,A.H., "Reconciling Data Flow Machines and Conventional Languages," CONPAR 81, 1981.

[WATS82] Watson,I. and J.Gurd, "A Practical Dataflow Computer," IEEE Computer, Feb.1982.

[WINT83] Winter,S.C., "A Cube Type Distributed Interconnection of Microcomputing Elements," Microcomputers, Euromicro 1983, North-Holland, 1983.

[WU 80] Wu,C.-L., T.-Y.Feng, "On a Class of Multistage Interconnection Networks," IEEE Trans. on Comp., Aug.1980.

APPENDIX A
ALGORITHM FOR STEP1 PERMUTATION

There are many methods to generate permutations [SEDG77]. The algorithm we choose is taken from [HOLL82] and [NIJE78], where successive permutations of M elements differ only by a transposition. The algorithm is given in FIGURE A.1.

Each permutation A(i), i=1,2,..,M is encoded by an array E(i), i=1,..,M-1 , called an inversion vector, such that E(i) gives the number of elements preceding A(i+1) that are larger than A(i+1). For example, for M=4 three possible permutations are encoded as follows.

A = (1,2,3,4) → E = (0,0,0)
A = (2,1,3,4) → E = (1,0,0)
A = (3,4,2,1) → E = (0,2,3)

If a permutation can be generated from its predecessor by interchanging A(1) and A(2), the signature, SIGN, of such a permutation is defined to be even, which is odd otherwise. The signature is set to be even on first entry and alternates between even and odd with each subsequent entry. Then, for each entry, if SIGN = 1 (even) we simply interchange elements A(1) and A(2) and set SIGN = -1 (odd) before return. If SIGN = -1, a number G(i) is computed to determine which entries to interchange next, and SIGN = 1 before return. When G(i) fails to satisfy the conditions to determine interchange indexes, this means that final permutation has been generated and LASTP = 1.

The average of the total number of computations involved is computed in [NIJE78] to be bounded by M!(2e-2). The total complexity, however, is o(M!).

ALGORITHM PERMUTE

(A) [first entry] A(i) = i , i = 1,..,M ,
SIGN = 1 , Return.

(B) [subsequent entries] If SIGN = -1 go to (C) . Else set
SIGN = -1, interchange A(1) and A(2) , Return.

(C) Set SIGN = 1,
Calculate

$$E(i) = \{ j \mid j \leq i,\ A(j) > A(i+1) \},$$

$$G(i) = \sum_{i=1}^{M-1} E(i) ,$$

until either G(i) is odd and E(i) < i , or G(i) is even and E(i) > 0 . In the first (second) case search A(k), k=1,....,i , for the largest (smallest) number less (greater) than A(i+1) and interchange the two. If A(i) = 1, i = 1,..,M , set LASTP = 1. Return.

FIGURE A.1 Algorithm PERMUTE

APPENDIX B
ALGORITHM FOR STEP2
COMPOSITION

The following algorithm adapted from [NIJE78] generates the next composition of M modules into N parts for the N processors every time it is invoked. Initially, before entry a flag is cleared i.e. LASTC = 0. After all the compositions corresponding to a permutation are generated, LASTC=1. Then, next permutation again resets LASTC = 0, the process repeating until after all compositions for the last permutation are generated.

The number of compositions of M into N non-zero parts is given by

$$\binom{M-1}{M-N}$$

Thus, the complexity of the process of generating all compositions is a function of M and N, and is lower when the value of M is close to N.

ALGORITHM COMPOSE

(A) [first entry] $L(1) = M - N + 1$; $L(k) = 1$, $2 \leqslant k \leqslant N$. Return.

(B) [subsequent entries] $h = \min \{k | L(k) \neq 1\}$; $T = L(h)$;
$L(h) = 1$; $L(1) = T - 1$; $L(h+1) = L(h+1) + 1$.
If $L(N) = M - N + 1$, set LASTC = 1.
Return.

FIGURE B.1 Algorithm COMPOSE

APPENDIX C
ALGORITHM FOR STEP3
INITIALIZATION OF ASSIGNMENT

This step finalizes the assignment generation phase and initializes the working arrays. Its complexity is o(M).

ALGORITHM_INITA

```
Procedure: INITA ;begin
           l:=1 ;  [index to P]
           For k:=1 to N do   [for each processor]
           begin
             C(k):=L(k) ;     [copy composition]
             For j:=1 to C(k) do    [for max.module capacity of k]
             begin
               i:=P(l) ;    [module]
               Y(k,j):=i ;
               O(i):=j ;
               X(i):=k ;
               S(k,j):=0 ;
               F(k,j):=0 ;
               l:=l+1
             end ;  [j]
           end ;  [k]
           end   [INITA]
```

FIGURE C.1 Algorithm INITA

APPENDIX D
ALGORITHM FOR STEP4
CONSTRAINT CHECKING

The feasibility of the assignment is checked using algorithm FEASA, given below. REJ = 0 before entry and is checked upon exit. If REJ = 1 an error return is taken to Step 2.

The complexity of the algorithm is a function of the number of dependent pairs in the process graph, i.e. between o(M) and $o(M^2)$.

ALGORITHM FEASA

(A) For all module pairs (i,j) in DSUC , check :
If (i,j) coresident and O(i) > O(j) go to (C).
If (i,j) non-coresident, check the distance between their processors (k,l) :
If D(k,l) > 2 go to (C).
Otherwise, go to (B).

(B) For all module pairs (i,j) in ISUC , check :
If (i,j) coresident and O(i) > O(j) go to (C).
Else, Return. [normal]

(C) Set REJ = 1 . Return. [error]

FIGURE D.1 Algorithm FEASA

APPENDIX E
ALGORITHMS FOR STEP5
LDF GENERATION

This step scans DSUC array just once and generates an LDF of the assignment by calling a routine GENLDF. It then computes the current bounds. For K = 0, current bound PTPX is compared to PTP. For K ≠ 0, current bound LIPX is compared to LIP. If current bound is not better than the last bound, the assignment is rejected and we return to Step 2. Otherwise, transfer table is checked. If it is empty, i.e. it= 1, the generated LDF is complete and we save the assignment : PTP = PTPX , LIP = LIPX and WSF = $\overline{\text{WSF}}$; and return to Step 2. If it > 1, we proceed to Step 6. Here, we present GENLDF and CBOUND (compute and check bounds) in FIGURE E.1 and FIGURE E.2, respectively.

The complexity of the algorithm is a function of the size of DSUC, i.e. the number of directly dependent module pairs in the process graph. Since for a process graph of M nodes, the maximum number of precedence pairs is M(M-1)/2, it is $o(M^2)$. For SEC graphs, complexity of the algorithm is between o(M) and $o(M^2)$.

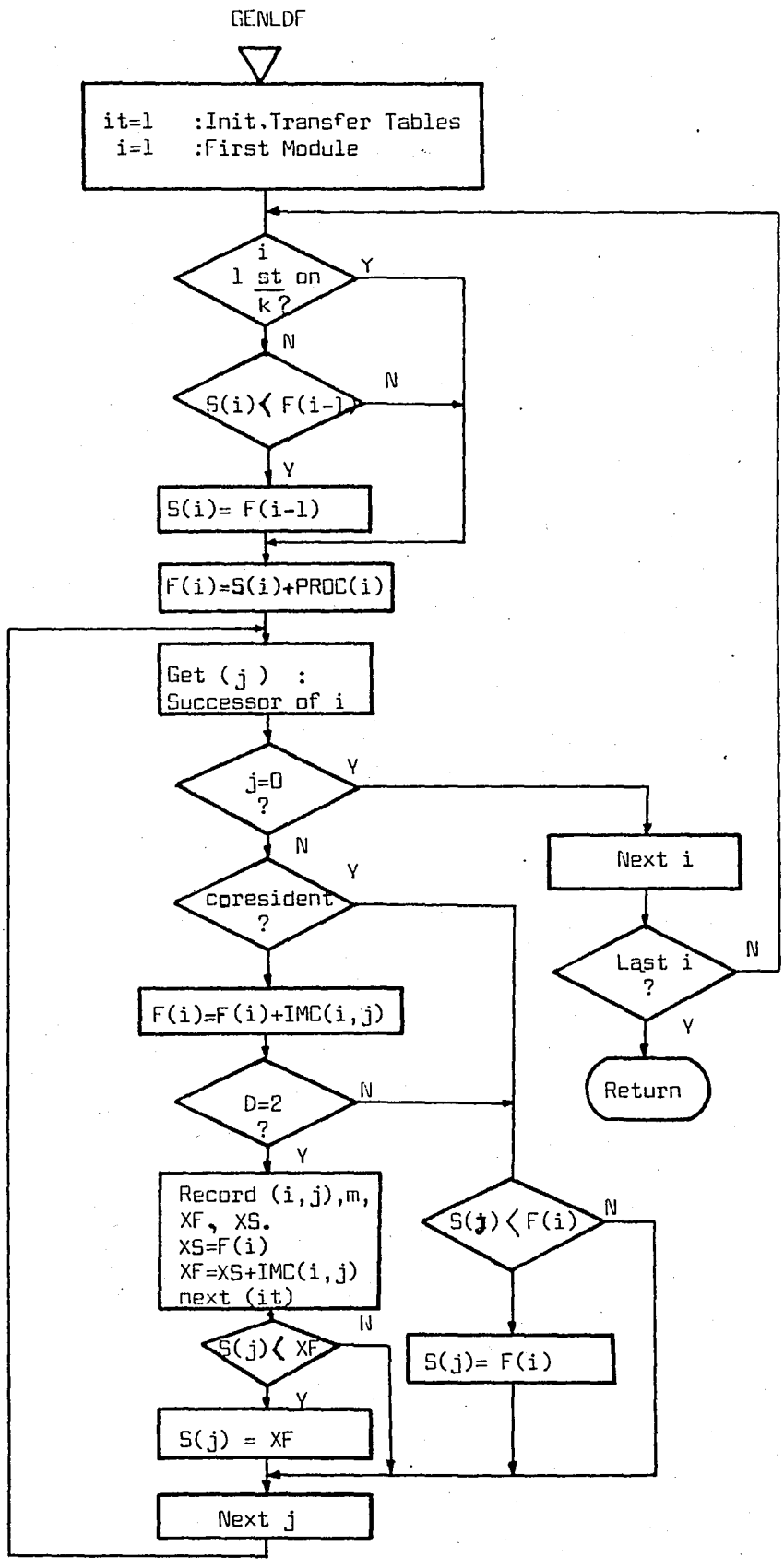


FIGURE E.1 Algorithm GENLDF

ALGORITHM CBOUND

(A) [compute current values PTPX and LIPX]
Set PTPX = F(1,C(1)) and
LIPX = F(1,C(1)) - S(1,1).
Then for k = 2,...,N check and compute :
If PTPX < F(k,C(k)) , set PTPX = F(k,C(k)).
If LIPX < F(k,C(k)) - S(k,1) , set LIPX = F(k,C(k)) - S(k,1).
Then, go to (B).

(B) If K = 0 and PTPX $\geq$ PTP or
If K $\neq$ 0 and LIPX $\geq$ LIP then set REJ = 1
[reject the assignment], Return.
Otherwise, Return.

FIGURE E.2 Algorithm CBOUND

APPENDIX F
ALGORITHMS FOR STEP6
TRANSFER TABLE MANIPULATION

This step scans the transfer table and for each entry, an algorithm XFER is called in order to insert the transfer module on available processors. A flag REJ is initialized to zero for each assignment and is tested upon each return. If REJ = 1, the current assignment is not valid and we return to Step 2. If REJ = 0 after all insertions, we have an optimal assignment candidate : PTP = PTPX , LIP = LIPX and WSF = $\overline{WSF}$. Then, we go to Step 2 for the next assignment.

We present a flow diagram in FIGURE F.1 for the algorithm XFER. The three algorithms used by XFER, namely, 1) CHK-INS, checks insertion , 2) UPRL, updates R and LIPX after each possible insertion, and 3)UPARR, updates { C,Y,S,F } arrays after insertion ; are presented in figures F.2 to F.4.

The complexity of the algorithm depends on the number of transfer table entries and NROUT. NROUT is at most 2 and for assignments that have not been rejected up to this step, the number of transfer modules is usually small. Since the first available processor is accepted in single-run operation mode, its complexity is negligible, whereas in multi-run mode at most two processors have to be checked for insertion and minimum LIPX.

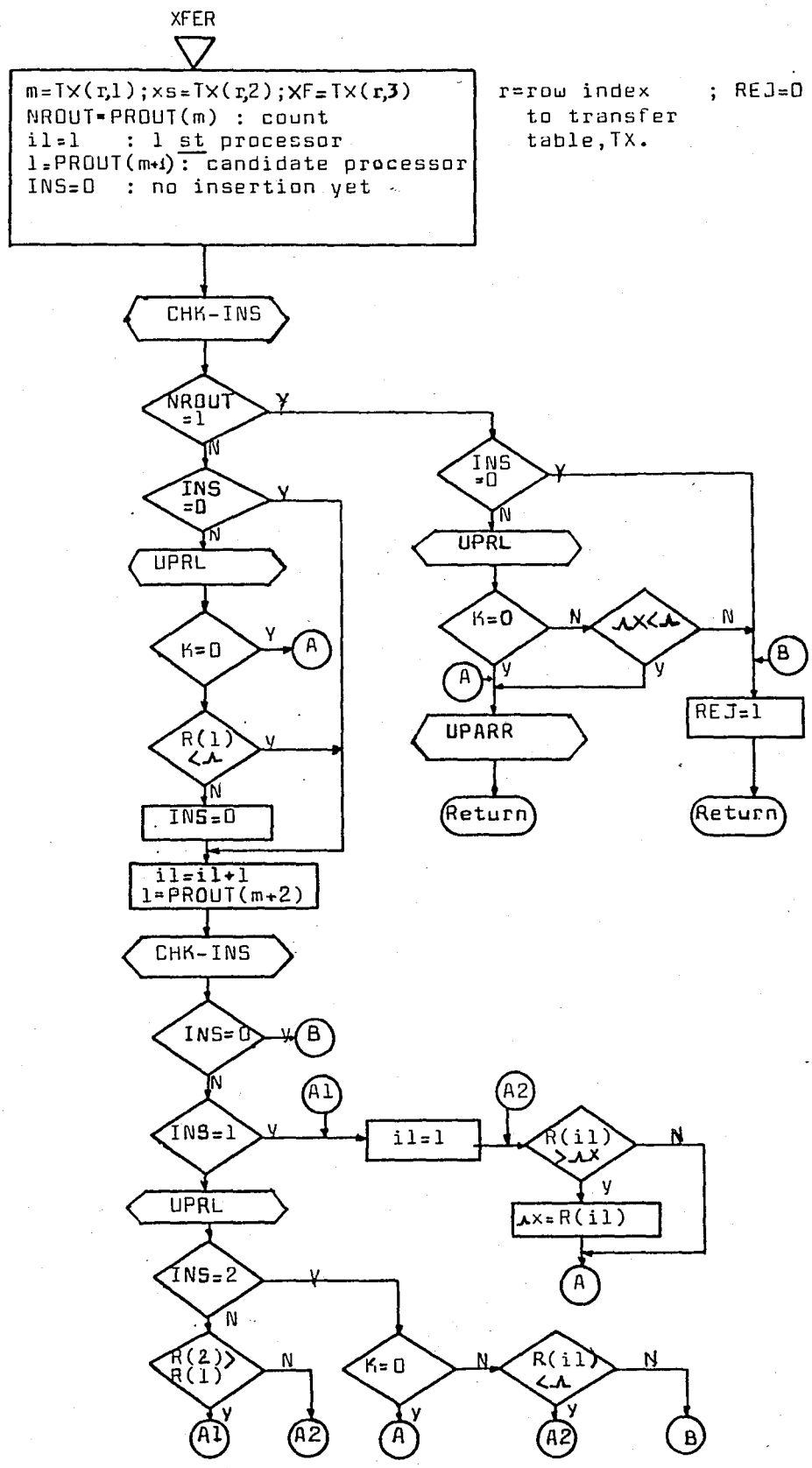


FIGURE F.1 Algorithm XFER

ALGORITHM CHK-INS(Check Insertion)

(A) ii= 1 [first module on l].
If S(l,ii) ≥ XF go to (D) [front empty]. Else,
if XS ≥ F(l,C(l)) , set ii= C(l)+ 1 and go to (D) [end empty].
Otherwise, go to (B) [intermediate].

(B) [search slot] For ii= 2,3,..,C(l), check :
If S(l,ii) ≠ F(l,ii-1) [a slot] and S(l,ii) > XS
[relevant slot] , go to (C).
Else, after ii= C(l) go to (E) [no slot].

(C) [correct time slot]
If F(l,ii-1) > XS [late start, cannot insert] go to (E).
Else, if S(l,ii) < XF [early finish, cannot insert] go to (E).
Otherwise, go to (D) for insertion.

(D) [insert] Set ip(il)=ii for the order of insertion on (il) th
processor checked (il= 1,2) , and set
INS = INS + il [update Insert flag].
Return.

(E) Return.

FIGURE F.2 Algorithm CHK-INS

```
ALGORITHM UPRL( Update R and LIPX )

(A)     If XF > F(l,C(l))  [finish time has changed]
           set R(il) = XF - S(l,1).
        Else, if XS < S(l,1)  [start time has changed]
           set R(il) = F(l,C(l)) - XS .
        Go to (B).

(B)     If NROUT = 1  and  R(il) > LIPX   [bound has changed and single
        processor]
           or
        If K = 0  and  R(il) > LIPX   [single-run mode]
        Then  LIPX = R(il)   [update LIPX].
        Return.
```

FIGURE F.3 Algorithm UPRL

```
ALGORITHM_UPARR( Update Y, S, F, C )

Procedure : UPDATE_ARRAYS ; Begin
            l:=PROUT(m+il) ;    [processor selected]
            i:=ip(il) ;         [position of transfer module]
            If i ≤ C(l) then    [insertion before the last]
              begin
                C(l):=C(l)+ 1 ;
                For  r:= C(l) downto i+1 do
                  begin
                    Y(l,r):= Y(l,r-1) ;
                    S(l,r):= S(l,r-1) ;
                    F(l,r):= F(l,r-1)
                  end
               end
            else C(l):= C(l)+ 1 ;   [insertion after the last module]
            Y(l,i):= i_max ;  [insert]
            S(l,i):= XS ;
            F(l,i):= XF ;
              i_max:= i_max+ 1
            end ;  [UPDATE_ARRAYS]
```

FIGURE F.4 Algorithm UPARR