PERFORMANCE MODELLING OF PARALLEL COMPUTATIONS

Ashok K. Adiga

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-88-11 April 1988

Abstract

The design of parallel computations involves numerous decisions which effect execution efficiency. The choice of an optimum configuration for a computation on a given architecture is essential for attaining the maximum efficiency in terms of achieved speedup. Some of the relevant factors in the configuration space of a computation include the granularity of a task, the communication model used, choice of dependencies between tasks and the host architecture on which the application is to be run. In this dissertation, we present a model for representing parallel computations which can be used to analyze their performance for various configurations. Our model is an extended Petri Net with facilities to model control and data flow mechanisms, as well as synchronization and communication primitives. A methodology is developed for representing the execution of a computation on a given architecture. The methodology consists of viewing the model as consisting of three distinct submodels (the computation, architecture and mapping submodels) which have standard interfaces between them. Specification of a structured methodology enables the automatic generation of model instances. In addition, it becomes possible to specify a library from which architectures can be selected to determine if they are suitable for a given computation. This modelling technique is then used to study the performance of computations under variations in their configuration parameters, including their actual run-time behavior on various target architectures.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. J.C.Browne, for his guidance and encouragement, without which this dissertation would not have been possible. I am grateful for the many invaluable discussions, as well as the unwavering support he has provided over the years.

I am grateful to Dr. M.K.Molloy for many Petri Net related discussions, which helped shape some of the ideas during the early stages of this work. I would also like to thank the rest of my dissertation committee, Dr. G.J.Lipovski, Dr. Vipin Kumar and Dr. J.Bitner for their encouragement and time.

I wish to thank the many friends and colleagues who helped me during my graduate career at UT Austin. In particular, I would like to thank Sanjay Deshpande for the many discussions on computer architecture, and Bapi Ahmad, Bernard Menezes and Krishna Palem for their comments and suggestions during various stages of the dissertation. Finally, I would like to express my gratitude to Nancy Macmahon for helping me survive innumerable crises.

This research was supported in part by the Department of Energy under grant DE-FG05-85-ER-25010 and by the DARPA under grant N00039-86-C-0167.

Ashok K. Adiga

The University of Texas at Austin
May, 1988

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The study of sequential algorithms has lead to well known methodologies for designing efficient computations for conventional sequential, or single processor, computers. The design of parallel computations, however, remains an active area of research. In a parallel computation, the overall computation is partitioned into smaller *tasks* which can be executed concurrently on multiple processor computers. As a consequence of this partitioning, the tasks often need to *communicate* among themselves to exchange intermediate results of the computation. Further, to ensure correct execution of the computation, *synchronization* of execution of the tasks is also required. A major step in the design of parallel computations is therefore the partitioning of the computation into smaller tasks. An inefficient partitioning could lead to prohibitive synchronization and communication overheads which would, in turn, lead to an inefficient realization of the computation. The execution efficiency of a parallel computation is dependent on the architecture on which it is executed, since the communication and synchronization overheads depend on the architectural support provided. To design an efficient partition for a given computation, it therefore becomes important to be able to *model* the execution of the computation on a given architecture. This would lead to a better understanding of the methodology used for parallel computations.

The specification of a parallel computation includes the specification of a collection of tasks and the dependencies between them [BRO85]. The dependencies could be simple synchronization or sequencing relationships, constraint relationships such as mutual exclusion for shared resources, or communication relationships signifying data transfer between two or more tasks. The basis for a model for parallel computations is to be able to predict the efficiency of execution of a parallel computation on a given architecture, and to study the effect of varying different configuration parameters of the computation. The *configuration space* of interest includes the following parameters:

- shared memory/ message model: the interaction between the tasks of a computation can be specified using a message model, a shared memory model, or a mixture of the two. This choice is based on the support provided by the host architecture and the volume and size of the information being exchanged.

- granularity of a unit: the efficiency of the computation is directly dependent on the size of each schedulable unit of computation (or task). Larger units would usually reduce the amount of parallelism possible, while smaller units could entail additional overheads of data movement.

- computation structure: this includes the specification of synchronization and sequencing of the units composing the computation.

- underlying or host architecture: on which the computation is mapped. If the computation maps naturally on the architecture, the performance observed will be much better.

It is clear that there are several decisions to be made when designing an efficient computation. The problem is further aggravated by the interdependence of the factors specified above. The granularity of a unit of computation may lead to an efficient implementation on one architecture, but not on another. Processor and memory availability on a particular architecture may decide the best computation structure for a computation. A modelling technique which captures the behavior of the computation is therefore invaluable as an aid to determining an optimal configuration for a given computation and architecture.

A further degree of complexity is engendered by the fact that computations are usually expressed in terms of some programming language which, in turn, imposes further constraints on the execution. As shown in Figure 1-1, analysis of the execution behavior of parallel algorithms is most often done by direct mapping of the algorithm, which is expressed in terms of the operations of some abstract machine to the operations of some real machine architecture. There is then often surprise when a program

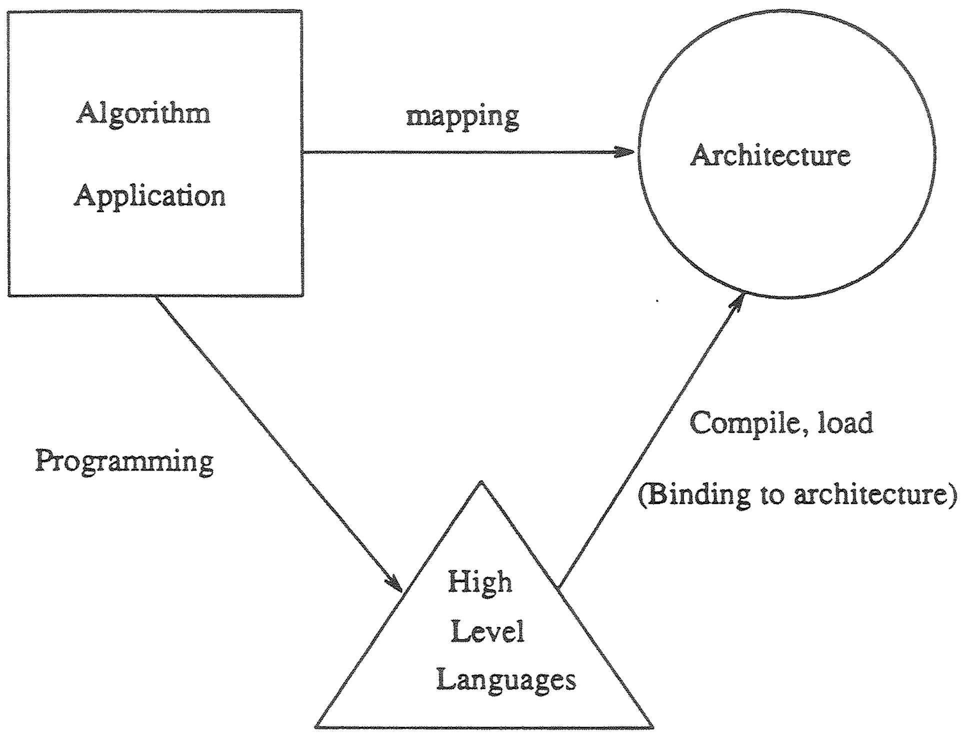


Figure 1-1: Models of Computation and their Mapping

written for the algorithm fails to perform in the manner suggested by the direct mapping of the algorithm to the architecture. A major source of such surprises is that the higher level language in which the program for the algorithm is usually expressed defines yet a third abstract machine based on yet a third model of parallel computation. There are thus two perhaps quite different mappings involved. Each mapping may introduce additional execution cost due to differences in abstract machine architecture. Mapping of control and communication structures, the major difference between models of computation, are often the major sources of additional execution cost. There is thus need for a capability to conveniently and cost effectively analyze the execution behavior on real architectures of parallel computations expressed in terms of programs in higher level languages.

To represent parallel computations as described, the model should:

- be able to represent, to some extent, the data state of the computation. Unless the data state is explicitly modelled, (deterministic) data dependent flow of control can only be modelled probabilistically, which can lead to inaccuracies in the predicted performance.

- be hierarchical in nature. Hierarchical representations allow modelling at various levels of granularity. In addition, changes to one level of the model should not effect the rest of the model

- have a mechanism to specify the metrics of interest in the performance study being conducted, with the aim of identifying bottlenecks and restructuring the computation to obtain a more efficient realization.

- support a systematic methodology for the development of executable representations.

There have been three basic approaches to modelling parallel systems: Discrete event simulation models ([NIE69],[EFR64]), Analytic Queueing models and Graph models (in particular, Petri Net models). Petri Nets [PET81] are useful for modelling parallel systems because of their ability to represent asynchronous behavior. In addition, they provide a natural way of representing holding of multiple resources and process blocking, both of which occur frequently in parallel systems. The major drawback with standard Petri Nets is that the concept of *time* is missing. It is only possible to arrive at a partial ordering of events that occur in the system (which are modelled by the firing of transitions). Several extensions to Petri Nets have been suggested which include features to gather timing information such as total execution times, resource holding times, communication and synchronization delays ([NUT72], [NOE78], [BER79], [MOL82], [SIF79], [MAR84]). Most of these models are strictly probabilistic in nature, and do not attempt to model the data dependent decisions which are inherent to modelling computations. There have been few models which incorporate the modelling of data ([EST86], [STO85]). All these models are described in greater detail in Chapter 2.

Chapter 3 contains the definition of the model proposed in this dissertation.

The model is based on Petri Nets, with extensions to facilitate the performance modelling of parallel computations. Attributes are added to the Petri Net transitions to introduce time into the model and to model the data state of the computation. The notion of hierarchy is introduced into the model by defining *subnets*. When combined with the parameterization of nodes, this allows the modelling of complex computations (including their representation in programming languages) and architectures in a simple and natural manner.

A methodology for representing computations is presented in Chapter 4. This helps in ensuring correct models by allowing the automatic generation of model instances for a given computation. It also permits the creation of an architecture library, containing verified models of representative architectures, from which candidate architectures can be selected for a modelling experiment.

A tool supporting the performance studies, based on the proposed model, is presented in Chapter 5. The front end graphical editor allows the creation of model instances, while simulation of the model instances yield performance statistics. The Chapter describes the algorithms used to trace the occurrences of events in the system. Validation results obtained for various examples are also presented.

The suitability of the model for representing parallel architectures is demonstrated in Chapter 6. The models are verified using results available from previous research, and the verified models are used to create an architecture library. In particular, the architectures studied can be classified as shared memory architectures, where processors in the system communicate via a shared address space. These architectures are then used in a modelling study of parallel computations, as described in Chapter 7.

Chapter 2

Graph Models for Parallel Systems

Several techniques have been developed over the years for modelling computer systems. This chapter presents some of the common classes of models proposed in the past, and attempts to trace their evolution. A *model* is an abstract representation of a system which omits details not essential to the purpose of the model. By manipulation of the representation, new knowledge about the modelled phenomenon can be obtained without the danger, cost or inconvenience of manipulating the actual system itself. This knowledge could be useful when designing a new system or when making changes to an existing one.

A model for a computer system has several applications. It serves as a formal specification of the modelled system. In addition, it can be used to determine two important aspects of the system: correctness and performance. Correctness implies proper system behavior under all possible external conditions, while performance specifies its efficiency under these conditions. The most commonly used modelling techniques are either mathematical or procedural. The former consists of sets of relations describing the system mathematically, while the latter is a description of the various parts of the system, usually using some programming language. Solution techniques used for computer system models are either simulation based or analytical. Mathematical models are often amenable to analytical solutions, which are preferred over simulation solutions due to their accuracy and speed. In some cases, however, such analysis proves to be mathematically intractable, in which case simulation becomes the only method of solution. Procedural models are usually studied using simulation.

Regardless of the technique used to study the model, *validation* of the model is an essential step which cannot be ignored. Validation ensures that the model does, indeed, mimic the behavior of the system being modelled. Informally, validation should ensure that given the same set of inputs, the model and the system display identical behavior and produce the same outputs.

The most common forms of procedural models fall under the category of *Discrete Event Simulations*. Discrete Event Simulation models involve the description of a system by a computer program and the simulation of the interactions within that system as they occur over a discrete time interval. The system being modelled is described using a *simulation language*. One of the earliest and best-known simulation languages is IBM's GPSS [EFR64]. While not used extensively for performance evaluation, this language is still popular in other areas of computer-based simulation. The major drawback of discrete simulation language models is the high cost incurred in development and use. Further, changes in the system often lead to major rewrites of the simulation program, which could prove to be too expensive. In most cases, the simulation model, itself, represents a significant piece of software which might be difficult to validate or verify.

Analytic Queueing models have been widely used to model computer systems [KLE75]. They are typically simple and easy to use, and provide a means to obtain results analytically, without having to resort to simulation. A queueing network consists of several service facilities (resources) at which customers (jobs) queue for service. Specification of a service facility includes specification of the rate at which customers arrive, as well as the structure and discipline of the facility itself. Further, a service facility can give preferential treatment to certain (classes of) customers. When a customer completes service at a facility, or node, it moves to a new node. These routing decisions are specified by the arcs interconnecting the nodes. If all service facilities satisfy certain constraints, it has been shown that analytical results can be derived for the entire network [BAS75].

There are some serious problems with using such models for parallel systems. Queueing models do not have any capabilities for proving correctness of the system being modelled. Due to their probabilistic nature, they are unable to model deterministic processes. It is difficult to model process blocking and the holding of multiple resources, both of which are common occurrences in parallel systems. To overcome these problems, extended queueing models such as PAWS [IRA86] and

RESQ [SAU82] have been proposed. The major addition in RESQ, for example, is the concept of *passive* resources which jobs can hold while receiving service at a node. Jobs can also block while waiting for a passive resource to become available. If passive resources are used, however, the model can no longer be solved analytically.

Several *Graph* models have been proposed for modelling parallel systems. A graph usually consists of nodes with interconnecting arcs. In most representations of parallel systems, the nodes have been interpreted as representing some type of action or transformation in the state of the system, while arcs represent flow of data and control. The basic difference lies in this interpretation placed upon the nodes and arcs. In fact, the queueing network models discussed previously can be viewed as graph models with a rather elaborate interpretation of the nodes as queue/server pairs. The interpretation includes, for each queue, a service discipline, and for each server, a service rate. Petri Nets, on the other hand, interpret nodes as being places (representing conditions) and transitions (representing actions). Most of the earlier graph models were used to determine the correctness of the system being modelled, and paid little attention to the system performance. Later models have incorporated features that allow these performance figures to be obtained. Graph models can be broadly classified into four groups.

- General Transition nets such as Standard Petri Nets [PET81], Computation Graphs [KAR66], Vector Addition Systems [KAR69], Self Modifying nets [VAL78] and Predicate/Transition nets [GEN81]. These models provide the basis for most of the later models, and are therefore extremely significant.

- Restricted Models such as E-Nets ([NUT72], [NOE73]) and Decision free Petri Nets [RAM80]. These models place restrictions on standard Petri Nets. Though this reduces the modelling power, the resulting model is easier to model and analyze. In addition, these models included the notion of time, thus allowing performance evaluation of the modelled system.

- Augmented Models such as the UCLA Graph Model of Behavior [VER83], SYSTEM [YAU83], Parallel Program Schemata [KAR69] and the Hierarchical Graph model [STO85]. These models consist of distinct submodels to model control and data domains. In addition, an interpretation domain is declared which contains semantic information.

- General Time-Extended models such as Stochastic Petri Nets [MOL81] and Time Extended Petri Nets [BER79]. These models allow standard Petri Nets to be defined with added facilities for performance evaluation.

The rest of this chapter is devoted to a study of these models. A somewhat detailed description of general transition nets is included, since most of the extended models proposed (including the model proposed in this dissertation), are based on them. Some pointers are also given to the commonly used analysis techniques for such nets.

2.1 General Transition Nets

2.1.1 Petri Nets

A Petri Net model is constructed out of two sets of vertices in a bipartite graph. One set of vertices is called the set of *places* and represents the set of conditions. The other set of vertices, called *transitions*, represents changes in state or operations. The arcs which connect the transitions and places represent the dependencies between the events and conditions.

More formally, a Petri Net is defined as a bipartite, directed graph described by the four tuple, PN = (P,T,I,O), where

P = (p_1,.....,p_n), a set of places, n ≥ 0;
T = (t_1,.....,t_m), a set of transitions, m ≥ 0;
I is the transition input function, I : T --> 2^P,
or, I is a subset of PxT;
O is the transition output function, O : T --> 2^P,
or, O is a subset of TxP;
and the sets P and T are disjoint.

The places specified by I, the Input function for a transition are called the *input places* for that transition. Similarly, the places specified by the Output function for a transition are its *output places*. The model defined above captures the static properties of the modelled system. The dynamic properties can be modelled by the addition of another primitive entity called a *token*. Tokens reside in places and signify the existence of a condition. A *marking* of a Petri Net is an assignment of tokens to places.

This distribution of tokens in the net may change with the *execution* of the net. The *Initial Marking*, the distribution of tokens before execution begins, is often specified as part of the net definition. Execution of the net proceeds by *firing* transitions which alter the net marking, using the following *firing rules*:

- A transition may fire if it is *enabled.* A transition is said to be enabled if each of its input places contains at least one token.
- The firing of a transition removes one token from each input place, and places one token in each output place.
- At any given instance, if more than one transition is enabled, exactly one of them is selected non-deterministically. The selected transition is then fired, leading to a new marking.

$$\begin{array}{lll}
PN & = & (P, T, I, O, M) \\
 & & \\
P & = & \{ P_1, P_2, P_3, P_4, P_5 \} \\
T & = & \{ T_1, T_2, T_3, T_4, T_5 \} \\
 & & \\
I & : & I(T_1) = \{ P_1 \} \\
 & & I(T_2) = \{ P_4 \} \\
 & & I(T_3) = \{ P_2 \} \\
 & & I(T_4) = \{ P_3 \} \\
 & & I(T_5) = \{ P_4, P_5 \} \\
 & & \\
O & : & O(T_1) = \{ P_2, P_3 \} \\
 & & O(T_2) = \{ P_2 \} \\
 & & O(T_3) = \{ P_4 \} \\
 & & O(T_4) = \{ P_5 \} \\
 & & O(T_5) = \{ P_1 \} \\
 & & \\
M & : & [1, 0, 0, 0, 0]
\end{array}$$

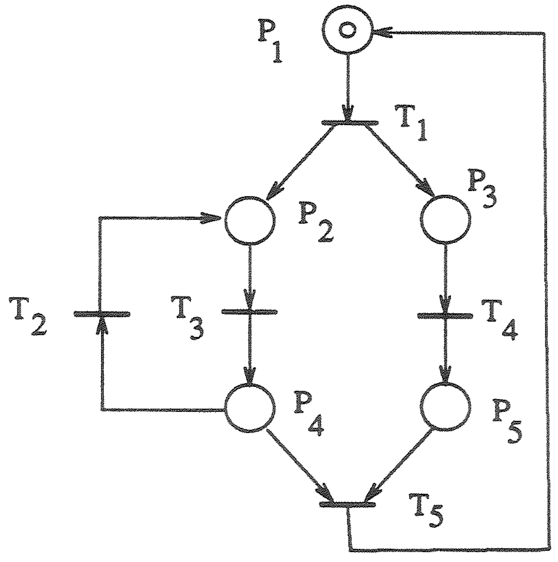


Figure 2-1: Graphical representation of a Petri Net

Figure 2-1 shows the definition of an instance of a Petri Net. As mentioned earlier, I and O are sets of functions specifying the input and output places for each transition. The marking vector, M, specifies the number of tokens in each place. In the figure, the only token in the net is in place P_1. This distribution of tokens to places (i.e. the marking) of the net at any instant defines the state of the net. The equivalent graphical representation of this Petri Net is also shown in the figure. Places are represented as circles, transitions as bars, and tokens as dots within places. For each transition, arcs are drawn from its input places to the transition, and from the transition to its output places. The only transition enabled for the marking shown in the figure is T_1. Firing transition T_1 causes a token to be removed from P_1 and leads to a new marking with tokens in places P_2 and P_3. Both T_3 and T_4 are now enabled and can fire independently, leading to a marking $M' = [\ 0, 0, 0, 1, 1]$. In this marking, both T_2 and T_5 are enabled, but the firing of either would disable the other. The two transitions are said to be in *conflict*, and one of them is selected non-deterministically and fired.

An execution of this Petri Net can be characterized by a sequence of transition firings which cause a succession of net markings from the initial marking M to a final marking M'. If the net does not contain two or more transitions which have identical input and output functions, this sequence of net markings also uniquely specifies the execution of the net. Correctness properties of the net are usually determined by enumerating all possible execution sequences by constructing a *reachability set*. The reachability set contains all markings reachable from the initial marking through a legal sequence of transition firings. This set is commonly represented as a graph with markings as nodes, and directed arcs connecting two markings if the firing of a transition takes the net from one marking to the other. The reachability graph for the net defined in Figure 2-1 is shown in Figure 2-2, where the arcs are labelled with the transition whose firing caused the change in marking. Since, for unbounded nets, this technique can lead to infinitely large reachability graphs, another technique called a reachability *tree* is often used. The method for constructing the tree is similar to that shown here, except that if firing a transition leads to a marking M_b which is greater than some marking M_a already encountered in the tree, all those positions (places) in M_b which are greater (have more tokens) than the corresponding position in M_a, are replaced with the symbol o. This ensures that the resulting tree is bounded in size, at the cost of losing some information about the exact markings reachable. The reachability graph can be used to answer some questions about the behavior of the net . Most of this analysis, however, is aimed at proving the *correctness* of the modelled system, and not its performance.

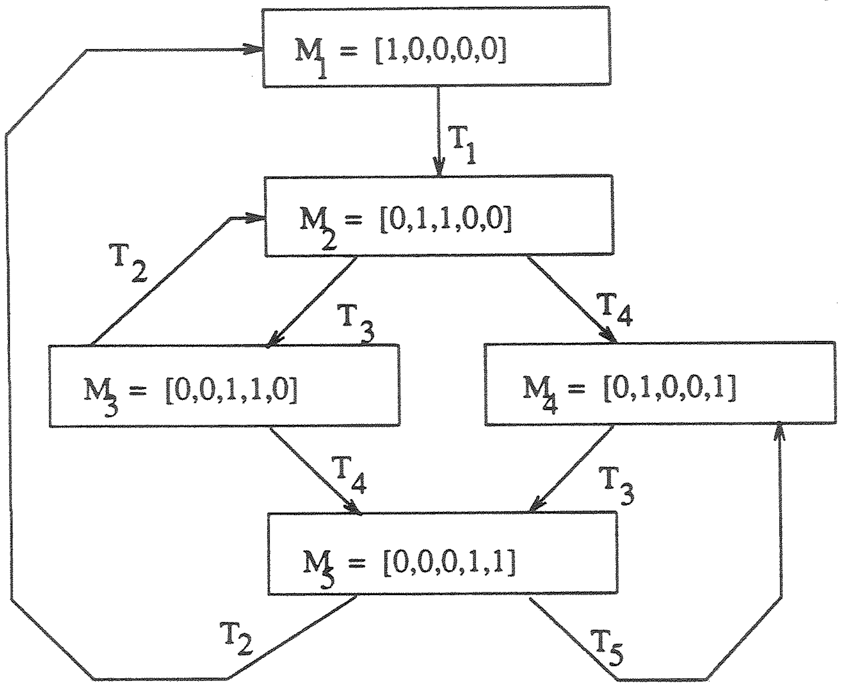


Figure 2-2: Reachability Graph for Figure 2-1

Another means of analyzing properties of such nets is the method of *invariants* which uses the structure of the net along with the firing rules, and provides a method for verifying certain *facts* (invariants) about the modelled system. An extensive study of the theory of invariants in such nets can be found in [REI82]. The basic idea is to identify sets of places such that the weighted sum of the number of tokens in these places is always constant. Using this invariant and the initial marking of the net, it is possible to generate relationships for the number of tokens in the places in the set. This is useful for proving properties such as mutual exclusion between two places.

Several extensions have been proposed to the Standard Petri Nets (PNs) defined in the previous Section. These extensions either extend the modelling power, or simply make the modelling effort easier by reducing the number of nodes in the graph (implicit introduction of hierarchy into the representation). It is known that the modelling power of PNs is less than that of a Turing machine [PET81]. However, some of the extensions introduced here lead to models equivalent to Turing machines. The extensions which reduce the number of nodes in the graph, however, can be shown to have equivalent representations using the basic PN model.

The most commonly used definition of Petri Nets allows weighted arcs between nodes. A weighted arc is equivalent to having multiple arcs between a place-transition pair. The firing rule now specifies that a transition is enabled only if each of its input places contain at least as many tokens as the weight of the arc that connects them. Further, firing a transition causes the appropriate number of tokens to be removed from the input places and placed in the output places. Petri Nets with weighted arcs are equivalent to standard Petri Nets in their modelling power.

Colored Petri Nets [PET80] are Petri Nets in which tokens are typed (or have colors). The state of a place node is now represented by a vector specifying the number of tokens of each type present in that place. The enabling rule for a transition is modified to require a certain combination of typed tokens at the input places for enabling the transition. When a transition fires, it removes its enabling tokens from the input places, and places a combination of typed tokens in its output places. This is implicit introduction of hierarchy into the representation basis of simple Petri Nets since a complex net structure in a simple Petri Net can be reduced to a single transition and its input/output place set in a Colored Petri Net. If the number of colors (types) allowed is finite, the resulting Colored Petri Net is equivalent to PN. If an infinite number of colors is allowed, however, the model becomes equivalent to a Turing Machine.

Another modification to PNs is to define an arc type called an *inhibitor* arc [KOS73]. This arc can only connect a place to a transition, and implies that the transition can be enabled only in the *absence* of a token at that place. Again, PNs with inhibitor arcs are equivalent in modelling power to Turing Machines [AGE74].

2.1.2 Self-Modifying Nets

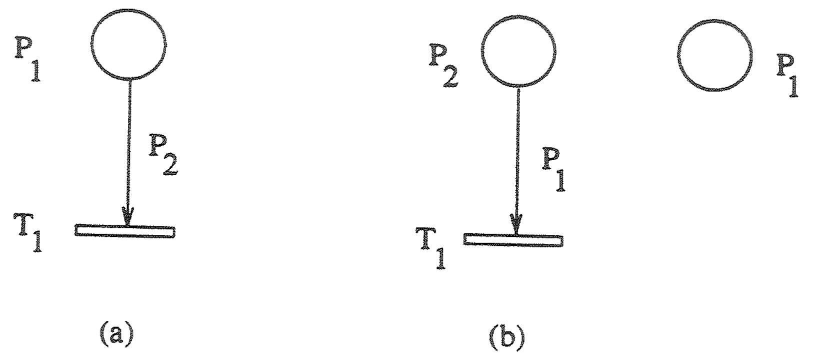


Figure 2-3: Examples of SM-nets

Self-Modifying Nets (SM nets, [VAL78]) is an extension to Petri Nets, in which firing rules of transitions can be changed over the execution of the net. The extension over PNs is that arcs can be labelled either with integer weights (as in the case of Petri Nets), or with an arbitrary place name (say, P_2), as shown in Figure 2-3(a). For transition T_1 to fire in a marking M, P_1 should contain as many tokens as are present in place P_2 in that marking. Since the number of tokens in P_2 can vary over the execution of the net, the firing rule for transition T_1 can be dynamically modified. Figure 2-3(b) shows how SM nets can easily model inhibitor arcs. In the figure, P_2 is a special place with no input arcs, and is therefore always empty. T_1 thus fires only if P_1 contains exactly zero tokens, which is equivalent to having an inhibitor arc from P_1 to T_1.

2.1.3 Computation Graphs

Computation Graphs were one of the earliest models of parallel computation proposed [KAR66]. They were mainly designed to represent execution of programs evaluating arithmetic expressions in parallel. A Computation Graph is a finite, directed graph where each vertex represents an operation, and each (directed) arc represents a queue of data items used by the target operation and produced by the source operation. Each arc has, associated with it, some control information. This information is expressed as a four tuple (I, V, W, T). I is the initial number of data items in the queue; V is the number of data items produced by the source for each execution; T, the threshold, is the number of items required by the target operation before it is enabled, and W is the number of items actually consumed by the target for each execution. Obviously, W ≤ T should hold for each arc.

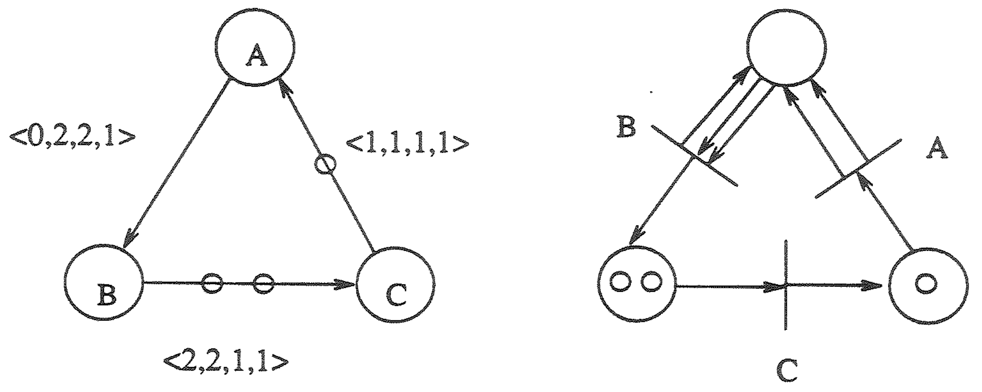


Figure 2-4: Computation Graph and its Equivalent Petri Net

Each instance of a Computation Graph has an equivalent Petri Net representation with each arc being replaced by a place and each vertex by a transition. The initial Petri Net marking is defined by the initial number of data items, I, in each arc. Figure 2-4 shows the transformation of a Computation Graph into an equivalent Petri Net. The modelling power of Computation Graphs is known to be less than that of Petri Nets [PET81].

2.1.4 Predicate/Transition Nets

Predicate/Transition Nets [GEN81] are a class of colored Petri Nets with predicates associated with transitions, and expressions (possibly containing variables) associated with arcs. In any given marking of the net, a transition is enabled if there is an assignment of tokens (colors) such that the arc expressions are satisfied (by binding any existing variables), and the transition predicates hold under this assignment. This generalization of ordinary Petri Nets allows more manageable descriptions of nets, due to the fact that equal subnets can be folded into each other yielding a much smaller net. The two common analysis techniques used for Petri Nets, the reachability tree and invariant approaches, can both be generalized to apply to these Predicate/Transition nets [GEN81].

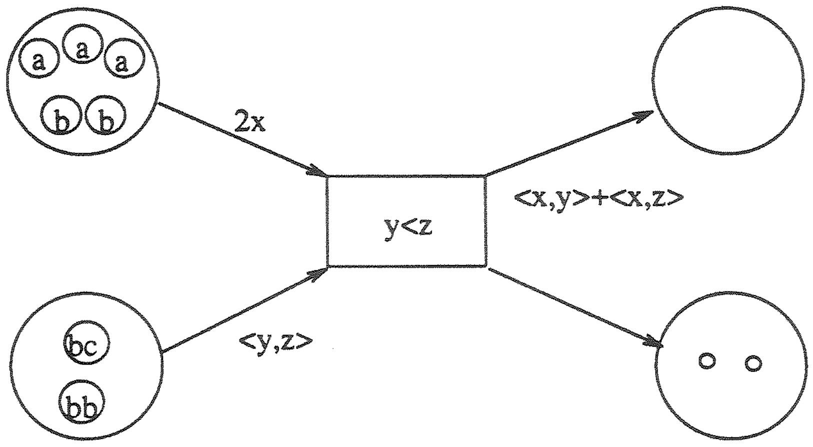


Figure 2-5: Transition in a Predicate/Transition Net

Figure 2-5 shows a predicate/transition net fragment. The places contain tokens which have structure and information (i.e. are colored). The arcs have expressions associated with them which contain variables (x, y and z), which are bound using the values of the tokens. The transition has a predicate associated with it, which must hold for the transition to be enabled. In the figure, the transition is enabled for two possible assignments of values to the variables (<x,y,z> = <a,b,c>, and <x,y,z> = <b,b,c>). The two firings conflict with each other (for the token 'bc' in the lower lefthand place), and only one can be fired for the marking shown. The arc label '<x,y>,<x,z>' indicates that upon firing, two tokens will be placed in the output place, and the information contained in those tokens will be <x,y> and <x,z> respectively.

2.1.5 Vector Addition and Replacement Systems

Vector Addition Systems (VAS) are mathematical models for analyzing systems of parallel processes [KAR69]. Because of their simple formulation, these systems are typically used for formal proofs of properties of equivalent models such as Petri Nets.

An r-dimensional VAS is defined by the two-tuple <d,W>, where d, the *start vector*, is an r-dimensional vector of non-negative integers and W is a finite set of r-dimensional integer vectors called *displacement* vectors. The Reachability set is the set of all vectors of the form:

$$d + w_1 + w_2 + \dots + w_s$$

such that

$$w_i \in W \qquad i = 1,2,\dots s$$

and

$$d + w_1 + w_2 + \dots + w_i \geq 0 \quad i = 1,2,\dots,s$$

In other words, a displacement vector may be applied at any point if the resulting r-vector has no negative integers. It has been shown that these systems are equivalent to self-loop free Petri Nets. The dimensionality, r, corresponds to the number of places in the corresponding Petri Net. The start vector corresponds to the initial marking of the Petri Net, and each displacement vector corresponds to the change in marking that occurs if a transition fires. Each displacement vector, therefore, corresponds to a transition in the Petri Net. Since the 'displacement' caused by a self-loop in a Petri Net is zero, this cannot be represented in a VAS.

Vector Replacement Systems (VRS) were defined to directly model Petri Nets with self loops in a vector addition system-like model [KEL74].

A VRS is defined to be a two tuple <d,W>, where d is defined as before, and W is a finite set of **pairs** of r-vectors:

$$W = \{ U_i, V_i \} \qquad i = 1,2,\dots m$$

where the U vectors are **test** vectors, and the V vectors are the displacement vectors.

Before a displacement vector can be applied, the corresponding test vector is used to check if it is legal. The reachability set is now redefined to be the set of all vectors of the form:

$$d + v_1 + v_2 + \ldots. + v_s$$

such that

$$v_i \; \varepsilon \; V \qquad i = 1,2,\ldots s$$

and

$$d + v_1 + v_2 + \ldots. + u_i \geq 0 \quad i = 1,2,\ldots,s$$

2.2 Restricted Models

In this section, the models described are restricted forms of the standard nets described in the previous section. These restrictions, while decreasing the modelling power, enhance the analytical power of the models.

2.2.1 Evaluation Nets

The Evaluation nets (E-nets) developed by Nutt [NUT72] are similar to Petri Nets in appearance and operation. There are, however, some restrictions and modifications to standard Petri Nets which are listed below:

1. All transitions are constrained to be of five predefined types.
2. Places can have at most one input and one output arc. This means that there cannot be any conflicting transitions. A place can hold zero or one token only: the net is safe.
3. A special place type (called a resolution place) is introduced which acts as a switch. Its operation is controlled by resolution procedures which can read token attributes.
4. Tokens are have attributes which distinguish them from each other. These attributes are modified by transition procedures which are invoked by the firing of a transition.
5. The firing of a transition introduces a fixed delay. This feature introduces time into the net.

6. Some global 'environment variables' are defined which maintain global state information. These variables can be modified by transition procedures.

The first two conditions simply reduce the state space and make the net easier to analyze. Conditions 3 & 4 imply that data dependent control flow is now possible, since the resolution 'switch' operates on token attributes. Conditions 4 & 5 allow timing information to be collected in tokens. This information can be eventually aggregated using an environment variable. The existence of environment or global variables is the major drawback in this model. This makes analysis of the behavior of the model very difficult. The only feasible way to study the behavior of these model seems to be by actual execution of the model.

An extension to E-nets was proposed which allowed macros to be defined [NOE73], and then using these macro nets to model more complex systems. While this does not add to the modelling power of E-nets, it makes the representation easier to understand.

These nets are capable of modelling complex systems. Noe and Nutt used macro E-nets to model the behavior of the CDC6400 system [NOE73].

2.2.2 Pro-Nets

This model is based on a modified form of E-nets [NOE78]. One basic type of transition is defined, which has a set of conjunctive (AND) inputs and a set of selected (OR) inputs. To be enabled, all the AND inputs should have exactly one token, and at least one of the OR inputs should have a token. Upon firing, each AND output gets a token, and one OR output gets a token. The interesting contribution of this work was to show how Pro-nets could be represented by Vector Addition Systems, which could then be used to mechanically abstract the original net to simpler nets by collapsing selected places and transitions.

2.2.3 Decision-free Petri Nets

Decision-free Petri Nets [RAM80] are Petri Nets in which each place has exactly one input transition and one output transition. Though these nets have several interesting properties, their major drawback is that they cannot be used to model contention, since transitions cannot share places. The transitions in these nets can be specified with fixed delays.

2.3 Augmented Models

The augmented models are implemented with three distinct submodels. The first is the control which is equivalent to one of the previous models. There are interpretation procedures which are associated with each operation in the control. Finally, there are data storage cells where interpretation procedures can read or write information.

Though these models offer an ease of expression for some complex systems, due to the encoding of state information into data representation, analytical representation of the states becomes cumbersome. In most cases, simulation of the system must be used to measure its performance.

2.3.1 UCLA Graph Model of Behavior

The control domain consists of nodes (corresponding to events), arcs which model precedence relations between events, and tokens which reside in arcs (which model presence of conditions required for an event to occur). The input or output arcs at a node can be related using logic operators (AND, OR, prioritized OR). The data domain consists of processors which are mapped to control domain nodes on a one-to-many basis. Each processor has predefined datasets from which it may read or write. A processor is activated when any one of its control nodes is activated. The data transforms performed and the delays associated with the processor are defined by the interpretation domain. This extended model allows various queueing phenomena to be modelled, such as different service types at control nodes. A detailed description of this model is given in [VER83] and [EST86].

2.3.2 SYSTEM

SYSTEM is a modified form of Petri Nets designed for modelling distributed software systems [YAU83]. The control domain is a Petri Net graph with certain modifications. Input and output arcs can be connected by logic operators, and logic expressions for output arcs can be data dependent. Each transition of the control domain is viewed as a software 'component', with associated data objects. This defines the data domain. Further, each transition has a set of abstract data types defined. A data transfer specification function is defined for each transition, which specifies what operations are to be applied to the data objects. This specification, along with the abstract data types, forms the interpretation domain.

2.3.3 Parallel Program Schema

The Parallel Program Schema [KAR69] consists of a set of M memory locations, a finite set A, of operations, and a control T for the sequencing of operations. The control T is basically a transition system consisting of a set of states, a finite alphabet with a set of initiation and termination symbols for each operation, and a next-state function. This forms the control domain. The data domain consists of the memory locations defined by M. Further, each operation has a domain of memory locations, D(a) from which it can read values, and a range of locations, R(a) to which it can write its results. The actual performance of an operation is called the interpretation of the operation.

2.3.4 Hierarchical Graph Model

The Hierarchical Graph model [STO85] was proposed as a tool for modelling software systems. In this model, a process is viewed as a three tuple: <D, SP, CF>, where D is the *data model*, SP is the *static program* model, and CF the *control flow* model. The data model is defined by an H-graph grammar which represents the possible local data states for the process. The static program model is a collection of 'basic blocks', where each block is a linear sequence of procedure calls. The inputs and outputs of the procedure calls are specified using selector functions to extract values from the corresponding data model. The control model is a timed Petri Net which specifies the sequence of execution of the basic blocks.

2.4 General Time-Extended Models

The models described here do not restrict Petri Nets, but extend the model to include timing information. Since these models use standard Petri Nets, the usual analysis can be done on them. The models we consider fall into two broad categories: Stochastic Petri Nets (SPNs) and General Timed Petri Nets (GTPNs). Stochastic Petri Nets are historically important since they provided the first bridge between queuing networks and Petri Nets.

2.4.1 Stochastic Petri Nets

Stochastic Petri Nets [MOL81] are Petri Nets which are extended by assigning random variables representing firing delays to each transition. The set of SPNs with exponentially (or geometrically) distributed delays is shown to be homomorphic to the set of all homogeneous Markov chains. The state of a Petri Net is represented by its marking. Using the equivalent Markov model, the steady state probability of each of these states can be calculated. This information can be used in evaluating performance measures such as average delay and average throughput. Two advantages of this model are the simplicity of specification and the verification step provided by the automatic generation of the state space. Some extensions to SPNs include allowing transitions with no firing delay [MAR84] and arbitrary delays [DUG85]. Transitions with zero delay have precedence over the other transitions. This reduces the set of reachable states in the extended model compared to the basic SPN. When arbitrary delays are introduced, however, the net cannot be solved using the usual Markovian analysis and must be simulated. One drawback with SPNs is the explosion in the size of the Markov chain as the size of the Petri Net is increased (i.e. as places or tokens are added to the net).

2.4.2 Time Extended Petri Nets

Time Extended Petri Nets (TEPNs) were proposed by Berlin [BER79]. The basic extension in TEPNs is to differentiate between 'active' and 'enabled' tokens. Each token has a delay associated with it. A transition is enabled when each of its input places contains an active token. These active tokens are now converted to 'enabled' tokens and can no longer cause any other transition to be enabled. Tokens are delayed for a time equal to the delay time associated with them. The Parallel-Program Reconfigurable-Architecture Performance (PRP) model [KAP82] associates delays with transitions rather than with tokens, thereby leading to a more natural representation. Each TEPN place has an asynchronous clock and a set of performance functions which allow the evaluation of performance measures for that place. Finally, tokens have attributes which can be modified by transitions. As is the case with SPNs, the degenerate case of TEPNs is the standard Petri Net upon which standard Petri Net analysis can be done. Another model, the Generalized Timed Petri Net model (GTPNs) proposed by Holliday and Vernon [HOL85] is the timed Petri Net equivalent of GSPNs. This model places no restrictions on the structure of the net, other than requiring it to be bounded. Delays are associated with transitions and can be deterministic or geometrically distributed. The model allows analysis by creating the embedded discrete-time Markov chain and solving for steady state values. As with SPNs, the GTPN has the drawback that the embedded Markov chain explodes in size as the size of the Petri Net is increased.

2.5 Conclusions

Graph models have been widely used for performance modelling of computer systems. The descriptions given in this Chapter were necessarily brief; a more comprehensive description and comparison of various graph models for performance can be found in [BRO87].

The previous sections have described several Petri Net based models for modelling parallel systems or computations. The difference between modelling parallel systems and parallel computations is that parallel computations are, in general, expected to terminate, whereas parallel systems are modelled assuming that they never terminate. Also, parallel computations have a strong notion of flow of control and data that is not explicit in parallel systems. In general, however, most models are capable of representing both parallel systems as well as parallel computations.

A performance model for parallel computations should have the following capabilities: the ability to represent the flow of control for individual processes in the computation; the ability to represent the data state for the computation; representations for interprocess communication and synchronization; inclusion of timing information in the model so that the performance of the computation can be determined by analysis or simulation of the model.

The general transition nets provide the basis for most of the other models. They are useful for formal specifications of parallel computations, but are inadequate for our purposes. A major drawback with these nets is that they do not include timing information and they cannot be used to make performance estimates. The restricted models (such as E-nets) can be analyzed to yield performance estimates, but lack the modelling power of the more general models. The augmented models have the power to model general parallel computations. The separation of the data and control domains, however, tends to obscure the interactions between processes in the computation. Stochastic Petri nets have proved to be extremely useful for modelling parallel systems. However, they are inadequate for representing parallel computations since there is no concept of a data state. It is not possible to make decisions depending on the current data state of the computation. A desirable model, therefore, would include the power of the general time extended nets and the augmented models' ability to represent data dependent control flow.

Chapter 3

A Model for Parallel Computations

Most of the models discussed in the previous Chapter dealt with the representation of parallel systems. The model presented here deals specifically with the representation of the execution of *parallel computations*. Although the two have many common issues, there are some problems unique to parallel computations. In particular, such models must allow some mechanism for representing data-dependent control flow. The flow of control and data in the model should faithfully reflect that of the actual program. A more serious problem is the existence of global variables in programs. There seems to be no elegant way to represent such variables since global variables, by definition, can be altered almost anywhere in the program. The choice of a Petri Net based model, however, allows most of the synchronization and communication primitives to be directly modelled. The Parallel Computation Model (PCM, [ADI86]) is based on Petri Nets [PET81], and is somewhat similar to the Parallel-Program Reconfigurable-Architecture Performance (PRP) model proposed in [KAP82], the main difference being the explicit control flow permitted in our model. The transition predicates and procedures proposed here are derived from the program verification model proposed in [KEL76]. The rest of this Chapter is organized as follows. Section 3.1 contains a brief discussion of the capabilities required for modelling parallel computations. A formal definition of the model follows in Section 3.2. Section 3.3 discusses some properties of the model such as representation of state of the net and its modelling power. Extensions to the model to include hierarchical modelling and capabilities to model dynamic parallel computations are introduced in section 3.4.

3.1 Modelling Parallel Computations

The Computation Model described in this Chapter was designed with several goals in mind. Before presenting a definition of the model itself, a description of these goals is given. The basic capabilities required of the model are as follows:

- representation of the execution behavior of parallel computations
- ability to model both high level language constructs and architectural features
- introduction of time into the model to enable performance evaluation
- flexibility to use the model to study the execution efficiency of the computation at different points in its configuration space

The rest of this Section describes each of these points in greater detail.

The basic idea of the model is to be able to represent the execution behavior of parallel computations. The model should be able to capture some of the standard features of parallel computations such as synchronization and communication between processes and parallel execution of processes. It should also be able to model sequential program constructs such as iteration and branching. An important factor is that the model should be able to capture the actual execution overheads involved when the computation is run on a given architecture. This implies that the model must have the capability to model the underlying architecture on which the computation is to be run. Also, as previously noted, there should be some representation for the data state to allow explicit data-dependent flow of control.

The model should be able to represent parallel computations expressed in high level languages, as well as architectural features. One possible use for the model would be to take existing programs in high level languages, and translate them into instances of the model. The behavior of this model instance could then be studied on various architectures to obtain the best match.

The model is to be used as a vehicle for predicting the execution efficiency rather than correctness of the computation being studied. Most of the design decisions, therefore, are predicated upon a performance evaluation viewpoint rather than a theoretical view of Petri Nets. Since standard Petri Nets do not have the notion of time, only a partial ordering of transition firings can be obtained by analyzing the net. An extension must be provided, therefore, to model time more accurately.

Since the model is to be used to study the execution of a computation under various configurations, it must be flexible enough to allow easy modification of an instance of the model. In particular, it should be possible to alter factors like the degree of parallelism, the communication model and the underlying architecture without excessive overhead.

3.2 Definition of the Model

This Section contains a formal definition of the Parallel Computation Model (PCM), a model which captures the execution behavior of parallel computations. The model is an extended form of Petri Nets, with additional features to enhance the modelling of data dependent control flow, as well as hierarchical modelling to allow different levels of abstraction.

3.2.1 Parallel Computation Model

The Parallel Computation Model (PCM) is defined by the three-tuple

PCM = (PN, SV, TA)

where

PN = (P, T, I, O, M), a Petri Net, where
P = $\{p_1, .., p_n\}$, a set of places, $n \geq 0$
T = $\{t_1, .., t_m\}$, a set of transitions, $m \geq 0$
I is the transition input function, I : T --> 2^n,
or, I is a subset of PxT;
O is the transition output function, O : T --> 2^n,
or, O is a subset of TxP;
M = $[\mu_1, ..., \mu_n]$, a vector of integers specifying
the initial marking of the net;
and the sets P and T are disjoint

SV is a two-tuple <V, IV>
and
V = $\{v_1, v_2, .., v_r\}$, a set of state variables, $r \geq 0$
IV = $[iv_1, ..., iv_r]$, a vector of integers specifying

```
        the initial values of the state variables;

TA = is a three-tuple  <Π, Φ, T>
   and
     Π(V)  =  {π₁(V), π₂(V),...,πₘ(V)},
                            a set of predicates,
     Φ(V)  =  {φ₁(V), φ₂(V),...,φₘ(V)},
                            a set of procedures,
     T(V)  =  {τ₁(V), τ₂(V),...,τₘ(V)},
                            a set of delays,
```

The basic underlying structure is the standard Petri Net PN. In addition, the model allows a set of program variables (V), and attributes associated with each transition (TA).

Each transition t_i in the model defined above has a corresponding predicate $\pi_i(V)$ and a transition procedure $\phi_i(V)$. The predicates are defined on program variables, and are used to enable transitions under the modified firing rules. The transition procedures also operate on elements of V and can be used to modify them. The function $\tau_i(V)$ specifies the delay associated with transition t_i, and can also be a function of the set of program variables.

3.2.2 Firing Rules

The definition of the model is completed by specifying the modified firing rules. A transition can be in one of three states: *disabled*, *enabled*, or *firing*. The rules governing transition state changes are as follows.

1. A *disabled* transition t_i is *enabled* if:

   $\forall p_j \in I(t_i),\ \mu_j > 0$

   and

   $\pi_i(V)$ **is true**

   each of its input places contains at least one token in its enable region and the predicate corresponding to the transition is true.

2. An *enabled* transition t_i enters the *firing* state by removing one token from each of its input places:

$$\forall p_j \in I(t_i),\ \mu_j \leftarrow \mu_j - 1$$

3. A transition t_i remains in the *firing* state for a period of time specified by its delay function $\tau_i(V)$. A the end of this period, it places a token in the enable region of each output place, and executes its associated procedure. It then returns to the *disabled* state:

   $$\forall p_j \in O(t_i),\ \mu_j \leftarrow \mu_j + 1$$
   and
   $\phi_i(V)$ is executed, updating the state vector V

4. An *enabled* transition must continue to satisfy rule 1 until it enters the *firing* state, failing which it reverts to the *disabled* state.

Rule 1 states the conditions for a transition to be enabled. As with ordinary Petri Nets, each of its input places should contain at least one token; in addition, its predicate should be true. When a transition enters the firing state (Rule 2), it merely removes its enabling tokens. Transition firings are no longer instantaneous, as in the case of standard Petri Nets. Tokens appear on the transition's output places only after a certain delay, as specified in Rule 3. There are two ways in which the firing of a transition can effect other transitions (Rule 4). When a transition starts firing, the removal of tokens from its input places could cause other enabled transitions to become disabled. Also, when a transition completes firing, it modifies the marking as well as the state of the program variable vector, thereby causing other transitions to be either enabled or disabled.

3.2.3 Transition Predicates and Procedures

Transition predicates are bi-valued logic expressions in terms of the set of state variables V. Evaluation of a predicate depends on the current (data) state of the model, and returns either a true or false value, which is used to condition the enabling of its associated transition. The completion of firing of a transition is marked by the execution of its procedure, which is merely a sequence of assignment statements which update the elements of V.

The transition predicates and procedures are defined by means of a context-free grammar. All non-terminals in the grammar are enclosed within angular brackets '<' and '>'. Any symbol not enclosed within the brackets is a terminal symbol. The terminals in the grammar consist of the common logical and arithmetic operators which allow general boolean and arithmetic expressions. In addition, two special terminals, VAR and NUM are defined. For the predicates and procedures, VAR always refers to a state variable (i.e. an element of the set V), while NUM is any integer. The grammar specified below can be roughly classified into three groups: arithmetic expressions, predicate expressions and procedure statements. Each will be described in more detail next.

Figure 3-1 shows the part of the grammar that specifies general arithmetic expressions. Since the only values allowed are signed integers, the operations defined in the expressions are addition, subtraction, multiplication, and integer division and modulus. The grammar is structured to enforce the appropriate precedence between the operations (i.e. left to right evaluation except when the second operation has a higher priority than the first.)

```
 1     <aexpr> ::= <aterm>
                 / <aexpr> + <aterm>
 3               / <aexpr> - <aterm>
 4     <aterm> ::= <afactor>
 5               / <aterm> MOD <afactor>
 6               / <aterm> / <afactor>
 7               / <aterm> * <afactor>
 8     <afactor> ::= ( <aexpr> )
 9                 / VAR
10                 / NUM
11                 / + NUM
12                 / - NUM
```

Figure 3-1: Grammar for simple expressions

The next segment of the grammar (Figure 3-2) defines the transition predicates. A predicate consists of *atoms* which are connected using the logical operators AND and OR. Again, the structuring of the grammar enforces the precedence relations between the operators NOT, AND and OR. Each atom represents a boolean condition of the form specified in rules 22-27, where two arithmetic expressions are compared using a relational operator.

```
13      <pexpr> ::= <pterm>
14                / <pexpr> OR <pterm>
15      <pterm> ::= <pfactor>
16                / <pterm> AND <pfactor>
17      <pfactor> ::= ( <pexpr> )
18                  / <patom>
19                  / TRUE
20                  / FALSE
21                  / NOT <pfactor>
22      <patom> ::= <aexpr> = <aexpr>
23                / <aexpr> <> <aexpr>
24                / <aexpr> <= <aexpr>
25                / <aexpr> >= <aexpr>
26                / <aexpr> > <aexpr>
27                / <aexpr> < <aexpr>
```

Figure 3-2: Grammar for Predicates

The transition procedure is a sequence of simple assignment statements. Each assignment (rule 30 in Figure 3-3) has a single state variable on the left hand side, and an arithmetic expression on the right. Execution of the procedure causes each arithmetic expression to be evaluated, and the result stored in the corresponding state variable.

```
28      <proc> ::= <assnmt>
29               / <proc> ; <assnmt>
30      <assnmt> ::= VAR <- <aexpr>
```

Figure 3-3: Grammar for Procedure

3.2.4 Transition Selection Policy

When two or more enabled transitions share a common input place, the selection of one of these transitions for firing could cause the others to become disabled. The definition of the model is incomplete without specification of a policy for selecting a transition for firing from a set of conflicting, enabled transitions. The nondeterminism introduced by this policy adds to the power of the model. Several selection policies have been used in previous models. The conflict resolution submodel implicit in the Stochastic Petri Net (SPN) model [MOL81] is based on competing transition delays. The transition with the shortest firing time wins the conflict. In the continuous time SPN model, only one transition can complete firing at any instant. Given a set of conflicting transitions, the probability that a certain transition t_i completes firing first is equal to t_i's firing rate divided by the sum of the firing rates of all the competing transitions.

In our model, however, the conflict is resolved before any of the transitions begin firing. At any instant, a transition is selected from the set of enabled transitions with *equal* probability. After this transition begins firing, a new transition is selected from the modified set of enabled transitions. This process is repeated until no transitions remain enabled. In other words, any transition that can start firing is guaranteed to do so immediately, unless it is disabled by the firing of a conflicting transition. Also, several transitions may complete firing at any instant.

3.3 Some Properties of the Model

The state of the PCM net at any instant can be specified by the state of each node in the net, and the data state at that time. The state of each place is the number of tokens currently residing in that place. The state of a transition is specified by its *Remaining Firing Time* (RFT, [ZUB80]), which specifies the time required for completion of firing of the transition. If the transition is not firing, its RFT is zero. Since the variables allowed in the model are all integers, the data state is given by a vector of integers, each specifying the current value of a variable.

If the state variables are disallowed, the model reduces to a timed Petri Net similar to the Generalized Timed Petri Nets defined by Holliday [HOL85], and similar techniques can be used to analyze the model. The addition of the unconstrained predicates and procedures, however, inhibits such analysis.

The modelling power of the PCM model is equivalent to that of a Turing machine. The proof of equivalence consists of showing the equivalence of the PCM model to Petri Nets with inhibitor arcs, which are known to be equivalent to Petri Nets in their modelling power. A formal definition of Petri Nets with inhibitor arcs, along with the reduction of the IPN to an equivalent PCM are shown in Appendix A.

3.4 Hierarchical Modelling and Parameterization

Petri Net models are well suited for hierarchical modelling. It is possible to collapse entire subnets into a single transition, where the firing of the transition implies execution of the contained subnet. This allows the user to model a system at different levels of abstraction. In addition, the representation shown below allows powerful parameterization features to be incorporated in an elegant manner. The parameterization greatly reduces the number of nodes in the Petri Net graph, and allows specification of subgraphs dependent on the dynamic state of the net. This feature has proved to be extremely useful for the representation of parallel computations, whose structure often depends on an intermediate computation state. It also allows a structured approach to modelling, which is the basis for the methodology presented in the next Chapter.

3.4.1 Subnets and their firing rules

PCM supports hierarchical modelling by allowing transitions to be specified as *subnets*. A subnet transition has a *subgraph* associated with it. This subgraph is constrained to begin and end with special transitions called the *initiating* and *terminating* transitions. Figure 3-4 shows a subnet transition and its associated subgraph. The firing rules for the subnet transition are slightly different from the general firing rules given earlier, and are specified next. As before, the transition can be in one of three states: disabled, enabled or firing. In the following text, subnet transitions are simply referred to as subnets.

1. A disabled subnet t_i is enabled if each of its input places contains at least one token and the predicate $\pi_i(V)$ corresponding to the subnet is true.

2. An enabled subnet t_i enters the firing state by removing one token in each of its input places. At this point, the subnet is said to be *active*. In addition, the initiating transition of its associated subgraph enters the firing state.

3. A subnet t_i remains in the firing state until the terminating transition of its associated subgraph fires. At this point, the subnet ceases to be active, places a token in the enable region of each output place, and executes its associated procedure $\phi_i(V)$. It then returns to the disabled state.

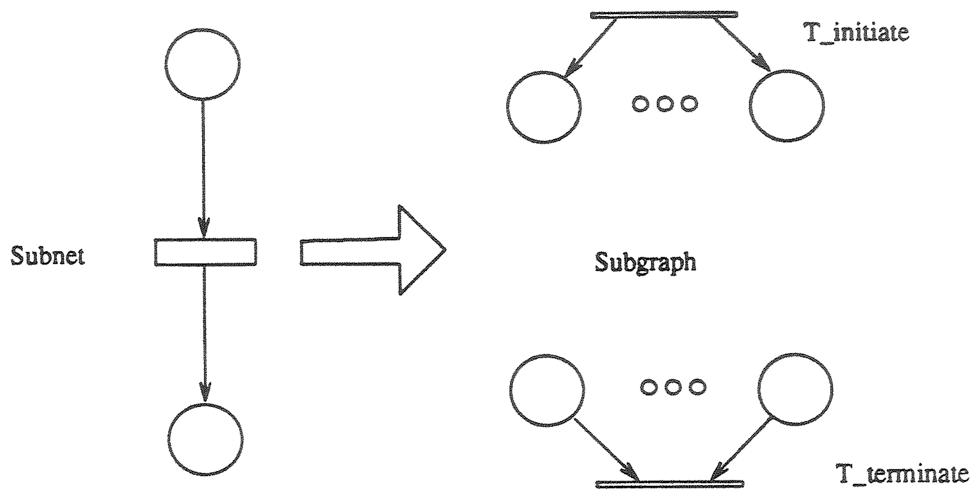


Figure 3-4: Representation of a subnet

4. An enabled subnet must continue to satisfy rule 1 until it enters the firing state, failing which it reverts to the disabled state.

The rules specifying conditions for the subnet to be enabled (Rules 1 & 4) are identical to those for general transitions. The other two rules specify the actual firing of the subnet. As in the case of ordinary transitions, subnets begin firing by moving tokens from their input places from the enable to hold regions. Instead of simply introducing a delay at this point, however, control is passed to the subgraph associated with the subnet. This is achieved by firing the *initiating* transition of the associated subgraph. Similarly, the firing of the *terminating* transition of the subgraph signals the completion of firing of the subnet.

The introduction of subnets into the model raises an important issue: can transitions within a subnet be enabled if their parent subnet is not active? The decision taken in our model is that for a transition to be enabled, it *should* belong to an active subnet. This choice is well founded if subnets are considered to consist of actions that are logically related. Unless the global conditions for the entire set of actions is true, it is meaningless to allow individual actions to occur. In other words, unless the subnet itself is firing, it makes little sense to allow its component transitions to fire.

3.4.2 Subnet and Place parameters

The model allows parameters to be associated with subnets and places. This feature provides a compact representation for multiple replications of identical net fragments. This is especially useful, since our experience with parallel computations has shown that they often contain several streams of identical statements to be executed in parallel. Also, the parameters can be dependent on the set of program variables, thus allowing the representation of dynamic nets.

Figure 3-5 shows a parameterized subnet and its equivalent expanded net. This expansion is done at the time the subnet becomes active. In the figure, the only subnet parameter specified is *i*, which takes values from 1 to 2.

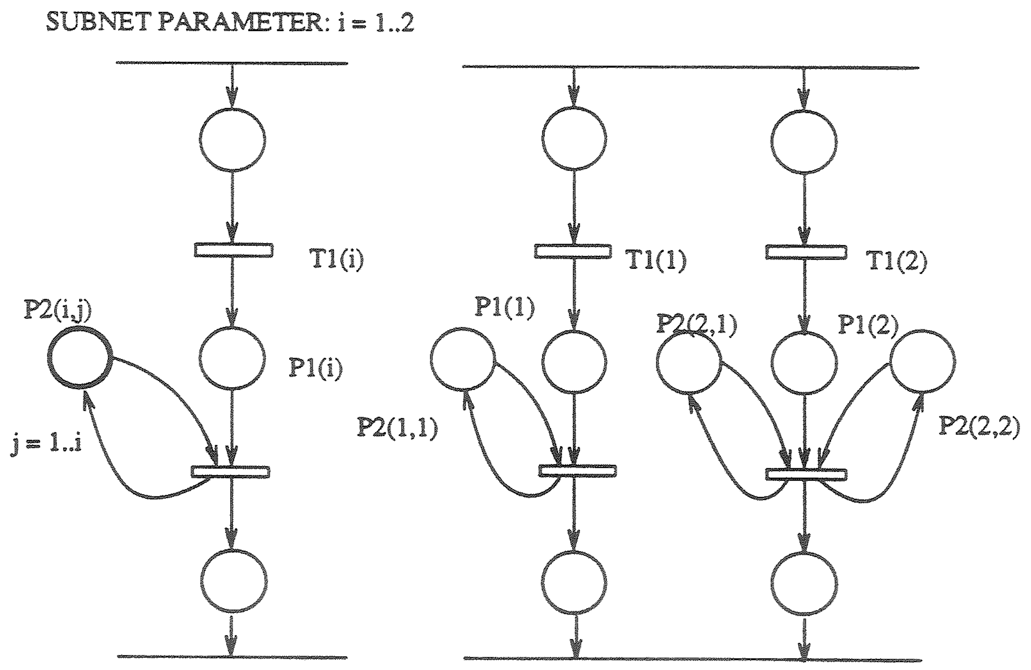


Figure 3-5: Expansion of a subnet with parameters

These values are applied to all nodes in the subnet with the exception of the initiating and terminating transitions. Application of a parameter to a node includes replication of the node with the parameter value substituted in all the node attributes. In Figure 3-5, for example, application of the parameter i to node T1(i) results in two nodes T1(1) and T1(2). If the parameter i occurs in any of transition T1's attributes (predicate, procedure or delay), it is replaced by either 1 or 2.

Places can also have parameters associated with them. Place P2(i,j) in Figure 3-5 shows a place with parameter j going from 1 to i. For the stream with i equal to 1, this implies a single place P2(1,1), while the other stream (with i equal to 2) has two places P2(2,1) and P2(2,2). This allows compact representation of multiple places as inputs or outputs to a single transition.

3.4.3 Scope rules for parameters

Bounds for subnet and place parameters can be expressed in terms of simple integer values, program variables, and other parameters. They can also be specified as expressions consisting of all three types. The use of integer values needs no clarification. Where program variables are used, the value used is the value of the variable at the time the subnet becomes active.

Whereas integers and program variables can be used anywhere in the net, the use of parameters as bounds for other parameters is allowed only in conjunction with some scope rules. There are two cases where this situation arises. The simple case is when a subnet parameter list contains two parameters, with one parameter being a bound for the other. In this case, the *horizontal* scoping rule applies, which states that a parameter be declared before it is used. For example:

```
i = 1 to 5; j = 2 to i;
```

is legal, whereas

```
j = 2 to i; i = 1 to 5;
```

is not.

The second rule, called the *vertical* scoping rule, describes how parameters filter through nested subnets. The vertical scope rule states that a parameter can be legally used within a subnet if:

- it has been declared as a parameter for that subnet, or
- it appears as the bounds for another parameter specification and satisfies the horizontal scope rules, or
- it is a legal parameter for the subnet's parent net. In this case, the parameter has a value associated with it which specifies the subnet's instantiation value at the higher level.

3.5 Comparison with Other Models

The heart of the PCM model remains the Standard Petri Net, which can be obtained by stripping the nodes (places and transitions) of the graph of all their attributes. The addition of the state variables and transition attributes leads to the additional modelling power, and the model becomes equivalent to a Turing machine. It has been shown that certain questions such as reachability and boundedness become undecidable for classes of Timed Petri Nets [JON77], and the evaluation of the net is accomplished by simulating its behavior.

The model defined here can be viewed as a form of a Predicate/Transition net [GEN81] (described in Chapter 2), with the state variables and transition predicates and procedures being used instead of structured tokens and arc expressions. The PCM model allows global information (using the state variables), whereas the PrT nets enforce the use of only local information. The PCM model is therefore a member of the class of Colored Petri Net models.

Since the number of replications of streams within a PCM subnet is dependent on the state of the net at the time of activation, the model is, in essence, equivalent to the self-modifying nets (SM nets, [VAL78]) described earlier. The effect of the firing a PCM subnet (its "firing rules") can vary over successive firings, as in the case of SM nets.

Chapter 4

A Methodology for Modelling Parallel Computations

Modelling the execution behavior of a computation can be viewed as consisting of three distinct submodels. The *computation* submodel specifies the computation at an abstract level, and includes details such as the processes that constitute a computation, the interaction between these processes and the logical dependencies between them. The *architecture* submodel specifies the details of the architecture on which the computation is to be executed. This submodel captures the particular details of the architecture which effect the execution of the computation such as the mechanisms provided for communication (shared memory or message passing), as well as processor delays incurred for ordinary sequential operations. The *mapping* submodel defines the binding of the processes and data objects in the computation submodel to processors and memories in the architecture submodel. This submodel also specifies scheduling decisions which may be required for initiating the binding between processes and processors.

A methodology for modelling parallel computations based on these submodels is described in this Chapter. The formulation of such a methodology eases the burden of modelling complicated computations, and helps to ensure that the obtained model faithfully reflects the actual execution behavior of the computation. Since the steps defined for constructing each submodel are general in nature, the entire procedure can be easily automated. By carefully defining the interfaces between the various submodels, it becomes possible to create a library of architecture submodels which can then be used when building a composite model. Further, using a uniform methodology ensures a fair comparison of alternate configurations of a given computation.

The various submodels are described in each of the following sections. The last Section highlights the advantages of using the three submodels to obtain the composite model.

4.1 Computation Submodel

The Computation Submodel models the execution of a computation on an abstract machine. The characteristic of the abstract machine is that it is an *ideal* machine for the computation, and does not introduce any restrictions or delays in the execution of the computation other than those due to the computation itself. Since a computation usually consists of processes that interact using either messages or shared data, the abstract machine has one one processor for each process, with links between those processors which need to communicate and shared objects accessible, with no additional overhead, to all processors which need to read or write them. This definition of the computation submodel is consistent with the definition of most languages which specify parallel computations: the language implicitly assumes an abstract underlying architecture, and compiling programs encoded in these languages for a given architecture introduces additional restrictions imposed by the architecture. The starting point for development of the computation submodel is a specification of the computation in some well defined language. In this Section, the computations are specified using a restricted form of Computation Graphs ([BRO85]), a formal model for specifying computations. Appendix B shows constructs in the computation submodel for representing programs written in two other parallel languages: the Computation Structures Language (CSL), and the Task-level Data Flow Language (TDFL).

4.1.1 Computation Graphs

A Computation Graph [BRO85] is a formal model for specifying computations, and is used as a starting point for developing the computation submodel from the specification of a computation. A Computation Graph is a program for some computation in terms of the operations of an abstract machine. It provides the framework for representing parallel computations by specification of actions and dependencies between actions. A computation graph is a directed graph in which the nodes represent schedulable units of computation and the arcs represent dependencies between them. Execution of the computation is attained by traversal of the directed graph along the paths defined by the dependencies. No restrictions are placed on the rules for traversal of the computation graph other than satisfaction of all dependency constraints. The Computation Graph model allows several types of dependency constraints, but only the following are considered here:

- sequencing dependencies specify an ordering relation between execution of the source and sink computation units.

- synchronization dependencies specify a set of computation units which cannot execute simultaneously (i.e. mutual exclusion).
- producer/consumer dependencies specify that the sink must receive some information from the source before it may execute. The most common type of information exchanged is data, in which case the constraint reduces to a simple data dependency.

Computation Graphs provide a flexible and powerful representation for parallel computations. The clean separation of computations and dependency relations allows each to be resolved individually and mapped onto the abstract machine. Hierarchical abstraction is easily achieved by grouping several computation units into a single unit. Resolution of the computation to greater detail is obtained by replacing a node by a subgraph.

The computation graph model can be used to represent several well known methodologies for parallel computation by appropriately choosing traversal rules and allowed dependency constraints. For example, if the only dependencies permitted are data dependencies, and the traversal rule states that the existence of data 'tokens' on all input arcs to a computation unit causes its execution and the execution of a computation unit creates data 'tokens' on each of its output arcs, the resulting representation would be equivalent to a basic form of data-flow [DEN72].

The firing rules selected in the restricted version are as follows: each node can be activated only when all its input dependencies are satisfied. Sequencing and data dependency arcs carry tokens which are placed on the arc upon completion of execution of the source node, and are consumed by the sink node when it is activated. If a node has only sequencing or data dependency arcs as its inputs, it is activated when all input arcs carry tokens. If two or more nodes satisfy all their input sequencing and data dependencies and are connected by a mutual exclusion arc, one of the nodes is selected at random and activated. Though the restricted model lacks the power of the original model, it provides base for development of the computation submodel.

The Computation Structures Language presented in Appendix B can be viewed as a procedural specification of a computation graph and its traversal. Execution of a CSL program is equivalent to a parallel traversal of the computation graph it specifies. The language provides primitives for user specification of certain types of dependency constraints. While these primitives do not cover the entire range specifiable using computation graphs, they are sufficiently powerful to be able to represent a wide variety of computations. Similarly, the Task-Level Data Flow language (TDFL), also described in Appendix B, is also an instance of the Computation Graph model with special types of dependencies defined between computation nodes.

4.1.2 Conversion of CGs to CSMs

Figure 4-1 shows an example of a computation expressed using the Computation Graph model. The graph consists of 6 computation units connected by arcs specifying the dependencies between them. The example contains all three types of dependencies allowed in the restricted model: sequencing (S), date (D) and mutual exclusion (M).

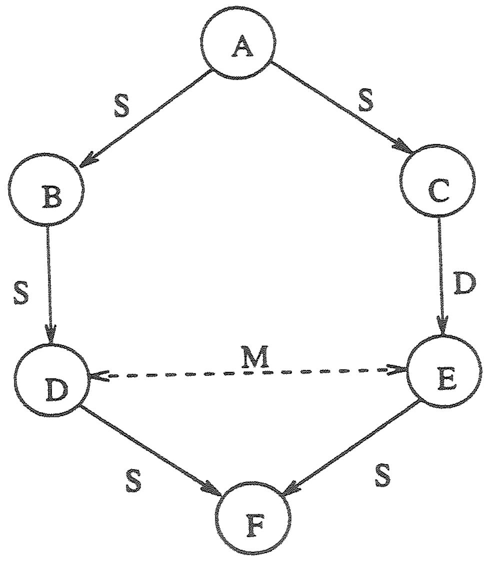


Figure 4-1: Example of a Computation Graph

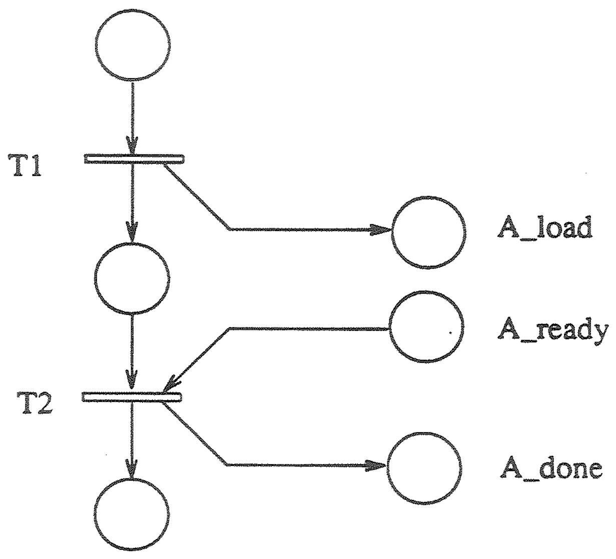


Figure 4-2: Execution of a Node

A computation submodel can be specified for this example by making the following observations. Each node in the computation graph is associated with a processing element in the underlying abstract machine. Since a node in a CG, by definition, is an independent entity, its execution has no side effects on any of the other nodes, and its contribution to the state of the overall computation is merely a delay representing the time taken for its execution. In the submodel, the execution of a single node, say node A, would be as shown in Figure 4-2. Before A can execute, it has to be loaded on (or, bound to) a processor. This binding is not explicit in the computation submodel. To initiate loading of the process, a token is placed in place *A_load* by the firing of transition *T1*. When the process has been loaded, a token arrives in place *A_ready*, and *T2* fires, signifying the execution of node A. Transition *T2* has a delay attribute equal to the expected execution time of the node. The process signals completion of execution by placing a token in *A_done*, which effectively releases the processor associated with A.

At this level of abstraction, architectural details such as the number of processors available and the time required to load a process on a processor are left unspecified. In fact, the 'ideal' architecture can be specified by properly terminating these requests in the mapping submodel.

The complete computation submodel for the example (Figure 4-1) is shown in Figure 4-3. The execution of each node is modelled by the firing of a single transition, and the loading of processes described above is not shown so as to simplify the figure.

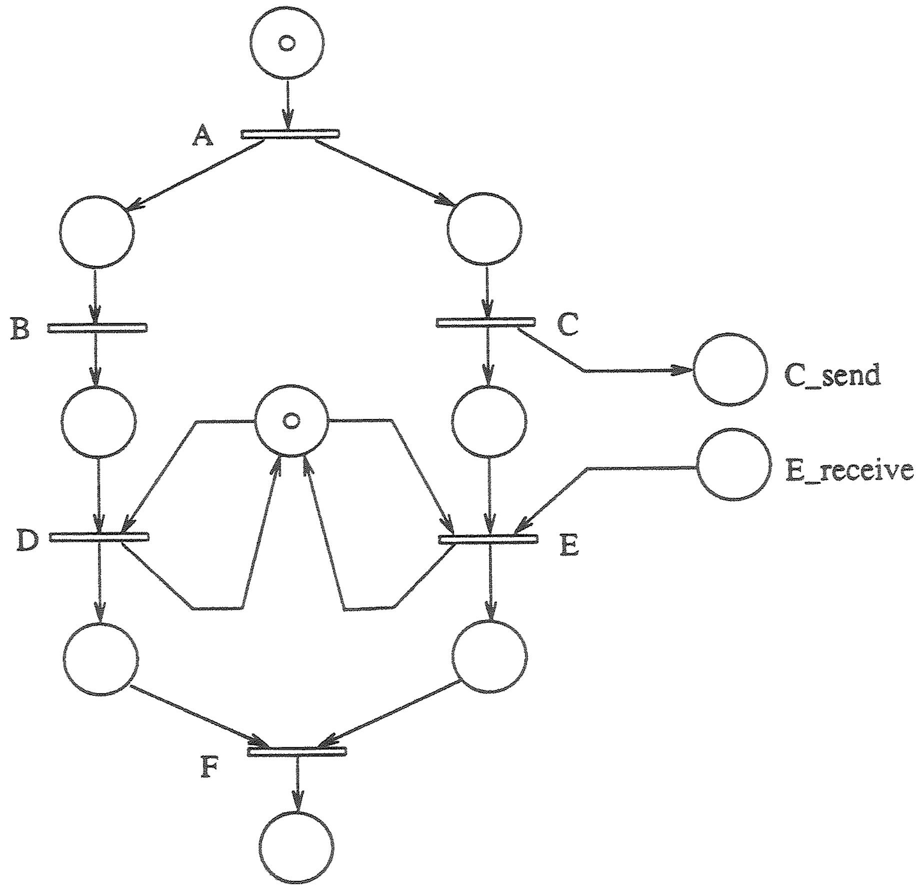


Figure 4-3: Computation Submodel Equivalent

Sequencing dependencies are modelled by simply connecting the corresponding transitions by a common place. The mutual exclusion dependency between nodes

D and E in the Computation Graph is modelled by adding the extra place *Mutex* in the in the submodel. After transition *C* completes firing, a token is placed in place *C_send*. Depending on the actual architecture, a token arrives in *C_receive* after some time delay. The additional information required for sending data to a process are the process id and the size of the data item. These are passed by associating two state variables with *C_send*.

Using these fixed translation rules, an equivalent computation submodel can be created for any given computation graph. Similar rules can be specified for any given language (see Appendix A). The interface between the computation submodel and the mapping submodel is defined in greater detail in the next Section.

4.2 Mapping Submodel

The mapping submodel represents the binding of entities in the computation submodel (processes, shared data objects, etc) to entities in the architecture submodel (processors and memory modules). The structure of the mapping submodel remains unchanged over variations in the actual mapping. Only the node names and attributes (of places and transitions) need to be altered. Due to this feature, the mapping submodel lends itself naturally to automatic generation using a predefined template. It also presents a standard interface to the architecture submodel, thus making the representation basis modular.

Figure 4-4 shows the mapping submodel to map a process *S* to a processor *P(0)*. The places prefixed by "S_" are shared between the mapping submodel and the computation submodel which defines the process *S*, while the places prefixed by "P_" are shared with the architecture submodel.

A token placed in place *S_load* initiates loading of the process. When loading is completed, a token is placed in *S_ready*. Also, when a process is loaded on a processor, the processor itself becomes busy (signified by the presence of a token in *P_busy(0)*). Once a process has been loaded, it can make a request. The three types of requests possible are accessing a shared object, sending a message to another process and receiving a message from another process. In addition, the request must be accompanied by additional information specifying the shared object name or the process

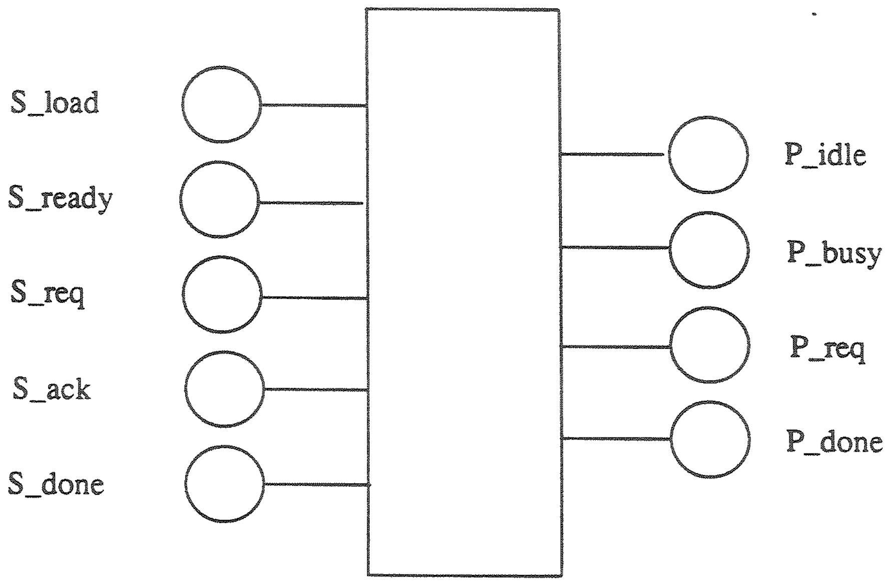


Figure 4-4: Functions available in the mapping submodel

id of the receiving (or sending) process along with the size of the message to be sent. This is done by associating three state variables with place *S_req*. The request is then translated in terms of the architecture, and a token is placed in *P_req(0)*. When the request has been satisfied, a token arrives in *P_done(0)*, and consequently, in *S_ack*. Finally, when process *S* has completed, it releases Processor 0 by signalling place *S_done*, which moves a token from *P_busy(0)* to *P_idle(0)*. The actual mapping submodel for this example is shown in Figure 4-5.

The state variables required for this subnet are: S_req_type, S_req_id, S_req_size, P_req_type(0), P_req_id(0) and P_req_size(0). In addition, each processor has a queue associated with it. When a process initiates loading, it is entered into the queue. When the processor becomes idle, it selects a process from the queue using some predefined strategy. Commonly used strategies include FCFS and Priority. Transition *T1* has a function which enqueues the process, while *T2* has a predicate which checks to see if the process is to be loaded next. Firing *T2* causes the process to be removed from the queue. Transition *T3* copies the request information from the

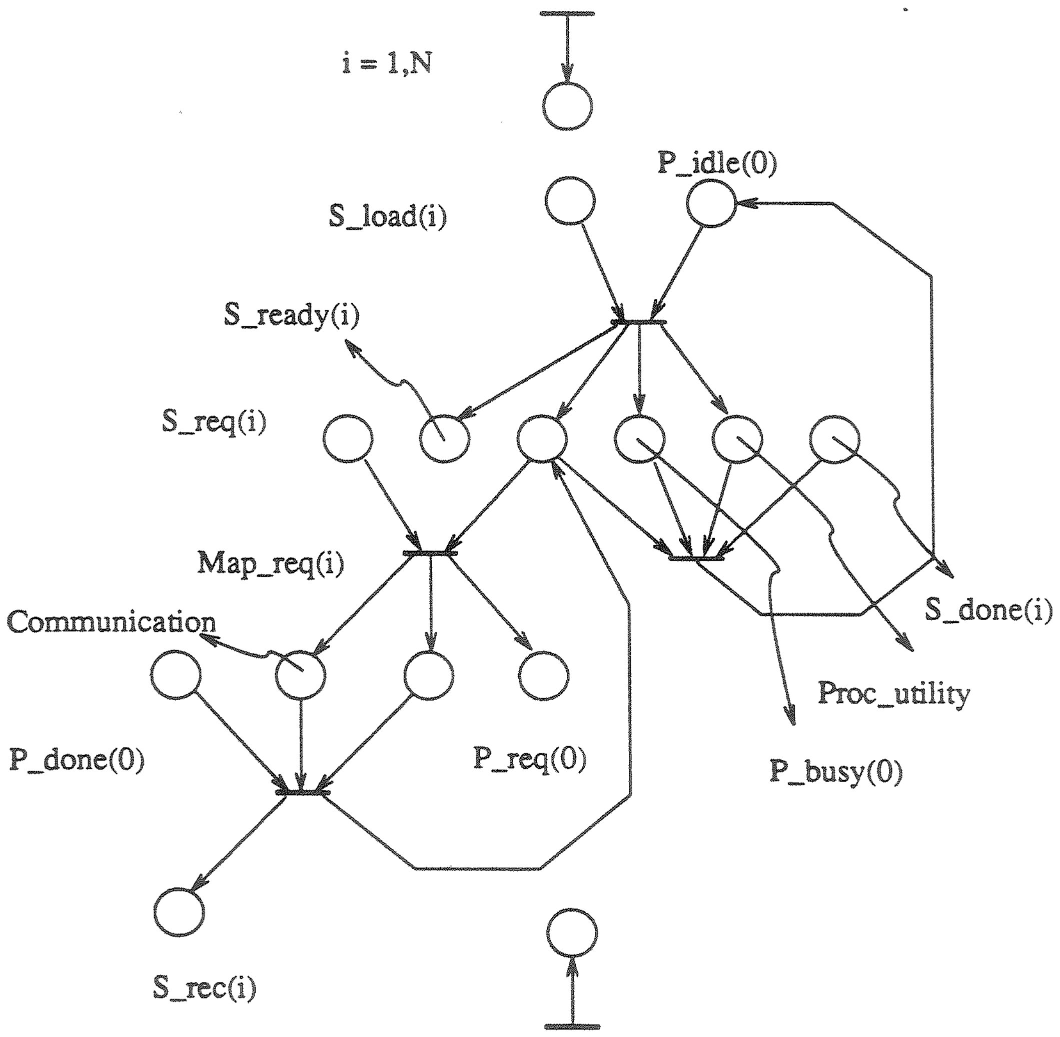


Figure 4-5: Mapping Process S to processor 0

process to the processor. The additional places shown in the figure (Proc_utility and Communication) are used to aid performance measurement.

The method described above for mapping the computation onto the architecture requires, in the worst case, one mapping subnet for each process in the computation. However, in most cases, computations are made up of several identical tasks, and the mapping is usually done in a regular manner. If these conditions are met, an entire set of (identical) tasks can be mapped using a single subnet merely by using appropriate indices for both the processes as well as the processors. Consider the following examples:

	Process Index	Processor Index
a)	i	0
b)	i	i mod 2
c)	i	i mod p

In case (a), all the processes are mapped onto processor zero. In case (b), all even numbered processes are loaded on processor 0, while all odd numbered processes are loaded on processor 1. If p is the number of processors in the system (by convention, the processors are always numbered from 0 to p), case (c) shows a mapping which 'wraps' the processes over the p processors.

If the execution behavior of the computation on the abstract machine is of interest, the mapping submodel can be greatly simplified. All requests to the architecture are acknowledged immediately, without being passed on to the architecture submodel. For example, the moment a token arrives in *S_load*, it is removed and a token placed in *S_ready*.

The mapping submodel provides a standard interface between the computation and architecture submodels, and, in effect, isolates them from each other. This allows the creation of an architecture submodel library which can be drawn upon to select the best architecture for a given computation. The architecture submodel is described briefly in the next Section.

4.3 Architecture Submodel

The architecture submodel represents the behavior of the architecture on which the computation is being executed. It captures the details of the architecture which affect the sending and receiving of messages, and the accessing of remote (shared) memory modules. The sequential behavior of each processor is not modelled explicitly, but is represented as a delay. Chapter 6 describes the architecture submodels for two types of shared memory architectures in detail. In this section, that part of the architecture which interfaces with the mapping submodel is described. Figure

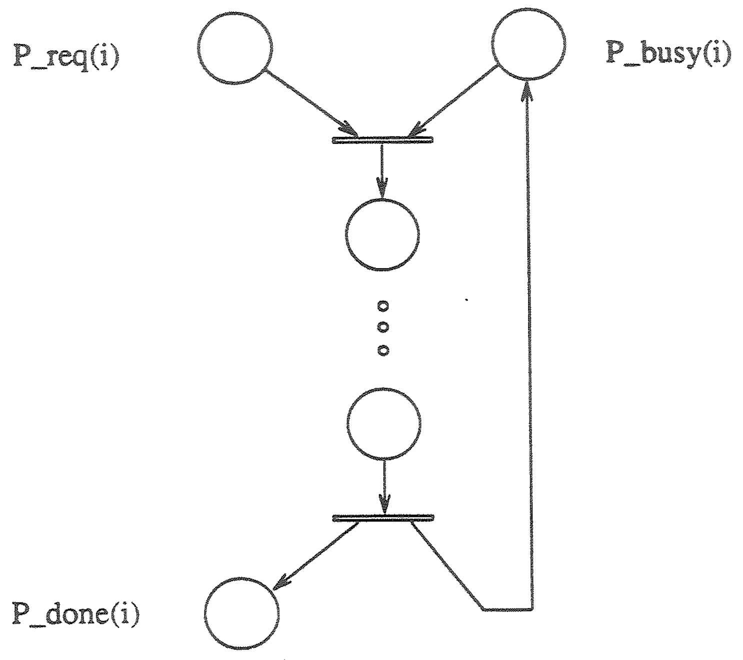


Figure 4-6: Architecture Interface with the Mapping Submodel

4-6 shows this fragment of the architecture submodel. Once a processor is busy, if a request arrives (a token in *P_req*), the processor is activated, and *T1* fires. The processor then determines the type of request and proceeds to satisfy the request. Once the request has been satisfied, *T2* fires, and the token in *P_done* signals completion to the mapping submodel. Since all processors in a parallel architecture are usually identical, a single subnet is sufficient to represent them.

Defining architectures in this uniform method has several advantages. The architectural submodels can be tested in isolation and then included in a library to be used in conjunction with any computation. Typical state variables for each architecture are the number of processors and memory modules, allowing a family of architectures to be represented. In addition, the models in the library are set up to report typical performance figures such as processor utilities and memory bandwidths, as well as communication delays incurred due to the architecture.

4.4 Advantages of this Organization

There are several advantages of using the submodels described in the previous Sections. The major advantage of this organization is the separation of the three, logically distinct, submodels. Changes to any submodel do not effect the behavior of any other submodel as long as the standard interface between them is maintained. To evaluate the effect of various mappings on the execution efficiency of a computation, for example, it is sufficient to modify *only* the mapping submodel, while leaving the other submodels unchanged. Similarly, the architecture can be modified without affecting any of the other submodels. Changing the computation submodel has a similar effect if the computation consists of the same tasks as before.

The architecture submodel can be tested in isolation to verify that it faithfully reflects the behavior of the actual architecture. The standard interface between the architecture and mapping submodels facilitates the creation and utilization of a library of architectures. The presence of such a library enables the search for a suitable architecture for a given computation. Further, by properly defining the mapping submodel, the behavior of the computation given an ideal architecture can be evaluated. This is useful since it yields upper bounds on the performance characteristics obtained.

Chapter 5

PCSIM: A Performance Evaluation Tool

This Chapter describes PCSIM, a tool for the design and performance evaluation of parallel computations. The tool consists of a graphics front end to edit PCM net instances, and a simulator which takes the net generated, simulates the Petri Net's dynamic behavior, and generates performance statistics. Section 5.1 discusses some features of the simulator and specifies the types of results that can be obtained. Places in the net contain special functions which are active during the simulation, and are used to gather performance statistics by monitoring the movement of the timestamped tokens. A specification of this time resolution mechanism is given in Section 5.2. Section 5.3 describes the attributes associated with the transitions and places in the net. Finally, Section 5.4 shows some validation results obtained for the simulator, where results obtained from the simulator are compared with results obtained from various sources. The simulator results are compared with analytical results as well as results obtained from actual execution of computations on parallel machines.

5.1 The Simulator Package

The simulator package consists of a graphics front-end and the PCM simulator which accepts nets created using the editor as its input, simulates them, and generates the required performance statistics. The graphics front-end facilitates the creation and modification of net instances prior to simulation. The package shares a common graphics editor with the Stochastic Petrinet Analyzer (SPAN, [MOL86]).

Since the statistics reported are all mean values, there is a need to ensure that the simulation has been conducted for long enough so as to give accurate results. The best method of ensuring this is to build in confidence intervals to control the length of the simulation [LAW82]. The implementation of PCSIM used in this dissertation, however, does *not* include the confidence intervals. Instead, the length of the simulation is controlled externally by specifying either the length of the simulation, or the number of runs desired. Careful study of the results must be done to ascertain that they are accurate.

The input specification for each simulation (other than the net itself), are as follows:

- Trace Level
- Trace File
- Number of executions of the net
- Length of simulation
- Initialization function for program variables.

The simulator prints a report of the various intermediate states during the execution of the net. This report can be used to verify the correctness of the model by studying the sequence of events in the system. This report can be in varying degrees of detail. This is controlled using the *trace level*, which can vary between zero and five. A trace level of zero produces no output whatsoever. The only output from the simulator is the total execution time. As the trace level increases, more information is printed on the specified *trace file*. Typical information printed is the enabling and firing times of transitions, the state of the net including the values of its program variables, and values returned by the transition delay functions.

There are two ways to control the length of a simulation. The *Number of executions* specifies how many times the net is to be executed. Each execution of the net begins with the initial marking and ends when no transitions are enabled. The *length of the simulation run* simply specifies that the simulation be interrupted when the simulation clock reaches a value greater than the specified length. The simulation results are normalized for the number of net executions. The *initialization function* specifies the initial values of the set of state variables. This function is invoked at the start of the simulation as part of the initialization of the net, and also whenever the net needs to be reinitialized.

The simulation is carried out using the conventional event list, where each event is the completion of firing of a transition at a fixed time. A list of enabled transitions is maintained and after firing all transitions at any instant, this list is scanned for more transitions which can start firing. An execution of the net is complete when the event list is empty. The transition input and output functions are represented using Vector Replacement Systems [KEL74]. The actual implementation uses sparse matrix techniques to compact the size of these vectors.

5.2 Time Resolution Mechanism

This section describes extensions to the model to allow performance evaluation. The basic idea is to associate time delays with each transition, as in ([KAP82], [BER79], [NUT72]). This is achieved by partitioning places into two regions - *hold* and *enable*. When a token arrives at a place, it is put in the enable region. When a transition starts firing, it moves its enabling tokens from the hold to enable regions of its input places, where they are delayed by a certain time specified by the transition. After this delay, the transition completes firing, removing the tokens from the hold regions of its input places, and placing tokens in the enable regions of its output places. The actual attributes required at places, transitions, and tokens are discussed next.

Since tokens are created and destroyed by transition firings, it is not sufficient for tokens alone to carry timing information. In addition, places are defined to have performance functions which are called whenever tokens in that place are moved. The definition and placement of place and token attributes is predicated upon a performance evaluation viewpoint rather than a theoretical view of Petri nets. The addition of these attributes, however, does not alter the semantic behavior of the underlying net.

The PCM transition is a composite entity defined as follows:

- Transition Delay Function (TDF): Each transition has a set of delay functions which can be deterministic, exponential, etc. This function is used to determine how long a transition takes to complete firing.
- Transition Delay (TD): Each time a transition is fired, its TDF is executed to determine the delay for this firing.

In addition to executing its delay function, the transition also invokes the place performance functions for its input and output places.

The PCM place is a composite entity defined by the following attributes:

- Place enable set: This holds a set of tokens which can be used to enable transitions.
- Place hold set: This holds a set of tokens held by a firing transition, i.e. tokens receiving service in conjunction with tokens from other places.
- Place performance functions: Functions to keep track of total times spent in the enable and hold set, number of tokens passing through the place, etc.

These place performance functions are invoked by a transition at two instances: when a transition begins firing, and when it finishes firing. These are the only two times when token movement takes place. The details of the place performance functions will be given later.

The PCM token is modified to have the following attributes:

- Token arrival time stamp (TAS): The time at which a token arrives at a place. When a transition fires and creates tokens, it initializes this timestamp using the time it completes firing.
- Token enable time stamp (TES): The time at which a token moves from the enable region to the hold region. The enabled transition will now fire after a time equal to the transition firing time.
- Token firing time stamp (TFS): The sum of TES and TD for that token.

Performance metrics can be extracted from the values stored in places and tokens. Alternately, if a state history of the execution of the net is maintained, this history can be used to derive the desired metrics. If intermediate values of tokens need to be stored, a copy of the token can be placed in a special place with no output arcs using a special transition which simply replicates the input token.

The different place performance functions are specified next. These functions collect performance figures while tokens are passed through the place. Each place has a set of local variables which are operated on by the functions:

- *Num_enable* is the number of tokens currently in the enable region of the place.
- *Tot_enable* is the total (cumulative) time spent by tokens in the enable region.
- *Token_enable* is the sum of the product of the number of tokens in the enable region and the time spent in the region. This product is useful for evaluating the mean number of tokens in a place.
- *Last_enable* is the time that *Token_enable* was last updated. This value is used to update *Token_enable* each time token movement occurs in the enable region of a place.
- *Num_hold*, *Tot_hold*, *Token_hold* and *Last_hold* are similar variables for the hold region.
- *Traffic* is the number of tokens which have passed through the place.

The place performance functions specified in Figure 5-1 are called by the transitions when they enter the firing state. The *enabling* tokens for a transition are those tokens which have caused the transition to be enabled.

The transition starts firing at the moment the last of its enabling tokens arrive at its input place. This is the maximum of all the TAS values of the enabling tokens. The transition delay function is used to calculate *Transition_delay*, the time for which the enabling tokens will be in the hold region. The TES and TFS values of each enabling token can now be updated.

The place variables are now updated as follows. The total time tokens spend in the enable region (*Tot_enable*) is merely the difference between the firing (TES) and arrival (TAS) time for each enabling token. Since the number of tokens in the enable and hold regions is being modified, the values of *Token_enable* and

```
Trans_start_firing ()

{
 TIME  start_firing_time;

 start_firing_time = MAX(TAS of enabling tokens);
 Transition_delay = invoke Trans Delay Function;

 For each enabling token t,
    t.TES  = start_firing_time;
    t.TFS  = Transition_delay;

 For each input place p & each enabling token t,
     p.Tot_enable +=  t.TES - t.TAS;

 For each input place p,
     p.Token_enable += p.Num_enabled
           *(start_firing_time - p.Last_enabled);
     p.Token_hold    += p.Num_hold
           *(start_firing_time - p.Last_hold);
     p.Last_enabled  = start_firing_time;
     p.Last_hold     = start_firing_time;
     p.Num_enabled -= # enabling toks for place p;
     p.Num_hold   += No  enabling toks for place p;
}
```

Figure 5-1: Function invoked when a transition begins firing

Token_hold are updated by adding to their old value, the new product of the number of tokens and the time since the last update. Finally, the time of the last updates (*Last_enabled* and *Last_hold*) are updated, and the number of tokens in the hold and enable regions is adjusted to reflect the start of transition firing.

The function called when a transition completes firing (Figure 5-2) is similar to the starting function. The main difference is that both input and output places are involved now, since tokens are moved from the hold region of the input places to the enable region of the output places. Also, when the tokens leave the hold region, the traffic variable of the input place is updated.

These functions update the place variables as tokens move through the net.

```
Trans_end_firing ()

{
  TIME  end_firing_time;

  end_firing_time = t.TFS of any enabling token;

  For each input place p & each enabling token t,
     p.Tot_hold +=  t.TFS - t.TES;

  For each input place p,
     p.Token_hold   += p.Num_hold
                  *(end_firing_time - p.Last_hold);
     p.Last_hold      = end_firing_time;
     p.Num_hold     -= No. of trans enabling tokens;
     p.traffic      += No. of trans enabling tokens;

  For each output place p,
     p.Token_enable += p.Num_enabled
              * (end_firing_time - p.Last_enabled);
     p.Last_enable    = end_firing_time;
     p.Num_enabled  += # output toks for trans;

}
```

Figure 5-2: Function invoked when a transition completes firing

When the simulation is completed, these variable values are interpreted to give statistics like mean number of tokens in the hold and enable regions, mean time spent in the hold and enable regions and throughput at each place. The interpretation given depends on the place type specified before simulation. Place types, and other attributes, are discussed in the next section.

5.3 Specification of Node Attributes

Places, transitions and subnets have associated attributes. In this section, these attributes are described in greater detail.

5.3.1 Place Attributes

The definition of the model in the previous chapter did not assign any attributes to places, other than the initial number of tokens in the place. The attributes are mainly to aid in the interpretation of timestamp values collected at the place. Each place in the net has the following attributes:

- place tag, a unique name identifying the place
- initial marking, the initial number of tokens in the place
- place parameter list, if the place is parameterized
- place type, for interpretation of timestamp values

The first two attributes need no explanation. The place parameter list is as described in the previous chapter. Place types can be of types:

- resource, token represents an available resource
- condition, token represents presence of a condition
- process, token represents a process waiting for some service
- cumulative time, place used to accumulate total elapsed time for some event
- duplicate, place definition is given elsewhere
- default, used for uninterpreted places

For resource places, the mean utilization of resources is the only metric reported. Condition places report the mean time the condition was true. The most important place is the process place, used to model processes queueing up for some service. They could also be requesting some resource, or be waiting for some synchronization condition to become true. The metrics reported for resource places include the mean queue length (mean number of tokens in the enable region), the mean number in service (mean number of tokens in the hold region), the mean wait time (mean time spent in the enable region), mean service time (mean time spent in the hold region) and throughput (mean rate tokens pass through the place).

5.3.2 Transition and Subnet Attributes

Transitions and subnets have similar attributes, except that subnets can be parameterized. If a subnet has no subgraph defined, it behaves exactly like a transition. Its delay is calculated using the delay attributes specified. If a subgraph is defined, however, the delay attributes are ignored and the subnet 'fires' until the subgraph completes execution. Transitions and Subnets have the following attributes:

- transition tag, a unique name identifying the transition.
- transition delay, the value is interpreted according to the delay type attribute specification
- transition delay type, the type of delay associated with the transition. Delay specifications can be of the following types:
    - Instant, or zero delay
    - Fixed, where the transition delay value is used
    - Uniform, where any value between zero and the transition delay value is selected with equal probability
    - Exponential, where the delay value is a random variable exponentially distributed with mean equal to the transition delay value
    - Function, where the delay value is obtained by invoking a user defined delay function
- transition delay function, an arbitrary function specified by the user if the transition delay type is function
- transition predicate, a user defined function
- transition procedure, a user defined function
- subnet parameter list, to be applied to the associated subgraph (for subnets only)

In the PCM package, all user defined functions are specified using the C programming language.

5.4 Validation of Simulator

The results of any simulation are meaningless unless the simulator can be validated. Validation of a simulator can be done in several ways. The obvious method for validating the simulator would be to model an existing system, and compare the simulator results with figures obtained from the actual system. In most cases, however, this is not possible, since the modelled system often does not exist yet, or it is not possible to obtain accurate figures from the existing system. An alternate validation method would be to analyze the system using analytical techniques, and to compare the results with the simulation results. The PCM simulator has been validated using both methods.

Results from the simulator are compared with analytical results obtained using the Stochastic Petrinet analyzer [MOL81]. The PCM model is also used to model the multiprocessor memory and bus interference problem and the results obtained are compared with exact (analytical) results published in [HOL87]. Further validation is provided by comparing results from the simulator with actual executions of TDFL programs on a Sequent Balance-21000 system.

5.4.1 Validation against results using Stochastic Petri Nets

Stochastic Petri Nets (SPNs) were introduced earlier in chapter 2. The distinguishing feature of SPNs is that all transitions have exponentially distributed transition firing times. In other words, each transition delay is specified as an exponentially distributed random variable with some mean λ. Since the PCM model permits the specification of exponentially distributed transition delays, it is easy to model systems which can be modelled using SPNs. The only other difference is the transition selection policy. SPNs allow all transitions that are simultaneously enabled to begin firing. The first transition to complete firing is the one selected to fire. This policy is different from that used by the PCM, where each transition has equal probability of being selected. For the comparisons shown below, the PCM simulator was modified to reflect a selection policy equivalent to that used by the SPN model.

The example used in [MOL81] is shown in Figure 5-3. The delay values shown in the figure are the mean (λ) values for each transition. The results show the throughput and the mean number of tokens at each place for different initial markings.

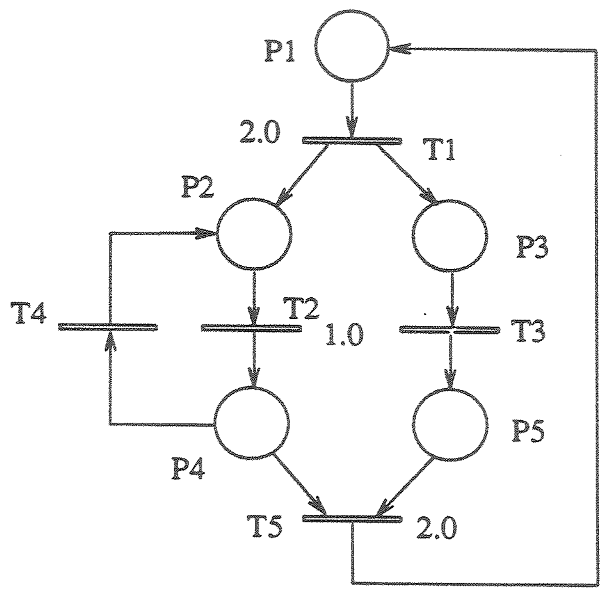


Figure 5-3: SPN Example

The results are shown in Tables 5-1 & 5-2 for different numbers of initial tokens at place P1. It can be seen that the difference is mostly less than one percent, which is acceptable.

5.4.2 Other Validation Results

The PCM simulator has also been validated by two other means. The multiprocessor memory and bus interference problem described in [HOL87] was modelled, and the results compared with previous results. The execution of a parallel algorithm for the Block Triangular Matrix system was modelled using PCM, and the results were validated using actual execution times from an implementation of the algorithm on a parallel computer. The previous section compared results obtained using the PCM model with analytical results obtained using the SPN model. Since the PCM simulator was modified to reflect the same transition selection policy as the SPN model, the actual Petri Net graphs used in both models were identical. In the rest of the validations, this is not the case. These validations are done by actually modelling a system, and comparing the results with those obtained by other means. Having successfully validated the model, further performance predictions are made with changes

Initial Marking Vector = [1,0,0,0,0]				
Place	Throughput (A)	Throughput (S)	# tokens (A)	# tokens (S)
P1	0.2326	0.2340	0.1163	0.1150
P2	0.7209	0.7186	0.7209	0.7249
P3	0.2326	0.2340	0.2326	0.2287
P4	0.7209	0.7186	0.1628	0.1600
P5	0.2326	0.2340	0.6511	0.6562

Initial Marking Vector = [2,0,0,0,0]				
Place	Throughput (A)	Throughput (S)	# tokens (A)	# tokens (S)
P1	0.3342	0.3342	0.1862	0.1857
P2	0.9244	0.9233	1.5873	1.5880
P3	0.3342	0.3342	0.4154	0.3987
P4	0.9244	0.9233	0.2265	0.2263
P5	0.3342	0.3342	1.3980	1.4156

Table 5-1: Comparison of Simulation Results with SPAN

in the system parameters. Since this now involves the actual modelling of parallel computations, presentation of the associated validation results is deferred until later. The multiprocessor memory and bus interference problem is described in the Chapter on modelling architectures (Chapter 6), while the Block Triangular Matrix solution results are presented in Chapter 7.

Initial Marking Vector = [3,0,0,0,0]				
Place	Throughput (A)	Throughput (S)	# tokens (A)	# tokens (S)
P1	0.3745	0.3702	0.2218	0.2154
P2	0.9804	0.9743	2.5320	2.5410
P3	0.3745	0.3702	0.5330	0.5030
P4	0.9804	0.9743	0.2461	0.2436
P5	0.3745	0.3702	2.2450	2.2810

Initial Marking Vector = [4,0,0,0,0]				
Place	Throughput (A)	Throughput (S)	# tokens (A)	# tokens (S)
P1	0.3901	0.3870	0.2382	0.2362
P2	0.9951	0.9908	3.5314	3.5165
P3	0.3901	0.3970	0.5997	0.5825
P4	0.9951	0.9908	0.2504	0.2477
P5	0.3901	0.3870	3.1621	3.2015

Table 5-2: Comparison of Simulation Results with SPAN (cont'd)

Chapter 6

Modelling Parallel Architectures

Petri Nets have been widely used to represent computer architectures ([PET81], [NUT72], [MAR86]). In this chapter, the ability of the PCM model to represent parallel architectures is investigated. In particular, a class of architectures called *shared memory* architectures is studied, and the PCM models created are included, with minor modifications, in the library of architecture submodels specified in Chapter 4. The distinguishing feature of shared memory architectures is that the various processors in the system are all capable of accessing one or more (shared) memory modules. The main difference in the various architectures is the method of interconnection between the processors and the memory modules.

The models constructed in this Chapter are used to study the behavior of the architectures under probabilistically generated loads and access patterns. This leads to a better understanding of the performance of the architecture for a wide range of situations which could occur when the architecture is being utilized. To study the suitability of an architecture for a particular computation, however, the access patterns and load should be generated in a deterministic, rather than probabilistic way. By exploiting the hierarchical nature of the PCM, only minor changes are needed to replace the probabilistic accesses by the standard architecture submodel interface which can respond to access requests generated by an actual computation.

The main results of this Chapter can be stated as follows:

- The various models constructed demonstrate the suitability of the PCM model for representing shared memory architectures.
- The study of the probabilistic behavior of these architectures has two advantages: Results obtained can be used to *validate* the models whenever similar results are available from other studies; new results can be ob-

tained which characterize the behavior of the architecture for configurations that have not been previously studied.

- Due to the hierarchical nature of PCM, these models can be converted to the standard form shown in Chapter 4. Using the standard interface, the behavior of the architecture can be studied for deterministic access patterns generated for some computation. These modified models can then be included in the architecture submodel library.

The rest of this chapter describes two common multiprocessor architectures, and their representation using the PCM model. Section 6.1 illustrates the modelling of a multiple bus system, while section 6.2 describes an interconnection network based architecture. Some of the results obtained for the multiple bus system are compared with previously published results for validation purposes.

In both examples, memory requests are generated by the processors based on a geometrically distributed random variable. This is sufficient in this case, since the aim of the study is to observe the effect of various access patterns on the system. When modelling the actual computation, however, the requests are generated whenever the *process* executing on the processor requires the use of a shared object (which is stored in a shared memory module). The models of the architecture will, therefore, form the lowest level of the computation hierarchy.

In each of the examples given below, the system to be modelled is described first, followed by the equivalent PCM representation, and, finally, performance results for various configurations of the system. Section 6.3 summarizes some observations on the modelling of multiprocessor architectures using Petri Nets.

6.1 Multibus Architectures

Figure 6-1 shows the configuration of a multiprocessor system where processing elements access shared memory modules through a single stage multibus (as opposed to multiple stage networks such as banyan networks). The processors are assumed to have local storage, and only access the shared memory modules occasionally. In the figure, three processors and three memory modules are interconnected using a multibus consisting of three bidirectional buses. In general, if the number of buses is equal to the number of memories, the resulting configuration is equivalent to a crossbar, and the only contention among the processors is for the memory modules. If the number of buses is reduced, however, processors will contend for buses as well as memories.

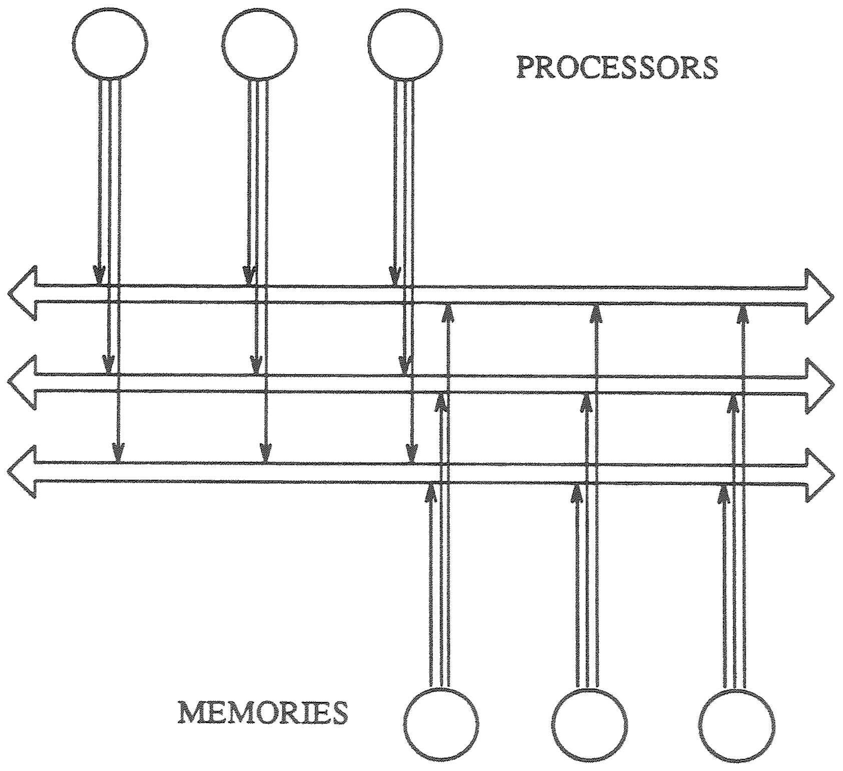


Figure 6-1: A 3x3 Multibus Multiprocessor System

6.1.1 System Description

The system description and terminology used here is derived from an earlier study done by Holliday and Vernon ([HOL87]). The process associated with each processor in Figure 6-1 can be in one of three states: local processing, waiting for a shared memory, or accessing a shared memory. The amount of time between requests (the time spent doing local processing) is called the *interrequest time*. This time is assumed to be a geometric random variable specified by the *memory request* *probability* (MRP). The MRP is the probability that a process will generate a request in the next clock cycle. As the MRP approaches 1, processes spend most of their time accessing or waiting to access the memory modules. A process with an outstanding memory request blocks until it obtains an arbitrary bus and the desired memory module. The time spent actively accessing a memory module is called the *memory access time*, while the distribution of accesses over the memory modules is the *memory access probability*. A uniform memory request probability distribution signifies that a process is equally likely to access any of the memory modules. Non-uniform accesses are usually modelled by specifying a *favorite memory probability* which is the probability that a process selects a special (favorite) memory over the others.

The performance metrics of interest are defined next. *Memory utilization* is the fraction of time that a memory is being accessed by some process. *Processor utilization* is the fraction of time that a processor has its associated process running on it (versus accessing or waiting for a memory). *Processor productivity* is the probability that a typical process is doing useful work (executing on its processor or accessing a memory). Memory utilization summed over all memories is the expected number of busy memory modules or the *effective memory bandwidth*. Processor utilization summed over all processors is called the *processing power*. Processor productivity summed over all processors is called the *speedup*.

This system has been the subject of several analytical and simulation studies to evaluate the effective memory bandwidth and processing power under variations in the configuration. Configuration parameters that are varied are the number of processors and memories, the number of buses, the memory request probability, and the favorite memory probability. A survey of these studies, along with their results, can be found in [HOL87]. The existence of a large set of results provides the means for validating the PCM model. Results obtained using the PCM model are compared with corresponding results from previous sources. Having validated the model, further configurations are set up to obtain new results.

6.1.2 Modelling the Multibus System

The PCM model instance for the multibus multiprocessor system is shown in Figure 6-2. The state variables for the net are shown below:

```
p   :  number of processors
m   :  number of memory modules
mrp : memory request probability
fmp : favorite memory probability
mat : memory access time
steady_state : time at which system is reset
queue[m] : queues containing processor requests
flag[p] : procs flagged when memory access is complete
```

The top level of the net (Figure 6-2(a)) consists of one transition (Reset) and two subnets (Procs and Mems). The Reset transition has a predicate which is true only when the simulation clock value exceeds the value of state variable *steady_state*, and its firing causes all timing information in the net to be erased. Using this special transition, performance statistics are only gathered after the system has reached a steady state. Procs and Mems are subnets modelling the processors and memories respectively. Also shown in the figure are four places which are common to both subnets. Tokens in the *bus* place represent free buses. The initial marking of this place denotes the number of buses in the current configuration. The other three places are used to evaluate the performance metrics described earlier. *mem_bw* initially contains no tokens, and denotes the memory bandwidth, while *speedup* and *proc_power* represent the speedup and processing power respectively, and are both initialized with a number of tokens equal to the number of processors in the configuration.

The processor subnet (Procs) is parameterized by the number of processors. When the subnet is activated, *p* copies of the subgraph are initiated, and tokens placed in each *P_local*. *P_local* denotes that a processor is engaged in local processing. It remains in this state until transition *P_setreq* fires, at which time a request is made to a shared memory. The delay introduced by transition *P_setreq* is a binomial random variable with mean { (1/MRP) - 1 }. The procedure associated with *P_setreq* selects a memory module using the memory access probability, and places a request in the appropriate memory *queue*. The request contains the processor id and the length of data

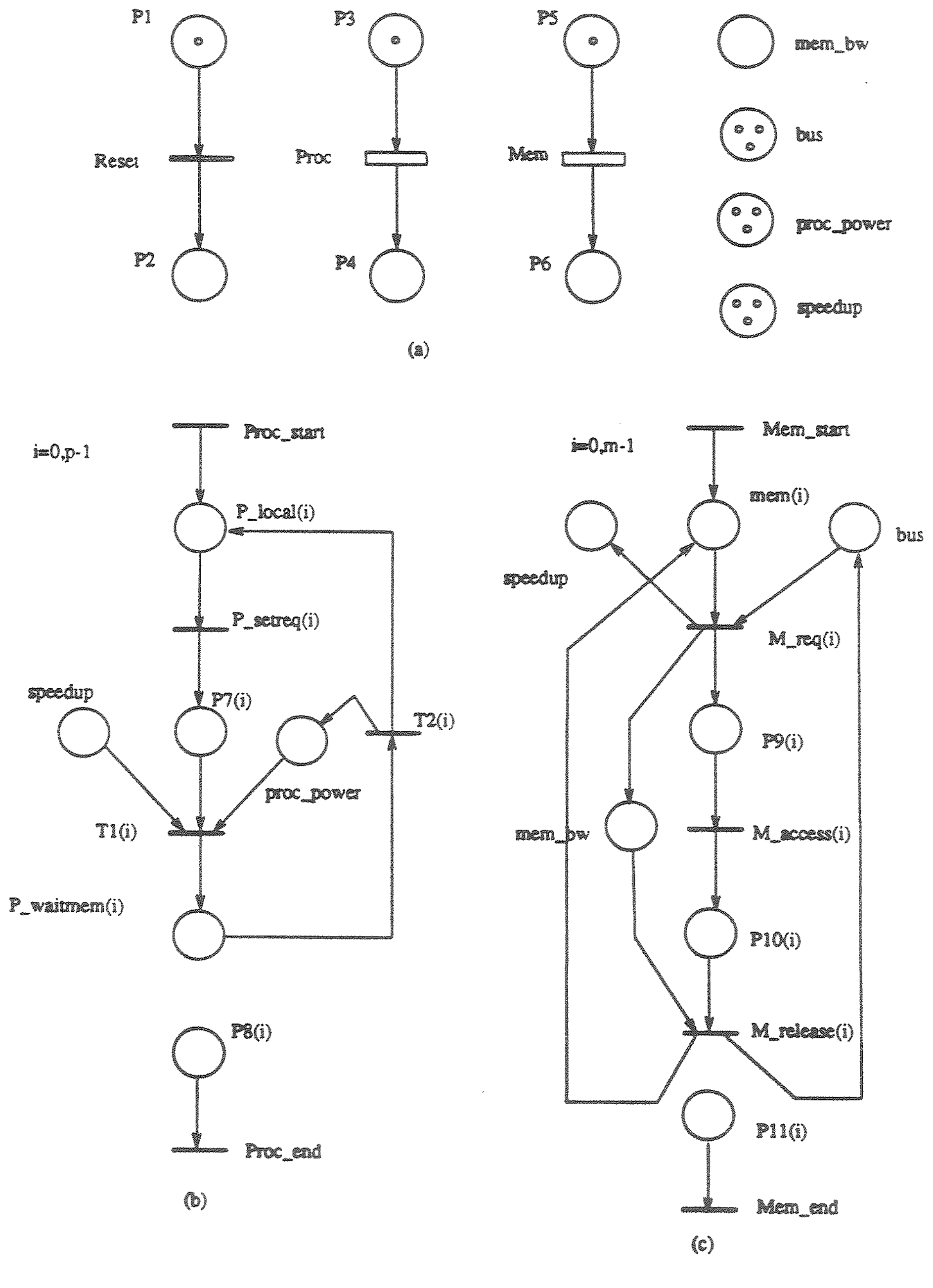


Figure 6-2: PCM Model for Multibus System

required. T1 is an instantaneous transition which removes tokens from the speedup and proc_power places. The predicate at transition T2 fires only when the flag corresponding to the processor has been set, denoting the completion of access. since the processor is now waiting to access the selected memory.

The memory subnet (Mems) is parameterized by the number of memory modules (*m*). M_req fires only when a request appears in the memory queue, and a bus is available (i.e. there is a token in place *bus*). A token is placed in place *mem_bw* for the duration of the memory access. Transition *Mem_access* has a delay proportional to the number of data bytes requested by the processor for the current access. Finally, *M_release* replaces tokens in the *mem* and *bus* places, and sets the flag for the waiting processor to signal completion of the access.

6.1.3 Modelling Results

Results obtained for various configurations of the multibus system are presented here.

Processors	Memories	GTPN	PCM
2	2	1.5000	1.4960
4	4	2.6210	2.5960
6	6	3.7809	3.7280
8	8	4.9471	4.9070
10	10	6.1150	6.0120
12	12	7.2835	7.2550
14	14	8.4527	8.3700
16	16	9.6225	9.6380

Table 6-1: Effective Memory Bandwidth Crossbar MRP = 1

The first four configurations assume constant memory access times of 1 cycle, and uniform memory accesses (i.e. no favorite memories). The results obtained from these configurations are compared with the results from [HOL87] using the GTPN model mentioned in Chapter 2. Table 6-1 shows the effective memory bandwidth for crossbars (ranging from 2 to 16 processors) with a memory request probability of one. The GTPN and PCM results for this configuration differ by less than one percent. Table 6-2 shows the effective memory bandwidth for a 16 processor/16 memory system for different numbers of buses. Again, the MRP is assumed to be one, implying that processors are never spending time on local processing. An interesting point here is that upto 6 buses can be removed from the crossbar configuration with minimal degradation in the overall performance.

Buses	GTPN	PCM
8	7.977	7.937
9	8.825	8.782
10	9.357	9.278
11	9.566	9.608
16	9.623	9.638

Table 6-2: Effective Memory Bandwidth 16x16 Multibus MRP = 1

Buses	GTPN	PCM
1	1.000	1.000
2	2.000	1.996
3	2.898	2.872
4	3.352	3.336
5	3.458	3.457
6	3.469	3.447
7	3.469	3.466
8	3.469	3.514

Table 6-3: Effective Memory Bandwidth 8x8 Multibus MRP = 0.5

The next configuration (Table 6-3) is an 8 processor/8 memory system with MRP = 0.5, studied under variations in the number of buses. In this case, if the number of buses is less than 3, the buses are the bottleneck. On the other hand, having more than 4 buses does not result in significant improvement in performance. Finally, Table 6-4 shows the effective memory bandwidth for different values of the MRP. The table shows the mean Inter Request Time (IRT), where the mean IRT is (1/MRP - 1).

Buses	Mean IRT	GTPN	PCM
1	8	0.9998	1.0000
	16	0.8588	0.8516
	32	0.4745	0.4814
2	4	1.9983	1.9996
	8	1.6560	1.6782
	16	0.9365	0.9274
	32	0.4793	0.4766

Table 6-4: Effective Memory Bandwidth 16x16 Multibus

In each of the configurations described above, the results obtained using the PCM simulator were within one percent of the exact results calculated using the GTPN model.

The next configuration studied is the effect of varying the MRP on the speedup. Figure 6-3 shows speedup as a function of the MRP and the number of buses for an 8 Processor/8 Memory system. As expected, the speedup decreases as the MRP is increased, since an increased MRP indicates that processors make more frequent requests to the memories thereby causing greater contention for the buses and memories. The speedup curves for 4 and 8 buses reiterate the earlier observation that using half the number of buses does not degrade the system performance significantly. In fact, further buses can be removed from the configuration if the MRP is low

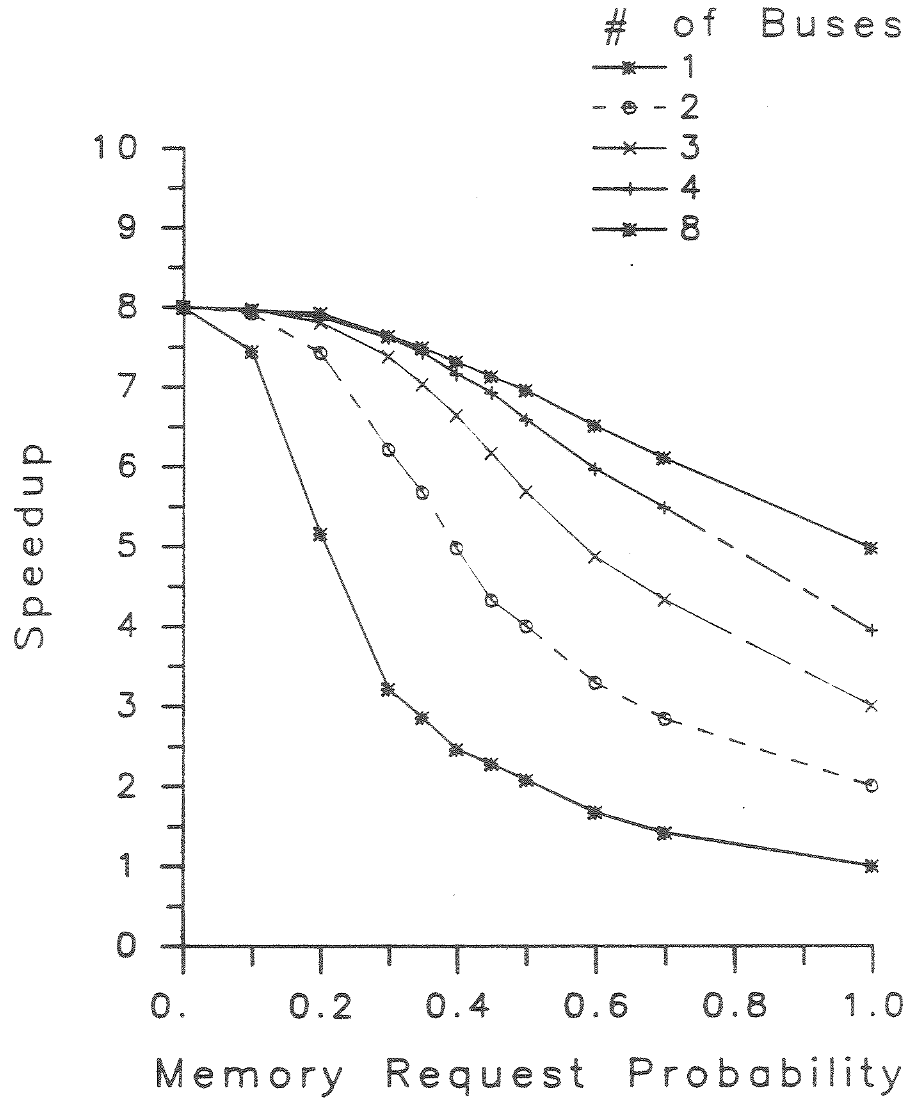


Figure 6-3: Speedup vs MRP for an 8 Processor/8 Memory System

enough. The sharp drops in the curves for smaller numbers of buses indicate, however, that these configurations have a critical value of MRP beyond which the system performance degrades rapidly.

In the previous examples, all memories had equal probabilities of being

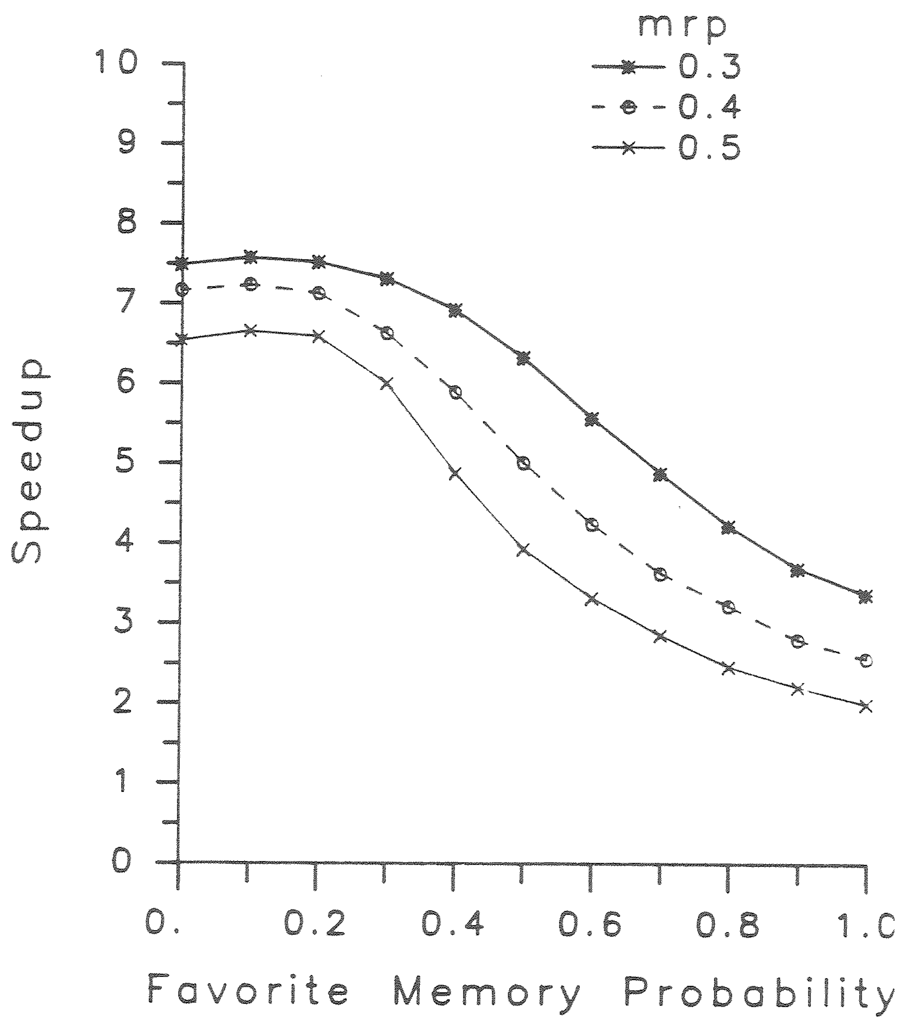


Figure 6-4: Speedup vs Favorite Memory Probability for an 8 Processor/8 Memory System

selected by a processor. In the next case, the probability of selecting a certain memory module is varied. The *favorite memory probability* (FMP) is the probability that a processor selects a certain "favorite" memory for its next access. All the other memories have an equal probability of being selected (this probability is [1 - FMP]/[m-1]). As the probability of selecting this "favorite" memory is increased, there will be greater contention by all processors for it. Figure 6-4 shows the variation of speedup with the favorite memory probability for an 8 processor/8 memory/4 bus system. A FMP of zero implies that the favorite memory is never selected, thus resulting in a system containing effectively only 7 memories. Until the FMP reaches 0.125, the favorite memory has a poorer chance of being selected than the other memory modules, and the speedup remains unchanged. Even as the FMP is increased beyond 0.125, the speedup is not significantly affected up to a certain point. The bottleneck as the FMP is increased is the contention by all processors for the favorite memory. As expected, this bottleneck is more pronounced for higher values of MRP, since processors are more likely to make memory requests.

Figure 6-5 shows the effect of varying the message lengths on the speedup for a 8 processor/8 memory/4 bus system. For each message length, the speedup is plotted as a function of the MRP. The effect of having a larger message length is that processors hold buses and memory modules for several cycles instead of just one. Regardless of the message size, the system always saturates at a speedup value of 3.95. This is due to the fact that there are only four buses in the system, and after a certain level of traffic in the system, on the average, four processors are engaged in useful work, while others wait for buses to become available. This traffic level is reached more rapidly for larger message lengths. The speedup obtained with a message length of 8 indicates that this configuration is very sensitive to message lengths, and a large message length coupled with even a moderate MRP of about 0.25 will result in poor system performance.

6.2 Interconnection Network Architectures

Many of the recently designed multiprocessor systems fall into the category of Interconnection Network (ICN) architectures ([PFI85], [SEJ80], [CRO85], [SIE81]). Figure 6-6 shows the topology of an ICN Architecture. The main characteristic of these architectures is the switching network that interconnects the processors and memories. One of the features of these networks is the existence of a unique path between each processor memory pair. While this is cost effective in terms of number of switch nodes required (compared to a crossbar), there is now the possibility of processors getting *blocked* when they attempt to access a memory. This blocking results from the fact that processors now contend with each other for switch nodes that lie in their path to a selected memory. To be able to capture this contention for

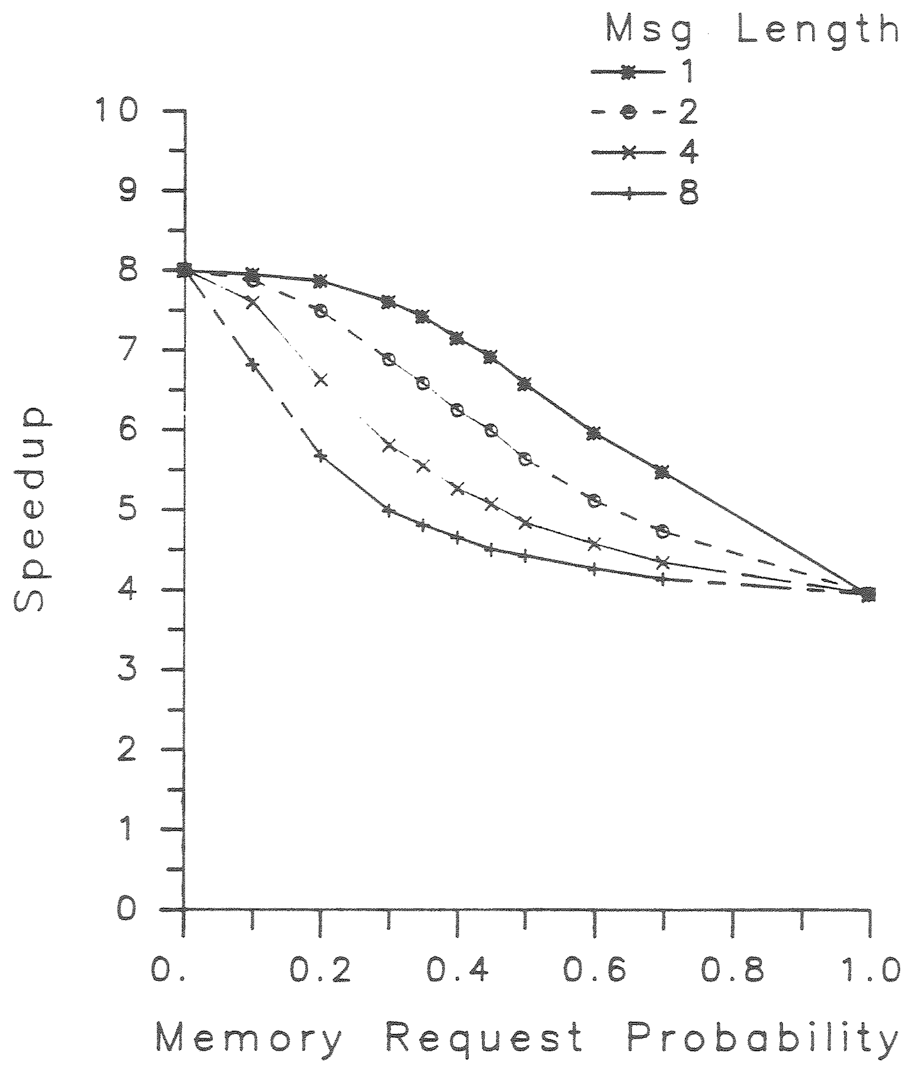


Figure 6-5: Effect of Message Lengths on Speedup

switches, it is necessary to model each individual switch and the interconnection between switches. The hierarchical nature of the PCM model allows the representation of high level processes executing on processors, as well as low level switch behavior.

The system under consideration is an 8 processor/8 memory system as shown

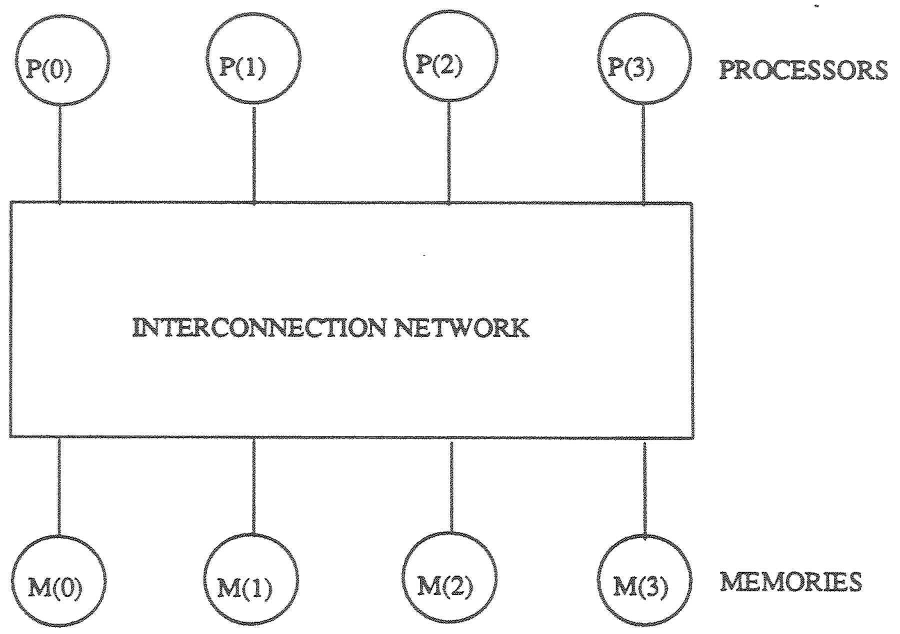


Figure 6-6: Interconnection Network Architectures

in Figure 6-7. The interconnection (or switching) network used here is a 2x2 Banyan network ([GOK73]) similar to that used in the TRAC Computer ([SEJ80]). Each processor in the system is an independent unit with local storage for data and instructions. The memory modules shown in the figure are used to share data between cooperating processes. Two different access methods are supported by the system. When a processor accesses a memory in the *packet switched* mode, the memory sends out streams of packets which are then routed through the appropriate switches to the requesting processor. In this mode, a packet takes one clock cycle to move between adjacent switches. A packet consists of a tag word and a data word. The tag word is the address (or id) of the destination processor. Each switch level interprets a different bit of the tag, and depending on its value, routes the packet either on its left or right link. An alternate method of memory access supported by the system is *circuit switching*, where a direct path is established between the communicating processor and memory. The link is maintained until the processor has completed its access. The establishment of circuits in the network is handled by a distributed architectural

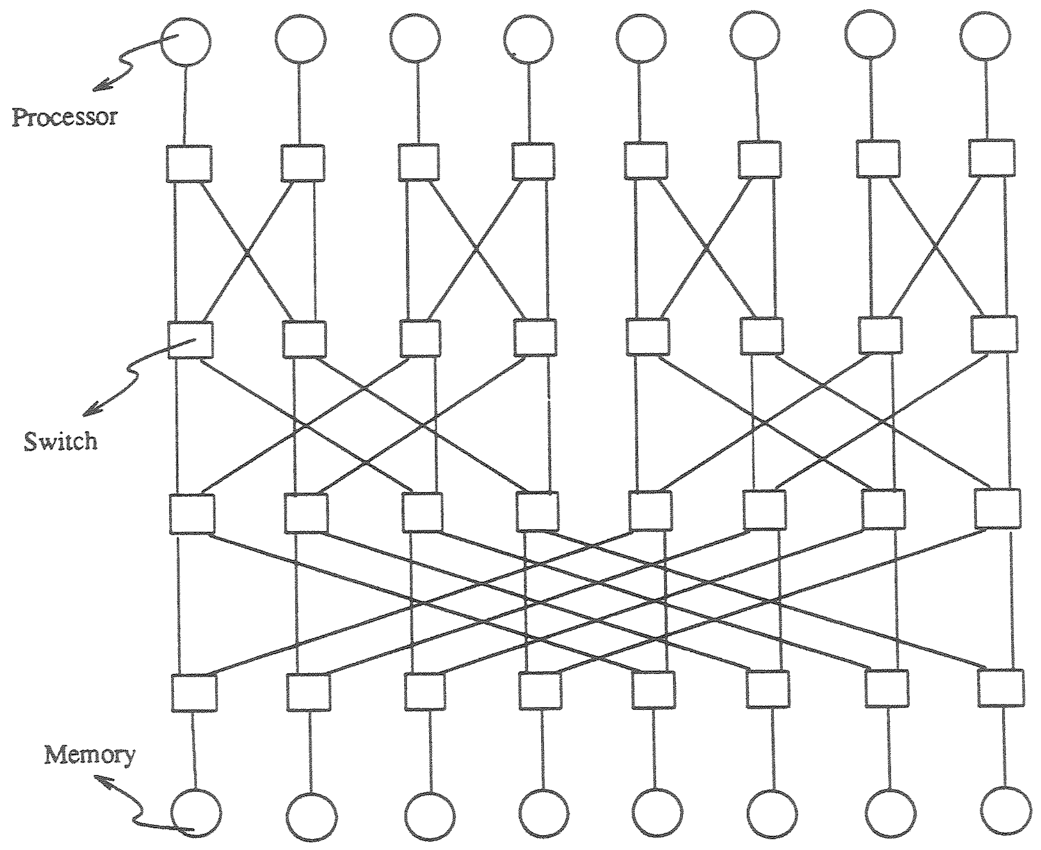


Figure 6-7: 8x8 Banyan Interconnection Network

resource called the Network Controller [JEN82]. The results of a study of the effectiveness of these two access methods for different MRPs and message lengths is reported in [ADI87a]. The description given here concentrates mainly on the modelling aspects of the study, to show the utility of the model for representing complex computer architectures.

6.2.1 System Description

The system to be modelled can be divided into four functional parts: Processors, Memories, Switches and Network Controller. A description of each component is given next. The switch behavior for both the packet and circuit switched modes goes down to the gate level in detail in order to model switch contention accurately.

6.2.1.1. Processors

A processor can be in one of four states: *local*, *request*, *wait* or *receive*. Each processor operates as follows:

- The processor begins in the *local* state, where it executes instructions from its local memory, and remains there until it is ready to begin accessing shared data. The period of time it remains in this state is selected randomly, and has a binomial distribution based on the *loading factor* of the network. The loading factor is the probability that a processor in the local state would make a request on the next cycle, and is equivalent to the MRP described in the previous section.

- Upon entering the *request* phase, a processor sends a request to a randomly selected memory module. Each memory module has an equal probability of being selected. In the packet switched mode, the request is sent directly to the selected memory module, while in the circuit switched mode, the request is first sent to the network controller to set up a circuit between the processor and memory. It is assumed that the number of data words to be accessed by all processors is a constant, and the effect of varying this *message size* on the network is studied. The processor keeps a record of this size until it receives the data. The processor spends a fixed amount of time in this state.

- After the processor has sent its request, it moves to the *wait* state where it waits for the response from the memory (or the network controller). The time spent by a processor in this state depends on, a) whether the memory is available, and b) the amount of time it takes for the data to reach the processor.

- When the first response packet reaches the processor, it moves into the *receive* state. In the case of circuit switched data transfer, it moves to this state when the required circuit is established. It remains in this state until the entire message has been received, at which point it returns to the *local* state.

6.2.1.2. Memories

Each memory is identical, and is in one of two states: *idle* or *sending*. Each memory is assumed to have a queue of processor requests. The memory modules behave as follows:

- The module remains *idle* until it gets request from a processor.
- Having received the request, the memory moves to the *sending* state, where it begins to send a message to the processor. When the entire message has been sent, the memory returns to the *idle* state. If there are other processor requests, the memory does not stay in the idle state, but moves back to the sending state.

6.2.1.3. Interconnection Network

The network consists of switches interconnected to form a regular (2,2) banyan (Fig. 6-7). Since only the messages sent by the memories are modelled, all data movement flows towards the processors. The switches are labeled [i,j], where i is the *level* and j the *offset* within the level. The network is modelled differently for packet and circuit switched modes of communication. For circuit switching, once a circuit is established within the network, the switches themselves provide passive data paths. The packet switched model is described next.

Packet Mode Switch

Each switch is modelled as shown in Figure 6-8. The switch consists of a buffer of fixed size which models the packet storage capacity of the switch, and the four interconnection links connecting the switch to its neighbors. Associated with each link are the variables *req* and *ack* shown in the figure. These two variables represent signals used in the network. *req* is set when a switch requests permission to send a packet along the link, while *ack* is set when permission to send on the link has been granted. In the figure, these variables are prefixed by l_ and r_ (left and right) and the incoming and outgoing links are distinguished by the "prime". Each switch is in one of three states: *request*, *transmit & acknowledge* or *receive*. Each of these states is described next:

- During the *request* phase, a switch requests permission to send a packet to its "parent". It executes the following steps:

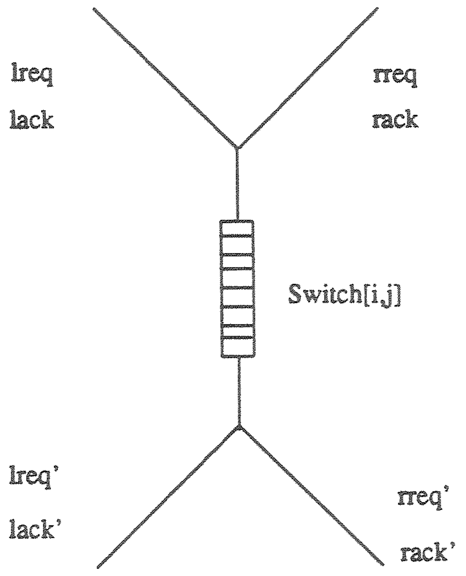


Figure 6-8: A Packet Mode Switch

```
if (not empty(buffer))
  if (route(front[buffer]) is left link)
     set l_req
  else
     set r_req
```

where route is a function of the tag word of the packet at the head of the buffer queue and decides which link it should be sent to. The corresponding req is set.

- After completing the request phase, the switch goes into the *transmit & acknowledge* phase. Here it checks if permission to send the packet was granted. If it was granted, it puts the head packet from the buffer queue on the appropriate link determined by its tag.

```
if (l_ack or r_ack)  {
   if (l_ack)
     reset l_ack;
     place front[buffer] in l_pkt;
```

```
      else
        reset r_ack;
        place front[buffer] in r_pkt;
     }
```

If neither l_ack nor r_ack are set, it implies that either no request was made, or permission to transmit a packet was not granted; hence nothing is done. After this the switch enters the acknowledge part of the phase:

```
  if (not full(buffer))  {
    if (l_req' and not r_req')
      set l_ack'; reset l_req';
    else if (not l_req' and r_req')
      set r_ack'; reset r_req';
    else if (l_req' and r_req')  {
      reset l_req'; reset r_req';
      if (priority(switch) is left)
        set l_ack';
      else
        set r_ack';
      change priority;
     }
    else                (* No buffer space *)
      reset l_req'; reset r_req';
  }
```

If no buffer space is available, the requests are turned down. If only one request was made, it is granted. If both children request, the switch uses a function "priority" to break the tie.

- The last state is the *receive* state, where the switch reads a packet into its buffer.

```
  if (l_ack' was set)
    store l_pkt' in back[buffer];
  else if (r_ack' was set)
    store r_pkt' in back[buffer];
  else          (* No packets to be received *)
    do nothing;
```

After completing this, the switch reenters the *request* state.

Each switch goes through an entire cycle of states in exactly one clock cycle.

The model of the switch is as shown in Figure 6-9. As before, the baseward links are suffixed with a "prime" and the two links on each side are prefixed l_ and r_. With each link are associated four variables: *Rq* (request), *Can* (cancel), *Gt* (grant) and *Busy*. The variables *Rq* and *Can* are mutually exclusive in time; the former is used to set up the circuits while the latter is used to disable them. The switch is in one of the three states: *reset*, *propagate* or *set status*.

- The switch begins in *reset* state. If any of the Rq, Can or Gt variables are set, it moves to the propagate state.

- In *propagate* state, it transmits the values of variables from its apexward end to baseward end, or vice versa:

```
if (l_can' or r_can')
   if (l_busy) set l_can;
   else set r_can;
else if (not (l_busy' or r_busy'))
     if (l_req' or r_rq')
        set l_rq; set r_rq;
     else    (* One of grants is set *)
        set l_gt'; set r_gt';
else      (* The switch is busy *)
   do nothing;
```

  If neither *l_busy*' nor *r_busy*' are true, it indicates that the switch is not busy, and can propagate the *rq* and/or *gt* signals. From this state, the switch moves to the state where it either activate or disables circuits.

- In the *set status* state, each switch evaluates the status of its baseward links, and sets or resets **busy**:

```
if (l_can') reset l_busy';
else if (r_can') reset r_busy';
else if (l_gt' and l_rq') set l_busy';
else if (r_gt' and r_rq') set r_busy';
else do nothing;
```

The switch then reenters the *reset* phase, where it resets all *rq* and *gt* signals that were set in the previous cycle. The switch goes through the three states in exactly one clock cycle. It is assumed that the Network Controller attempts to set up or reset a circuit only once every cycle. The success of a set-up attempt is detected by the setting of *busy* on links interfacing with the processor and memory pair.

Circuit Mode Switch

As mentioned before, in the circuit switched mode, the switches play a passive role in transfer of data. Their important function however, is to allow or block activation of potential circuit depending on their current state. No two independent circuits are allowed to share a switch.

The banyan network provides a unique path between a pair of apex and base nodes. Every apex node is at the root of a tree which connects it to all base nodes. Similarly, each base node is at the root of an inverted tree which connects it to all apex nodes. The unique path between an apex node and a base node is found by topologically intersecting the trees rooted at the two nodes. This is done by propagating two signals in opposite directions in the network. These signals originate at the base and apex nodes to be connected, and are propagated along the two trees rooted at the base and apex nodes. Only nodes lying in both trees receive both the signals, and these nodes form the only path between the selected apex and base.

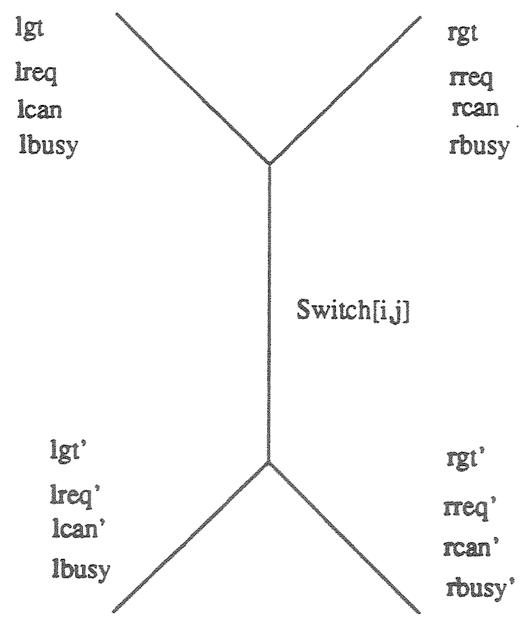


Figure 6-9: A Circuit Mode Switch Node

6.2.2 PCM Representation of the ICN

In this section, representations for the ICN system in the extended Petri net model are given. The parameterization features allow replications of subnets, and it only becomes necessary to specify one instance each of the processor, memory and switch nodes. Also, by changing a single size parameter in the state variable set, the model could be changed from an 8x8 system to a 16x16 system.

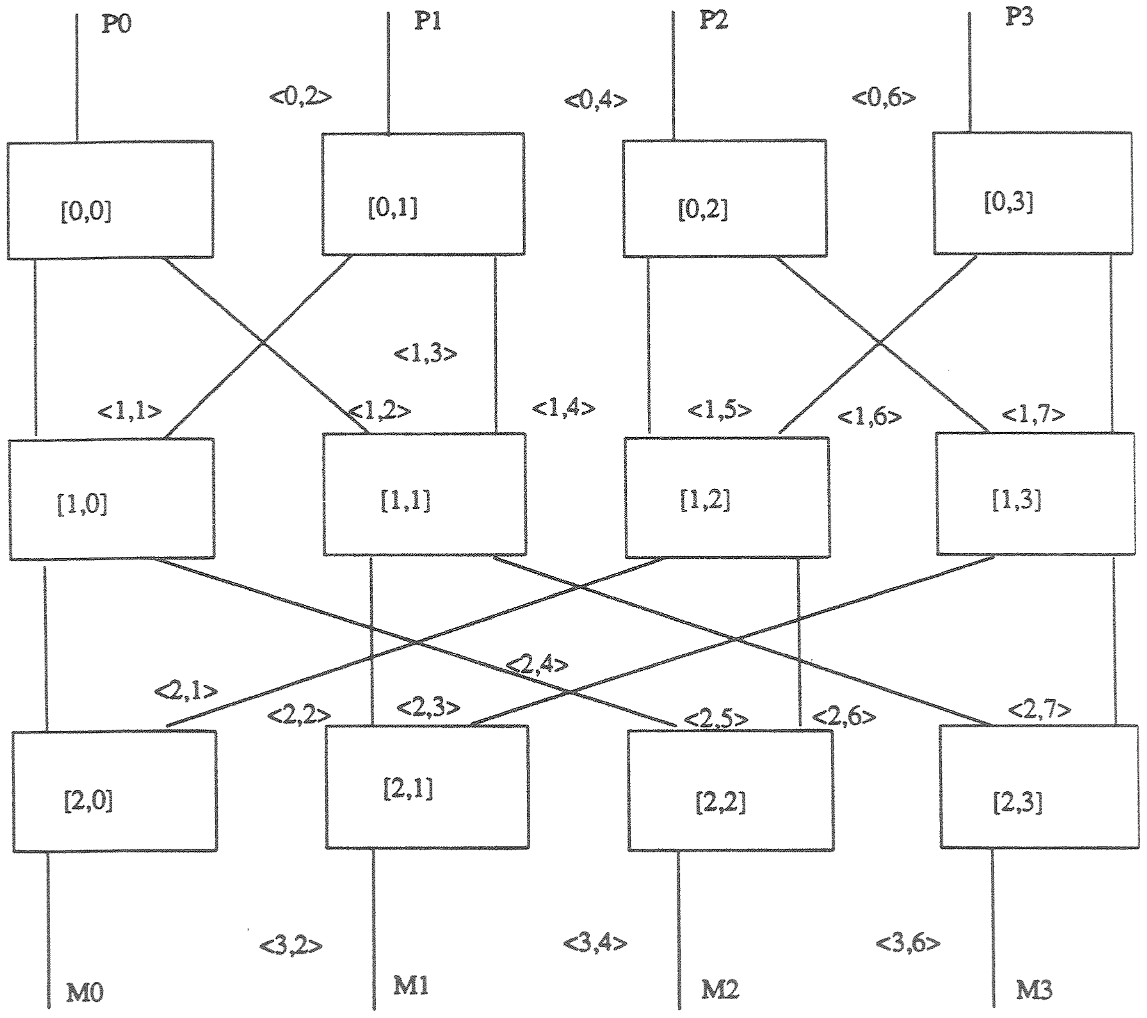


Figure 6-10: Labelling the ICN Switches

The link indices are set up so that the appropriate switches in the banyan network share a link. The switch and link labels are shown in Figure 6-10. The lower left and right indices for the bottom switch level is different from the other levels. If the system used an interconnection network other than the banyan, only the link indices would need to be changed to model the new network. In the rest of this section, the links and their variables will be referred to as l_, r_ etc., as before. The equivalent indexed representation for link variable *req* is shown below:

for switch levels 0 to log(n)-1:

l_req(i,j)	req(i, 2j)
r_req(i,j)	req(i, 2j+1)
l_req'(i,j)	$req(i+1, 2j - ((j/2^i)mod2)*(2^{i+1}-1))$
r_req'(i,j)	$req(i+1, 2j + 2^{i+1} - ((j/2^i)mod2)*(2^{i+1}-1))$

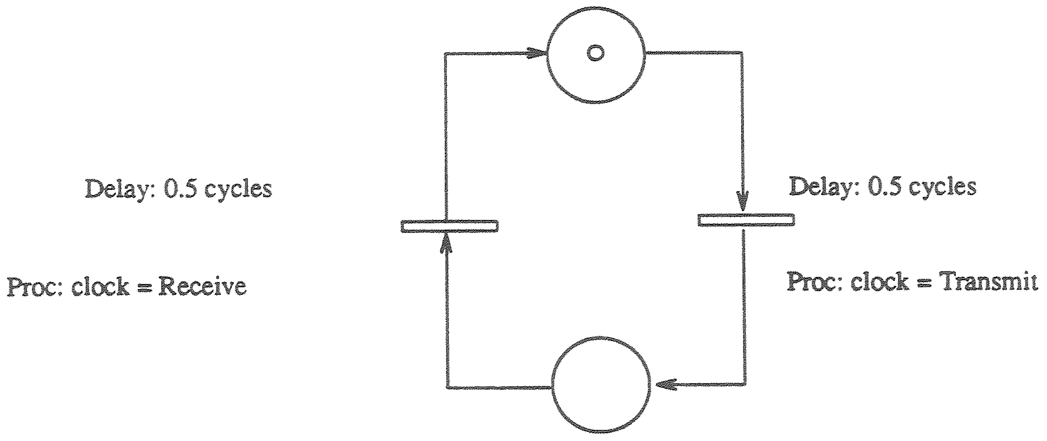


Figure 6-11: Implementation of a Global Clock

Since the architecture under consideration is completely synchronous, there is a requirement in the model to synchronize the firing of various transitions in the Petri Net model. This synchronization is enforced by implementing a "global clock" as shown in Figure 6-11. The clock shown in the figure is used for the packet switching case, where only two clock phases need to be distinguished (the transmit and receive switch phases). Both transitions have half cycle delays, and they modify a state variable which is set to either transmit or receive. When clock synchronization is required, this variable is used in transition predicates to introduce the appropriate delay.

6.2.2.1. Model for the Packet Switched ICN

The packet switched ICN consists of the processor, memory and switch nodes described earlier. The global clock consists of two phases: transmit and receive. Since packets carry routing information, they must be modelled explicitly as state variables. Also, each memory has a buffer for storing processor requests, and each switch has a buffer to hold intermediate packets between clock cycles. When a processor makes a request, it enqueues its request in the buffer of the selected memory. The memory checks this buffer whenever it is idle.

The set of state variables required for the packet switched model is given below:

```
n      : the number of processors
sw     : the number of switch levels
msg_len : length of each message
queue[sw][n] : one queue for each switch node
priority[sw][n] : priority for each switch
link_pkt[sw+1][2*n] : packet buffer for each link
mem_buf[n] : memory queue containing proc id & msg_len
clk_phase : clock phase: either transmit or receive
```

The switch node

Figure 6-12 shows the Petri net equivalent for each switch node. Associated with each switch node is a buffer to store messages in transit, which is represented as a state variable. The link signals req and ack are modelled by four places (req, nreq, ack and nack) to detect the presence as well as absence of the signal. Each message in the buffer contains the destination processor's id. When the subnet becomes active, T0 fires, and the net is replicated as specified by the ranges of *i* and *j*. Transition T1's predicate is true if its message buffer is not full (implying that the switch is ready to accept another packet). On the other hand, T6's predicate is true if the buffer is full. Depending on the state of the message buffer, either T1 or T6 will fire. T2 and T3 are represent the routing choice made at the switch. Depending on the destination proces-

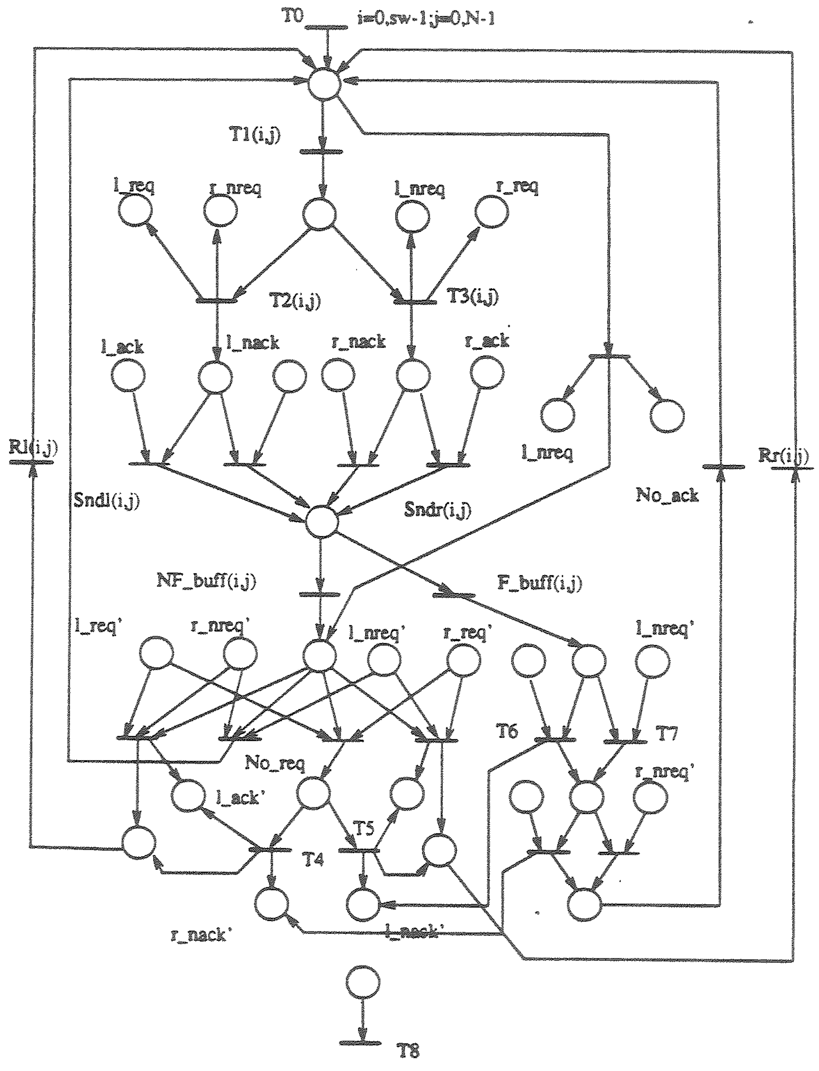


Figure 6-12: Packet Switched Switch Model

sor id of the message to be sent, the predicate of either T2 or T3 becomes true, and the appropriate request is placed on the link, following which, the switch awaits either an ack or a nack. Sndl (Sndr) can fire only after it receives a l_ack (r_ack) on the link. Upon firing, Sndl places a packet on the link. All transitions other than Rl, Rr, No_req and No_ack can fire during the transmit phase.

If the switch buffer is not full, NF_buff fires, and the switch can ack a req it receives. If no requests were made (l_nreq' and r_nreq'), No_req fires during the receive phase. If only one request is made, (l_req' and r_nreq'; l_nreq' and r_req') the corresponding ack is set. If both reqs are set, transitions T4 and T5 compete to decide which request to ack. A state variable is maintained to specify the priority (either left or right). The predicates of T4 and T5 are set up so that they check this variable, and only one of them is true. Also, upon firing, they execute a procedure to flip the priority. Finally, Rl and Rr fire only in the receive phase, and they accept a message from the link and place it in the message buffer. F_buff fires if the switch buffer is full and cannot accept another packet. T6, T7, etc. are used to send nack signals to any requests made by neighboring switch nodes. No_ack then fires during the receive phase.

Transition T5 never becomes enabled (since its input place is disconnected from the rest of the subnet), which ensures that the subnet remains active at all times.

The Processor Node

The processor node is shown in Figure 6-13. When the subnet becomes active, T0 fires, and a token is placed in P_local. Setreq has a delay time which is selected randomly from a binomial distribution with mean equal to the loading factor, a state variable. The procedure associated with Setreq selects a memory module at random, and upon firing, a request is placed in the buffer of that memory module. The processor now waits for a response from the memory module. This waiting time is represented by the amount of time a token remains in the place First_pkt. Wait_msg fires only when it gets a request for transmission of a packet from the switch below it. Setreq also sets a state variable with the message size which is decremented by Rec_pkt each time it fires. Msg_done can fire only when this state variable reaches zero, signifying completion of reception of the message. The processor then returns to the local processing state.

The additional place Full_msg has been introduced only for evaluating the time taken to receive the entire message. The other transitions T1 and T2 simply remove nreqs sent to the processor from the switch. Again, T3 never fires, and the subnet remains active forever.

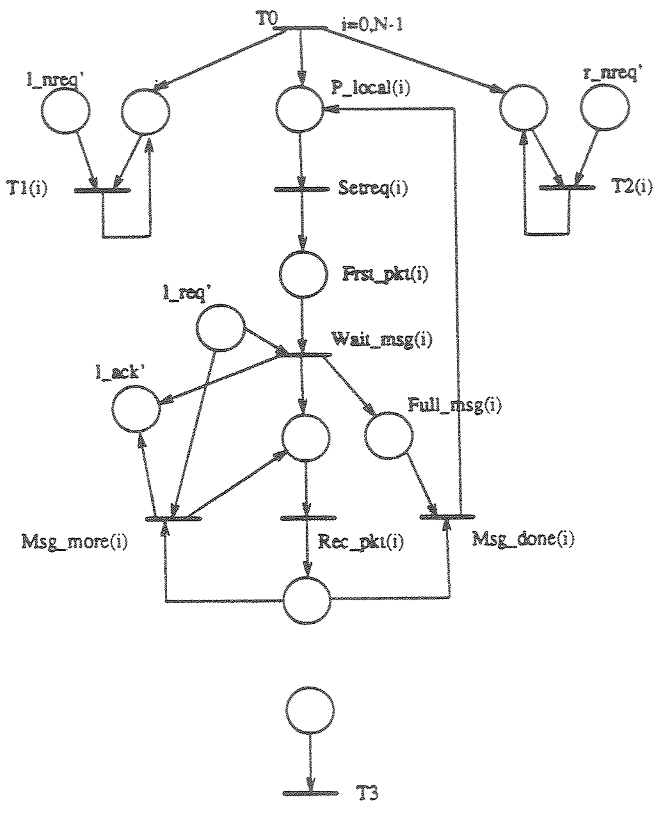


Figure 6-13: Packet Switched Processor Model

The Memory Node

A token in place Idle (Figure 6-14) implies that the memory is lying idle, awaiting a processor request. The predicate for P_req checks the buffer associated with this memory, and returns true only when it is not empty. The procedure for P_req dequeues the request from the buffer, and sets the number of packets to be sent. T3 simply sends a request to the switch above it. If an ack is received, M_send fires, placing a token on the link connecting the memory to the switch. M_send also decrements the number of packets to be sent. While this number is not zero, M_more fires, and the memory continues to send packets. If a nack is received, T4 fires, and the memory retries the send. When the entire message is sent, M_done fires, and the memory returns to the idle state.

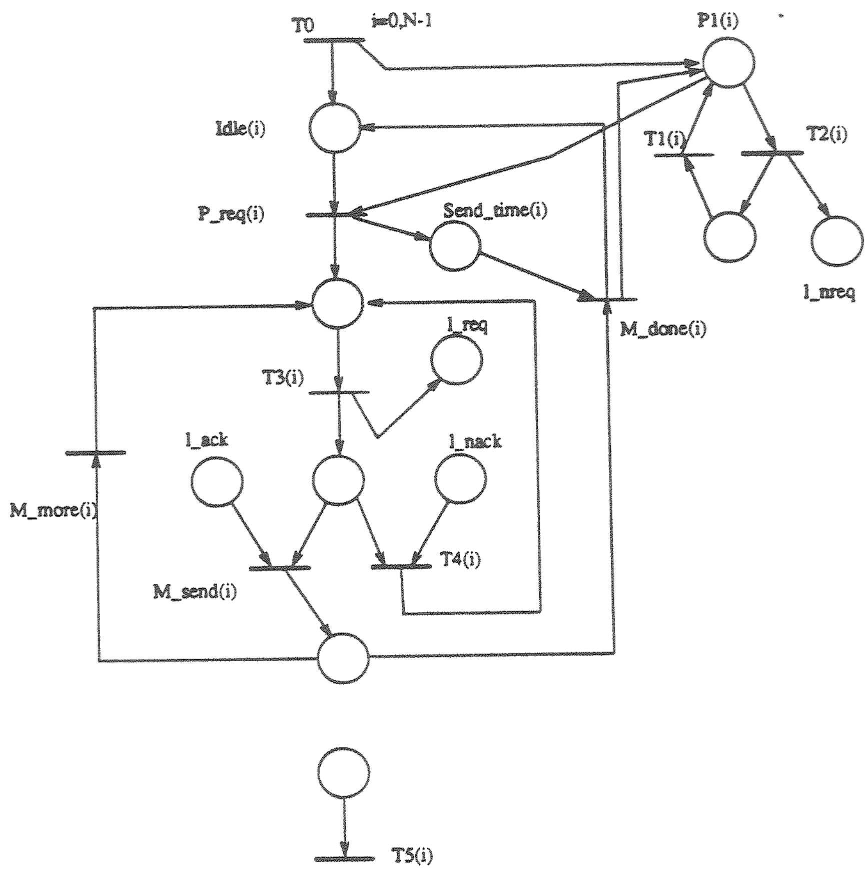


Figure 6-14: Packet Switched Memory Model

The small loop consisting of transitions T1 and T2 is used to generate nreqs once every cycle when the memory is idle. The special place Send_time captures the total time taken by the memory to send the entire message.

6.2.2.2. Model for the Circuit Switched ICN

The circuit switched model has an additional entity to be modelled: the network controller. All requests from the processors are sent to the network controller, which queues them and tries to satisfy each request in FIFO fashion. If a request cannot be satisfied, the network controller requeues the request in its buffer, and proceeds with the next request. Each request takes exactly one cycle to be resolved. The two types of requests allowed are *setup* and *cancel*. The setup request is made by a processor when it wants a circuit to be set up to some selected memory module. After the ensuing message transfer, the processor issues a cancel request to break the circuit. Both types of requests compete for the network controller's attention (i.e. they are treated with equal priority).

The clock implementation here has three cycles: propagate, set and reset, as described earlier. Again, the network controller buffer is modelled using the state variables.

Switch Node

The model for a circuit switched switch node is shown in Figure 6-15. Again, the link variable *busy* is modelled using two places (busy and nbusy). The switch node consists of two independent components. The upper part represents the function when a *cancel* request is being satisfied, while the lower part is used during a *setup* request to check if the circuit can be set up. If the switch receives a lcan' signal, T4 fires, and resets lbusy'. The can signal is then propagated to either the upper left switch, or the upper right switch, depending on which link is busy.

The second part of the model handles the setting up of circuits using the req and gt signals. During the propagate phase, if any req or gt signal is received, the switch propagates the signal unless it is busy. Transitions T6 and T7 fire if the switch is not busy (rnbusy' and lnbusy' are true), and a req arrives either either link (lreq' or rreq'). Transition T_loop fires during the set phase, and ensures that the signal is propagated exactly once. During the set phase, T12 fires if both lgt' and rgt' contain tokens. Firing T12 causes lbusy' to be set, indicating that the link (switch) is now busy. The gt signal is treated in the same manner, except that it is propagated from the upper to lower links.

During the reset phase, transitions T10, T11, etc. fire and reset any signals that were set in the propagate phase.

Processor Node

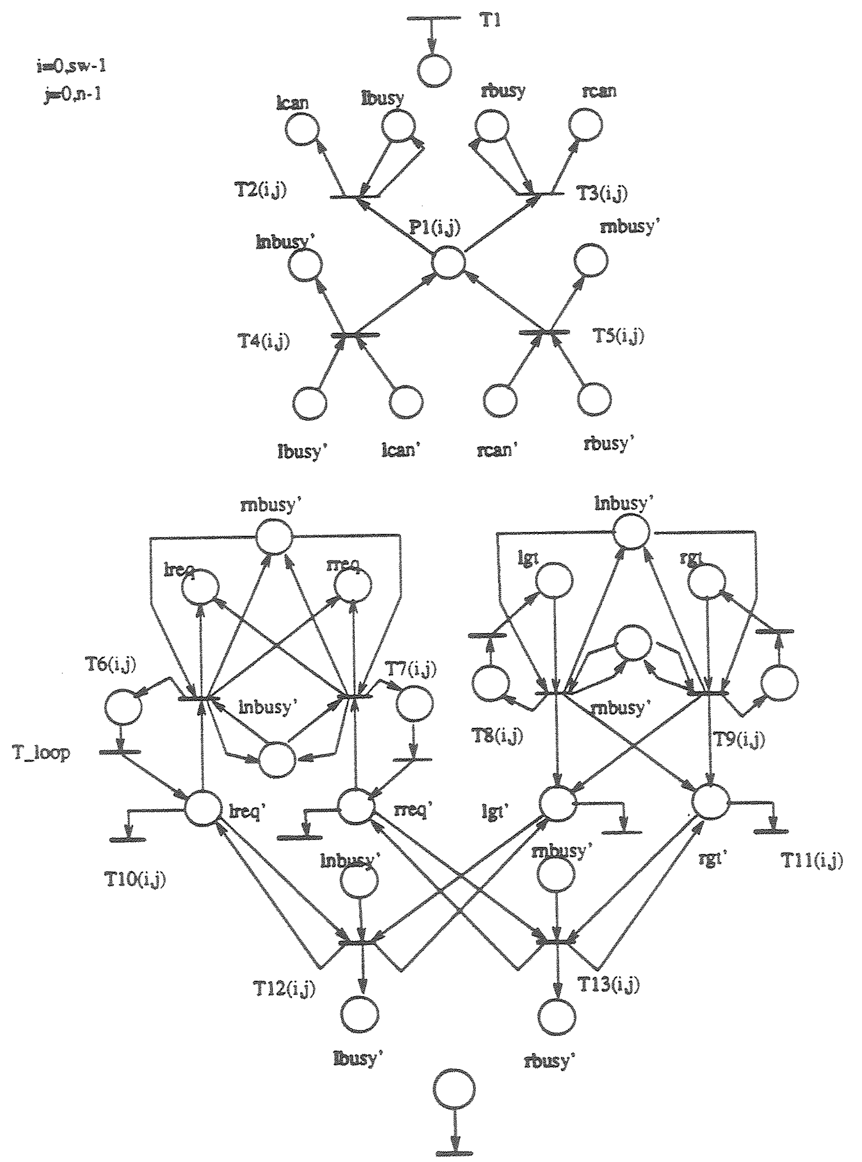


Figure 6-15: Circuit Switched Switch Model

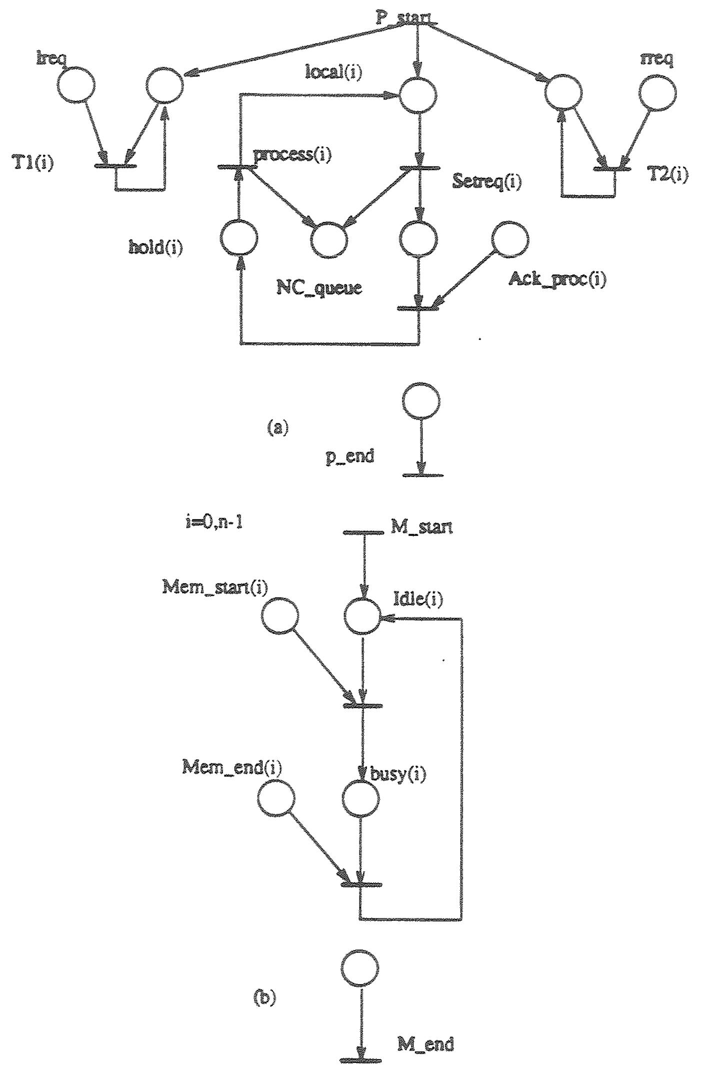


Figure 6-16: Circuit Switched Processor and Memory Models

Each processor node is modelled as shown in Figure 6-16(a). When P_start fires, it places tokens in the *local* places of each processor. Transition setreq models the time spent in local processing. It has a delay specified as a random variable with a binomial distribution with the probability of success in the next cycle being specified by the loading factor. Upon firing, this transition selects a memory, and places its *setup* request in the network controller's buffer. The processor then awaits an acknowledgement from the network controller indicating successful setup of the circuit. When the network controller places a token in place Ack_proc, the processor shifts to the receive phase. The delay introduced at transition process is the message size (since each word takes one cycle for transmission). When transition process completes firing, a *cancel* request is made to the network controller. Transitions T1 and T2 are used to reset any req signals during the req phase.

Memory Node

The only function of the memory model is to capture idle and busy times for each memory module. The model (Figure 6-16(b)) shows that the memory awaits a token in Mem_start to become busy, and Mem_end to become idle. These tokens are placed by the network controller at those instants when a circuit is set up or cancelled.

Network Controller

The network controller model shown in Figure 6-17(a) completes the circuit switched ICN model. Initially, the network controller is idle as it awaits requests from processors. When a request arrives in its buffer, getreq fires, and the request is dequeued. Depending on the type of request, either subnet Cancel or set is activated. The set subnet (Figure 6-17(b)) has two parameters i and j. These parameters are set to the requesting processor and selected memory before the subnet is activated. Though there is only one replication of the subgraph, the correct values of i and j are substituted in all node attributes. Firing of transition setup places tokens in gt(0,2i), the gt signal on the link connecting the processor to the top level switch and req(sw,2j), the link connecting the memory to the bottom level switch. This causes the two signals to be propagated through the interconnection network. When the phase changes to set, T2 fires if the req signal at the processor link is set. Firing T2 causes busy(0,2i) to be set, and a state variable is set to indicate successful circuit setup. A token is also placed in Mem_start, causing the selected memory to move to the busy state. If T2 has not fired until the reset phase, T1 fires, removes the gt signal, and sets the success state variable to false.

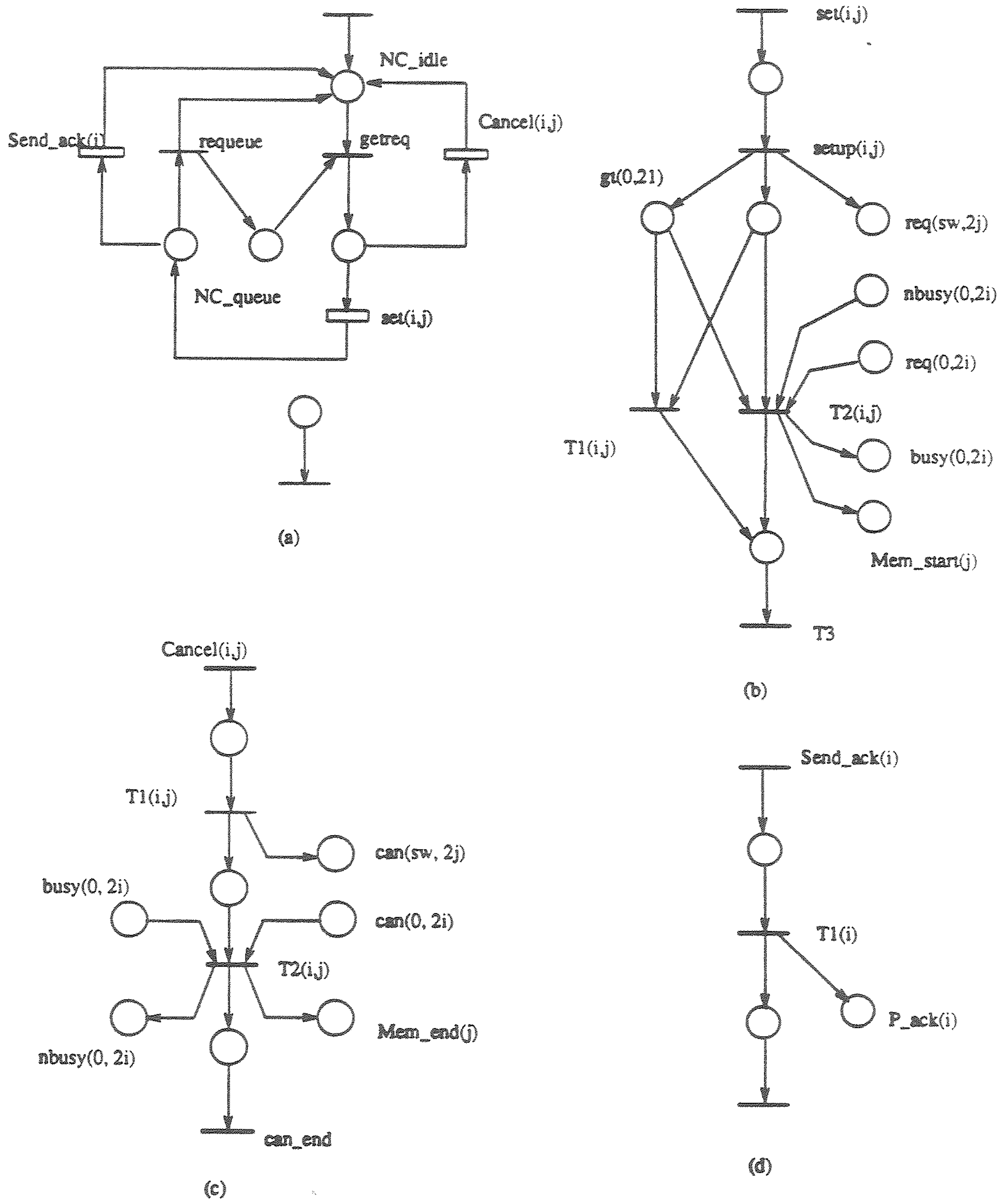


Figure 6-17: Network Controller Model

After the set subnet has completed, the success state variable reflects decides whether the request was satisfied, or must be requeued for future consideration. In the latter case, transition requeue fires, placing the request back in the network controller queue. If the request was successful, subnet send_ack becomes active (Figure 6-17(d)). The firing of T1 places a token in P_ack, informing the appropriate process that the circuit has been set up.

The cancel subnet (Figure 6-17(c)) is active if the request is of type cancel. T1 places a token in can(sw, 2j) which propagates upwards resetting all busy links in the circuit. T2 fires in the reset phase, resetting the top level can signal as well as the top level busy signal. It also places a token in Mem_end, causing the memory to become idle.

6.2.3 Modelling Results for Banyan ICN

Figure 6-18 shows the variation of the Network Controller Queue Length for different load levels in the network. Since the NC is a centralized resource in an otherwise distributed system, it is crucial that it should not become a bottleneck.

It can be seen that for a given sized network, there is an upper bound on the queue size which is twice the number of processors in the system, since a processor may delete a connection and immediately request a new connection, and thereby be present twice in the queue.

Several observations can be made from the graph. Obviously, the Network Controller Queue length is directly proportional to the loading factor. This is due to the fact that at lower loading factors, processors spend more time doing local processing, and do not make requests for circuits. Thus they do not contribute to the NC queue length. Another interesting point is that for a loading factor of 1, processors never do local processing. Therefore, they are either actively accessing shared memory, or have placed a request in the NC queue and are awaiting the setup of a circuit. For large message lengths, the circuits can be thought to be stable, implying that the 8x8 banyan under a maximum load, supports about 4 active circuits. It can also be seen that regardless of the loading factor, the network reaches this steady state for large message lengths. For smaller message lengths, the circuits are no longer stable; they are created and destroyed much more frequently. Now, for high loading factors, processors often have two outstanding requests in the queue at a time, since they cancel a previous circuit and immediately make a new request. Also, since can-

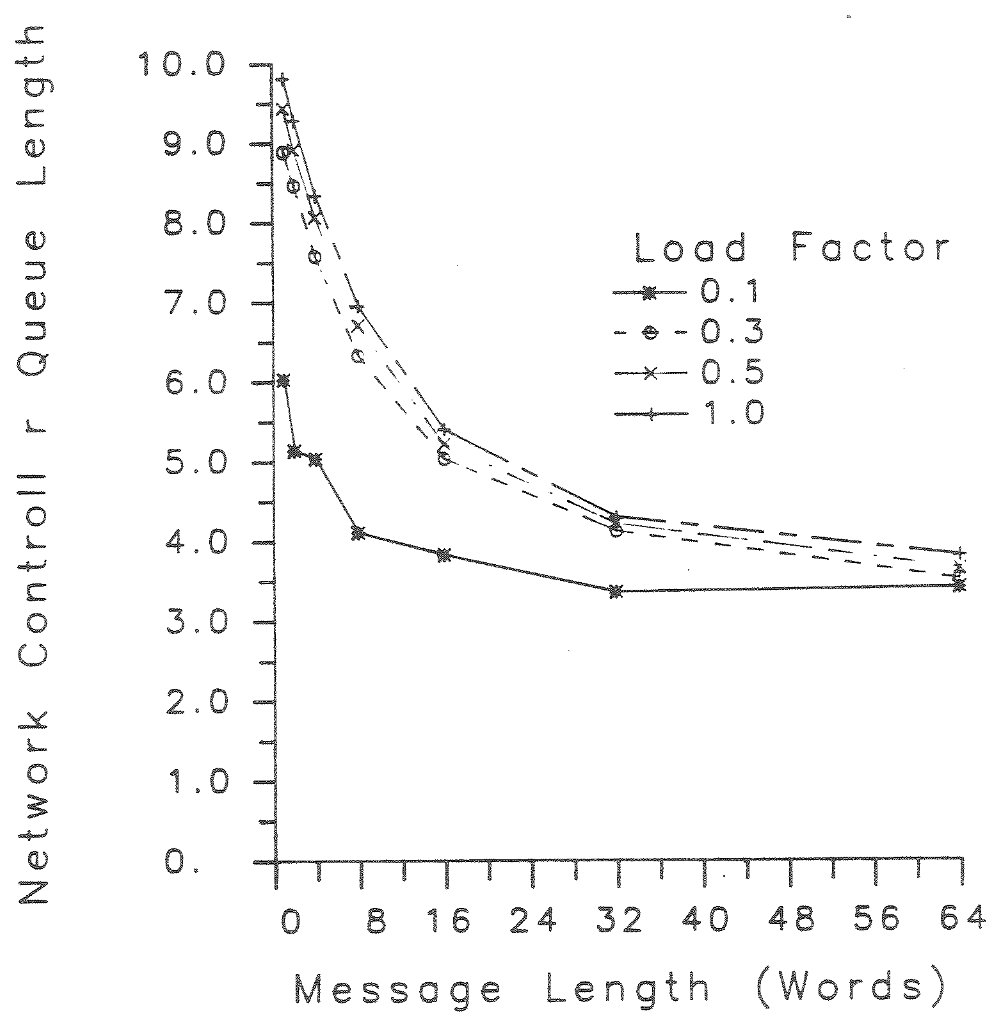


Figure 6-18: Variation of Network Controller Queue Length

cel requests are queued along with the setup requests, they are delayed by a period of time equal to the length of the NC queue (the NC satisfies only 1 request per clock cycle). This, in turn, causes additional congestion in the network, leading to a larger mean queue length at the controller. The network controller does not seem to be a bottleneck when the network is heavily loaded. Significant performance improvements, especially at low loads, can be achieved by prioritizing the cancel requests.

From Figure 6-19 it can be seen that, for the load factor of 1.0, for the

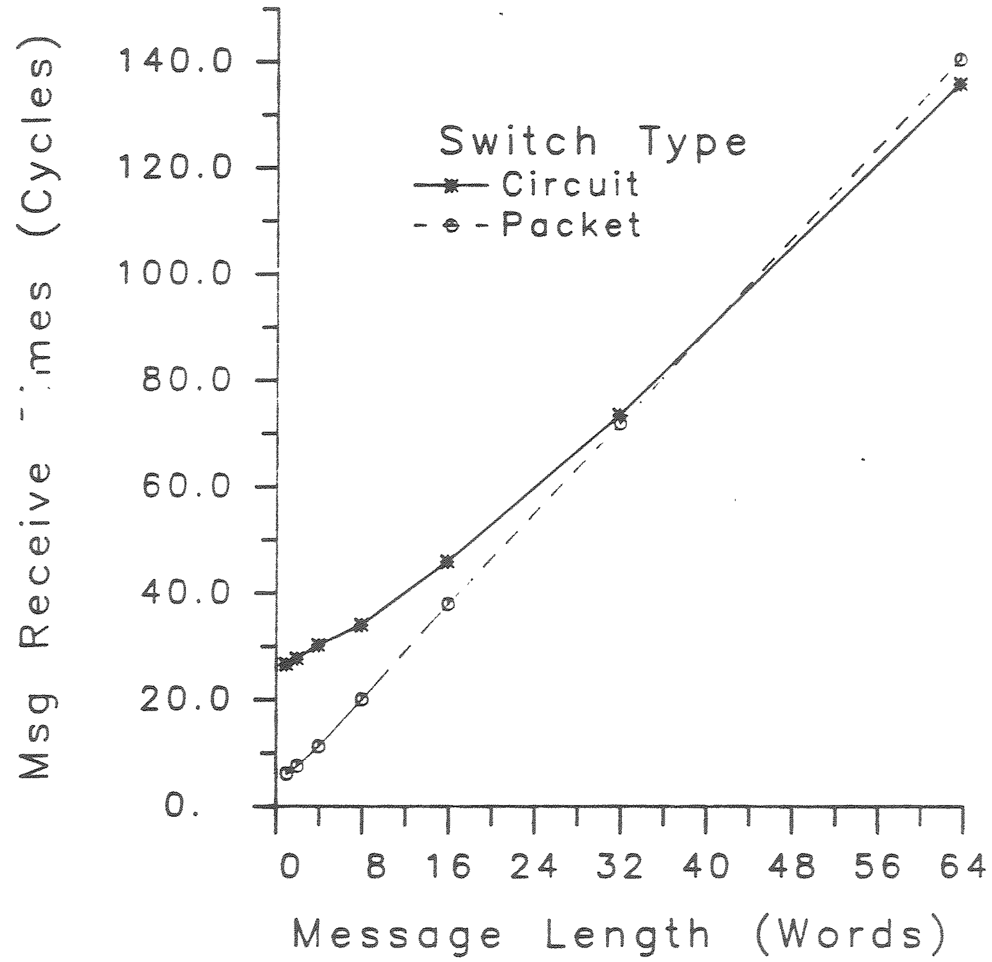


Figure 6-19: Comparison of Message Receive Times

smaller message lengths, the performance of packet switched communication is better than that for circuit switched communication. However, the difference between the message receive times for the two modes narrows as the message size is increased, and in fact, beyond a message length of about 40, circuit switching performs better. This is to be expected, since with increased message lengths, packets tend to collide more often at switch nodes, whereas for circuit switching, after obtaining a circuit, a fixed delay is sufficient to transfer the entire message.

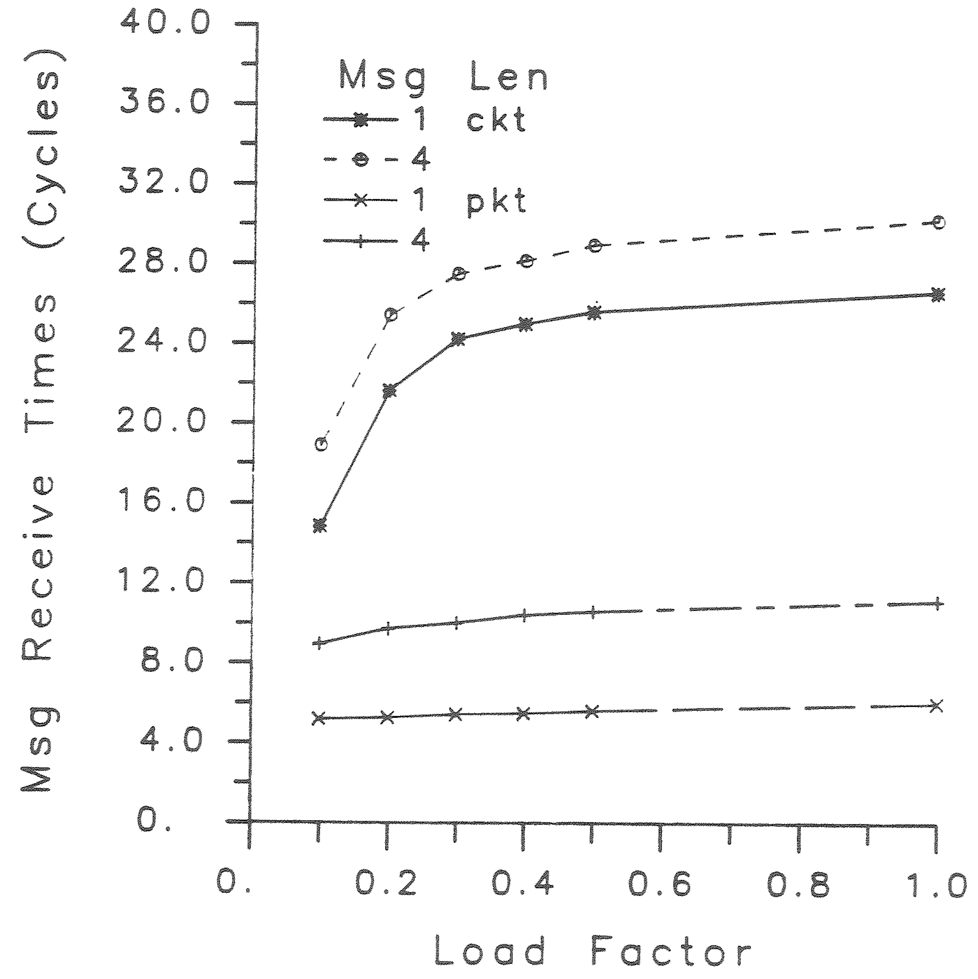


Figure 6-20: Message Receive Times vs Load Factor

Figure 6-20 shows variation of message receive times for message lengths of 1 and 4 for both packet and circuit switching. There are several interesting points to be observed here. Firstly, as expected, the message receive times for the packet mode are lower than those for the circuit mode. This is a result of blocking within the network.

As the load factor increases, more requests are made for making and breaking circuits. This increases the effect of both the Network Controller bottleneck and blocking, resulting in rapid increase in the message receive times for circuit switching. However, beyond a certain point, the increase in the Network Controller queue due to bottleneck has an effect opposite to that of increasing loading factor, since the processors spend more time in the Network Controller queue. This results in flattening of the circuit switching curves. For packet switching, on the other hand, the curves are almost straight lines and the load factor has little effect.

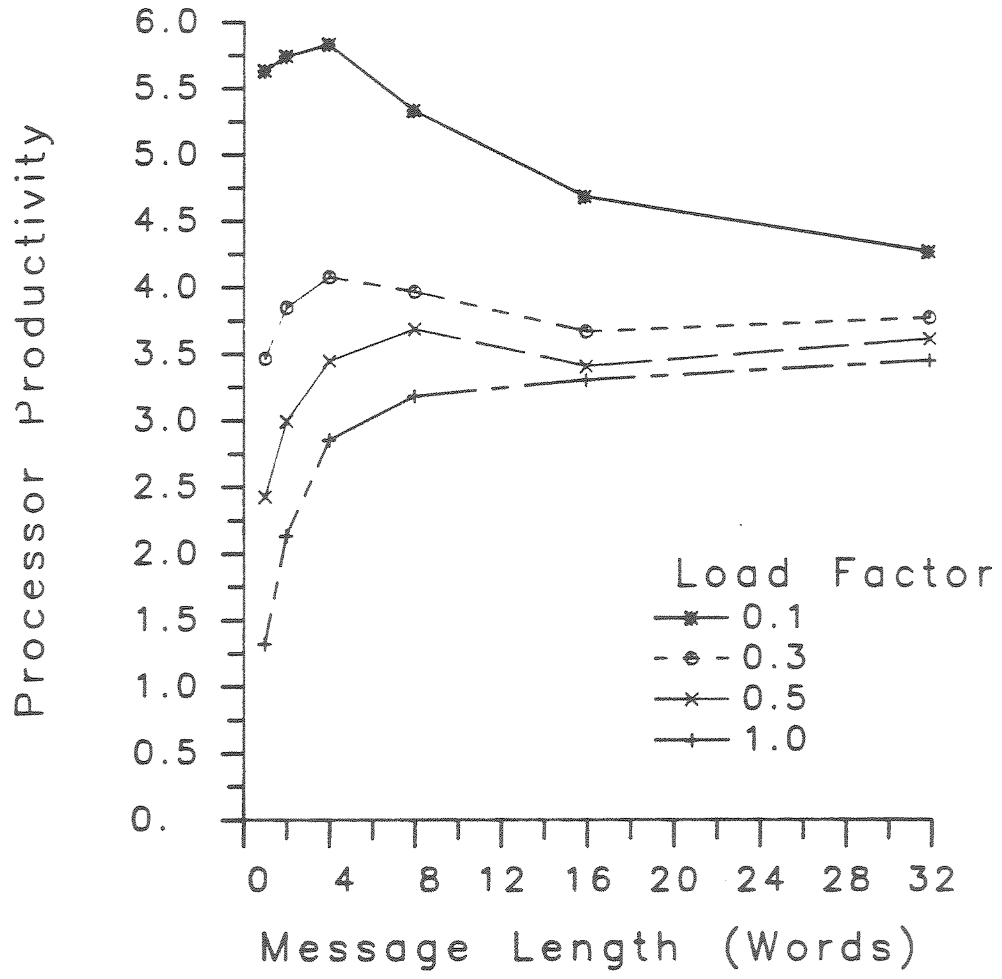


Figure 6-21: Processor Productivity vs Msg Length (packet)

Figure 6-21 shows variation of processor productivity with message length for various values of load factor. The curves for higher load factors lie below those for lower load factors. This is to be expected, since at higher load factors processors spend less time doing local processing and more time waiting for messages. Since a packet passes through at most four layers of switches and since the priority of the switches alternates, there is an upper bound on the amount of delay a packet undergoes before it reaches its destination. This collective overhead decreases relatively as the message length increases for higher load factors. On the other hand, for lower load factors, the collective overhead of all packets of a message has an adverse effect on the relative amount of time a processor spends in the local state, since that is independent of the time spent receiving a message. This results in depressing processor productivity at lower load factors while enhancing it at higher load factors. Finally, for large message lengths, the processors spend most of their time waiting for messages to arrive. The difference between the times they spend in the local state, as an effect of load factor, becomes relatively unimportant, and all curves approach a common value asymptotically.

The productivity curves shown for circuit switching (Figure 6-22) are, in fact, mirror images of the network controller queue lengths. By the definition of productivity used here, a processor does useful work while it is either engaged in local processing or actively accessing shared memory. In other words, the only time a processor is not 'useful' is when it has a pending request at the Network Controller.

6.3 Concluding Remarks

Petri Net based models are well suited to modelling computer architectures. The PCM model has been shown to be capable of representing complex multiprocessor architectures such as the ICN architectures. The parameterization features of the model make it necessary to specify precisely one instance of each component in the system. It is, for example, necessary to specify only one switch instance. The subnet parameterization used in conjunction with the functions mapping switch indices onto link indices effectively result in a banyan interconnection when the switch subnet becomes active. Further, other interconnection networks such as the Omega or Baseline networks can be synthesized merely by changing the mapping function.

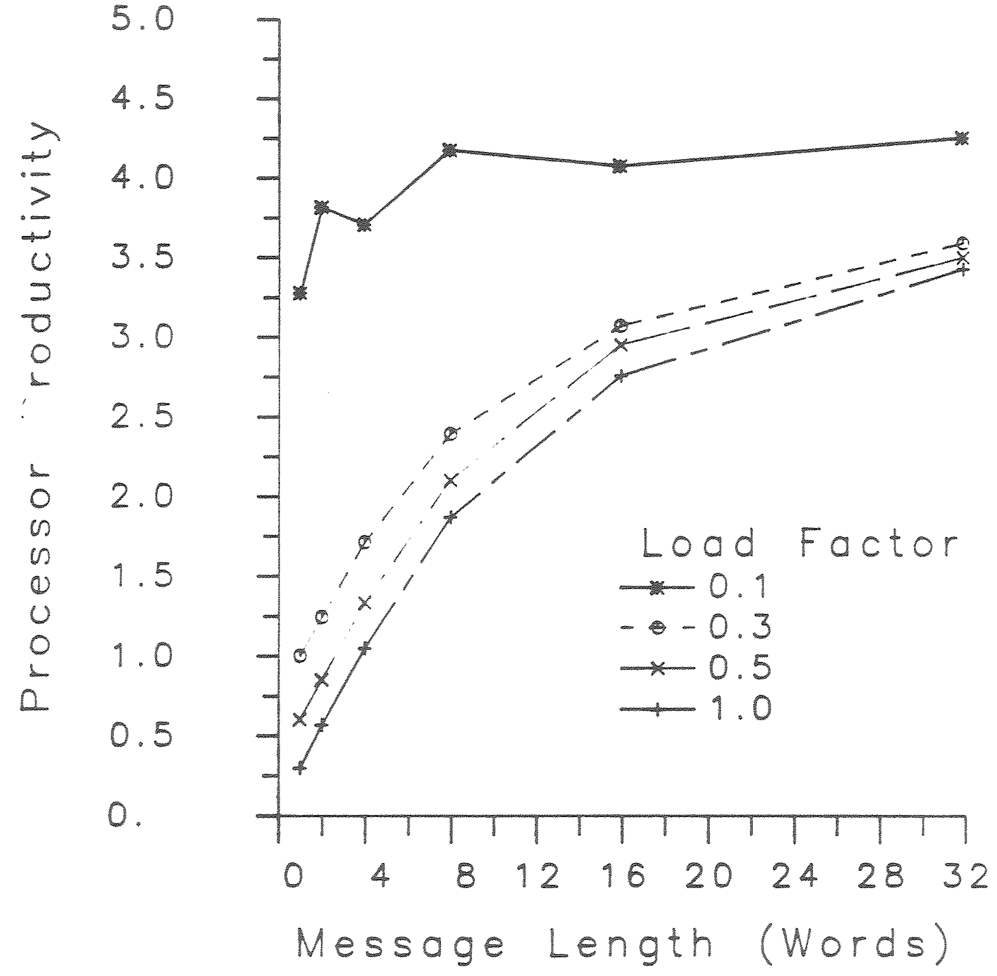


Figure 6-22: Processor Productivity vs Msg Length (circuit)

The multibus architecture shown here demonstrates the simple and compact nature of the PCM model. An advantage of using this model is that each processor, memory and bus are modelled separately, thus preserving their identities. This is in contrast with previous models in which all processors, for example, are modelled in one combined net. The advantage of preserving the identities of the architectural resources becomes obvious when modelling the execution of computations on the architecture, since the access patterns are no longer probabilistic.

There are some problems with using Petri nets to model computer architecture. In particular, most actions taken in a digital circuit usually depend on either the presence or absence of a signal. If the signal is modelled by a Petri net place, it becomes impossible to force an action to occur when the signal is absent (i.e. there is no token in the place). In the extended model, there are two ways to avoid this situation. As in the case of standard Petri nets, the signal could be modelled by two places: one representing the absence of the signal. A token would then be always present in either of these two places, and this token would cause subsequent actions. Another method is to include as a state variable a boolean counter for each place which models a signal. This counter could then be used in transition predicates which could check for the absence of the signal.

Another problem is the lack of a global clock in Petri Nets. Most computer systems do contain several components which behave in a synchronous manner. The only way to enforce this synchronous behavior among the transitions in the PCM model is to implement a clock as shown earlier.

Chapter 7

Modelling Computations using the PCM

This Chapter demonstrates the utility of the PCM model and its development/execution environment, PCSIM, through model studies of the effects of task and communication unit granularity and of the effectiveness of different realizations of processor-memory connections on two example computations. These studies begin with models of *programs* for the computations which capture the patterns of resource usage of the computations. Introduction of patterned resource usage allows detailed analysis of the effects of varying parameters in both the computation and the underlying architecture.

The two computations modelled are CSL programs for the solution of a Block Lower Triangular (BLT) system and solution of a Block Tridiagonal matrix by the odd-even elimination method in a formulation due to Kapur [KAP82]. The computation structures of these programs are expressed in PCM and mapped upon several variations of the multibus and banyan ICN network architecture described in the previous Chapter. The PCM models are parameterized to implement a spectrum of granularity for the computational tasks and the elementary units of communication (access to non-local memory by a processor). The granularity of the units of computation is determined by the block size for the submatrix in each algorithm while the granularity of communication is either copying of an entire submatrix block, or word-by-word access. The architecture variants are a banyan ICN and a multiple bus connections of processors to memories.

Section 7.2 contains the modelling results for the Block Lower Triangular System. A parallel algorithm for solving the BLT system, along with encodings of the algorithm in CSL and TDFL are presented. The programs are presented mainly to allow the comparison of the actual models created with the equivalent CSL programs to highlight the similarity between them. The results obtained for a message based formulation of the Block Lower Triangular system are validated against results obtained from actual execution of the algorithm on a Sequent Balance-21000. The simulation studies of the BLT system are discussed at the conclusion of the Section.

The computations are expressed using the parallel languages described in Appendix B (TDFL and CSL). The two formulations differ mainly in that computations expressed in CSL include both shared variables and messages as means of communication, while TDFL, being a data flow language, allows only message based communication between concurrent processes. The PCM model instances for each of these programs are then obtained by using the transformations given in the Appendix. The transformation from CSL or TDFL programs to the model instance is incomplete since it lacks timing information as well as details of the underlying architecture. After including these details, the model is used to study the performance of the computation. Though the models used in this Chapter closely resemble those that would have been obtained using the transforms, they have been simplified for clarity. As illustrated by the examples in this Chapter, the performance of the computation can be studied for several interesting configurations.

Section 7.3 describes the Odd-Even Elimination algorithm for solving Block Tridiagonal System, and then outlines the model and simulation results for several configurations of the computation. The final section contains some concluding remarks.

7.1 Interface between the Architecture and Mapping Submodels

The architectural models specified in Chapter 6 were designed to study the behavior of the architectures under probabilistic loads. In order to use them in conjunction with parallel computations, they are modified to present a standard interface to the mapping submodel, as described in the methodology. The mapping submodel binds processes (or tasks) to processors and transmits process requests to processors, and the processor response back to the process.

A process *S* behaves in the following manner. When it is initiated, it sends a *load* request, by placing a token in *S_load*. It then awaits confirmation from the processor (a token in *S_ready*). An initiated process can execute by placing a request to its processor. A request consists of a (possibly empty) sequence of remote accesses and a computation delay. A remote access is merely a two-tuple specifying a remote memory number and the number of words to be transferred from that memory. Since all the applications that are studied here are numeric and involve only floating point numbers, the size of the request will be specified in terms of floating point words (one floating point word is equivalent to four words). The remote accesses are assumed to be *uniformly* distributed over the computation delay.

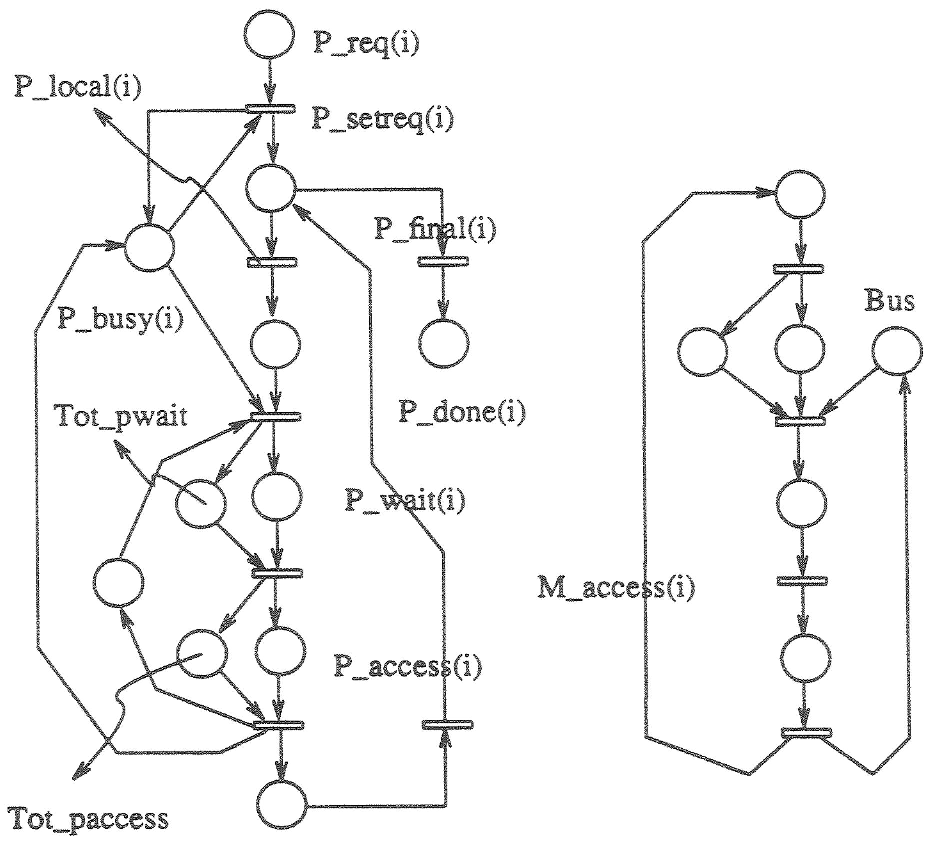


Figure 7-1: Processor and Memory Models for the Multibus Architecture

A processor can be in one of four states: *idle*, *busy*, *accessing* a remote memory, or *waiting* to access a remote memory. The processor is initially in the *idle* state until it receives a process request for loading. This causes the processor to move to the *busy* state, signalled by a token in *P_busy* (Figure 7-1). When a process starts execution, it makes a request to the processor, which is signalled by a token in *P_req*. It is assumed that a processor is dedicated to a single process once the process has been loaded. The arrival of a token in *P_req* causes *P_Setreq* to fire. The transition procedure for *P_Setreq* sets up the request in a request queue associated with the processor. It also calculates the number of bus accesses required to complete the entire request, and the resulting computation delay between successive accesses. *P_local* models this computation delay, after which the remote access is made. *Tot_paccess* and *Tot_pwait* are used to collect overall statistics for all processors. After each access, P_setreq updates the processor's request queue, and the next computation delay ensues. When the last access is complete, the predicate for *P_final* becomes true, and a token is placed in *P_Done* signifying the completion of the execution/remote access. Execution of a process without remote accesses is modelled by making a request with a computation delay, but no remote accesses, and results in the firing of *P_final* without ever enabling *P_local*. The entire computation delay is assigned to this transition.

The memory model for the multibus architecture is the same as that shown in the previous Chapter. Each memory has a FCFS queue which contains processor requests, and when a request is present in the queue, the memory checks if a bus is available before introducing a delay equal to the access delay. After this delay, the memory signals completion of the access to the processor.

The model for the Banyan interconnection network architecture is similarly modified. The only other interesting detail for the Banyan architecture model is the introduction of a "micro clock" to reduce the number of events in the system. The micro clock has the effect of only enabling the interconnection network when there is at least one processor actively accessing a remote memory.

Both architectures are assumed to be based on the Motorola 68020 microprocessor. All computation delays are assigned using timing values obtained using a SUN-3/50 workstation with a 68881 floating point coprocessor. Using the individual delays for each high level statement, the delay for an entire process is expressed in terms of its size (or granularity). The cost of transferring a floating point word from a remote memory is specified as τ_{sh}, and is of the order of two to five times the cost of a local access. A local access is assumed to be equal to 0.24 μsec. τ_{cs} is the overhead incurred when loading a process on a processor. It is assumed that this is a fixed cost, and is not affected by the nature or size of the process itself. A simplified model is used for representing I/O. The assumption made here is that each processor has access to a single I/O device which has DMA capability. This device can satisfy several requests simultaneously, and each DMA transfer causes a processor to incur a delay proportional to the number of words transferred. In particular, it is assumed in the following examples that the DMA transfer of floating point data from the device into the local store of a processor causes a delay equal to 0.8*n, where n is the number of floating point words transferred.

7.2 Block Lower Triangular System

The Block Lower Triangular Algorithm is a parallel solution of a lower triangular matrix:

$$Tx = b$$

where T is an nxn lower triangular matrix, x and b are n-vectors.

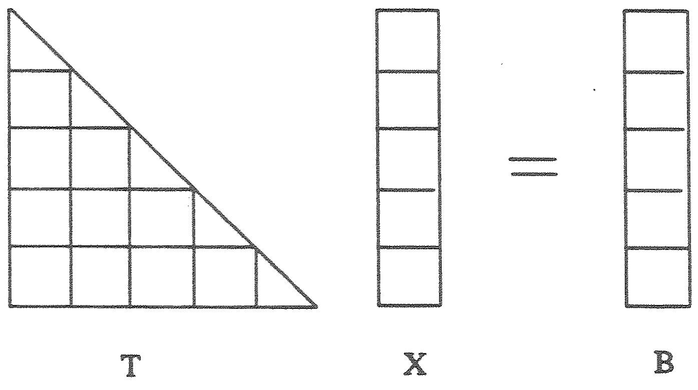


Figure 7-2: A Triangular System

The algorithm decomposes the matrix into blocks as illustrated in Figure 7-2, and allows the solution of different blocks to be carried out in parallel. There are three types of tasks in the computation: *init*, *solve* and *matvect*. The tasks are described next:

init(i)	initializes one block row of the system.

solve(i) triangular solver for the *i*th triangular diagonal block. It solves for x_i in the system:

$$T_{ii}x_i = b_i$$

This can be done only after all matvect tasks for row i have completed. Observe that solve(1) (solve for block 1) can begin immediately after initialization, and does not depend on any matvect tasks.

matvect(i, j) executes the transformation

$$T_{ij}x_j - b_i \dashrightarrow b_i$$

on the *i*th block in column j. This step can only be executed if solve(j) has been completed.

7.2.1 BLT Solution using CSL and TDFL

Figure 7-3 shows the CSL program which specifies the Block Lower Triangular computation. The CONSTRUCT statement contains declarations of the tasks, shared variables and communication channels involved in the computation. The declaration of Solve, for example, specifies N tasks Solve(1), Solve(2)...Solve(N). All Solve tasks are identical, and their compiled (object) code is found in file C2. A list of shared objects accessible to each task completes the specification of the Solve tasks. The Matvect tasks process the non-diagonal blocks of the matrix. Associated with each Matvect is a boolean task condition which is set after each execution of the task.

The program starts off N parallel streams, as denoted by the outer COBEGIN. Each parallel stream begins by executing a diagonal (solve) task first, and then all non-diagonal (matvect) tasks in that column in parallel (inner cobegin). The WAIT statement ensures that the execution of each solve task begins only after all matvect tasks in its row signal their completion by setting their task conditions. The statement:

```
WITH T(j,i), X(j) : X(i) DO
   EXECUTE  Matvect (j,i);
```

specifies that the task matvect requires exclusive access to shared objects T(j,i) and X(j), and wishes to use X(i) in read-only, or non-exclusive mode. Finally, the CSL program demonstrates the power of the RANGE statement as a construct for parameterizing the computation.

```
JOB Triangle;

VAR N : integer;

BEGIN
  N := 4;
  CONSTRUCT
      TASKS
        Init(i) : C1 [ T(i,j), X(i) ]
                  RANGE j = 1 to i,
                        i = 1 to N;
        Solve(i) : C2 [T(i,i), X(i) ]
                  RANGE i = 1 to N;
        Matvect(i,j) : C3[T(i,j), X(i), X(j)]
                       CONDITION C(i,j)
                  RANGE j = 1 to i-1,
                        i = 1 to N;
  END; { CONSTRUCT }


  WITH T(i,j), X(i) DO
    EXECUTE Init(i)  RANGE j = 1 to i,
                           i = 1 to N;

  COBEGIN
    ( // WAIT C(i,j) RANGE j = 1 to i-1;
         WITH X(i), T(i,i) DO
             EXECUTE Solve(i);
         COBEGIN
           (// WITH T(j,i), X(j) : X(i) DO
                  EXECUTE Matvect(j,i)
           ) RANGE j = i+1 to N;
         COEND
    ) RANGE i = 1 to N;
  COEND;
END.
```

Figure 7-3: CSL Program for Triangular Solver

The TDFL solution is shown in Figure 7-4. The single Initialize task in the CSL solution has been partitioned into two tasks, one to read and distribute the blocks of the T matrix, and the other to read and distribute the initial values of B. Each Solve task, as before, solves a smaller system, and passes its results to all MVs in its column. The MVs receive this solution and use this value to update the B vector. All final results are collected by a special Write task.

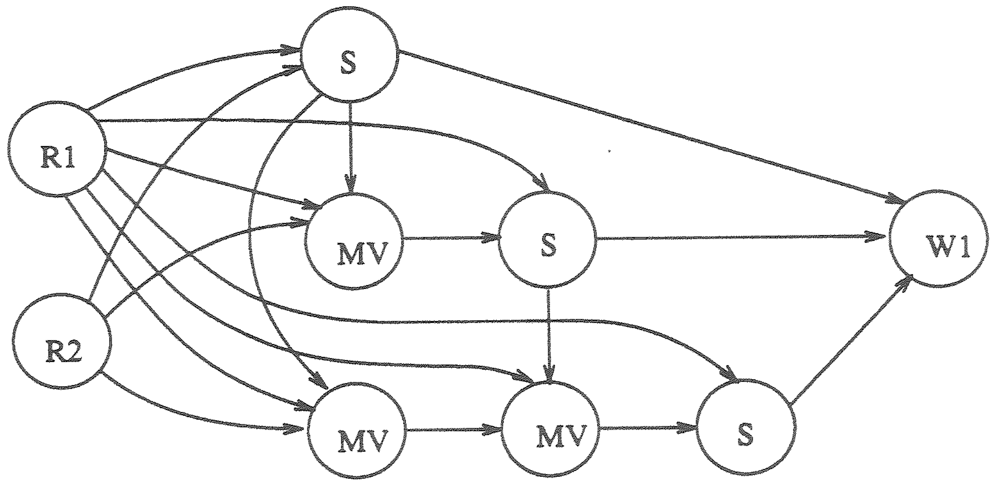
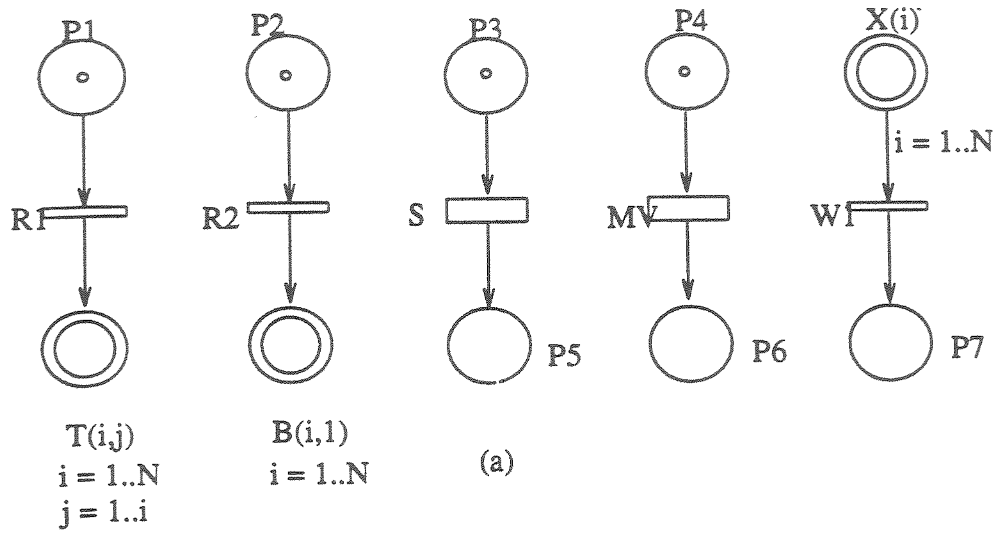


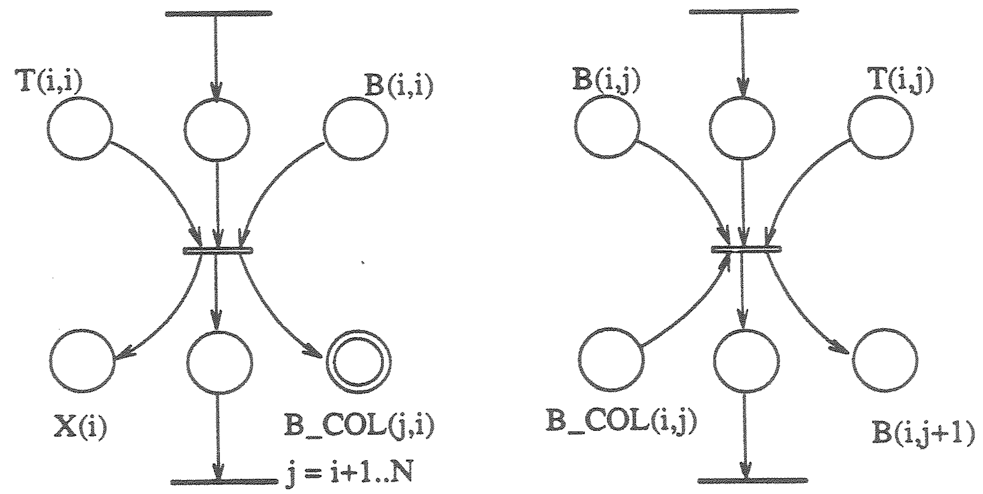
Figure 7-4: TDFL Solution for BLT System

One difference between the two solutions is that the TDFL solution, being message based by nature, avoids conflicts for shared data by having multiple copies of data required by more than one task, thereby increasing the communication delays. A Detailed description of the PCM model for the TDFL formulation can be found in [ADI87b].

The PCM model was used to model the behavior of an implementation of BLT using TDFL on a Sequent Balance-10000. The delay values for individual tasks were obtained by running the tasks in isolation on the Balance. The validation comprised of several configurations of number of partitions as well as number of processors. The resulting metrics were the overall execution time, and the processor utilization, both of which agreed closely with the timing values from the actual runs. The PCM model for this experiment was as shown in Figure 7-5.



(a)



Subnet S
i = 1..N

(b)

Subnet MV
i = i..N
j = i+1..N

(c)

Figure 7-5: PCM model for TDFL Solution

Validation results for TDFL Executions				
nxp	exec time (A)	exec time (S)	proc util (A)	proc util (S)
1x1	13001.1	12295.5	0.9997	1.000
2x1	13158.5	13167.8	0.9994	1.000
2x2	13111.6	13081.7	0.5032	0.5033
3x1	12977.2	12919.6	0.9989	1.0000
3x2	10055.1	10058.9	0.6418	0.6422
3x3	10043.9	10058.9	0.4292	0.4281
4x1	12918.6	12863.8	0.9982	1.000
4x2	8073.4	8083.6	0.7958	0.7957
4x3	8108.2	8083.6	0.5313	0.5305
4x4	8115.0	8083.6	0.4032	0.3979
6x1	12907.2	12787.2	0.9956	1.000
6x2	7186.6	7158.4	0.8908	0.8932
6x3	6123.0	6112.2	0.6965	0.6974
6x4	5780.2	5772.9	0.5533	0.5538
6x5	5779.0	5772.9	0.4428	0.4430
6x6	5794.6	5772.9	0.3687	0.3692

Table 7-1: Comparison of Simulation Results with actual executions

7.2.2 Modelling the BLT System

The X vector is the only shared data object in this algorithm, and is placed in the remote memories. Figure 7-6(a) shows the PCM model for the *init* tasks which initialize the values of X. The mechanism used for modelling the loading of tasks on processors, making requests for reading/writing remote memories and for signalling the completion of execution of a task is identical for all tasks, and will be described only for the init tasks shown here. When a token is placed in I_load, the process (or task) is loaded on a processor specified by the mapping function. The loading may be delayed if the requested processor is busy. When the process has been loaded, a token arrives in I_ready. Requests for remote memories are flagged by placing a token in I_req. Associated with each req place (i.e. with each request) are two state variables containing the id of the requested memory and the number of data words to be transferred. The request is relayed to the appropriate processor using the mapping function, and when the the transfer is complete, a token is placed in I_rec. Finally, placing a token in I_done signals the termination of the task and causes its associated processor to be released.

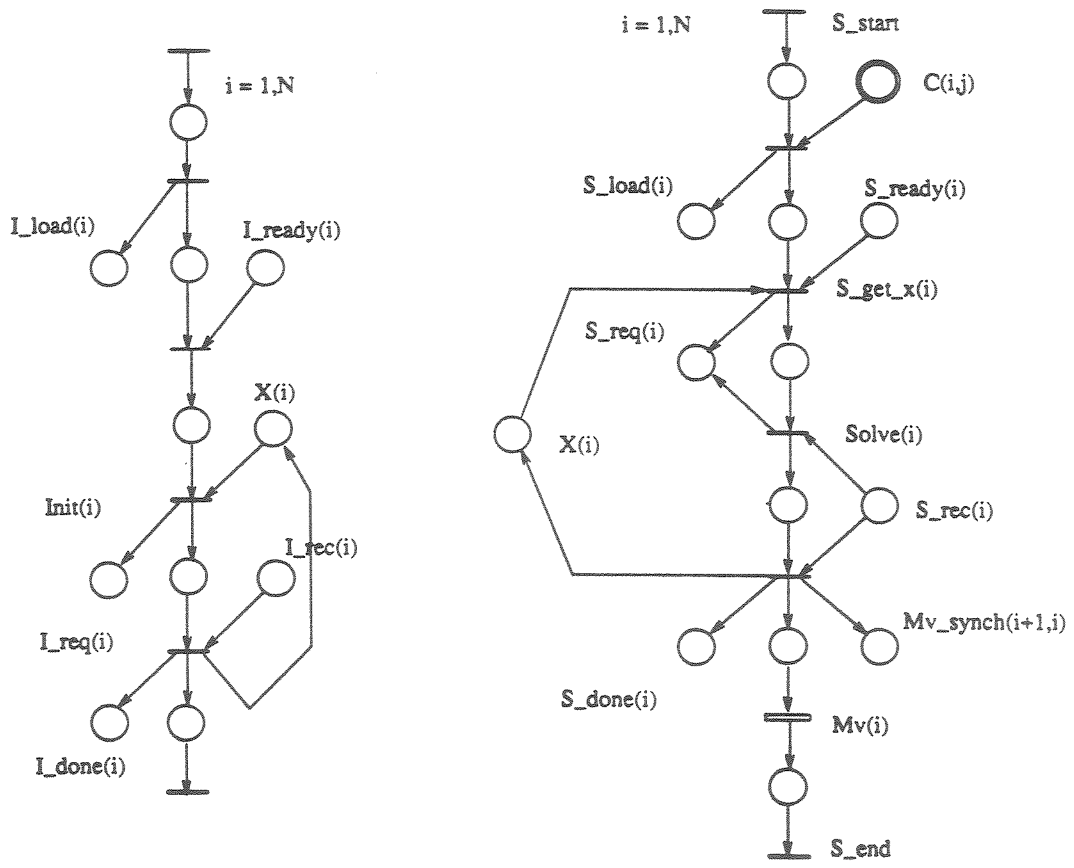


Figure 7-6: PCM Representation of Init (a) and Solve (b) tasks

After *init(i)* is loaded, the shared object X(i) is locked and transition I_init(i) fires, placing a token in I_req. The request is accompanied by the name X(i) and the delay for copying (or accessing) it. The mapping submodel converts this logical name (X(i)) into a physical memory module number (say, memory number 2). A token arrives in *I_rec* only after the access and the delay have been carried out.

Each Solve task (Figure 7-6(b)) awaits completion of all Matvect tasks in its row (tokens in places C(i,1), C(i,2),..., C(i,i-1)) before it initiates loading. Transition S_get_xi is used to lock X(i) and initiate the copying of X(i) into local memory. Transition Solve models the actual execution of the task by introducing a delay which is specified as a function of the size of a block. After Solve(i) completes, all Matvect tasks in its column can begin. This is represented by the subnet Mv(i) which is parameterized as shown in Figure 7-7. The model for each Matvect task is similar to that for the Solve, and requires no further explanation.

Figure 7-8 shows the delays assumed for each of the tasks in the computation. The delays are all in µsecs, and are specified in terms of the size of each block.

The modelling results presented here are for a 512x512 BLT system. The execution behavior of this system is studied on both the Banyan and the multibus architectures. The number of partitions is varied (causing a variation in the granularity of a task), and its effect on the performance is studied. For the multibus architecture, two sets of runs are conducted: one with τ_{cs} = 1.2 msecs and 8 buses, and the other with τ_{cs} = 0.2 msecs and only 4 buses. Henceforth, the 4 bus multibus will be referred to as multibus I, and the 8 bus multibus as multibus II.

Figure 7-9 shows the variation of execution time with the number of partitions for the banyan and multibus (II) cases. The general trends for both the curves are similar, and can be easily explained. When the number of partitions is increased from 1 to 2, the algorithm remains, in essence, a sequential algorithm, since the order Solve(1), Matvect(2,1), Solve(2) must be followed sequentially. In fact, there is the extra overhead caused by the additional reads and writes from the remote memory module, which actually causes an *increase* in the execution time! As the number of partitions is further increased, the execution time drops rapidly at first, but then starts to even out. This is because as the number of partitions is increased, the overheads in the computation also increase, and the greater parallelism is offset by this overhead. In

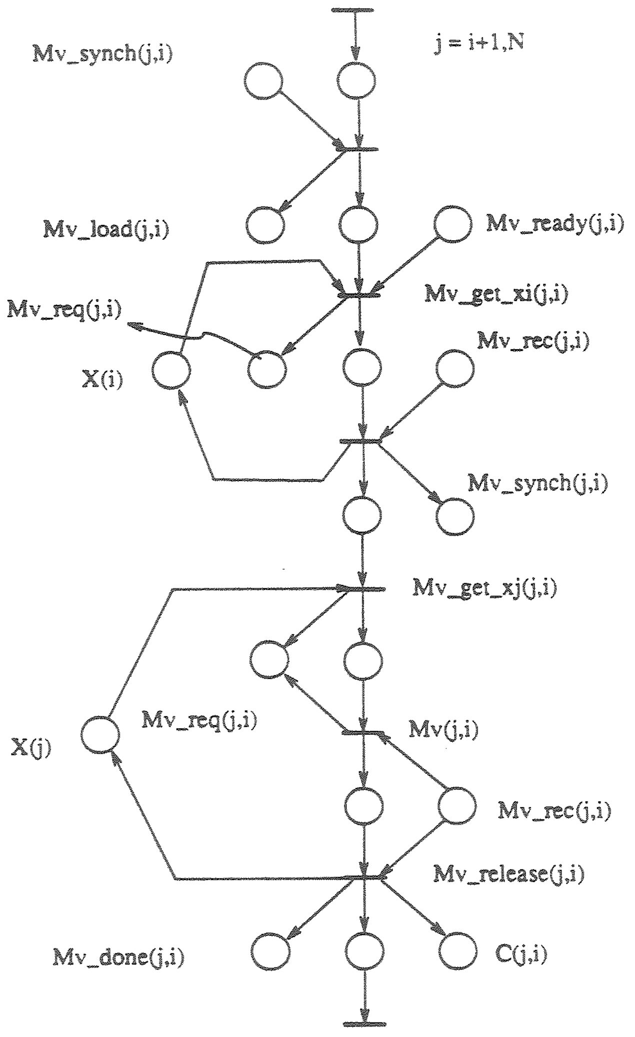


Figure 7-7: PCM Representation of Matvect tasks

For block size s,

matvect	s(30.48s + 1.96)
Solve	s(15.24 - 9.96)
Init	0.8s

Figure 7-8: Delays (in µsec) associated with some operations

this case, the most significant overhead is τ_{cs}, since it is directly associated with the

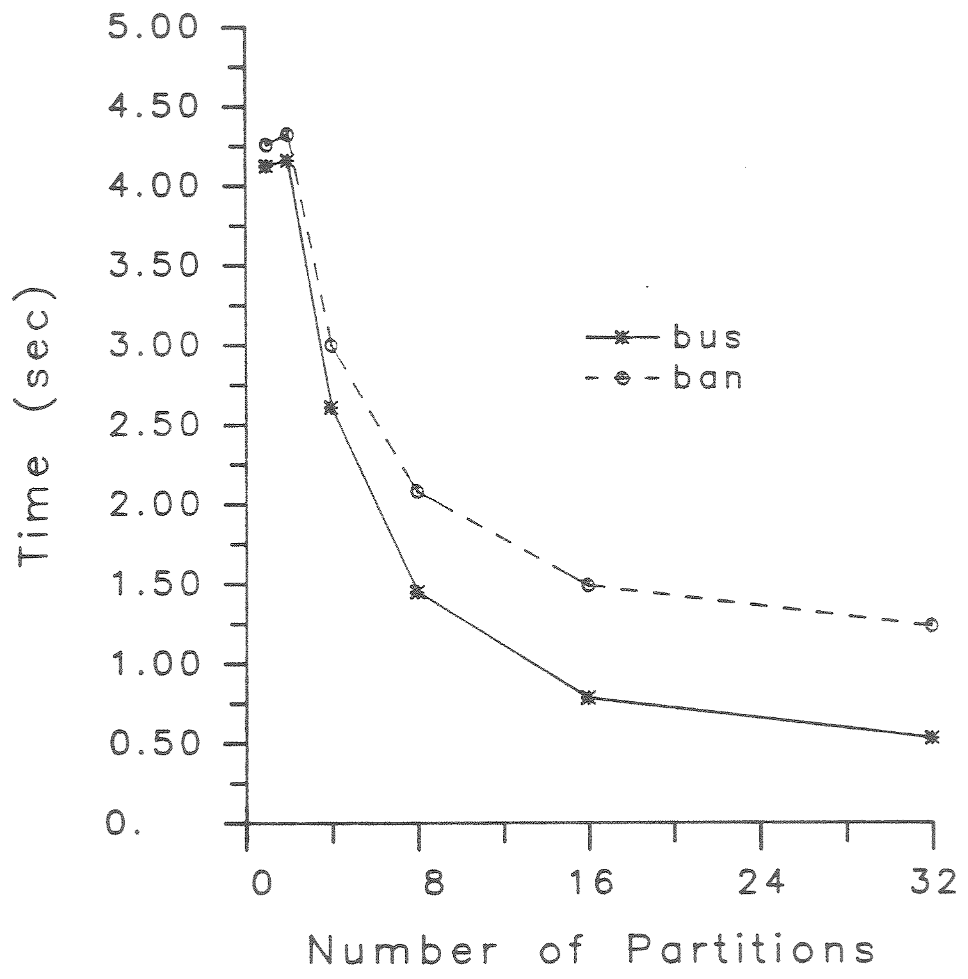


Figure 7-9: Execution Time vs No. of Partitions

number of partitions in the system. The Banyan case has higher execution times, since the granularity of each access is much larger than for the multibus (II) case. Once a circuit has been set up, it is held for the entire period of time that the remote memory is accessed, and thus contributes towards blocking in the network. Figure 7-10 shows that as the number of partitions is increased from 32 to 64, the execution time increases, since the overheads become prohibitively large. Also, it can be seen that bus contention does not seem to be a problem for the 4 bus case, since there is no significant performance degradation.

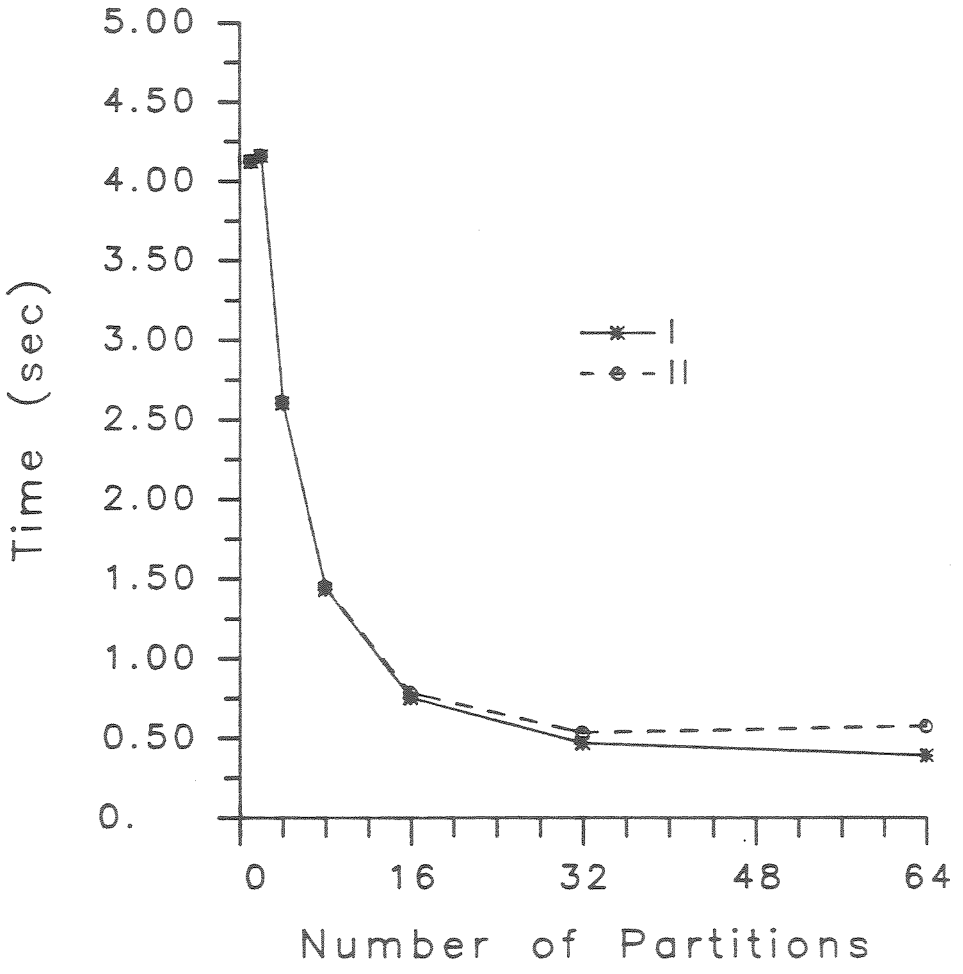


Figure 7-10: Execution Time vs No. of Partitions

Figure 7-11 shows the effect of granularity on the Speedup. (Speedup is used in the traditional sense as being the ratio of the sequential execution time to the time taken for the parallel solution.) The ideal curve was calculated using a simplified model which assumes a PRAM - CRCW type of architecture with infinite numbers of processors, and *no* overheads of any kind, which leads to the following expression:

$$T_{comp} = \tau_{init} + N\tau_{solve} + (N-1)\tau_{mv}$$

This follows from the observation that if there is no contention for resources

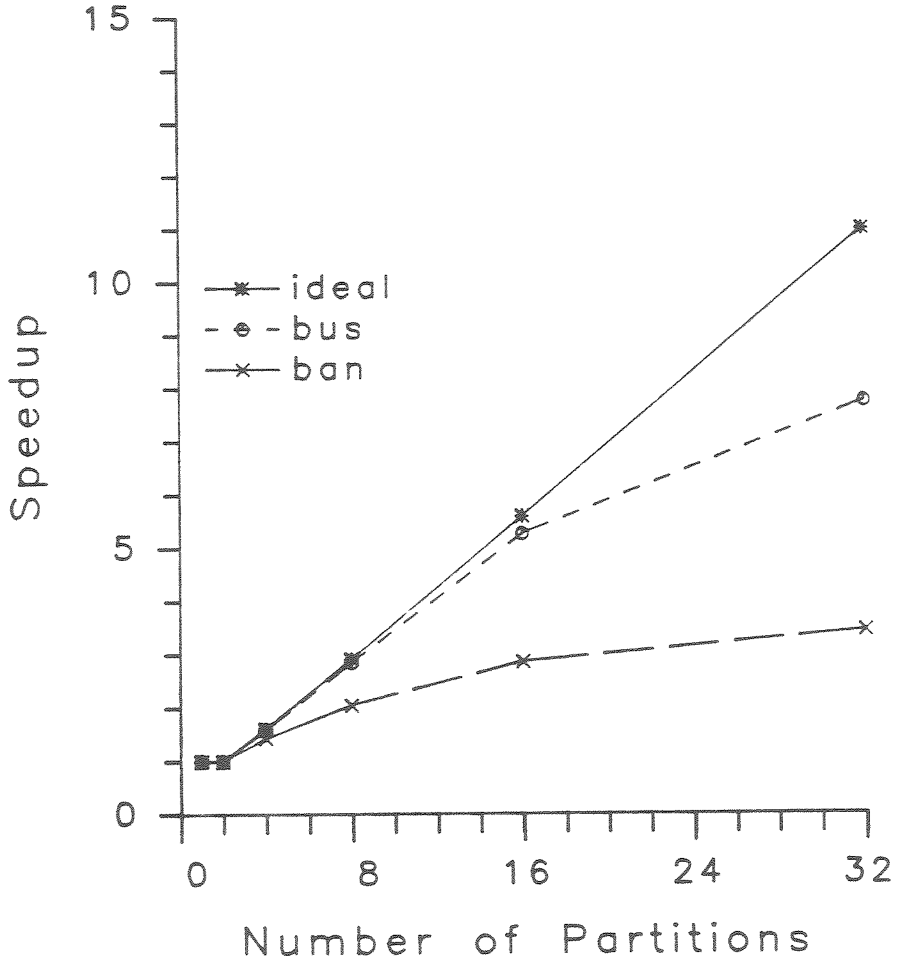


Figure 7-11: Speedup vs No. of Partitions

(processors and memories), execution of the solve and matvect tasks will be interleaved. This is a lower bound on the time taken to execute the BLT solution on any architecture. The multibus case shows near ideal speedups for upto 16 partitions, but then begins to decrease as processor contention as well as communication overheads take effect. Again, the larger granularity of the memory accesses in the banyan case are reflected in the lower speedups.

Figure 7-12 shows the processor utilizations for all three cases. When the

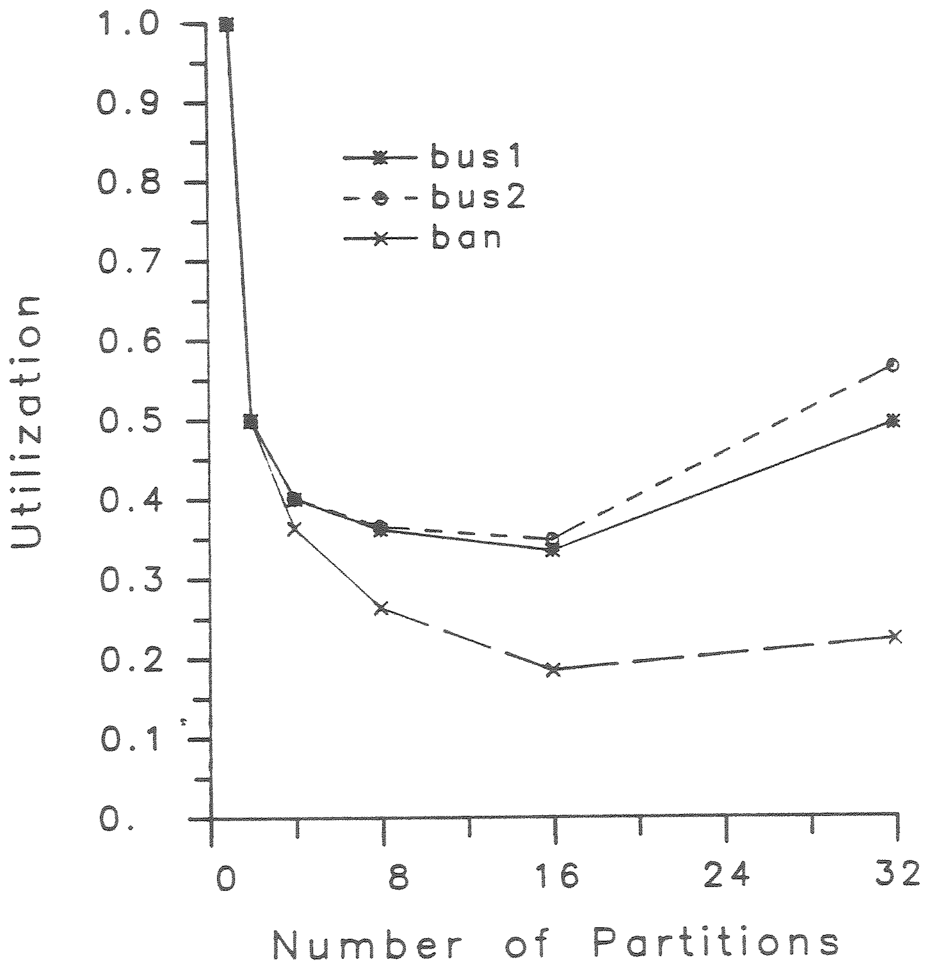


Figure 7-12: Processor Utilizations vs No. of Partitions

number of partitions is increased beyond 16, the processor utilizations actually increase, since the smaller granularity of each task now causes a decrease in the time spent by other processes for synchronization. Figure 7-13 shows that the network controller queue length increases with the number of partitions. This is because the processes make more frequent, but smaller, accesses from remote memories. This has the same effect as a larger "loading factor" as defined in the previous Chapter.

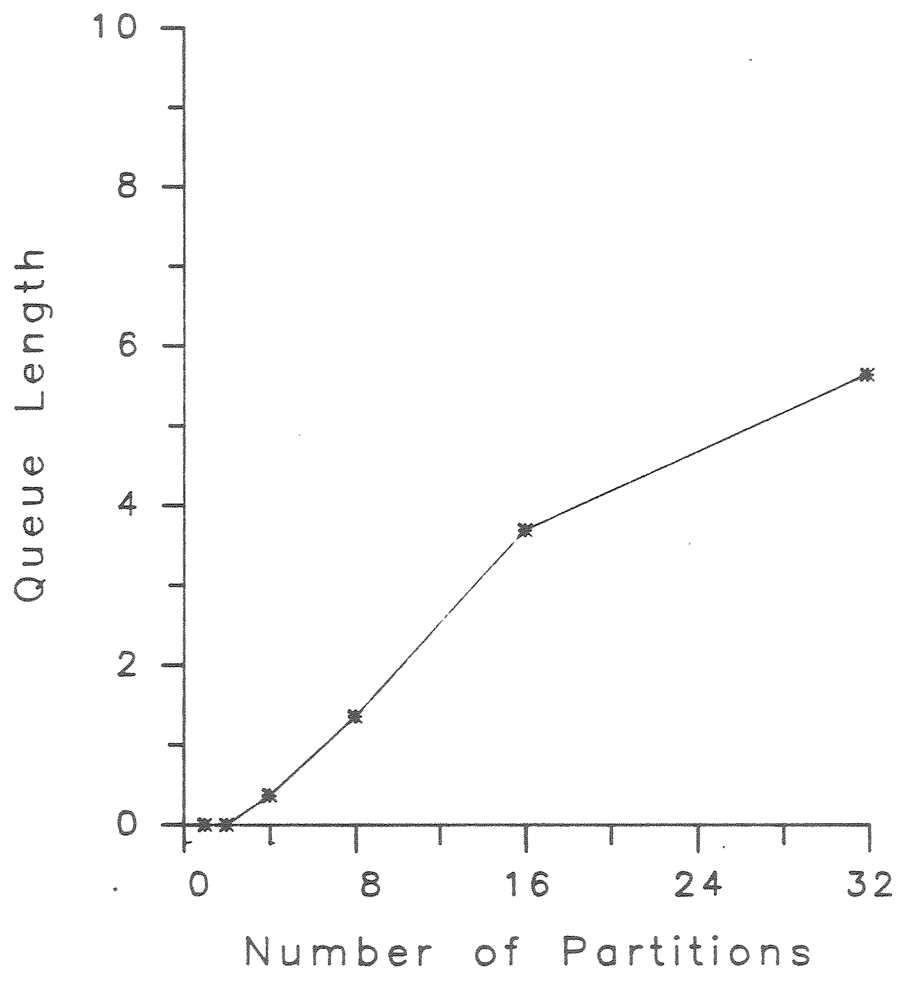


Figure 7-13: Network Controller Queue Lengths

7.3 Block Tridiagonal System

A Block Tridiagonal matrix can be defined as follows.

$A = (a(j), b(j), c(j))_N$

where b(i) is an nxn matrix and a(1) = 0, and c(N) = 0. A Block Tridiagonal System is then a system of equations:

$$Ax = v, A = (a(j), b(j), c(j))_N$$

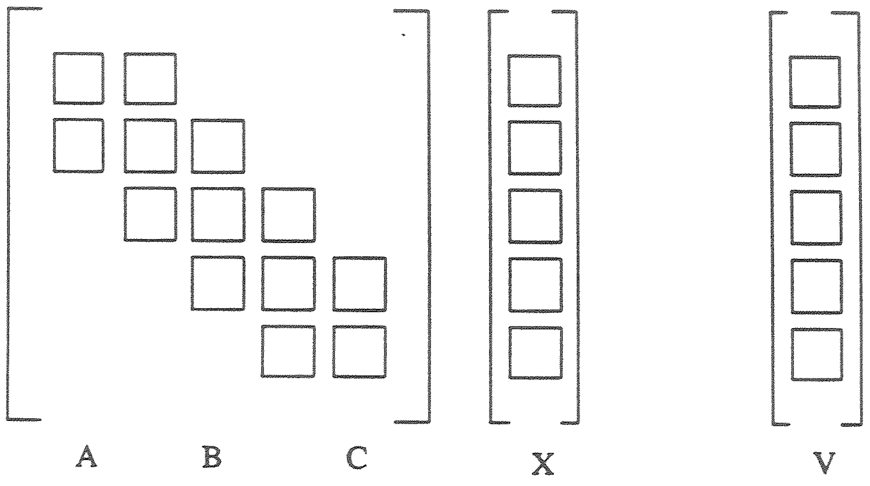


Figure 7-14: A Block Tridiagonal System

as shown in Figure 7-14. One solution to this system is called the Odd Even Elimination technique [HEL77] and can be described as follows: First, select three consecutive block equations.

$$
\begin{array}{llllll}
a_{k-1}x_{k-2} + & b_{k-1}x_{k-1} + & c_{k-1}x_{k} & = & v_{k-1} & (k-1) \\
a_{k}x_{k-1} + & b_{k}x_{k} + & c_{k}x_{k+1} & = & v_{k} & (k) \\
a_{k+1}x_{k} + & b_{k+1}x_{k+1} + & c_{k+1}x_{k+2} & = & v_{k+1} & (k+1)
\end{array}
$$

Multiplying equation (k-1) by $-a_k b_{k-1}^{-1}$ and equation (k+1) by $-c_k b_{k+1}^{-1}$ and add, the resulting equation can be seen to have only factors with x_{k-2}, x_k and x_{k+2}. In other words, the resulting system still has only three non zero block diagonals, but they are now further apart. This process is repeated until only one block diagonal remains, and the system can be easily solved. The algorithm can be expressed in the following manner [KAP82]:

1. Solve for b(k) [a'(k)c'(k)v'(k)] = [a(k)c(k)v(k)]

2. b(k).2 ← b(k).1 - a(k).1c'(k-1) - c(k).1a'(k+1)

3. v(k).2 ← v(k).1 - a(k).1v'(k-1) - c(k).1v'(k+1)

4. a(k).2 ← -a(k).1a'(k-1)

5. $c(k).2 \leftarrow -c(k).1c'(k+1)$

The algorithm modelled here consists of the following tasks:

Init(i)	Initializes the ith row. In this algorithm, a, b, c, v, a', c' and v' must all be in shared memory. However, a', c' and v' need not be initialized.
LU(i)	Does the LU decomposition for b(i). This is then used to solve for a'(i), c'(i) and v'(i) in (1) above.
Oe(i)	For the ith row, uses a'(k-1), a'(k+1), c'(k-1), c'(k+1), v'(k-1) and v'(k+1) to evaluate the new values of a, b, c, and v. Since each Oe needs values from each of its neighboring rows, and updates its own row, the algorithm imposes a synchronization by first executing Oe(i) for odd i, and subsequently for even i.
Swap(i)	The swaps do an inverse perfect shuffle of the rows, thus giving rise to two tridiagonal systems of half the size.
Solve(i)	After log(n) iterations (where n is the initial number of block rows, and the logarithm is to base 2), the system is reduced to a single block diagonal. This can then be solved using the Solve tasks.

7.3.1 Model for the Tridiagonal System

The PCM model for the Tridiagonal System is described next. Only a brief description is given, since it is regular in structure, and merely carries out the operations specified earlier for each task. The topmost level of the model (Figure 7-15) is simply a loop which iterates k times (where k is log of the number diagonal blocks), finally reducing the system to a block diagonal system which can then be solved. The state variables "g" and "s" trace the number of groups and the size of each group, respectively. For each iteration, subnet Tridia is initiated once, while after the final iteration, Solve is called to obtain the final solution.

Since subnet Tridia (Figure 7-16) is parameterized with "g", the number of groups, each initiation causes twice as many streams. The index "s" is used within the

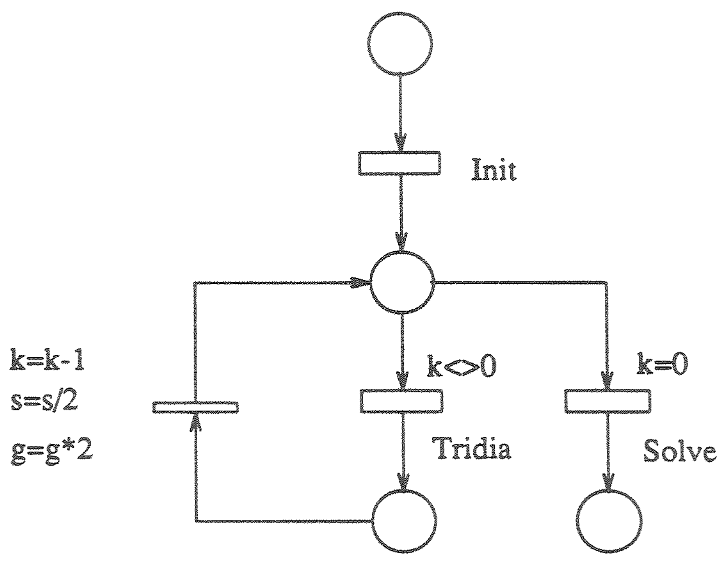


Figure 7-15: Top Level of Tridiagonal Solution

subnets comprising Tridia (Lu, Oe and Swap), which leads to the binary tree-like computation structure. The reason two Oe(odd) subnets are required is that the first row in the system (initially, row 1) does not need to access the previous row (row zero), and hence can evaluate its functions with only rows i and i+1. Similarly, the Oe(even) tasks have two copies since the last row (say, i), does not need a next row (i+1). Finally, the swap merely copies the contents of one memory into another, to achieve an inverse perfect shuffle.

Since all the subnets are essentially the same, only one, Lu(i,j), is described. Note that the indices i and j specify the group number and position within the group respectively. Lu_load, Lu_ready, and Lu_done have the same interpretations as in the BLT case. Once the process is loaded, it executes in three phases:

- accesses b_{i*s+j} and evaluates its LU decomposition
- evaluates the values of a', b' and c' for the same row (i.e. index i*s+j), and writes them back to remote memory.

The configuration parameters varied in this study are the following:

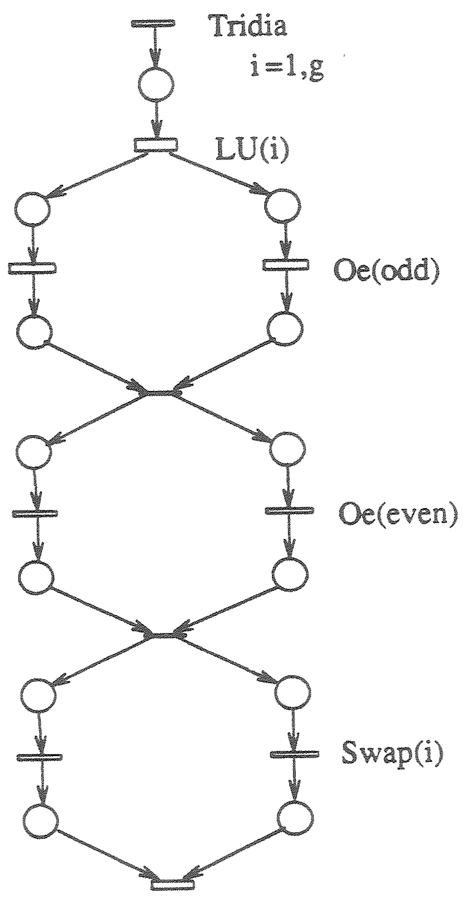


Figure 7-16: Model for Tridiagonal Subnet

- number of buses for the multibus architecture (examples use 1 or 2 buses),
- the time delay for the access of each floating point word (examples use 0.5 or 1.0 μsec),
- size of the computation, keeping the number of partitions fixed (i.e. there are always 16 diagonal blocks, but the size of each block is varied from 2 to 16)
- remote accesses with and without copying.

The decision of whether to copy or not has a large impact on the performance of a

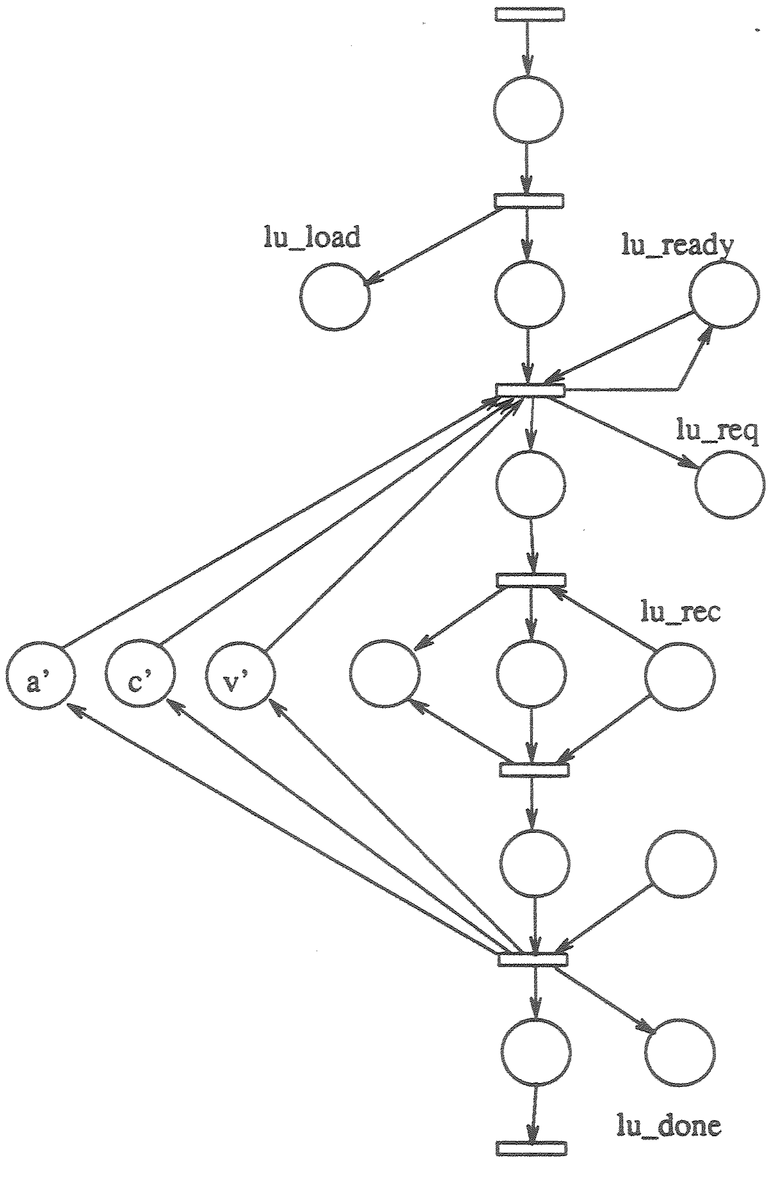


Figure 7-17: Model for Lu Subnet

computation, due to the fact that remote access are more expensive, as well as the fact that they contribute towards the bus (or network, in the case of the banyan) congestion. Another critical factor is the complexity of the computation itself in terms of the number of remote accesses. If the number of accesses grows rapidly, it may be better to first copy the entire object into local store, and *then* begin the computing. Table 7-2 shows the effects of the various options on the overall computation time.

$\tau_{ra} = 0.5$				
Size of block	2	4	8	16
1 bus, no copy	40.0	132.0	756.5	5317.9
2 bus, no copy	39.8	131.8	755.8	5317.7
1 bus, copy	41.8	130.2	686.6	4557.7
2 bus, copy	41.6	130.1	686.5	4557.5

$\tau_{ra} = 1.0$				
Size of block	2	4	8	16
1 bus, no copy	42.8	137.1	781.6	5483.0
2 bus, no copy	40.5	135.1	776.8	5467.0
1 bus, copy	43.3	134.1	701.1	4612.2
2 bus, copy	42.1	131.2	690.5	4573.0

Table 7-2: Tridiagonal Simulation Results

From the table, it can be seen that the architecture is being operated well below its capacity. The addition of a second bus does not effect the performance, since the first bus itself is underutilized. Since the computation being modelled consists mainly of matrix operations ($O(n^3)$), copying quickly becomes an attractive alternative to all the extra bus accesses. In all the cases, for block sizes larger than 2, it is beneficial to use copying.

7.4 Conclusions

The studies presented in this Chapter have demonstrated that PCM models which are compact and easy to develop can return accurate evaluations of the impact on performance of fairly subtle variations in computation structure and architecture. The validation of the model across a spectrum of task granularities was very satisfactory. The methodology was demonstrated to fulfil its claims of facilitating variations in model structure and isolation of cause and effect relationships.

Chapter 8

Conclusions

The model proposed in this dissertation is an extended form of Standard Petri Nets which allows the representation and performance evaluation of the *execution behavior* of parallel computations on given architectures. One enhancement in the PCM model over other models is the introduction of hierarchy and parameterization into the model. Parameterization was shown to be of immense value when modelling parallel computations and architectures. The representation of the entire switching interconnection network, for example, was reduced to the specification of just one switch, properly parameterized to reflect its nearest neighbors. The models of the computations, too, required the specification of only one process of each kind, with proper parameterization accounting for the rest. This has the added advantage that only the basic structure or pattern of the computation or architecture need be captured while building the model.

The introduction of a systematic hierarchy allows development of a methodology for building models. The methodology allows the systematic development of correct models, and makes the technique extremely flexible. Once a computation has been modelled, factors such as the underlying architecture and the mapping between the computation and the architecture can be easily modified. The library of architectures proposed can be extended to include shared memory architectures as well as message based, or distributed systems. This would require a redefinition of the mapping interface between a process and a processor to include the concept of sending a message to another processor. The library concept can also be extended to include abstract computation structures.

The model has the advantage over other analytical models in that it can simulate the behavior of large, complex systems, since it does not have to enumerate the entire state space. However, as the nets begin to get large, the memory requirement of the model also increases rapidly, eventually becoming a bottleneck. One solution to this would be to attempt some type of reduction of the nodes in the net by clustering several nodes if the clustering does not effect the flow of control, and by assigning the aggregate transition a new delay representing the entire cluster.

The example studies here have shown both the convenience of using PCSIM and the capability of PCM for accurately capturing small variations in both logical computation structures and architectures.

There are several streams of future research. The most immediate and important is to conduct a series of model studies of the impact of variations in computation structures and architecture on widely used algorithms from several disciplines. Each study will be grounded in validation on some real architecture. There are also further possible enhancements that may further enhance the representation convenience of PCM. Addition of types of tokens is one such enhancement, a third area of research is efficiency of execution. Model formulation strategies which compact the state representation by masking out unnecessary events and utilizing hierarchy to reduce execution time will be studied. Finally, a parallel structuring of PCSIM, itself, will be formulated and studied.

Appendix A.

Equivalence of the PCM Model to Petri Nets with Inhibitor Arcs

The modelling power of Standard Petri Nets is known to be less than that of Turing machines due to their inability to test for the absence of tokens in a place [PET81]. It has been shown that Petri Nets with inhibitor arcs are equivalent in their modelling power to Turing machines [AGE74]. In this Section, the PCM model is shown to be equivalent to Petri Nets with Inhibitor arcs (IPNs). The equivalence is proven by showing that for any instance of and IPN, there exists an equivalent PCM instance.

A.1 Definition of the IPN model

The IPN model is the Standard Petri Net with the addition of *inhibitor* arcs which are used to test for the absence of tokens at a place. The IPN model is defined as:

```
IPN = (P, T, I, J, O, M), a Petri Net, where
  P = {p_1,..,p_n}, a set of places,n ≥ 0
  T = {t_1,..,t_m}, a set of transitions,m ≥ 0
  I is the transition input function, I : T ---> 2^n,
  or, I is a subset of PxT;
  J is the transition inhibitor function, J : T ---> 2^n,
  or, J is a subset of PxT;
  O is the transition output function, O : T ---> 2^n,
  or, O is a subset of TxP;
  M = [μ_1,...,μ_n], a vector of integers specifying
      the initial marking of the net;
  and the sets P and T are disjoint
```

The main difference between this definition and that of standard Petri Nets is the addition of inhibitor arcs specified by J. A transition is now enabled when all its *input* places contain tokens, and all its *inhibitor* places contain NO tokens. More formally, the dynamic behavior of an IPN can be stated as follows:

- transition t_i is enabled iff:

$$\forall p_j \in I(t_i),\ \mu_j > 0$$
and
$$\forall p_j \in J(t_i),\ \mu_j = 0$$

- at any instant, an enabled transition can be selected and fired
- firing a transition, t_i, is instantaneous and has the following effect:

$$\forall p_j \in I(t_i),\ \mu_j \leftarrow \mu_j - 1$$
$$\forall p_j \in O(t_i),\ \mu_j \leftarrow \mu_j + 1$$

A.2 Conversion from IPN to PCM

Given an IPN = (P, T, I, J, O, M), an equivalent PCM is constructed.

Define:

$$J_i = \{p_a \mid p_a \in J(t_i)\}\ 0 \le i \le m$$

$$J = \cup_{i=0,m} J_i$$

$$II_i = \{p_a \mid p_a \in J \wedge p_a \in I(t_i)\}\ 0 \le i \le m$$

$$IO_i = \{p_a \mid p_a \in J \wedge p_a \in O(t_i)\}\ 0 \le i \le m$$

For each transition t_i, J_i specifies the set of places connected to it by inhibitor arcs. J is the union of all J_is. For a transition t_i, II_i specifies the subset of its input places which are inhibitor places for some transition (i.e. which belong to J). Similarly, IO_i specifies the subset of the output places of transition t_i which are inhibitor places for some transition.

The equivalent PCM is defined as:

```
   PCM = <PN, SV, TA>

where
   PN = (P,T,I,O,M)

   SV = <V, IV>
```

```
    where
      $V = \{ v_{p_i} \mid p_i \in J \}$
and $IV = \{ iv_{p_i}$, the initial value of $v_{p_i}$, $= \mu_i \}$

  $TA = <\Pi, \Phi>$
    where
      $\pi_j(V) = \wedge(v_{p_i} = 0), \forall p_i \in J_i$

      $\phi_j(V)$ = sequence of assignments:
            $v_{p_i} = v_{p_i} - 1 \; \forall \; p_i \in II_j$
            $v_{p_i} = v_{p_i} + 1 \; \forall \; p_i \in IO_j$
```

In the equivalent PCM model, the basic Petri Net (PN) is simply the original IPN without the inhibitor arcs. Since the only way of testing a place for zero tokens in the PCM model is by using a state variable, each place P_i in the IPN which is an inhibitor place for any transition has a state variable (v_{p_i}) in the PCM, which counts the number of tokens in the place. The transition predicates are defined so that whenever an IPN transition has an inhibitor place, the corresponding PCM transition's predicate checks that the associated variable is equal to zero. Similarly, if an IPN transition either removes or adds tokens to a place which is an inhibitor for some transition, the PCM transition procedure correctly updates the corresponding state variable.

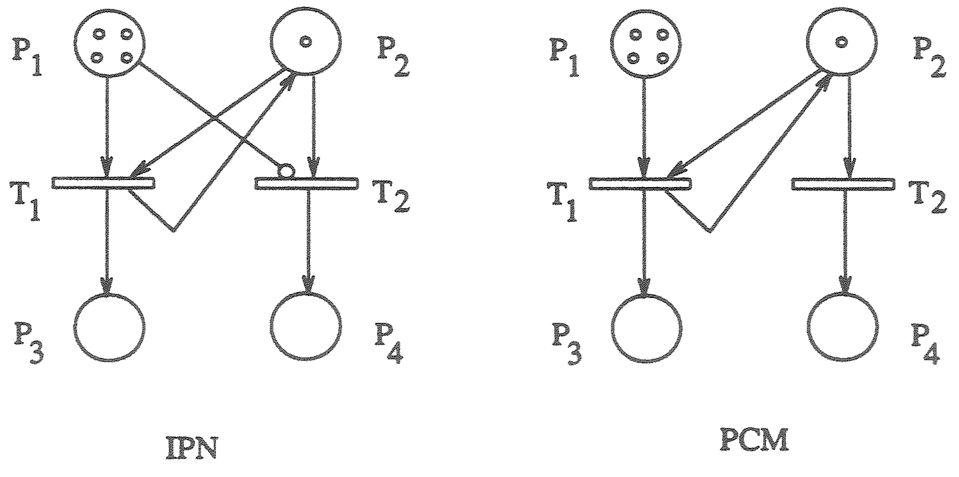


Figure A-1: An Example

The construction is demonstrated by means of an example (Figure A-1). The IPN shown graphically in the figure corresponds to:

```
      IPN = (P,T,I,J,O,M)
where
      P = {p1, p2, p3, p4}
      T = {t1, t2}
      I(t1) = {p1, p2}  J(t1) = {∅}
      I(t2) = {p2}      J(t2) = {p1}
      O(t1) = {p2, p3}  O(t2) = {p4}
      M = [4,1]
```

In the equivalent PCM model, the only state variable is v_{p_1}.

```
      PCM = <PN, SV, TA>
where
      PN = (P,T,I,O,M), as for the IPN

      SV = <V, IV>
        V = {v_p1}, IV = {iv_p1 = 4}

      TA = <Π, Φ>
        π1 = True,  π2 = (v_p1 = 0)
        φ1 = (v_p1 = v_p1 - 1)
```

It has been shown that the PCM model can represent any IPN net. To complete the proof of equivalence between the PCM and IPN nets, the reverse transformation (from PCM to IPN) must be shown. The proof is based on representing all integer state variables as places, and their values as tokens within the places. Each PCM transition is now replaced by IPN fragments which simulate the transition predicate and procedure. The existence of such an equivalent IPN fragment for a PCM predicate (or procedure) is proved by providing equivalent IPN structures for each grammar rule of the predicate (or procedure), and chaining them together to obtain the IPN fragment equivalent to the entire predicate. The details of the transformation can be found in [ADI88]. Thus, the PCM model is equivalent to the IPN model (and, therefore, to the Turing Machine Model).

Appendix B.

Modelling high level languages

This Appendix outlines a method for translating programs written in high level languages into instances of the PCM model. The two languages selected here are the Computation Structures Language (CSL, [BRO82]) and the Task-level Data Flow Language (TDFL) [SUH87], both of which are experimental languages developed at the University of Texas. The choice of these languages is appropriate since they belong to the two different approaches to parallel languages - CSL is a language that expresses explicit parallelism, while TDFL is a typical dataflow language which expresses implicit parallelism with procedure level units of computation. Since implementations of these systems exist, it becomes possible to validate the results obtained from the model against actual executions of the computations. In the following sections, each language is discussed separately, with details of how different constructs in the languages are represented in the model.

B.1 Representation of CSL Constructs

The Computation Structures Language is a language that allows the specification and programming of multitype, multiphase computations It supports dynamic structuring of computations through multiple phases, each of which may display different types and degrees of parallelism and differing requirements of data and interprocess communication. CSL was originally designed for the Texas Reconfigurable Array Computer (TRAC) [SEJ80], but has since been implemented on alternate architectures.

CSL is a block structured language with several Pascal-like constructs, with constructs to specify synchronization and communication, in addition to typical sequential constructs such as branching and iteration. The basic computation unit in CSL is a *task*, which is a sequential unit of code that is encoded using a standard high level language (like Pascal or C). The state of the CSL program includes only the information necessary for control of the computation. A CSL program specifies only the synchronization and communication between these tasks, while the actual task bodies are specified in an ordinary high level language. In this section, some of the main CSL constructs are introduced and their equivalent model instance is presented.

B.1.1 CSL Variables

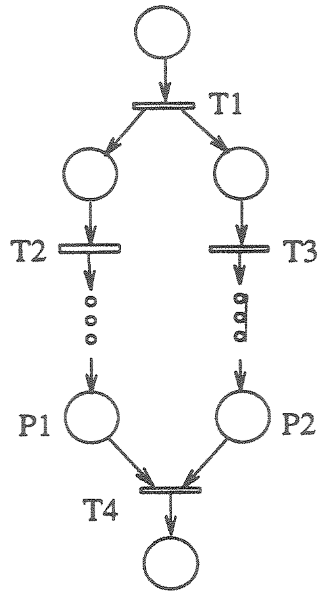
CSL job variables can be of three types - integer, boolean and condition. All integer and boolean variables are represented as state variables and form the vector V defined previously. These variables can occur in transition predicates, and can be modified by transition procedures. Since condition variables behave like binary semaphores, it seems more natural to represent them as places. Each condition is modelled by two places. A token in one place means the condition is 'on', while a token in the other place indicates the condition is 'off'. These conditions are used in the WAIT and SIGNAL synchronization primitives described later.

B.1.2 The CONSTRUCT statement

The CONSTRUCT statement is used to define the resources used in a segment of a CSL program. In particular, the types of resources declared are the tasks to be executed, the variables they share among themselves, and channels through which they may exchange messages. Each shared variable is modelled by a single place with one initial token. Channel declarations are specified with buffer sizes, which indicate how many outstanding messages are allowed between any two tasks. Each channel is modelled by two places- one indicating the number of free buffers, and the other indicating the number of buffers used. These places are described further in the section on Communication.

B.1.3 Execution of tasks

The EXECUTE statement is used to initiate execution of a task. Task executions are modelled by transitions, where the transition delay represents the task execution time. As shown in Figure B-1, parallel execution of tasks is achieved using a COBEGIN-COEND pair with any number of parallel streams contained within. Transition T1 models the COBEGIN, and forks two parallel streams. both streams are complete, tokens arrive in P1 and P2, and T4 fires. Control does not pass beyond the



```
COBEGIN
   // EXECUTE (Task1);
          o
          o
          o
   // EXECUTE (Task2);
          o
          o
          o
COEND;
```

Figure B-1: CSL EXECUTE and COBEGIN Statements

COEND until all streams have completed execution. After both streams are complete, tokens arrive in P1 and P2, and T4 fires.

B.1.4 Iteration and Branching

CSL provides most of the common branching and iteration constructs like IF-THEN-ELSE, FOR-DO and REPEAT-UNTIL. Figure B-2 shows the IF statement and its equivalent representation. All variables in *cond* are modelled by state variables. When a token is present in P1, if *cond* evaluates to true, transition T1's predicate is true, and T1 becomes enabled. Similarly, if cond is false, T2 becomes enabled. The FOR statement (Figure B-3) is modelled by having a special transition (T4) which increments the loop variable (i) when it fires. Transition T1 initializes the loop variable, while T2 and T3 decide if the loop has completed.

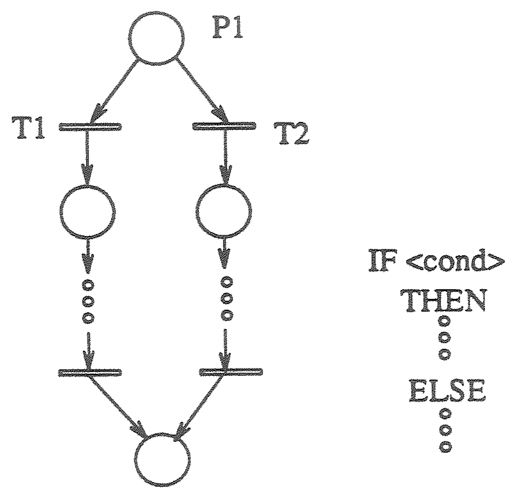


Figure B-2: CSL IF Statement

B.1.5 Synchronization

CSL provides synchronization primitives based on Dijkstra's semaphores. The two most constructs are the WAIT and SIGNAL statements, both of which operate on CSL condition variables. Figure B-4 shows how these statements can be expressed very naturally using Petri Nets. In addition to the CSL condition variables, each task has a condition (boolean) variable which can be set from within the task body. C and C' are used to model the condition. The statement WAIT(C) is modelled by transition T1 awaiting a token in C. Since SIGNAL(C) is non-blocking, it must be modelled by two transitions T2 and T3. If C is already set, T2 fires, resulting in no change in the system. If C is not set, a token is present in C', and T3 fires. In both cases, after the SIGNAL, C becomes set.

B.1.6 Communication

Tasks in CSL can communicate either by sending messages or by means of shared variables. Messages can be sent between tasks on a channel. Figure B-5 shows how message channels are modelled. A SEND can occur only when an empty buffer is available (place buff_avail has a token), while a RECEIVE would cause the receiving

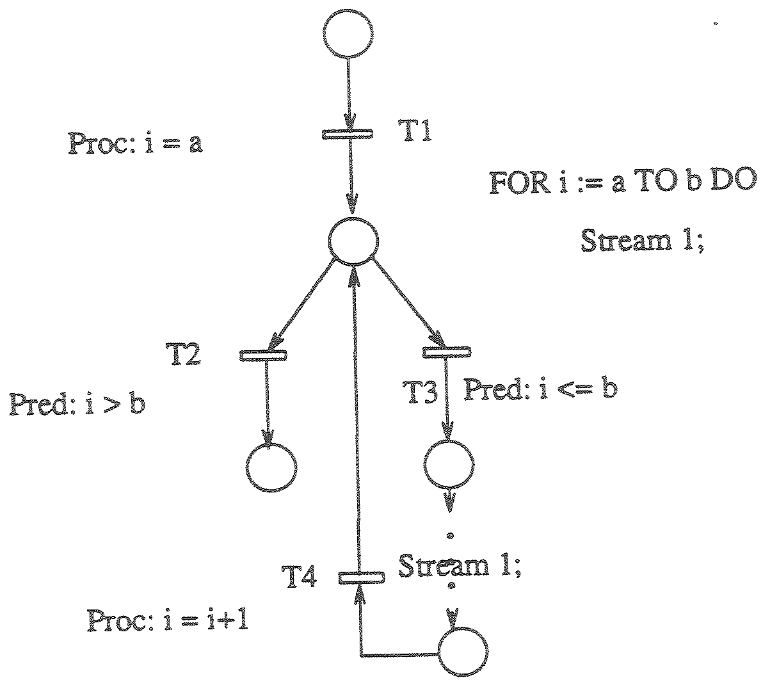


Figure B-3: CSL FOR Statement

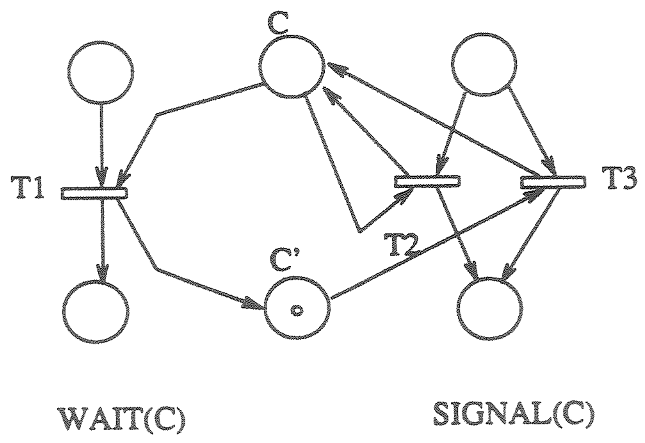


Figure B-4: CSL WAIT and SIGNAL Statements

task to wait for a token in place msg_avail. Depending on the underlying architecture,

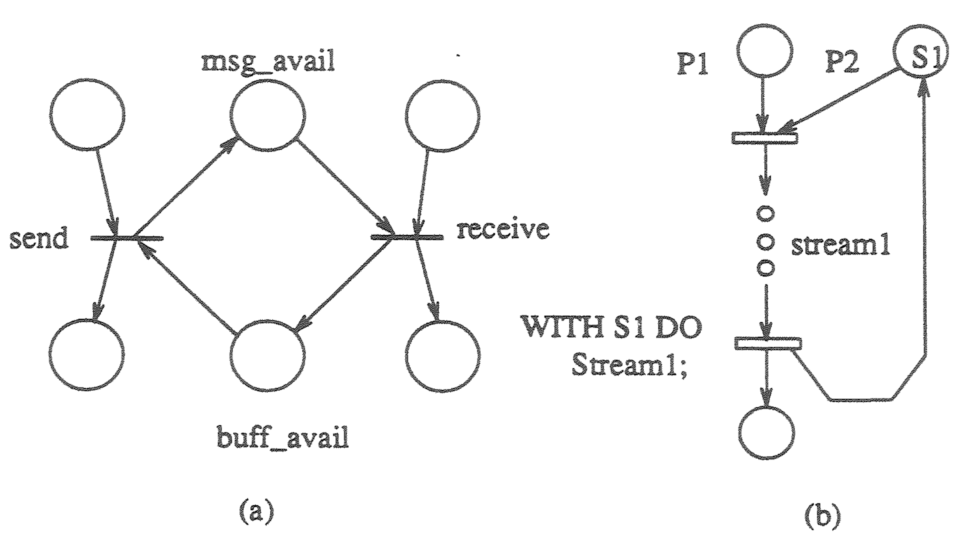


Figure B-5: CSL SEND, RECEIVE and WITH Statements

the delay at the send transition would model the message delay. Since message delays can be functions of the state variables, it is possible to specify load dependent message delays. Further, if desired, transition send may be replaced by a subnet which models the underlying architecture in greater detail.

Figure B-5 shows the basic synchronization primitive: the WITH-DO statement. Shared variables can be accessed either in exclusive (write) mode, or in non-exclusive (read only) mode. Since shared variables are modelled by single tokens, exclusive access is easily modelled by holding the token for the duration of the execution of the task. In most cases, the underlying architecture enforces exclusive access to shared memory (as in TRAC), and non-exclusive access is modelled by accessing the variable in exclusive mode, copying its contents into local memory, and releasing the exclusive 'lock'. If the architecture permits non-exclusive access, state variables have to be introduced as shown in Figure B-6. Here, the state variables (R & W) track the number of active readers and writers at all times. The writers are guaranteed exclusive access by removing the 'shared' token (in P3) as before. The readers, however, increment the 'read-count' to indicate that they are using the variable. No writer can gain

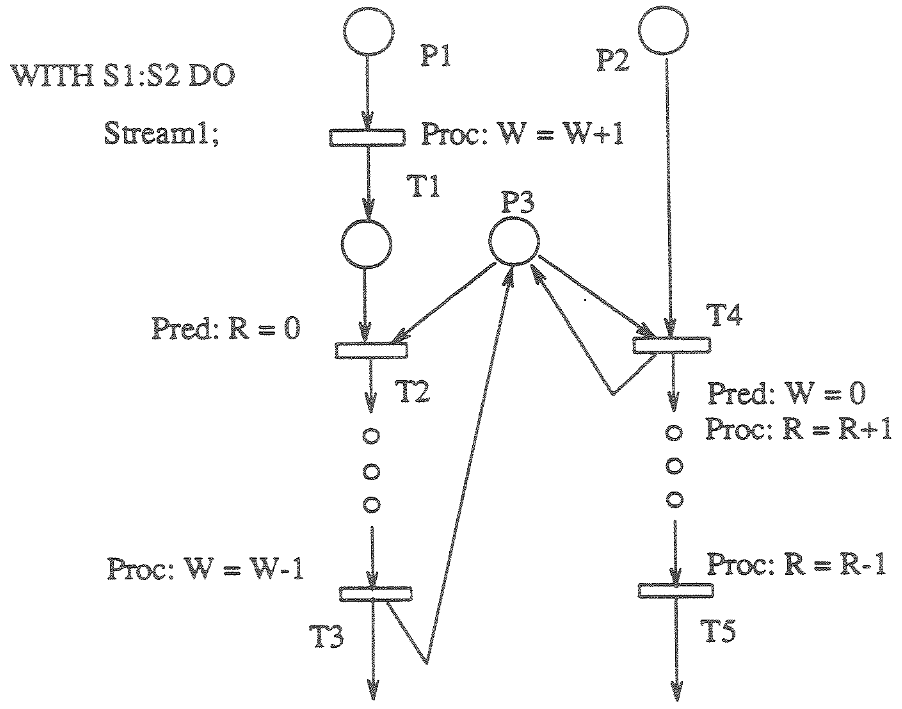


Figure B-6: Shared Access with Readers and Writers

access to the shared variable until all readers have completed (R = 0). Transition T1 enforces priority for the writers by incrementing W.

Given a CSL program, it is possible to automatically generate its equivalent net. The constructs for each statement are concatenated by overlapping the 'output' place of one construct with the 'input' place of the next. This method leads to a net containing many unnecessary places and transitions. Automatic generation of the net, however, guarantees that the net faithfully reproduces the CSL program. It also provides an excellent initial model for further refinement. This approach of providing equivalent Petri Net nets for each high level construct of a language has also been reported in [SHA85], where the technique is used to detect communication patterns in ADA programs.

B.2 Task-level Data Flow Language

TDFL is a coarse-grain data-driven data flow language [SUH87]. The nodes of the data flow graph represent functions written in C or Pascal, and are equivalent to CSL tasks. The execution of the node function is triggered by the presence of tokens on its input arcs and after a node function completes, tokens are placed on its output arcs. TDFL consists of six different types of nodes which have different rules for triggering execution, as well as removal and placement of input and output tokens. All global state information is carried in the tokens, and some of the nodes can save internal state between executions.

The general representation of TDFL graphs (programs) in the PCM model is as follows. TDFL nodes are represented by transitions in the model, with the transition delay being the time taken by the TDFL node function to execute. TDFL arcs are naturally modelled by places, since they represent the entire state of the system. The rest of this section describes each of the six types of TDFL nodes, and presents an equivalent representation in the model.

B.2.1 General Node

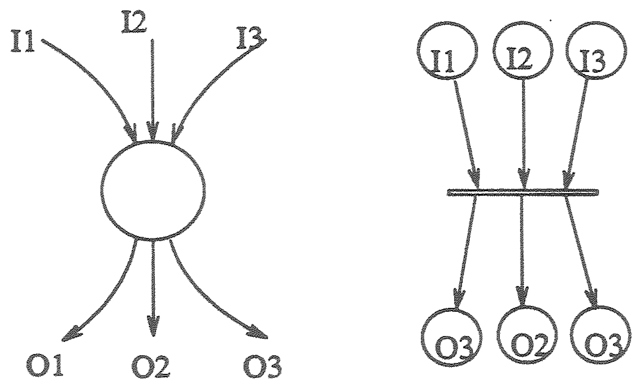


Figure B-7: A TDFL General Node

A general node has an arbitrary number of input arcs and an arbitrary number of output arcs. An output arc of a general node may coincide with an input arc of the same node. This looping arc denotes the retention of state within the node. A general node fires when it has a token on *each* of its input arcs. It removes one token from each node, executes its node function, and places one token on each output arc. The definition of the general node coincides with the firing rule for firing Petri net transitions, and the equivalent representation in the PCM model is simply a single transition (Figure B-7).

B.2.2 Merge Node

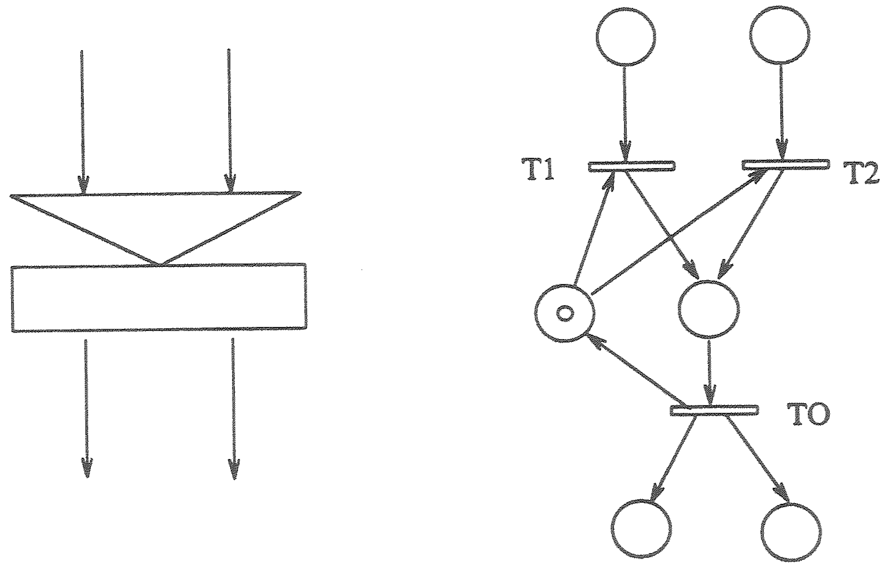


Figure B-8: A TDFL Merge Node

The merge node fires whenever a token is available on *any* input arc. Upon firing, the merge node removes exactly one input token, executes its node function, and places one token on each output arc. If more than one input arc has a token, the arc is selected nondeterministically. Figure B-8 shows the equivalent PCM representation, where T1 and T2 model the selection of the input token, while TO introduces a delay equal to the node function execution time, and then places tokens on all outputs. P1 prevents the node from firing twice simultaneously.

B.2.3 Do-all Node

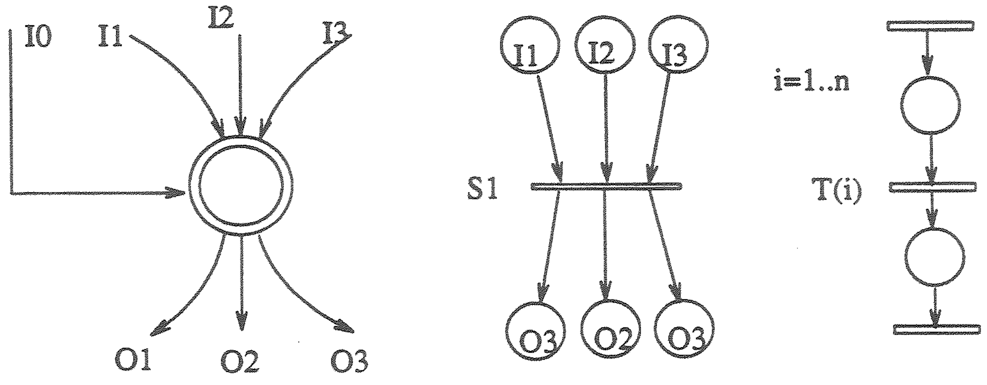


Figure B-9: A TDFL Do-all Node

The do-all node is a special case of the general node in that it has an extra input arc that delivers a non-negative integer with each firing. The value of this integer determines the number of parallel executions of of the node function. Each execution uses the same set of input tokens, and when all executions are complete, one token is placed on each output arc. This is modelled using a state variable (n) which contains the number of executions desired. Using this variable as a subnet parameter (for subnet S1) causes n replications of transition T(i) to fire, as shown in Figure B-9.

B.2.4 Loop Node

A loop node (Figure B-10) has two pairs of arcs called the main (input and output) and feedback (input and output) arcs. It has a boolean predicate that can be evaluated taking as argument a token that arrives on an input arc. This node also holds internal state information telling it whether to accept a token from the main input (MI) or the feedback input (FI). Initially, a token is accepted from MI, and, depending on the boolean predicate, a token is placed on either MO or FO. If FO was selected, the

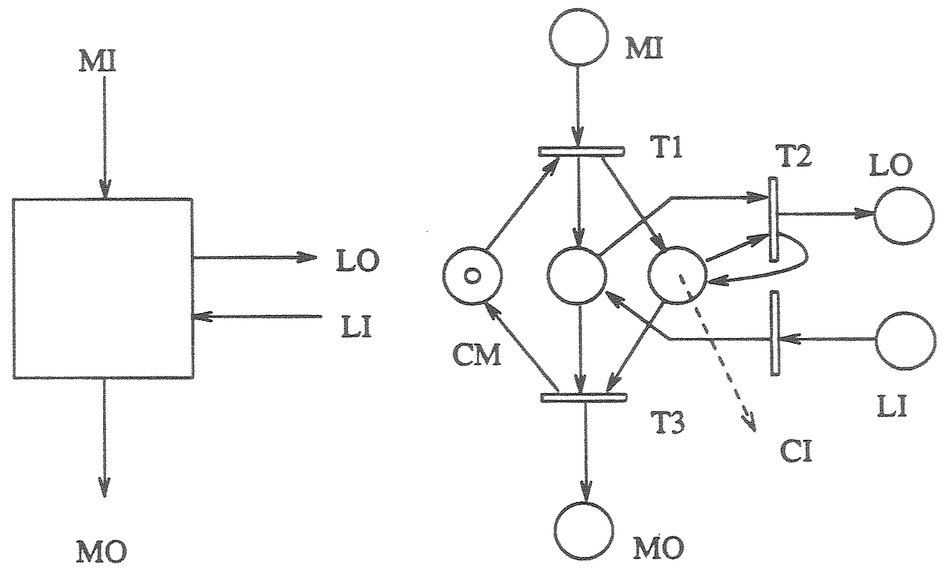


Figure B-10: A TDFL Loop Node

node now awaits the return of a token on the feedback input arc (FI), else it waits for MI. The equivalent PCM fragment has places (MI, MO, FI and FO) corresponding to the four arcs. In addition, the internal state is represented by two places CM and CI, with a token present initially in CM. The boolean predicate is modelled by the two transition predicates T2 and T3 which are complements of each other.

B.2.5 Case and EndCase Nodes

The Case and Endcase nodes are described together, since they perform complementary operations. The Case node takes all input tokens and places a token on *one* output arc (Figure B-11. In addition, a token is placed on a special 'control' arc specifying which output arc received the token. The EndCase node receives this 'control' token, and waits to receive a token on the corresponding input arc. Upon receiving this token, it places tokens on all output arcs. The control token arc ensures that tokens enter and leave the case-endcase pair in the same order. Since tokens in the Petri net model are not typed, the state information required is a queue, specifying the order in which tokens enter the case node. The transition predicates of the first set of transitions (T1(1), T1(2), etc.) are set up so that exactly one of them is enabled. Upon

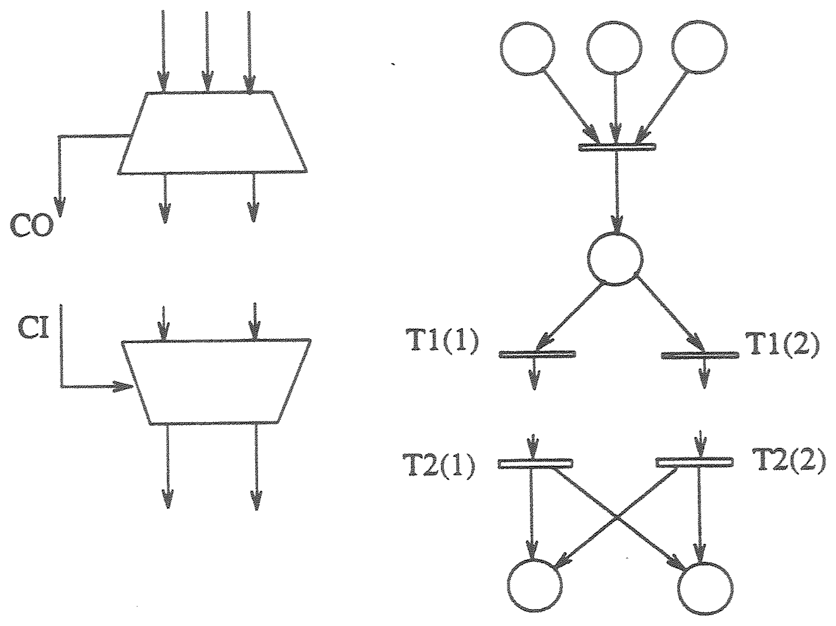


Figure B-11: TDFL Case and Endcase Nodes

firing, the selected transition's procedure enqueues the value of its index. Similarly, the transition predicates of transitions T2(i) are true if their index matches the index at the front of the queue, and upon firing, the procedure causes the index to be removed.

Bibliography

[ADI86]	Adiga, A.K, & Browne, J.C., "A Graph Model for Parallel Computations Expressed in the Computation Structures Language", *Proceedings of the 1986 Intl. Conf. on Parallel Processing*, August 1986, pp.880-887.
[ADI87a]	Adiga, A.K., & Deshpande, S.R., "Evaluation of Effectiveness of Circuit Based and Packet Based Interconnection Networks via Petri Net Models", *Proceedings of the 1987 Intl. Conf. on Parallel Processing*, August 1987, pp.533-542.
[ADI87b]	Adiga, A.K., & Browne, J.C., "Performance Modelling of Parallel Programs", *working paper*, Department of Computer Science, University of Texas, Austin, Texas 78712, January, 1987.
[ADI88]	Adiga, A.K., "Equivalence of the PCM Model to Petri Nets with Inhibitor Arcs", *Technical Report*, Department of Computer Science, University of Texas, Austin, Texas 78712.
[AGE74]	Agerwala, Tilak, "A Complete Model for Representing the Coordination of Asynchronous Processes", *Hopkins Computer Research Report #32*, Computer Science Program, The Johns Hopkins University, Baltimore, Maryland, July 1974.
[BAS75]	Baskett, F., Chandy, K.M., Muntz, R.R., & Palacios, F., "Open, Closed and Mixed Networks of Queues with Different Classes of Customers", *Journal of the ACM*, vol. 22, no. 2, April 1975, pp.248-260.
[BER79]	Berlin, F.B., "Time Extended Petri Nets", *Masters Thesis*, Department of Computer Science, The University of Texas at Austin, August 1979, 152 pages.

[BRO82] Browne, J.C. et al., "A Language for Specification and Programming of Reconfigurable Parallel Computation Structures", *Proceedings of the 1982 International Conference on Parallel Processing*, August 1982.

[BRO85] Browne, J. C., "Formulation and Programming of Parallel Computations: A Unified Approach", *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985, pp.624-631.

[BRO87] Browne, J.C., & Adiga, A.K., "Graph Structured Performance Models", *Performance Modeling of Supercomputers*, J.L.Martin ed. North Holland, in preparation.

[CRO85] Crowther, W., et. al., "The Butterfly^{TM} Parallel Processor", *IEEE Computer Architecture Technical Committee Newsletter*, September 1985, pp. 18-45.

[DEN72] Dennis, J.B., Fosseen, J.B., & Linderman, J.P., "Dataflow Schemas", *Theoretical Programming*, Springer-Verlag, Berlin, 1972, pp.187-216.

[DUG85] Dugan, J. B., Bobbio, A., Ciardo, G. & Trivedi, K.,"The Design of a Unified Package for the Solution of Stochastic Petri Net Models", *Proceedings of the International Workshop on Timed Petri Nets*, Torino, Italy, July 1-3, 1985, pp.6-13.

[EFR64] Efron, R., & Gordon, G., "A General Purpose Digital Simulator and Examples of its Application, Part I: Description of the Simulator", *IBM System Journal*, vol 3, no. 1, 1964, pp.22-34.

[EST86] Estrin, G., et. al., "SARA (System Architects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems", *IEEE Trans. on Software Engineering*, Vol. SE-12, No. 2, February 1986.

[GEN81] Genrich, H.J., & Lautenbach, K., "System Modelling with High-Level Petri Nets", *Theoretical Computer Science*, North-Holland, vol 13, 1981, pp.109-136.

[GOK73] Goke, G.R. & Lipovski, G.J., "Banyan Networks for Partitioning Multiprocessor Systems", *1st Annual Symposium on Computer Architecture*, December 1973, pp. 21-28.

[HEL77] Heller, D., "Direct and Iterative Methods for Block Tridiagonal Linear Systems", *PhD. Dissertation*, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1977.

[HOL85] Holliday, M. A. & Vernon, M. K., "A Generalized Timed Petri Net Model for Performance Analysis", *Computer Sciences Technical Report #593*, Computer Sciences Department, University of Wisconsin-Madison, May 1985.

[HOL87] Holliday, M.A. & Vernon, M.K., "Exact Performance Estimates for Multiprocessor and Bus Interference", *IEEE Transactions on Computers*, January 1987, Vol. C-36 No. 1, pp.76-85.

[IRA86] "PAWS: Introduction and Technical Summary", Information Research Associates, Austin, Texas, 1986.

[JEN82] Jenevein, R.M. & Browne, J.C., "A Control Processor for a Reconfigurable Array Processor", *Proc. of the 9th Symposium on Computer Architecture*, Silver Spring, MD, 1982, pp. 81-89.

[JON77] Jones, N.D., Landweber, L.H. & Lien, E.Y., Complexity of some Problems in Petri Nets", *Theoretical Computer Science*, North-Holland, vol 4, 1977, pp.277-299.

[KAP82] Kapur, R.N., "On the Synthesis and Analysis of Reconfigurable Computer Programs", *PhD. Dissertation*, Department of Electrical Engineering, University of Texas at Austin, May 1982.

[KAR66] Karp, R.M. & Miller, R.E., "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing", *SIAM Journal of Applied Math*, vol.14, No.6, November 1966, pp.1390-1411.

[KAR69] Karp, R. M., & Miller, R. E., "Parallel Program Schemata", *Journal of Computer and System Sciences*, pp. 147-195, 1969.

[KEL74] Keller, R., "Vector Replacement Systems: A Formalism for Modelling Asynchronous Systems", *TR-117*, CS Laboratory, Princeton University, Princeton, New Jersey, December 1972, Revised January 1974.

[KEL76] Keller, R., "Formal Verification of Parallel Programs", *Communications of the ACM*, Vol. 17, No. 7, July 1976, pp.371-384.

[KLE75] Kleinrock, L., "Queueing Systems Vol I: Theory", Publishers *John Wiley and Sons*, New York, 1975.

[KOS73] Kosaraju, S., "Limitations of Dijkstra's Semaphore Primitives and Petri Nets", *Operating Systems Review*, Vol. 7, No. 4, pp. 122-126, October 1973.

[LAW82] Law, A.M., & Kelton, W.D., "Simulation Modeling and Analysis", McGraw Hill, 1982, 400 pages.

[MAR84] Marsan, A.M., Balbo, G., & Conte, G., "A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems", *ACM Transactions on Computer Systems*, May 1984, pp.93-122.

[MAR86] Marsan, M.A., Balbo, G. & Conte, G., "Performance Models of Multiprocessor Systems", *MIT Press Series in Computer Systems*, The MIT Press, Cambridge, Massachusetts, 1986, 281 pages.

[MOL81] Molloy, M.K., "On the Integration of Delay and Throughput Measures in Distributed Processing Models", *PhD. Dissertation*, Department of Computer Science, University of California at Los Angeles, September 1981.

[MOL82] Molloy, M. K., "Performance Analysis using Stochastic Petri Nets", *IEEE Transactions on Computers* Vol.C-31 No.9 pp. 913-917.

[MOL86] Molloy, M.K., "A CAD Tool for Stochastic Petri Nets", *Proceedings of the Fall Joint Computer Conference*, November 1986, pp. 1082-1091.

[NIE69] Nielsen,N.R., "ECSS: An Extendable Computer System Simulator", *Proceedings of the Third Conference on Applications of Simulation*, 1969, pp.114-129.

[NOE73] Noe, J.D., & Nutt, G.J., "Macro E-nets for Representation of Parallel Systems", *IEEE Transactions on Computers*, vol. C-22, No.8., August 1973, pp.718-727.

[NOE78] Noe, J.D., "Hierarchical Modeling with Pro-Nets", *Proc. of the National Electronics Conference*, vol. 23, October 1978, pp.155-160.

[NUT72] Nutt, G.J., "Evaluation Nets for Computer Systems Performance Analysis", *Proceedings of the 1972 Fall Joint Computer Conference*, Montvale, New Jersey: AFIPS Press, December 1972, pp. 279-286.

[PET80] Peterson, J.L., "A Note on Colored Petri Nets", *Information Processing Letters*, August 1980, pp.40-43.

[PET81] Peterson, J.L., "Petri Net Theory and the Modelling of Systems", *Prentiss Hall Inc*., Englewood Cliffs, NJ 07632, 1981, 290 pages.

[PFI85] Pfister, G.F., et. al., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture", *Proc. 1985 Conf. on Parallel Processing*, August 1985, pp. 764-771.

[RAM80] Ramamoorthy, C. V., & Ho, G. S., "Performance Analysis of Asynchronous Concurrent Systems using Petri Nets", *IEEE-Transactions on Software Engineering*, vol. SE-6, No.5, Sept. 1980, pp. 440-449.

[REI82] Reisig, W., "Petri Nets, An Introduction", *EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, 1982, 161 pages.

[SAU82] Sauer, C.H., MacNair, E.A., & Kurose, J.F., "The Research

Queueing Package Version 2: Introduction and Examples", *IBM Research Report*, RA-138, Yorktown Heights, New York, 1982.

[SEJ80] Sejnowski, M.C., et. al., "An Overview of the Texas Reconfigurable Array Computer", *Proc. of AFIPS NCC Conference*, 1980.

[SHA85] Shatz, S.M., & Cheng, W.K., "Static Analysis of ADA Programs Using the Petri Net Model", *Proceedings of the International Symposium on Circuits and Systems*, ISCAS 1985, pp.719-722.

[SIE81] Siegel, H.J., "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition", *IEEE Transactions on Computers*, December 1981.

[SIF79] Sifakis, J., "Performance Evaluation of Systems using Nets" *Proc. of the Advanced Course on General Net Theory of Processes and Systems*, Hamburg, Oct. 1979, pp. 307-320.

[STO85] Stotts, P.D., "A Hierarchical Graph Model of Concurrent Software Systems", *PhD. Dissertation*, Department of Computer Science, University of Virginia, Charlottesville, Virginia, May 1985.

[SUH87] Suhler P.A. & Biswas J., "The Task-Level Data Flow Language", *Technical Report*, Department of Computer Sciences, University of Texas, Austin, January 1987.

[VAL78] Valk, R., "On the Computational Power of Extended Petri Nets", *Lecture Notes in Computer Science*, Springer-Verlag, No. 64, 1978, pp.526-535.

[VER83] Vernon, M., de Souza e Silva, E. & Estrin, G., "Performance Evaluation of Asynchronous Concurrent Systems: The UCLA Graph Model of Behavior", *Proc. of the 9th International Symposium on Computer Performance Modelling, Measurement, and Evaluation*, PERFORMANCE 1983, pp. 153-171, May 1983.

[YAU83] Yau, S. S., & Caglayan, M. U., "Distributed Software System

Design Using Modified Petri Nets", *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 6, Nov. 1983.

[ZUB80] Zuberek, W.M., "Timed Petri Nets and Preliminary Performance Evaluation", *Proc. of the* 7th *annual Symposium on Computer Architecture*, 1980, pp. 88-96.