12-1-1987

COSMIC: A Model for Multiprocessor Performance Analysis

William W. Carlson
Purdue University

Jose A. B. Fortes
Purdue University

COSMIC: A Model for Multiprocessor Performance Analysis

William W. Carlson
Jose A. B. Fortes

TR-EE 87-13
December 1987

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

COSMIC: A Model for Multiprocessor Performance Analysis

TR-EE 87-13

William W. Carlson and Jose A.B. Fortes
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

Abstract

COSMIC, the Combined Ordering Scheme Model with Isolated Components, describes the execution of specific algorithms on multiprocessors and facilitates analysis of their performance. Building upon previous modeling efforts such as Petri nets, COSMIC structures the modeling of a system along several issues including computational and overhead costs due to sequencing of operations, synchronization between operations, and contention for limited resources. This structuring allows us to isolate the performance impact associated with each issue. Finally, studying the performance of a system while executing a specific algorithm gives insight into its performance under realistic operating conditions. The model also allows us to study realistically sized algorithms with ease, especially when they are regularly structured.

During the analysis of a system modeled by COSMIC, a set timed Petri nets is produced. These Petri nets are then analyzed to determine measures of the systems performance. To facilitate the specification, manipulation, and analysis of large timed Petri nets, a set of tools has been developed. These tools take advantage of several special properties of the timed Petri nets that greatly reduce the computational resources required to calculate the required measures. From this analysis, performance measures show not only total performance, but also present a breakdown of these results into several specific categories.

Table of Contents

1. Introduction

The complexity and variety of recent multiprocessor developments require new models which allow studies and comparisons of their diverse features and functions. Such models must allow researchers to concentrate their efforts on the essential issues confronting multiprocessors and ignore those features which merely serve to distract. To this end, we propose the use of a new model, called *COSMIC* (Combined Ordering Scheme Model with Isolated Components), for the study and comparison of multiprocessor systems. The underlying principles of this model are the isolation of individual performance issues and the study of systems under realistic operating conditions. COSMIC consists of both formal parameters that describe a multiprocessor system and the algorithm it executes, and analysis techniques that produce performance measures.

Our goals in developing this model have been to gain the ability to study multiprocessor systems with a variety of schemes for ordering operations. The model must also represent time and resource utilization so comparative studies may be made. The major concern is a class of systems called *Combined Data Flow and Control Flow Systems* which were describe in a previous paper [CaF87]. To achieve this goal, our model must represent both hardware and software issues, as well as the binding mechanism between them. The binding mechanism, which we call the *Ordering Scheme,* describes how operations are ordered on a multiprocessor system.

Previous work in modeling multiprocessors has centered in several distinct areas. Program behavior models endeavor to model the fundamental properties of a program without regard for hardware considerations or performance measurement. They center on the important areas of investigating such problems as the determinacy, boundedness, and termination of programs. Models which fit into this category include Petri nets [Pet66] and Parallel Program Schemata [KaM66]. The second major category of current models we call machine behavior models as they describe the behavior of machines in their execution of programs as opposed to the behavior of programs themselves. Examples of this class include Turing Machines, Functional Programming Systems, and the von Neumann Model [Bac78]. Classification models describe the configuration and operation of multiprocessors, including Flynn's Model [Fly66], Handler's Classification System [Han77], and the "essential issues" of Gajski and Peir [GaP85]. Stochastic models based on queueing theory [Kle75] have also been used to model multiprocessor systems.

Using these previous models as a basis, COSMIC combines both program and machine descriptions, as well as performance measures. Its usefulness is in this combination, allowing the study of complete systems under varied conditions.

This report is organized into 4 major sections. Section 2 contains a survey of previous efforts in modeling multiprocessors. Its purpose is to present a background for our model, as several key concepts found in previous efforts are used. Section 3 formally presents COSMIC, describing the parameters of the model, its performance measures, and a method used to determine the measures from the parameters. Section 4 contains descriptions of several tools which were developed to expedite the analysis of modeled systems. Finally, Section 5 consists of several examples of systems modeled using COSMIC. It shows that COSMIC can be used to determine the performance of systems which vary greatly. Appendices provide documentation for the modeled systems.

2. Computer System Models

Models allow researchers to disregard distracting details associated with real systems and concentrate on the issues considered essential. A model of computer systems is simply a mechanism to describe some aspect of the system's operation. Some models of computation, such as those presented by Karp and Miller [KaM66], are used to represent execution of programs to investigate such problems as determinacy, boundedness, and termination. Other models, such as those by Flynn [Fly66], are used to represent the organization of a computer. These models are sometimes called classification systems and are used to classify the modes of operation and interconnection of computer elements. Finally, a third class of models, such as the von Neumann model, describe how programs execute on a machine.

As stated in the introduction, our goals require a model to not only represent the organization of a computer system, but also its schemes for executing programs. The requirement for comparative studies places heavy demands on a modeling system. This section presents a survey of several models showing that, while none are appropriate for our needs, several provide a foundation for COSMIC.

Current models of computation can be divided into three categories. The first category contains models which can be described as *program behavior models* because researchers use them to study the abstract execution of programs in a parallel environment. These models lack convenient methods to describe or experiment with system organizations. Most would also present insurmountable problems for our comparative study as they fail to incorporate time and resource consumption. The second category consists of models that can be described as *machine behavior models* because they are used to describe the behavior of machines while they execute programs, as opposed to individual program behavior. These models cannot describe the exact execution of a given program and do not contain machine organizational details. The third category contains models that can be described as *classification models,* as they are used to categorize and describe systems. Unfortunately, they contain no capacity to represent the execution of a program, or even to define the operation of the system.

Aside from the categorization just discussed models may be informally classified as either tight or loose. Tight models consist of well defined mathematical descriptions. They are useful in proving characteristics of the systems they describe, and developing accurate performance predictions. Alternatively, loose models offer only vague descriptions and classifications of system issues. Loose models are useful for several reasons, not the least of which are course nomenclature and taxonomy. However, because of the comparative nature of our study, only tight models are acceptable.

Another distinction which can be drawn between the various models is their ability to describe deterministic and non-deterministic behavior. Deterministic models describe constant events and are considerably easier to analyze. Models which describe non-deterministic events usually associate random variables with decision points allowing the description of more complex phenomena. Unfortunately, the analysis of such models is far less straight forward.

2.1. Program Behavior Models

Our survey is not the first of models used to study the execution of parallel programs. Consequently we endeavor not to repeat past work. Miller [Mil73] presents a good survey of models in the program behavior class, describing and comparing several of the numerous theoretical models for representing parallel processes. The models described (and focused on here) are Petri nets [Pet66], Computation Graphs [KaM66], and Parallel Program Schemata [KaM69]. Queueing network models [Kle75] are also briefly discussed. These models fit our category of program behavior models and while each is tight, all have other shortcomings that make them unsuitable for our research. Several may be used to describe either deterministic or non-deterministic events.

Petri Nets

Petri nets, developed by Petri [Pet66], have been used widely to describe the sequencing of concurrent events [Pet81]. A Petri net is a directed bipartite graph, with vertices called *places* and *transitions.* Places can hold *tokens,* the collection of which is called a *marking.* Every Petri net has associated with it an *initial marking* which is the marking present before any transitions fire. In effect, it is the initial state of the net. The set of *input places* for a transition consists of those places joined by an arc entering the transition. Correspondingly, an arc joins each transition to the members of its set of *output places.* If every input place for a transition contains at least one token then the transition is *active,* and may *fire* at any time. The firing of a transition removes a single token from each input place and adds a new token to each output place. Concurrent events are modeled by the simultaneous firing of a Petri net's transitions. The presence of tokens at input places represents the satisfaction of conditions on the occurrence of an event. The firing of one transition will in turn cause other "events" to become active and subsequently fire.

An extension of this concept, called the timed Petri net [Ram74], associates a non-negative number with each transition representing the time between the consumption of a token from an input place and the production of a token at the output. Figure 2.1 illustrates this phenomena in a simple timed Petri net, showing its markings before, during, and after the firing of a 3 time unit transition. Timed Petri nets are analyzable using timed reachability graphs or reduction techniques to determine the composite firing time of an entire net. Additionally, branching probabilities can be assigned to outputs of transitions to allow conditional firing. The concept of a random variable associated with the time each transition requires to fire has also been studied in stochastic Petri nets [Mol82].

In attempting to use Petri nets to model the execution of algorithms on multiprocessors, a deficiency becomes apparent. The entire system (including organization, software, operating schemes, etc.,) would have to be incorporated into a single Petri net. For example, to change a sequencing policy an entirely new model would have to be developed, which would in turn make any comparative study infeasible. Despite this shortcoming, the model is extremely useful for describing arbitrary concurrent events. The Petri net concept is used in the development of COSMIC for this purpose.

Computation Graphs

Computation graphs were proposed by Karp and Miller [KaM66] as a graph-theoretic model for the description and analysis of parallel computations. Vertices correspond to computation steps and arcs represent a queue for data directed from one vertex to another. Associated with each arc is the 4-tuple (A, U, W, T), where the elements are:

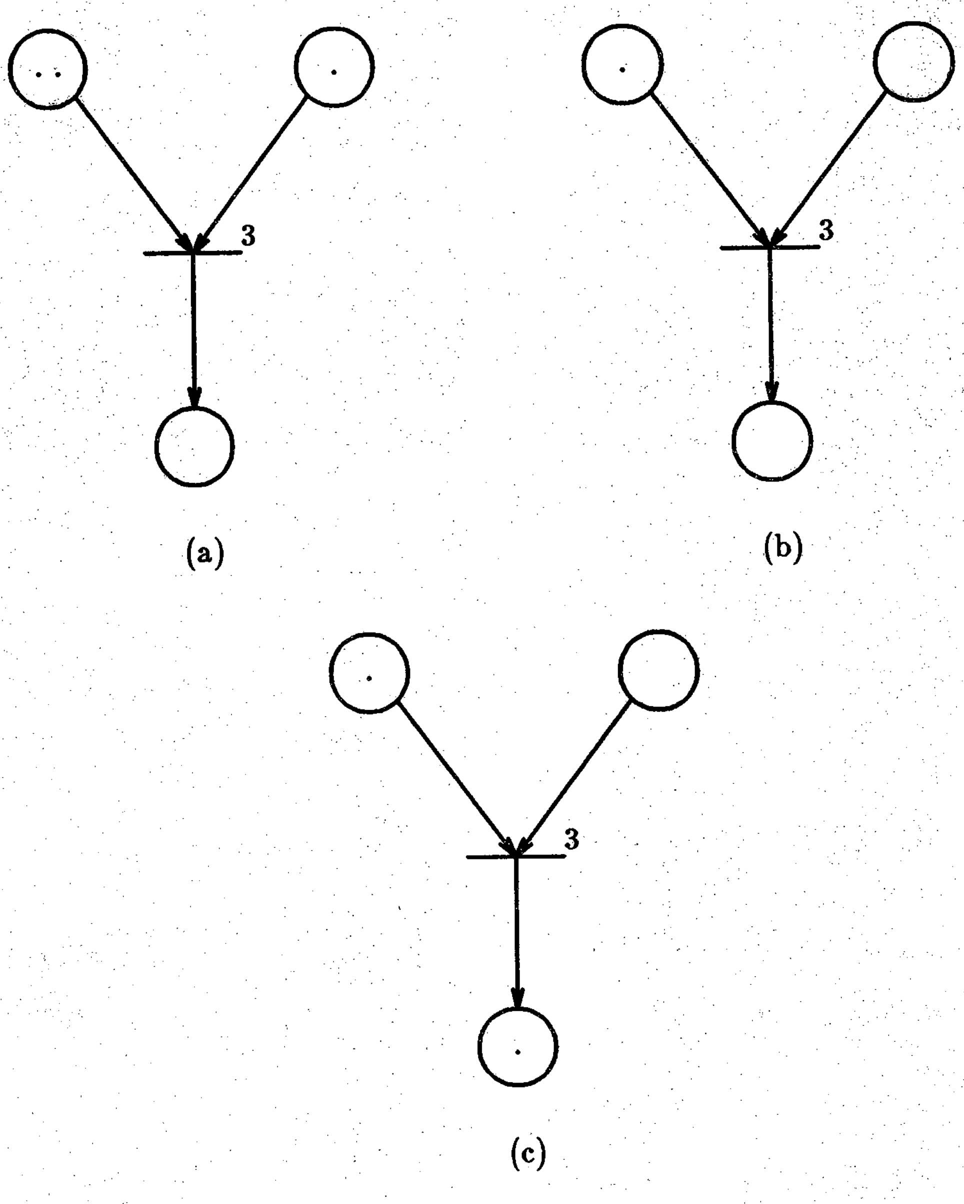


Figure 2.1

Timed Petri Net Markings: (a) Before Firing ($t < t_0$); (b) During Firing ($t_0 \leq t < t_0 + 3$); (c) After Firing ($t_0 + 3 \leq t$).

A	data items initially in a queue;
U	data items added whenever an operation is performed by the tail vertex of the arc;
W	data items deleted when an operation is performed by the head vertex of the arc;
Y	the minimum length of queue to allow the operation at the head vertex of the arc to begin.

Computation Graphs are most useful when studying the parallelism in simple repetitive processes, such as "inner loops" of computations. For example, the computation of Fibonacci numbers is simply a graph with a single vertex and an arc joining the vertex to itself. The 4-tuple (A,U,W,T) is (2,1,1,2) indicating that there are 2 items initially in the queue, each operation produces and consumes a single item, and that there must be 2 numbers in the queue to start an operation. From this it can be seen that the operation never terminates, as queue length is limited to 2 and each item placed in the queue is used twice. Computation graphs facilitates analysis of this form.

Several major problems prevent us from adopting computation graphs for performance modeling. The most severe difficulty results from the model's inability to represent the time consumed by a system, making such a comparative study impossible. The description of complex processes (e.g. sequencing schemes) with computation graphs would be difficult, and heightened by the requirement of cyclic computation. Additionally, the flexibility of computation graphs is limited. Only one arc may leave each vertex, preventing the modeling of conditional branching or fork and join constructs. These limitations make computation graphs infeasible for modeling complex systems.

Parallel Program Schemata

Karp and Miller [KaM69] also developed Parallel Program Schemata. This model is the most complex of the several models presented here and also the most flexible. A Parallel Program Schema (M,A,T) consists of a set M of memory locations; a finite set A of operations; and the transition function T to sequence the operations. Associated with each operation in A is a set of domain locations in M and a set of range locations, also in M. To represent conditional transfers, each operation has several possible outcomes. An interpretation function, I, can specify the exact function of the operations in A. Karp and Miller use this model to determine the conditions under which programs are determinate, bounded, and repetition free.

In one respect Parallel Program Schemata serves as an inspiration for our model of computation, as the concept of dividing the model into components that work together to describe complex issues is most applicable to our problem. Unfortunately, the model still lacks in the areas of defining time requirements of operations and in the incorporation of certain organizational aspects of systems.

Queueing Network Models

Queueing Network Models (QNM) have been widely use to describe the operation of multitasking computer systems and numerous analysis techniques exist to analyze such systems [Kle75]. Operating system terminology is used to describe the components of a QNM, as the study of such systems has been their chief application. The components of a QNM are *sources, servers,* and *queues.* Servers generally represent resources demanded by *jobs,* and require some time to service, specified by a random variable. Sources create jobs at a certain rate, specified by another random variable. Associated with servers are queues to hold waiting jobs. The connection of these components is a queueing network model. Figure 2.2 shows a simple example of a QNM, with one source, one server, and one queue. Jobs are placed in Q by either the completion of service or creation by a source. This model can be analyzed to determine performance criteria like system throughput and mean queue length.

The major advantage of this type of model is its ability to be solved by well known analytic techniques, and its faithful representation of time consumption. The problem in using this of model is its inherent adoption of the use of queues to contend for limited resources. Queues may not realistically model events in the systems in our study.

2.2. Machine Behavior Models

Backus [Bac78] classifies several models of computation including Turing machines, Functional Programming systems, and the von Neumann model. These models are termed machine behavior models because they describe the

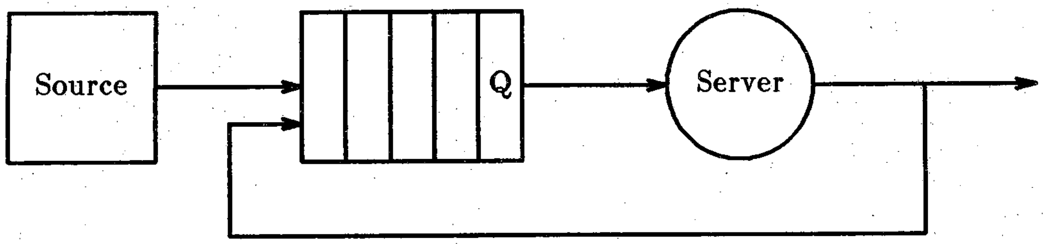


Figure 2.2
Queueing Network Model for Single Server System

behavior of machines in their execution of programs as opposed to the behavior of programs themselves. They do not contain mechanisms to describe the sequencing of specific programs, nor specifics of machine organization. Therefore, all are unsuitable for our needs. For illustrative purposes, three models spanning this category are briefly described.

Turing Machines

A Turing machine is an example of what Backus terms simple operational models. An example of such a machine is a finite state machine with an attached memory device. This machine has the ability to advance the memory in either direction and either read or write to the memory. This is a tight model that has a simple and concise mathematical foundation resulting from the finite state machine. It uses storage to save information that can later affect the behavior of programs. This makes the model history sensitive. However, the clarity is not great as only simple state transitions are allowed. This model is clearly not adequate for representing such complex issues as ordering schemes and multiprocessor organization.

Functional Programming Systems

Functional Programming Systems are an example of what Backus terms the applicative model. Such systems consist simply of a set of objects, a set of functions which map objects onto objects, and an application operation which applies a function to an object producing another object. Models like this have no concept of storage, are not history sensitive, and programs written for them are clear. Such a model is the basis of several data flow systems and languages. While this form of model is useful in describing programs or programming systems, they will be of little use in a comparative study such as ours.

The von Neumann Model

The von Neumann model of machine behavior is simply an extension of the computer developed by von Neumann many years ago. In its simplest form it has a single central processing unit, a store, and some method to connect the processor to the store. This model is the basis of all "conventional" programming languages. As the concept of these languages is the stored program, they are history sensitive and not mathematically concise. Again, note that machine behavior models do not provide the support needed to describe the specifics of program sequencing and machine organization.

2.3. Classification Models

Classification models allow description of the configuration and operation of parallel systems. While this class of models is wide, all members share similar deficiencies for use as performance characterization models. In general there is a lack of both tightness and ability to concisely describe the program portion of a system. Three classification models are discussed. Flynn [Fly66] presents perhaps the most well know system, describing instruction and data streams. Next, Handler [Han77] proposes a more complex system. Gajski [GaP85] presents "essential issues" for parallel processing. In a recent presentation, Browne [Bro85] suggested a new classification scheme based on the time of binding. All of these classification schemes can be used to describe multiprocessing systems in varying detail.

Flynn's Model

Flynn's models [Fly66] for parallel processing are perhaps the most widely used in describing computer systems. Flynn classifies computers with two criteria: the parallelism in instruction streams and data streams. There are four possible combinations: SISD, SIMD, MISD, and MIMD; referring to single and multiple instruction and data streams. SISD machines are "conventional" uniprocessor systems, in which one thread of control operates on a single set of data. SIMD machines allow multiple data items to be operated on simultaneously by the available processing units, yet still under unified control. Generally SIMD machines are either vector or array processing architectures. MIMD computers allow more than one thread of control, each with either single or multiple data items operated on by each instruction stream. Many modern multiprocessors are MIMD, including multiprocessor data flow machines. MISD architectures would require multiple operations on a single thread of data. They are far less common and one generally needs to stretch definitions to find an example.

This model of parallel processing is unacceptable for use in performance modeling. Not only does it not allow any consideration of the execution of programs, but also does not go into enough detail to describe such issues as interconnection systems and memory organization.

The Erlangen Classification System

Handler proposed a more explicit classification system, which he called the Erlangen Classification System [Han77]. Under this classification, each system is divided into three levels of complexity. The Elementary Logic Circuit level (ELC) is the most basic, operating at the single bit level. The Arithmetic and Logic Unit level (ALU) executes sequences of operations at the ELC level. Finally, the Program Control Unit level (PCU) interprets a program instruction by instruction. Each PCU can control several ALU level components, and in turn each ALU can control several ELC level components, each dedicated to one bit position. A system under this model is the 3-tuple $t = (k \times k', d \times d', w \times w')$, where the elements are:

k	Parallelism at PCU level;
k'	Length of each PCU pipeline;
d	Parallelism at ALU level;
d'	Length of each ALU pipeline;
w	Parallelism at ELC level;
w'	Length of each ELC pipeline.

With this system it is possible to describe the parallel and pipeline complexity of a system at the three levels shown.

Unfortunately, this model also suffers several fatal flaws when considering applicability to our goals. Again, no memory or interconnection issues are addressed, nor any concept of a program executing on such a system. The expansion of Flynn's stream concept allows more detailed description of systems but does not bridge the gap to understanding the operation of such systems. However, the concept of hierarchical control contributed to the development of COSMIC.

Gajski's Essential Issues

While Gajski's ideas on the essential issues in parallel processing [GaP85] do not form a formal classification system as in the previous two examples, they do tend to classify systems by their common and divergent traits and thus are discussed now. They describe five issues that must be confronted when designing an effective multiprocessor system. These issues are control, partitioning, scheduling, synchronization, and memory access.

Partitioning is the job of dividing a program into units that can be executed in parallel, given proper synchronization and scheduling. It involves first detecting parallelism and then clustering several operations based on some optimality considerations. The control issue is similar to other classification systems in that it defines several levels of control and if each of those levels operate in serial or parallel. Sequencing is the process of determining which operations occur after the completion of other within a segment. Synchronization is required to ensure data dependencies are observed between two separately sequenced modules. Finally, the memory access issue takes into account the overhead that a program will encounter because accesses to memory including network traversal and contention issues. By simply comparing and contrasting the methods various systems use to confront these issues, a basic classification system is devised. These issues are indeed essential and influenced the development of COSMIC.

Browne's Binding Time

A classification scheme has been developed by Browne [Bro85]. This classification is based on the *binding time,* an indication of when functions are bound to the resources that will execute them and when they are bound to the time they will begin execution. For example, a system with dynamic scheduling will have a later binding time (run-time) than one with static scheduling (load-time). This system can classify a large number of systems, ranging from systolic arrays where binding will occur at design time to dynamic data flow machines where binding can happen immediately before execution.

2.4. Survey Summary

A wide variety of mechanisms have been used to describe multiprocessors. While some of these models will serve as a useful basis for the design a model fitting the goals specified in this report, none would be appropriate unmodified. In the next section COSMIC is presented. The inspiration from several of the models described above will be quite evident in its development.

3. The COSMIC Performance Evaluation Model

To analyze the performance of multiprocessors executing specific algorithms, we have developed *COSMIC,* the Combined Ordering Scheme Model with Isolated Components. COSMIC consists of both formal parameters describing a multiprocessor system (including the algorithm it executes) and analysis techniques producing performance measures. The underlying principles of this model are the isolation of individual performance issues and the study of systems under conditions close to those encountered when a system is performing useful calculations. This section describes the parameters and analysis of systems represented by COSMIC.

3.1. COSMIC Parameters

A system S is defined by the triple $S = (O \ , G_d \ , OS)$, where O is the system's organization; G_d a dependence graph describing a specific algorithm; and OS is the ordering scheme used to execute algorithms on the organization. Included in a system's organization are such features as the number and speed of processing elements, the amount and organization of memory, and the interconnection amongst processors and memory. The dependence graph is simply an operation level precedence graph for a certain algorithm. This graph includes only algorithmic constraints, not those induced by operation sequencing or programming languages. Finally, the ordering scheme describes how algorithms are executed on the organization. The ordering scheme is further segmented into descriptions of a system's mechanisms for partitioning, sequencing, resource allocation, and memory utilization.

3.1.1. Organization (O)

The organization represents the arrangement of hardware elements of a system. Every multiprocessor has three basic components; processing elements (PE), memory locations, and some method to interconnect them. Input and output devices are simply specialized processing or memory elements. Consequently, our model for organization is represented by the triple $O = (P, M, I)$, where P, M, and I are:

- P -- A set of processing elements. Each processing element has a set of instructions that it can execute and a relative speed. The ordering scheme describes how these instructions are ordered.

- M -- A set of memory locations.
- I -- An interconnection function $M \times P \rightarrow M \times P$. This function defines the possible interconnections, and with each outcome there is a related cost function that describes the cost of traversing that connection. Local memory on a PE can be modeled by a low cost function (perhaps zero). Inaccessible memory (another PE's local memory) can be modeled by I being partial on $M \times P \rightarrow M \times P$.

3.1.2. Data Dependency Graph (G_d)

The data dependence graph is an arc and vertex weighted directed graph in which vertices represent operations and arcs represent data dependencies between operations. The weight of a vertex represents the relative time that it will consume when executed. The weight of an arc represents the size of the data needing transfer to satisfy the dependency. These weights can also be viewed as the number of "atomic operations" required to complete the computational or transfer operation. This graph is acyclic, as any loops in a program are unfolded in creating the dependency graph. Data dependent behavior is not considered, but will be included in future research.

3.1.3. Ordering Scheme Function (OS)

The ordering scheme for any system is a function mapping the dependency graph into an ordering net, based on the organization parameters. An *ordering net* is a timed Petri net [Ram74] which depicts ordering constraints placed on the execution of operations, as well as the cost of each operation in the modeled system. The ordering scheme for an organization O can be defined as:

$$OS(O)\colon G \rightarrow N,$$

where G is the set of all possible dependency graphs and N is the set of all possible ordering nets. This function is the composition of several smaller, more easily defined functions. Thus the ordering scheme function,

$$OS(O) = \gamma(O) \circ \mu \circ \lambda \circ \phi(O) \circ \tau(O),$$

(where the usual composition notation implies that $(f \circ g)(x)$ is equivalent to $f\ (g\ (x))$,) contains the component functions:

$\tau\colon G_d \rightarrow N$	Creates an ordering net G_d ,
$\phi(O)\colon N \rightarrow N$	Adds partitioning constraints,
$\lambda\colon N \rightarrow N$	Adds sequencing constraints,
$\mu\colon N \rightarrow N$	Adds memory access constraints,
$\gamma(O)\colon N \rightarrow N$	Adds resource constraints.

While the next sections detail each function, several observations apply to all. Functions described as *system independent* never change over all possible ordering schemes. A function will be *system dependent* if the function itself changes from one ordering scheme to the next. Additionally, some functions may be *organization dependent* indicating that the result of applying the function to a net varies according to some aspect of the organization. A function may be organization independent while being system dependent. For example, τ is system and organization independent, ϕ is system and organization dependent, and λ is system dependent and organization independent.

Note that because an ordering net is a timed Petri net, it has an underlying directed bipartite graph. In this bipartite graph nodes represent places and transitions, and edges occur only between places and transitions. In the following descriptions we take the liberty of referring to the features of this graph as if they were features of the ordering net. For example, the indegree of a transition refers to the indegree of the corresponding vertex in the underlying graph.

Related to each ordering scheme function is a measure which indicates the performance impact of that function. A measure is a triple whose elements are the serial time and the critical path time and space requirements to fire an ordering net. While a precise definition of these measures is provided in Section 3.2, this section provides a brief description of the measures corresponding to the composite OS functions.

Computation Function (τ)

The computation function creates an initial ordering net from a data dependency graph. Its sole purpose is to change domains from data dependency graphs to ordering nets and is organization and system independent. Formally the computation function is defined as

$$N_c = \tau\left(G_d\right),$$

where N_c is the computation ordering net for a given G_d produced by τ. This function requires five steps, which are now detailed.

Step τ_1 -- Arcs and Vertices in G_d

- For each vertex in G_d create a transition in N.
- For each arc in G_d create a place in N.
- For each arc, a_i, in G_d, with head v_{ih}, create an arc in N joining the transition corresponding to v_{ih} to the place corresponding to a_i.
- For each arc, a_i, in G_d, with tail v_{it}, create an arc in N joining the place corresponding to a_i to the transition corresponding to v_{it}.

This step creates the first approximation to an ordering net from a data dependency graph. Transitions in the net will correspond to operations and places to the storage or movement of data between operations. The structure of the net is identical to that of the dependency graph. An example G_d is shown in Figure 3.1a and the result of Step $\tau_1$is shown in Figure 3.1b.

Step τ_2 -- Initial Transition

- Create an initial transition.
- For each transition created by step τ_1 with indegree zero, create a place.
- For each newly created place, create an arc joining the initial transition to the place.
- For each newly created place, create an arc joining the place to the corresponding transition with indegree zero.

This step ensures that there is only one input to the ordering net. Figure 3.1c shows the result of applying τ_2 to the net in Figure 3.1b.

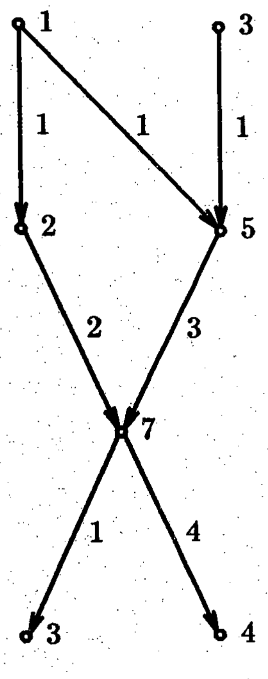
Step τ_3 -- Initial Place

- Create an initial place, containing a single token.
- Create an arc from the initial place to the initial transition.

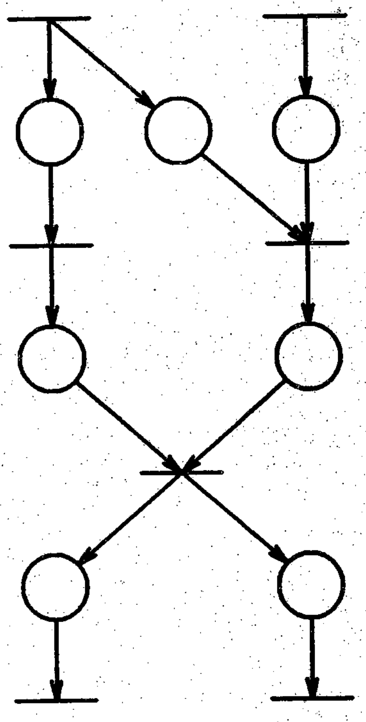
This step creates an initial place for the ordering net and makes the initial transition active. Figure 3.1d shows the result of applying τ_3 to the net is Figure 3.1c.

Step τ_4 -- Final Transition

- Create a final transition.
- For each transition with outdegree zero, create a place.
- For each newly created place, create an arc joining the corresponding transition with outdegree zero to the place.

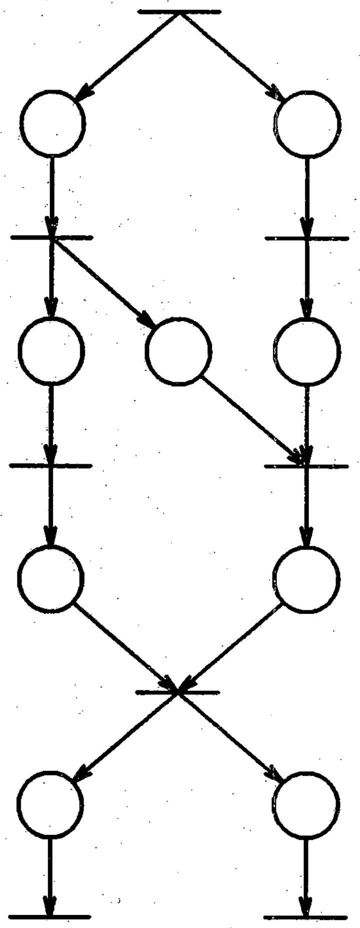


(a)



(b)

Figure 3.1
Computation Ordering Net Creation From Data Dependency Graph (a) Shows: (b) Initial Ordering Net; (c) Initial Transition; (d) Initial Place; (e) Final Transition; and (f) Weights.


(c)

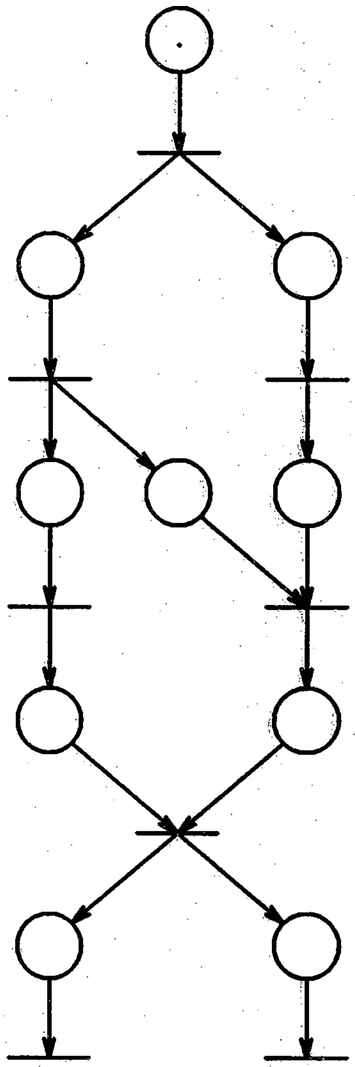
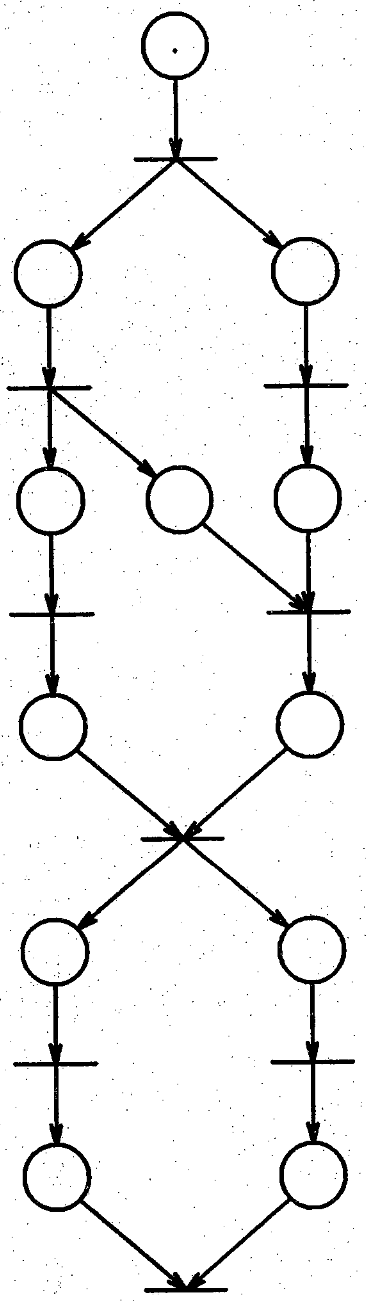
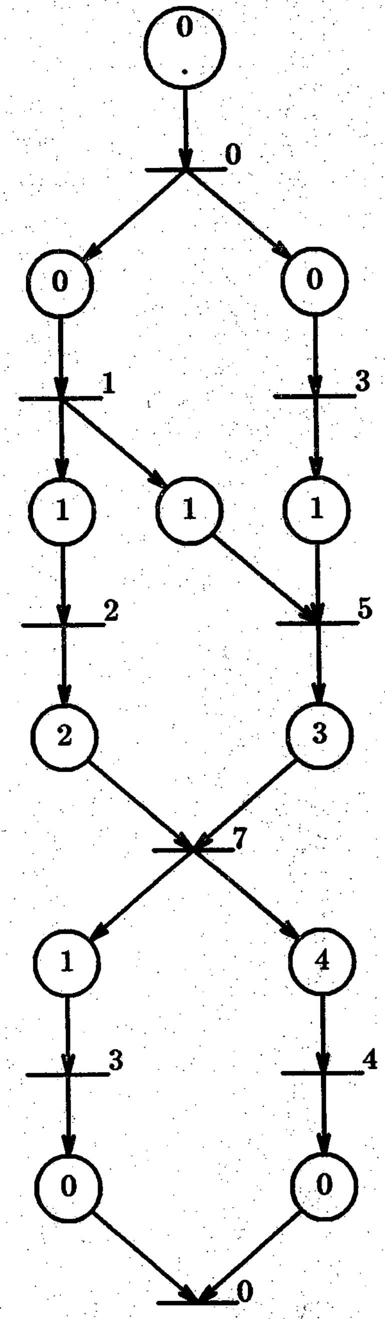

(d)

Figure 3.1, continued.


(e)



(f)

Figure 3.1, continued.

- For each newly created place, create an arc joining the place to the final transition.

This step ensures that there is only one output of the ordering net. Step τ_4 is illustrated by the transformation from Figure 3.1d to Figure 3.1e.

Step τ_5 -- Transition Firing Times and Place Weights

- Associate with each transition in the ordering net a firing time. Those transitions that correspond to vertices in G_d should be assigned the weight of the vertex. The initial and final transitions should be given firing times of zero.
- Associate with each place in N a weight proportional to the arc weights in G_d. The places added in steps τ_2 and τ_4 are given weight zero.

This step associates appropriate weights with each place and transition. The weight of a transition is its firing time, which is the elapsed time between the consumption of tokens from input places and the generation of tokens at output places. The weight of a place is representative of the amount of information stored in the place or transferred between two transitions. Figure 3.1f shows the ordering net resulting from the application for τ_5 to the ordering net shown in Figure 3.1e. The result of this function is a live timed Petri net in which only the initial transition may fire.

When N_c is analyzed by the technique described in Section 3.2 the computation measure (M_c) is obtained. This measure indicates the time and space requirements of the system when only computational operations are concerned and serves as a baseline for further studies of overhead. Note that this measure is always independent of ordering scheme and organization.

Partitioning Function (ϕ)

Partitioning is the process of dividing a program into segments to allow their execution on possibly distinct execution units. This division requires the addition of explicit synchronization between segments to preserve data dependencies. The partitioning function creates a new ordering net, N_{part}, based on these added synchronization requirements:

$$N_{part} = \phi\left(O, N_c\right).$$

Unlike the computation function, the partitioning function is system dependent but always follows a similar form, described in the following steps. (An example is presented after the description of all steps.)

Step ϕ_1 -- Segment Construction

- Assign each transition in the input net to one, and only one, segment.
- Assign each place for which all neighboring transitions belong to the same segment to that segment. All other places are not contained in any segment.

This step defines the creation of segments from the input net. The system dependent portion of this step is the algorithm used to choose the transitions assigned to a specific segment.

Step ϕ_2 -- Adding Synchronization Operations

- For each unassigned place, create two places and a transition.
- Add an arc joining each transition for which the unassigned place is an output place to the first newly created place.
- Add an arc joining the second newly created place to each transition for which the unassigned place is an input place.
- Add an arc joining the first newly created place to the new transition and one joining this transition to the other new place.

This step adds constraints to the model of the system which represent synchronization requirements added by a given partitioning of a net. This step is system independent.

Step ϕ_3 -- Firing Times and Place Weights

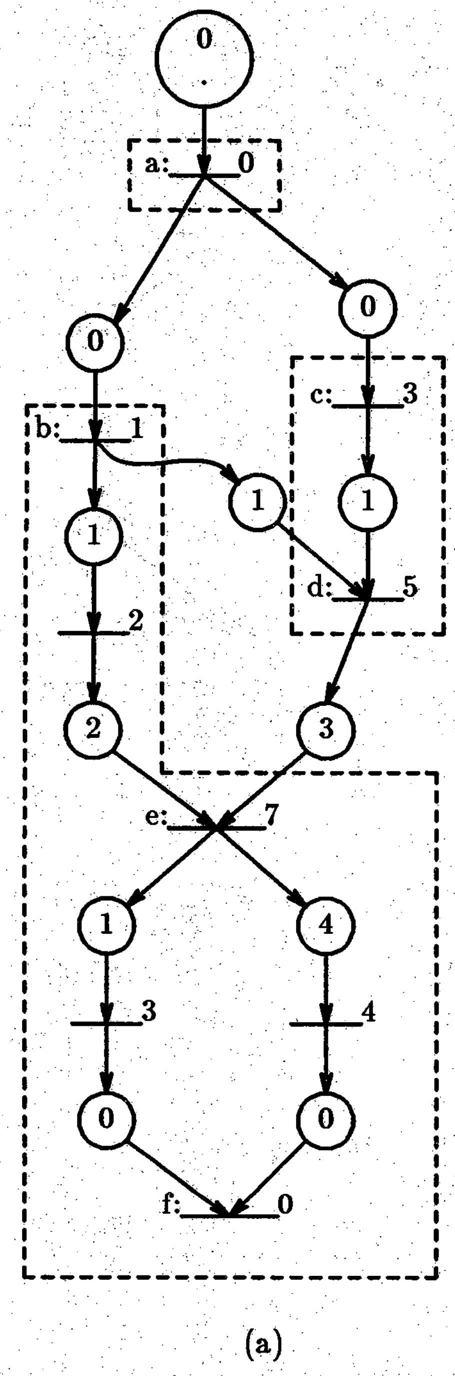
- For each transition added in step ϕ_2, assign a firing time.
- For each place added in step ϕ_2, assign a weight.

This step assigns costs to synchronization requirements. The firing time and weight chosen are organization dependent and proportional to the cost of synchronization in a given system.

Step ϕ_4 -- Recursion

- Apply this function to each newly created segment if additional partitioning is required for a given system.

An example of the application of the partitioning function is shown in Figure 3.2. Three segments are shown encircled by dashed lines in the initial net in Figure 3.2a, which is the result of applying step ϕ_1. Four places require synchronization; between the a-b, a-c, b-d, and d-e transitions. Figure 3.2b shows the new net, after transitions and places have been added. The new elements are shown in boxes and segments encircled in dashed lines for clarity.



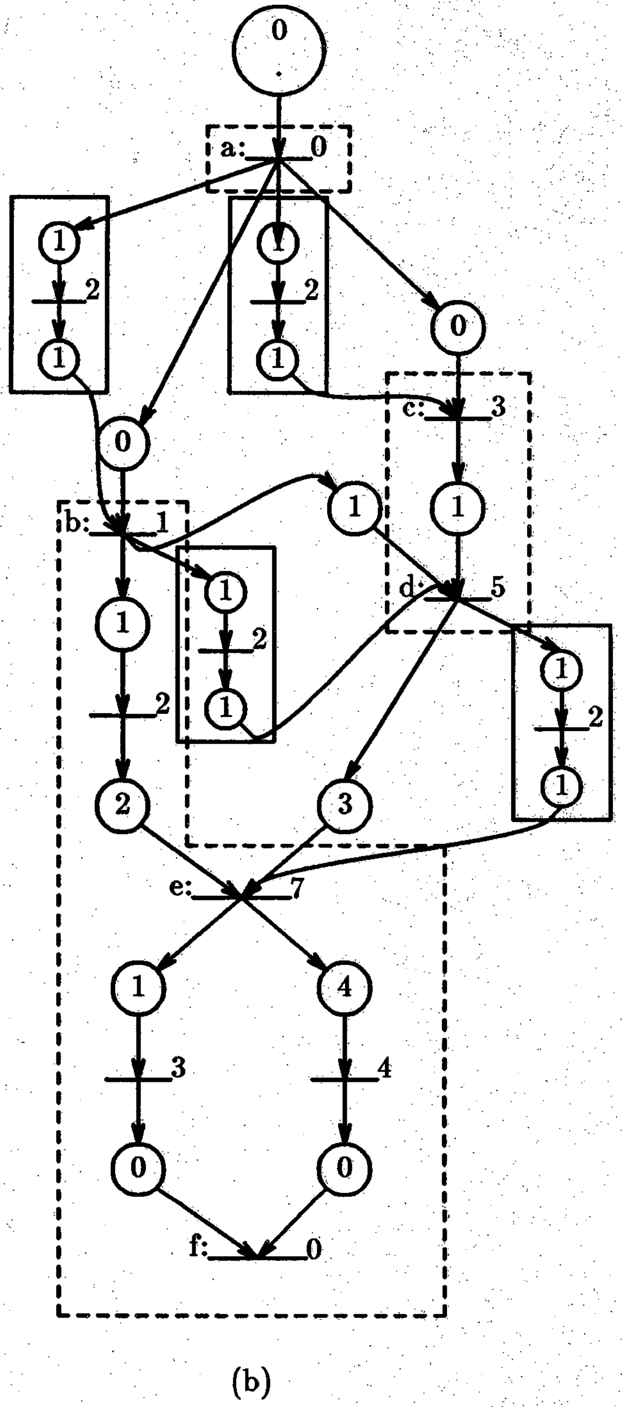


Figure 3.2
Partitioning of Ordering Net (a) Produces Ordering Net (b)

When N_{part} is analyzed a measure is produced. After M_c subtracted from this measure, the partitioning measure, M_{part} is arrived at. This measure indicates the overhead induced on the system by partitioning the algorithm according to the ordering scheme.

Sequencing Function (λ)

The sequencing function is responsible for adding constraints to the model induced by the sequencing of operations within the smallest segments of a given system, as well as the sequencing amongst those segments. Basically, this function causes the interpretation of either control flow, data flow, or combined schemes. Formally, the sequencing function produces a new ordering net from its input:

$$N_{seq} = \lambda \left(N_{part} \right).$$

Again, this function is system dependent, and must be specifically defined for each system. The following steps describe the sequencing function:

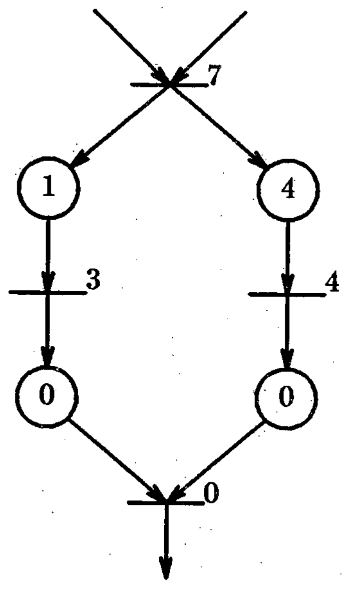
Step λ_1 -- Operation sequencing

- For each smallest segment produced by ϕ, determine one or more execution traces of transitions which will occur when the segment is executed on the system. This trace must not violate any dependency currently existing in the net (i.e. do not create a dependency loop.)
- For each execution trace, add a place-transition-place sequence between each pair of adjacent transitions along the trace.
- The firing time of the added transition will be proportional to the cost of sequencing on the system. The weight of the added places will be proportional to the amount of information transfer required to sequence.

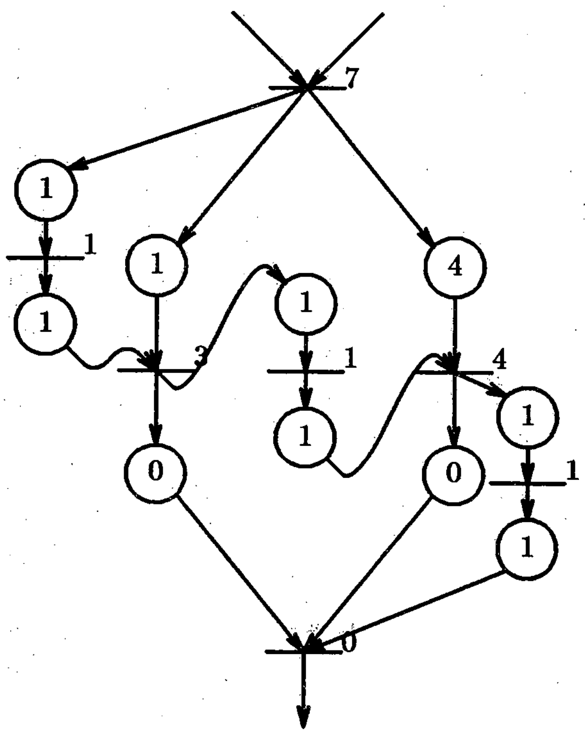
This step creates one or more execution traces for a given schedulable segment which was previously defined by partitioning. It reflects the implementation of the sequencing scheme at the operation level. Figure 3.3a shows a segment of the ordering net shown in Figure 3.1f, and Figure 3.3b shows the result of the application of this step, with a single execution trace.

Step λ_2 -- Segment level sequencing

- Connect each transition at the tail of a trace list produced by step λ_1 to the head of some other trace list or the final transition, by a place-transition-place sequence. If this connection is already made by a synchronization operation, do not add a new one. The trace list must not



(a)



(b)

Figure 3.3
Sequencing Example Showing (a) Partial Ordering Net and (b) Corresponding Operation Sequencing Net

violate any previously existent dependencies (i.e. no dependency loops).

- Connect the initial transition to the head of any list not used in the last step by a place-transition-place sequence.
- The firing time of each added transition is set equal to the segment sequencing cost for this system. The weight is again proportional to the required information transfer for segment sequencing. Modify the weights and firing times of synchronization operations appropriately.

This step implements the sequencing of groups of operations at various levels, dependent on the ordering scheme of the system. Figure 3.2 can also demonstrate segment level sequencing. If the ordering scheme required a serial trace of segments, a sequencing operation could be placed between the transitions labeled "d" and "b". This would ensure the segments themselves execute in a serial fashion while operations within the segments could operate in parallel.

When N_{seq} is analyzed a new performance measure is produced. When $M_c + M_{part}$ subtracted from this measure, the the sequencing measure, M_{seq} is arrived at. This measure indicates the overhead induced on the system by sequencing the algorithm according to the current ordering scheme.

Memory Access Function (μ)

To this point, the weight of each place in an ordering net has held all memory access and interconnection network traversal information. The memory access function produces a new ordering net which reflects added constraints induced by these factors.

$$N_{ma} = \mu\left(N_{seq}\right).$$

This function is independent of any system specifics and simply replaces each non-zero weighted place with a place-transition-place sequence. The new transition is given a weight equal to the weight of the place it replaces. Figure 3.4 shows the result of applying μ to the initial ordering net, which was shown in Figure 3.1f.

When N_{ma} is analyzed a measure is produced. Then, after $M_c + M_{part} + M_{seq}$ is subtracted from this measure, the memory access measure, M_{ma} is obtained. This measure indicates the overhead induced on the system by accessing memory.

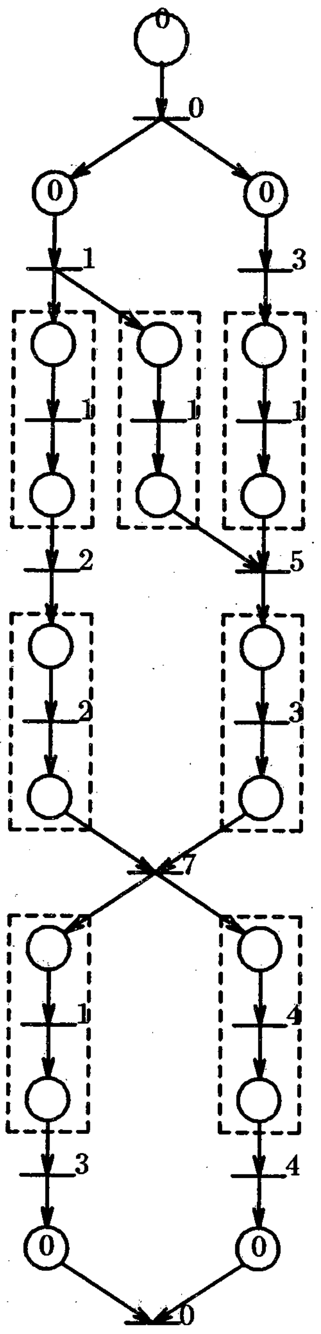


Figure 3.4
Ordering Net with Memory Access Constraints

Resource Allocation Function (γ)

Resource allocation is the process of assigning a set of vertices (transitions and places) to a set of resources (processing elements and memory locations.) Transitions represent computation and are assigned to processing elements, while places represent data storage (or transfer) and are assigned to memory locations. This function produces a new ordering net which reflects these constraints:

$$N_{ra} = \gamma\left(O\ ,\ N_{ma}\right)$$

It should be noted that if one wishes to model resource contention for memory devices, the memory access function μ must be applied before γ. After the application of γ the ordering net may again have μ applied to show the cost of the memory access added by γ. The resource allocation function is organization dependent and must be defined for each system modeled. Fortunately, all resource allocation functions follow the same general form:

Step γ_1 -- Resource Set Designation

- Create mutually exclusive sets of resources to be allocated before run-time. In this case a resource is considered a member of the set $P \bigcup M$, the set of all processing and memory elements.
- For each resource set, determine the subsets to be allocated at run time.

This step requires the designation of which groups of resources will be allocated before run time, and within those groups, which will be allocated at run time. This allows for the modeling of both static and dynamic resource assignment, as well as hybrid approaches.

Step γ_2 -- Resource Creation

- Create a place for each resource set to be allocated before runtime.
- Place one token for each run time allocated subset contained within a resource set in the corresponding place.
- Assign a weight of zero to each new place.

This step creates resources, and is system independent. Obviously the number of resources created is organization dependent. Each place is used to represent a distinct pool from which a resource may be taken. The tokens in this place represent the degree of run time parallelism available from that resource pool.

Step γ_3 -- Resource Assignment

- With the exception of the initial and final transitions, assign each transition in N to one or more resource sets which are required for its execution.
- For each transition assigned to a given resource set, add arcs to create a directed cycle containing both place corresponding to the resource set and the transition. The outdegree of each non-resource place on the cycle must be one.

This is system dependent and assigns the individual transitions to a specific resource pool. There are two convenient methods of resource assignment available:

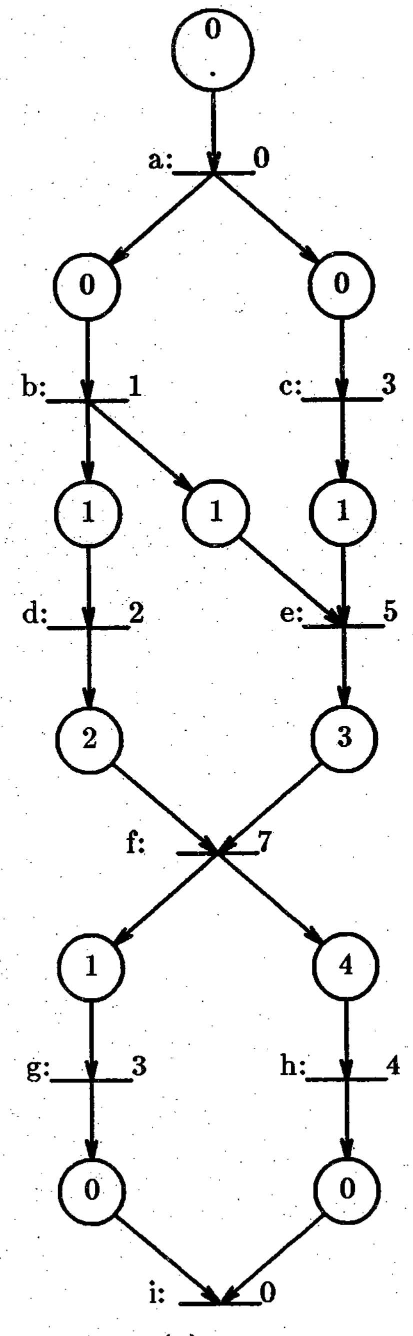
1) For each transition, create a length two cycle containing the transition and the resource place.
2) If a segment of N with one input transition and one output transition is to use a resource, an arc can join the resource place to the input transition and another can join the output transition back to the resource place. This models a system which allocates a resource to a group of operations that hold it until their completion.

Step γ_4 -- Place Weight and Firing Time Adjustment

- For each place in N, adjust its weight to account for added communication required.
- For each transition in N, adjust its firing time to adjust for changes in processor speed.

This step is system dependent, adjusting the firing times and place weights. Firing times must be adjusted after the resource assignment to account for differences in processor speed. Place weights must be adjusted based on the cost of communication, as described in the system's organization.

As an example, consider Figure 3.5. Figure 3.5a shows the same N_c found in Figure 3.1f, with the transitions labeled a through i for convenience. Figure 3.5b shows two resources are added, with the places labeled R1 and R2. Each of these resources is a one element resource set, as defined by step γ_1. Resource 1 is used to execute the subgraph consisting of transitions b through f. Resource 2 is used to execute transitions g and h. Additionally, the weights of the input places of g and h were doubled to reflect interconnection network delays. It can be seen that by the rules of Petri net firing, each resource can only be used for one transition (or subgraph) at a time, and that the token representing the resource is preserved.



(a)

Figure 3.6
Resource Constraints Featuring (a) Labeled Ordering Net And (b) Corresponding Net With Added Constraints

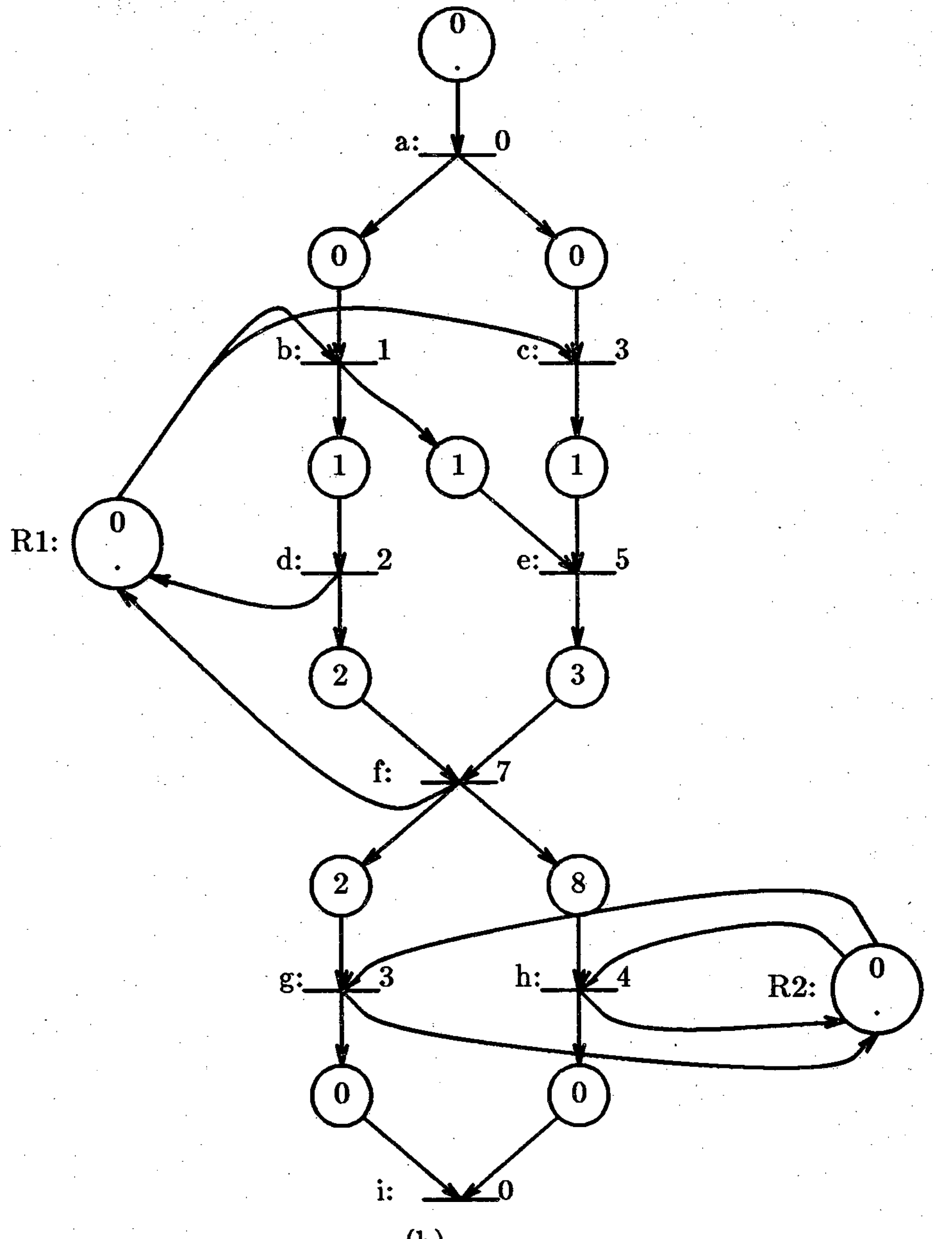


Figure 3.6, Continued.

An analysis of N_{ra} produces a measure. When the sum of previous measures $(M_c + M_{part} + M_{seq} + M_{ma})$ is subtracted, the resource allocation measure, M_{ra}, is arrived at. This measure indicates the overhead induced on the system by resource allocation the algorithm according to the current ordering scheme.

3.2. Analysis and Measures

After a system has been described by the parameters of COSMIC, it is analyzed to determine several performance measures. This analysis involves the determination of the time between the firing of the initial and final transitions in an ordering net. Computerized analysis tools, detailed in the next section, aid in this determination. The analysis begins by creating ordering nets using a high-level description language that enables the specification of parameterized nets. Generally, these parameters include the problem size and relative costs for computation, sequencing, and synchronization. A compiler then fixes values for these parameters and produces a set of interconnected places and transitions. Next, a net analyzer determines the various measures by firing the net following the rules of timed Petri nets. .Finally the results of many analyses are gathered into a database for further off-line studies. The entire system is capable of analyzing nets up to about 50,000 places and transitions while consuming reasonable computational resources. This enables the analysis of moderately large problems.

Three values are associated with each performance measure: the serial time, the critical path time, and the number of resources required to achieve the critical path time. These values describe both the time and space requirements of the modeled system for a given configuration. Two classes of measures are used: *primary measures* represent consumption of resources directly related to the algorithmic requirements of the system, and *overhead measures* show the consumption of resources unrelated to any algorithmic requirements. The analysis consists of the application of two analysis functions. The *serial analysis* function is:

$$AN_{Serial} : N \longrightarrow I\!R,$$

where $I\!R$ represents the set of real numbers and N the set of ordering nets. It computes the time required to fire all nodes in an ordering net, with the added constraint that no two transitions may fire simultaneously. The *critical path* *analysis* function is:

$$AN_{Crit} : N \longrightarrow I\!R \times I\!N,$$

where $I\!R$ represents the set of real numbers, $I\!N$ the set of non-negative integers, N the set of ordering nets, and $\times$ the cross product. It computes the time required to fire all nodes in an ordering net, with only the constraints presented by the net, as well as the number of resources required to achieve that level of performance. Finally the general analysis function,

$$AN : N \longrightarrow I\!R \times I\!R \times I\!N,$$

simply combines of the two previous functions, the result of which is a triple of values: (Serial Time, Critical Path Time, Critical Path Space).

If M is such a triple, the total execution measure for a model with organization O, ordering scheme OS', and data dependency graph G_d is:

$$M_{execution} = AN\left(OS\left(G_d, O\right)\right).$$

The execution measure is also, by definition, the sum of the five previously defined measures:

$$M_{execution} = M_c + M_{part} + M_{seq} + M_{ma} + M_{ra},$$

where M_c is the computation measure, M_{part} the partitioning costs measure, M_{seq} the sequencing costs measure, M_{ma} is the memory access measure, and M_{ra} is the resource allocation costs measure. These measures are also triples, the addition of which is defined in the usual manner by adding corresponding entries. These measures represent the analysis of an ordering net resulting from the application of a subset of the ordering scheme function. M_c is the primary measure, while the others are overhead measures. Overhead measures may contain negative entries for Critical Path Space since, as the critical path time grows, the space required to achieve that performance may decrease. These measures were described in the previous section.

Each of these measure components is unitless and should be expressed in a base unit related to one component. For example, if each synchronization operation takes one measure unit, then memory access or computation computations could take n synchronization operations. Alternatively, the base unit could be taken to be the memory access measure or scheduling measure. The important feature to observe is that these measures are represented in relative terms and the exact relationship must be determined. This data can also generate a measure of the efficiency of a system by considering the ratios of the primary and overhead measures to the total execution measure.

3.3. COSMIC Summary

This section has presented the model with examples to show its mechanical operation. By allowing the specification of complete systems in an orderly manner, COSMIC allows performance analysis of complex systems. This orderly specification in turn allows individual performance issues (e.g. sequencing overhead) to be separated from the total performance. The next section describes several tools required for successful analysis of larger systems.

4. Tools

This section describes in some detail a set of tools developed to aid in the analysis of systems modeled by COSMIC. The first tool discussed compiles a hierarchical description of a timed Petri net into a single level internal format. The second tool analyzes this internal representation to determine various measures. Finally, several ancillary tools allow the user to examine the internal descriptions and debug networks. This section will present the language describing timed Petri nets and discuss the implementation of the two major tools. Examples are given to illustrate their use.

4.1. Hierarchical Timed Petri Net Language

HTP is a language designed to hierarchically describe timed Petri nets. A timed Petri net consists of a set of places, a set of transitions, a set of directed arcs connecting transitions to places and vice versa, a real number associated with each transition which is its firing time, and a marking which indicates the number of tokens initially at each place. In addition to HTP's ability to represent each of these items, it also allows for parameterization, conditional expressions, and repeat loops. Capabilities also exist to allow groupings of places and transitions in the form of arrays. The grammar for this language is given in Appendix A.

The basic strategy used when describing a net with HTP is successively finer detailing until individual places and transitions are defined. By this method even large, complex nets can be described with relative ease. The mechanism used to accomplish this hierarchical description is the *subnet,* which is analogous to the subroutine in a conventional programming language. A subnet is capable of describing an arbitrarily complex net which can be used many times in the overall description of the net. Of course subnets may themselves refer to other subnets. To allow a "calling" net access to the internal components within a subnet, each has a defined set of input and output ports. In describing a Petri net, a subnet's name and a port name are used to refer to a specific place or transition within that subnet. In reality places and transitions themselves are simply special subnets. Each has one input and one output which are used to access the place or transition. They are also the quantum units of the language in that every reference used in a connection must evaluate to an individual place or transition.

This section serves as a programmer's (or more accurately "describer's") manual for HTP, providing information about the language as well as examples of its use. While many HTP features differ from programming languages, there are common features. In these areas readers familiar with the C programming language [KeR78] will notice similarities.

The major components of HTP are *parameters, expressions, declarations, statements,* and *global definitions.* Parameters hold values during the defining process so they may be reused and combined with other parameters. Expressions may be used to manipulate parameters and constants. Declarations define the components of a net and associate names with them. Statements operate on these components. Finally global definitions syntactically hold the definitions of the net or its subnets.

4.1.1. Lexical Conventions

This subsection presents some lexical conventions used in HTP. All correspond to equivalent C language constructs.

Comments

Comments start with the character sequence "/*" and end with the sequence "*/". Any characters in a comment are ignored by the compiler. Comments do not nest.

Identifiers

An identifier is any sequence of characters starting with a letter and consisting of the letters, numbers and the "_" character. All characters in an identifier are significant. An identifier may not be a keyword.

Keywords

The following are the reserved keywords in HTP, and may not be used as identifiers.

else	**model**	**repeat**
if	**output**	**subnet**
input	**place**	**trans**

Constants

Integer constants may be expressed in either decimal, octal, or hexadecimal notation, using the same conventions as used in the C language. Octal numbers are signified by a leading "0" character, hexadecimal numbers by the sequence "0x". Floating point constants are only accepted in decimal notation. The precision of constants is machine dependent.

4.1.2. Expressions

Expressions allow the combination of parameters to form new parameters. All expressions correspond to equivalent C language expressions. Expressions may be parenthesized to indicate precedence, which is left to right if unspecified.

Arithmetic Operations

Addition, subtraction, multiplication, division, and modulus operations are defined using the conventional symbols : "+", "-", "*", "/", and "%" respectively. Each combines the expressions on its right and left, performing the appropriate operation.

Logical Operations

The logical AND and OR operations are defined using the "&&" and "||" symbols respectively. These operations are logical, as opposed to bitwise, producing either 1 for true or 0 for false. For example, the expression "A && B" would produce 1 if A and B were both nonzero expressions, 0 otherwise.

Comparison Operations

The comparison operations yield either 1 indicating true or zero indicating false based on the comparison of two expressions. Implemented operations are equal; not equal; less than; greater than; less than or equal; and greater than or equal. Symbolically these operations are represented by "==", "!=", "<", ">", "<=", and ">=" respectively.

4.1.3. Declarations

Declarations define the places, transitions, and subnets to be used in forming a net or subnet, as well as the input and output ports of the net or subnet. The general format of a declaration is:

type decl_list ;

where *type* is one of **trans, place, input, output,** or **subnet** and indicates which item is being declared, and *decl_list* describes the names and parameters of each item. It takes the form:

name1 (param1, param2 , ...) , name2 , name3 [size1] [size2] ...

where one or more items may be declared, perhaps with parameters assigned to the items declared. A multi-dimensional array may be declared using the square bracket convention. Parameters will be used to specify the firing time or initial marking for each transition or place.

trans declaration

The " **trans** *decl_list*" statement defines one or more transitions. One parameter is allowed which indicates the firing time of the transition. The default value is 1. In an array of transitions, each element has the same firing time. Each transition has two hard-wired ports, one input called "i" and one output called "o". These are used when forming connections to the transition.

place declaration

The " **place** *decl_list*" statement defines places. Two parameters are allowed which indicates the weight of the place and the initial marking. The default value of the weight is 1, while the default of the marking is zero. In an array of places, each element has the same weight and marking. Each place has two hard-wired ports, one input called "i" and one output called "o".

subnet declaration

The " **subnet** *name decl_list*" statement defines occurrences of the subnet with the name *name*. No parameters are allowed. The subnet mentioned must be defined elsewhere in the description. Recursive definitions are not allowed.

input declaration

The " **input** *decl_list*" statement defines input ports to the current definition (i.e. net or subnet). No parameters are allowed.

output declaration

The " **output** *decl_list*" statement defines one or more output ports to the current definition. No parameters are allowed.

4.1.4. Statements

Statements are used to define a timed Petri net. They perform the "action" of forming a complete net when compiled.

assignment statement

A new parameter can be created from previously defined parameters and constants using the assignment operation:

param_name = expression ;

Parameters may be redefined at any time, with the new value replacing the old. Parameters do not carry a specific type and do not require declaration.

connect statement

The connection statement is used to connect one or more declared ports to one or more others, using the "->" symbol.

out_port_list -> in_port_list ;

In this statement each list is a comma separated list of ports, either of the form "portname", previously declared as an input or output port; or as "itemname.portname", where itemname is the name of a previously declared place, transition, or subnet and portname is the name of an input or output port associated with that item. If an item on the left side of the operator evaluates to a place, the right side must evaluate to transitions. Alternatively, if an item on the left side of the operator evaluates to a transition, the right side must evaluate to places. One of the list must have only one element.

repeat statement

The repeat statement is used to repeat a group of statements several times, with a parameter indicating which loop instance is currently executing.

```
repeat (name,min_expression,max_expression ) {
    statements;
}
```

In this case *name* is a parameter which is created by the execution of the **repeat** statement. It has a value which ranges from *min_expression* to *max_expression,* one per instance of the loop. Therefore a total of *max_expression* − *min_expression* instances of the body will be created. Each instance is independent of all others.

if-else statement

The if-else statement is used to conditionally execute a group of statements based on the outcome of a conditional expression. The else portion of this statement is not required.

```
if ( conditional_expression ) {
    statements;
} else {
    statements;
}
```

4.1.5. Global Definitions

A HTP net is made up of one model definition and zero or more subnet definitions.

model definition

One model definition is required in each net definition and serves as the base of expanding the net. The model_name is used only for external identification.

model *model_name* {
  declarations ;
  statements ;
}

Note that items must be declared before used.

subnet definition

Any number of subnet definitions may be used in defining a network. Each time the subnet is referred to in another definition, a copy is made. The subnet_name is used to refer to the subnet in declarations. Subnets may not refer to themselves.

subnet *model_name* {
  declarations ;
  statements ;
}

4.1.6. Examples

Figure 4.1 shows HTP definitions produce the Petri net shown in Figure 3.3b. Note that the subnet "seq" will appear three times in the net produced by the compiler, called s1, s2, and s3 respectively. The connections and declarations inside are automatically duplicated in each instance. Appendices B through E show more complex examples of HTP net descriptions.

4.2. HTP Compiler

Once a Petri net has been defined in HTP, it is "compiled" into an internal format by the program **mknet.** The compiler has three distinct phases in processing the input description of a Petri net. The first phase simply parses the input description into a tree structure to allow more efficient manipulation. The second phase expands the description into a single flat network of places

```
SEQTIME = 1;                        /* a parameter definition          */
model figure3_3_b {                 /* the model is called figure3_3_b */
        input i1,i2;                /* 2 inputs                        */
        output o;                   /* a single output                 */
        trans t1(7),t2(3),t3(4),t4(0); /* declare 4 transitions        */
        place p1(1),p2(4),p3(0),p4(0); /* declare 4 places             */
        subnet seq s1,s2,s3;        /* 3 copies of subnet seq          */

        i1,i2 -> t1.i               /* the inputs connected to t1      */
        t1.o -> s1.i,p1.i,p2.i;     /* t1 out to 1 seq and 2 places    */
        s1.o,p1.o -> t2.i;          /* seq and one place to t2         */
        t2.o -> p3.i,s2.i;          /* t2 out to 1 place and 1 seq     */
        p2.o,s2.o -> t3.i;          /* p2 and s2 out to t3 in          */
        t3.o -> p4.i,s3.i;          /* t3 out to p4 and s3             */
        p3.o,p4.o,s3.o -> t4.i;     /* p3,p4,s3 all to t4              */
        t4.o -> o;                  /* t4 is the net output            */
}
/*
 * a subnet to handle a sequencing operation
 */
subnet seq {                        /* the subnet is called seq        */
        input i;                    /* one input...                    */
        output o;                   /* ... and one output              */
        place p1,p2;                /* two places                      */
        trans t(SEQTIME);           /* one transition, param time      */

        i -> p1.i;                  /* in -> p1 -> t -> p2 -> out      */
        p1.o -> t.i;
        t.o -> p2.i;
        p2.o -> o;
}
```

Figure 4.1
HTP Description of Figure 3.3b

and transitions with appropriate interconnection. The final phase assigns a unique number to each node in the network and produces an output file which is a simple node list. This section describes each phase in some detail, discussing the algorithmic complexity of each phase. To allow the analysis of the largest possible nets, the algorithmic complexity must be kept at a minimum.

During the first phase the input description is parsed using a simple parser built by the YACC compiler compiler [Joh75] and LEX lexical analyzer [Les75]. These tools are generally available under the UNIX operating system. Each declaration, statement, and definition causes the creation of a node in a tree data structure. In this data structure each definition has as its children all statements within that definition. Similarly statements which may syntactically contain other statements (i.e. the **repeat** and **if** statements) have as their children the statements they contain. Finally statements requiring lists such as declarations and connections have as their children the elements of those lists. When a symbol is defined by definition or declaration it is placed in a symbol table and referenced to the item it defines (e.g. a subnet or transition.) During this phase symbol conflicts and syntax errors are detected and reported to the user.

The tree structure allows the easy expansion of subnets and other structures by simply replicating the internal structure of their definition. In assessing the complexity of this phase the two major components are building a symbol table of N_{sym} symbols and creating a tree of N_{state} statements. As N_{sym} is generally small, the symbol table is implemented as an unordered list in which the lookup and insertion of each symbol requires $O(N_{sym})$ time and $O(1)$ space. Therefore symbol table complexity is $O(N_{sym}^2)$ time and $O(N_{sym})$ space. The parsing of statements and creating nodes for them is a simple one pass process requiring $O(1)$ time and space per statement. Adding the symbol and statement complexity, the complete phase requires $O(N_{sym}^2 + N_{state})$ time and $O(N_{sym} + N_{state})$ space. The number of statements and symbols used in describing a Petri net is related to how well the describer can break the net into subnets. In general this complexity is small as even very large nets do not require many statements or symbols. Experience verifies this result as the first phase requires a very small portion of the total execution time of the mknet program.

The second phase of this program is the most complex and where the most attention to performance needed to be placed. The process begins by simply "expanding" and "executing" the model definition. Expansion of a net involves creating all the places and transitions within it, while execution creates the connections between them.

For each place and transition declared in a definition, a duplicate of a master place or transition node is created and appropriate firing times or weights assigned. If an array of places or transitions is declared, a multi dimensional tree of duplicates is created each node having appropriate weights or firing times. Subnet declarations cause the immediate expansion and execution of the subnet definition in question if it has not already been processed. Obviously if a subnet definition declares other subnets they too will be expanded and executed in a recursive manner. After a subnet's processing is complete, its tree is pruned to remove all information not required for duplication and I/O port accessing. This step significantly reduces both the space and time requirements to copy the structure. The remaining subnet definition is duplicated as many times as required to fulfill the declaration. Input and output declarations create simple nodes which are used by external statements to access the places and transition of a net or subnet.

After all declarations have been expanded, the statements of the definition are executed. The connect statement will simply cause the connection of places and transitions. Error reports are issued if a statement connects a place to a place or a transition to a transition. Also, either the source or destination list must have only one element, and all nodes and ports must be defined. The body of an **if** statement will only be executed if the condition is true. A **repeat** statement simply executes the loop the specified number of times after defining a new local symbol. The symbol is removed after the execution of the body.

In assessing the complexity of this phase of the compilation process, consider the costs of two operations: the duplication of defined items and the execution of connections and other statements. Each node (i.e. place or transition) requires $O(1)$ space and time to define and duplicate it. If N_{node} nodes exist in the final net, then the space and time requirements will be $O(N_{node})$. The additional space requirements of the original nodes were either accounted for in the parsing phase (for subnets) or $O(1)$ (for places and transitions.) The connection processes requires two symbol lookups for each connection statement executed.

Note here that because of our hierarchical definition there is not a connection statement per node as subnet connections are performed before subnet duplication. Assuming $N_{connect}$ connections statements are executed, the time complexity is $O(N_{connect} * N_{sym})$. No additional space is required for this operation. Finally, the execution of **if** and **repeat** statements can be assumed to require $O(N_{connect})$ as at most one **if** or **repeat** statement is in general executed per connection execution. In summary this phase requires $O(N_{connect} * N_{sym} + N_{node})$ time and $O(N_{node})$ space. Experience shows that in general N_{node} is much larger than $N_{connect}$, as each the hierarchical nature of HTP requires connections within a subnet be created only once, regardless of how often the subnet is used. Thus, the duplication time becomes dominant.

Finally, the third phase uniquely numbers the nodes in the network by recursively traversing it in $O(N_{node})$ time and then traverses it again to output each place and transition. This process is also used to check the connectivity of nodes in the net. Totally unused nodes are ignored by the traversal. The total time complexity of the entire compilation process is

$$O(N_{state} + (N_{sym} + N_{connect})N_{sym} + N_{node}).$$

In assessing this complexity, assume $N_{state} \approx N_{sym} \approx N_{connect}$ (which is the usual condition). The total time complexity reduces to

$$O({N_{sym}}^2 + N_{node}).$$

The total space requirements are

$$O(N_{state} + N_{sym} + N_{node}).$$

The only non-optimal portion of this performance is the symbol table lookup, but experience shows that as, N_{node} much larger than N_{sym}, the code overhead for a more complex lookup algorithm is not justified. Space requirements are optimal as each node, symbol, and statement requires storage.

4.3. HTP Analyzer

After an HTP description has been compiled the place and transition list is passed to the net analyzer, **anet.** This program is responsible for determining the measures described in Section 3. Several distinct phases make up this program, depending on the results desired. During first phase the net is read into a multiply linked data structure. The next several phases determine the critical path time, serial time, and optionally the critical path width. Finally results are output indicating the value of the various measures. This section describes the actions of these phases and their time and space complexity.

The input phase is implemented as a state machine which parses the list output of mknet. For each node read, one is created and connected according to the input specifications. Additionally, any parameters are assigned to the node. Any errors in syntax (which should not occur as **mknet** produces this input), are reported, as well as exceeding a compiled parameter for the maximum number of nodes allowed. This limit is currently set at 50,000 nodes but could be increased at a cost of about 40 memory bytes per node. As no searching is required this phase requires $O(N_{node})$ in both time and space. At the end of this phase a single extra place, connected to each transition with a non-zero firing time is created. This place, called the *resource limitation place* is used to control concurrency during analysis. The limitation is accomplished by placing a number of tokens, corresponding to the desired limitation, in the place. This results in allowing only this limited number of transitions to be in an active state simultaneously. This connection process has time complexity of $O(N_{node})$ and uses $O(1)$ space.

The next several phases perform composite firing time analysis, each phase with a different limit on the number of transitions which may fire simultaneously. In each case the time required to fire all transitions in the net is determined. The first analysis phase places no limit on concurrency (i.e. an infinite supply of tokens is available at the resource limitation place) and an analysis shows the critical path time. Serial time is next found placing a single token in the resource limitation place and performing another firing time analysis. Finally a binary search is performed to determine the number of resources required to achieve critical path performance. In each step of the search an analysis is performed. The lower bound on the limits of this search is the ratio of serial to critical path time, and the upper bound is the maximum number of transitions firing simultaneously during the critical path measurement. The search terminates when it finds the smallest number of resources required to achieve critical path performance.

Each analysis phase computes the firing time of the net with a specified limit placed on concurrency. The typical method for accomplishing this calculation would be to build a reachability tree for the net. A reachability tree is a graph in which each vertex represents a marking, edges connect markings reachable from other markings in which a single transition fires, and edges are weighted with the firing time of that transition. After construction of the tree, one would then find the shortest path through this tree from the initial marking to the final marking. Unfortunately the reachability tree grows exponentially with the number of nodes in a net, making this method unusable for all but the smallest of nets. In fact, any method which relies on searching the state space of net markings will be unsuitable for our goal of analyzing large nets.

To overcome the problem of state space explosion we have developed a heuristic approach to the problem. Several strategies are used to pick a single thread of the reachability tree between the initial and final markings. In each pass, one of these strategies is chosen and the minimal time over all strategies is taken as the composite time. These strategies, which are similar to scheduling strategies in multiprocessors, allow us to quickly analyze the net while achieving a close approximation to the actual shortest path in the reachability tree.

The approach to each pass of the analysis phase (during each of which one of the heuristic strategies is applied) is an iterative approach. First the net is restored to its initial marking which is saved during the input phase. A number of iterations then take place in which three steps occur. First a list of candidate transitions is found. A candidate node is one that has at least one token in each input place. The list is then pruned using the strategy to a firing list, which is conflict free. This achieves the goal of reducing the tree fanout to one. Finally the firing list is fired. The firing of transitions is accomplished by an event driven simulation. First a token is removed from each input place. Then an event is scheduled to occur after the firing time has elapsed. When this event occurs, tokens are given to each output place. These steps iterate until the candidate list is empty.

In examining the complexity of the analysis process, note that each transition is fired once. The time complexity is thus limited to $O(N_{node}{}^2)$ as there will be at most N_{node} searches for candidate nodes, each taking $O(N_{node})$ time to complete. However, due to fanout, fewer candidate lists are actually formed. Also, a faster search for candidate transitions is employed by maintaining a list of potentially active transitions (those with a token in at least one input place.)

Under ideal conditions this can lower the time complexity to $O(N_{node}\log N_{node})$. No additional storage is required for this operation. The total analysis phase will require $O(N_{pass}N_{node}^2)$ time, were N_{pass} is the number of iterations required to determine the critical path width. Due to the binary search used, N_{pass} is $O(\log cpath\_width)$.

The final phase simply reports accumulates data for further processing, in constant time. Combining the time and space requirements for this entire tool, the time complexity has a best case of

$$O(N_{pass}N_{node}\log N_{node}),$$

and a worst case of

$$O(N_{pass}N_{node}^2).$$

Space complexity of the algorithm is $O(N_{node})$. As an example of the execution times, Table 4.1 shows execution time required to analyze various net sizes on a CCI 6/32 computer system.

Table 4.1

Analyzer Execution Times

N_{node}	N_{pass}	Execution Time (seconds)
1500	7	15
4500	8	60
10000	9	250

4.4. Future Tools and Conclusions

The tools described in this section perform the most mechanical and longest tasks in the analysis of COSMIC models. They allow us to analyze realistically sized problems and perform a wide variety of experiments on them. However, we would like to expand our set of tools to allow the automatic generation of ordering nets from a higher level description of the algorithms, ordering scheme, and organization. This tool would allow the study of a much wider variety of algorithms than can currently be generated. The next section describes several experiments performed using the tools described in this section. Finally, it should be noted that the programs described here should be portable to most UNIX environments and are available from the authors.

5. Examples Using COSMIC

The use of COSMIC is illustrated by several experiments conducted to study the behavior of iterative algorithms on combined data flow and control flow multiprocessors [CaF87]. These experiments were performed on a hypothetical architecture capable of executing instructions under a variety of control schemes ranging from control flow to data flow. The major variable of experimentation was the number of partition elements (segments) and consequently the granularity. This report presents a fragment of these results to give a flavor of COSMIC's use. More complete results are available in [CaF87]. This section proceeds by describing the organization, data dependency graphs, and the various functions of the ordering scheme that manipulate them. The numerical results from these experiments are presented graphically and in the form of polynomial equations.

As a compromise between the infinite variability of this hypothetical architecture and the availability of computational resources to analyze systems, a restriction is imposed on the experiments. Specifically, the scope of the analysis is limited by assuming resource allocation constraints will be ignored. This will lead to the resource allocation function being set equal to the identity function. In turn, this can be justified by assuming equally fair and efficient implementations on all systems. However, resource allocation factors may effect system performance and ongoing research is aimed at eliminating the restrictions.

5.1. The Organization

The organization parameters of consequence for the hypothetical architecture are the number and speed of the processing elements and the speed of memory access. Both are treated as variables in these experiments. It is also assumed that all processing and memory elements are interconnected, with a parameterized fixed communication cost from any source to any destination. Future research is planned to investigate the effects of interconnection topology on combined system performance by a more complex model of interconnection.

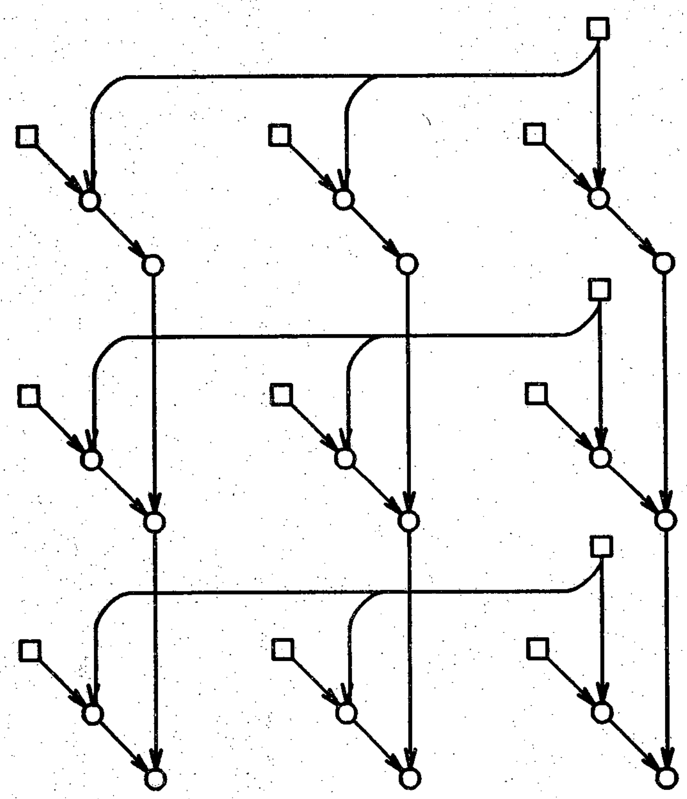
5.2. The Dependency Graphs

The first algorithm studied is for matrix-vector multiplication using the algorithm shown in Figure 5.1a, in which the matrix has size ($SIZE \times SIZE$). In forming the data dependency graph for this algorithm, note the central operations in the algorithm are the multiplication of two numbers and then the addition of the result to a running sum. This central operation will occur $SIZE^2$ times in the dependency graph. Therefore, a base structure is created to connect two vertices by a directed arc. The vertex at the tail of the arc represents the multiplication operation, while the vertex at the head represents the addition. Two arcs enter the multiplication vertex, representing the matrix/vector input values, and one additional arc enters the addition vertex to represent the previous value of the running sum. The addition vertex has a single output arc. Therefore, creating a dependency graph for the algorithm involves replicating this structure $SIZE^2$ times and interconnecting appropriately. Added to this graph are $SIZE$ vertices representing the input vector and $SIZE^2$ vertices representing the input array. Figure 5.1b shows such a graph for the case when $SIZE = 3$. In this figure the computational vertices are represented by circles and the input matrix/vector vertices by squares. Note that the input vertices are connected to the multiplication operation and the addition operations are chained to form the complete dot product operation.

The second algorithm studied computes a 4-point iterative relaxation function, using the algorithm shown in Figure 5.2a, in which the matrix has size ($SIZE \times SIZE$) and computes $ITER$ iterations. When all loops are unfolded into their basic components, a central computational block is again repeated many times throughout the algorithm. Here, the computational block consists of three additions and a division, therefore resulting in a 4 vertex graph with 4 inputs and one output. This basic graph is repeated $SIZE^2 \times ITER$ times and appropriate interconnections are made. As the dependency graph for the complete algorithm is complex, Figure 5.2b shows only the central computational block. In this algorithm indices which are out of the valid range of array subscripts "wrap-around" using the modulus function. For simplicity initial input arcs are ignored.

```
For i From 1 To SIZE Do
    For j From 1 To SIZE Do
        result[i] = result[i] + a[i,j] * b[j];
    EndDo
EndDo
```

(a)

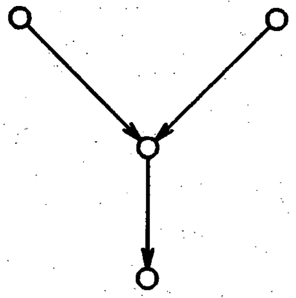


(b)

Figure 5.1
Matrix-Vector Multiply (a) Algorithm and (b) Data Dependency Graph.

```
For r From 1 To ITER Do
    For i From 1 To SIZE Do
        For j From 1 To SIZE Do
            a[i,j] = (a[i-1,j] + a[i+1,j] + a[i,j-1] + a[i,j+1])/4
        EndDo
    EndDo
EndDo
```

(a)



(b)

Figure 5.2
Iterative Relaxation (a) Algorithm and (b) Data Dependency Graph Fragment.

5.3. The Ordering Schemes

The experiments investigated two classes of ordering schemes. Both are two level combined approaches which require the partitioning of an ordering net into segments. The segment size is a variable for experimentation. The first ordering scheme, denoted *Cpart,* sequences segments using a control flow ordering scheme, while individual operations within a segment are sequenced using data flow concepts. The other ordering scheme, denoted *Dpart,* sequences segments using a data flow ordering scheme, while individual operations within a segment are sequenced using control flow concepts. In this section the specifics of the partitioning and sequencing functions will be discussed for each case. The generation function, τ, assigns firing times to the transitions it creates based on a parameter of the experiments called the *computation time*. Again note that γ, the resource allocation function, is the identity function.

The partitioning function, ϕ, is the same for both the Cpart and Dpart ordering schemes. As both algorithms have a grid structure, the initial ordering net is partitioned first by columns in that grid of operations, and then if required by rows. For example, if 3 segments were to be created from a matrix example with *SIZE* = 3, each column in the grid of operations (see Figure 5.1b) would be placed in its own segment. If six segments were required, then each of those segments would be divided in two. This strategy keeps operations that communicate most often in the same segment whenever possible. Synchronization operations are then placed between each pair of connected computational vertices in different segments. The firing time of the additional transitions is variable and called the *synchronization time*.

The sequencing function for the Cpart ordering scheme, λ_C, creates sequencing operations to cause segments to be sequenced using control flow techniques. When more segments are created than columns in the grid structure of operations, the segments are sequenced so that in each group of independent segments must entirely complete before the next group is started. To this end λ_C also forces each ply of segments to complete before starting the next, enforced by adding a single transition and many connecting places between plies. A data flow sequencing operation is placed in parallel with each unsynchronized place, to enforces a low level data flow scheme. The firing times of the additional transitions is variable and called the *sequencing time*.

The sequencing function for the Dpart ordering scheme, λ_D, enforces data flow sequencing amongst the segments, which is already accomplished by the previously added synchronization operations. To enforce control flow sequencing within segments, it places a sequencing operation between operations within segments to assure that no concurrency will take place within a segments (i.e. a single trace of operations is executed serially.) The firing time of the additional transitions is again called the *sequencing time*.

The memory access function for both ordering schemes, μ, simply replaces each non-zero weighted place with a memory access operation whose transition's firing time is called the *memory access time*.

5.4. The Ordering Nets

This section presents the parameterized ordering nets for both algorithms and ordering schemes. These ordering nets are the result of applying the ordering scheme function just described to the appropriate organization and data dependency graph. Appendices B through E give the complete, parameterized nets for the two algorithms and two ordering schemes. Each Petri net description has a main definition which describes its overall structure. Subnets are included to describe the exact operation at each point in the grid, as well as operations for synchronization, sequencing, and memory accessing.

The Petri net described in Appendix B is for the CPART ordering scheme and matrix multiply algorithm. Parameters specify the size of problem, number of partitions, and costs for each type of operation. These parameters are the ones varied in the experiments. Next the model defines all the places, transitions, and subnets used to create the net, and the overall structure is formed by their connection. Note that repeat structures are used to allow variable sized nets to be constructed, and conditional expressions enforce sequencing and scheduling of operations based on the desired partitioning. The subnet "innerprod" calculates a point of the grid of operations, and contains all sequencing and memory access details required. Finally, three simple subnets provide for a description of simple sequencing, synchronization, and memory access operations.

The Petri net described in Appendix C is for the DPART ordering scheme and matrix multiply algorithm. The structure of this description is quite similar to that shown in Appendix B. The differences occur in the sequencing operations which enforce the different ordering scheme and require different connections in the model definition. The parameters and operation subnets are identical to the CPART case.

The Petri net described in Appendix D is for the CPART ordering scheme and iterative relaxation algorithm. An additional parameter describes the number of iterations to be modeled. Again, the model defines all the places, transitions, and subnets used to create the net, and the overall structure is formed by their connection. Repeat structures are again used to allow variable sized nets to be constructed, and conditional expressions enforce sequencing and scheduling of operations based on the desired partitioning. The subnet "calc" calculates a point of the grid of operations, and contains all sequencing and memory access details required. Finally, three simple subnets provide for a description of simple sequencing, synchronization, and memory access operations.

The Petri net described in Appendix E is for the DPART ordering scheme and iterative relaxation algorithm. The structure of this description is quite similar to that shown in Appendix D. As with the matrix multiplication algorithms, the differences occur in the sequencing operations which enforce the different ordering scheme and require different connections in the model definition. The parameters and operation subnets are identical to the CPART case.

5.5. The Experiments

Four experiments were conducted to determine the system's sensitivity to changes in problem size and in the relative time required to execute computational, synchronization, sequencing, and memory access operations. In each case, the execution measure was determined, i.e. the triple $M_{execution}$. Two values were varied in each experiment: that corresponding to the experiment's name (e.g. sequencing time) and the number of segments (and consequently the size of every segment.) The problem size in these experiments was varied from 4 by 4 to 12 by 12. The segment sizes range from 1 to the problem size squared.

After numerical results from the experiments were obtained, those related to the critical path execution time were fit to polynomial curves based on the number of segments. Except for a few "off by one" errors at extreme segment sizes, all cases exhibit a piecewise linear relationship between the number of segments and the critical path performance of the algorithm. Next, several equations from experiments corresponding to variations of the time variables were combined to obtain polynomial equations for each measure based on both the number of segments and the time variables (e.g. sequencing time). Again all equations could be combined in a piecewise linear fashion. At this point in the analysis several equations represented each measure, one in terms of each time variable. These equations were then unified to a single equation for each measure in terms of all the time variables and the number of segments. These equations can be verified by substituting appropriate constant for the time variables to obtain the component equations. Finally, the results of experiments on different problems sizes were combined to obtain the final critical path equations for each measure.

The critical path measurement equations are shown in Tables 5.1 and 5.2 for the matrix multiplication and iterative relaxation algorithms respectively. In these tables (and the remainder of this paper), N represents the number of segments; S the problem size; T_c the computation time; T_{sync} the synchronization time; T_{seq} the sequencing time; and T_{ma} the memory access time. Also, note the ceiling function $\left\lceil x \right\rceil$ represents the smallest integer $\geq x$ and θ represents the unit step function:

$$\theta(x) = \begin{cases} 0 & \text{if } x<0 \\ 1 & \text{if } x\geq 0. \end{cases}$$

Figures 5.3 through 5.6 show graphical interpretations of the results of the experiments, for the 8 by 8 problem size, showing critical path time. In each figure four plots depict the family of curves resulting from plotting the number of segments versus critical path execution time for various values of one of the cost variables. Figures 5.3 and 5.4 are related to the matrix-vector multiplication experiments, while Figures 5.5 and 5.6 correspond to the relaxation experiments. Figures 5.3 and 5.5 are results for the Cpart ordering scheme, while Figures 5.4 and 5.6 show those from the Dpart scheme.

Examination of the measure equations yields a good understanding of the performance of these two algorithms. The matrix multiplication algorithm's computation measure is $\left(S+1\right)T_c$, which is easily explained by examining

Table 5.1
Matrix Multiplication Critical Path Measures

Measure	Equation
Computation	$\left(S+1\right)T_c$
Partitioning	$T_{sync} + \theta\left(N-2\right)\left\lceil \frac{N}{S} \right\rceil T_{sync} + \theta\left(N-S\right)\theta\left(S^2-N\right)\theta\left(T_{sync}-T_c\right)\left(T_{sync}-T_c\right)$
Sequencing (CPART)	$\left\lceil \frac{N}{S} \right\rceil \left(3T_{seq}+T_c-T_{sync}\right) + \left(S-1\right)T_{seq} + \theta\left(N-S\right)\theta\left(S^2-N\right)$ $\left(\theta\left(T_{sync}-2T_c-T_{seq}\right)\left(T_{sync}-2T_c-T_{seq}\right) - \theta\left(T_{sync}-T_c\right)\left(T_{sync}-T_c\right)\right)$
Sequencing (DPART)	$\left(S-\left\lceil \frac{N}{S} \right\rceil\right)T_c - \theta\left(N-S\right)\theta\left(S^2-N\right)\theta\left(T_{sync}-T_c\right)\left(T_{sync}-T_c\right)$ $+\,2\left(S+1-\left\lceil \frac{N}{S} \right\rceil\right)T_{seq} + \theta\left(T_{seq}-T_{sync}\right)\left(T_{seq}-T_{sync}\right)$
Memory Access (CPART)	$\left(2\,S+3+\left(3+2\theta\left(N-2\right)\right)\left\lceil \frac{N}{S} \right\rceil\right)T_{ma}$
Memory Access (DPART)	$\left(4\,S+6+\left(4-2\theta\left(N-2\right)\right)\left\lceil \frac{N}{S} \right\rceil\right)T_{ma}$

$$\theta(x) = \begin{cases} 0 & \text{if } x<0 \\ 1 & \text{if } x\geq 0. \end{cases} \qquad \left\lceil x \right\rceil = \text{Smallest integer} \geq x.$$

N – Number of Segments S – Problem Size T_c – Computation Time

T_{seq} – Sequencing Time T_{sync} – Synchroniztion Time T_{ma} – Memory Access Time

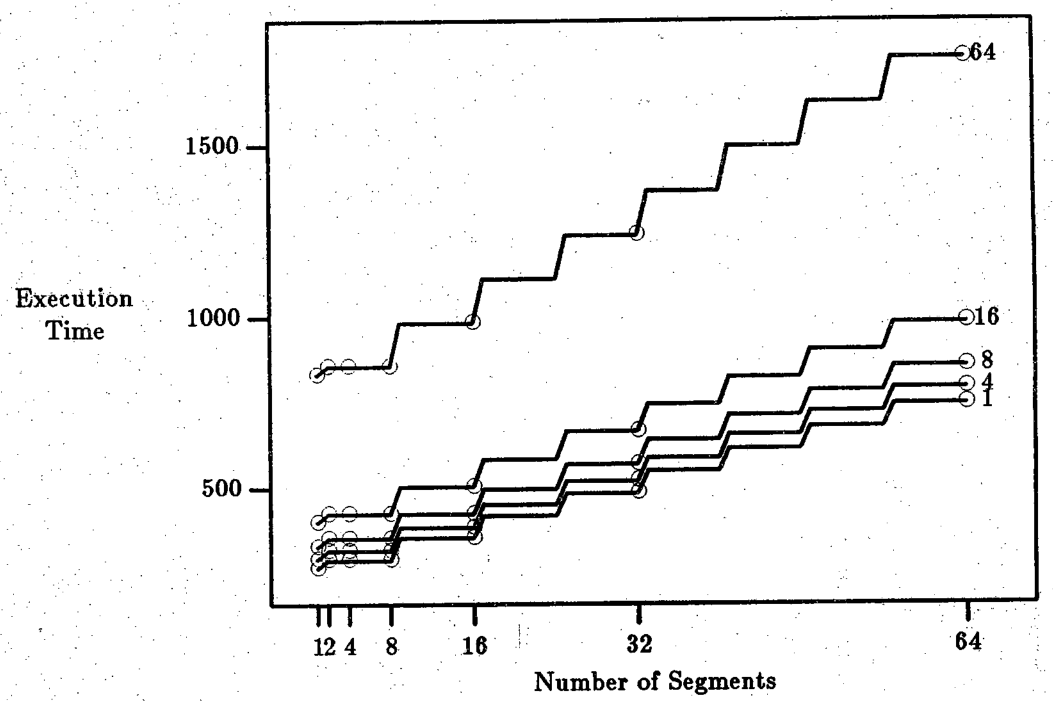
Table 5.2
Iterative Relaxation Critical Path Measures

Measure	Equation
Computation	$\left(12\,S - 3\right) T_c$
Partitioning	$\min\left\{ \left(3N - 3\right),\ \left(\frac{N}{S} + 3\,S - 4\right) \right\} T_{sync} + \theta\left(N - 2\right) 2\ T_{sync}$
Sequencing (CPART)	$\left(\left(9S - 9\right) \left\lceil \frac{N}{S} \right\rceil - 3S + 3 \right) T_c + \left(\left\lceil \frac{N}{S} \right\rceil \left(3S - 4\right) - 20 - \theta\left(N - 2\right) 2 \right) T_{sync}$ $+ \max\left\{ \left(18S - 6 - 3N\right),\ \left(9\,S + \left\lceil \frac{N}{S} \right\rceil \left(6S - 6\right)\right) \right\} T_{seq}$
Sequencing (DPART)	$\left(3\,S - 3 \left\lceil \frac{N}{S} \right\rceil + 2\theta\left(S - N\right)\right) T_c + \left(14\,S - 1 - 6 \left\lceil \frac{N}{S} \right\rceil + 4\,\theta\left(S - N\right)\right) T_{seq}$ $+ \left(2S - 2 - 2 \left\lceil \frac{N}{S} \right\rceil - \theta\left(S - N\right)\left(2S - 2 - 2N\right) + 2\theta\left(N - 2\right)\right) T_{syn}$
Memory Access (CPART)	$\left(18\,S + \left(18\,S - 18\right) \left\lceil \frac{N}{S} \right\rceil \right) T_{ma}$
Memory Access (DPART)	$\left(30\,S - 5 - \left\lceil \frac{N}{S} \right\rceil + \theta\left(S - N\right)\left(N - S + 4\right)\right) T_{ma}$

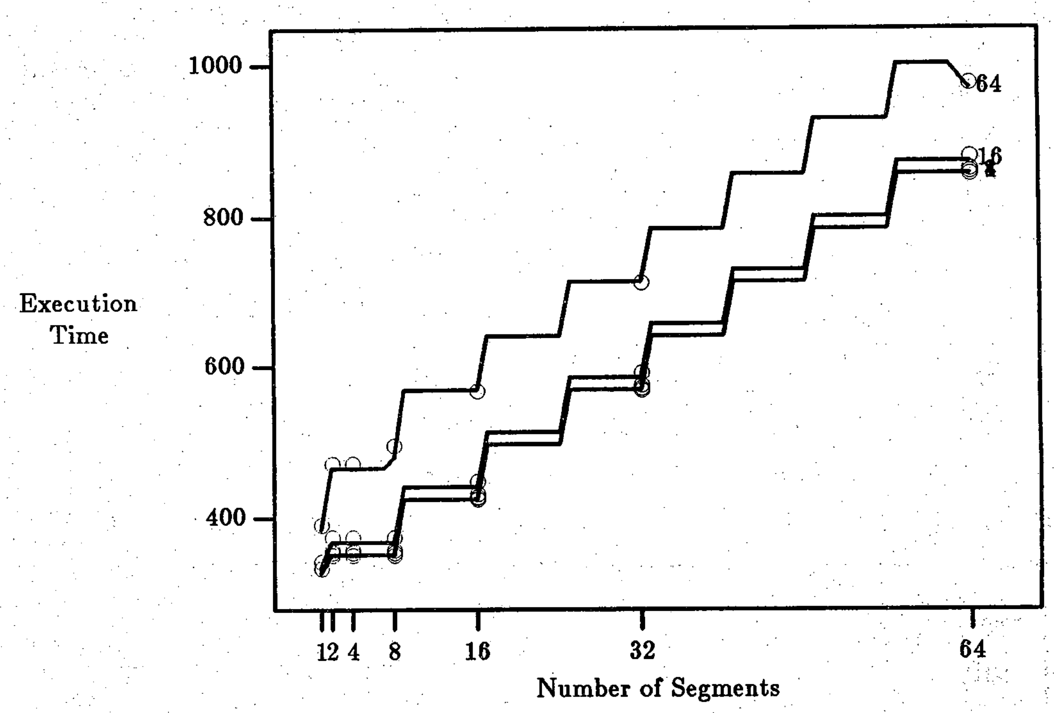
$\theta(x) = \begin{cases} 0 & \text{if } x<0 \\ 1 & \text{if } x\geq 0. \end{cases}$ $\left\lceil x \right\rceil$ = Smallest integer $\geq x$.

N – Number of Segments S – Problem Size T_c – Computation Time

T_{seq} – Sequencing Time T_{sync} – Synchroniztion Time T_{ma} – Memory Access Time



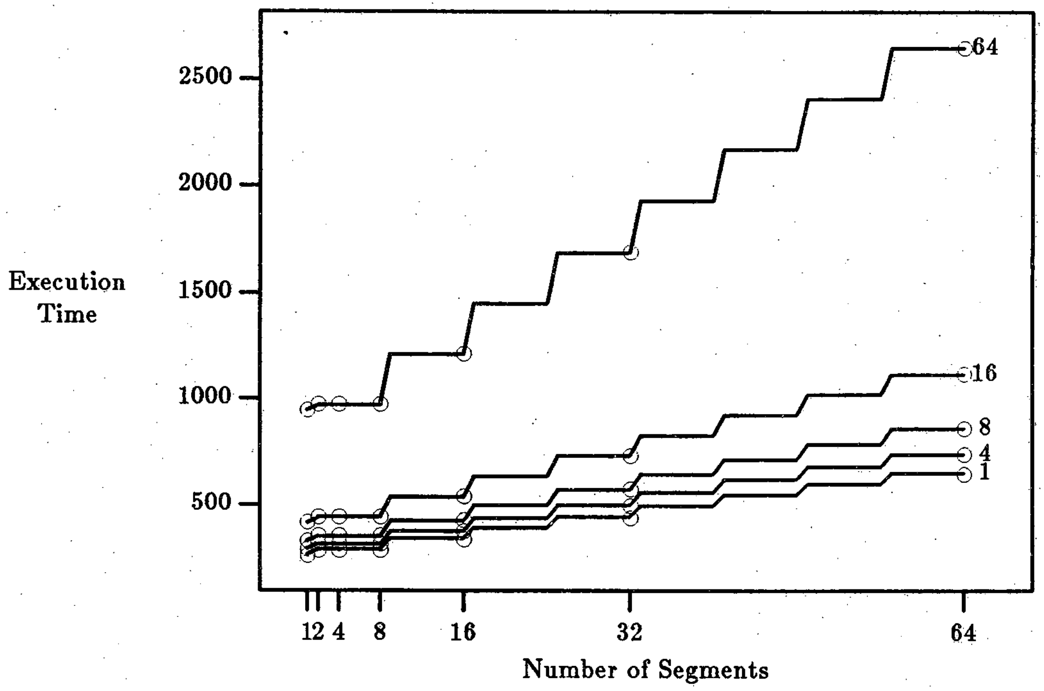
(a) Computation Time Experiment



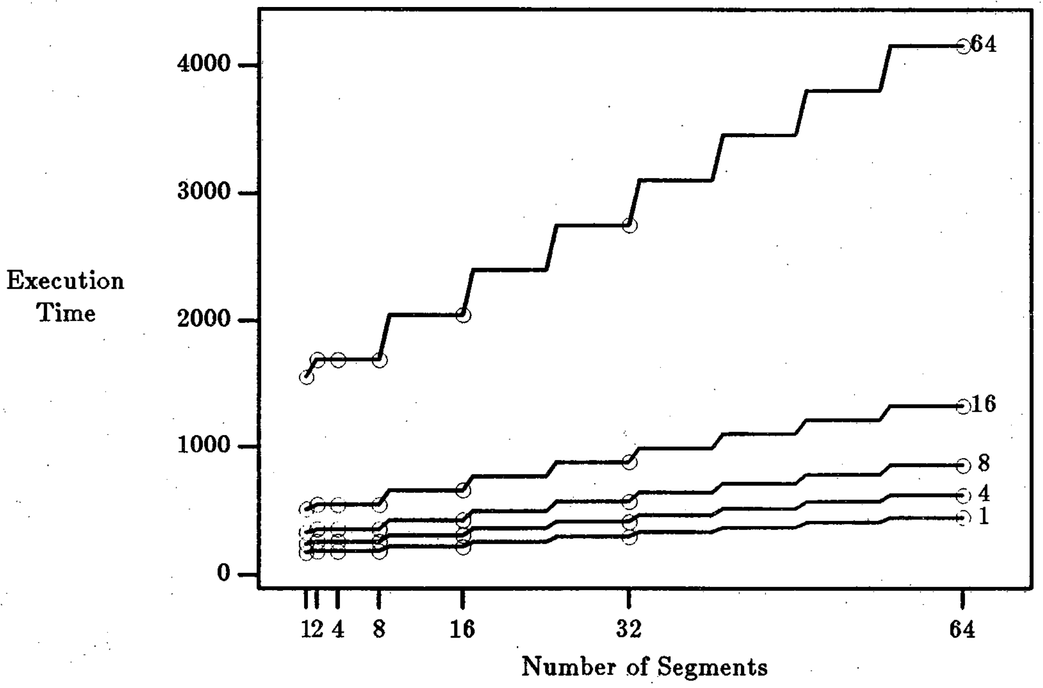
(b) Synchronization Time Experiment

Figure 5.3

CPART Matrix Multiplication Critical Path Execution Time. Circles indicate actual measures; curves show polynomial curve fit; and family of curves represent varied values of experimental time.

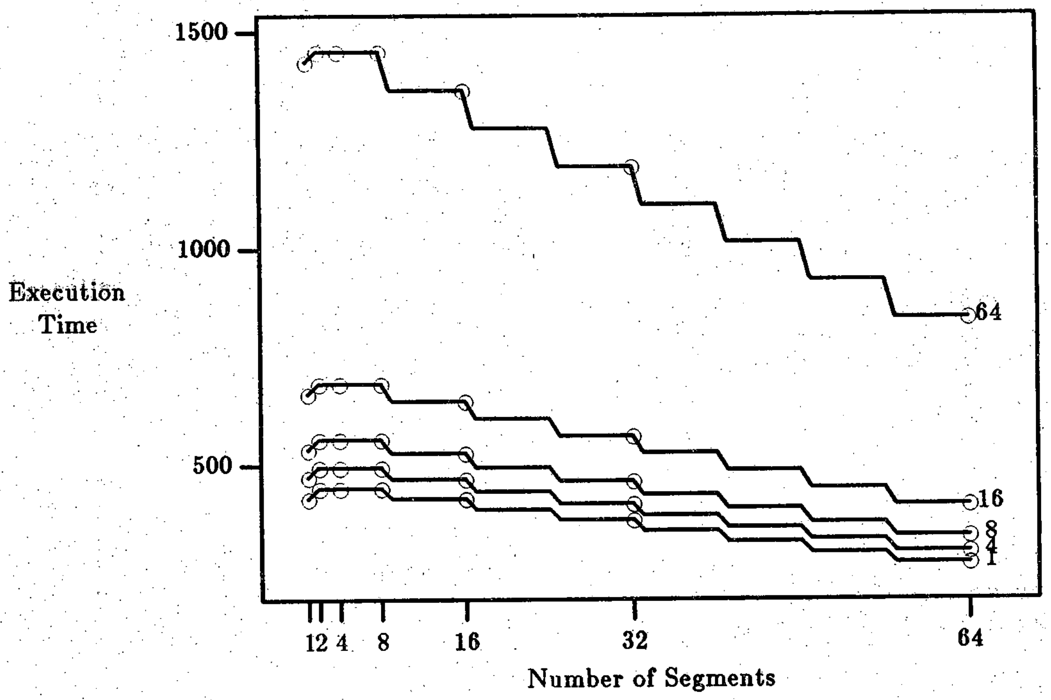


(c) Sequencing Time Experiment

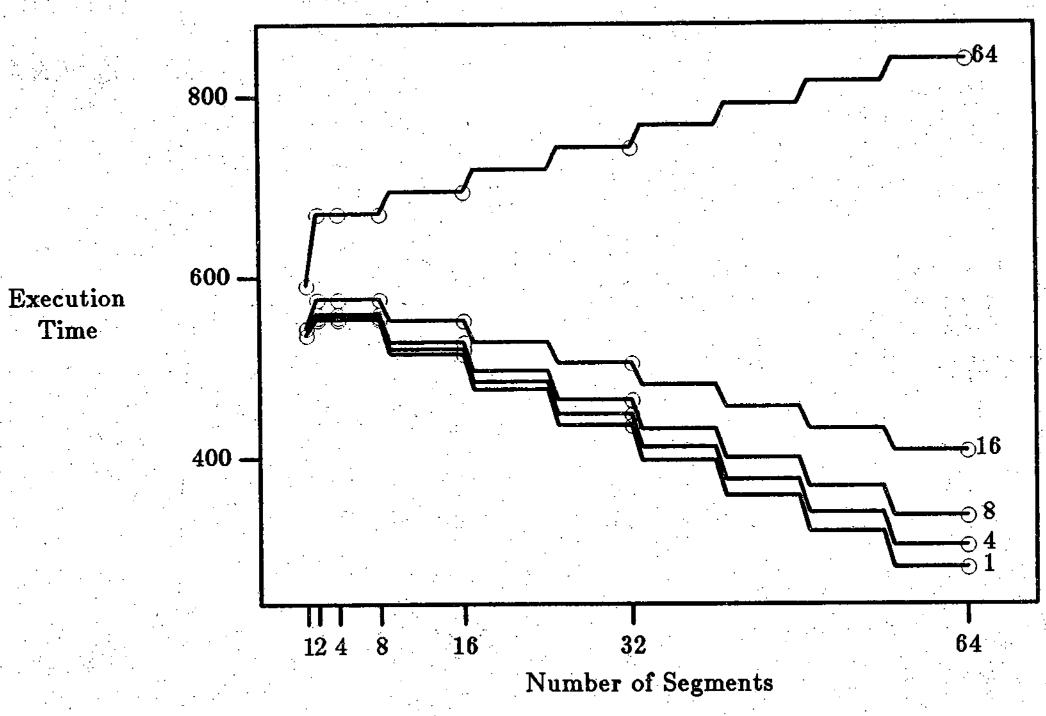


(d) Memory Access Time Experiment

Figure 5.3, continued.



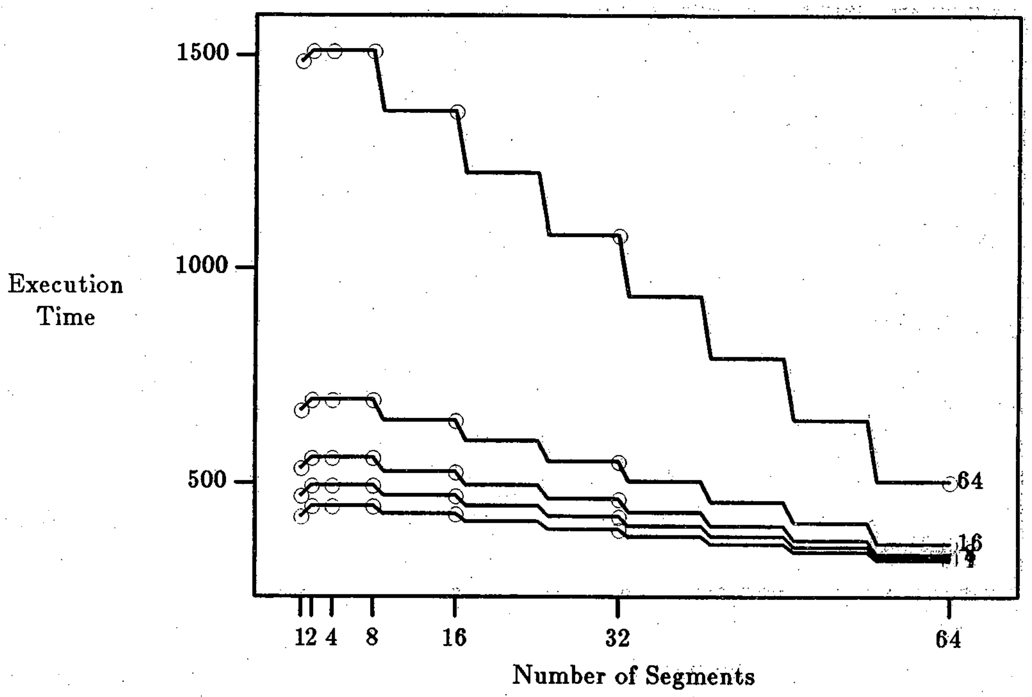
(a) Computation Time Experiment



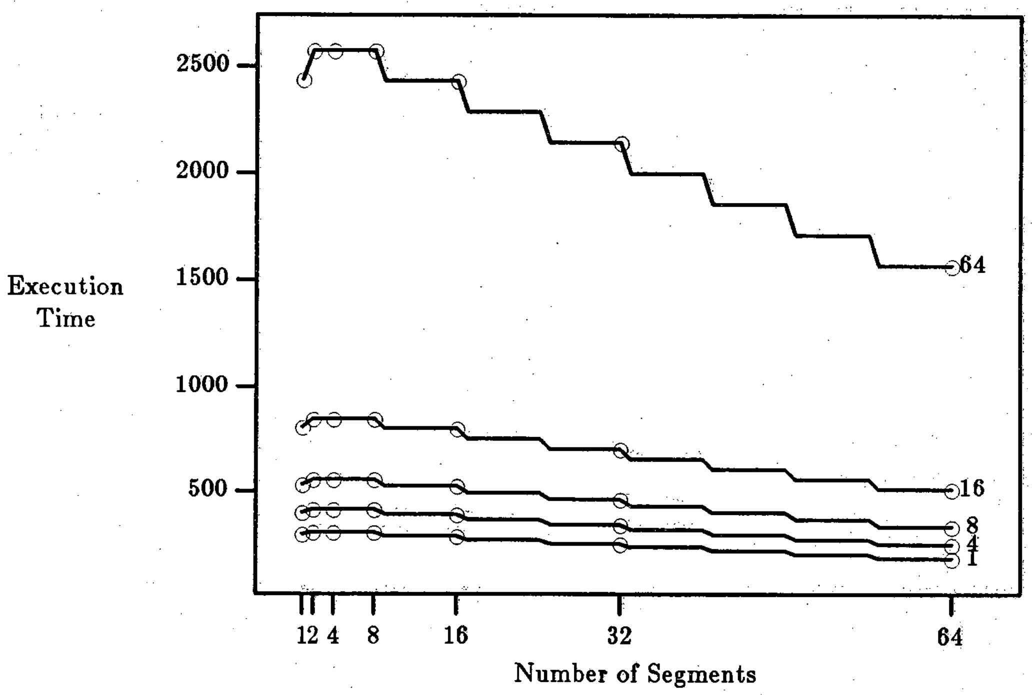
(b) Synchronization Time Experiment

Figure 5.4

DPART Matrix Multiplication Critical Path Execution Time. Circles indicate actual measures; curves show polynomial curve fit; and family of curves represent varied values of experimental time.

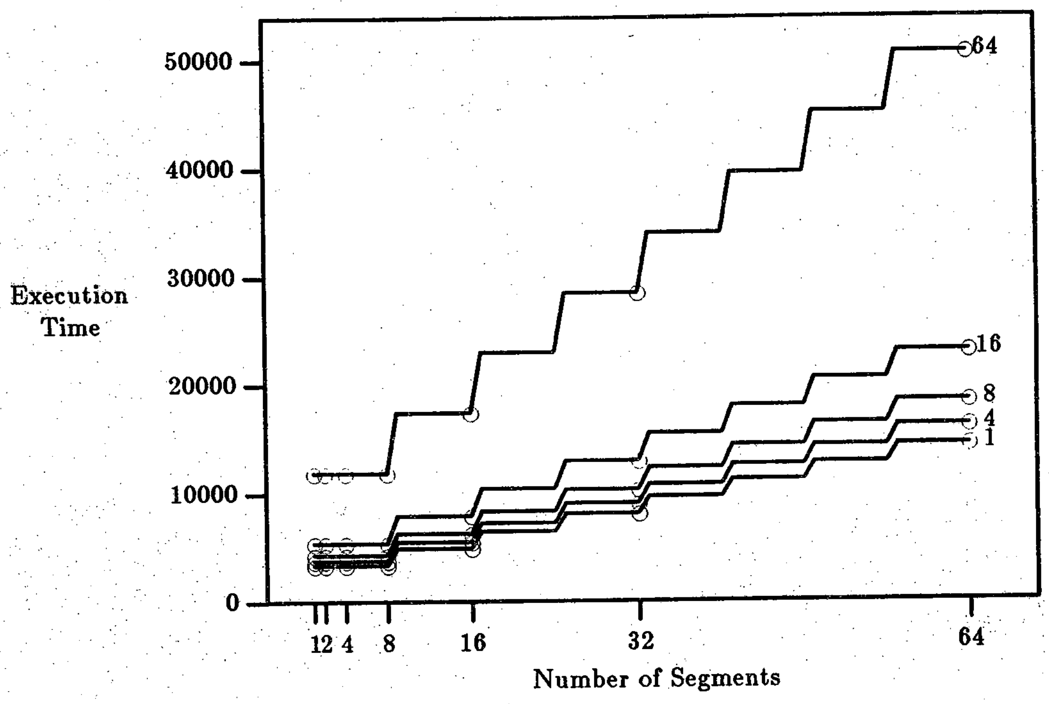


(c) Sequencing Time Experiment

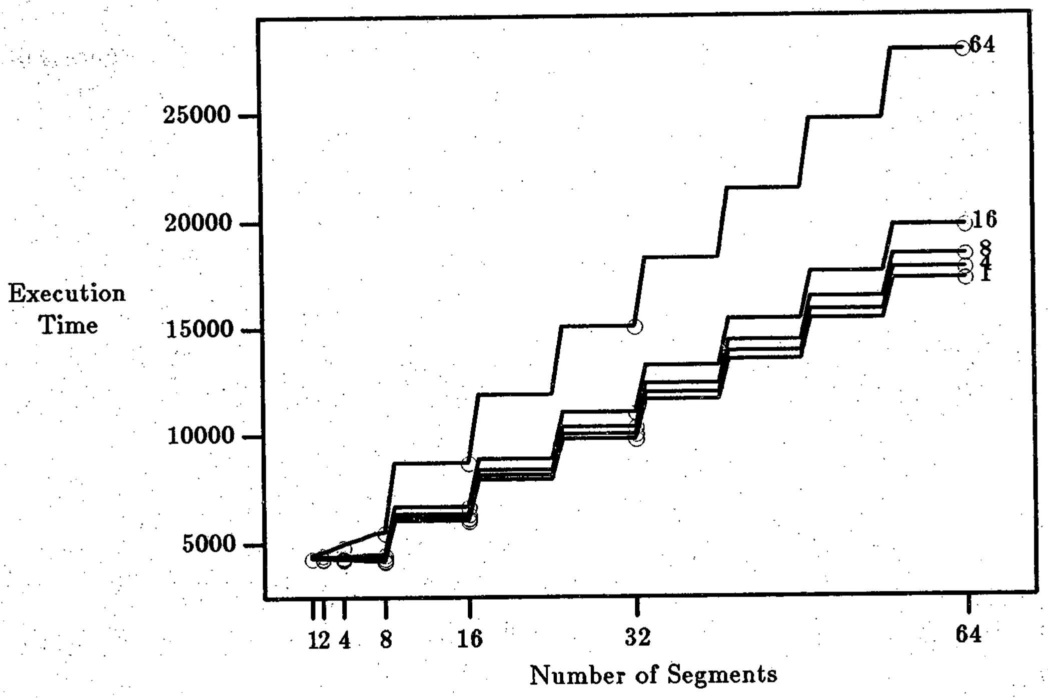


(d) Memory Access Time Experiment

Figure 5.4, continued.



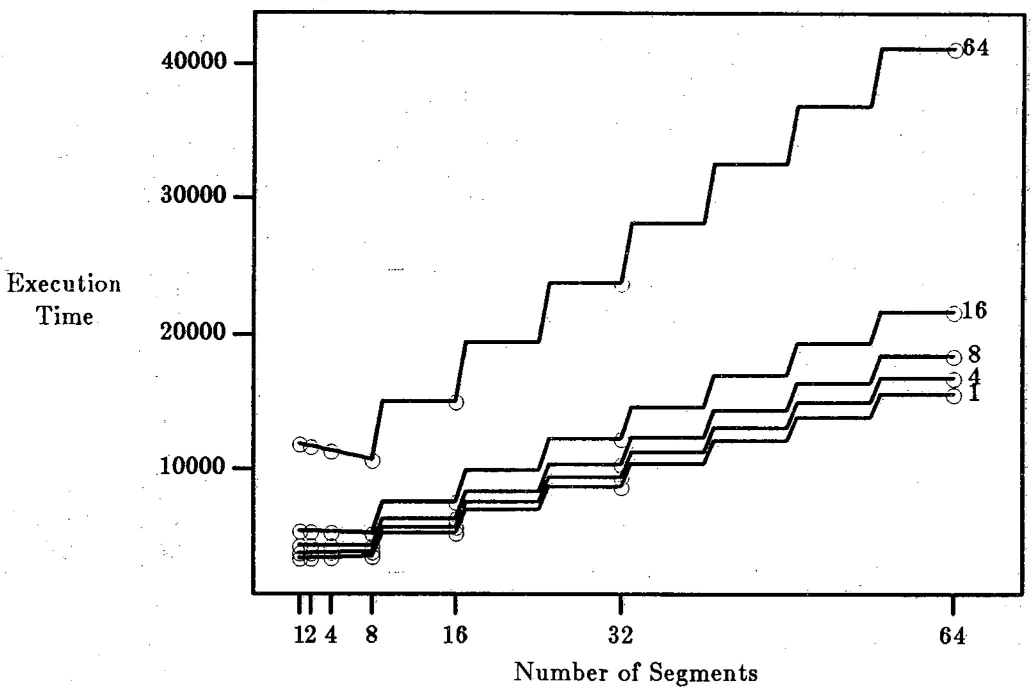
(a) Computation Time Experiment



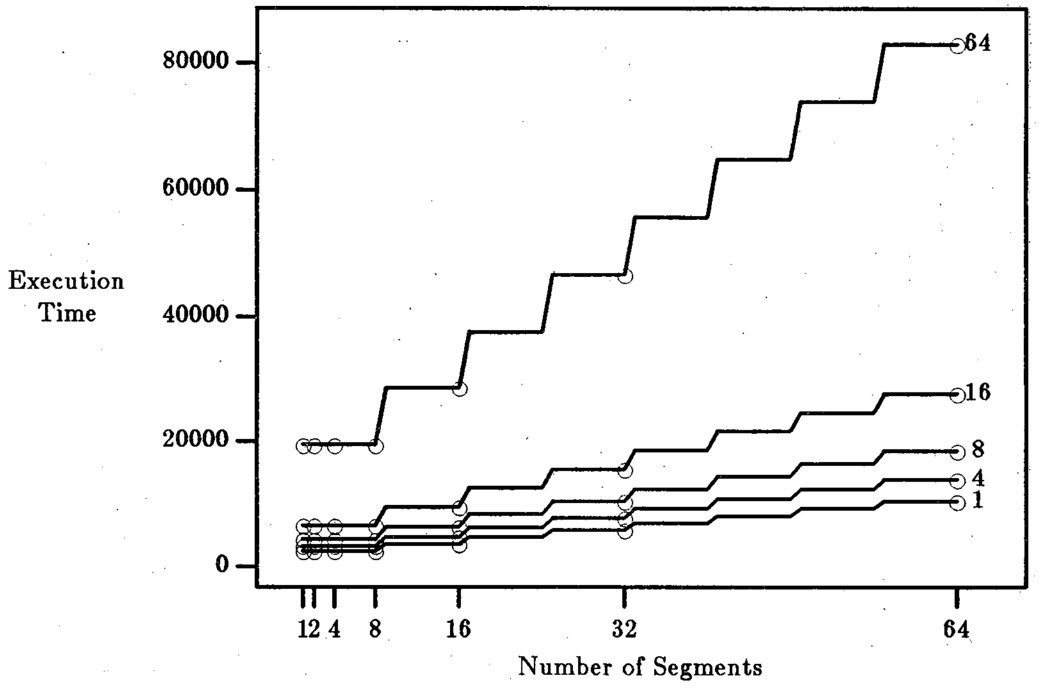
(b) Synchronization Time Experiment

Figure 5.5

CPART Iterative Relaxation Critical Path Execution Time. Circles indicate actual measures; curves show polynomial curve fit; and family of curves represent varied values of experimental time.

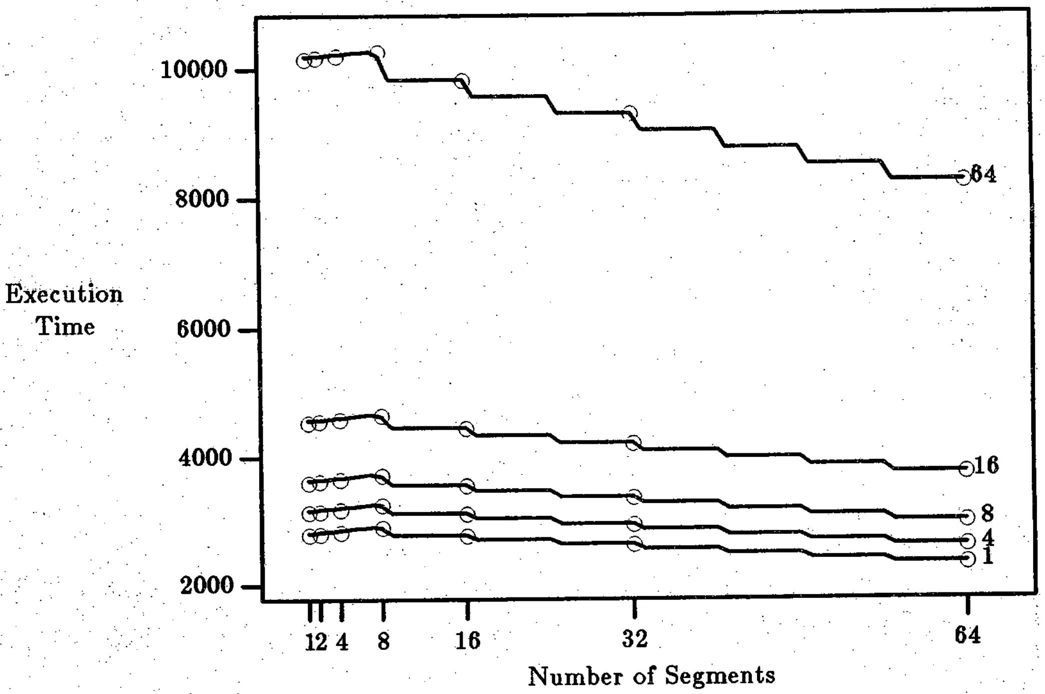


(c) Sequencing Time Experiment

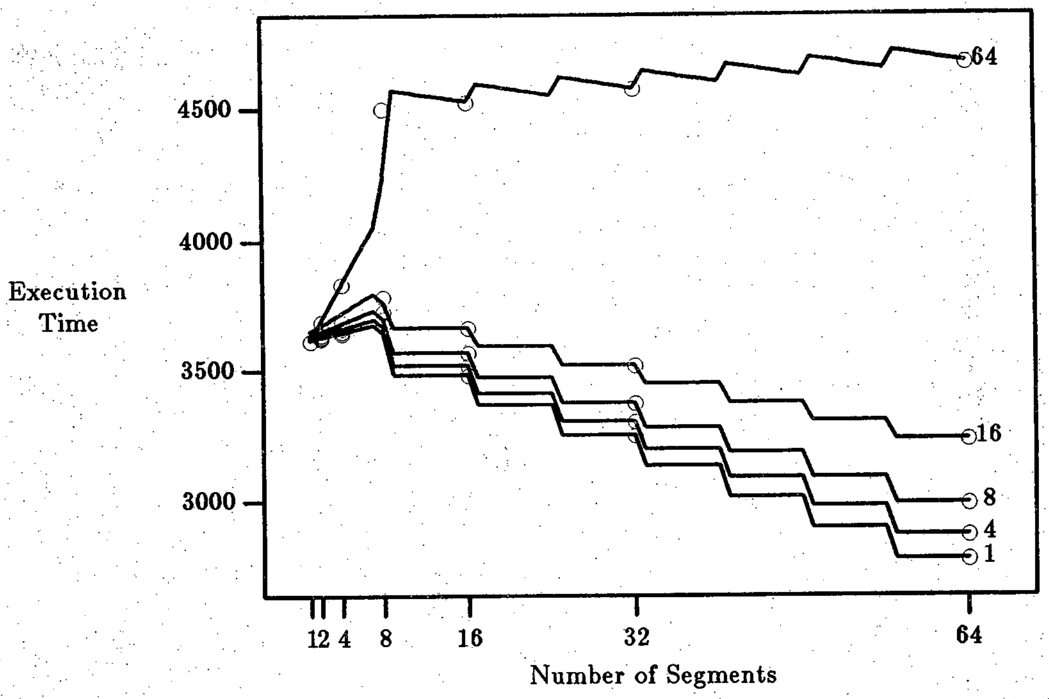


(d) Memory Access Time Experiment

Figure 5.5, continued.

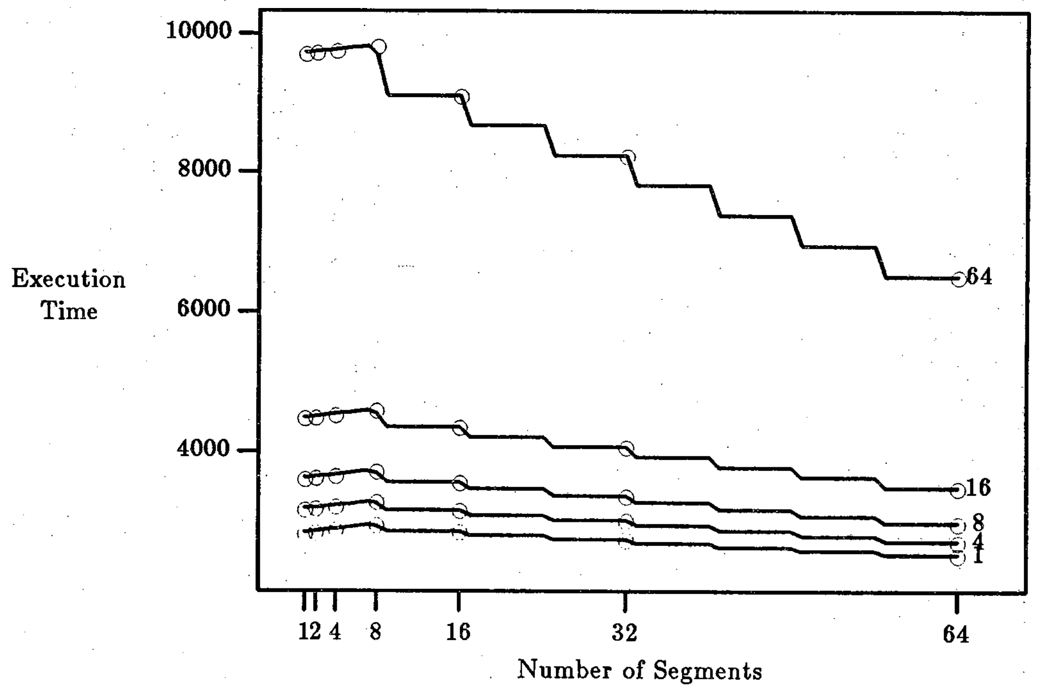


(a) Computation Time Experiment

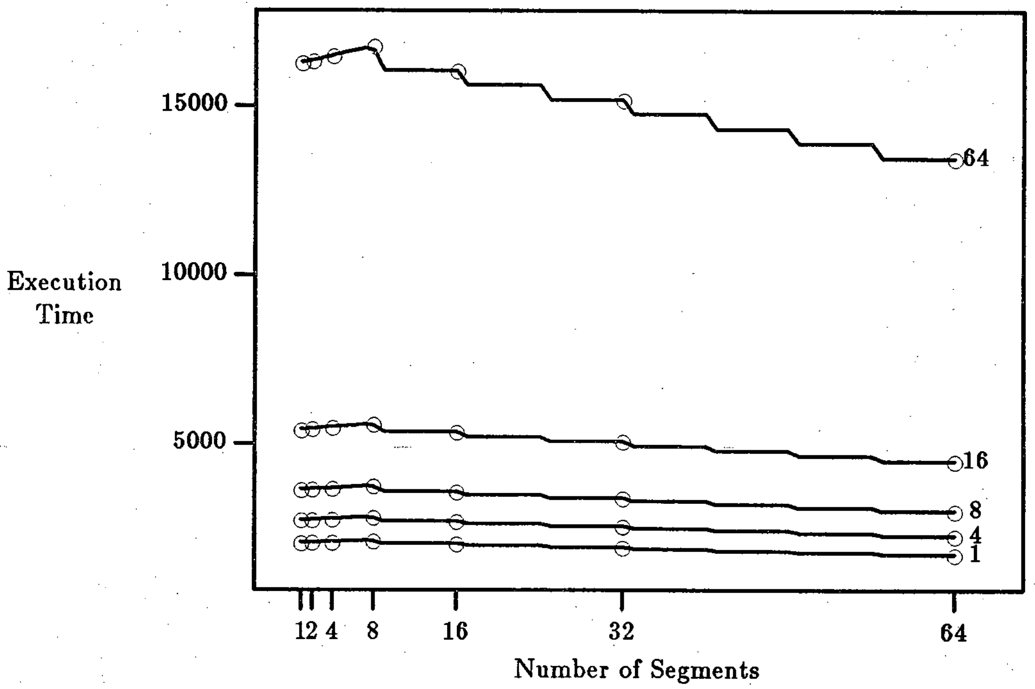


(b) Synchronization Time Experiment

Figure 5.6
DPART Iterative Relaxation Critical Path Execution Time. Circles indicate actual measures; curves show polynomial curve fit; and family of curves represent varied values of experimental time.



(c) Sequencing Time Experiment



(d) Memory Access Time Experiment

Figure 5.6, continued.

Figure 5.1b. The length of a critical path is one greater than the size of the problem, and each computation requires T_c to complete. Therefore, the entire time is that given.

This algorithm's partitioning measure contains three components. The first two indicate that two synchronization operations will enter the critical path when $N < S$. This number increases with the number of segments after it exceeds the problem size. Two initial operations result from the synchronization operations required to start and end each segments. The increasing factor that exists when there are more segments than columns of computations (S) results from added synchronization operations needed between serial segments. This increase produces the staircase nature of Figures 5.3 and 5.4 and results when a single segment is added to a "uniform" number the causing critical path length to increase. The final factor results from an increased dominance of a synchronization operation in parallel with a computation operation.

The sequencing time measures for the two respective ordering schemes are obviously more complex. The Cpart ordering scheme's sequencing measure consists of three parts. The first indicates that an additional computational vertex per "ply" will come into the critical path due to the sequencing function. The next component indicates that there are $S - 1$ sequencing operations in the critical path, one between each stage of the computation, plus those required for the synchronization operations. The third, and most complex, factor indicates that the added sequencing constraints induce some synchronization operations (specifically those between plies) to leave the critical path. There is, however, an upper bound on the operations that may be removed before still other operations appear in the critical path. Finally, this factor includes an adjustment similar to the final factor of the partitioning measure.

The Dpart ordering scheme's sequencing measure consists of four terms. The first factor indicates that indicates that as the number of segments decreases, computational operations enter the critical path. This will continue until S operations are present. The third factor shows the same trend as the first and that there are two sequencing operations associated with each computation. Sequencing operations enter the critical path as the number of segments decreases. The second and fourth factors are similar adjustment factors similar to those found in the CPART sequencing measure.

The final measures are those related to memory access. As each operation (i.e. computation, sequencing, and synchronization) in these experiments was given the same memory access time, T_{ma}, the measures' dependence on only that time, problem size and number of segments is quite logical. Each measure simply reflects the weight of the places previously along the critical path.

Now consider the iterative relaxation experiments. In these experiments, three iterations of the algorithm were run (i.e. $ITER = 3$) which indicates that the critical path (using a wavefront strategy) will be four times the size of the problem, minus 1. As the critical path through a single operation is 3 operations long, the computation measure is $(12\ S - 3)T_c$. The partitioning measure indicates that when $N < S$ three synchronization operations are required for each segment: one between each stage of the wavefront. As with the matrix multiplication algorithm, additional "ply" oriented synchronization operations exist above this level. The final factor is a minor correction for the $N = 1$ case.

The Cpart sequencing measure consists of three components, one for each time variable. The first component indicates that as sequencing constraints are added to the model more of the computational operations fall along the critical path. When fewer than S segments are present, this is a constant factor for any given problem size. Above this number an increase is seen proportional to the number of segments. The second component shows synchronization operations that fall along the critical path, which has similar form to the added computational operations. This component also includes a corresponding correction factor to the one in the partitioning measure. Finally, the sequencing operations added are linear below S segments and increase proportionally above that level.

The Dpart sequencing measure is similar in form to the Cpart measure, except that the weight of the computational and sequencing terms decreases above S segments instead of increasing. These factors are also responsible for the discontinuities that exist at exactly S segments. The final two terms of this expression indicate the removal of synchronization operations is limited, as in the matrix multiplication sequencing measures.

Finally, the memory access measures are of the same form found in the matrix multiplication memory access measures. Once again these measures show that the costs of other operations do not enter as memory access costs are constant for all classes of operations. Each measure simply reflects the weight of the places previously along the critical path.

The following general observations result from the outcomes of the experiments, as depicted in Tables 5.1 and 5.2 and Figures 5.3 through 5.6.

- The relationship between granularity and execution time.

  Figures 5.3 through 5.6 show that granularity has a noticeable effect on the execution time performance of these algorithms in the combined environment. Figure 5.3 demonstrates that, as N increases, the execution time increases. This is a logical outcome for the Cpart scheme, as parallelism is restricted when the segment size drops below the size containing a complete column of the calculation. Figure 5.4, however, shows decreasing execution time with increasing N. Again, this is logical as the Dpart scheme restricts parallelism when there are many calculations in a single segment. Interestingly, that analogous general trends hold in the relaxation algorithm, as illustrated by Figures 5.5 and 5.6. Tables 5.1 and 5.2 confirm these results.

- The effects of changing the relative costs of computation, synchronization, and sequencing.

  Tables 5.1 and 5.2 show the relationships between execution time and T_c, T_{seq}, T_{sync}, and T_{ma} are all linear for a given problem size and number of segments.

- The dominant costs in the performance of these algorithms.

  Figures 5.3 through 5.6 show that memory access time is dominant, followed by the computation and sequencing time. The effect of increasing or decreasing computation and sequencing time cost by a constant factor increases or decreases the execution time by a factor at least three times the effect of changing the synchronization time by the same amount. Tables 5.1 and 5.2 confirm these results as the largest factors are associated with T_{ma} and there are larger factors associated with the T_c and T_{seq} terms than the T_{sync} terms.

- The optimal number of segments.

  In the experiments, the optimal (in the sense of critical path execution time) number of segments varies and is dependent on the relative costs of computation, synchronization, sequencing, and memory access operations.

- Matrix Multiplication, Cpart Ordering Scheme -- Figure 5.3 shows the optimal N is 1 for all cases.
- Matrix Multiplication, Dpart Ordering Scheme -- Figure 5.4b shows that as synchronization time increases the optimal number of segments changes from 64 (S^2) to one.
- Iterative Relaxation, Cpart Ordering Scheme -- Figure 5.5c shows that as sequencing time becomes dominant, the optimal number of segments is 8 (S), while Figure 5.5b shows that when the synchronization time becomes dominant the optimal number is one.
- Iterative Relaxation, Dpart Ordering Scheme -- Figure 5.6b illustrates that as synchronization time becomes dominant, the optimal segment size moves from 64 (S^2) to 1, while Figure 5.6c demonstrates the opposite trend.

• The effect of changing problem size.

  Tables 5.1 and 5.2 show that problem size plays two roles in the performance of these algorithms. The first is the linearly increasing critical path execution time with increasing problems size, which is the critical path performance of these algorithms. The second role is the determination of the "uniform" number of segments as evidenced by the $\left\lceil \frac{N}{S} \right\rceil$ terms throughout these tables.

To summarize the results of these experiments, we have seen that the granularity of partitioning is crucial to the performance of these algorithms in a combined environment. Moreover, the optimal granularity is related to a number of system parameters. These experiments have not only given us a better understanding of combined systems, but have also shown COSMIC to be a useful system to model their performance.

6. Conclusions and Further Work

We have shown COSMIC can be used to study combined systems. We illustrated the use of COSMIC with two algorithms where we showed the impact of partition size on a system's performance. This allowed us to identify the optimal partition size in relation to given system parameters. While these results apply directly only to two iterative algorithms (differing mainly in their interconnectivity), they provided hints as to what factors effect the performance of combined systems. Future work will focus on efforts to generalize these results to other algorithms and include the effect of resource allocation.

7. References

[Bac78] J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM,* Vol. 21, August 1978, pp. 613-641.

[Bro85] J.C. Browne, "Characterization of Parallel Architecture," *1985 Int'l. Conf. on Parallel Processing,* August 1985, pp. 665.

[CaF87] W.W. Carlson and J.A.B. Fortes, "On the Performance of Combined Data Flow and Control Flow Systems: Experiments Using Two Iterative Algorithms," *1987 Int'l. Conf. on Parallel Processing,* August 1987, pp. 671-679.

[Fly66] M.J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE,* Vol. 54, December 1966, pp. 1901-1909.

[GaP85] D.D. Gajski and J-K. Peir, "Essential Issues in Multiprocessor Systems," *IEEE Computer,* Vol. 18, June 1985, pp. 9-27.

[Han77] W. Handler, "The Impact of Classification Schemes on Computer Architecture," *1977 Int'l. Conf. on Parallel Processing,* August 1977, pp. 7-13.

[Joh75] S.C. Johnson, *Yacc: Yet Another Compiler Compiler,* Computer Science Technical Report No. 32, Bell Laboratories, 1975.

[KaM66] R.M. Karp and R.E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM J of App. Math,* Vol. 14, November 1966, pp. 1390-1411.

[KaM69] R.M. Karp and R.E. Miller, "Parallel Program Schemata," *Journal of Computer and System Sciences,* May 1969, pp. 147-195.

[KeR78] B.W. Kernighan and D.M. Ritchie, *The C Programming Language,* Prentice Hall, Englewood Cliffs, NJ, 1978.

[Kle75] L. Kleinrock, *Queueing Systems,* John Wiley and Sons, New York, 1975.

[Les75] M.E. Lesk, *Lex - A Lexical Analyzer Generator,* Computer Science Technical Report No. 39, Bell Laboratories, 1975.

[Mil73] R.E. Miller, "A Comparison of Some Theoretical Models of Parallel Computation," *IEEE Transactions on Computers,* Vol. C-22, August 1973, pp. 710-717.

[Mol82] M.K. Molloy, "Performance Analysis Using Stochastic Petri Nets," *IEEE Transactions on Computers,* September 1982, pp. 913-917.

[Pet66] C.A. Petri, *Communication with Automata,* Supplement to RAD C-TR-65-337, Graffis Air Force Base (translated from Kommunikation mit Automatin, Univ. Bonn, Bonn, Germany, 1962), 1966.

[Pet81] J.L. Peterson, *Petri Net Theory and the Modeling of Systems,* Prentice Hall, Englewood Cliffs, N.J., 1981.

[Ram74] C. Ramchandani, *Analysis of Asynchronous Concurrent Systems by Timed Perti Nets,* MAC-TR-120, Project MAC, MIT, 1974.

Appendix A -- HTP Syntax

This appendix contains the the syntax for the HTP definition language. All upper case words are keywords with the exception of NAME which is any identifier and NUMBER which is any constant number.

```
definition:
    node_defs
    ;

node_defs:
    node_def
    |
    node_defs node_def
    ;

node_def:
    MODEL NAME { sdefs }
    |
    SUBNET NAME { sdefs }
    |
    SUBNET NAME decl_list ;
    |
    NAME = expression ;
    |
    PLACE decl_list ;
    |
    TRANS decl_list ;
    ;

sdefs:  sdef
    |
    sdefs sdef
    ;

sdef:
    SUBNET NAME decl_list ;
    |
    REPEAT ( NAME , expression , expression ) { statements }
    |
    IF ( expression ) { statements }
    |
    IF ( expression ) { statements } ELSE { statements }
    |
    PLACE decl_list ;
    |
    TRANS decl_list ;
    |
    INPUT decl_list ;
    |
    OUTPUT decl_list ;
    |
        NAME = expression ;
        |
        con_list -> con_list ;
        ;


statements:
            statement
        |
        statements statement
        ;

statement:
        REPEAT ( NAME , expression , expression ) { statements }
        |
        IF ( expression ) { statements }
        |
        IF ( expression ) { statements } ELSE { statements }
        |
        NAME = expression ;
        |
        con_list -> con_list ;
        ;



con_list:
        con_element
        |
            con_list , con_element
        ;

con_element:
        NAME
            |
        NAME index_list
        |
        NAME . NAME
        |
        NAME index_list . NAME
        |
        NAME . NAME index_list
        |
        NAME index_list . NAME index_list
        ;

decl_list:
        decl
        |
        decl_list , decl
        ;
decl:
        NAME
        |
        NAME ( expression )
        |
        NAME ( expression , expression )
        |
        NAME index_list
        |
        NAME index_list ( expression )
        |
        NAME index_list ( expression , expression )
        ;

    index_list:
            [ expression ]
            |
            index_list [ expression ]
            ;

    expression:
        NAME
        |
        NUMBER
        |
        ( expression )
        |
        expression + expression
        |
        expression - expression
        |
        expression * expression
        |
        expression / expression
        |
        expression % expression
        |
        expression == expression
        |
        expression != expression
        |
        expression < expression
        |
        expression > expression
        |
        expression <= expression
        |
        expression >= expression
        |
        expression && expression
        |
        expression || expression
        ;
```

Appendix B -- CPART Matrix Multiplication Net

This appendix contains the source code for the CPART matrix multiplication algorithm.

```
/*
 * Matrix-Vector Multiplication.
 *
 * High level control flow, low level data flow.
 *
 */


/*
 * Some constants
 */
SIZE = 8;                           /* the size of the problem */
NPART = 8;                          /* the number of partitions */
ROWMOD = (SIZE*SIZE)/NPART;         /* modulus operation for rows */
COLMOD = SIZE/NPART;                /* modulus operation for columns */
CTIME = 1;                          /* the computatation time */
SYNTIME = 1;                        /* the synchronization time */
SEQTIME = 1;                        /* the sequencing time */
MTIME = 1;                          /* memory access time */

/*
 * model matmult_part is the top level model
 */
model matmult_part {

        input in;                   /* the input to the model */
        place ip(0,1);              /* the initial place, one initial token
        trans it(0),ft(0);          /* initial and final transitions */
        place ip1[SIZE];            /* secondary initial places and trans --
        trans it1[SIZE](0);         /*  -- to reduce initial syncs --one/sub
        place fp[SIZE](0);          /* final places */
        subnet seq fseq[SIZE];  /* the final sequencers */
        subnet innerprod icalc[SIZE][SIZE]; /* inner product ops */
        subnet seq iseq[SIZE][SIZE]; /* inner product seqs */
        subnet sync isync[SIZE][SIZE]; /* inner product syncs */
        trans cfseqt[SIZE](SEQTIME); /* the control flow sequencers */
        subnet mem cfseqp1[SIZE][SIZE]; /* first place */
        subnet mem cfseqp2[SIZE][SIZE]; /* second place */
        subnet start vectstart[SIZE]; /* SIZE vector start locations */
        subnet start arrstart[SIZE][SIZE]; /* SIZE^2 arr "     " */
        subnet sync vsync[SIZE][SIZE]; /* vector synchronizations */
        subnet sync ssync[SIZE]; /* startup synchronizations */
        subnet sync esync[SIZE]; /* ending synchronizations */
        subnet seq aseq[SIZE][SIZE]; /* the sequencer for array inputs *
        subnet seq vseq[SIZE][SIZE]; /* the sequencer for vector inputs

        in -> ip.i;                 /* in->ip->it */
        ip.o -> it.i;
```

```
repeat (i,1,SIZE) {      /* for each vector element and array row */
        it.o -> ip1[i].i; /* it->ip1->it1->... */
        ip1[i].o -> it1[i].i;
        it1[i].o -> vectstart[i].i,vectstart[i].is,icalc[i][1].i;
        if (((NPART >= SIZE) && (i != 1))
            || ((NPART < SIZE) && (NPART > 1)
                && (i != 1) && ((i%COLMOD) == 1))) {
                it.o -> ssync[i].i;
                ssync[i].o -> it1[i].i;
        }
        repeat (j,1,SIZE) { /* for each array element */
                if ((NPART <= SIZE) || (j < (ROWMOD+1))) {
                        it1[i].o -> arrstart[i][j].i,arrstart[i][j].
                } else {
                        cfseqt[(((j-1)/ROWMOD)%SIZE)*(ROWMOD)].o ->
                          arrstart[i][j].i,arrstart[i][j].is;
                }
                vectstart[j].o -> icalc[i][j].iv;
                if (((NPART >= SIZE) && (i != j)) ||
                    ((NPART < SIZE) && ((i%COLMOD) != (j%COLMOD))))
                        vectstart[j].o -> vsync[i][j].i;
                        vsync[i][j].o -> icalc[i][j].isyn;
                } else {
                        vectstart[j].o -> vseq[i][j].i;
                        vseq[i][j].o -> icalc[i][j].isyn;
                }
                arrstart[i][j].o -> icalc[i][j].ia;
                arrstart[i][j].o -> aseq[i][j].i;
                aseq[i][j].o -> icalc[i][j].isyn;
        }
        repeat (j,1,SIZE-1) {
                icalc[i][j].o -> icalc[i][j+1].i;
                if ((NPART>SIZE) && ((j%ROWMOD) == 0)) {
                        icalc[i][j].o -> isync[i][j].i;
                        isync[i][j].o -> icalc[i][j+1].o;
                        /*
                         * here we sequence using control flow...
                         */
                        icalc[i][j].o -> cfseqp1[i][j].i;
                        cfseqp1[i][j].o -> cfseqt[j].i;
                        cfseqt[j].o -> cfseqp2[i][j].i;
                        cfseqp2[i][j].o -> icalc[i][j+1].it;
                } else {
                        icalc[i][j].o -> iseq[i][j].i;
                        iseq[i][j].o -> icalc[i][j+1].it;
                }
        }
        icalc[i][SIZE].o -> fp[i].i,fseq[i].i;
        fseq[i].o,fp[i].o -> ft.i;
        if (((NPART >= SIZE) && (i != 1))
            || ((NPART < SIZE) && (NPART > 1)
                && (i != 1) && ((i%COLMOD) == 1))) {
                icalc[i][SIZE].o -> esync[i].i;
```

```
                                esync[i].o -> ft.i;
                        }
                }
        }

        /*
         * subnet start is simply a holder of an initial value
         */
        subnet start {
                input i,is;
                output o,os;

                place p(0);                     /* zero memory time */
                trans t(0);                     /* zero execute time */
                subnet seq seq;

                i -> p.i;                       /* i->p->t->o */
                is -> seq.i;
                seq.o,p.o -> t.i;
                t.o -> o;
        }

        /*
         * subnet inner prod does a multiply and add operation
         */
        subnet innerprod
        {
                input i,it;                     /* the input to accumulate to */
                input iv;                       /* one multiply input */
                input ia;                       /* the other multiply input */
                input isyn;                     /* the synchronization input */
                subnet mem p[3];                /* three places, to hold 3 inputs */
                subnet mem pint;                /* an internal place */
                trans t[2](CTIME);              /* the computations, takes CTIME to comp
                subnet seq intseq;              /* the internal sequencer */
                output o;                       /* the  output */

                iv->p[1].i;                     /* each input goes to one place */
                ia->p[2].i;
                i->p[3].i;

                isyn,p[1].o,p[2].o -> t[1].i;
                t[1].o ->pint.i,intseq.i;
            it, intseq.o, pint.o ,p[3].o -> t[2].i;
                t[2].o -> o;                    /* trans goes to output */
        }

        /*
         * subnet sync performs the place,trans,place synchronization
         */
        subnet sync {
                input i;                        /* input and outputs */
                output o;
        subnet mem p1,p2;          /* two places */
        trans t(SYNTIME);          /* one trans, STIME is sync time */

        i -> p1.i;                 /* i->p1->t->p2->o */
    p1.o -> t.i;
        t.o -> p2.i;
        p2.o -> o;
}

/*
 * subnet seq performs the place,trans,place sequencing
 */
subnet seq {
        input i;                   /* input and outputs */
        output o;
        subnet mem p1,p2;          /* two places */
        trans t(SEQTIME);          /* one trans, STIME is sync time */

        i -> p1.i;                 /* i->p1->t->p2->o */
        p1.o -> t.i;
        t.o -> p2.i;
        p2.o -> o;
}


/*
 * subnet mem performs the place,trans,place memory accessing
 */
subnet mem {
        input i;                   /* input and outputs */
        output o;
        place p1,p2;               /* two places */
        trans t(MTIME);            /* one trans, STIME is sync time */

        i -> p1.i;                 /* i->p1->t->p2->o */
        p1.o -> t.i;
        t.o -> p2.i;
        p2.o -> o;
}
```

Appendix C -- DPART Matrix Multiplication Net

This appendix contains the source code for the DPART ordering scheme, matrix multiplication algorithm.

```
/*
 * Matrix-Vector Multiplication.
 *
 * Sequencing Net, High level data flow, low level control flow.
 *
 */


/*
 * Some constants
 */
SIZE = 8;                       /* the size of the problem */
NPART = 16;                     /* the number of partitions */
ROWMOD = (SIZE*SIZE)/NPART;     /* modulus operation for rows */
COLMOD = SIZE/NPART;            /* modulus operation for columns */
CTIME = 1;                      /* the computatation time */
SYNTIME = 1;                    /* the synchronization time */
SEQTIME = 1;                    /* the sequencing time */
MTIME = 1;                      /* memory access time */

/*
 * model matmult_part is the top level model
 */
model matmult_part {

        input in;               /* the input to the model */
        place ip(0,1);          /* the initial place, one initial token *
        trans it(0),ft(0);      /* initial and final transitions */
        place ip1[SIZE];        /* secondary initial places and trans --
        trans it1[SIZE](0);     /*  -- to reduce initial syncs --one/subn
        place fp[SIZE](0);      /* final places */
        subnet seq fseq[SIZE];  /* the final sequencers */
        subnet innerprod icalc[SIZE][SIZE]; /* inner product ops */
        subnet sync isync[SIZE][SIZE]; /* inner product syncs */
        subnet seq iseq[SIZE][SIZE]; /* inner product seqs */
        subnet start vectstart[SIZE]; /* SIZE vector start locations */
        subnet start arrstart[SIZE][SIZE]; /* SIZE^2 arr "     " */
        subnet sync vsync[SIZE][SIZE]; /* vector synchronizations */
        subnet sync ssync[SIZE]; /* startup synchronizations */
        subnet sync esync[SIZE]; /* ending synchronizations */
        subnet seq aseq[SIZE][SIZE]; /* the sequencer for array inputs */
        subnet seq vseq[SIZE][SIZE]; /* the sequencer for vector inputs *

        in -> ip.i;             /* in->ip->it */
        ip.o -> it.i;

        repeat (i,1,SIZE) {      /* for each vector element and array row
                it.o -> ip1[i].i; /* it->ip1->it1->... */
                ip1[i].o -> it1[i].i;
                it1[i].o -> vectstart[i].i,icalc[i][1].i;
                if (((NPART >= SIZE) && (i != 1))
                    || ((NPART < SIZE) && (NPART > 1)
                        && (i != 1) && ((i%COLMOD) == 1))) {
                        it.o -> ssync[i].i;
                        ssync[i].o -> it1[i].i;
                }
                repeat (j,1,SIZE) { /* for each array element */
                        it1[i].o -> arrstart[i][j].i;
                        vectstart[j].o -> icalc[i][j].iv;
                        if (((NPART >= SIZE) && (i != j)) ||
                            ((NPART < SIZE) && ((i%COLMOD) != (j%COLMOD))))
                                vectstart[j].o -> vsync[i][j].i;
                                vsync[i][j].o -> icalc[i][j].isyn;
                        } else {
                                vectstart[j].o -> vseq[i][j].i;
                                vseq[i][j].o -> icalc[i][j].isyn;
                        }
                        arrstart[i][j].o -> icalc[i][j].ia;
                        arrstart[i][j].o -> aseq[i][j].i;
                        aseq[i][j].o -> icalc[i][j].isyn;
                }
                repeat (j,1,SIZE-1) {
                        icalc[i][j].o -> icalc[i][j+1].i;
                        if ((NPART>SIZE) && ((j%ROWMOD) == 0)) {
                                /*
                                 * here we sequence using data flow...
                                 *
                                 * no need to sequence, already synced
                                 */
                                icalc[i][j].o -> isync[i][j].i;
                                isync[i][j].o -> icalc[i][j+1].it;
                        } else {
                                /*
                                 * control flow sequence between steps
                                 */
                                icalc[i][j].o -> iseq[i][j].i;
                                iseq[i][j].o -> icalc[i][j+1].isyn;
                        }
                }
                icalc[i][SIZE].o -> fp[i].i,fseq[i].i;
                fseq[i].o,fp[i].o -> ft.i;
                if (((NPART >= SIZE) && (i != 1))
                    || ((NPART < SIZE) && (NPART > 1)
                        && (i != 1) && ((i%COLMOD) == 1))) {
                        icalc[i][SIZE].o -> esync[i].i;
                        esync[i].o -> ft.i;
                }
        }
}

/*
 * subnet start is simply a holder of an initial value
 */
subnet start {
        input i;
        output o;

        place p(0);                 /* zero memory time */
        trans t(0);                 /* zero execute time */

        i -> p.i;                   /* i->p->t->o */
        p.o -> t.i;
        t.o -> o;
}

/*
 * subnet inner prod does a multiply and add operation
 */
subnet innerprod
{
        input i,it;                 /* the input to accumulate to */
        input iv;                   /* one multiply input */
        input ia;                   /* the other multiply input */
        input isyn;                 /* the synchronization input */
        subnet mem p[3];            /* three places, to hold 3 inputs */
        subnet mem pint;            /* an internal place */
        trans t[2](CTIME);          /* the computations, takes CTIME to comp
        subnet seq intseq;          /* the internal sequencer */
        output o;                   /* the  output */

        iv->p[1].i;                 /* each input goes to one place */
        ia->p[2].i;
        i->p[3].i;

        isyn,p[1].o,p[2].o -> t[1].i;
        t[1].o ->pint.i,intseq.i;
        it, intseq.o, pint.o ,p[3].o -> t[2].i;
        t[2].o -> o;      /* trans goes to output */
}

/*
 * subnet sync performs the place,trans,place synchronization
 */
subnet sync {
        input i;                    /* input and outputs */
        output o;
        subnet mem p1,p2;           /* two places */
        trans t(SYNTIME);           /* one trans, STIME is sync time */

        i -> p1.i;                  /* i->p1->t->p2->o */
        p1.o -> t.i;
        t.o -> p2.i;
        p2.o -> o;
}
```

```
/*
 * subnet seq performs the place,trans,place sequencing
 */
subnet seq {
        input i;                /* input and outputs */
        output o;
        subnet mem p1,p2;       /* two places */
        trans t(SEQTIME);       /* one trans, STIME is sync time */

        i -> p1.i;              /* i->p1->t->p2->o */
        p1.o -> t.i;
        t.o -> p2.i;
        p2.o -> o;
}


/*
 * subnet mem performs the place,trans,place memory accessing
 */
subnet mem {
        input i;                /* input and outputs */
        output o;
        place p1,p2;            /* two places */
        trans t(MTIME);         /* one trans, STIME is sync time */

        i -> p1.i;              /* i->p1->t->p2->o */
        p1.o -> t.i;
        t.o -> p2.i;
        p2.o -> o;
}
```

Appendix D -- CPART Iterative Relaxation Net

This appendix contains the source code for the CPART ordering scheme, iterative relaxation algorithm

```
/*
 * 4-point relaxation problem:
 *
 *    do r = 1, ITER
 *       do i = 1, SIZE
 *          do j = 1, SIZE
 *             a(i,j) := (a(i-1,j) + a(i,j-1) + a(i+1,j) + a(i, j+1)) / 4
 *          end_do
 *       end_do
 *    end_do
 */

/*
 * Sequencing Net. High Level Control Flow (Sequenced by "ply")
 *                  Low Level Data Flow (Sequenced by data)
 *
 *   Partitioned into NPART partitions.
 *
 */

SIZE = 8;                       /* the size of the problem */
ITER = 3;                       /* the number of relaxations to perform *
NPART = 4;                      /* the number of partitions */

ROWMOD = (SIZE*SIZE)/NPART;     /* modulus operation for rows */
COLMOD = SIZE/NPART;            /* modulus operation for columns */

CTIME = 1;                      /* time to do an add or division */
SYNTIME = 1;                    /* the synchronization time */
SEQTIME = 1;                    /* the sequencing time */
MTIME = 1;                      /* memory access time */

model relax
{
    input in;

    subnet calc calc[ITER][SIZE][SIZE];
    subnet sync sync1[ITER][SIZE][SIZE];
    subnet sync sync2[ITER][SIZE][SIZE];
    subnet sync sync3[ITER][SIZE][SIZE];
    subnet sync sync4[ITER][SIZE][SIZE];
    subnet seq seq1[ITER][SIZE][SIZE];
    subnet seq seq2[ITER][SIZE][SIZE];
    subnet seq seq3[ITER][SIZE][SIZE];
    subnet seq seq4[ITER][SIZE][SIZE];
    trans cfseqt[ITER][SIZE](SEQTIME);
    subnet mem cfseqp1[ITER][SIZE][SIZE];
    subnet mem cfseqp2[ITER][SIZE][SIZE];
```

```
place ip(0,1);
trans it(0);
place fp[SIZE][SIZE];
trans ft(0);

in  -> ip.i;
ip.i -> it.i;

repeat (i, 1, SIZE) {
    repeat (j, 1, SIZE) {
        if (i != SIZE) {
            it.o -> calc[1][i][j].i[1];
            if ((NPART <= SIZE) || ((i%ROWMOD) != 1)) {
                it.o -> seq1[1][i][j].i[1];
                seq1[1][i][j].o -> calc[1][i][j].it[1];
            }
        }
        if (j != SIZE) {
            it.o ->calc[1][i][j].i[2];
            if ((NPART == 1) ||
                ((NPART < SIZE) && ((j%COLMOD)!=1))) {
                it.o -> seq2[1][i][j].i[1];
                seq2[1][i][j].o -> calc[1][i][j].it[1];
            }
        }
        if (i == 1) {
            it.o -> calc[1][i][j].i[3];
            if ((NPART <= SIZE) || ((i%ROWMOD) != 0)) {
                it.o -> seq3[1][i][j].i;
                seq3[1][i][j].o -> calc[1][i][j].it[2];
            }
        }
        if (j == 1) {
            it.o -> calc[1][i][j].i[4];
            if ((NPART == 1) ||
                ((NPART <= SIZE) && ((j%COLMOD)!=0))) {
                it.o -> seq4[1][i][j].i;
                seq4[1][i][j].o -> calc[1][i][j].it[2];
            }
        }
    }
}

repeat (r,1,ITER) {
    repeat (i,1,SIZE) {
        repeat (j,1,SIZE) {
            if (r < ITER) {
                /*
                 * Connection 1 connects to previous row, same column
                 */
                if (i > 1) {
                    calc[r][i][j].o -> calc[r+1][i-1][j].i[1];
            /*
             * On diff partition only if more part than colum
             * and when mod is not right.
             */
            if ((NPART > SIZE) && ((i%ROWMOD) == 1)) {
                calc[r][i][j].o -> sync1[r+1][i-1][j].i;
                sync1[r+1][i-1][j].o -> calc[r+1][i-1][j].it[
            } else {
                calc[r][i][j].o -> seq1[r+1][i-1][j].i;
                seq1[r+1][i-1][j].o -> calc[r+1][i-1][j].it[1
            }
        } else {
            calc[r][i][j].o -> calc[r][SIZE][j].i[1];
            /*
             * On diff partition only if more part than colum
             */
            if (NPART > SIZE) {
                calc[r][i][j].o -> sync1[r][SIZE][j].i;
                sync1[r][SIZE][j].o -> calc[r][SIZE][j].it[1]
            } else {
                calc[r][i][j].o -> seq1[r][SIZE][j].i;
                seq1[r][SIZE][j].o -> calc[r][SIZE][j].it[1];
            }
        }

        /*
         * Connection 2 connects to same row, previous colum
         */
        if (j > 1) {
            calc[r][i][j].o -> calc[r+1][i][j-1].i[2];
            /*
             * On diff. partition when more partitions than
             * columns or when mod eqn. is satisfied
             */
            if ((NPART >= SIZE) ||
                ((NPART > 1) && ((j%COLMOD)==1))) {
                calc[r][i][j].o -> sync2[r+1][i][j-1].i;
                sync2[r+1][i][j-1].o -> calc[r+1][i][j-1].it
            } else {
                calc[r][i][j].o -> seq2[r+1][i][j-1].i;
                seq2[r+1][i][j-1].o -> calc[r+1][i][j-1].it[
            }
        } else {
            calc[r][i][j].o -> calc[r][i][SIZE].i[2];
            /*
             * On diff. partition whenever more than 1 parti
             */
            if (NPART > 1) {
                calc[r][i][j].o -> sync2[r][i][SIZE].i;
                sync2[r][i][SIZE].o -> calc[r][i][SIZE].it[1
            } else {
                calc[r][i][j].o -> seq2[r][i][SIZE].i;
                seq2[r][i][SIZE].o -> calc[r][i][SIZE].it[1]
                }
            }
        } else {
            if (i == 1) {
                calc[r][i][j].o -> calc[r][SIZE][j].i[1];
                /*
                 * On diff partition only if more part than columns.
                 */
                if (NPART > SIZE) {
                    calc[r][i][j].o -> sync1[r][SIZE][j].i;
                    sync1[r][SIZE][j].o -> calc[r][SIZE][j].it[1];
                } else {
                    calc[r][i][j].o -> seq1[r][SIZE][j].i;
                    seq1[r][SIZE][j].o -> calc[r][SIZE][j].it[1];
                }
            }
            if (j == 1) {
                calc[r][i][j].o -> calc[r][i][SIZE].i[2];
                /*
                 * On diff. partition whenever more than 1 partition
                 */
                if (NPART > 1) {
                    calc[r][i][j].o -> sync2[r][i][SIZE].i;
                    sync2[r][i][SIZE].o -> calc[r][i][SIZE].it[1];
                } else {
                    calc[r][i][j].o -> seq2[r][i][SIZE].i;
                    seq2[r][i][SIZE].o -> calc[r][i][SIZE].it[1];
                }
            }
            calc[r][i][j].o -> fp[i][j].i;
            fp[i][j].o->ft.i;
        }

        /*
         * Connection 3 connects next row, same column
         *
         *    This is where partition sequencing is done
         */
        if (i < SIZE) {
            calc[r][i][j].o -> calc[r][i+1][j].i[3];
            /*
             * On diff partition only when more partition than
             * columns and mod eqn is satisfied.
             */
            if ((NPART > SIZE) && ((i%ROWMOD) == 0)) {
                calc[r][i][j].o -> sync3[r][i+1][j].i;
                sync3[r][i+1][j].o -> calc[r][i+1][j].it[2];
                /*
                 * Here we sequence using control flow...
                 */
                calc[r][i][j].o -> cfseqp1[r][i][j].i;
                cfseqp1[r][i][j].o -> cfseqt[r][i].i;
                cfseqt[r][i].o -> cfseqp2[r][i][j].i;
            cfseqp2[r][i][j].o -> calc[r][i+1][j].it[2];
        } else {
            calc[r][i][j].o -> seq3[r][i+1][j].i;
            seq3[r][i+1][j].o -> calc[r][i+1][j].it[2];
        }
    } else {
        if (r < ITER) {
            calc[r][i][j].o -> calc[r+1][1][j].i[3];
            if (NPART > SIZE) {
                calc[r][i][j].o -> sync3[r+1][1][j].i;
                sync3[r+1][1][j].o -> calc[r+1][1][j].it[2];
            }
            /*
             * control flow sequencing between iterations
             */
            calc[r][i][j].o -> cfseqp1[r][i][j].i;
            cfseqp1[r][i][j].o -> cfseqt[r][i].i;
            cfseqt[r][i].o -> cfseqp2[r][i][j].i;
            cfseqp2[r][i][j].o -> calc[r+1][1][j].it[2];
        }
    }

    /*
     * connection 4 connects same row, next column
     */
    if (j < SIZE) {
        calc[r][i][j].o -> calc[r][i][j+1].i[4];
        /*
         * Here we are in different partitions if there is
         * more than 1 partition and either there are more
         * partitions than columns and the mod eqn. is satis
         */
        if ((NPART > 1) &&
            ((NPART > SIZE) || ((j%COLMOD)==0))) {
            calc[r][i][j].o -> sync4[r][i][j+1].i;
            sync4[r][i][j+1].o -> calc[r][i][j+1].it[2];
        } else {
            calc[r][i][j].o -> seq4[r][i][j+1].i;
            seq4[r][i][j+1].o -> calc[r][i][j+1].it[2];
        }
    } else {
        if (r < ITER) {
            calc[r][i][j].o -> calc[r+1][i][1].i[4];
            /*
             * Here we go to different partitions only
             * if is more than 1 partition
             */
            if (NPART > 1) {
                calc[r][i][j].o -> sync4[r+1][i][1].i;
                sync4[r+1][i][1].o -> calc[r+1][i][1].it[2];
            } else {
                calc[r][i][j].o -> seq4[r+1][i][1].i;
                seq4[r+1][i][1].o -> calc[r+1][i][1].it[2];
```

```
                                }
                            }
                        }
                    }
                }
            }
        }


subnet calc
{
        input i[4];
        input it[2];
        output o;

        subnet mem pin[4];        /* places to receive inputs */
        subnet mem padd[3];       /* internal places */
        trans tadd[3](CTIME);     /* three addition operation */
        trans tdiv(CTIME);        /* the division */
        subnet seq intseq[3];     /* internal sequencing */

        i[1] -> pin[1].i;
        i[2] -> pin[2].i;
        it[1], pin[1].o, pin[2].o -> tadd[1].i;
        tadd[1].o -> padd[1].i, intseq[1].i;

        i[3] -> pin[3].i;
        i[4] -> pin[4].i;
        it[2], pin[3].o, pin[4].o -> tadd[2].i;
        tadd[2].o -> padd[2].i, intseq[2].i;

        intseq[1].o, intseq[2].o, padd[1].o, padd[2].o -> tadd[3].i;
        tadd[3].o -> padd[3].i, intseq[3].i;
        intseq[3].o, padd[3].o -> tdiv.i;
        tdiv.o -> o;
}

/*
 * subnet sync performs the place,trans,place synchronization
 */
subnet sync {
        input i;                  /* input and outputs */
        output o;
        subnet mem p1,p2;         /* two places */
        trans t(SYNTIME);         /* one trans, SYNTIME is sync time */

        i -> p1.i;                /* i->p1->t->p2->o */
        p1.o -> t.i;
        t.o -> p2.i;
        p2.o -> o;
}
```

```
/*
 * subnet seq performs the place,trans,place sequencing
 */
subnet seq {
        input i;                /* input and outputs */
        output o;
        subnet mem p1,p2;       /* two places */
        trans t(SEQTIME);       /* one trans, SEQTIME is seq time */

        i -> p1.i;              /* i->p1->t->p2->o */
        p1.o -> t.i;
        t.o -> p2.i;
        p2.o -> o;
}


/*
 * subnet mem performs the place,trans,place memory accessing
 */
subnet mem {
        input i;                /* input and outputs */
        output o;
        place p1,p2;            /* two places */
        trans t(MTIME);         /* one trans, STIME is sync time */

        i -> p1.i;              /* i->p1->t->p2->o */
        p1.o -> t.i;
        t.o -> p2.i;
        p2.o -> o;
}
```

Appendix E -- DPART Iterative Relaxation Net

This appendix contains the source code for the DPART ordering scheme, iterative relaxation algorithm

```
   /*
    * 4-point relaxation problem:
    *
    *     do r = 1, ITER
    *        do i = 1, SIZE
    *         do j = 1, SIZE
    *             a(i,j) := (a(i-1,j) + a(i,j-1) + a(i+1,j) + a(i, j+1)) / 4
    *           end_do
    *        end_do
    *     end_do
    */

   /*
    * Sequencing Net.
    *
    *   Partitioned into NPART partitions.
    *
    */

SIZE = 10;                              /* the size of the problem */
ITER = 3;                               /* the number of relaxations to perform */
NPART = 5;                              /* the number of partitions */

ROWMOD = (SIZE*SIZE)/NPART;             /* modulus operation for rows */
COLMOD = SIZE/NPART;                    /* modulus operation for columns */

CTIME = 1;                              /* time to do an add or division */
SYNTIME = 1;                            /* the synchronization time */
SEQTIME = 1;                            /* the sequencing time */
MTIME = 1;                              /* memory access time */

model relax
{
    input in;

    subnet calc calc[ITER][SIZE][SIZE];
    subnet sync sync1[ITER][SIZE][SIZE];
    subnet sync sync2[ITER][SIZE][SIZE];
    subnet sync sync3[ITER][SIZE][SIZE];
    subnet sync sync4[ITER][SIZE][SIZE];
    subnet seq seq[ITER][SIZE][SIZE];

    place ip(0,1);
    trans it(0);
    place fp[SIZE][SIZE];
    trans ft(0);

    in  -> ip.i;
```

```
ip.i -> it.i;

repeat (i, 1, SIZE) {
    repeat (j, 1, SIZE) {
        if (i != SIZE) {
            it.o -> calc[1][i][j].i[1];
        }
        if (j != SIZE) {
            it.o ->calc[1][i][j].i[2];
        }
        if (i == 1) {
            it.o -> calc[1][i][j].i[3];
            it.o -> seq[1][i][j].i;
            seq[1][i][j].o -> calc[1][i][j].it[2];
        }
        if (j == 1) {
            it.o -> calc[1][i][j].i[4];
        }
    }
}

repeat (r,1,ITER) {
    repeat (i,1,SIZE) {
        repeat (j,1,SIZE) {
            if (r < ITER) {
                /*
                 * Connection 1 connects to previous row, same columr
                 */
                if (i > 1) {
                    calc[r][i][j].o -> calc[r+1][i-1][j].i[1];
                    /*
                     * On diff partition only if more part than colur
                     * and when mod is not right.
                     */
                    if ((NPART > SIZE) && ((i%ROWMOD) == 1)) {
                        calc[r][i][j].o -> sync1[r+1][i-1][j].i;
                        sync1[r+1][i-1][j].o -> calc[r+1][i-1][j].it
                    }
                } else {
                    calc[r][i][j].o -> calc[r][SIZE][j].i[1];
                    /*
                     * On diff partition only if more part than colum
                     */
                    if (NPART > SIZE) {
                        calc[r][i][j].o -> sync1[r][SIZE][j].i;
                        sync1[r][SIZE][j].o -> calc[r][SIZE][j].it[1
                    }
                }

                /*
                 * Connection 2 connects to same row, previous colum
                 */
                if (j > 1) {
```

```
                calc[r][i][j].o -> calc[r+1][i][j-1].i[2];
                /*
                 * Syncronize if more partitions than size or
                 * mod equation works.
                 */
                if ((NPART >= SIZE) ||
                    ((NPART > 1) && ((j%COLMOD)==1))) {
                    calc[r][i][j].o -> sync2[r+1][i][j-1].i;
                    sync2[r+1][i][j-1].o -> calc[r+1][i][j-1].it[1];
                }
            } else {
                calc[r][i][j].o -> calc[r][i][SIZE].i[2];
                /*
                 * On diff partition whenever more than one partitio
                 */
                if (NPART > 1) {
                    calc[r][i][j].o -> sync2[r][i][SIZE].i;
                    sync2[r][i][SIZE].o -> calc[r][i][SIZE].it[1];
                }
            }
        } else {
            if (i == 1) {
                calc[r][i][j].o -> calc[r][SIZE][j].i[1];
                /*
                 * On diff partition only if more part than columns.
                 */
                if (NPART > SIZE) {
                    calc[r][i][j].o -> sync1[r][SIZE][j].i;
                    sync1[r][SIZE][j].o -> calc[r][SIZE][j].it[1];
                }
            }
            if (j == 1) {
                calc[r][i][j].o -> calc[r][i][SIZE].i[2];
                /*
                 * On diff partition whenever more than one partitio
                 */
                if (NPART > 1) {
                    calc[r][i][j].o -> sync2[r][i][SIZE].i;
                    sync2[r][i][SIZE].o -> calc[r][i][SIZE].it[1];
                }
            }
            calc[r][i][j].o -> fp[i][j].i;
            fp[i][j].o->ft.i;
        }

        /*
         * Connection 3 connects next row, same column
         *
         *    This is where partition sequencing is done
         */
        if (i < SIZE) {
            calc[r][i][j].o -> calc[r][i+1][j].i[3];
            /*
         * On diff partition only when more partition than
         * columns and mod eqn is satisfied.
         */
        if ((NPART > SIZE) && ((i%ROWMOD) == 0)) {
            /*
             * Here we sequence using data flow between
             * partitions.
             *
             * already synched, no need to seq
             */
            calc[r][i][j].o -> sync3[r][i+1][j].i;
            sync3[r][i+1][j].o -> calc[r][i+1][j].it[2];
        } else {
            /*
             * Control Flow Sequencing Between steps
             */
            calc[r][i][j].o -> seq[r][i+1][j].i;
            seq[r][i+1][j].o -> calc[r][i+1][j].it[1];
        }
    } else {
        if (r < ITER) {
            calc[r][i][j].o -> calc[r+1][1][j].i[3];
            /*
             * Data flow sequencing between iterations
             */
            /*
             * no need to seq if already synched
             */
            if (NPART > SIZE) {
                    calc[r][i][j].o -> sync3[r+1][1][j].i;
                    sync3[r+1][1][j].o -> calc[r+1][1][j].it[
            } else {
                    calc[r][i][j].o -> seq[r+1][1][j].i;
                    seq[r+1][1][j].o -> calc[r+1][1][j].it[1]
            }
        }
    }

    /*
     * connection 4 connects same row, next column
     */
    if (j < SIZE) {
        calc[r][i][j].o -> calc[r][i][j+1].i[4];
        /*
         * Here we are in different partitions if there is
         * more than 1 partition and either there are more
         * partitions than columns and the mod eqn. is satisf
         */
        if ((NPART > 1) &&
            ((NPART > SIZE) || ((j%COLMOD)==0))) {
            calc[r][i][j].o -> sync4[r][i][j+1].i;
            sync4[r][i][j+1].o -> calc[r][i][j+1].it[2];
        }
```

```
                } else {
                    if (r < ITER) {
                        calc[r][i][j].o -> calc[r+1][i][1].i[4];
                        /*
                         * Here we go to different partitions only
                         * if is more than 1 partition
                         */
                        if (NPART > 1) {
                            calc[r][i][j].o -> sync4[r+1][i][1].i;
                            sync4[r+1][i][1].o -> calc[r+1][i][1].it[2];
                        }
                    }
                }
            }
        }
    }
}


subnet calc
{
        input i[4];
        input it[2];
        output o;

        subnet mem pin[4];        /* places to receive inputs */
        subnet mem padd[3];       /* internal places */
        trans tadd[3](CTIME);     /* three addition operation */
        trans tdiv(CTIME);        /* the division */
        subnet seq intseq[3];     /* internal sequencing */

        i[1] -> pin[1].i;
        i[2] -> pin[2].i;
        it[1], pin[1].o, pin[2].o -> tadd[1].i;
        tadd[1].o -> padd[1].i, intseq[1].i;

        i[3] -> pin[3].i;
        i[4] -> pin[4].i;
        intseq[1].o,it[2], pin[3].o, pin[4].o -> tadd[2].i;
        tadd[2].o -> padd[2].i, intseq[2].i;

        intseq[2].o, padd[1].o, padd[2].o -> tadd[3].i;
        tadd[3].o -> padd[3].i, intseq[3].i;
        intseq[3].o, padd[3].o -> tdiv.i;
        tdiv.o -> o;
}

/*
 * subnet sync performs the place,trans,place synchronization
 */
subnet sync {
          input i;                /* input and outputs */
```

```
         output o;
         subnet mem p1,p2;        /* two places */
         trans t(SYNTIME);        /* one trans, SYNTIME is sync time */

         i -> p1.i;               /* i->p1->t->p2->o */
         p1.o -> t.i;
         t.o -> p2.i;
         p2.o -> o;
}

/*
 * subnet seq performs the place,trans,place sequencing
 */
subnet seq {
         input i;                 /* input and outputs */
         output o;
         subnet mem p1,p2;        /* two places */
         trans t(SEQTIME);        /* one trans, SEQTIME is seq time */

         i -> p1.i;               /* i->p1->t->p2->o */
         p1.o -> t.i;
         t.o -> p2.i;
         p2.o -> o;
}


/*
 * subnet mem performs the place,trans,place memory accessing
 */
subnet mem {
        input i;                  /* input and outputs */
        output o;
        place p1,p2;              /* two places */
        trans t(MTIME);           /* one trans, STIME is sync time */

        i -> p1.i;                /* i->p1->t->p2->o */
        p1.o -> t.i;
        t.o -> p2.i;
        p2.o -> o;
}
```