

Old Dominion University

## ODU Digital Commons

---

Electrical & Computer Engineering Theses & Dissertations

Electrical & Computer Engineering

---

Fall 1993

# Cyclo-Static Scheduling of Large Grain Dataflow Algorithms on a Local Area ATAMM Multicomputing Testbed

Sudepto Roy  
*Old Dominion University*

Follow this and additional works at: [https://digitalcommons.odu.edu/ece\\_etds](https://digitalcommons.odu.edu/ece_etds)



Part of the [Computational Engineering Commons](#), [Computer Engineering Commons](#), and the [Theory and Algorithms Commons](#)

---

### Recommended Citation

Roy, Sudepto. "Cyclo-Static Scheduling of Large Grain Dataflow Algorithms on a Local Area ATAMM Multicomputing Testbed" (1993). Master of Science (MS), Thesis, Electrical & Computer Engineering, Old Dominion University, DOI: 10.25777/tzke-ct37  
[https://digitalcommons.odu.edu/ece\\_etds/499](https://digitalcommons.odu.edu/ece_etds/499)

This Thesis is brought to you for free and open access by the Electrical & Computer Engineering at ODU Digital Commons. It has been accepted for inclusion in Electrical & Computer Engineering Theses & Dissertations by an authorized administrator of ODU Digital Commons. For more information, please contact [digitalcommons@odu.edu](mailto:digitalcommons@odu.edu).

**CYCLO-STATIC SCHEDULING OF  
LARGE GRAIN DATAFLOW ALGORITHMS ON A  
LOCAL AREA ATAMM MULTICOMPUTING TESTBED**

by

Sudepto Roy

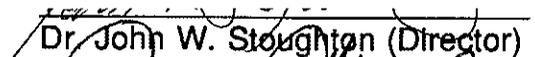
B.E.(Cp.E.), August 1992, The Maharaja Sayajirao University of Baroda, India

A Thesis Submitted to the Faculty of  
Old Dominion University in the Partial Fulfillment  
of the Requirements of the Degree of

**MASTER OF SCIENCE  
COMPUTER ENGINEERING**

**OLD DOMINION UNIVERSITY  
December, 1993**

Approved by:

  
Dr. John W. Stoughton (Director)

\_\_\_\_\_  
Dr. Roland R. Mielke

  
Dr. James F. Leathrum

## **ABSTRACT**

### **CYCLO-STATIC SCHEDULING OF LARGE GRAIN DATAFLOW ALGORITHMS ON A LOCAL AREA ATAMM MULTICOMPUTING TESTBED**

Sudepto Roy  
Old Dominion University, 1993  
Director : Dr. John W. Stoughton

A strategy for cyclo-statically scheduling deterministic large grain dataflow (LGDF) algorithms for distributed execution on loosely coupled multicomputer architectures is presented in this research. The computational paradigm used is the ODU/NASA developed Algorithm To Architecture Mapping Model (ATAMM), which consists of marked graphs and Gantt chart representations that model the iterative execution of deterministic LGDF algorithms for different values of throughput and computation time. It is postulated that the behavior of these algorithms could be represented by the aggregate execution of an ensemble of cyclically shifted threads of a specific node sequence. Assuming the existence of one or more such sequences, a cyclo-static scheduling policy for mapping LGDF nodes onto processors for different relative iterations is proposed. A scheduling policy is classified as fully cyclo-static, block cyclo-static or static based on the relationship between nodes, iteration numbers and processors. The scheduling policy forms the basis of a distributed ATAMM Multicomputer Operating System (AMOS), which allows processors to concurrently participate in distributed dataflow and node scheduling operations. A distributed AMOS has been developed and implemented on a multicomputing testbed comprised of local area networked personal computers. Experimental results that confirm the veracity of the scheduling policy are reported. The communication overhead of the message passing mechanism of the testbed is quantitatively analyzed. Benefits claimed of this scheduling approach are, reduction in scheduling overhead, distribution of AMOS operations and the potential for using heterogeneous multicomputers.

**To Ma and Baba.**

## **ACKNOWLEDGEMENT**

This gives me an opportunity to express my gratitude and appreciation for the education, guidance and inspiration that I received from Dr. John W. Stoughton throughout the course of my graduate studies and research at Old Dominion University. Numerous technical conversations with Dr. James F. Leathrum aided my understanding of the ATAMM model and issues related to high performance computing. I would like to thank Dr. Roland R. Mielke, Dr. Stoughton and Dr. Leathrum for granting me the privilege to participate in ATAMM research.

I would like to specially mention Dr. Chester E. Grosch (of ODU Computer Science), for helping me realize the crucial distinction between the quantitative and aesthetic aspects of computer architecture design. Thanks are also due to Robert L. Jones (of the NASA Langley Research Center) for providing the software and directions for using the ATAMM Analysis Tool. To my friends Sudhir and Sudheer, I offer thankfulness for their goodwill and companionship.

While striving to achieve career milestones, support from one's family is always tacitly assumed. However, I would like to reiterate that but for the constant encouragement, support and care that I received from my mom, dad and sister, this work would have remained unfulfilled.

## TABLE OF CONTENTS

	PAGE
LIST OF FIGURES.....	vii
LIST OF TABLES.....	x
LIST OF ABBREVIATIONS.....	xii
CHAPTER	
1. INTRODUCTION.....	1
1.0 Research Focus.....	1
1.1 Context of Research.....	1
1.2 Research Objective.....	5
1.3 Thesis Organization.....	6
2. THE ATAMM DATAFLOW COMPUTING MODEL AND KEY TERMINOLOGY.....	8
2.0 Introduction.....	8
2.1 The Dataflow Paradigm.....	8
2.2 The Algorithm To Architecture Mapping Model.....	12
2.2.1 Marked Graph Models.....	14
2.2.2 Graphical Representation of Execution Behavior.....	24
2.3 Implementation of the ATAMM Model.....	26
2.3.1 Centralized AMOS Graph Manager.....	32
2.4 Performance Analysis for the ATAMM Modelling Process.....	34
2.5 Time Measurements for the ATAMM Model.....	36
2.6 Summary.....	39

3. A HYPOTHESIS FOR CYCLO-STATIC SCHEDULING AND DEVELOPMENT OF A LOCAL AREA MULTICOMPUTING TESTBED.	40
3.0 Introduction.....	
3.1 A Hypothesis for Cyclo-Static Scheduling.....	41
3.1.1 Possibility of the Existence of Periodic Node Execution Patterns.....	41
3.1.2 Cyclo-static node scheduling.....	43
3.1.3 Examples of Cyclo-Static Schedule Loops.	47
3.2 Design of a Distributed AMOS Graph Manager.....	55
3.2.1 Translation of Elements of the Hypothesis Into a Set of Requirements.....	55
3.2.2 Software Implementation of Distributed Graph Management Strategies.....	58
3.3 AMOS State Diagram and Functional Characteristics	60
3.3.1 "FIRE", "EXECUTE" and "DONE/DATA" States.....	64
3.3.2 AMOS Characteristics.....	65
3.4 Transformation of a LAN Environment to Support Multicomputing.....	70
3.5 Testbed Operation.....	75
3.6 Contrasting Distributed and Centralized AMOS Operations.....	77
3.7 Summary.....	80
4. EXPERIMENTAL RESULTS.....	84
4.0 Introduction.....	84
4.1 Utilization and Modification of Testbed Features for Experimentation.....	84
4.2 Experiments Pertaining to Scheduling Policy.....	85

4.3 Special Dataflow Graphs.....	94
4.4 FDT Time Marks for AMOS Events and Communication Overhead.....	104
4.5 Summary.....	122
5. CONCLUSION.....	123
5.1 Executive Summary.....	124
5.2 Enhancement of Testbed Features.....	126
5.3 Topics for Future ATAMM Research.....	128
REFERENCES.....	131

## LIST OF FIGURES

FIGURE	PAGE
1.1 Classification of Computer Applications on the Basis of Four Main Attributes.....	3
2.1 The ATAMM Multicomputing Environment.....	13
2.2 Components of the ATAMM Dataflow Model.....	15
2.3(a) An Example Algorithm Directed Graph.....	16
2.3(b) An Example Algorithm Marked Graph.....	16
2.4(a) Complex Node Marked Graph (NMG).....	18
2.4(b) Simplified Node Marked Graph (NMG).....	18
2.4(c) Predecessor-Successor Node Relationships.....	20
2.4(d) Node Before Firing.....	20
2.4(e) Node After Firing.....	20
2.5 CMG for the AMG in Figure 2.3(b).....	21
2.6 AMG That Requires Control Buffers and Forwarded Data Tokens.....	23
2.7 SGP Diagram for the AMG in Figure 2.3(b).....	25
2.8(a) Overlapped SGP Frames for the AMG in Figure 2.3(b).....	27
2.8(b) TGP Diagram for the AMG in Figure 2.3(b).....	28
2.9 Key Elements of an ATAMM Dataflow Multicomputer.....	30
2.10 Enhanced View of the Centralized AMOS Graph Manager....	33
3.1 Existence of R Cyclically Shifted Threads of a Node-Sequence in the Loop Frame.....	44
3.2(a) Types of Static Scheduling Policies.....	48
3.2(b) Time Measurements for a Schedule loop.....	48

3.3	Fully Cyclo-Static Schedule Loop for the AMG in Figure 2.3(b).....	49
3.4	Block Static Schedule Loop for the AMG in Figure 2.3(b).....	50
3.5	Static Schedule Loop for the AMG in Figure 2.3(b).....	51
3.6	Fully Cyclo-Static Schedule Loop for the AMG in Figure 2.3(b) Which Requires Five Resources.....	53
3.7	Events That Occur in the AMOS "FIRE" State.....	66
3.8	Events That Occur in the AMOS "DONE/DATA" State.....	67
3.9	State Machine View of Distributed AMOS Graph Manager....	68
3.10	Interaction Between AMOS States and Data Structures.....	69
3.11	Distributed AMOS and its Functional Relationship With Standard Multicomputer Operating System Components.....	71
3.12	ATAMM Testbed Components : Processing Elements and Local Area Network.....	73
3.13	Modelling of Distributed Shared Memory, DSM, for the ATAMM Dataflow Testbed.....	76
4.1	ATAMM Analysis Tool Output for Cyclo-Static Scheduling of the AMG in Figure 2.3(b).....	90
4.2	ATAMM Analysis Tool Output for Block Cyclo-Static Scheduling of the AMG in Figure 2.3(b).....	93
4.3	ATAMM Analysis Tool Output for Purely Static Scheduling of the AMG in Figure 2.3(b).....	96
4.4	CMG and TGP for a Dataflow Graph Which Exhibits Multiple Instantiation.....	97
4.5	Block Cyclo-Static Schedule Loops for the AMG in Figure 4.4.....	99

4.6	Multiple Instantiation of Node One in the AMG of Figure 4.4.	101
4.7	CMG and TGP for an Eight Node Dataflow Graph Which is Executed on a Set of Six Processors.....	102
4.8	Cyclo-Static Schedule Loop for the Eight Node AMG in Figure 4.7.....	103
4.9	ATAMM Analysis Tool Output for the Eight Node AMG in Figure 4.7.....	106
4.10	FDT Events in One TGP Frame for the AMG in Figure 4.7..	107
4.11	AMOS Events and FDT Time Marks.....	108
4.12	FDT Events Associated With Distributed AMOS Operations..	109
4.13	Time Measures Associated With the Execution of a Single Node.....	112
4.14	FDT Events in One TGP Frame of Figure 4.1.....	114
4.15	FDT Events in One TGP Frame of Figure 4.6.....	115
4.16	FDT Events in One TGP Frame of Figure 4.9.....	116
5.1	Enhanced State Machine View of Distributed AMOS Graph Manager.....	127

## LIST OF TABLES

TABLE		PAGE
2.1	Node Execution Time Units for the AMG of Figure 2.3(b).....	25
3.1	Possible Schedule Loops for the AMG in Figure 3.2(b).....	54
3.2	Basic Connection Matrix for the AMG in Figure 2.6.....	61
3.3	Transpose of Connection Matrix for the AMG in Figure 2.6...	61
3.4	Augmented Connection Matrix for the AMG in Figure 2.6.....	62
3.5	Scheduling Table.....	62
3.6	Initialization Table.....	63
3.7	Assignment Table.....	63
3.8	Modulo Operator Table.....	63
3.9	Features of Distributed and Centralized AMOS.....	81
4.1	Format of the Specifications File.....	86
4.2	On Line Screen Display for Testbed Operations.....	87
4.3	Format of the System Log File.....	87
4.4	FDT File Format as Input to ATAMM Analysis Tool.....	88
4.5	Specifications File for the TGP in Figure 4.2.....	89
4.6	Specifications for a Bloc Cyclo-Static Schedule for the AMG in Figure 2.3(b).....	92
4.7	Specifications for a Static Schedule for the AMG in Figure 2.3(b).....	95
4.8	Specifications for a Cyclo-Static Schedule for the AMG in Figure 4.4.....	100

4.9	Specifications for a Cyclo-Static Schedule for the Eight Node AMG in Figure 4.7.....	105
4.10(a)	TBO Measurements for Figure 4.14.....	117
4.10(b)	TBO Measurements for Figure 4.15.....	117
4.10(c)	TBO Measurements for Figure 4.16.....	118
4.11	TBI Measurement for Figure 4.9.....	120

## LIST OF ABBREVIATIONS

ABBREVIATION	AMPLIFICATION
ADG	Algorithm Directed Graph.
ADAM	ATAMM Dataflow Multicomputer.
AGM	AMOS Graph Manager.
AMG	Algorithm Marked Graph.
AMOS	ATAMM Multicomputer Operating System.
ATAMM	Algorithm To Architecture Mapping Model.
CAMG	Centralized AMOS Graph Management.
CB	Length of Control Buffer on a Control Arc.
CMG	Computational Marked Graph.
CCr	Critical Circuit.
CP	Critical Path.
Ctrl.	Control.
D	"Done/Data" sub-node of the simplified NMG.
DAGM	Distributed AMOS Graph Management
F	"Fire" sub-node of the simplified NMG.
FDT	Fire, Data, Time events for the ATAMM Analysis Tool.
FGDF	Fine Grain Dataflow.
FU	Functional Unit.
<i>i</i>	Iteration number <i>i</i> .
IE	Input Buffer Empty.
IF	Input Buffer Full.
IPC	Inter-Processor Communication.

LAN	Local Area Network. The testbed uses a 10 MBps thin-ethernet with peer-to-peer (Novell NetWare Lite) software.
LGDF	Large Grain Dataflow.
MIMD	Multiple Input Multiple Data (parallel computers).
NMG	Node Marked Graph.
N	Number of AMG nodes.
NCO	Number of Control Tokens Output.
NDO	Number of Data Tokens Output.
$N_{PE}$	Number of Processing Elements required to execute an AMG.
NT	Intrinsic Node Execution Time.
PE	Processing Element.
PID	Processor Identification Number.
PR	Process Ready.
R	(i) The number of processors required to satisfy a specified TGP. (ii) "Resource" command used by a CAGM.
RAM-disk	Portions of RAM of a Personal Computer configured as Disks.
$R_{max}$	The number of processors required to execute an AMG with $TBO = TBO_{LB}$ .
SA	Static Assignment.
TBI	Time Between Inputs.
TBIO	Time Between Inputs and Outputs (computing time).
$TBIO_{LB}$	Lower Bound on TBIO.
TBO	Time Between Outputs (throughput).
$TBO_{LB}$	Lower Bound on TBO.
$TBO_{min}$	Minimum TBO due to overhead requirements.
TCE	Total Computing Effort.
$T_{wait}$	The total waiting time between nodes of a node-loop.

(X,Y,Z)

Connection Matrix entries of the form : (Iteration Increment on Data Arc, Number of Initial Data Tokens, Length of Control Buffer on Corresponding Control Arc).

# CHAPTER ONE

## INTRODUCTION

### 1.0 Research Focus

A strategy for executing deterministic large grain dataflow algorithms in a distributively scheduled, multicomputing environment is presented in this thesis. Modelling concepts and multicomputer performance metrics specified by the ODU/NASA developed Algorithm To Architecture Mapping Model, ATAMM, form the theoretical basis of this research. ATAMM uses marked graph models to specify the criteria for achieving dead-lock free execution of iterative and deterministic large grain dataflow algorithms on multicomputer architectures [STOUGHTON86]. An ATAMM multicomputing testbed comprised of networked personal computers is developed in order to evaluate and demonstrate the proposed distributed scheduling approach.

### 1.1 Context of Research

Computing applications are characterized by a set of functional criteria such as level of user-interaction, throughput rate and processing capability. The computational complexity of these problems typically range from  $10^6$  to  $10^{15}$  or more megaflops. The corresponding spectrum of computational resources includes PCs (under 10 megaflops), low and medium range workstations (10-100 megaflops), multiprocessor workstations, multicomputer workstation clusters, servers such as mainframes and entry-level supercomputers (parallel vector processors or massively parallel processors under 1000 megaflops) and supercomputers that deliver performance in the gigaflops range. The performance boundaries of these classes of computers are constantly advancing. At the same

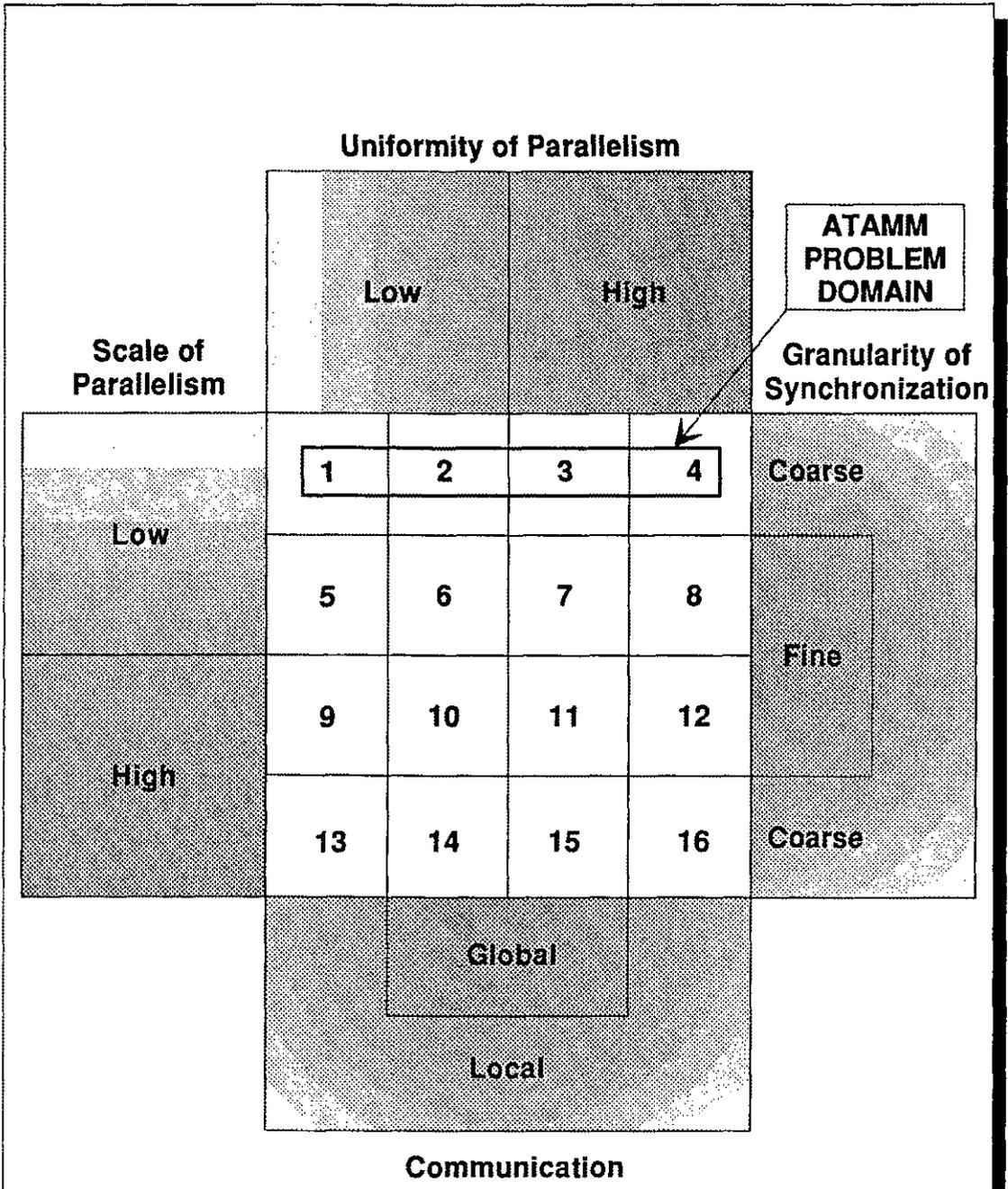
time applications are evolving to demand more computational power. Hence it is important to associate computational characteristics of applications with computing capabilities of appropriate classes of resources.

Problems can be classified not only on the basis of complexity and functional requirements but also in terms of their algorithmic and computational features. A taxonomy of applications is presented in Figure 1.1 that uses four main attributes: scale of parallelism, uniformity of parallelism, granularity of synchronization and communications. For simplicity, each of the four attributes is shown with only two values. In reality, each attribute constitutes a continuum.

An application's scale of parallelism corresponds to the number of processors that could be kept busy if an unlimited number of processors were available. Uniformity of parallelism is the uniform average of the scale of parallelism as a function of time. An application has coarse grain synchronization if there are many operations between synchronization points. In contrast, the application is fine grained if the number of these operations is small. Local communications occur across single links to which a resource is directly connected where as global communications occur across multiple links [FURTNEY93].

Of interest to this research are applications that are computationally intensive but require coarse grain synchronization and communication. A class of computers that are increasingly being applied to solve such problems are networked workstations. The interconnection networks for loosely coupled workstation clusters (forming a "meta computer") have long latencies and low communication bandwidths in comparison with tightly coupled supercomputers, making them suitable for problems with coarse granularity. These machines collaborate on problem-solving, using a communications mechanism such as a message-passing library or distributed shared memory [FURTNEY93].

The problem domain addressed by ATAMM includes Large (coarse) Grain DataFlow (LGDF) applications that are deterministic in nature. A dataflow algorithm is characterized by data driven computations as opposed to control driven



**Figure 1.1 Classification of Computer Applications on the Basis of Four Main Attributes, [FURTNEY93].**

computations in conventional stored program computers. Large grain dataflow problems contain macro blocks of code (instructions) that get executed whenever required data is available at input. Of special interest are iterative LGDF algorithms whose block computation times remain constant. These algorithms are termed as deterministic, since they exhibit periodic behavior in steady state.

A dataflow algorithm is represented as a directed graph in which nodes and arcs stand for instruction blocks and their data dependencies, respectively. Given such a decomposed dataflow algorithm, ATAMM uses a set of marked graph models to expose its control as well as data dependencies. Using Gantt chart representations of algorithm performance in steady state, the ATAMM model specifies measures for throughput and computing time that form the criteria for predicting performance based on the number of available computing resources. ATAMM is manifested in software as the ATAMM Multicomputer Operating System, AMOS, which is implemented on real-time multicomputer architectures to achieve predictable, reliable and deadlock-free performance of LGDF computations.

An earlier version of AMOS (developed at the NASA Langley Research Center), is the Advanced Development Module (ADM), a four processor architecture based on the Westinghouse MIL-STD-VHSIC 1750A instruction set processor. A present version of AMOS is being developed for the Generic VHSIC Spaceborne Computer, GVSC, a spaceborne four processor breadboard which also is based on the VHSIC 1750A [MIELKE90].

In these embodiments of ATAMM, AMOS code is executed on all processors in a multi-threaded fashion. Processor assignment is governed by a processor queue that allows only one processor to schedule a node for execution at a time. The operation of scheduling a node for execution is performed dynamically by a processor upon examining the current state of execution of the algorithm. Consequently, the processor assignment for a given node and iteration is not known beforehand. It becomes necessary to redundantly broadcast AMOS data structures and computed data across all processors of the system. The

operation of AMOS in the ADM and GVSC relies on preserving the total graph view across every processor at all times. AMOS operation in these systems may be termed as centralized since only one processor at a time can supervise a node scheduling operation. The assignment policy of a centralized AMOS with a single queue requires processors to be homogeneous. However a distributed heterogeneous processing environment can be created by establishing different queues for each class of computing resource in the system.

An alternative to the above is a static and deterministic scheduling approach that establishes a specific mapping of nodes to processors for every iteration of the dataflow algorithm. The motivation for this form of scheduling is derived from the periodic execution behavior observed in deterministic LGDF algorithms. It is likely that the steady state execution of nodes gives rise to periodic node execution patterns, which are repeated in time and node-space, thereby possibly forming the basis for establishing a static mapping of nodes to processors for different relative iterations. An AMOS that incorporates predetermined node schedules offers several new features. Scheduling operations may be distributed among processors such that processors concurrently participate in scheduling nodes for execution. In this sense, AMOS graph management operations such as node scheduling and processor assignment may be performed in a distributed manner. Since it is known beforehand for every iteration which processor executes a given node, it becomes possible to obviate the need for redundant communication. More importantly, since a static schedule is specified at compile time, the run time overhead associated with dynamic scheduling is not seen. A potential benefit of deterministic scheduling is the ability to directly incorporate heterogeneous processors in a distributed ATAMM multicomputing system.

## **1.2 Research Objective**

The research developed in this thesis is aimed at,

- [1] developing a strategy for distributively and deterministically scheduling LGDF algorithms under AMOS and

- [2] implementing the scheduling policy on a multicomputing testbed that would provide an experimental vehicle for demonstrating and evaluating distributed dataflow computations in an ATAMM environment.

It may be possible to equate the periodic execution behavior of deterministic LGDF algorithms to the aggregate execution of an ensemble of cyclically shifted threads of a specific sequence of algorithm nodes. Assuming the existence of such node sequences, a hypothesis for distributively scheduling LGDF nodes shall be proposed. This scheduling policy shall be used to develop a distributed AMOS that would permit processors of an ATAMM multicomputing system to participate in distributed graph management. In order to evaluate and demonstrate this approach, a testbed incorporating a distributed AMOS shall be developed. Personal computers networked through an ethernet LAN shall form the hardware for this testbed. The testbed shall be configured to generate data compatible with multicomputer evaluation software such as the ATAMM Analysis Tool [JONES93].

### **1.3 Thesis Organization**

The theoretical background for the research described in the thesis is established in Chapter Two. The dataflow paradigm which constitutes the problem domain addressed by ATAMM, is discussed initially. The components of the ATAMM model are described. The interaction between elements of the ATAMM multicomputing environment, an offline modelling process, an ATAMM Multicomputer Operating System (AMOS), and the targeted multicomputer architecture are described. A state machine view of a centralized AMOS is presented along with a discussion of macro operations that characterize centralized graph management. Time measures that aid performance analysis of an ATAMM system are also described.

An inductive view of the design, development and implementation of distributed AMOS operations on a local-area dataflow testbed is presented in Chapter Three. The steady state execution of iterative dataflow algorithms may

be characterized by periodic node execution patterns. It is postulated that such periodic node sequences can form the basis of a hypothesis for deterministically scheduling nodes. Deterministic node schedules are classified as cyclo-static, block cyclo-static or static to represent a relative variance in node to processor mappings during algorithm execution. Based on the hypothesis for cyclo-static behavior of node sequences, a strategy for distributed processor assignment and node scheduling for AMOS is developed. An information structure that is sufficient to implement the hypothesis is developed. Subsequently, a state machine view of a distributed AMOS is developed. The transformation of a LAN environment to support multicomputing is then described. It is shown that peer-to-peer accessible network directories can be used to represent a distributed shared memory model that is used for message passing. The key features of centralized and distributed ATAMM operations are contrasted. The chapter concludes with an executive summary of the concepts developed.

The testbed's ability to execute dataflow algorithms scheduled by a distributed AMOS is established in Chapter Four. Results of numerous preliminary experiments are presented in this chapter to demonstrate the capabilities of the testbed. The ATAMM Analysis Tool [JONES93] is used to portray graphically the execution behavior of dataflow algorithms run on the testbed. The handling of the three types of cyclo-static scheduling possibilities is demonstrated. The execution of graphs with special dataflow properties is shown including an eight-node example that uses all six processors of the testbed. The message passing mechanism of the testbed introduces a communication overhead that causes execution to deviate from ideal conditions. AMOS events that contribute to the communication overhead are quantified.

An executive summary of the research effort forms the initial part of Chapter Five. The capabilities of the testbed are critically evaluated. Possible enhancements to the testbed's current features are suggested and areas for future ATAMM research are identified.

## **CHAPTER TWO**

### **THE ATAMM DATAFLOW COMPUTING MODEL**

#### **2.0 Introduction**

The underlying theoretical framework for the research developed in the thesis is established in this chapter. The dataflow concept is introduced in Section 2.1 and its effectiveness in circumventing the bottlenecks of conventional parallel processing is asserted. The ODU/NASA developed Algorithm To Architecture Mapping Model, ATAMM, which associates itself with the execution of large grain dataflow computations, is presented in Section 2.2. Terminology and techniques that define the modelling process are also introduced here. The discussion in this Section leads to the identification of key system components of an ATAMM based multicomputer, which are presented in Section 2.3. Performance measures for a dataflow algorithm executed under the ATAMM environment are quantified in Section 2.4. The relationship between computing effort and resource requirements is examined in Section 2.5.

#### **2.1 The Dataflow Paradigm**

Dataflow has proven to be an attractive computational model for Multiple Input Multiple Data (MIMD), machines for its ability to alleviate control dependence from conventional parallel programming [AGERWALA82]. Data driven computation eliminates performance bottlenecks of stored program computers, namely, the narrow locality of control and the overhead of task synchronization.

A program in a high level dataflow language is directly translatable into a directed graph whose nodes (circles) represent asynchronous operations and whose arcs (arrows) represent data dependencies as well as communication paths

between nodes. In a dataflow graph, data values are represented as tokens (bullets) on the arcs. For a node to become enabled (i.e. empowered for execution), tokens must be present on each of the input arcs incident on the node. Any enabled operation can be "fired" (executed) by removing one token from each input arc, applying the specified operations to the data present on the token, and placing token(s) labeled with the resulting value on the output arcs [RISHE91].

A key advantage is the spontaneity with which node operations become enabled for execution. The instant all requisite input tokens are made available, a node can consume them and proceed with task execution. In turn, this implies that each node in the algorithm can be fired at a rate that is ideally dictated only by the inter nodal data dependencies of the algorithm. Thus, the dataflow paradigm satisfies the key execution requirements of general concurrent algorithm structures, i.e. asynchrony and functional independence [ERCEGOVAC86]. This permits a true match between actual execution characteristics and the natural representation of the intended "graph play". The term graph play in this context, refers to the execution pattern that we wish to derive out of a dataflow algorithm.

Dataflow computing can be classified on the basis of the level of granularity of dataflow operations. The number of primitive operations assigned to a node, defines the granularity of the algorithm. Dataflow graphs may possess nodes that symbolize simple operations or relatively complex blocks of instructions. For instance, nodes may represent atomic (fine grain) operations such as additions, multiplications and simple binary operations, or stand for non-atomic (large grain) functions such as digital filtering and FFT computation. Consequently we familiarize ourselves with two forms of dataflow computation, Large Grain Data Flow (LGDF) and Fine Grain Data Flow (FGDF). These forms of dataflow computation are also termed as macro or micro dataflow in the literature [ERCEGOVAC86]. Computational models and support mechanisms are available for implementing macro or micro dataflow algorithms on dataflow parallel computers [BABB82],[LEE87],[STOUGHTON88]. The ATAMM model, for instance,

is primarily concerned with the iterative and deterministic execution of Large Grain Dataflow algorithms.

Large grain dataflow models are increasingly being used for real time control and signal processing algorithms in diverse areas such as aerospace, factory automation, military applications, nuclear power, etc. Characteristics desired of real time dataflow computers encompass the ability to iteratively execute algorithms reliably with a high degree of performance predictability (determinism) and the capability to comply with well defined graph output characteristics. A large grain dataflow computational model aptly suits the industry's requirement of being able to run real time computing problems on decentralized and homogeneous multicomputer architectures [SOM93]. A few example architectures developed for aerospace and military applications are the Advanced Development Model (ADM), avionics computers [MIELKE90], the Generic VHSIC Spaceborne Computer (GVSC) [MIELKE90] and the MAX dataflow machine[RASMUSSEN87].

In addition, the LGDF computing paradigm has been used as a natural model to describe signal processing systems where LGDF nodes are second order recursive digital filters, FFT butterfly operators, adaptive filters, and so on. Such a description exhibits much of the available concurrency in a signal processing algorithm, making multiple processor implementation easier to achieve. An example of a pipelined dataflow architecture for digital signal processing is the NEC  $\mu$ PD7281 [CHASE84].

In order to execute a LGDF algorithm on a parallel computer, other than inter-nodal data dependencies, additional information related to node scheduling, node execution and overall algorithm management are required. Enhanced graph models have been proposed, that reflect the control and dataflow processes that collectively govern algorithm execution. Information entities pertaining to the above manifest as control tokens or data tokens. A control token bears information ranging from execution data to opcode and operations, depending on the dataflow implementation. A data token is formed by combining result values with destination information. The execution process begins by scheduling nodes for execution with

available processors of a dataflow parallel computer. Node scheduling refers to the mapping of nodes onto processors of a parallel system. Scheduling has to be performed in a coherent manner that assures that data are available to execute a node that is invoked on processor. In order to achieve this goal, dataflow tokens are passed among various processors (resources) in the parallel system. It may be noted peripherally that this particular form of message passing allows a dataflow computer to readily assume a distributed organization [HWANG84].

Scheduling strategies may assign nodes for execution dynamically. In this case a runtime supervisor determines when nodes are ready for execution and schedules them onto processors as they become free. This runtime supervisor may be a software routine or specialized hardware and is the same as the control mechanisms generally associated with dataflow. This is a costly approach, however, in that the supervisory overhead can become severe, particularly if relatively little computation is done each time a block is invoked [LEE87].

However for certain classes of LGDF applications which are deterministic in nature, scheduling may be performed statically, thereby obviating the control overhead associated with dynamic scheduling. Deterministic LGDF algorithms are characterized by fixed node execution times, regular inter-nodal communications and constant throughput rates. Examples of such algorithms occur abundantly in hard real time and DSP applications. The importance of such algorithms is that scheduling of nodes can be done at compile time (statically) so that runtime overhead evaporates. Specifically, a static large grain compiler determines the order in which nodes can be executed and constructs a static schedule of node execution for every processor in the system. Communication between nodes and between processors is set up by the compiler, so no runtime control is required beyond the traditional sequential control in processors. Strategies for static scheduling and the feasibility of large grain compilers are discussed in [LEE87] and [SCHWARTZ85].

To fortify the concepts introduced in this Section, the relative advantages of dataflow computing are reiterated [AGERWALA82]. They are:

- [1] elimination of control dependence which obviates the need of inter-processor synchronization, as required in conventional MIMD machines;
- [2] execution of algorithms that fall within the class termed as general concurrent, implying thereby the ability to execute dataflow instructions spontaneously and asynchronously;
- [3] dataflow operations behave like stand-alone functions, thus providing freedom from any side effects (that would otherwise appear if control were explicitly sequential);
- [4] dataflow instructions do not introduce sequencing constraints other than the ones imposed by the data dependencies of the algorithm.
- [5] dataflow programs can be verified effortlessly and software errors can be confined with relative ease;
- [6] dataflow nodes are modular and may be reused in new system designs; and
- [7] dataflow computers exhibit modularity and scalability of hardware.

## **2.2 The Algorithm to Architecture Mapping Model**

The Algorithm To Architecture Mapping Model, ATAMM, is a marked graph dataflow model for mapping algorithms onto multicomputer architectures for operation in real time. ATAMM is involved with the implementation of periodic, decision free, large grain algorithms in ATAMM based multicomputer architectures. The modelling process of ATAMM is manifested in software as the ATAMM Multicomputer Operating System, AMOS. The multicomputer architecture is assumed to consist of few identical Processing Elements, PEs, or resources, each having capabilities for processing, operand and code storage and communication. The interaction between the logical components of an ATAMM large grain dataflow multicomputer and its attributes are graphically portrayed in Figure 2.1.

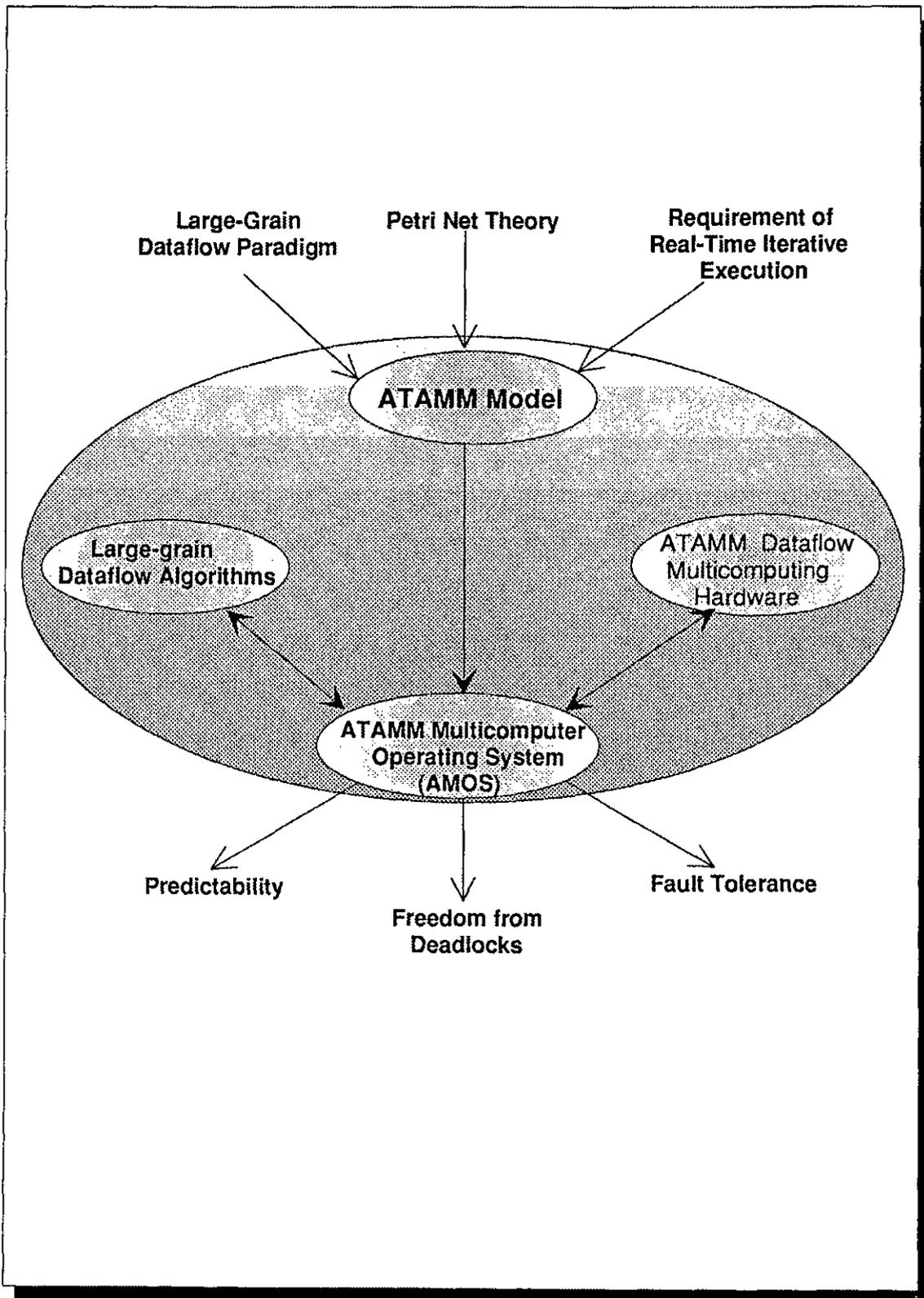


Figure 2.1 The ATAMM Multicomputing Environment [SOM93].

### 2.2.1 Marked Graph Models

The ATAMM model provides the analytical means to integrate algorithm dataflow with the target architecture. It is based on a special class of timed Petri nets which lend themselves to dataflow system modelling. A timed Petri net is a directed graph capable of describing data and control flow within a computational system. Decision free, timed Petri nets can be simplified into marked graphs. Petri nets are discussed in detail in [PETERSON81].

A set of three marked graphs, the Algorithm Marked Graph (AMG), the Node Marked Graph (NMG) and the Computational Marked Graph (CMG) constitute the main components of the ATAMM model [STOUGHTON88]. A flow diagram portraying the ATAMM modelling steps is presented in Figure 2.2.

Given an algorithm decomposition, the Algorithm Directed Graph (ADG) is used to describe data dependencies. An example ADG is portrayed in Figure 2.3(a).

The AMG is a marked graph representation of the ADG. Circles on the graphs denote algorithm nodes ("chunks" of macro dataflow code). The edges of the AMG represent data dependencies between predecessor and successor nodes, while bullets on these edges represent the presence of data tokens. Squares represent sources and sinks, thereby providing data entry and output collection points.

An algorithm node is enabled for firing when it has data tokens on all its incoming data arcs. The node fires by encumbering all input tokens, delaying for some time interval and depositing one data token on each outgoing edge. The AMG fuses algorithm data dependencies with the those imposed by execution requirements for the ADG. For example the AMG for the ADG in Figure 2.3(a) is presented in Figure 2.3(b). An initial data token has been deposited on the data edges from node two to one and node three to two, thus satisfying the initial execution conditions for the graph.

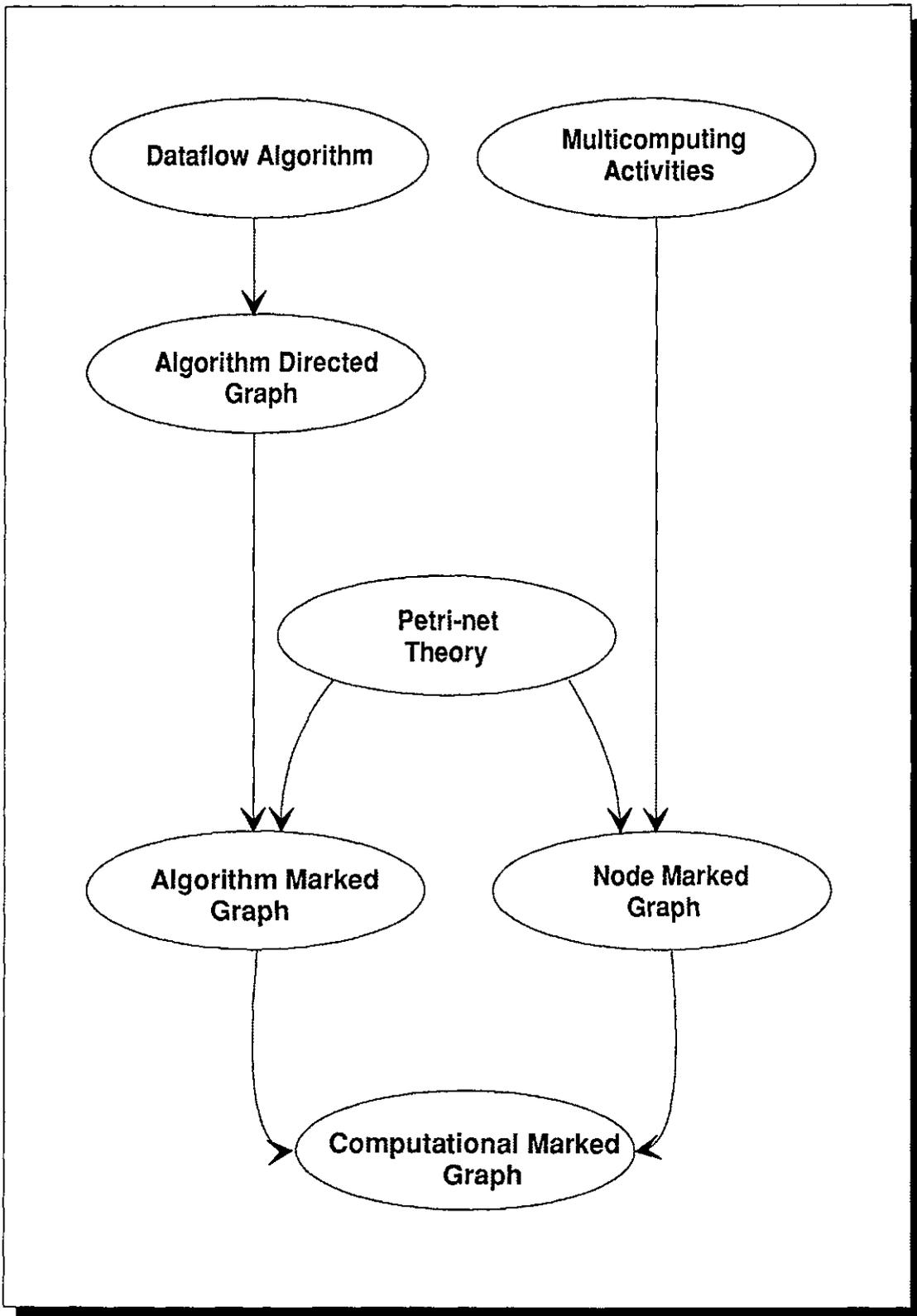
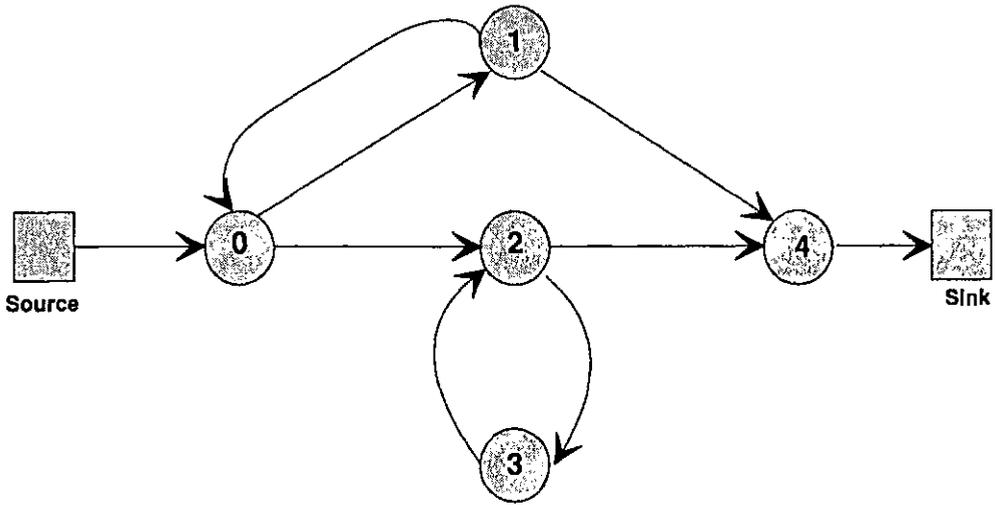
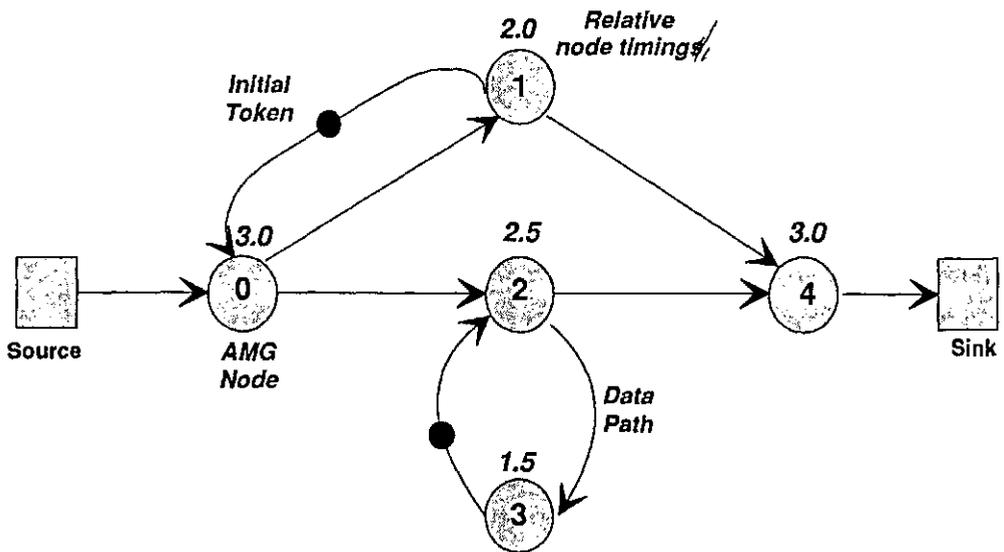


Figure 2.2 Components of the ATAMM Dataflow Model.



2.3(a) An Example Algorithm Directed Graph (ADG).



2.3(b) An Example Algorithm Marked Graph (AMG).

The AMG thus represents task decomposition and data dependence between processes in an effective fashion. However, it fails to illustrate the ensemble of multicomputer activities that need to be performed in order to ensure proper graph execution. To overcome this drawback, the PE activities pertaining to a primitive (node) operation, are modeled in the Node Marked Graph (NMG). Given certain assumptions about the computing environment, the NMG describes node specific activities and primitive control cum data dependencies that need to be satisfied, to guarantee deadlock free dataflow operation. One assumption is that the computing environment contains global memory for storage of data associated with each AMG edge. The global memory is distributed or shared among PEs, to aid the execution of primitive operations of the AMG. Each PE also contains local memory for the storage of data and the code to execute any primitive operation of the AMG. A PE must read data from global memory into its local data container, process the data and write the data back to global memory for access by other PEs. It is also assumed that a PE will not be able to start processing a primitive operation until output data for that operation has been read which ensures that data are not lost.

The NMG portrayed in Figure 2.4(a) describes the above node activities. The NMG specifies not only the activities to be performed at a PE but also the conditions which enable them to be performed. The read node cannot be fired on a PE until the processor is ready, input is available, and the output has been read by the successor operation. Once the PE is assigned to "fire" the read transition, it will remain assigned in order to process and write the data before becoming available once again [JONES90].

It can be shown that the NMG depicted in Figure 2.4(a) can be remodeled with fewer transition edges, which satisfy the necessary and sufficient conditions for algorithm execution [TYMCHYSHYN88]. The reduced NMG is presented in Figure 2.4(b). The reduced AMG contains "Fire" and "Done/Data" nodes.

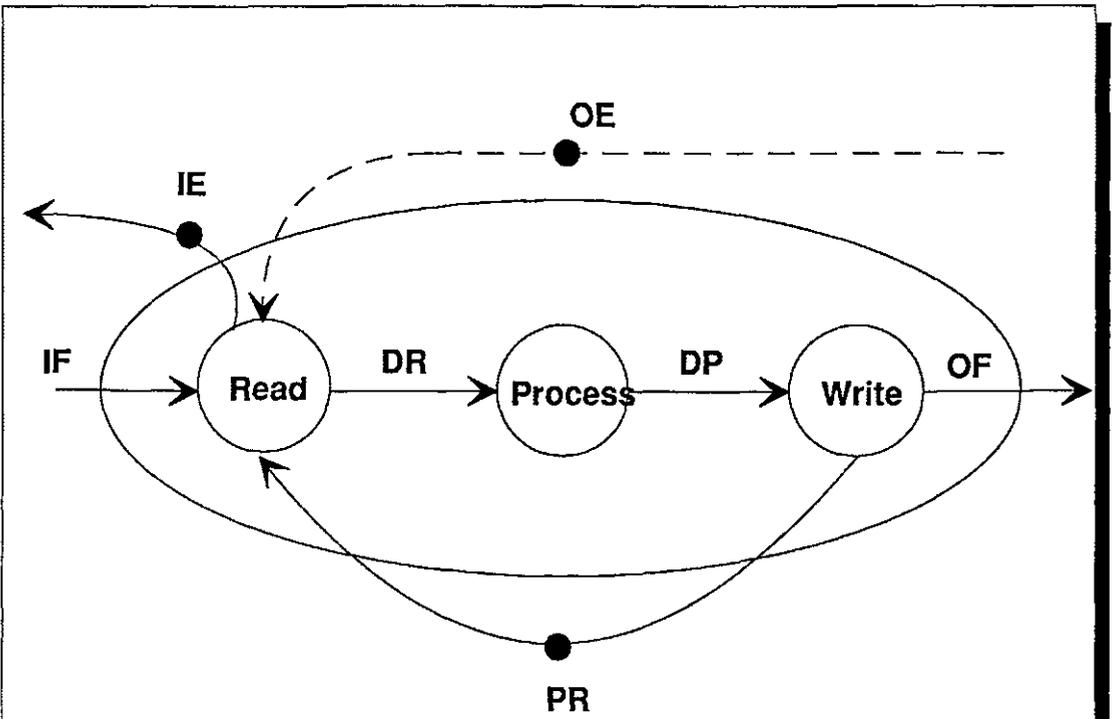


Figure 2.4 (a) Complex Node Marked Graph (NMG).

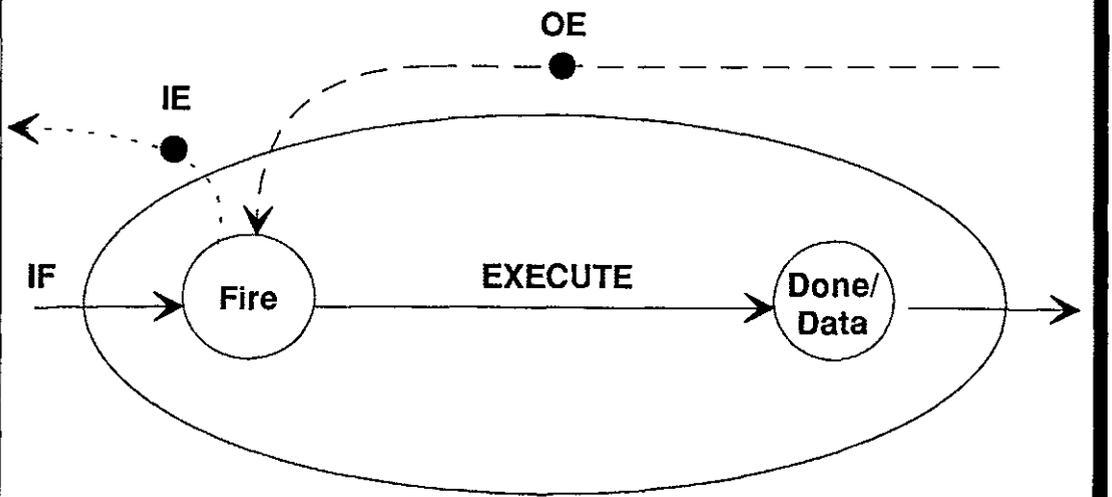


Figure 2.4 (b) Simplified Node Marked Graph (NMG).

The execution states of the NMG are shown in Figures 2.4(c) through 2.4(e). The "fire" node receives two kinds of edges, data and control. Incoming data edges pertain to the data paths from predecessor nodes. Incoming control edges arrive from the fire nodes of successor nodes. Analogously, the "done/data" node sends data edges to the fire nodes of successor nodes. This relationship between the nodes of predecessor and successor nodes is portrayed in Figure 2.4(c).

The activities pertaining to the execution of a node are shown in Figures 2.4(d) and 2.4(e). The necessary condition to be satisfied before a node is fired, is to ensure the presence of all requisite data and control tokens on respective edges. A polling process within the fire node regularly samples its local memory to test for the presence of all required tokens. Nodes fire by consuming these tokens, once they become available. Upon completion of the firing process, the done/data node generates output tokens on all of its outgoing data edges. These output data tokens in turn, become the input data tokens for the successor nodes.

The AMG with its portrayal of data dependencies and the NMG with its depiction of node activities steer us to the next logical step in the ATAMM model. A fusion of these marked graphs generates the Computational Marked Graph, (CMG), which incorporates the aggregate information content of the AMG and NMG. The CMG displays the data and control flow necessary to implement a decomposed algorithm on a dataflow architecture. The CMG is constructed by replacing each AMG node by the NMG. Each AMG edge is replaced with one forward directed edge for dataflow and one backward edge representing control flow. The CMG built out of the AMG in Figure 2.3(b) is shown in Figure 2.5, with initial markings [JONES90].

The presence of CMG control and CMG data edges for every AMG data edge creates loops in the resultant CMG. Before a node in a dataflow graph can be fired, it must have a token on every incoming edge. Consequently, initial control tokens are needed on control edges, in order to ensure the first time execution of

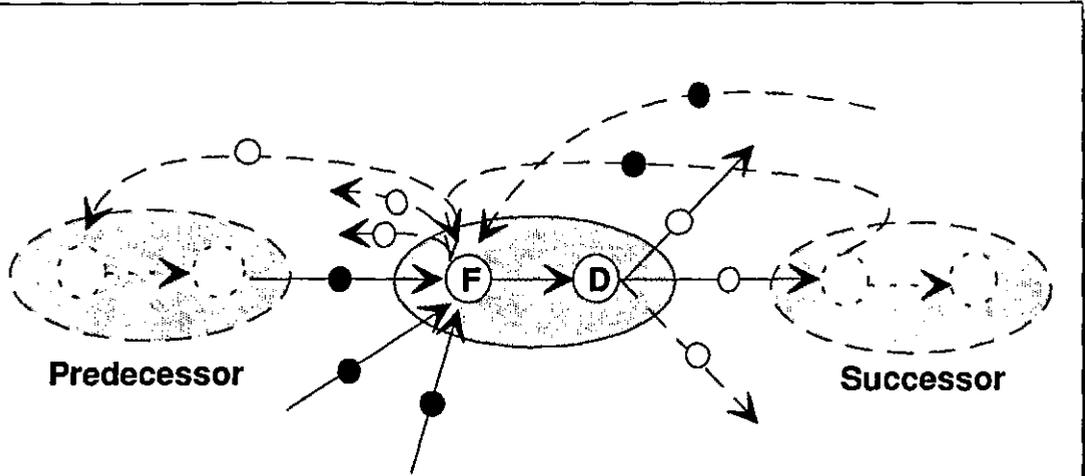


Figure 2.4 (c) Predecessor-Successor Node Relationships.

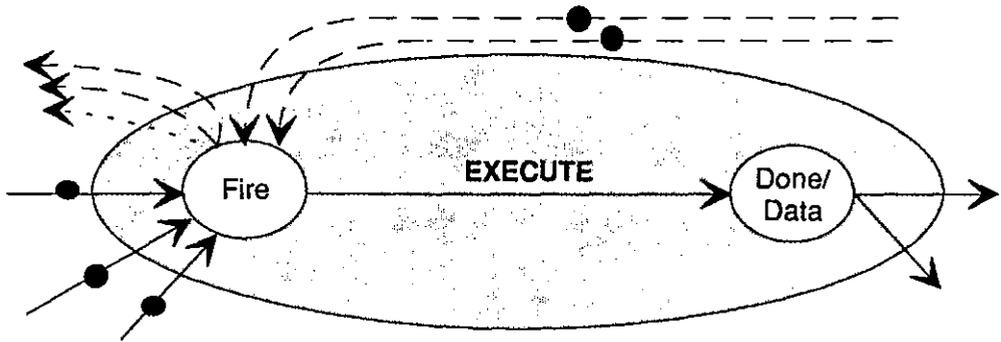


Figure 2.4 (d) Node Before Firing.

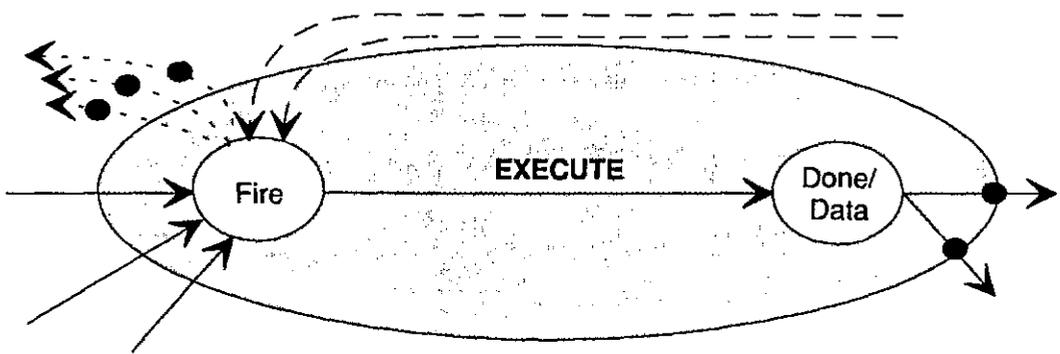


Figure 2.4 (e) Node After Firing.

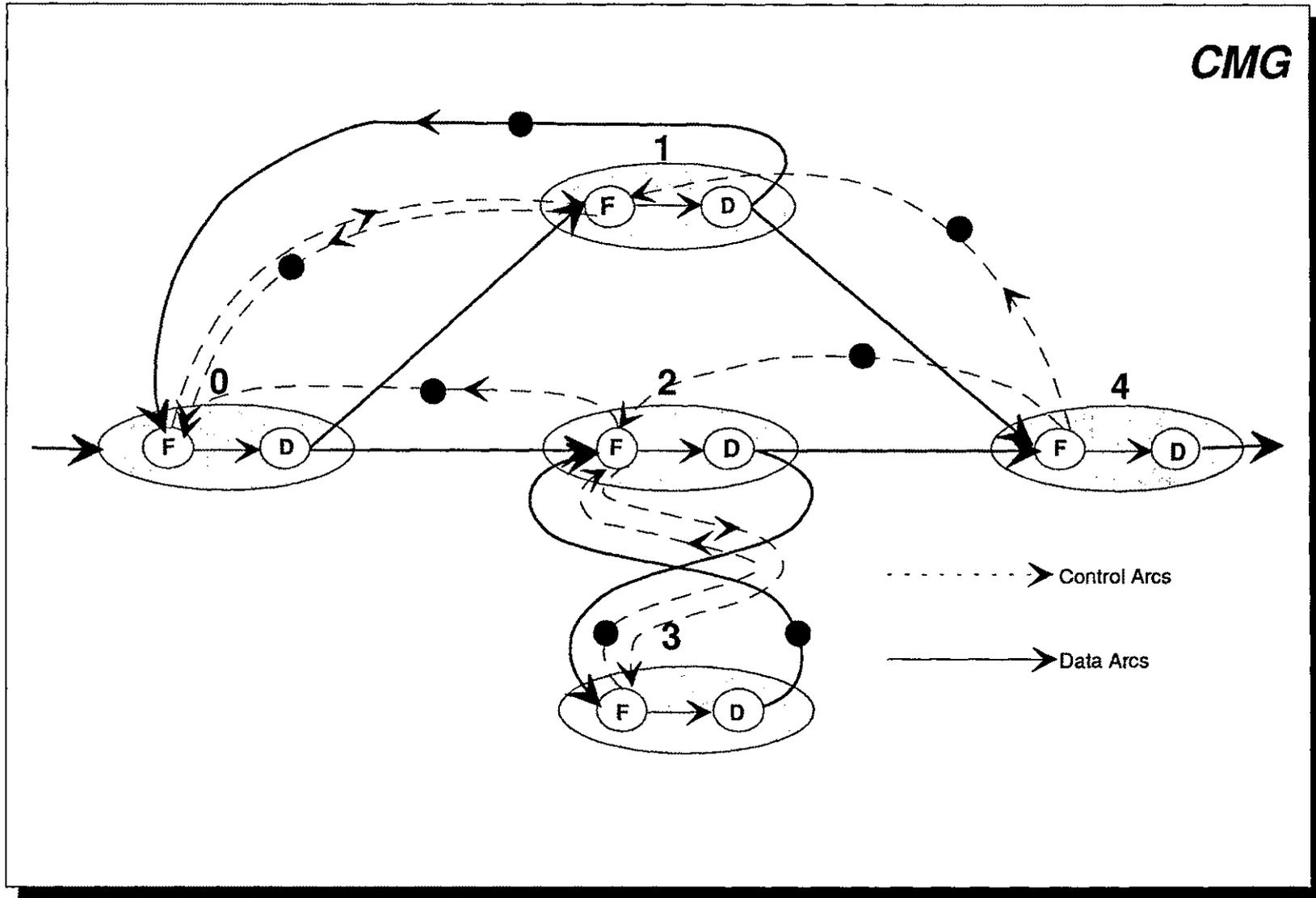


Figure 2.5 Computational Marked Graph for the AMG in Figure 2.3(b).

all AMG nodes. The initial tokens shown in Figure 2.5 satisfy initial control token dependencies of the graph. Note that data dependencies are automatically satisfied as the CMG is executed.

Special dataflow execution requirements (in terms of graph features) can also be incorporated in the CMG. For instance, in order to sustain a specific rate of execution, additional control tokens may be needed on incoming CMG control edges to a particular node. The presence of more than one control tokens on CMG control edges establishes CMG control buffers. Furthermore, data tokens on outgoing data edges are usually generated for the current iteration index. However for certain types of data edges (such as loops), data tokens may need to be generated for future iterations. Such data tokens are termed as forwarded data tokens. Iteration increments corresponding to forwarded tokens can be marked against data edges in the CMG. The AMG and CMG of a dataflow graph that requires buffers on CMG control arcs and forwarded data tokens is shown in Figure 2.6(a) and 2.6(b) respectively.

Execution of the CMG results in live, reachable, safe, deadlock free and consistent behavior. Liveness indicates that every transition of the graph can be fired from the initial marking. Reachability implies that an output will be produced for every input. The CMG is safe because the backward control edges prevent data from being overwritten, by disallowing the enablement of a node until previous output data are picked up. The CMG is also deadlock free because once assigned to a node, a PE always completes execution. Consistency implies that the CMG periodically produces output when input is applied periodically [MIELKE88]. This also implies that nodes are executed periodically.

The CMG decomposition of an algorithm creates an opportunity to realize both parallel and pipeline execution concurrencies. Operations belonging to the same data set which are independent of each other may be executed simultaneously. This is referred to as parallel concurrency and provides parallelism on a single data set.

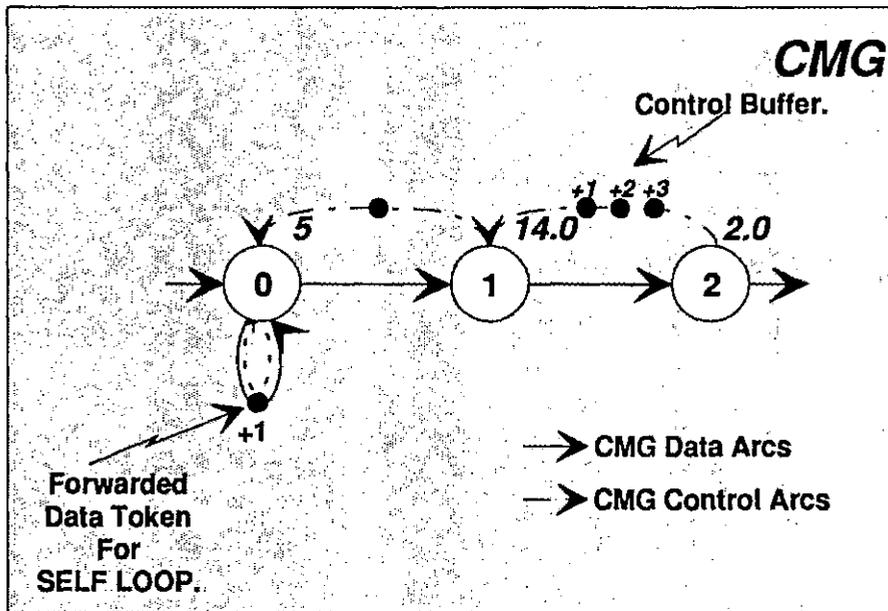
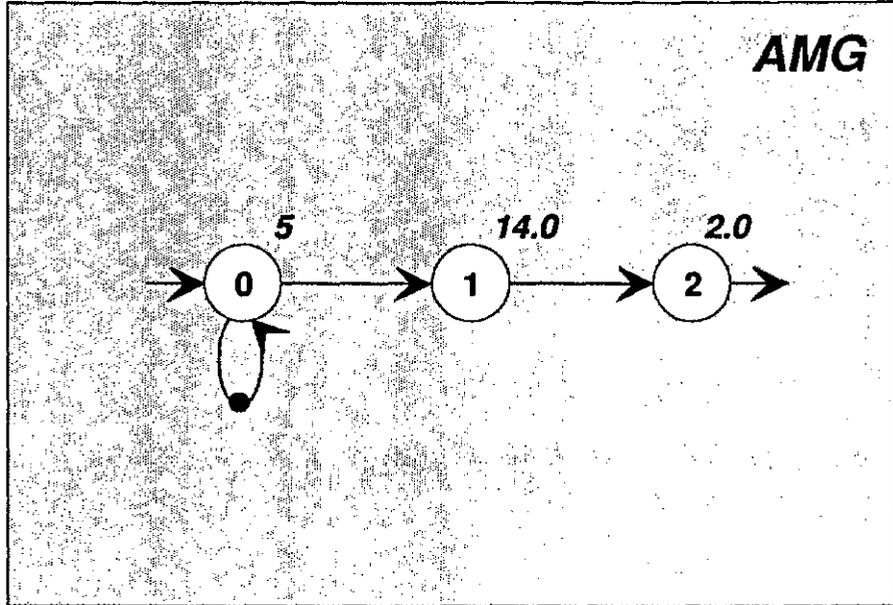


Figure 2.6 AMG That Requires Control Buffers and Forwarded Data Tokens. (Refer to Figure 4.4 for Execution Behavior of this Graph).

The amount of parallel concurrency possible depends on the number of parallel, mutually exclusive paths in the algorithm decomposition and the number of PEs available. Secondly, new data sets can be accepted for execution before the completion of previous data set computations. The simultaneous processing of different data sets is referred to as pipeline concurrency. The amount of pipeline concurrency possible depends on the ability of the algorithm decomposition to accept new data sets and the number of PEs available.

### 2.2.2 Graphical Representation of Execution Behavior

Two diagrams which illustrate execution behavior are labelled as the Single Graph Play (SGP) and the Total Graph Play (TGP). These diagrams are useful for determining the number of resources needed to achieve specific performance goals. In this context, they may be compared with Gantt chart representations, which are used for depicting pipelined activities. The SGP diagram provides a view of the parallel concurrency derivable from a given AMG, while the TGP provides a view of both parallel and pipeline concurrency.

The SGP diagram displays the execution of each node of the AMG as a function of time. The diagram is constructed for a single input data packet under the assumption that unlimited resources are available to play the graph. Node activity is denoted by a double ended ray. When several nodes are active at the same time, lines or bars indicating node activity are stacked vertically so that computing concurrency is apparent. The SGP diagram for the AMG of Figure 2.3(b) is shown in Figure 2.7. The SGP diagram has been constructed on the basis of specified execution time units for each node in the AMG, as shown in Table 2.1.

The TGP diagram displays the execution of each graph node when the graph is operating periodically in steady state. It shall be shown later that the operation period is the Time Between Outputs, TBO. As with SGP, the TGP is constructed under the assumption that unlimited resources are available to play the graph.

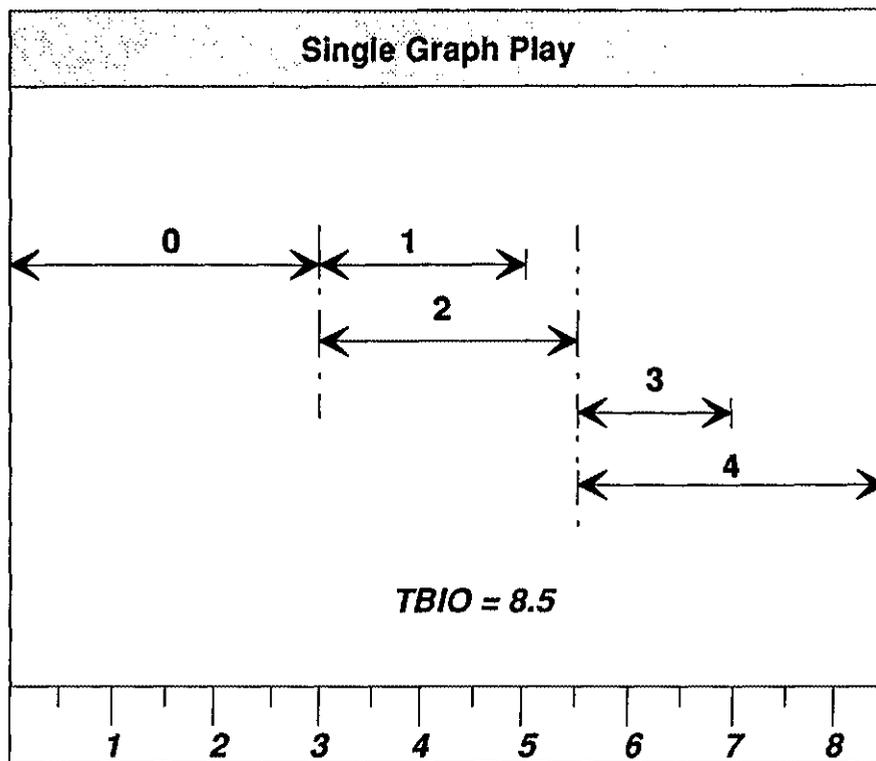


Figure 2.7 SGP Diagram for the AMG in Figure 2.3(b).

Node	*Execution Time
0	3.0
1	2.0
2	2.5
3	1.5
4	3.0

\* In CPU Time Units

Table 2.1 Node Execution Time Units for the AMG in Figure 2.3(b).

However, a different diagram results for each new value of operating period (TBO). The TGP diagram is drawn using information from contiguous and overlapped SGP frames. The steady state SGP is divided into segments of required period, which are overlaid to form the TGP. This process is displayed in Figure 2.8(a). Each segment from the SGP represents a new input data packet. Data Packets are numbered sequentially so that the packet numbered  $i$  is input to the graph TBO time units after the packet numbered  $i-1$ . To illustrate these concepts, the TGP for the AMG in Figure 2.3(b) is drawn in Figure 2.8(b).

### 2.3 Implementation of the ATAMM Model

Given a decomposed decision free macro dataflow algorithm with execution times for each node accurately specified, and a targeted multicomputer architecture, the ATAMM model prescribes performance criteria for algorithm execution based on the number of available Processing Elements  $R$ , and amount of pipeline and/or parallel concurrency desired. When sufficient resources are available, an ATAMM dataflow computer executes dataflow algorithms with maximum throughput and minimum computing time. When only limited resources are available due to resource starvation or lack of resources, the graceful degradation in performance is predicted by the ATAMM model.

Thus the key benefit of the offline modelling process is ATAMM's ability to predict the execution behavior of a dataflow algorithm, given the tradeoffs between decreasing throughput or increasing computing time and  $R$ . Therefore, an ATAMM dataflow computer can predictably alter the execution pattern in real time, based on different sets of values for throughput, execution time and  $R$ .

Given the above nature of multicomputer performance, we can identify the key elements of an ATAMM Large grain Dataflow Multicomputing system. They are:

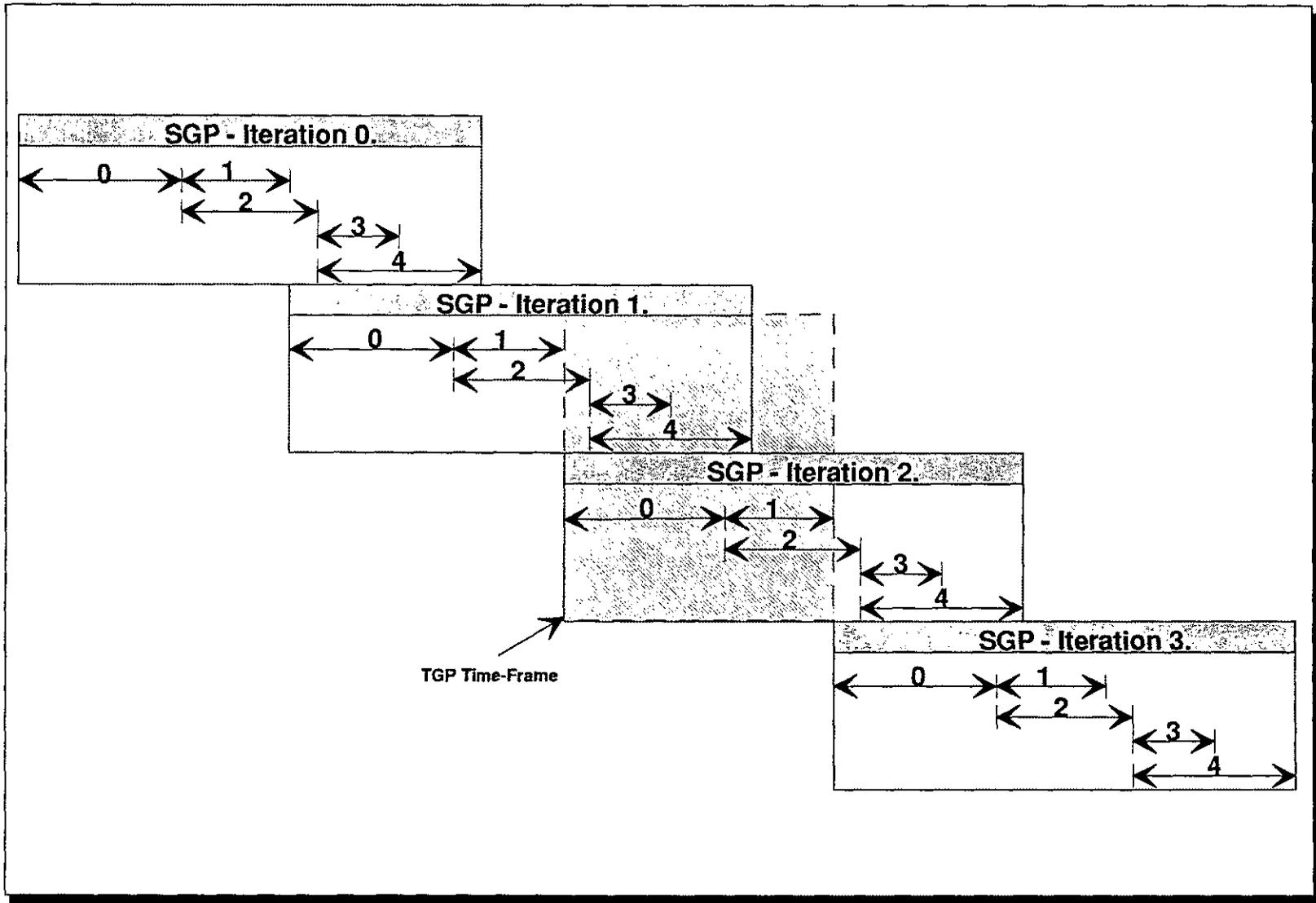
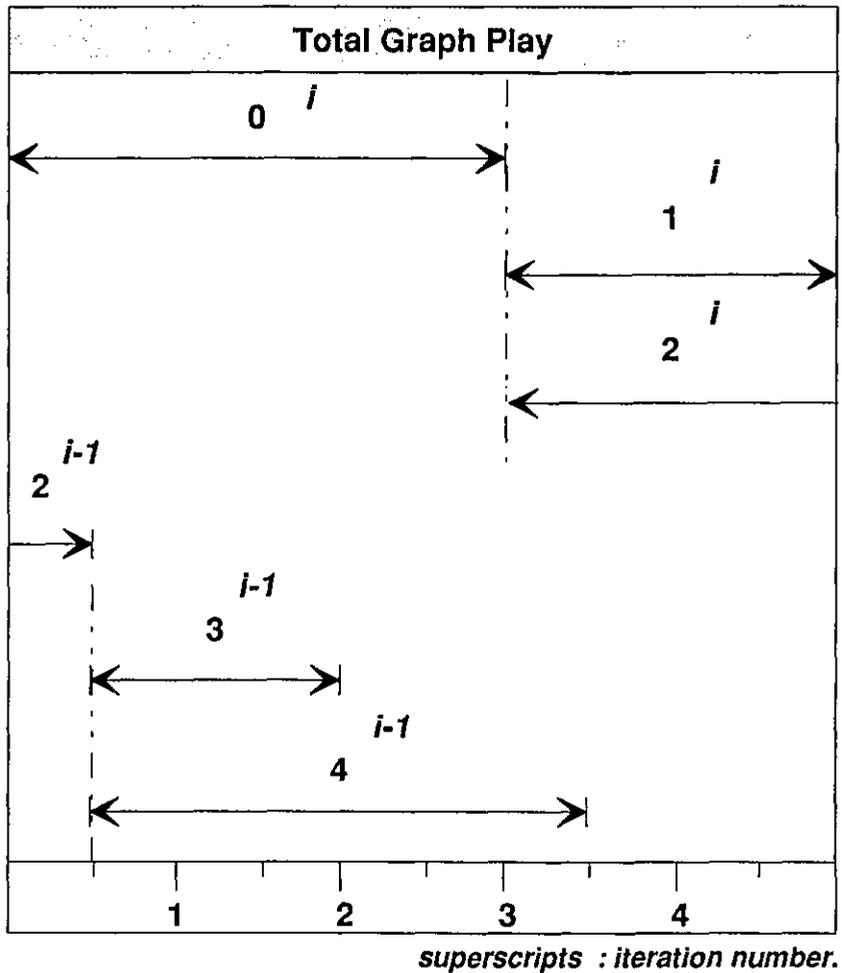


Figure 2.8(a) Overlapped SGP Frames for the AMG in Figure 2.3(b).



$$\text{TCE} = 3.0 + 2.0 + 2.5 + 1.5 + 3.0 = 12.$$

$$\text{TBO} = 5.0.$$

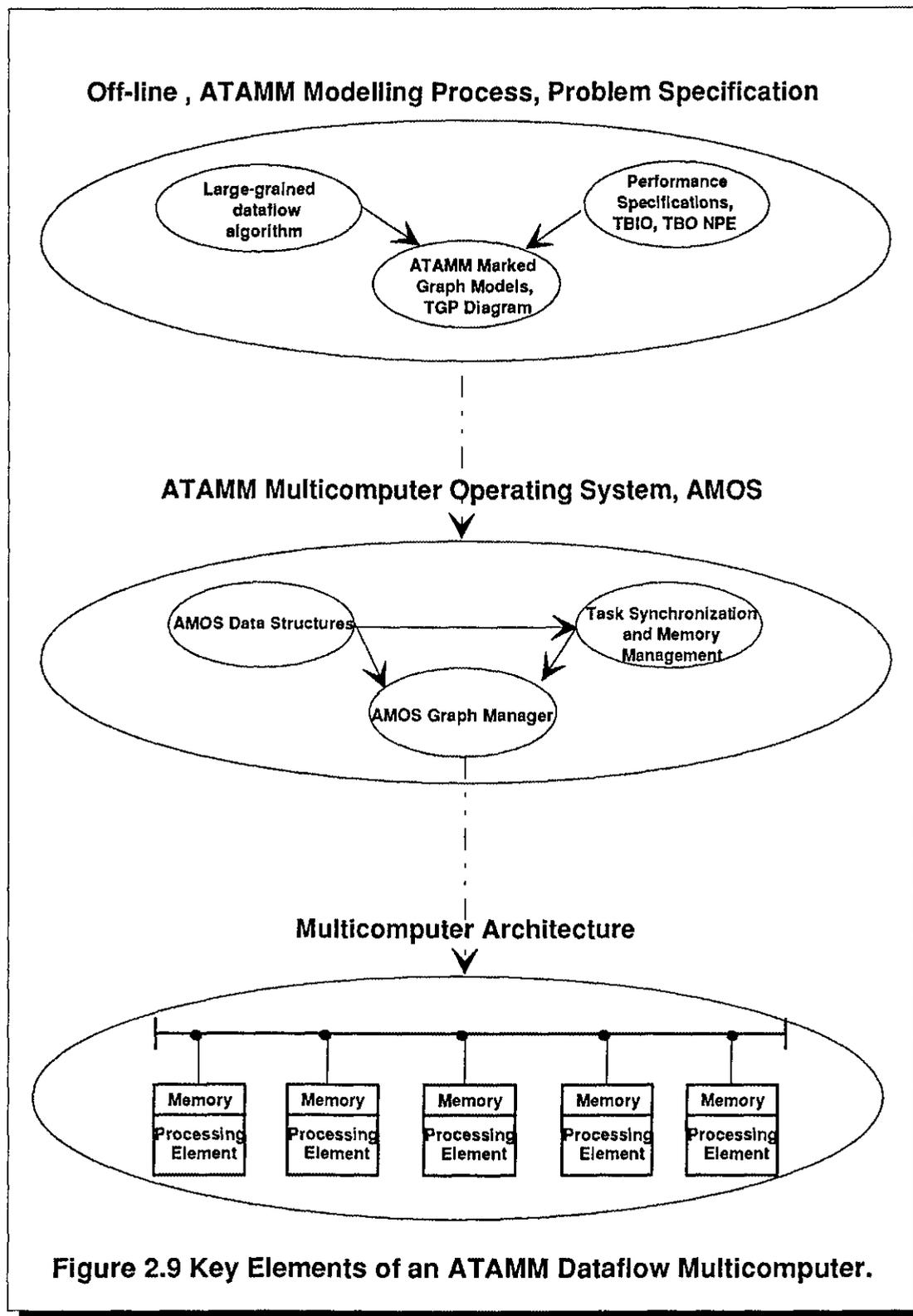
Figure 2.8(b) TGP Diagram for the AMG in Figure 2.3(b).

- [1] A decomposed, decision free, large grain dataflow algorithm (with functionally independent node code);
- [2] Performance specifications for desired throughput, execution time and R, as stipulated by ATAMM, in terms of marked graph models and the TGP diagram;
- [3] ATAMM Multicomputer Operating System, AMOS, with the following components,
  - [3.1] Data structures to represent data and control dependencies, node execution times, R and other execution parameters as translated from the performance specifications derived offline;
  - [3.2] A centralized or distributed graph manager (task scheduler), which schedules and performs NMG activities for each node in the graph; and
- [4] A multicomputer architecture, with functionally identical Processing Elements (PEs); (Each PE would be required to possess local memory for storing code and data and a multicomputer message passing structure for transmitting and receiving control and data information).

The interaction between these elements of an ATAMM dataflow computer has been shown graphically in Figure 2.9.

The next element of an ATAMM dataflow computer, graph and execution criteria, was subject to discussion in the previous Section. These criteria are generated as a result of the ATAMM modelling process. Numerous software tools have been developed to predict, simulate and analyze ATAMM system behavior, given tradeoffs between throughput, execution time and R.

The ATAMM Multicomputer Operating System AMOS, is a logical interface between the dataflow graph and the multicomputer hardware. The main components of AMOS include a variety of data structures which translate graph and execution parameters into an AMOS readable form, a graph manager, and a system task scheduler.



AMOS data structures represent the computational problem and the specific dataflow execution paradigm to be satisfied. Some of these structures are a graph connection matrix representing the AMG's data connections, information pertaining to buffer lengths and iteration increments along control and data arcs, initialization information etc. The specifics of these logical data units for the ATAMM testbed are presented in Chapter Three.

The AMOS Graph Manager (AGM) performs the task scheduling operations of the operating system. The AGM may be classified as centralized or distributed, based on how CMG information is used during graph play. The taxonomy pertains to either having a single task master, which concerns itself with a composite view of the CMG at every point of execution, or having an ensemble of logically coherent AGM pieces which are responsible only for executing sub-partitions of the CMG. This essentially implies that a centralized graph manager (CAGM) exists as a monolithic task scheduler and takes part in performing all node activities as required to play the CMG. A Distributed AMOS Graph Manager (DAGM), on the other hand, is partitioned into unique but logically contiguous fragments. Each fragment monitors and executes a unique partition of the CMG.

The sufficiency conditions to ensure distributed graph management are established in Chapter Three. The remaining portions of this Section are devoted to studying the operation of the CAGM.

### 2.3.1 Centralized AMOS Graph Manager, CAGM

The logical components of a CAGM are a PE queue, a view of the global memory (distributed and shared), AMOS data structures and the algorithm for the CAGM itself. The graph manager uses status information communicated to it by the PEs to update the marking of the CMG. For each node in the AMG, the CAGM begins by examining the global memory for the presence of all required control and data files as necessary to fire the node. Once the CAGM determines the presence of enabled nodes, it endeavors to assign PEs (according to priority if more than one node is enabled) from a queue of available PEs [STOUGHTON88].

A state diagram description of the CAGM is shown in Figure 2.10. It is composed of seven major states, IDLE, EXAMINE CMG, FIRE, EXECUTE, DONE/DATA, SELF TEST and UPDATE. A PE initially starts in the IDLE state. It remains idle until it finds its processor identification number, PID, at the top of the resource queue which is organized in a first in, first out fashion. Upon finding its PID assigned, the functional unit progresses to the EXAMINE CMG state and stays there until it locates an enabled node. Once a node is found, the functional unit, progresses to the FIRE state, where it removes its PID from the top of the resource queue, updates the CMG, reads the input data and broadcasts a "F" (fire) command to the other PEs. The "F" command contains the updated version of the CMG, the updated resource queue, and the ID of the functional unit processing the AMG node. This broadcast along with others discussed next, provides the status information necessary for the graph manager to maintain the logical integrity of the CMG across all PEs.

The PE progresses to the EXECUTE state after the broadcast and remains there until processing of the node is complete. Thereafter, the PE migrates to the DONE/DATA state, where it writes the output data to global memory, updates the CMG, and broadcasts the "D" (done/data) command.

The "D" command provides the updated CMG and the data generated by the node operations to the other functional units. Before returning to the idle state, the PE enters a SELF TEST state where it checks for system failures and determines its availability for future assignments. This state provides the means to remove a functional unit from the system for inspection during real time operation. If the PE emerges successfully from the test state, it will place its PID at the bottom of the resource queue and broadcast the "R" (resource) command containing the updated resource queue to the other PEs.

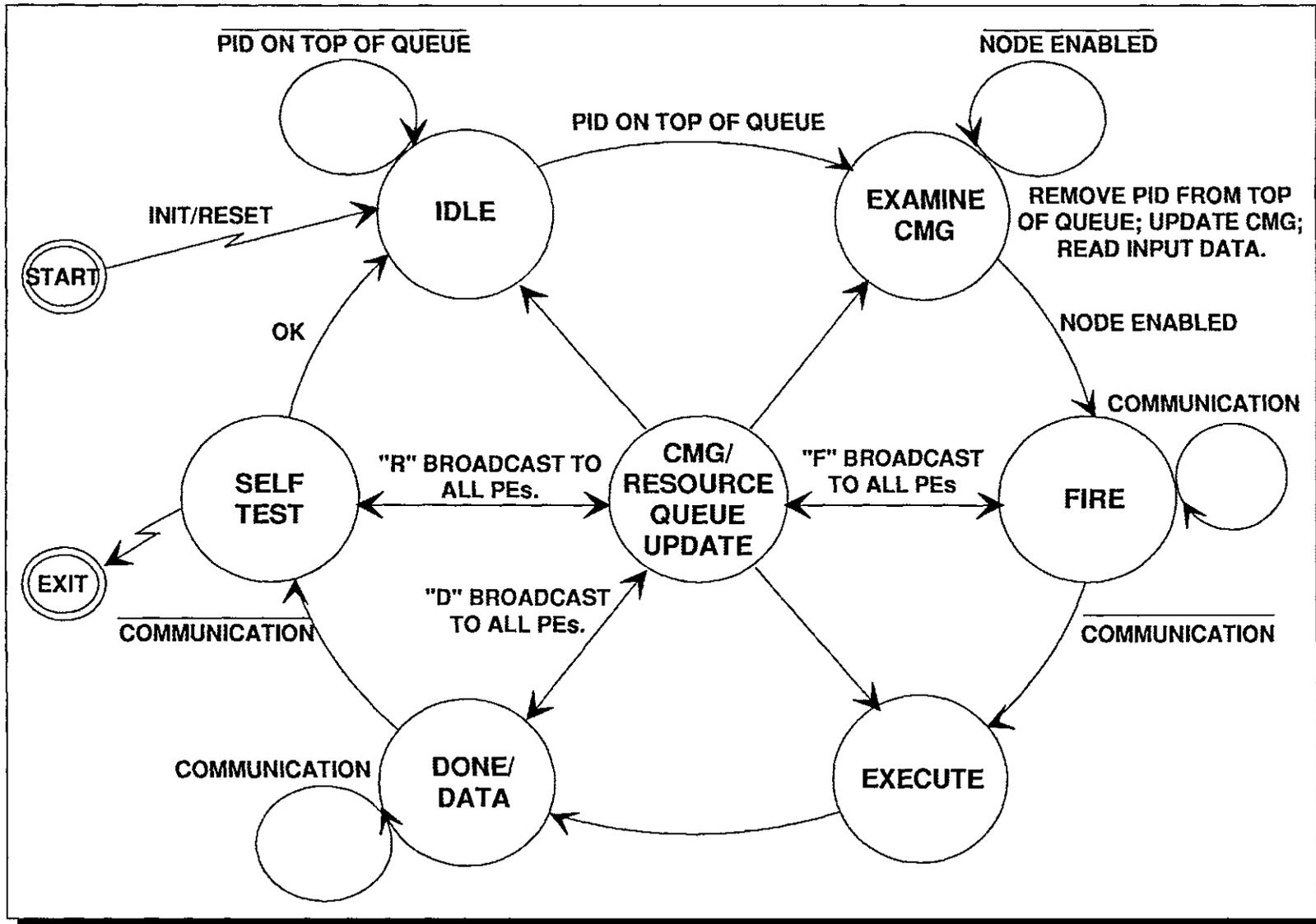


Figure 2.10 Enhanced View of the Centralized AMOS Graph Manager.

The asynchronous nature of the centralized graph management process necessitates the CAGM to be interrupt driven. When a broadcast is received, a functional unit will be interrupted and would be required to enter the update state. It remains in this state long enough to update the CMG, global data and the resource queue.

The chief advantage of the centralized graph manager is its ability to preserve a unique execution view across the entire resource spectrum. The regularity of the underlying state model allows PEs to repeatedly perform AMOS or node related activities and make themselves available for future tasks.

However, the CAGM with a dynamic scheduler also poses certain disadvantages.

These are:

- [1] The CAGM introduces a significant amount of communication overhead. The requirement of generating "F", "D" and "R" commands continually for each node enablement, adds to the existing communication overhead of transmitting computed data.
- [2] The nature of the CAGM's operations constrains all communication to be directed across all PEs. Redundant distribution of tokens profoundly aggravates the communication costs of the multicomputing system.
- [3] The above communication overhead is related to the overhead of dynamic scheduling. For deterministic dataflow algorithms, compile-time or static scheduling can reduce this overhead.

## **2.4 Performance Analysis for the ATAMM Modelling Process**

Time is the measure of computer performance. Performance can be measured as either throughput or response time. The total amount of work done in a given time is termed as throughput. On the other hand, the time between the start and the completion of an event is termed as response time or execution time [HENNESSEY90].

In the ATAMM environment, the execution patterns of a given CMG can be evaluated by using equivalent performance measurements, Time Between Outputs (TBO) and Time Between Input and Output (TBIO). TBO is a measure of the time interval between algorithm outputs and its inverse indicates throughput. Therefore, TBO reflects the amount of pipeline concurrency realizable. TBIO is an indicator of computing speed (execution time) since it shows the amount of parallel concurrency attainable.

The process of algorithm decomposition imposes bounds on the amount of parallel concurrency and pipeline concurrency that can be derived from a given dataflow problem [SOM89]. If sufficient computing resources are available, an attempt is made to execute dataflow algorithms with minimum TBO and minimum TBIO. Lower bound values for TBO and TBIO can be computed using the CMG.

The graph theoretical lower bound for TBO,  $TBO_{LB}$  is a parameter indicating how quickly CMG iterations can be repeated periodically. It is the shortest time possible between successive outputs. Let  $C_i$  be the  $i^{\text{th}}$  directed circuit in the CMG and  $T(C_i)$  denote the total path time associated with  $C_i$ . Also let  $M(C_i)$  denote the number of tokens contained in  $C_i$ . Then  $TBO_{LB}$  is defined as

$$TBO_{LB} = \text{Max}[ T(C_i)/M(C_i) ] \quad (2.1)$$

where the maximum is taken over all circuits in the CMG [MIELKE88].  $TBO_{LB}$  is thus the largest time per token of all CMG circuits. The CMG circuits that determine  $TBO_{LB}$  are called critical circuits.

The graph theoretical lower bound for TBIO,  $TBIO_{LB}$  can be determined from the AMG by determining the longest path between the input source and the output sink. Formally, if  $P_i$  is the  $i^{\text{th}}$  directed path in the AMG and  $T(P_i)$  the total path time associated with  $P_i$ ,  $TBIO_{LB}$  over all paths in the AMG is then defined as

$$TBIO_{LB} = \text{Max}[ T(P_i) ] \quad (2.2)$$

The performance bounds established above are ideal values. The graph theoretical values for TBO and TBIO need to be modified in order to accurately reflect the execution constraints imposed by assignment, scheduling, execution and inter processor communication operations of AMOS.

In [JONES90] it is shown that for a non recursive AMG the minimum TBO can be expressed as the summation of activity times required for numerous tasks of AMOS and graph play. In the AMG, a critical circuit establishes  $TBO_{LB}$ , which associates itself with processing and token reading/writing operations only. However, within a TBO interval, all transitions within the critical circuit must be fired. There are certain overheads associated with AMOS operation that are introduced during these firing processes. Consequently a more realistic picture for  $TBO_{LB}$  is a minimum TBO time,  $TBO_{min}$ . An example of the list of elements that form  $TBO_{min}$  are,

$$\begin{aligned}
 TBO_{min} = & \quad \text{Time to Fire a node} \\
 & + \quad \text{Time to complete operations related to Done/Data} \\
 & + \quad \text{Time to broadcast interrupts during the TBO interval} \\
 & + \quad \text{Time to recover from F,D,R interrupts from other PEs} \\
 & + \quad \text{Time to recover from the R broadcast of preceding PE} \\
 & + \quad \text{Time to recover from the F,D broadcasts of successor} \\
 & \quad \text{PE.} \qquad \qquad \qquad (2.3)
 \end{aligned}$$

Similarly, a more accurate figure for  $TBIO_{LB}$  may be arrived at by considering all node and AMOS activities, as necessary to execute the nodes present in the longest path from source to sink, in the AMG.

## 2.5 Time Measurements for the ATAMM Model

In real time applications it is of paramount importance to satisfy the opposing requirements of maintaining a high throughput (low TBO) while sustaining maximum computing speed (low TBIO). The ATAMM model provides the means to match algorithm requirements with resource availability for achieving a balance between TBIO and TBO and also establishes the criteria for predictable

performance. Predictability is attained by maintaining an input injection rate within the range determined by ATAMM [MIELKE90].

The resource requirements to execute a single data packet (i.e an instantiation of the CMG for a particular iteration) are obtained by counting the number of concurrently active nodes in TGP diagrams, drawn for different values of TBO. The peak resource requirement, denoted by  $R_{max}$ , represents the maximum number of resources necessary to execute the graph with  $TBIO = TBIO_{LB}$  and  $TBO = TBO_{LB}$ . However, for insufficient resources, performance cannot reach the bounds established by  $TBO_{LB}$  and  $TBIO_{LB}$ . Consequently, the resource requirement would be different from  $R_{max}$ , if an algorithm were to be operated with measures other than  $TBIO_{LB}$  and  $TBO_{LB}$ . In this Section, the characteristics of resource usage, maximum resource requirement and resource imposed performance bounds are summarized. An analysis of resource requirements would require familiarity with ATAMM terminology. Consequently, relevant theory from [SOM89] is presented here.

Performance metrics for algorithm execution are time measures that characterize various aspects of run time behavior. A unit of computer time is indicated by the availability of a PE over one unit of execution time. For instance, the use of four PEs over ten units of execution time indicates 40 units of computer time. Computing Capacity, CC, is the total available units of computer time over an interval of time T. If R resources were used over T, the Computing Capacity is  $R * T$  units. In other words, CC is the total units of computer time available over a specified time interval. Correspondingly, Computing Effort, CE, is defined as the total units of computer time used over the interval T. The Total Computing Effort, TCE, is denoted by the total units of computing effort required to execute all AMG transitions once.

These definitions can be used now to state an important theorem that concerns node scheduling and PE assignment strategies for AMOS [SOM89].

Theorem : Minimum TBO for R Resources.

The minimum value of TBO for an algorithm marked graph operated periodically with R resources is always greater than, or equal to,  $TCE/R$ .

Expressing the above in an equation form,

$$[TBO] \geq [TCE/R]. \quad (2.3)$$

This can be restated as,

$$[R \text{ Processors}] * [TBO] \geq [TCE * 1 \text{ processor}]. \quad (2.4)$$

In other words, Computing Capacity (expressed in processor time units as  $R * TBO$ ) equals or exceeds the Total Computing Effort exerted by a single processor. The expression  $[TCE * 1 \text{ processor}]$  represents the aggregate time span that a processor requires to discharge TCE units of work. Moreover, after completing a node, a processor may be required to wait for a while before it could assign itself to another node. Within a TBO time interval, the aggregate time a processor waits in order to be assigned is termed  $T_{\text{wait}}$ . Based on these observations, according to [STOUGHTON93], the inequality in Equation 2.4 can be removed by restating that, for one processor,

$$[\text{Computing Capacity}] = [\text{Total Computing Effort} + \text{Waiting Effort}] \quad (2.5)$$

In other words,

$$[R \text{ Processors} * TBO] = [TCE + T_{\text{wait}}] * 1 \text{ Processor}. \quad (2.6)$$

It shall be shown in Chapter Three that Equation 2.6 establishes conditions that can be used to validate processor assignment strategies.

## 2.6 Summary

The objective of this chapter is to establish an adequate theoretical background for supporting the concepts developed in subsequent chapters. This goal is achieved by providing a succinct summary of relevant research exerted previously under the ATAMM project.

Salient features of dataflow computing are reviewed initially. The modelling ideas of the ATAMM paradigm appear next. The main elements of an ATAMM based multicomputing system are addressed subsequently. This discussion is succeeded by a brief description of the logical components of the Centralized AMOS Graph Manager and its operational principles. Relevant terminology, theorems and equations which aid in performing an analysis of execution behavior are presented in conclusion.

The ATAMM model (based on Centralized AMOS Graph Management) has been implemented on a four processor VHSIC computer based on the MIL-STD-1750A instruction set architecture. The architecture, called the Advanced Development Model (ADM), has been constructed by the Westinghouse Electric Corporation. ADM's implementation of the AMOS Graph Manager is based on centralized task scheduling (CAGM). A detailed description of ADM's features can be found in [SOM93]. Enhancements to the ATAMM model have been incorporated in the Generic VHSIC Spaceborne Computer [MIELKE90].

## **CHAPTER THREE**

### **A HYPOTHESIS FOR CYCLO-STATIC SCHEDULING AND ITS IMPLEMENTATION ON A LOCAL AREA MULTICOMPUTING TESTBED.**

#### **3.0 Introduction**

The conceptualization of a cyclo-static scheduling strategy for the ATAMM Multicomputer Operating System, and its implementation on an ATAMM testbed consisting of networked personal computers, are presented in this chapter. The transition from ATAMM's theoretical framework to a physical testbed is based upon a creative interpretation of modeling concepts and criteria for performance analysis. Numerous observations and conclusions are deduced from established precepts of ATAMM theory in order to formulate a hypothesis that defines a strategy for cyclo-static scheduling and assignment for an AMOS. A design for the ATAMM Testbed is conceived by identifying system elements and logical structures that are required to support the assertions in the hypothesis. The path of this gradual progression from theory to machine is traced in this chapter.

An examination of the steady state behavior of the AMG and the time relationship between Computing Capacity and Total Computing Effort suggests the presence of periodic node sequences that can be employed to implement distributed operations within an ATAMM Multicomputer Operating System, AMOS [STOUGHTON93]. A study to this effect is conducted and a hypothesis for cyclo-static scheduling is presented in Section 3.1. An actual dataflow algorithm is used here, for illustrative purposes. A discussion on the formulation of a Distributed AMOS Graph Management strategy, and its implications on the ATAMM computing model, is presented in Section 3.2. The resultant state machine view for AMOS is described in Section 3.3.

Features that specifically describe the development and operation of the testbed, are presented in the Sections 3.4 and 3.5. Key functional differences between distributed and centralized AMOS operations are identified in Section 3.6. A summary that reiterates the salient features of the cyclo-static scheduling policy and the ATAMM testbed is presented in Section 3.7.

### 3.1 A Hypothesis for Cyclo-static Scheduling

Several observations are made on AMG performance in steady state, in order to postulate the existence of periodic node execution patterns. It is proposed that one way of distributing node scheduling operations would be to assign processors to mutually exclusive sequences of nodes [STOUGHTON93]. To illustrate these ideas, scheduling alternatives for a five node AMG are discussed.

#### 3.1.1 Possibility of the Existence of Periodic Node Execution Patterns

A strategy for distributed scheduling and assignment may be arrived at by considering the steady state behavior of an  $N$  node AMG. In steady state, AMG execution is characterized by the Total Graph Play diagram. The TGP is an instantiation of the AMG over a TBO time period. Within a TGP frame, each node of the AMG is executed once (for a relative iteration number).

Consider an execution time window for an  $N$  node AMG that consists of a set of  $R$  contiguous TGP frames. The time span of this block of  $R$  TGP frames is  $R * TBO$ . A TGP frame represents the execution schedule of every AMG node. In  $R$  TGP frames, all nodes of the AMG are executed  $R$  times. Consequently, every AMG node is also executed for  $R$  successive iterations, giving rise to an ensemble of node execution traces. Assume the existence of an execution pattern that represents a specific sequence for successively fired AMG nodes. Of interest is the construction of one or more periodic node sequences such that,

- [1] each sequence represents a set of nodes containing exactly one instance of each of the  $N$  nodes of the AMG and therefore illustrates a single execution of every AMG node in a specific order;

- [2] nodes are selected in a time exclusive manner so as to ensure that a node in the sequence is fired only after the completion of the node that precedes it in the sequence;
- [3] an iteration index relationship gets established between successive nodes in the loop which ensures that while migrating from a node to an adjacent one, the iteration index gets incremented by zero or more; and
- [4] a periodic repetition of the sequence is seen across contiguous frames of length  $R * TBO$ , as a result of which, if a node in the sequence is associated with iteration  $i$  in a given frame, it gets associated with iteration  $i+R$  in the next frame.

Due to its periodic behavior, a node sequence represents a chain of nodes that form a *node loop*. This is true since for a given frame, the completion of the last node in the sequence leads to the execution of the first node in the next frame. Analogously, the frame of length  $R * TBO$ , for which the node loop is identified, may be termed as a *loop frame*.

A single loop frame of length  $R * TBO$ , (which may be termed as the *frame of reference*), can be created by overlapping  $R$  successive loop frames, where every loop frame is shifted from its predecessor by one  $TBO$  time unit. Assume that the frame of reference bears a node loop, (which is termed as the *basic node loop*). Consequently, every overlapping frame also contains a *thread* of the basic node loop that is cyclically shifted by integral  $TBO$  units with respect to the basic node loop in the frame of reference. Therefore, when  $R$  loop frames are overlapped to construct a single frame of reference, it contains in addition to the basic node loop,  $R-1$  loops that are phased by integral  $TBO$  units. As a result, the frame of reference consists of  $R$  node loop threads, that are unique, cyclically shifted, mutually exclusive images of the basic node loop.

Each of the  $R$  threads refers to a unique  $N$  node set. The cyclical order of sequencing nodes is preserved in each of these sets. However, the occurrence of a particular node in each node set is associated with a different iteration number.

Consequently though all threads contain the same set of AMG nodes, each combination of node and relative iteration number is unique. For example, the combination of an initial node and an initial iteration number for the node is uniquely defined for each thread. Therefore, each thread begins with a node that is associated with an specific iteration index.

Since a loop frame represents the execution of  $R$  AMGs and a node loop represents one AMG, every node in the frame of reference belongs uniquely to one of the  $R$  threads of the basic node loop. In other words, the threads of a node loop are mutually exclusive and collectively exhaustive. A graphical description of these concepts appears in Figure 3.1.

### 3.1.2 Cyclo-static node scheduling

Assuming the existence of periodic node loops that satisfy the criteria specified above, a hypothesis that leads to a method of distributing assignment and scheduling operations within AMOS can be postulated. The assertions proposed by the hypothesis are:

- [1]  $R$  processors are required to periodically execute the  $R$  threads of a basic node loop. A processor is uniquely assigned to a thread of the node loop. The processor executes its thread by migrating from one node to another in the prescribed sequence and by associating nodes with particular iteration indices. The processor repeats its cycle of execution periodically across consecutive loop frames. Consequently, it preserves a modulo  $R$  relationship between iterations associated with nodes. Each of the other  $R-1$  processors also is assigned uniquely to one of  $R-1$  threads. Linking a processor to a unique thread guarantees that processors execute nodes in a mutually exclusive fashion.

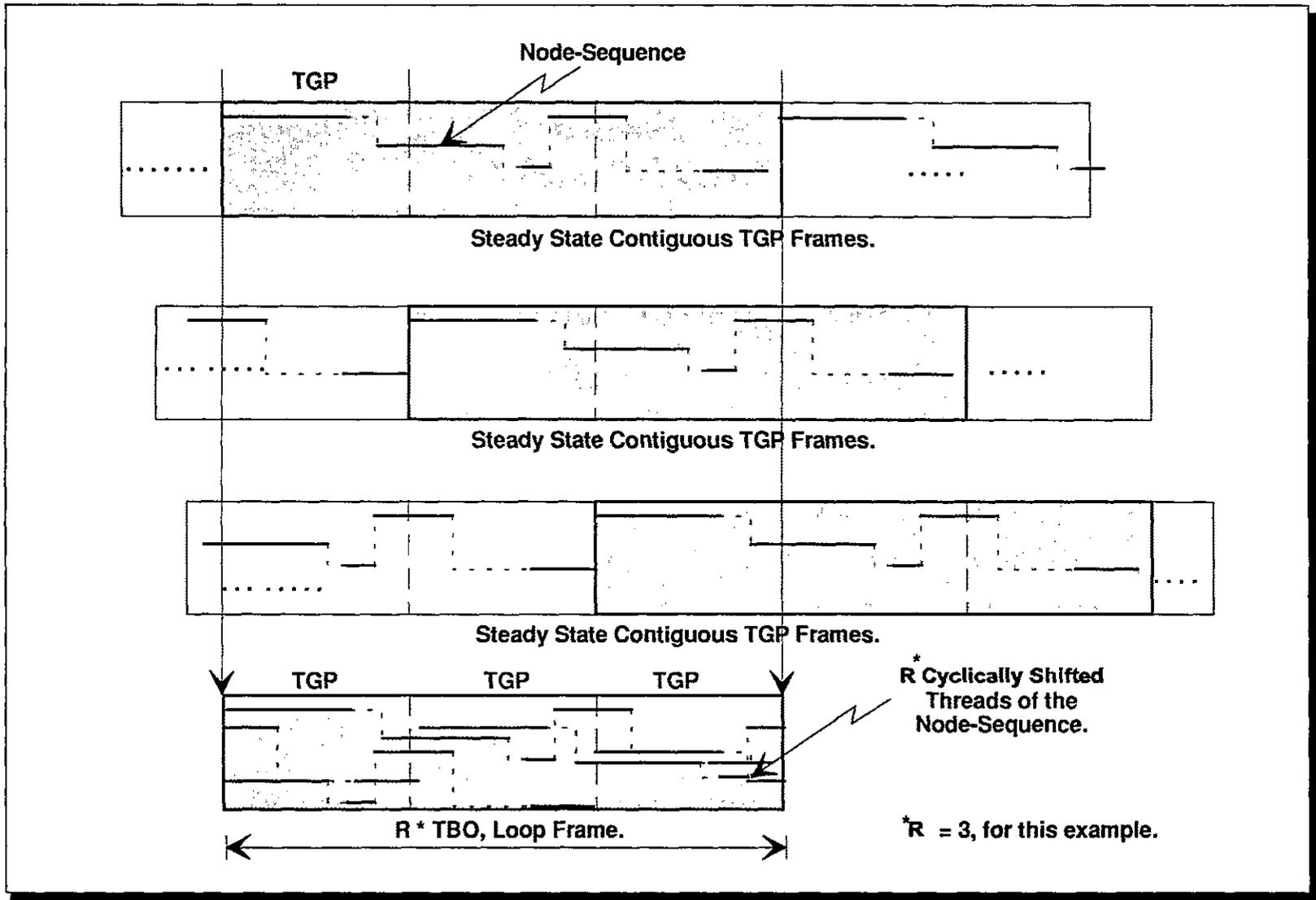


Figure 3.1 Existence of  $R$  Cyclically Shifted Threads of a Node-Sequence in the Loop-Frame.

Furthermore, since every node that appears in a loop frame belongs to an unique thread of the node loop, the association of  $R$  processors with  $R$  threads also establishes the collective exhaustiveness of the assignment process. Such a node loop is said to be fully *cyclo-static*, since processors cycle through its nodes in one loop frame, but repeat the same schedule loop in contiguous loop frames. Thus, a processor assigned to execute a fully cyclo-static schedule is assigned to execute every AMG node once in a loop frame.

The above discussion may be formalized as follows. If the schedule loop for processor  $R_k$ ,  $k \in \{0, R-1\}$ , is periodic in  $R$  TGP frames, then processor  $R_k$  can be successively assigned to all nodes of the AMG, once, with a period of  $R * TBO$ . If resource  $R_k$  is assigned to node  $N_m$ ,  $m \in \{0, N-1\}$ , in TGP frame  $k$ ,  $\forall k \in \{0, R-1\}$ , then, resource  $R_j$ ,  $j \in \{0, R-1\}$ , is assigned to node  $N_m$  in TGP frame  $k \neq j$ . Such a time interlocked relationship for every pair of resources  $\{R_j, R_k\}$  and every node  $N_m$ , for one loop frame, establishes a scheduling loop for each resource in the system. Furthermore, if a node  $N_m$  is executed by processor  $R_k$  in iteration  $i$ , it is executed again by processor  $R_k$  in every iteration that satisfies a modulo( $i, R$ ) relationship.

- [2] A time measure for a cyclo-static assignment can be determined through a restatement of Equation 2.6,

$$[R * TBO] * 1 \text{ Processor} = [TCE + T_{\text{wait}}] * 1 \text{ Processor} \quad (3.1)$$

The Equation states that within a  $R * TBO$  frame, a single processor produces a computational effort that exactly equals TCE. If a processor is assigned to operate on an  $N$  node schedule loop in a  $R * TBO$  frame, it produces TCE units of work since the execution of all  $N$  nodes of the AMG requires TCE effort. In addition, the term  $T_{\text{wait}}$  in Equation 3.1 is the sum of all inter-nodal idle times present in

the cyclo-static schedule loop. In other words, after completing the execution of a node, a processor may be required to wait before it can pick up the next node. The cumulative sum of such idle times determines  $T_{wait}$ . It may be noted that  $T_{wait}$  depends on the specific sequencing of nodes in a loop.

The necessary time measurement condition to be satisfied by every valid cyclo-static schedule loop is also established by Equation 3.1. Each scheduling loop must be able to equate the Computing Capacity ( $R * TBO$ ) required for the AMG to a sum of Computing Effort and cumulative loop wait time ( $T_{wait}$ ), for one processor.

Equation 3.1 may be used in two ways. Given an AMG that requires a computing effort of TCE, a throughput of TBO, and a node sequence that contributes a inter nodal wait time of  $T_{wait}$ ,  $R$  processors are necessary to execute the schedule loop among  $R$  processors in a cyclo-static fashion. Analogously, given TCE, TBO and a maximum resource availability of  $R$ , a cyclo-static schedule loop should be able to provide a  $T_{wait}$  that exactly satisfies equation 3.1

The inherent periodicity of node execution patterns in steady state, creates the opportunity to determine other types of scheduling possibilities that exhibit limited forms of cyclo-static behavior. For instance, given an AMG, it may be possible to define a scheduling policy using a set of schedule loops, each of which contain fewer than  $N$  nodes. In order to satisfy the parallel and pipeline concurrency present in steady state, the set needs to contain two or more mutually exclusive blocks of limited node schedule loops. These blocks impose an implicit partitioning on the AMG. Consequently, a given AMG node can appear in only one block in the set. Two or more processors could be assigned to periodically execute a particular block in a restricted cyclo-static manner. Similar assignments of processors to the remaining blocks of the system ensure the collective exhaustiveness of the assignment process. However, the processors now are scheduled to execute only those nodes that are contained in a block (as opposed to executing every AMG node in a cyclo-static schedule). Furthermore, the iteration

numbers associated with nodes in a block bear a modulo  $R_b$  relationship, where  $R_b$  is the number of processors assigned to execute the block. With this scheduling policy, cyclo static behavior is limited to executing blocks of schedule loops rather than a single  $N$  node schedule loop. Consequently, this form of cyclo-static behavior may be termed as *block cyclo-static*.

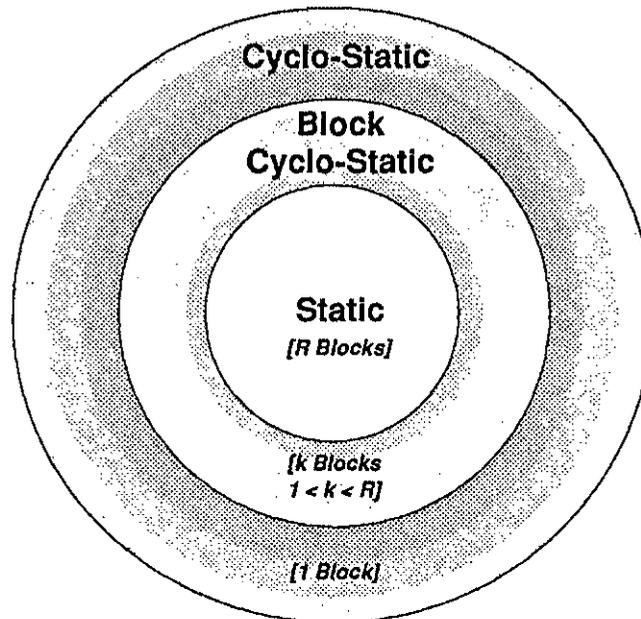
For a given AMG, if  $R$  blocks of schedule loops are created, a new scheduling pattern emerges. In this case each processor becomes solely responsible for a single schedule loop across all iterations. This represents an extreme form of block cyclo-static scheduling, which may be termed as *static*. The process is characterized by the division of an  $N$  node AMG into  $R$  mutually disjoint partitions. In a static schedule, a node is executed by the same processor for every iteration of the AMG. A diagram describing the three forms of deterministic scheduling appears in Figure 3.2. In particular, it should be noted that a fully static schedule consists of a set of  $R$  mutually exclusive node sequences, a block cyclo-static schedule contains  $k$  node sequences (where  $1 < k < R$ ), and a cyclo-static schedule bears only 1 node-loop containing  $N$  nodes.

### 3.1.3 Examples of Cyclo-static Schedule loops

Scheduling examples representing fully cyclo-static, block cyclo-static and fully static schedule loops are presented in Figures 3.3, 3.4 and 3.5 respectively. Schedule loops shown here pertain to the AMG of Figure 2.3 (b), whose TGP appears in Figure 2.8(b). Each of these loops specifies a value of  $T_{wait}$  that requires the utilization of  $R_{max}$  processors.  $R_{max}$  is defined as the number of processors required to operate a TGP with  $TBO = TBO_{LB}$  [SOM88].  $R_{max}$  processors are sufficient to satisfy the parallel and pipeline concurrency of a AMG.

The AMG selected for illustration contains five nodes, has a TCE of 12 and a  $TBO_{LB}$  of 5.  $R_{max}$  under these conditions is three. Of interest is the identification of instances of each type of schedule loops which satisfy Equation 3.1 for  $R = R_{max} = 3$ , thereby warranting a  $T_{wait}$  given by

$$T_{wait} = [R_{max} * TBO] - TCE = [3 * 5] - 12 = 3. \quad (3.2)$$



3.2 (a) Types of Static Scheduling Policies.

For a Cyclo-Static Schedule-loop, a Processor Must Satisfy...

$$[R * TBO] = [TCE * 1 Processor] + [Twait * 1 Processor]$$

Figure 3.2 (b) Time Measurements for a Schedule-loop.

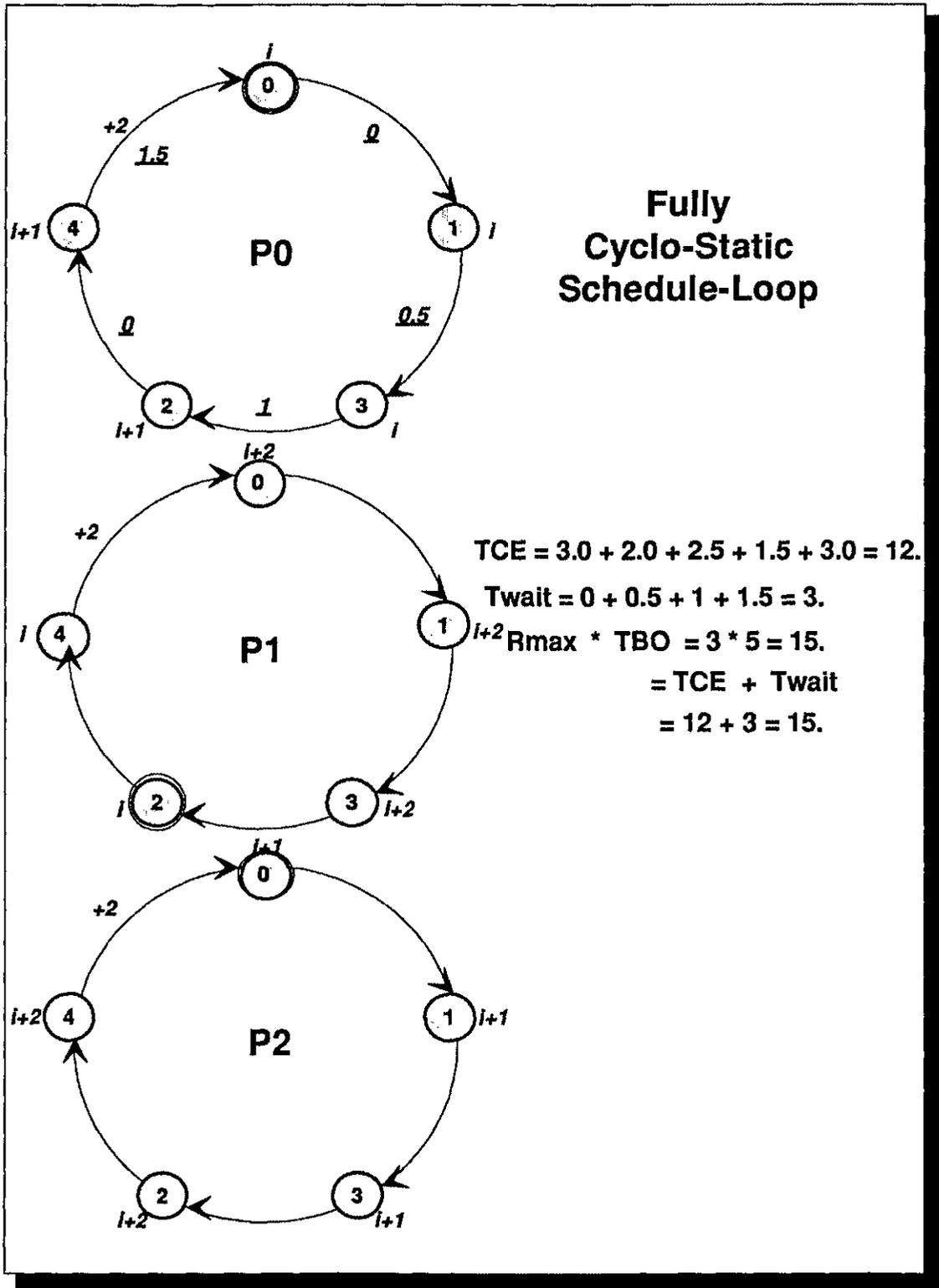


Figure 3.3 Fully Cyclo-Static Schedule-Loop for the AMG in Figure 2.3(b).

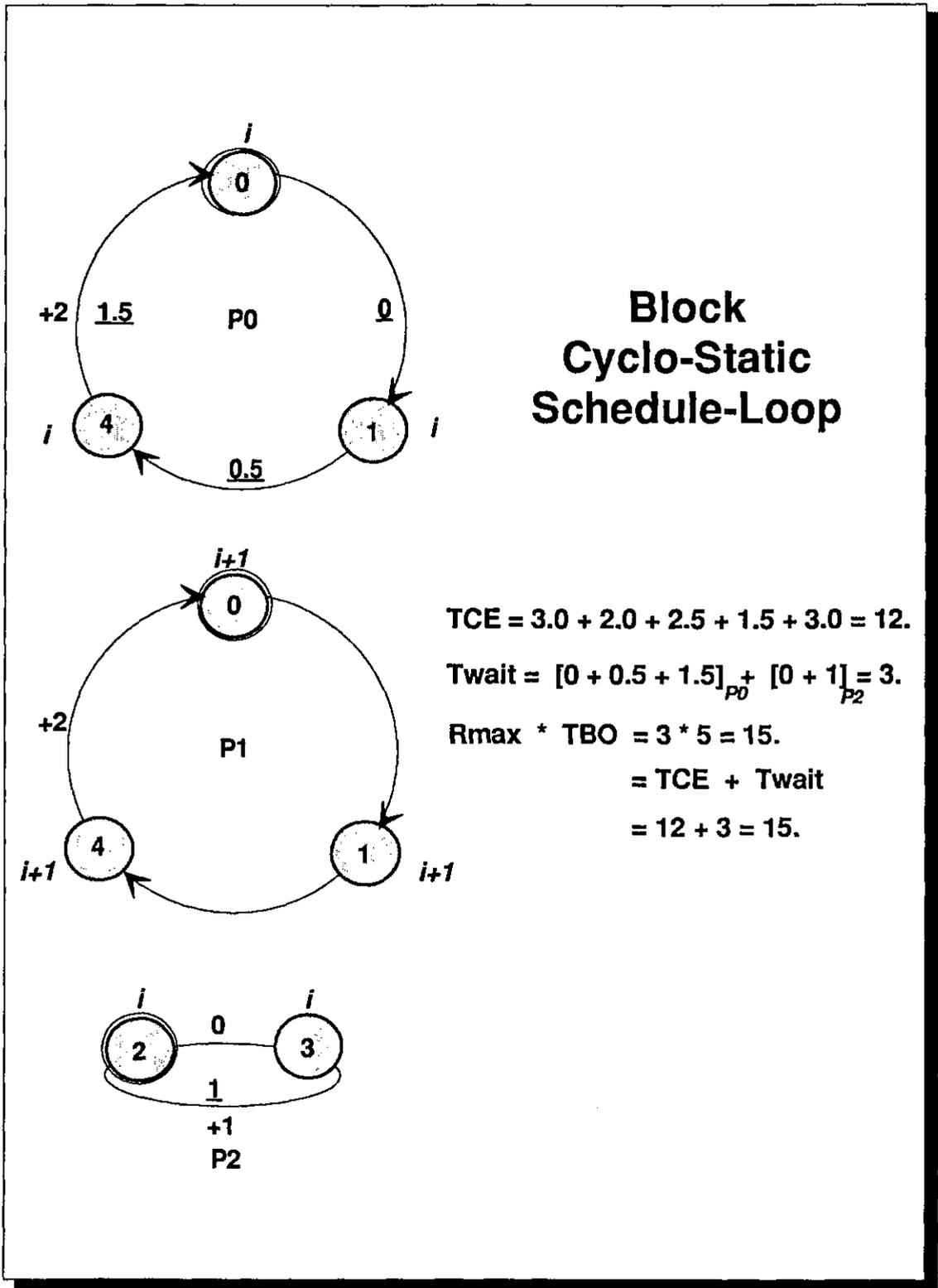
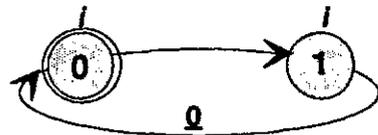
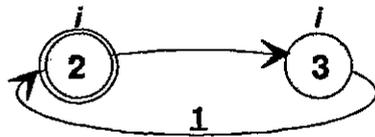


Figure 3.4 Block Cyclo-Static Schedule-Loop for the AMG in Figure 3.2(b).



P0

## Static Schedule-Loop



P1

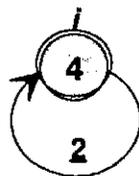
$$\text{TCE} = 3.0 + 2.0 + 2.5 + 1.5 + 3.0 = 12.$$

$$\text{Twait} = [0]_{P0} + [1]_{P1} + [2]_{P2} = 3.$$

$$\text{Rmax} * \text{TBO} = 3 * 5 = 15.$$

$$= \text{TCE} + \text{Twait}$$

$$= 12 + 3 = 15.$$



P0

Figure 3.5 Static Schedule-Loop for the AMG in Figure 2.3(b).

The schedule loops shown in Figures 3.3, 3.4 and 3.5 possess the value for  $T_{wait}$  as shown in Equation 3.2. However, certain schedule loops may have nodes defined in a sequence that introduce additional waiting time for processors. In order to satisfy Equation 3.1 for such loops, more than  $R_{max}$  processors are needed. An example of one such schedule loop, which requires more than  $R_{max}$  processors, is shown in Figure 3.6. The schedule loop requires  $R = 5$  processors. The value of  $T_{wait}$  required for this example is calculated as,

$$T_{wait} = [R * TBO] - TCE = [5 * 5] - 12 = 13. \quad (3.3)$$

Figure 3.6 also demonstrates that it may not be possible to execute an arbitrarily selected schedule loop using  $R_{max}$  processors, if the desired TBO is to be retained. The schedule loop in Figure 3.6, for example, requires five processors.

Every figure contains a "bubble diagram" for an assigned processor. The diagram represents a cyclical schedule loop for the processors. Bubbles represent AMG nodes. Each node in a bubble diagram is associated with an iteration number which is unique with respect to the same node in any other bubble diagram. Note that the transition from a node to its neighbor may also require an iteration number change. Furthermore, each thread of the schedule loop contains a node that is encapsulated by double lined circles. This is the initial node for the processor assigned to execute the thread. Determination of initial nodes is done by considering the steady state behavior of node loop threads within a loop frame. Finally, a comprehensive list of all possible N node schedule loops for the example AMG is presented in Table 2.1 along with associated  $T_{wait}$  and R values.

In sum, in this Section, the existence of periodically executable schedule loops is postulated. If one or more such node sequences can be constructed for a AMG which is to be operated for a given TBO, it is hypothesized that processors can be assigned to execute different threads of the node loop in a mutually exclusive but a collectively exhaustive manner. Every valid assignment satisfies the time measures suggested by Equation 3.1.

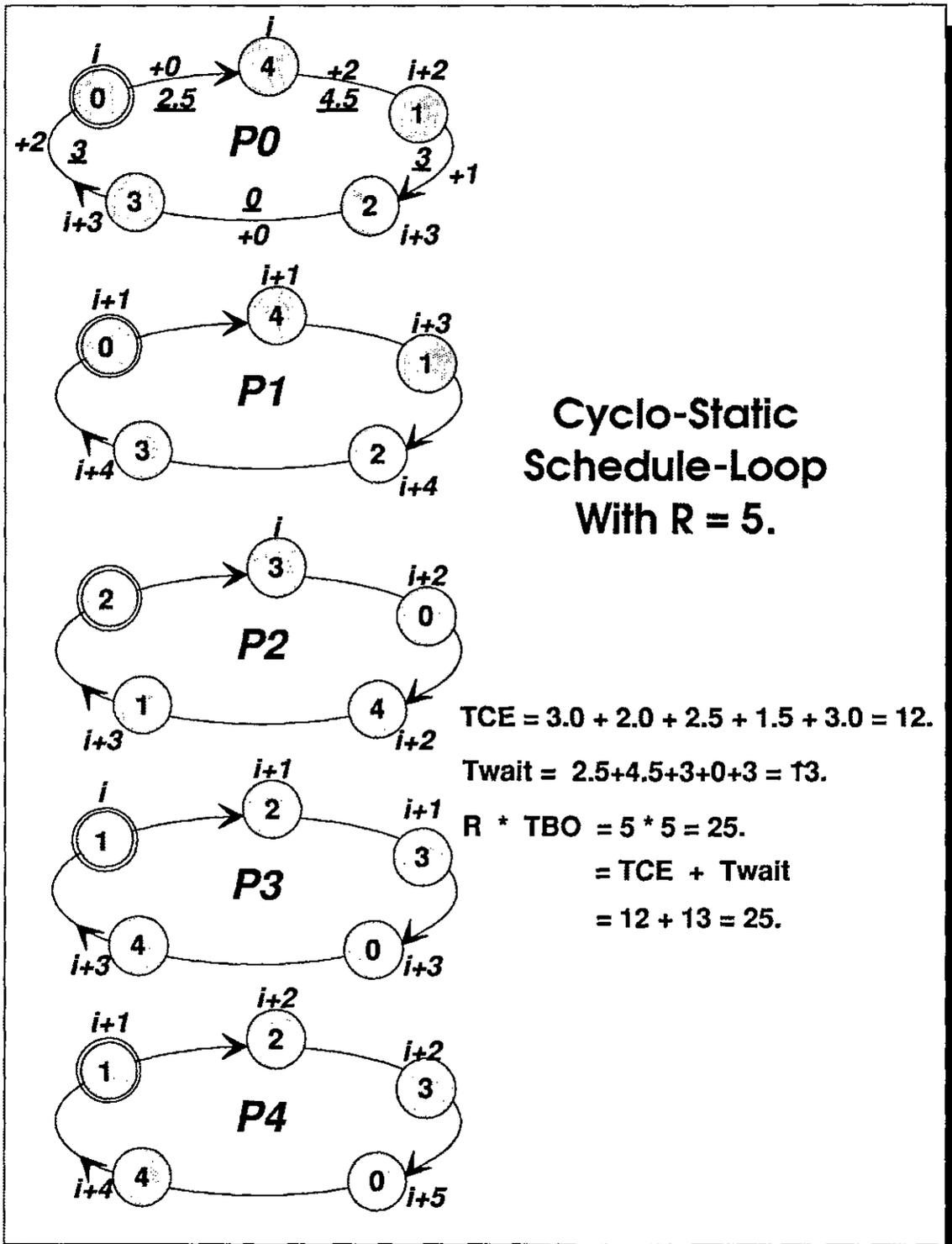


Figure 3.6 Cyclo-Static Schedule-Loop for the AMG in Figure 2.3(b) Which Requires Five Resources.

Table 3.1 Cyclo-Static Schedule Loops for the AMG in Figure 3.2(b).			
No.	Node Sequence	Processors	$T_{wait}$
1.	0 1 3 2 4	3	3
2.	0 2 3 1 4	3	3
3.	0 2 4 3 1	3	3
4.	0 1 2 3 4	4	8
5.	0 1 2 4 3	4	8
6.	0 1 4 2 3	4	8
7.	0 1 4 3 2	4	8
8.	0 2 1 3 4	4	8
9.	0 2 1 4 3	4	8
10.	0 2 4 1 3	4	8
11.	0 3 1 2 4	4	8
12.	0 3 2 1 4	4	8
13.	0 3 2 4 1	4	8
14.	0 4 2 3 1	4	8
15.	0 2 3 4 1	4	8
16.	0 4 3 2 1	4	8
17.	0 1 3 4 2	5	13
18.	0 3 1 4 2	5	13
19.	0 3 4 2 1	5	13
20.	0 4 1 2 3	5	13
21.	0 4 1 3 2	5	13
22.	0 4 3 1 2	5	13
23.	0 4 2 1 3	5	13
24.	0 3 4 1 2	6	18

### 3.2 Design of the Distributed AMOS Graph Manager

Based on the hypothesis for cyclo-static behavior of node loops, a strategy for distributed processor assignment and node scheduling for AMOS is developed in this Section. The key concept that can be transported from the hypothesis to a strategy for distributed AMOS is an information base that correlates processor assignment operations, cyclo-static node schedules and iteration index relationships. The information structure that is required to implement the assertions contained in the hypothesis is generated. The specific manner in which information elements manifest as data structures for distributed AMOS software is explained next. The concepts in this Section use cyclo-static scheduling as a generic case for illustration. However, these ideas could also be applied to block cyclo-static and static scheduling.

#### 3.2.1 Translation of Elements of the Hypothesis Into a Set of Requirements

For a given AMG, a node loop specifies a sequence of nodes that need to be executed chronologically. It also indicates iteration increments associated with node transitions in the loop. Given a schedule loop which fulfils the previously specified criteria for loop selection, it becomes possible to pre-assign a thread of the node loop to each of the  $R$  processors of the system. Aided by the explicit sequencing imposed by the node loop, a processor can continuously re-assign itself for task execution. Attributes of this policy are,

- [1] processors deterministically migrate from one node to another, in the manner established by the pre-determined scheduling policy;
- [2] reference iteration indices are updated while migrating from node to node as specified in the loop sequence; and
- [3] each node loop is repeated periodically, thereby maintaining a unique modulo  $R$  relationship for the iteration indices associated with nodes in the loop.

Given the convenience of pre-determined schedule loops, processors can sequentially schedule nodes for execution, without incurring any run-time

scheduling overhead. The collective but autonomous execution of all node loop threads results in a conflict free execution of the AMG. The fact that a processor is made exclusively responsible for executing a sequence of nodes periodically for specific iterations hints at a strategy for self governed and decentralized scheduling operations for an AMOS graph manager.

When assigned to a node loop, a processor remains in a state of continuous assignment, during which it picks up a node for execution, operates on the node, terminates its execution, and seeks to repeat the cycle on the next node in the schedule loop. However, this view of the assignment policy is only concerned with the process of continuous task execution on a single processor. Consequently, it needs to be augmented with the inclusion of dataflow operations that satisfy the data and control dependencies of the underlying CMG. For example, before firing a node for iteration  $i$ , a processor has to transmit control tokens to predecessor nodes for a future iteration  $i+CB$  (where  $CB$  is the length of the control buffer for a given control arc). Analogously, after completing a node, a processor generates data tokens for successor nodes (which are associated with the present or future iterations).

Based on the above description of node scheduling and maintenance of graph control and data dependencies, the required pieces of information that suffice a strategy for self governed operations can be enumerated. In order to execute a schedule loop and ensure correct dataflow operation, each processor needs:

- [1] a view of the CMG which indicates data and control relationships between nodes;
- [2] a schedule loop which specifies which node to do next;
- [3] relative iteration indices for every node in the loop;
- [4] a starting node associated with an initial iteration number to begin loop execution;
- [5] information which details processor assignments for every node, for  $R$  successive iterations; and

- [6] a modulo coefficient to associate with the repeated execution of each node in the schedule loop.

The first information element relates to a representation of the CMG that indicates data arcs (and therefore control arcs) and special AMG features such as control buffers and forwarded tokens. For every node present in the node loop, an image of the CMG informs a processor of predecessor and successor node relationships.

A schedule loop pertains to a specific sequence of nodes that every processor in the system executes. Since a processor executes a node for only a specific modulo  $R$  iteration number, the nodes of each thread that is assigned to a processor need to be tagged with relative iteration numbers. Threads for a schedule loop differ not only with respect to iteration indices for constituent nodes, but also in the node positions where assigned processors commence with loop execution. Consequently, a processor needs to be aware of a starting node and an initial iteration number for which it begins executing the node.

It was mentioned in Chapter Two that the execution of a node is accompanied by the generation of control tokens for future iterations and of data tokens for present and/or future iterations. However, these tokens must be forwarded to processors which execute the nodes that actually require them. A solution to this problem is obtained at by realizing that due to the mutually exclusive but periodic execution of  $R$  unique node loop threads by  $R$  processors, it becomes feasible to foretell the processors that are assigned to execute a particular node for  $R$  successive iterations. Consequently, if a token is to be generated for a particular node for a specific iteration number, it is possible to target the token to the unique processor that will execute the node. It may be noted that this requirement is directly related to restricting unnecessary token movement via broadcast within a dataflow multicomputer.

For block cyclo-static or fully static scheduling policies, nodes belonging to a particular block are associated with iterations that bear a modulo  $R_b$  relationship, where  $R_b$  is the number of processor assigned to repeat the block in a cyclo-static

manner. Consequently, the modulus coefficient  $R_b$  associated with every node for a block cyclo-static or static schedule needs to be specified. Observe that for cyclo-static schedules, all nodes possess a modulus coefficient that equals  $R$ .

### 3.2.2 Software Implementation of Distributed Graph Management Strategies

In the previous Section, it is seen that in order to implement a processor governed assignment and node scheduling policy, an information structure needs to be specified. Our design for the AMOS Graph Manager represents the required information in the form of simple, tabular data structures. These are:

- [1] a connection matrix, that specifies CMG attributes and the topology of the underlying AMG;
- [2] a scheduling table, that contains the node sequence with information pertaining to relative iteration indices;
- [3] an initialization table that provides every processor in the system with a starting node and an initial iteration number to associate with;
- [4] an assignment table that identifies a processor for a given node and iteration number; and
- [5] a modulo operator table that associates modulus coefficients with nodes in a schedule loop.

The format of each of these structures are explained by constructing each of them for a specific design example. The example used here refers to the AMG portrayed in Figure 3.2(b) and its corresponding CMG in Figure 2.6. The relevant TGP diagram appears in Figure 2.8(b).

The first step in the design process is the construction of a connection (topology) matrix. Assuming eight or fewer nodes, the CMG is represented as an [8 X 8] tabular data structure, which contains information about the presence of data arcs from predecessor nodes (identified by row indices) to successor nodes (identified by column indices). Non-negative entries in the matrix indicate a data relationship between corresponding row and column nodes. A data relationship implicitly also determines the presence of associated CMG control arcs from the

column nodes to row nodes. A transpose of the basic connection matrix yields information about control arcs in the system. Tables 3.2 and 3.3 demonstrate the basic connection matrix and its transpose.

Several other pieces of data augment the information content of the connection matrix. They provide the opportunity to implement a variety of dataflow graph attributes such as forward data tokens, initialization of circuits and establishment of buffers on CMG control arcs. In the current implementation of the testbed, the connection matrix contains three pieces of data, represented as  $(X,Y,Z)$ , for every (row, column) entry. In this taxonomy,  $X$  is an iteration increment for the corresponding data arc. Usually, parameter  $X$  is zero for data tokens that are produced by a node for the current iteration. However,  $X$  is one or more, if a data token is to be sent forward for use in a future iteration. Parameter  $Y$  is the number of initial data tokens needed on a data arc that belongs to a graph circuit. Parameter  $Z$  is a buffer length for the CMG control arc that corresponds to the data arc. For a given row and column, negative values for  $X$ ,  $Y$  and  $Z$  indicate that the corresponding row node has no data relationship with the column node. The augmented connection matrix for our example is shown in Table 3.4. In addition, it should be noted here that source and sink nodes serve no specific purpose in the formation of the connection matrix. Consequently, the testbed software structure treats source and sink nodes as regular AMG nodes. In addition, the software does not incorporate mechanisms for injection control of input data packets.

The cyclo-static node loop shown in Figure 3.3 is used as the basis for the scheduling policy in this example. A data structure termed as the scheduling table is used by processors to schedule a node for execution. The scheduling table contains information pertaining to the organization of the node loop and iteration indices for constituent nodes. The cyclical organization of nodes in the loop is implemented using a present node/next node state table that specifies the next node in the loop for a every node in the AMG. Moreover, as mentioned earlier, nodes in a loop are associated with relative iteration indices. However, while

migrating from one node to another, iteration indices are associated with zero or positive increments. Consequently, it is sufficient to identify the values for iteration increments associated with every node to node transition in the schedule loop. This information is indicated in an iteration increment table. An example of the scheduling table appears in Table 3.5.

Data structures for the remaining information can be easily constructed on the basis of the scheduling table. A processor must be provided with an initial node and an initial iteration to start with. This information is determined while constructing the schedule loop. An example of an initialization table is shown in Table 3.6.

In order to communicate data and control tokens to successor and predecessor nodes for specific iterations, a processor needs to know the processor assignments of these nodes for the required iterations. This information is relayed through an assignment table. For a given node and iteration number, the assignment table contains a processor that executes the node. An assignment table is shown in Table 3.7. Finally we are left with a modulo operator table, which specifies the modulo number associated with every AMG node, that a processor uses to interrogate the assignment table. This is shown in Table 3.8.

### **3.3 AMOS State Diagram and Functional Characteristics**

From the inception of the testbed project, research has been focussed towards building a system that satisfies certain design criteria. A key necessity is the implementation of an AMOS Graph Manager that executes independently on individual processing elements, while ensuring a consistent dataflow operation. Fundamental to this requirement are features such as natural dataflow graph play, elimination of redundant Inter Processor Communication (IPC), autonomous and non-preemptive task execution. The functionality that we needed the testbed AMOS to assume, was shaped by three chief factors:

Table 3.2 Basic Connection Matrix for the AMG in Figure 2.6.								
	0	1	2	3	4	5	6	7
0		0	0					
1	0				0			
2				0	0			
3			0					
4								
5								
6								
7								

Table 3.3 Transpose of Connection Matrix for the AMG in Figure 2.6.								
	0	1	2	3	4	5	6	7
0		0						
1	0							
2	0			0				
3			0					
4		0	0					
5								
6								
7								

[ ] indicates the absence of an arc; 0 indicates an arc from row to column nodes.

	0	1	2	3	4	5	6	7
0		<b>0,0,1</b>	<b>0,0,1</b>					
1	<b>1,1,0</b>				<b>0,0,1</b>			
2				<b>0,0,1</b>	<b>0,0,1</b>			
3			<b>1,1,0</b>					
4								
5								
6								
7								

\* Each (x,y,z) entry indicates (data arc iteration increment, initial data tokens on data arcs, buffer length for control arc).

Pr. Node	Next Node	Iter.Incr.
0	1	0
1	0	+1
2	3	0
3	2	+1
4	4	+1

<b>Table 3.6 Initialization Table</b>		
PID	Init. Node	Init. Iter.
0	0	0
1	2	0
2	4	0

<b>Table 3.7 Assignment Table</b>			
For a given node number and a modulo(iteration, operator) value, this table specifies the processor which shall execute the node.			
Node #	0	1	2
0	0	0	0
1	0	0	0
2	1	1	1
3	1	1	1
4	2	2	2

<b>Table 3.8 Modulo Operator Table</b>	
PID	% Operator
0	3
1	3
2	3

- [1] a requirement of maintaining a natural flow of control and data tokens across the dataflow system, within the ATAMM framework;
- [2] the necessity to ensure for each PE a cyclo-static scheduling policy that features distributed AMOS operations; and
- [3] and the need to implement standard multicomputing operating system features through modules for initialization/synchronization, IPC/memory management, task scheduling and processor assignment.

Design considerations for the testbed, derived from these criteria are presented in the following sub-sections.

### 3.3.1 "FIRE", "EXECUTE" and "DONE/DATA" States

The first design criterion defines a system view which considers token movement related to the communication of primitive control and data tokens. Other than fulfilling the data dependencies outlined by an AMG, token communication also helps in fulfilling the "fireability" conditions for nodes. A node is considered to be enabled, if all requisite control and data tokens are present on the arcs from nodes, with whom the node to be fired has an AMG dependency relationship. The availability of data tokens (from preceeding nodes) and control tokens (from succeeding nodes) is the sole requirement that needs to be satisfied for enabling a node for execution. Once a node is found to be enabled, it informs its predecessor nodes of its "fire commencement" status, by transmitting control tokens to these nodes. Information about CMG data and control dependencies is derived from the connection matrix. In a multicomputer system, this step translates into three sub tasks:

- [1] forming a data or control token entity with appropriate source, destination and iteration number;
- [2] determining processors which are assigned to execute the predecessor nodes for particular a value of a future iteration and

- [3] invoking AMOS IPC functions to physically transmit these control tokens to appropriate destinations (processors).

The functions described above are subsumed within the "FIRE" state in the AMOS state diagram as shown in Figure 3.7. After executing a node, a node is ready to transmit computed data. It repeats the three steps, with respect to transmitting data tokens to successor nodes (after identifying recipients and building appropriate data token headers). These sub tasks are outlined in Figure 3.8.

With the termination of data communication, the overall execution of the current node is concluded. Using a cyclo-static scheduling policy (AMOS data tables), a processor determines a node that it can execute next. This function satisfies the second design criterion of ensuring self determined scheduling policies. Combining the operations of firing, execution, communication and task scheduling results in a state machine view for a distributed AMOS, which is presented in Figure 3.9. A pictorial description of the interaction between AMOS states and the data structures identified in the preceeding Section is portrayed in Figure 3.10.

### 3.3.2 AMOS Characteristics

The third design criterion, i.e. the need to implement AMOS components as generic Multicomputer Operating System (MOS) features, lends structure to the AMOS state machine. Briefly, the key components of a MOS are:

- [1] an *initialization and synchronization* mechanism that ensures orderly, lock stepped execution of all system operations;
- [2] a *memory manager* that manages the usage of local and global memory and ensures code and dataset protection;
- [3] a *communication manager* to handle inter processor communication, IPC, between multicomputer PEs;
- [4] a *task scheduler* that schedules tasks that are ready for execution with available processors in a manner that prevents deadlocks and avoids abnormal program termination, and

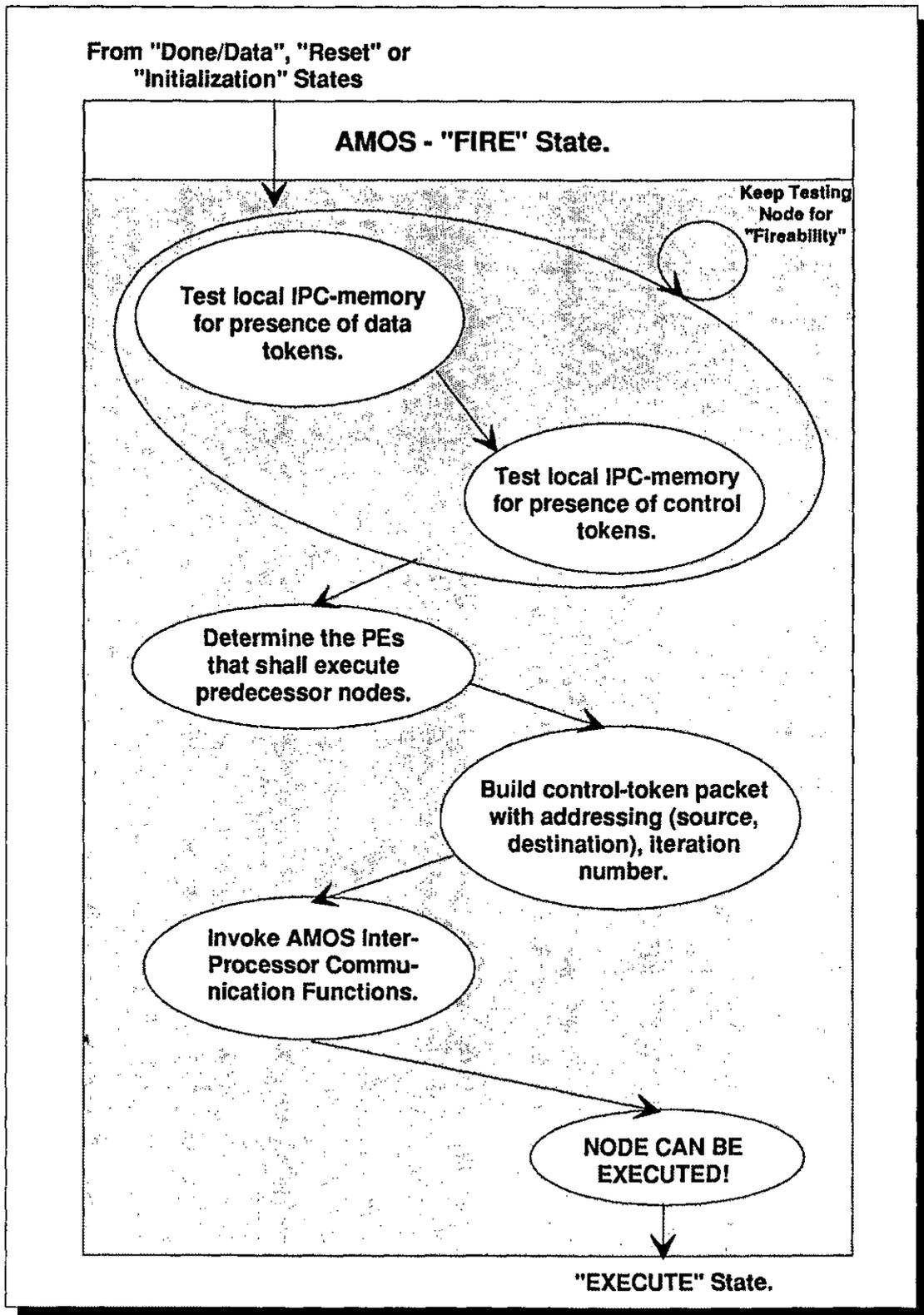


Figure 3.7 Events that Occur in the AMOS "FIRE" State.

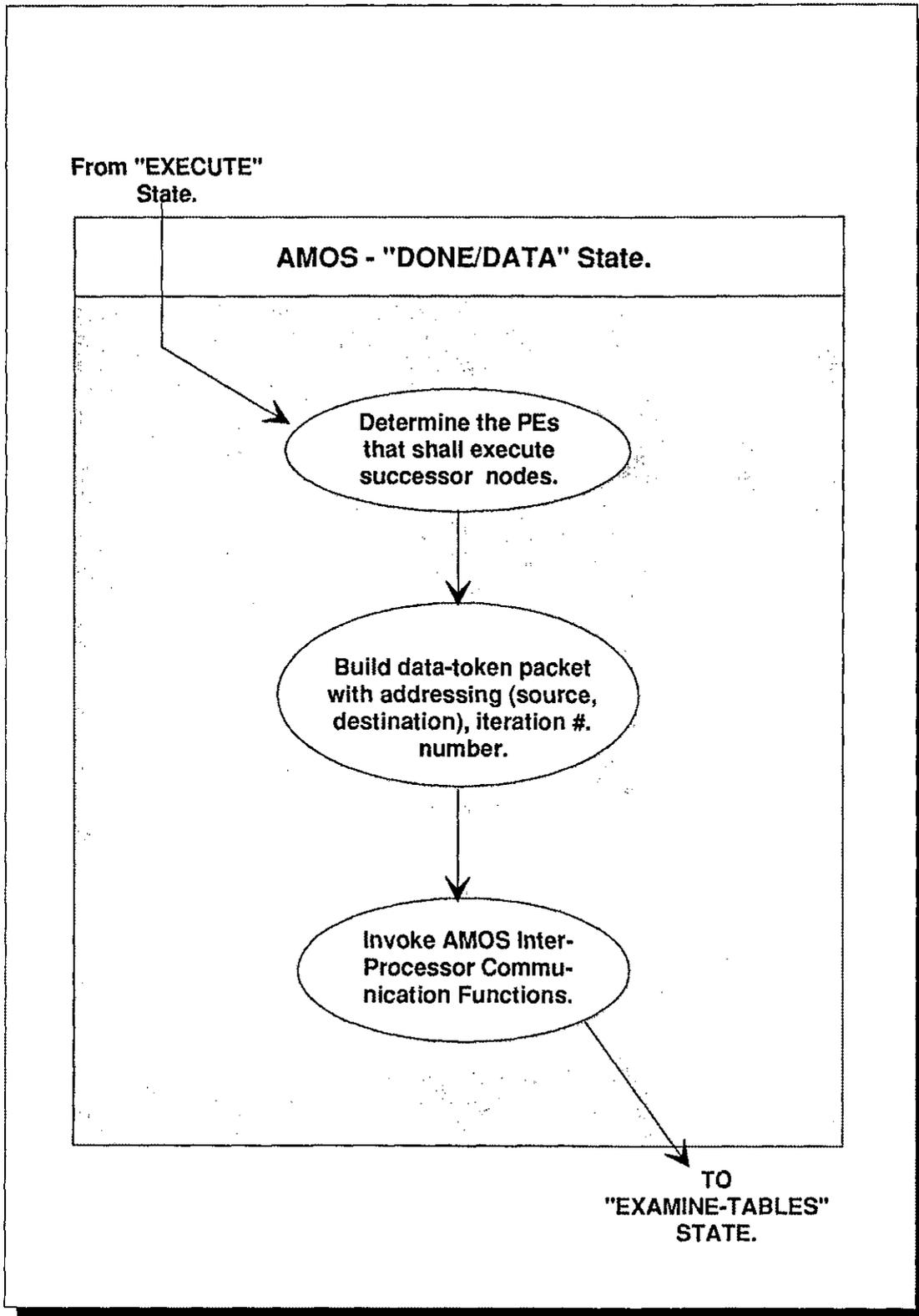


Figure 3.8 Events that Occur in the AMOS "DONE/DATA" State.

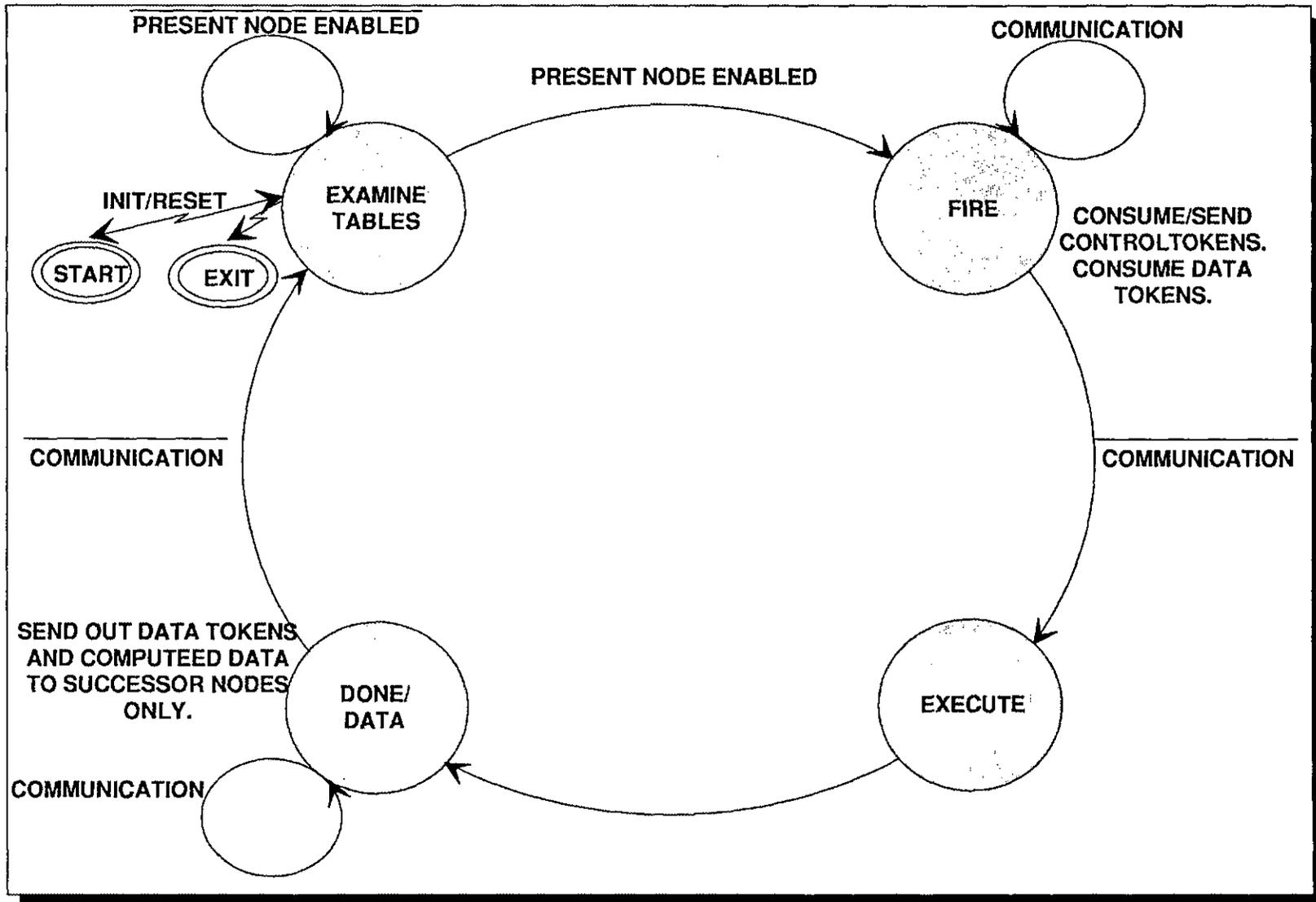


Figure 3.9 State-Machine View of Distributed AMOS Graph Manager.

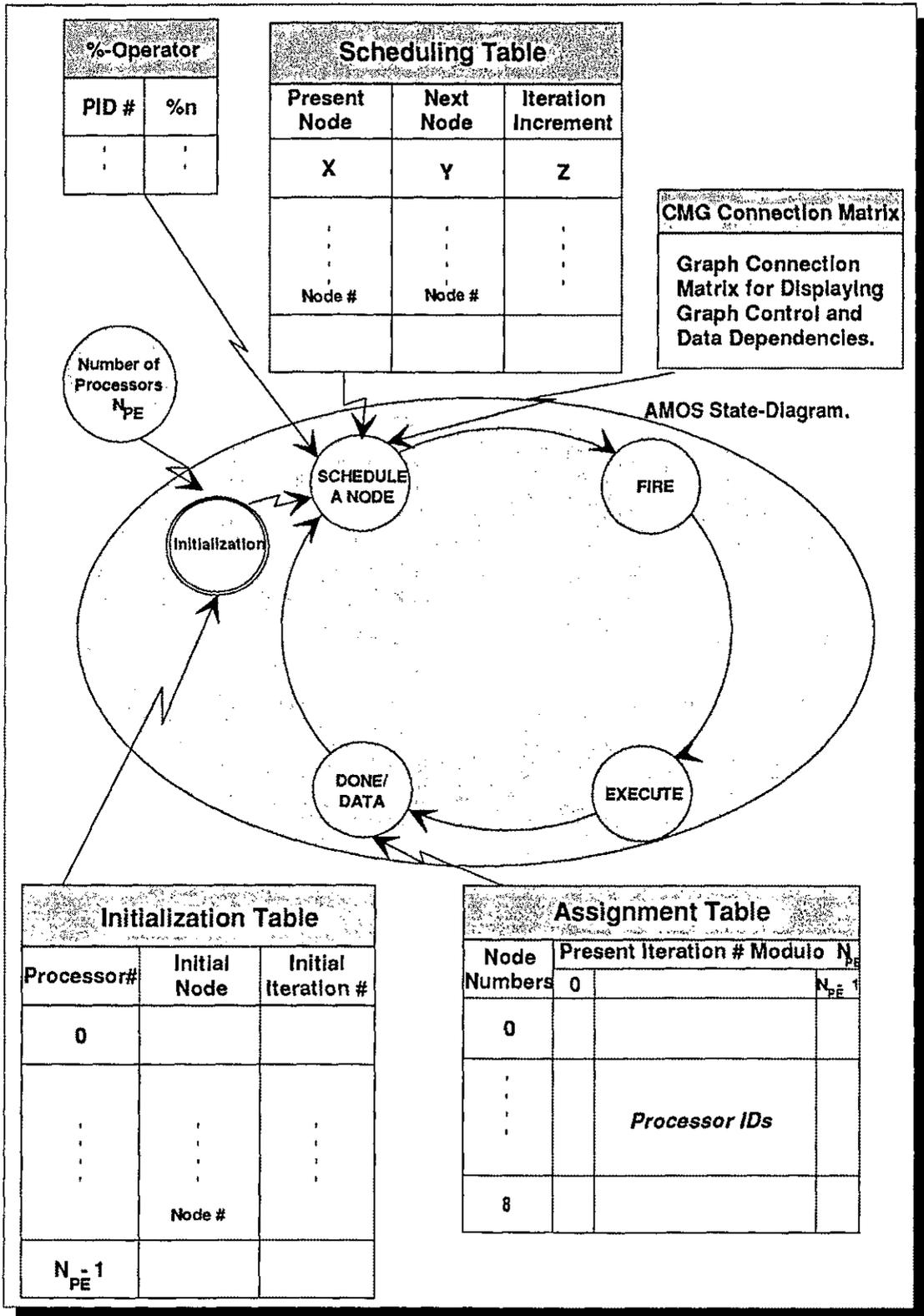


Figure 3.10 Interaction Between AMOS States and Data Structures.

- [5] a *resource manager* that allocates(assigns), removes and manages computing resources (processors) within the system;

Other associated tasks handle I/O (with peripheral devices, file systems etc.), perform processor load balancing and implement reliability features to ensure a graceful degradation of performance in the event of a failure [HWANG84].

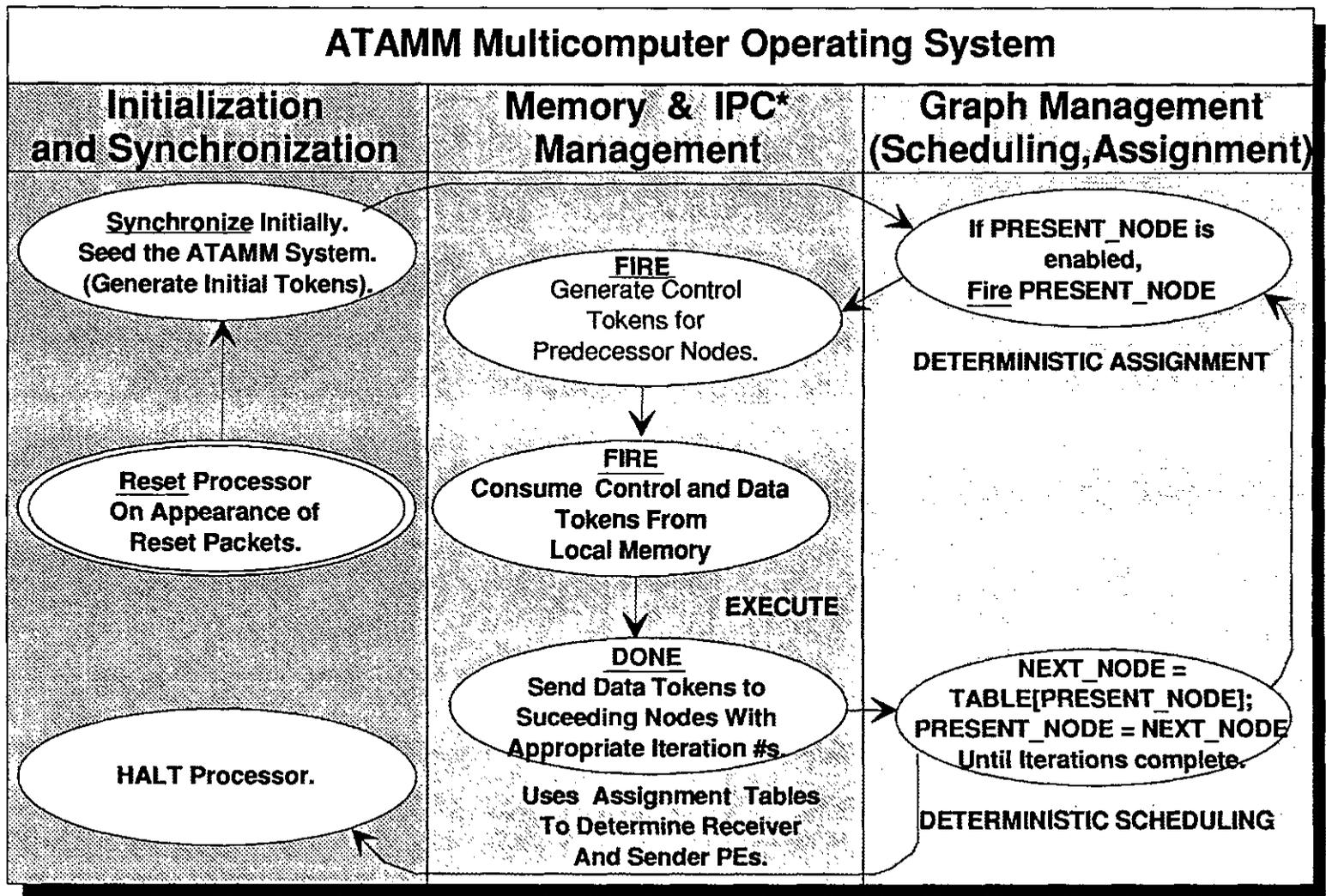
It can be quickly seen that the ATAMM modelling process naturally handles issues related to deadlock avoidance, maintenance of processor load symmetry and assurance of graceful degradation (via specification of ATAMM operating points). Therefore, the five major components stated above are the ones whose functionality should be visible in ATAMM. These functional constituents have been collapsed into three principal tasks, as shown in Figure 3.11. Correspondingly, the AMOS state machine has been molded to concur with these task descriptions.

The ecumenical effect of decentralizing computation and communication tasks is realized in the form of dataflow operations, that can execute at a natural speed defined solely by data and control dependencies and unrestrained by operating system overheads.

### **3.4 Transformation of a LAN Environment to Support Multicomputing**

The hardware characteristics of the ATAMM testbed are described in this Section. Fundamental system entities that support and interact with the AMOS kernel are described. The basis for selecting a local area personal computing system as the hardware for the ATAMM testbed is also established.

From its inception, our research activity has been aimed at implementing an ATAMM testbed that uses networked personal computers or workstations. The idea is to utilize economical resources to rapidly develop a multicomputing system that can demonstrate dataflow computation and message passing requirements and behavior. Among many other reasons, the prime factor which helped in defining this goal is a requirement to minimize hardware development.



\*IPC : Inter-Processor Communication

Figure 3.11 Distributed AMOS and its Functional Relationship with Standard Multicomputer Operating System Components.

This in turn, helped in reducing development costs and contributed toward maintaining a pre-determined time schedule for testbed design. Computational speed is not an issue, since testbed concepts can be scaled to more adequate hardware implementations. The overriding design constraints for the targeted hardware includes the necessity to ensure simple inter-connections between processing elements and to ensure a modularity of hardware interfaces (which allows scaling). The intent is to quickly surface from an OS level communication layer and focus ensuing research effort on implementing an AMOS that is endowed with a distributed graph manager.

Interconnection mechanisms for macro and micro dataflow computers vary from parallel multiple bus implementations and mesh structures to switched arbitration networks [HWANG84]. However, the ATAMM model is concerned primarily with large grain dataflow operations. This implies that problems that can be scheduled using ATAMM possess the characteristic of demanding greater computational time than corresponding inter-processor communication time. Consequently, one suitable interconnection mechanism for an ATAMM based system is a bus oriented environment. An easy method of achieving this across discrete personal computers is to network them through an ethernet LAN. The corresponding realization of the testbed hardware is portrayed in Figure 3.12.

Commercial LANs are configured to transfer files efficiently. Consequently, it was decided to achieve message passing by utilizing the file transfer support provided by MS DOS and network software. Alternatively, an implementation that utilizes message passing through packet communication could be used. This approach reduces the operating system overhead of file processing. However, the use of packet transfer mechanisms requires handling of a corresponding increase in the complexity of processor addressing.

The overhead of file processing is significant. File handling requires file names, file handles, File Allocation Table, FAT entries, directory entries and other system data structures to be created and manipulated. Overhead time is increased further by the use of hard drives as storage space for files. Consequently

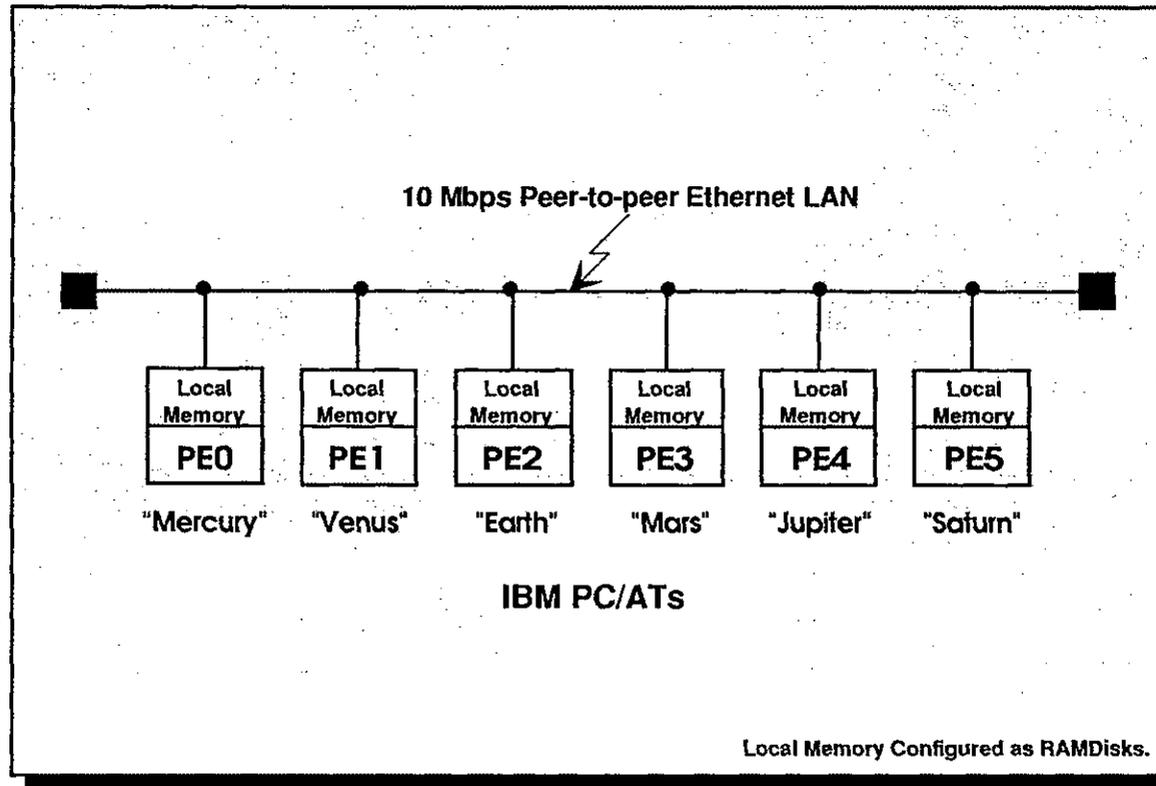


Figure 3.12 ATAMM Testbed Components : Processing Elements and Local-area-network.

added system overheads arise due to significant delays caused by operations such as disk access, file read and file write. In order to circumvent the serious communication delays that a true disk based file system would cause, RAM disks are used in place of hard disks. RAM disks are portions of RAM that are specially configured to emulate a virtual disk. It implements the features of a disk based file system such as FATs and directories in a specific portion of system RAM. A RAM disk based file processing mechanism offers a simple addressing scheme and extremely fast file transfers and I/O.

Processors in a multicomputing system require local memory for storing dataflow algorithm code/data and operating system code/data. The testbed uses the conventional RAM as local memories for individual dataflow processors. In addition, a multicomputing system needs to possess some amount of shared memory which is accessible by all processors, in order to achieve message passing. One way of building a shared memory is to map portions of local memory on individual processors onto a shared memory space, such that every processor is granted access to these specially allocated local memories. Shared memory configured in the above manner is termed as Distributed Shared Memory, DSM, (an introductory discussion of which appears in [TANENBAUM92]). Portions of a DSM which form a logically contiguous, universal memory element, are actually distributed among processors of the system.

DSM can be modelled in networked environments by a LAN based system that offers a peer-to-peer operation, thus permitting individual computers to access directories on other computers. In our implementation, this translates into the capability of being able to access the RAM disk of every other computer. Consequently the operations of token movements translate into generating files that physically reside in the RAM disk spaces of destination processors. Correspondingly, in order to test for the presence of requisite data and control tokens (before firing a node), a processor only needs to look into its RAM disk for the availability of these tokens. Thus, local memory and memory for IPC are generated from a combination of conventional RAM and memory configured as

virtual disks. In sum, the RAM disk implementation helps in neatly tying the concepts of memory communication and inter processor communication. A pictorial description of the logical synapses between testbed components and the modelling of a DSM space is presented in Figure 3.13.

### 3.5 Testbed Operation

A description of the aggregate behavior of testbed hardware and AMOS is presented in this Section. The ensuing narrative traces through the operational steps suggested for implementing a dataflow algorithm on the testbed.

As outlined in Chapter Two, dataflow operations commence with the construction of ATAMM marked graph models, the AMG and the CMG. During the course of the modelling process, SGP and TGP diagrams are constructed to specify the parallel and pipeline concurrency desired. Upon finding a valid schedule loop for the problem, tabular data structures as outlined previously are detailed in a task specification file. The file contains elements such as the augmented connection matrix, initialization table, scheduling table, assignment table and the modulo operator table.

Testbed operations are started by configuring PEs for peer-to-peer LAN communication between all local RAM disks. Identical copies of AMOS, node code and the task specification file are deposited in local RAM disks of all PEs. Thereafter, AMOS is invoked on each processor. All PEs wait for an initial trigger signal from the external environment, such as a key press from the system user. The provision of an initialization signal allows the processing elements an opportunity to synchronize local clocking mechanisms. It is necessary to preserve a global time sense, especially for diagnostic purposes.

After receiving an trigger, each copy of AMOS examines its task specification file. The task of generating initial control and data tokens as required by the CMG specifications, is bestowed on processor zero in the testbed. This artificial creation of initial tokens provides an opportunity to seed the ATAMM system with initialization data and control tokens.

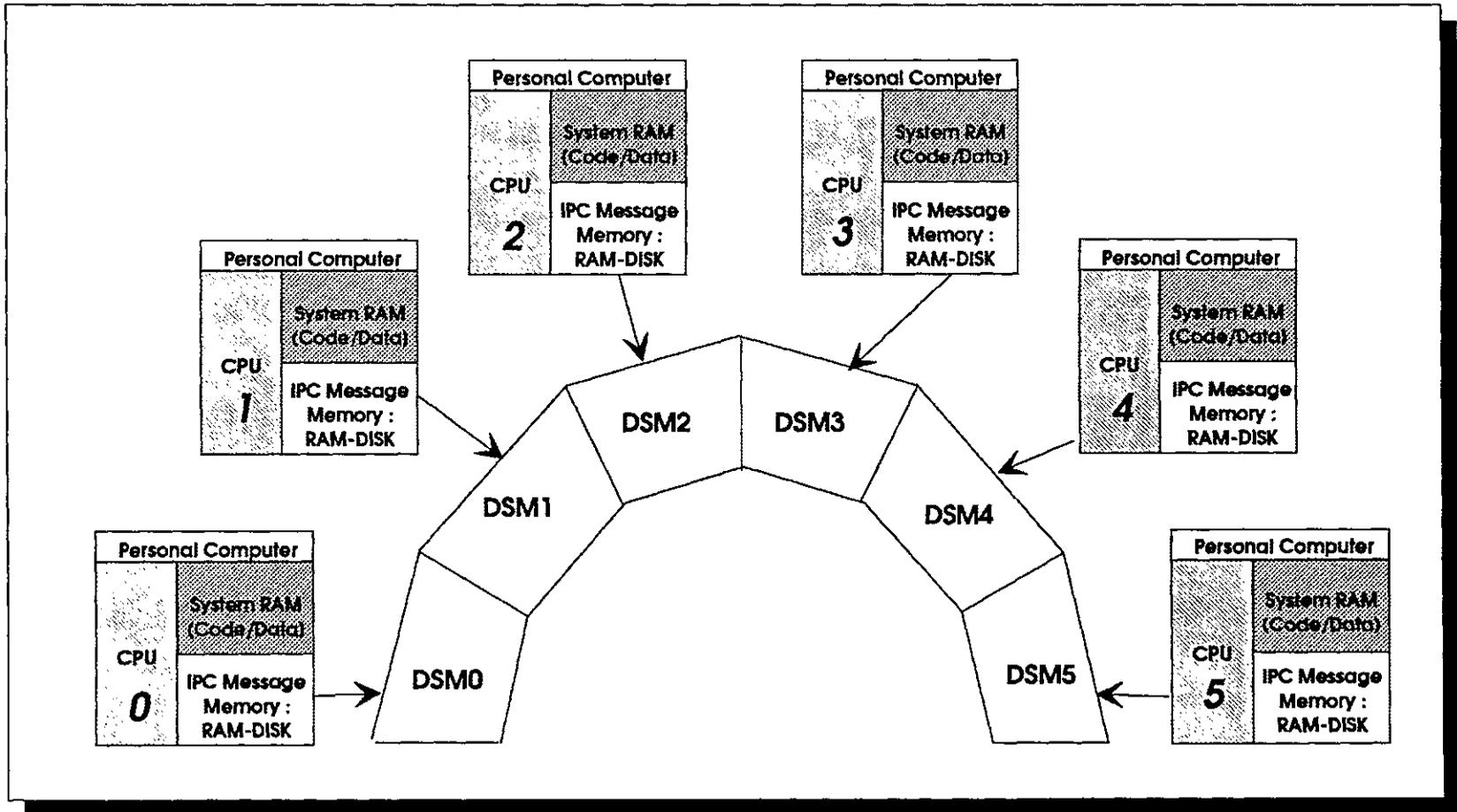


Figure 3.13 Distributed Shared Memory, DSM, for the ATAMM Dataflow Testbed.

Thereafter, using the initialization table, a processor identifies a node to execute. It also determines the initial iteration index for which it is configured to begin execution. A node which is scheduled for execution on a processor is fired only after ensuring that all requisite control and data tokens are available in the local memory of the processor. During the firing process, incoming control tokens are consumed, control tokens for predecessor nodes are generated and incoming data tokens are consumed. While regenerating control tokens, AMOS uses the assignment table and modulo operator table to determine the processors on which predecessor nodes are to be executed. Similar operations occur after the termination of a node, when data tokens are to be transmitted. Special requirements, such as the generation of forwarded data tokens (which are associated with incremented iteration indices) are taken into consideration before overall node execution is terminated.

Terminating conditions (such as a maximum iteration number or exceptions) are specified within the task specification file. The testbed AMOS also engages itself in collecting execution information. Such information is relayed to the user environment in three ways: first statistics are displayed while jobs are in progress; second, a system log file is created which records the time of occurrence and total time taken up by various system tasks; third, a "Fire Data Time" (FDT) file is generated which contains similar information, but in a format compatible with multicomputer performance analysis tools (such as NASA's ATAMM Analysis Tool [JONES93]).

### **3.6 Contrasting Distributed and Centralized AMOS Operations**

Graph Management in AMOS pertains primarily to the tasks of node scheduling and processor assignment. Node schedules may be determined dynamically during run-time or specified statically during compile-time. Associated with these operations are functions that verify the existence of initial "fireability" conditions (such as input tokens) that are needed to execute a node, and those which satisfy graph dependencies upon completion of node execution, in order to

ensure deadlock free dataflow operation. The specific manner in which these tasks are performed defines the distinction between centralized and distributed graph management. The ensuing discussion begins with a description of these activities under centralized graph management and thereafter contrasts them with corresponding strategies for distributed operation. The discussion assumes that node scheduling under a centralized AMOS is dynamic, while that under distributed AMOS is static. Furthermore generic operations under a centralized AMOS are considered to be representative of corresponding functions of the AMOS implementation in the GVSC and ADM dataflow multicomputers [MIELKE90]. The discussion pertaining to CAGM operations is a recapitulation of a broader description presented previously in Section 2.3.1.

A Centralized AMOS Graph Manager, CAGM, performs graph management in a multi-threaded fashion. Similar but distinct "threads" (copies) of intrinsic CAGM functions are executed on all processing elements, PEs, of a CAGM based system. Processor assignment is governed by an cyclic Processor Queue, PQ, which contains a list of Processor IDentification numbers, PIDs, for those PEs which are available for assignment. Identical copies of the PQ are circulated among all PEs, to retain the PQ's consistency across the dataflow system. The processor that finds itself at the front of the PQ, proceeds with node scheduling functions of the graph manager, while all other processors wait for future enablement. Node scheduling is performed by inspecting data structures that represent the current state of execution of the CMG. A processor examines the CMG in order to find a node for which all requisite input tokens are available in memory. Upon finding an enabled node, the processor removes its PID from the queue and broadcasts a FIRE command to all other processors to indicate an updated version of the CMG, an updated resource queue and other relevant information. The processor progresses to the EXECUTE state for processing the node. Thereafter it broadcasts a DONE/DATA command that contains an updated CMG and computed data. A processors that successfully completes a task, places its PID at the bottom of this queue. Communication is redundantly performed

through system wide broadcasts, so that the CAGM doesn't have to contend with the relatively complex task of identifying specific processors, that get scheduled to operate on successor or predecessor nodes.

Central to the CAGM's node scheduling algorithm is its disjoint relationship with processor assignment functions. In other words, assignment of a processor for task execution is determined only by the PE's position in PQ and the task's current "fireability" state in the CMG. The CAGM performs scheduling and assignment dynamically by preserving a composite view of the current status of the CMG across all processing elements, which is achieved through a redundant broadcast of CMG and processor queue status, coupled with a corresponding distribution of output data tokens. This essentially implies that the mapping of LGDF nodes to processors under a CAGM is dynamically established. These operations have been illustrated in Figure 2.11.

A distributed graph manager differs widely in operation from the above. The goal of a distributed graph manager is to decentralize assignment and task scheduling operations. Instead of executing multi-threaded operations, identical copies of the AMOS kernel are allowed to execute autonomously on different processors.

Multicomputer operating system functions of scheduling/assignment are solely determined at compile-time by cyclo-static policies that are highly deterministic. Each processor in the system, can pick up a node to execute, without waiting to get specifically assigned for execution by extraneous means, such as processor queues. The static scheduling approach thereby helps to reduce the overhead of run-time scheduling that is present in CAGM operations.

Inter Processor Communication is performed in a peer-to-peer basis, which obviates a need for redundant broadcast of tokens. Interrogating the data tables for self scheduling and assignment, a processor can determine which nodes its peer processors are scheduled to execute. Therefore, a processor can transmit tokens pertaining to successor/predecessor nodes to host processors, even before these PEs have had a chance to schedule the nodes for execution. In other words,

token passing operations require the involvement of only recipient and transmitting processors. An additional effect of distributed IPC is that messages are not required to be interpreted to derive implicit information about token movement within a CMG. Transmission of control and data tokens explicitly define atomic dataflow operations. An important aspect of distributed control is the capability of being able to execute graph partitions. The spectrum of possibilities now range from a fully static assignment (for which a processor executes the same set of nodes for every iteration) to a fully cyclo-static assignment (where, a processor does the same node after every  $R$  iterations). The above discussion is summarized in Table 3.9, where the contrasting features of centralized and distributed AMOS operations are stated.

### 3.7 Summary

A succinct recapitulation of the ideas developed in this chapter is presented in this Section. The hypothesis for cyclo-static scheduling is reviewed and the salient features of the testbed implementation are reiterated. In addition, an attempt is made to outline the benefits of the distributed processing environment offered by the ATAMM testbed.

The inherent periodicity of deterministic LGDF algorithms is characterized by the TGP diagram. By tracing the periodic execution of an  $N$  node LGDF graph through (some appropriate)  $R$  number of contiguous steady state TGP frames, it may be observed that the aggregate execution behavior of nodes is represented by a periodic repetition of  $R$  time exclusive, cyclically shifted threads of a specific sequence of AMG nodes. Such a sequence containing an instance of each of the  $N$  nodes of the AMG in a specific order, is termed a node-loop. Assuming the existence of node loops, a hypothesis for statically mapping nodes onto  $R$  processors for  $R$  successive iterations is proposed. As a result of this scheduling strategy, each processor in a multicomputer system gets associated with a cyclically shifted thread of a basic node sequence.

<b>Table 3.9 Features of Distributed and Centralized AMOS</b>		
<b>#</b>	<b>Centralized AMOS</b>	<b>Distributed AMOS</b>
1	At a given instant, only one processor can look for a schedulable node. Scheduling operations are considered to be centralized in this sense.	More than one processor can schedule a node for execution. Therefore, scheduling operations are considered to be distributed among numerous processors.
2	Processors have to be explicitly assigned for execution via means of a Processor Queue.	Processors remain in a state of continuous assignment.
3	AMOS is multi threaded.	AMOS is truly distributed.
4	Scheduling is dynamic, thereby incurring the overhead of run-time scheduling.	Scheduling is static since it is pre determined (during compile-time).
5	Scheduling (i.e. a mapping of a node to a processor) is non deterministic and unpredictable.	The processor assignment for a given node and iteration can be explicitly determined. Scheduling is highly deterministic and predictable.
6	Non deterministic scheduling requires a redundant broadcast of messages.	Deterministic scheduling allows specific message passing operations to be performed between peer processors.
7	Message passing involves F, D and R broadcasts, from which implicit token information has to be extracted.	Atomic token passing operations obviate the need to specially interpret messages.
8	Assignment and scheduling operations are disjoint. A processor is assigned when its PID surfaces to the top of the queue. Scheduling depends on the current state of the CMG.	Every processor remains continuously assigned for node execution and schedules a node by inspecting system tables that aid cyclo-static scheduling.
9	Graph partitions cannot be handled naturally.	Block cyclo-static or static schedule loops allows graph partitions to be handled.

The scheduling policy generates a unique mapping of nodes to processors for  $R$  relative iteration numbers. As a result it establishes a mutually exclusive and collectively exhaustive node schedule that guarantees that every node associated with a unique, relative iteration within  $R$  contiguous TGP frames, will be executed on a specific processor. This particular schedule is repeated across multiple  $R * R$  TGP frames to produce the iterative behavior of the AMG. Such a scheduling policy is termed as cyclo-static. Execution of a cyclo-static loop on a multicomputer system allows every processor in the system to execute all AMG nodes in  $R$  contiguous TGP frames. However, limited forms of cyclo static behavior can also be observed. An example of this is a block cyclo static scheduling policy, in which the overall node schedule is characterized by a set of  $k$  ( $1 < k < R$ ) blocks of nodes. Each of these blocks contains fewer than  $N$  nodes, is cyclo-static within itself and is mutually exclusive with respect to other blocks. In other words, such blocks constitute a partition on the AMG. An extreme case of block cyclo static behavior is the one with  $k = R$  blocks. Such a scheduling policy is termed as static. In a static schedule, processors constantly execute a limited set of nodes.

Since cyclo static node loops imply unique node to processor mappings, the resultant multicomputer can naturally assume a distributed processing architecture, in which processors are made responsible for autonomously executing their pre-assigned threads of a node sequence, while satisfying LGDF data and control dependencies. The implementation of a cyclo-static node loop on an ATAMM multicomputing system is aided by the creation of a variety of data structures that contain information such as topology and special dataflow characteristics of the AMG, the specific sequence of nodes in the loop, the relative iteration increments required while migrating among nodes in the loop, initialization information etc. This information structure is used as a basis for performing AMOS operations such as node scheduling, checking for enablement conditions of nodes, communication of data and control tokens with peer processors and execution of nodes.

An AMOS implementing a static scheduling policy has been coded in C for the testbed. The implementation consists of personal computers connected through a peer-to-peer ethernet LAN. The PCs act as the processors of a multicomputing system. They communicate tokens through file transfers on the ethernet. Processors can be specifically targeted for peer-to-peer communication. This obviates the need for redundant broadcast of computed data and control tokens. This particular communication mechanism acts as a model for Distributed Shared Memory for the system. Therefore, given a collection of networked personal computers, the testbed implementation transforms the facility into a viable environment for supporting distributed LGDF multicomputing.

Some of the potential benefits of the testbed are:

- [1] The six processors of the testbed execute deterministic LGDF AMGs containing upto eight nodes. Prior to execution, a specific node-sequence needs to be identified for maintaining a desired TBO (throughput) and translated to fit the information structure of the testbed AMOS. Beyond this initial effort, testbed operation is autonomous and independent of any supervisory control.
- [2] Execution of the AMG results in natural dataflow operations with highly diminished scheduling overhead (as compared to that incurred during dynamic scheduling).
- [3] A distributed execution of AMOS graph management strategies is seen. Analogously, the execution of AMG nodes is also distributed.
- [4] Communication of data and control tokens is performed on a need basis, meaning that only peer processors get involved in sending and receiving tokens.
- [5] A degree of predictability is added to the ATAMM system since it can be predicted beforehand, which processor shall execute a given node for a particular iteration.
- [6] Block cyclo-static or fully static schedules create the opportunity for executing LGDF algorithms in heterogeneous architectures.

## **CHAPTER FOUR**

### **EXPERIMENTAL RESULTS**

#### **4.0 Introduction**

Experiments that establish the testbed's ability to execute dataflow algorithms scheduled by a distributed AMOS are reported in this chapter. A brief discussion on the testbed environment created for supporting experimentation appears in Section 4.1. Experiments that demonstrate the handling of cyclo-static, block cyclo-static and static schedule loops are described in Section 4.2. Details on experiments that involve graphs with special dataflow properties, appear in Section 4.3. A cyclo-static schedule for an eight node example that requires six processors is also shown in 4.3

The generation and consumption of tokens prior to and after node execution consumes a measurable fraction of total node execution time. AMOS events that contribute towards the communication overhead are discussed in Section 4.4. Results obtained through the ATAMM Analysis Tool [JONES93] are used to graphically visualize the events that take place during AMOS operation.

#### **4.1 Utilization and Modification of Testbed Features for Experimentation**

The testbed has been designed for autonomous operation that requires minimal user intervention. The user supplies AMOS node code and data structures such as the graph connection matrix, node loop and an initialization sequence. Problem specifications are listed out in a text file, which is subsequently distributed among the processors of the system. To suit certain aspects of experimentation, additional fields are introduced in the specifications file. For example, information pertaining to artificial node execution timings appear as node

delays in current versions of the specifications file. A sample format of the specifications file appears in Table 4.1. For obtaining results from experiments, methods for statistical collection of results are necessary. The testbed reports results in three formats:

- [1] Visual Display (of execution characteristics, global time etc.),
- [2] A log of system activities in a text file,
- [3] An FDT file in the ATAMM Analysis Tool format [JONES90].

Sample formats of the output from each of these appear in Tables 4.2 through 4.4.

An experiment is begun with the formation of a specifications file and is terminated with the inspection of the on line display, system log file or the FDT file on the ATAMM Analysis Tool. For each experiment discussed in subsequent sections, the following details have been shown: a brief explanation of experimental aims, an AMG/CMG, a TGP, a specifications file, analyzer report and comments on results.

## **4.2 Experiments Pertaining to Scheduling Policy**

Examples that demonstrate the three scheduling categories (cyclo-static, block cyclo-static and static) are presented in this section. These scheduling policies are shown with reference to the AMG in Figure 2.3(b), whose TGP appears in Figure 2.8(b). It is indicated in Table 3.1, that a variety of scheduling strategies are possible for a given graph. As specific illustrations of scheduling possibilities, the schedule loops shown in Figures 3.3, 3.4 and 3.5 are used in these experiments. The effect of these schedule loops on AMG execution is discussed in this section.

The specifications file for a cyclo-static schedule is shown in Table 4.5. The results of processing are shown graphically in Figure 4.1. These results have been extracted from the output display of the ATAMM Analysis Tool. In this example, processors execute AMG nodes in the cyclo-static sequence defined in Figure 3.3.

Table 4.1 Format of the Specifications File		
Entry #	Entry Name	Description
1	Connection-Matrix for CMG.	[8 x 8 x 3] Matrix to indicate graph data structures. A non negative entry in the matrix indicates the presence of an data edge from row (predecessor node) to column (successor node). Each (x,y,z) entry indicates that x is the <b>iteration increment</b> for the corresponding data arc, y is the number of <b>initial data tokens</b> needed on the data arc and z is the <b>buffer length</b> for the control arc.
2	LP.	Number of Logical Processors required for the problem.
3	Initialization Table.	[1 x LP x 2] Matrix to indicate initial node to Processing Element scheduling patterns. Each (x,y) entry indicates (node number, iteration-number).
4[a]	Scheduling Table : Next Node.	[1 x LP] table to indicate the next node scheduling pattern.
4[b]	Scheduling Table : Iteration Incr.	[1 x LP] table to indicate the iteration number increments.
5	Assignment Table.	[8 x LP] table to indicate node to physical element assignments for LP iterations.
6	Modulo Operators.	[1 x LP] table to indicate modulo (%) values for each node. AMOS uses these modulo values to compute a processor ID in the assignment table. They are the same as the number of processors tied to a circuit.
7	Maximum Iteration #.	The total number of iterations to be done. This is the number specified in the field plus one.
8	Node Delays.	The individual node execution times.
9	Drive Letters.	[LP * 3 characters] string table for drive letters for the RAM disks of PEs involved in the system.

Note : The expression [A x B] indicates a matrix with A rows and B columns.

<b>Table 4.2 On Line Screen Display For Testbed Operations</b>	
ATAMM Dataflow Computer (Cyclo-Static) : PE [PID of the processor]	
Time elapsed since big bang	: [hrs:mins:secs], [milliseconds]
Worked on iteration number	: [last iteration number]
Last node executed by PE [PID]	: [last node number]
Node code Execution Time	: [seconds]
Data communication Time	: [seconds]
Control communication Time	: [seconds]
AMOS Overhead	: [seconds]
[System Messages]	

Note : Information shown above in square brackets, [], are filled in by AMOS.

<b>Table 4.3 Format of the System Log File</b>						
Local Area Multicomputing System.						
System log on	: [day] [month] [date] [time in hh:mm:ss] [year]					
Processor #	: [PID on which this log is taken]					
Initialization time	: [milliseconds for time taken to initialize]					
Iteration number	Node number	Exec. Time	Data Time	Control Time	Idle Time	Recorded at
[Iter.#]	[Node #]	[ms]	[ms]	[ms]	[ms]	[Global Time]

Note : A System Log file is create for each processor, which are subsequently collected an concatenated to form a larger file.

Table 4.4 FDT File Format as Input to ATAMM Analysis Tool					
Time at which the FDT event occurred	Event Type	Node Number	Color (Simplex)	Processor Identification Number.	Iteration Number
[milli-seconds]	One of following:  Reset Fire Node Control In Control Out Data in Process Begin Process End Data out Done node	[NODE#]	Always 1.	[PID_#]	[Iter. #]

Note : FDT Events are one of the following:

- Reset : Indicates information about system reset.
- Fire Node : Indicates time at which a node initiated execution.
- Control Out : Indicates time at which control tokens were regenerated.
- Control In : Indicates time at which control tokens were read in.
- Data In : Indicates time at which data tokens were read in.
- Process Begin : Indicates time at which execution of node code was begun.
- Process End : Indicates time at which execution of node code was stopped.
- Data Out : Indicates time at which data tokens were generated.
- Done Node : Indicates time at which a node completed execution.
- Halt : Indicates time at which a processor terminated its activities.

Table 4.5 Specifications File for the TGP in Figure 4.2								
Field	Description							
Connection Matrix	1,1,1	-,-,-	-,-,-	-,-,-	0,0,1	-,-,-	-,-,-	-,-,-
	-,-,-	-,-,-	-,-,-	0,0,1	0,0,1	-,-,-	-,-,-	-,-,-
	-,-,-	-,-,-	1,1,1	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-
	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-
	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-
	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-
	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-
	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-	-,-,-
Logical Processors	3							
Initialization Table	0,0	2,0	0,1					
Next Node	1	3	4	2	0	-	-	-
Next Increment	0	0	0	1	2	-	-	-
Assignment Table	0	2	1					
	0	2	1					
	1	0	2					
	0	2	1					
	1	0	2					
	-	-	-					
	-	-	-					
Modulo Operators	3	3	3	3	3	-	-	-
Maximum Iteration	9							
Node Delays	3.0	2.0	2.5	1.5	3.0	-	-	-
Drive Letters	E:	F:	G:					

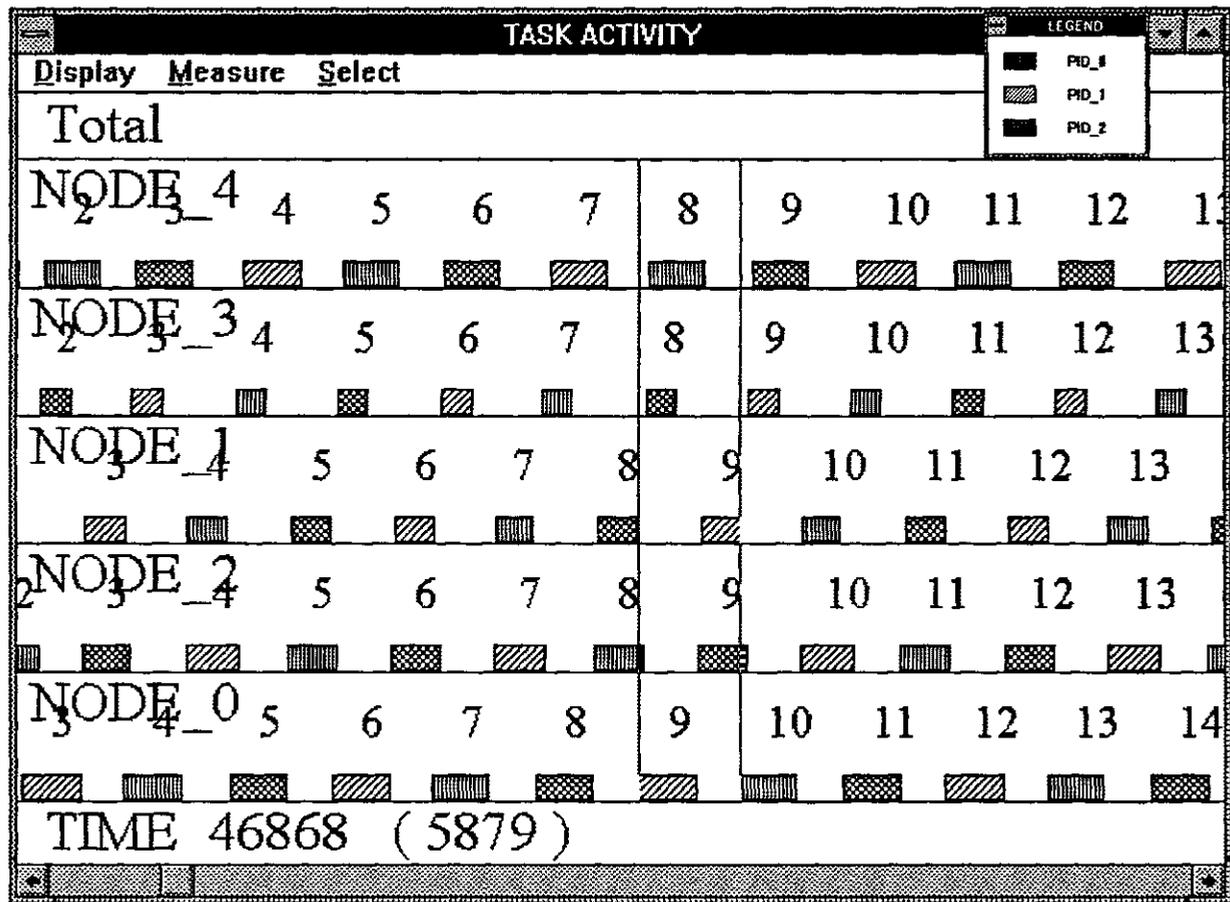


Figure 4.1 ATAMM Analysis Tool Output for Cyclo-Static Scheduling of the AMG in Figure 2.3(b).

For this example, processor zero is scheduled to operate on the node sequence 0,1,3,2,4, beginning with node zero for iteration  $i$ . Processor two is scheduled to operate on the same sequence but it executes its initial node, node zero, for relative iteration  $i+1$ . Similarly processor one executes its thread of the sequence beginning with node zero for iteration  $i+2$ . This behavior of processors can be traced out in Figure 4.1 by following the hatched blocks that represent them. For instance, we notice that processor zero (marked as PID\_0) executes node zero for iteration four, then it migrates to node 1 for iteration four, then to node three for iteration four. The next node it executes is node two, but this is associated with an iteration increment of one. Consequently, node two is executed for iteration five. Similarly, node four is executed for iteration five. Processor zero repeats execution of the loop with node zero for iteration seven. Note that the corresponding increment in iteration index for the transition from node four to zero in the bubble diagram of Figure 3.3 is +2. The sequence of node executions for the remaining processor may be similarly verified. For instance, the execution sequence for processor one is: node zero - iteration three, node one - iteration three, node three - iteration three, node four - iteration four, node two - iteration four and finally node zero again but for node six. The interesting aspect of cyclo-static operation as shown in Figure 4.1 is the mutual exclusivity and collective exhaustivity of the scheduling process that becomes apparent. Though each processor performs every node once in a  $3 * TBO$  time frame, the relationship between node, processor and iteration is unique. Consequently, this ensures a deadlock free operation, as seen in Figure 4.1. A variation can be seen between the desired TBO of 5 and the actual TBO of 5.879 in Figure 4.1. It shall be explained in Section 4.4 that this variation is due to a communication overhead.

The specifications for a block cyclo-static schedule appear in Table 4.6 and the results of execution in Figure 4.2. Notice that processor zero executes nodes zero, one and four only in every  $3 * TBO$  time frame. So does processor one, but for different relative iteration numbers. However, processor two invariably switches its attention between executing nodes two and three.

Table 4.6 Specifications for a Block Cyclo Static Schedule for the AMG in Figure 2.3(b)								
Field	Description							
Connection Matrix	---	0,0,1	0,0,1	---	---	---	---	---
	1,1,0	---	---	---	0,0,1	---	---	---
	---	---	---	0,0,1	0,0,1	---	---	---
	---	---	1,1,0	---	---	---	---	---
	---	---	---	---	---	---	---	---
	---	---	---	---	---	---	---	---
	---	---	---	---	---	---	---	---
	---	---	---	---	---	---	---	---
Logical Processors	3							
Initialization Table	0,0	0,1	2,0					
Next Node	1	4	3	2	0	-	-	-
Next Increment	0	0	0	1	2	-	-	-
Assignment Table	0	1	-					
	0	1	-					
	2	-	-					
	2	-	-					
	0	1	-					
	-	-	-					
	-	-	-					
	-	-	-					
Modulo Operators	2	2	1	1	2	-	-	-
Maximum Iteration	12							
Node Delays	3.0	2.0	2.5	1.5	3.0	-	-	-
Drive Letters	E:	F:	G:					

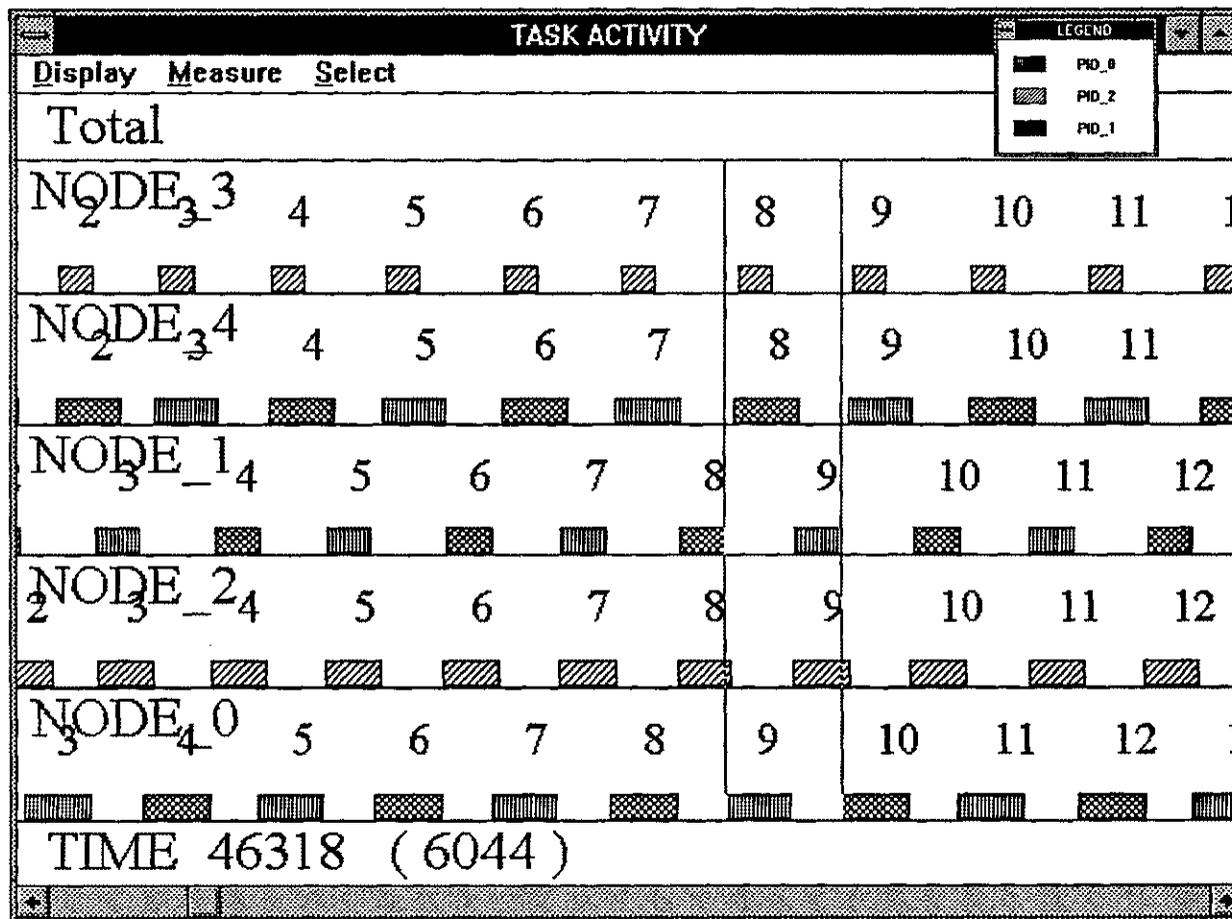


Figure 4.2 ATAMM Analysis Tool Output for Block Cyclo-Static Scheduling of the AMG in Figure 2.3(b). <sup>69</sup>

The actual TBO for the block cyclo-static scheduling example is 6.044 instead of the ideal 5. However, despite this deviation from ideal behavior, the assignment and scheduling process is satisfied as seen from the execution trace in Figure 4.2.

Specifications and results for a purely static schedule appear in Table 4.7 and Figure 4.3 respectively. Notice that processor zero always performs nodes zero and one, processor one does nodes two and three while processor two executes node four. The actual TBO is 6.153 now. As noted earlier this difference which can be attributed to the communication overhead shall be accounted for in Section 4.4.

In essence, the experimental data presented in this section verify the sufficiency of the distributed graph management paradigm for executing cyclo-statically scheduled dataflow algorithms.

### 4.3 Special Dataflow Graphs

Other than exhibiting data and control dependencies, dataflow graphs may expose additional features that are necessary to support special execution requirements. For instance, in order to execute nodes bearing self loops, data tokens need to be generated for future indices. Similarly, the concurrency exhibited by a dataflow algorithm may require a node to be multiply instantiated during the same time interval. In situations like these, it becomes necessary to generate multiple tokens (corresponding to successive iterations) on data or control paths of the CMG.

The CMG for dataflow algorithm that contains a self loop and exhibits multiple instantiation, is presented in Figure 4.4. In this example, node zero redirects one of its outgoing data arcs to itself, thus creating a self loop. Data tokens are usually generated for the same iteration number. However, in a self loop, tokens are generated for a future iteration. In this example the increment for the forwarded iteration index is one, i.e. node zero generates for itself a data token that is used in the next iteration.

Table 4.7 Specifications for a Static Schedule for the AMG in Figure 2.3(b)								
Field	Description							
Connection Matrix	0,0,1	0,0,1						
	1,1,0			0,0,1	0,0,1			
			0,0,1	0,0,1				
		1,1,0						
Logical Processors	3							
Initialization Table	0,0	2,0	4,0					
Next Node	1	0	3	2	4	-	-	-
Next Increment	0	1	0	1	1	-	-	-
Assignment Table	0	-	-					
	0	-	-					
	1	-	-					
	1	-	-					
	2	-	-					
	-	-	-					
	-	-	-					
	-	-	-					
Modulo Operators	1	1	1	-	-	-	-	-
Maximum Iteration	12							
Node Delays	3.0	2.0	2.5	1.5	3.0	-	-	-
Drive Letters	E:	F:	G:					

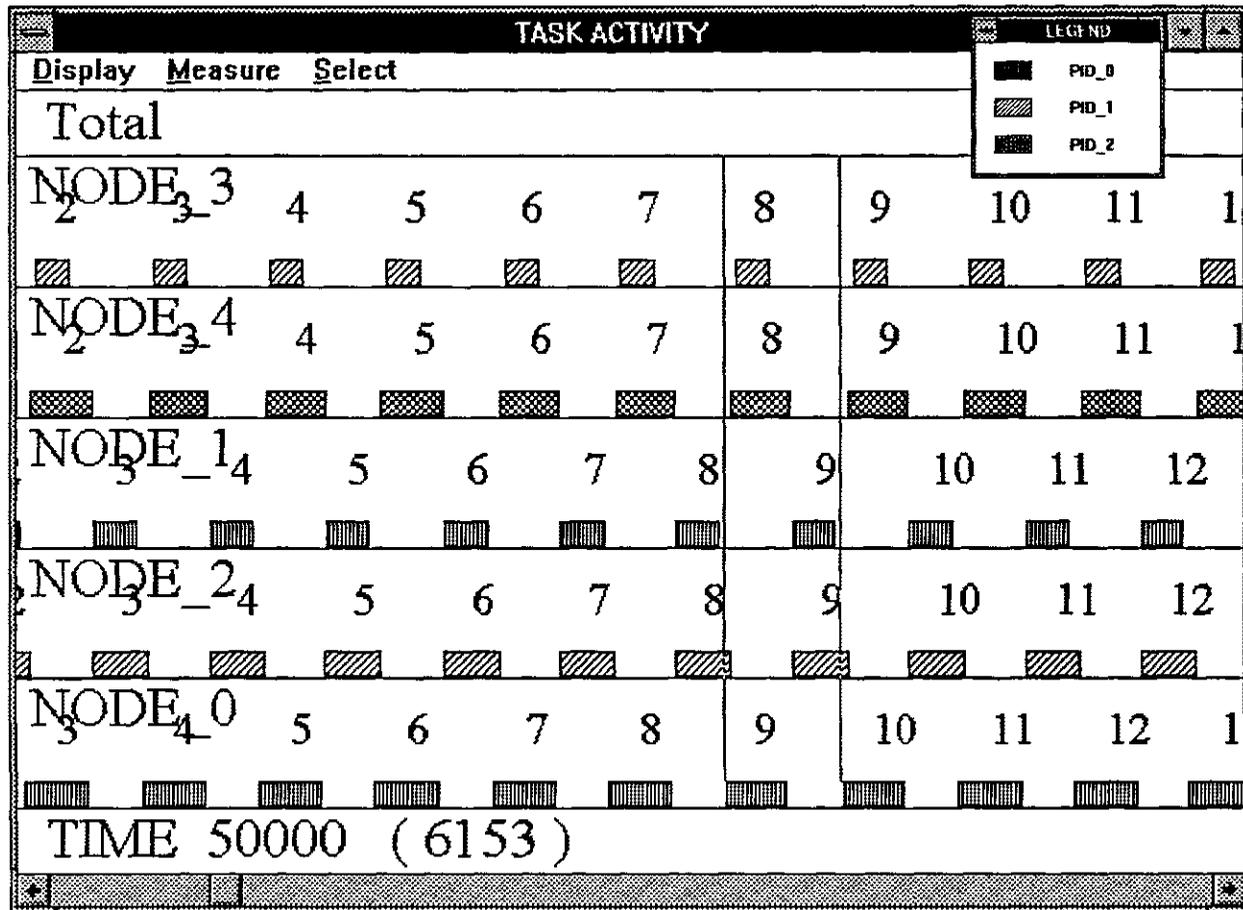


Figure 4.3 ATAMM Analysis Tool Output for Purely Static Scheduling of the AMG in Figure 2.3(b).



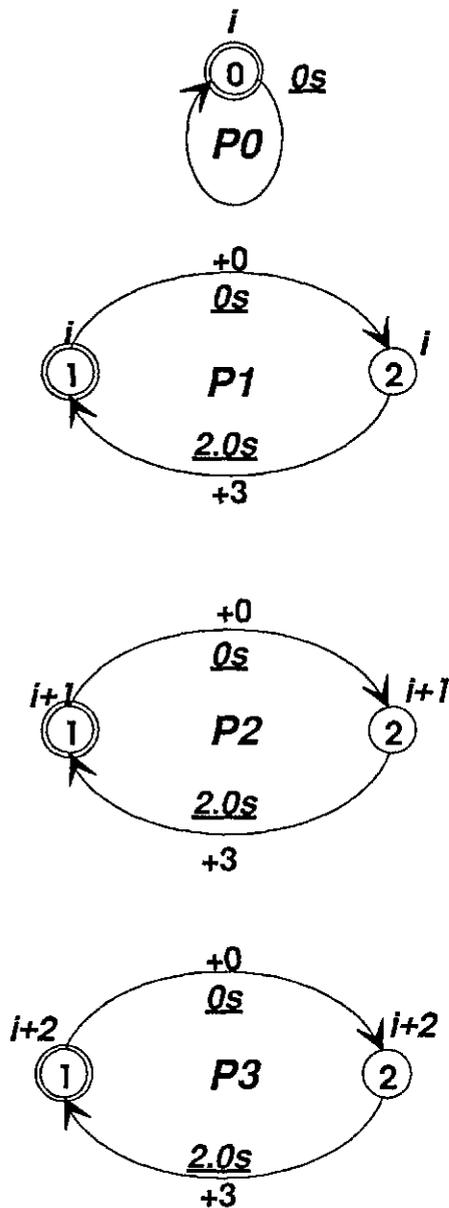
An examination of the TGP for this example (in Figure 4.4), reveals a situation that requires three concurrent instantiations of node one. CMG control arcs are usually associated with a control token buffer length of one. However, this is clearly insufficient for supporting multiple instantiations of a node. Consequently, the CMG in Figure 4.4 contains a buffer of three tokens on the incoming control arc for node one. The presence of these tokens satisfy the concurrency generated due to multiple instantiation of node zero.

A block cyclo-static schedule has been selected for this example, which is portrayed in Figure 4.5. Processor zero is made solely responsible for executing node zero. Processors one, two and three cyclo-statically execute a schedule loop that constants nodes one and two. Notice that processors one, two and three have been initialized to execute node one for three successive iterations. Calculations shown in Figure 4.5 establish the veracity of the schedule loop by equating computing capacity to computing effort. Specifications for this problem appear in Table 4.8. Analysis Tool output for 14 iterations of this problem are shown in Figure 4.6.

In Table 4.8, a control buffer of 3 is established for the CMG control from node two to node one. Similarly a data increment and an initial data token are established on the self loop for node zero. In the execution trace shown in Figure 4.6, the multiple instantiation of node 1 is clearly shown. For instance, in iterations 10, 11 and 12, node one is executed by processors one, two and three respectively. The block static scheduling policy also scheduled node zero permanently with processor zero. The effect of this schedule can also be clearly seen in Figure 4.6. It shall be noticed that instead of an ideal TBO of 6, a value of 6.648 is seen.

The next example presented in this section exploits the capabilities of the testbed to their maximum. The AMG for an eight node dataflow problem is shown in Figure 4.7. The corresponding TGP diagram, also shown in Figure 4.7, exhibits a concurrency that requires a  $R_{\max}$  of six. A cyclo-static schedule loop has been chosen for this AMG. The details of this appears in Figure 4.8.

## Block Cyclo-Static Scheduling and Multiple Instantiation.



$$\begin{aligned} \text{TCE} &= 6.0 + 14.0 + 2.0 \\ &= 22.0. \end{aligned}$$

$$\begin{aligned} \text{Twait} &= 0_{P0} + (0 + 2)_{P1} \\ &= 2.0. \end{aligned}$$

$$\begin{aligned} \text{Rmax} * \text{TBO} &= 4 * 6.0 \\ &= 24.0 \end{aligned}$$

Figure 4.5 Block Cyclo-Static Schedule-Loops for the AMG in Figure 4.4.

Table 4.8 Specifications for a Cyclo-Static Schedule for the AMG in Figure 4.4.									
Field	Description								
Connection Matrix	1,1,1	0,0,1	---	---	---	---	---	---	---
	---	---	0,0,3	---	---	---	---	---	---
	---	---	---	---	---	---	---	---	---
	---	---	---	---	---	---	---	---	---
	---	---	---	---	---	---	---	---	---
	---	---	---	---	---	---	---	---	---
	---	---	---	---	---	---	---	---	---
	---	---	---	---	---	---	---	---	---
Logical Processors	4								
Initialization Table	0,0	1,0	1,1	1,2					
Next Node	0	2	1	-	-	-	-	-	-
Next Increment	1	0	3	-	-	-	-	-	-
Assignment Table	0	0	0	0					
	1	2	3	1					
	1	2	3	1					
	-	-	-	-					
	-	-	-	-					
	-	-	-	-					
	-	-	-	-					
Modulo Operators	1	3	3	-	-	-	-	-	-
Maximum Iteration	13								
Node Delays	6.0	14.0	2.0	-	-	-	-	-	-
Drive Letters	E:	F:	G:	H:					

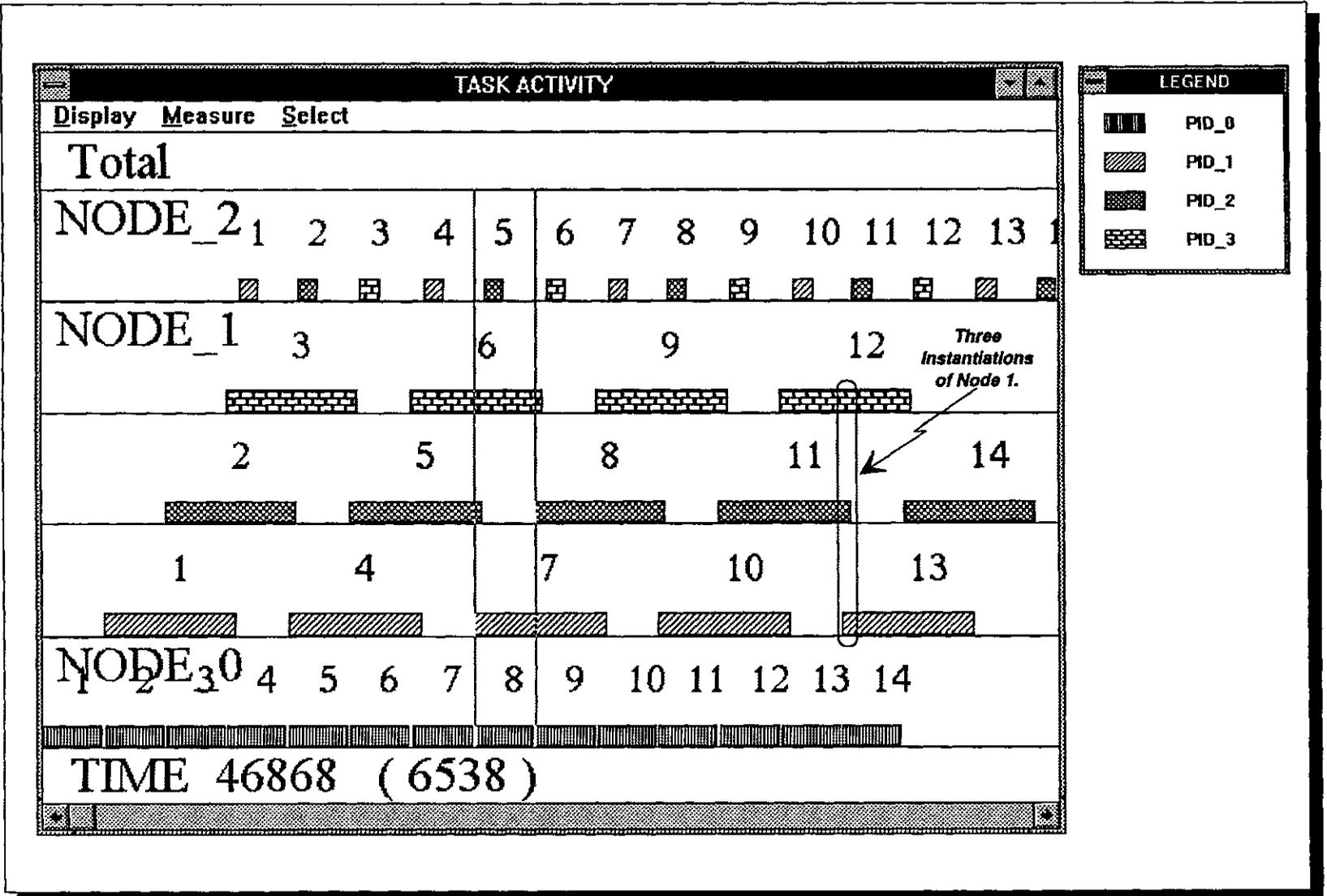
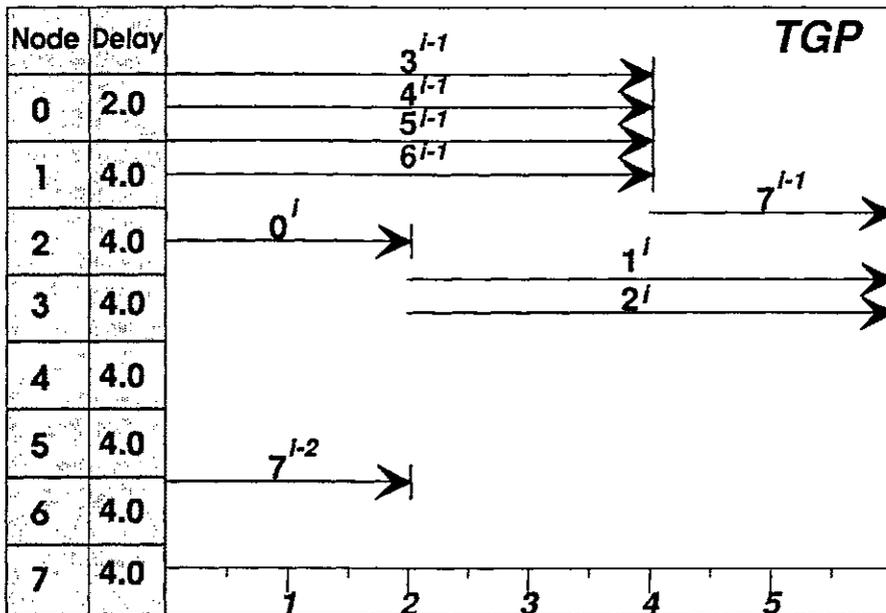
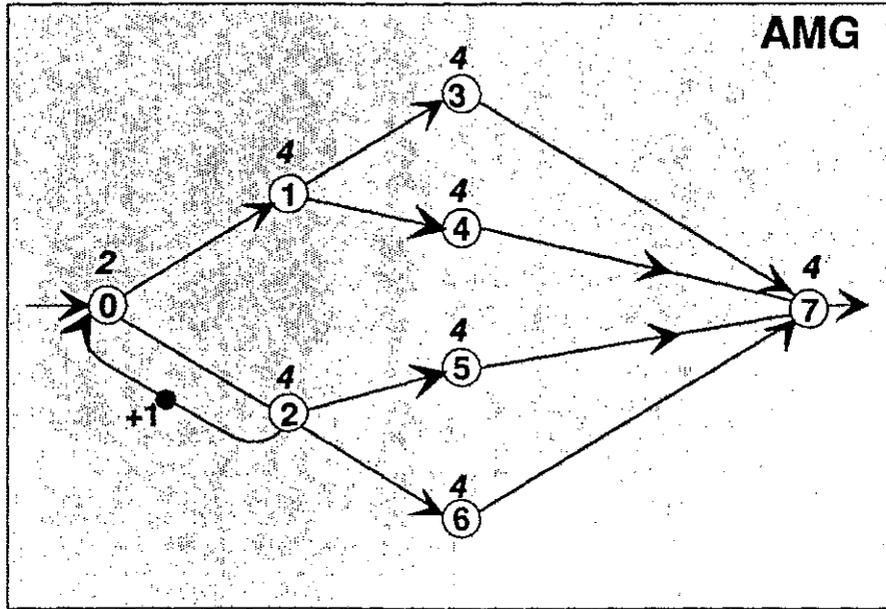


Figure 4.6 Multiple Instantiation of Node One in the AMG of Figure 4.4.



**TBO = 6.0**

Figure 4.7 CMG and TGP for an Eight Node Dataflow Graph Which is Executed on a Set of Six Processors.

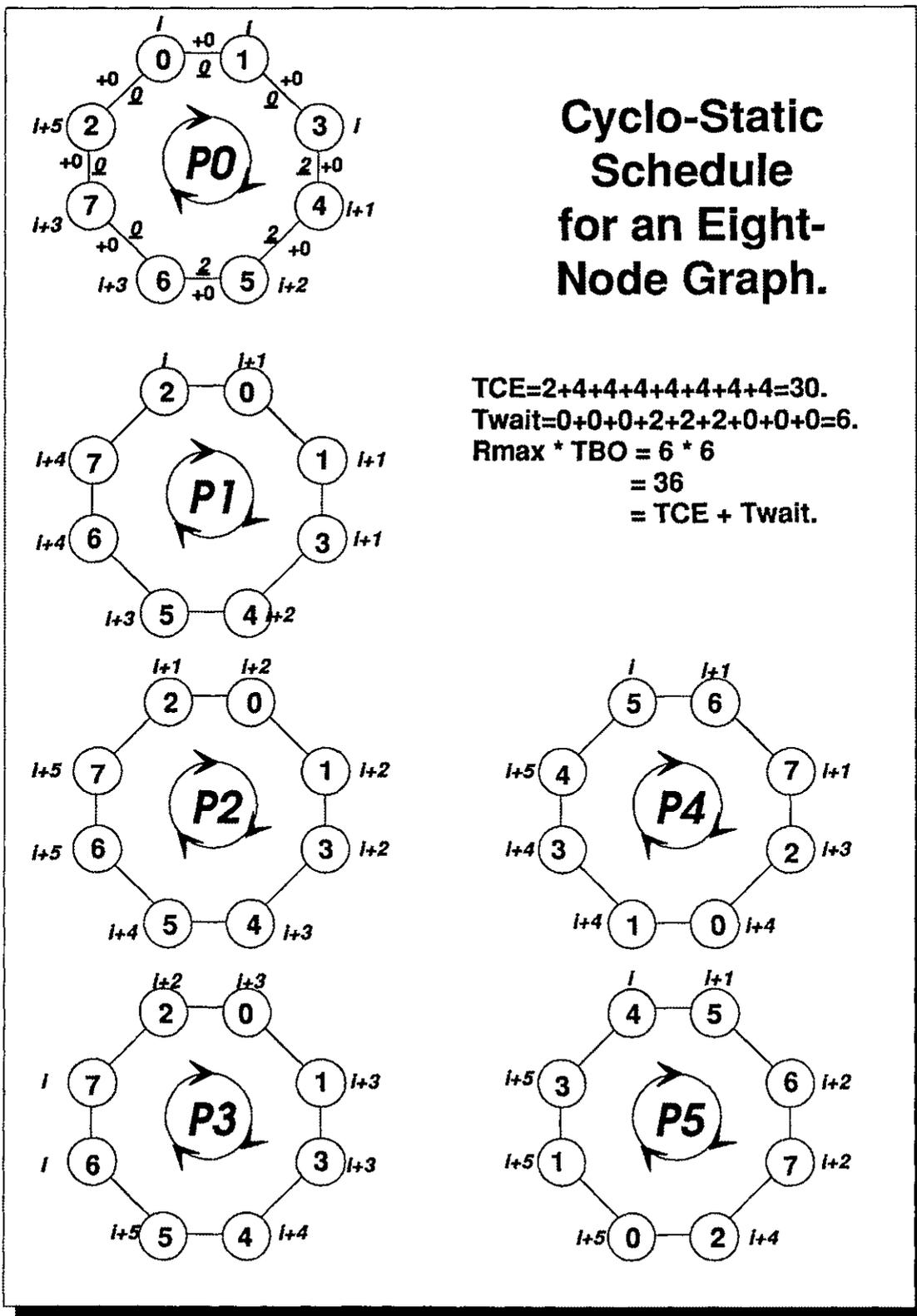


Figure 4.8 Cyclo-Static Schedule-Loop for the Eight-Node AMG in Figure 4.7.

The initialization of processors at a specific node for a particular iterations has been done by observing the steady state execution behavior of the problem. The time measure of the loop satisfies equation 3.1. AMOS specifications are presented in Table 4.9. Eleven consecutive iterations of this problem are shown in Figure 4.9. Instead of an ideal TBO of 6 a TBO of 7.253 is seen. However, the cyclo-static schedule is closely followed. For instance, processor zero follows the sequence of nodes, 0,1,3,4,5,6,7,2 for iterations 1,1,1,2,3,4,4,6.

#### 4.4 FDT Time Marks for AMOS Events and Communication Overhead

The testbed software generates FDT output in the format specified in Table 4.4. The graphical representation of FDT events for one TGP frame for the AMG in Figure 4.7 is shown in Figure 4.10. Different hatching patterns mark individual FDT events in the diagram for each node. In particular, these events are graphically described in Figure 4.11.

A processor starts its operations by marking a RESET. It begins operations by entering the "Examine Tables" state (as shown previously in Figure 3.9). Using its data tables, the processor schedules a node for itself. It ascertains the "fireability" of the node by constantly checking for requisite tokens. Once a node is enabled, the processor marks the FDT event FIRE NODE. Thereafter, the processor proceeds with the generation of control tokens for predecessor nodes. Before commencing with this operation, the processor marks a CONTROL OUT. Thereafter, the processor begins with the consumption of control tokens and data tokens. The beginning of these events is marked by CONTROL IN and DATA IN FDT events, respectively. The processor is now ready to execute a node. It marks the beginning of node execution with a PROCESS BEGIN. Correspondingly, the termination of the node is marked with a PROCESS END. Before generating data tokens for successor nodes, the processor generates a DATA OUT. Task completion is subsequently marked by a DONE NODE. The cessation of processor activities are marked with a HALT. The occurrence of these events has been marked in the AMOS state diagram as shown in Figure 4.12.

<b>Table 4.9 Specifications for a Cyclo-Static Schedule for the Eight Node AMG in Figure 4.7.</b>								
Field	Description							
Connection Matrix	0,0,1	0,0,1	0,0,1	0,0,1	0,0,1	0,0,1	0,0,1	0,0,1
Logical Processors	6							
Initialization Table	0,0	2,0	2,1	6,0	5,0	4,0		
Next Node	1	3	0	4	5	6	7	2
Next Increment	0	0	1	1	1	1	0	2
Assignment Table	0	1	2	3	4	5		
	0	1	2	3	4	5		
	1	2	3	4	5	0		
	0	1	2	3	4	5		
	5	0	1	2	3	4		
	4	5	0	1	2	3		
	3	4	5	0	1	2		
	3	4	5	0	1	2		
Modulo Operators	6	6	6	6	6	6	6	6
Maximum Iteration	10							
Node Delays	2.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0
Drive Letters	E:	F:	G:	H:	I:	O:		

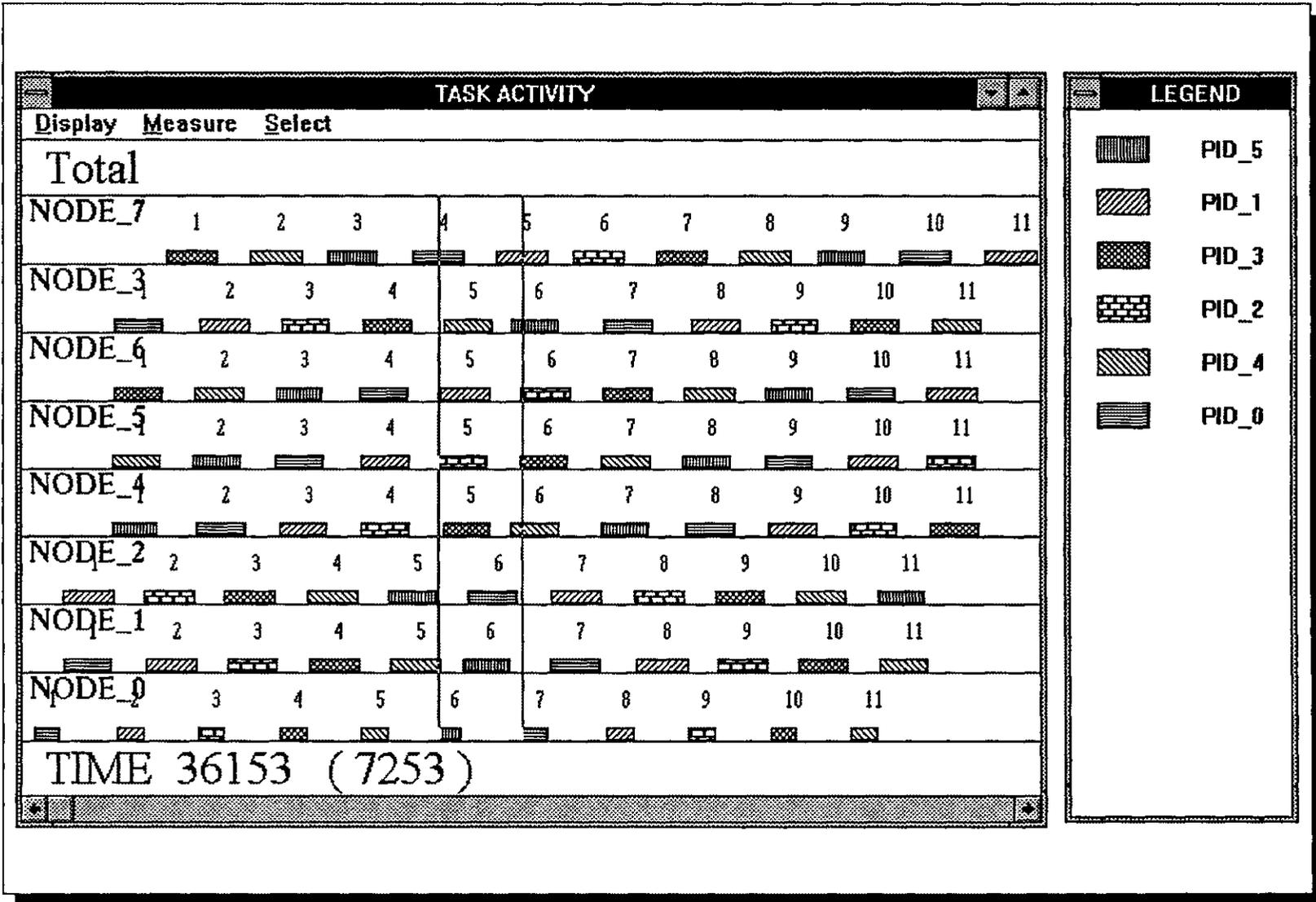


Figure 4.9 ATAMM Analysis Tool Output for the Eight-Node AMG in Figure 4.7.

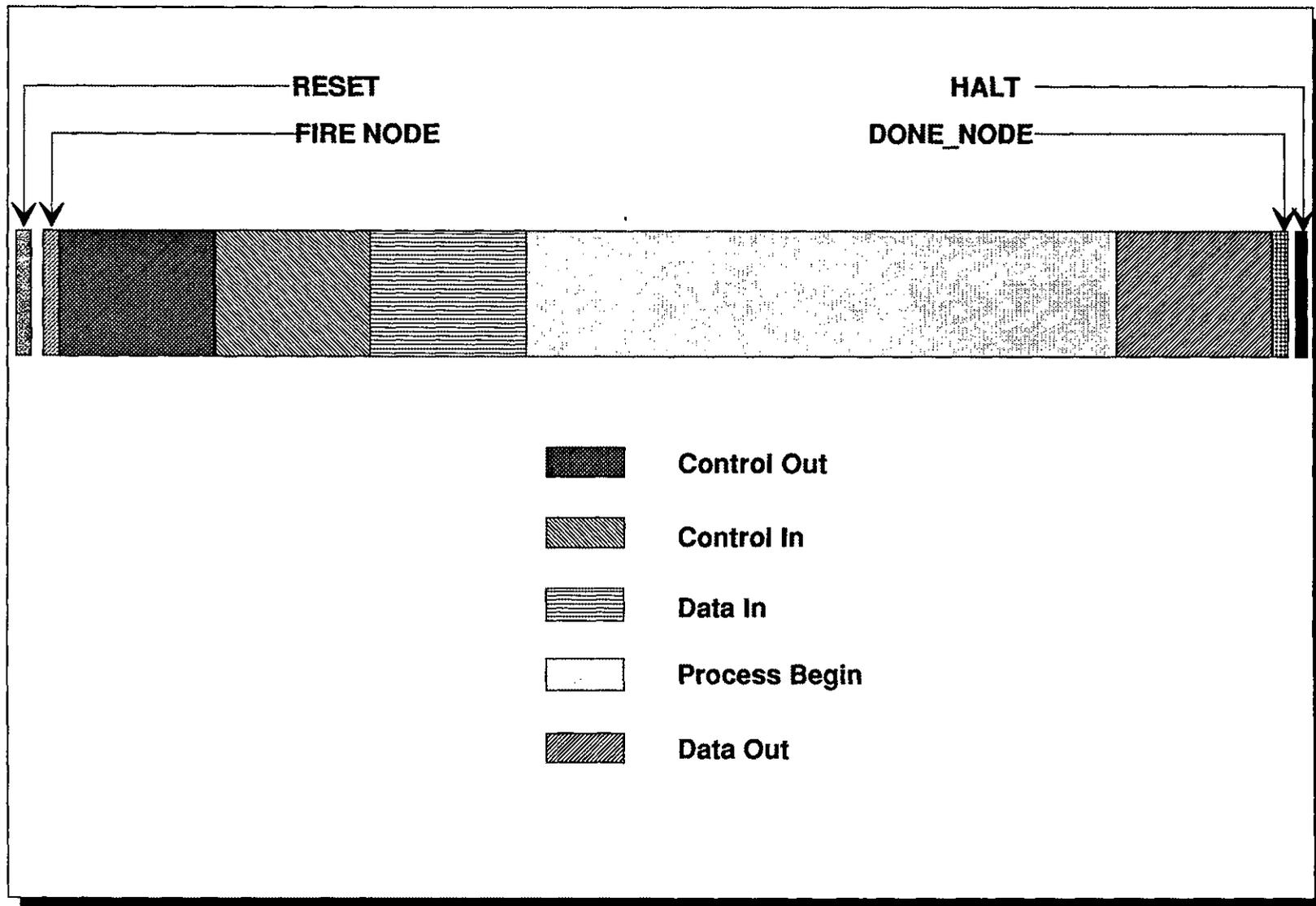


Figure 4.11 AMOS Events and FDT Time Marks.

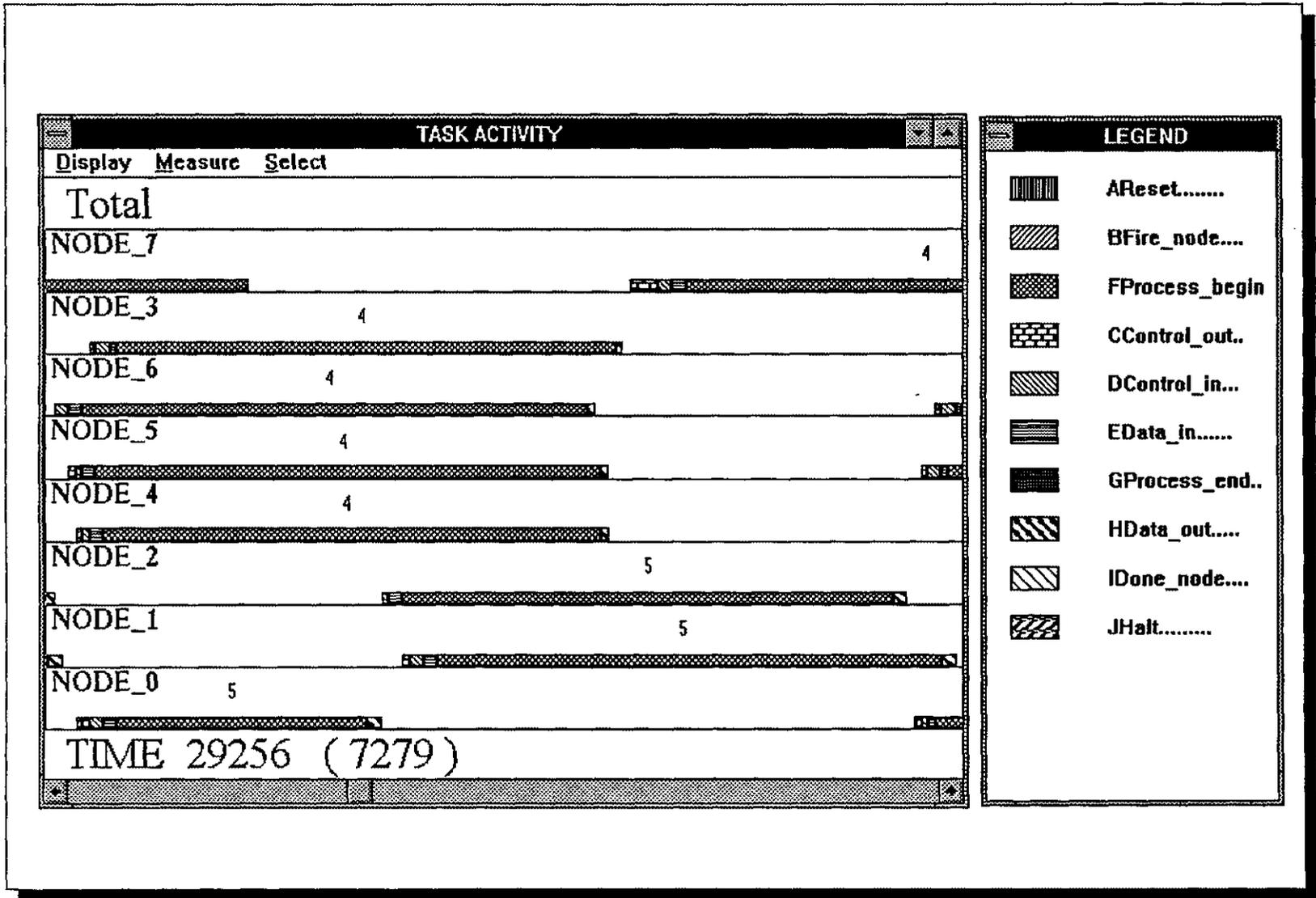


Figure 4.10 FDT Events in One TGP Frame for the AMG in Figure 4.7.

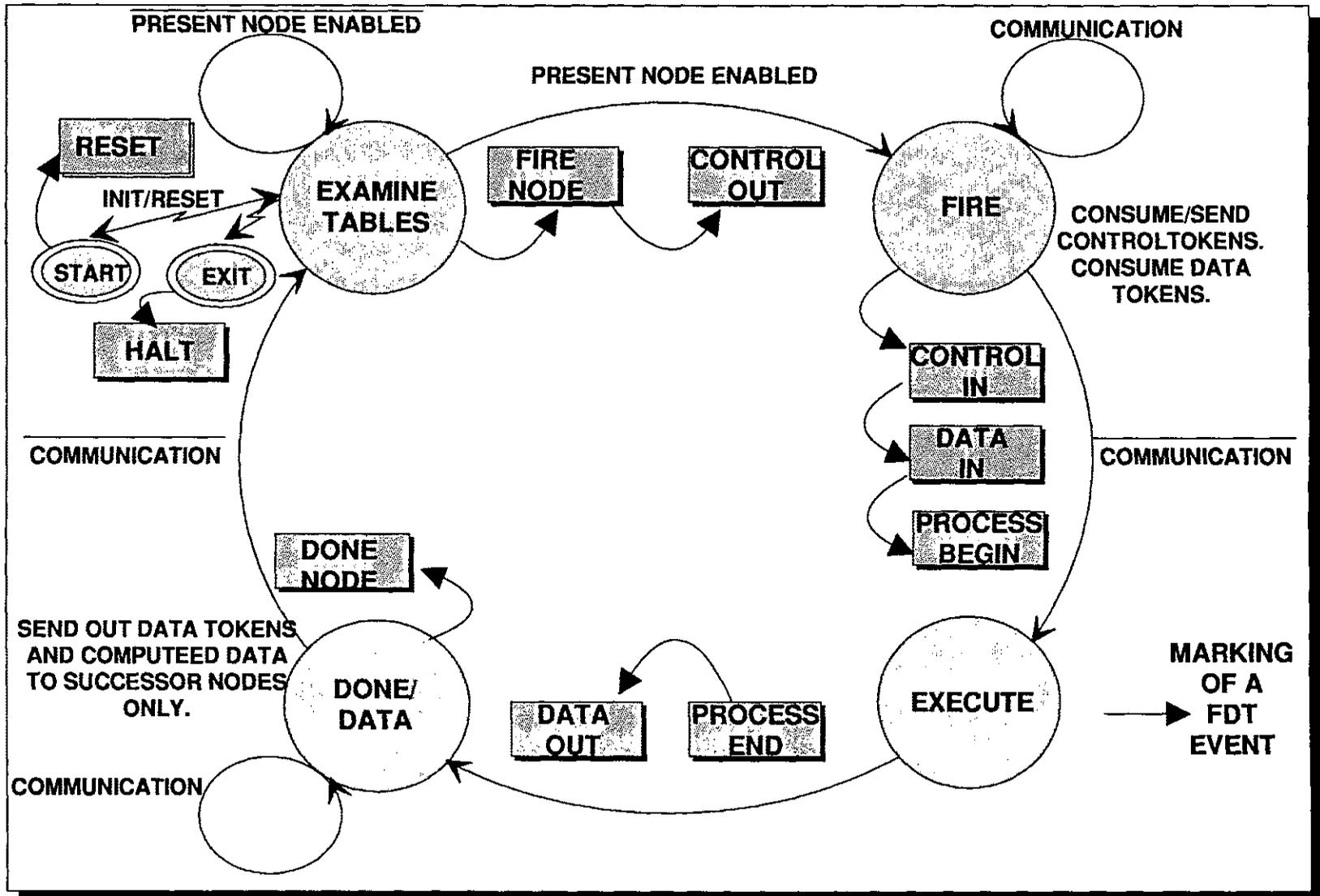


Figure 4.12 FDT Events Associated with Distributed AMOS Operations.

Each communication event (i.e. CONTROL OUT, CONTROL IN, DATA IN and DATA OUT) is associated with the generation and consumption of files. As a result, communication events invoke network file management functions. It has been observed that an atomic time slot of 55ms characterizes network activity. For instance, the generation of a certain number of tokens (files), say  $n$ , usually takes  $n * 55\text{ms}$ . This is the minimum time associated with the generation of  $n$  tokens. If two or more processors submit a network access request at the same time, a collision occurs. However, more often a processor requiring access to a network directory may find a busy network channel. Therefore, a contention for access to network services arises among peer processors. In the event of collisions or contention, a processor requires additional 55ms slots to complete the generation of all tokens. However, when consuming tokens, a processor usually takes only one 55ms slot, since it finds the required tokens(files) in its local directory (which is mapped as a network directory). This may also get aggravated due to contention or collisions. As a result, a processor requires additional time over the ideal node execution time, to complete the communication associated with the execution of a node. The above discussion may be summarized as follows:

$$\begin{aligned} \text{Total Node Execution Time} = \\ \text{Ideal Node Execution Time} + \text{Communication Overhead} \end{aligned} \quad [4.1]$$

$$\begin{aligned} \text{Communication Overhead} = \\ \text{Time to generate control tokens} + \text{Time to consume control tokens} + \\ \text{Time to consume data tokens} + \text{Time to generate data tokens.} \end{aligned} \quad [4.2]$$

Minimum values for the communication components in equation 4.2 may be quantified further as,

$$\begin{aligned} \text{Minimum time to generate control tokens} = \\ \text{Number of Control Tokens Output (NCO)} * 55\text{ms.} \end{aligned} \quad [4.3]$$

Minimum time to consume control tokens=  $1 * 55$  ms. [4.4]

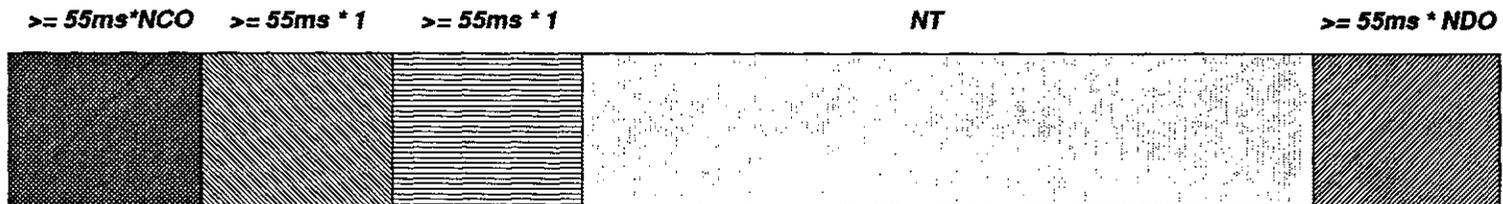
Minimum time to consume data tokens= $1 * 55$  ms. [4.5]

Minimum time to generate data tokens=  
 Number of Data Tokens Output (NDO) \* 55ms. [4.6]

It should be noted that the above time measures are absolute minimums. Due to contention or collisions, any of the above communication events may need additional 55ms network activity slots. Furthermore, the maximum size of an ethernet packet is about 1.5 KBytes. Hence tokens which exceed this size require two or more ethernet packet transmissions. There shall be a corresponding increase in the number of network activity slots required to complete the communication associated with the transmission of multiple ethernet packets. Moreover, the communication overhead increases linearly with the number of tokens that need to be generated per node. Consequently, AMGs bearing nodes with multiple interconnections present a significant communication overhead during execution. A graphical description of the above discussion appears in Figure 4.13.

With reference to the AMG in Figure 4.7 and its corresponding real time events in one TGP frame (as shown in Figure 4.10), it is seen that node zero (on iteration five) consumes a 330ms time slot before being fired. This slot corresponds to the generation of one control token (for node six) and the consumption of two control tokens (from nodes one and two) and one data token (from node six).

Though these events should have taken a minimum of three 55ms slots (thus totalling 165ms), node zero actually takes three additional 55ms slots to complete its operations due to the concurrent network activities of nodes three, four, five and six (pertaining to iteration four). The display bar for node seven (iteration three) shows no network activity at output, since node seven being a sink node does not generate any data tokens.



-  Control Out, NCO : Number of Control Tokens Output.
-  Control In, NCI : Number of Control Tokens Input.
-  Data In, NDI : Number of Data Tokens Input.
-  Process Begin, NT : Node ExecutionTime.
-  Data Out, NDO : Number of Data Tokens Output.

**Minimum Execution Time :**

<b>Time for Generating Control Tokens</b>	<b>NCO</b>	<b>*</b>	<b>55ms</b>
<b>+ Time for Consuming Control Tokens</b>	<b>1</b>	<b>*</b>	<b>55ms</b>
<b>+ Time for Consuming Data Tokens</b>	<b>+ 1</b>	<b>*</b>	<b>55ms</b>
<b>+ Intrinsic Node Execution Time</b>	<b>+ NT</b>		
<b>+ Time for Generating Data Tokens</b>	<b>+ NDO</b>	<b>*</b>	<b>55ms</b>

Figure 4.13 Time Measures Associated With the Execution of a Single Node.

Each of nodes three, four, five and six for iteration four shows a 55ms network activity slot at output, which corresponds to the generation of a data token for node seven. Similarly, nodes one, two and three for iteration five show  $2 * 55\text{ms}$  slots at output since each of them generates two data tokens. Node seven for iteration four shows a CONTROL OUT time measurement of 200ms which equals four 55ms slots. This is correct since node seven generates four control tokens for predecessor nodes three, four, five and six.

In Figures 4.14, 4.15 and 4.16, TGP frames from Figures 4.1, 4.6 and 4.9 respectively, have been magnified in order to display pertinent node execution and communication activities. Execution measurements for these figures have been tabulated in Tables 4.10(a), 4.10(b) and 4.10(c) respectively. These results account for the deviation from ideal behavior seen in the actual execution activity seen in Figures 4.14, 4.15 and 4.16.

The critical circuit (which determines TBO) in the AMG in Figure 2.3(b), contains nodes zero and one. Consequently, in Figure 4.15 TBO gets determined by the execution of nodes zero and one. The ideal TBO for this AMG is 5.0 seconds (5000 milliseconds). Factoring in the communication overhead (determined through using Equations 4.1 through 4.6), a minimum TBO value of 5550 ms is expected. However, the actual TBO turns out to be 5879 ms. This difference is due to contentions which occur due to simultaneous network access requests by peer processors. The figure of 5879 ms is derived by adding the actual cumulative execution times for nodes zero and one, which are 3517 ms and 2362 ms respectively.

In Figure 4.15 concurrent network activity during the "FIRE" state of node zero (iteration nine) is seen due to the "DONE" state of node two (iteration eight) and node one (iteration eight). Similarly concurrent network activity is seen between the "DONE" state of node zero (iteration nine), "FIRE" state of node two (iteration nine), "FIRE" state of node one (iteration nine) and "DONE" state of node four (iteration eight).

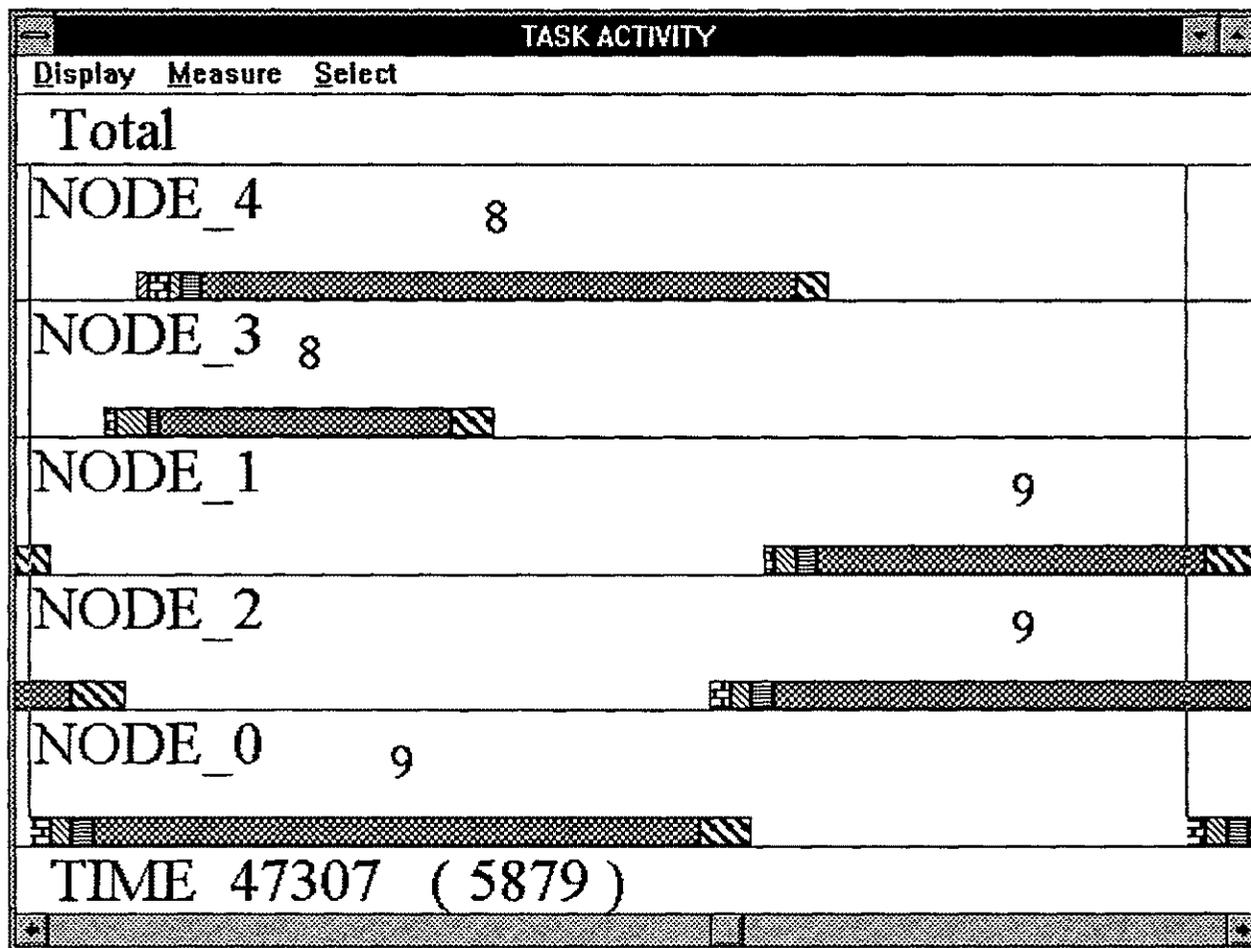


Figure 4.14 FDT Events in One TGP Frame of Figure 4.1.

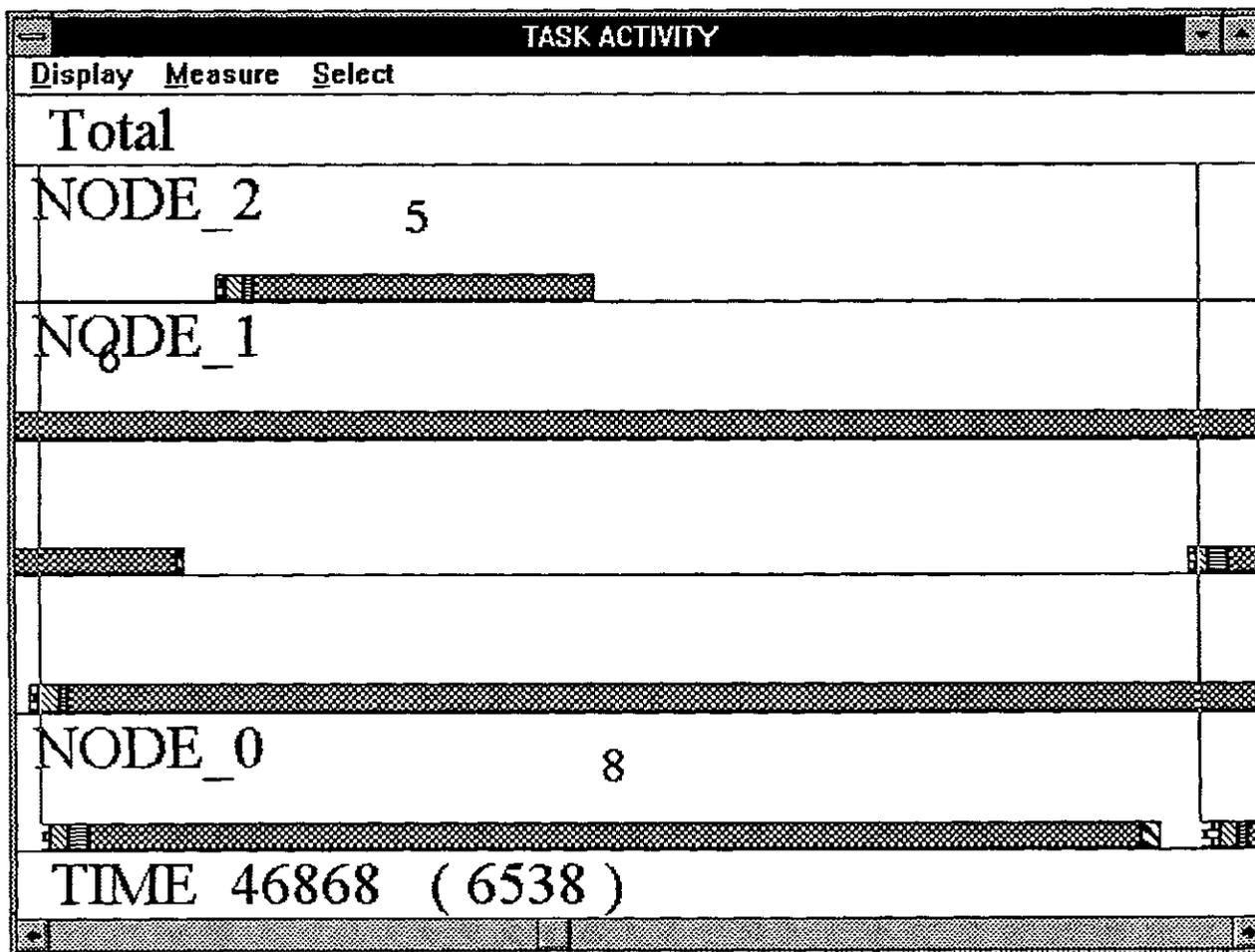


Figure 4.15 FDT Events in One TGP Frame of Figure 4.6.

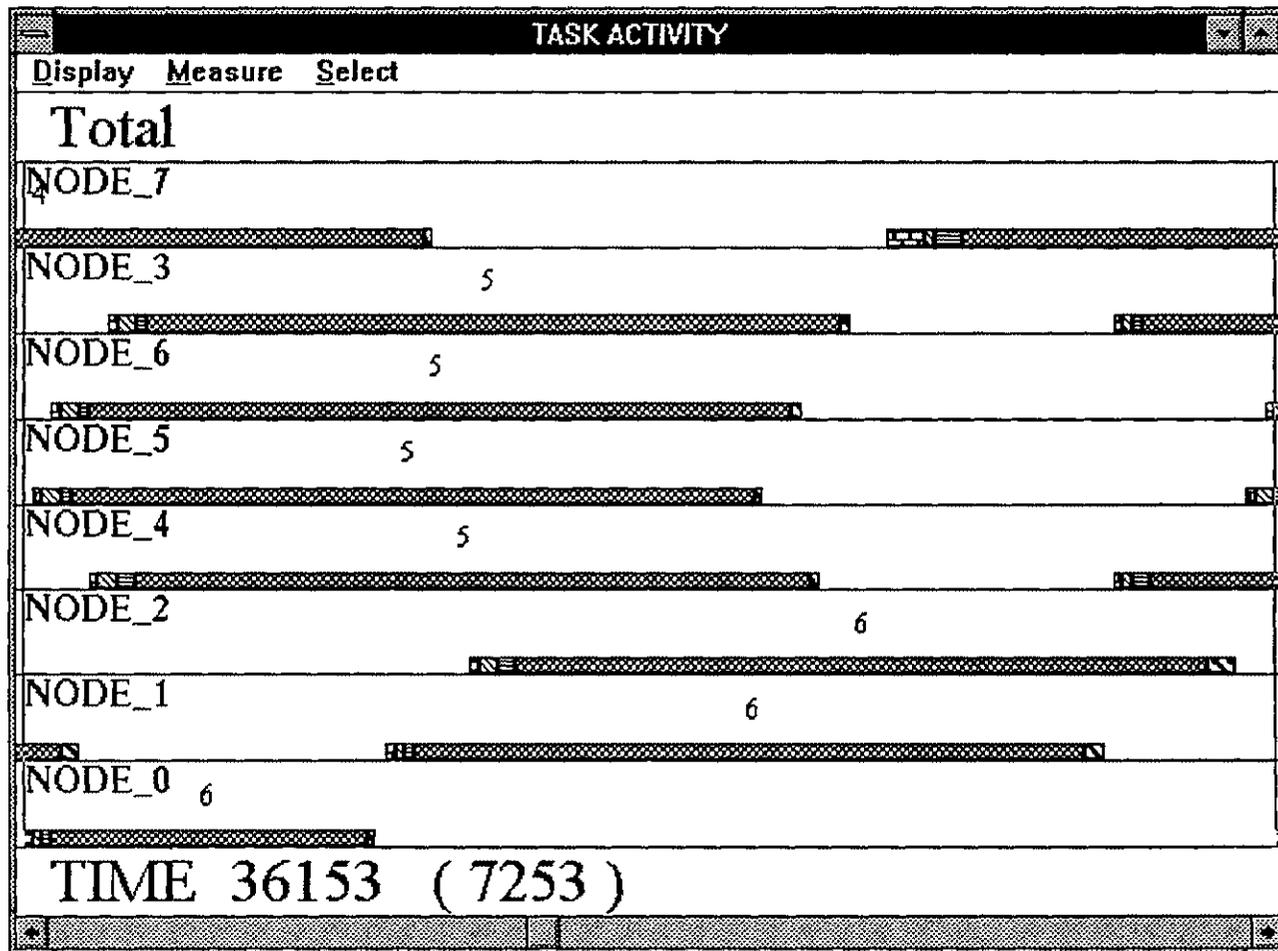


Figure 4.16 FDT Events in One TGP Frame of Figure 4.9

Table 4.10 (a) TBO Measurements for Figure 4.14.											
CCr Node	Tokens & Timing (ms)	Output Ctrl Tokens	Input Ctrl Tokens	Input Data Tokens	Output Data Tokens	Node Time (ms)	Comm. Ov'hd. (ms)	Total Time (ms)	Idle Time (ms) <sup>6</sup>	Min TBO (ms)	Actual TBO (ms)
0	Tokens	1	2	1	2					5550	5879
	Min.	55	55	55	110	3000	275	3275	0		
	Actual	55	110	110	165	3077	440	3517	0 <sup>#</sup>		
1	Tokens	1	2	1	2					5550	5879
	Min.	55	55	55	110	2000	275	2275	0		
	Actual	55	110	110	165	1922	440	2362	0 <sup>#</sup>		
Table 4.10 (b) TBO Measurements for Figure 4.15.											
0	Tokens	1	2	1	2					6275	6538
	Min.	55	55	55	110	6000	275	6275	0		
	Actual	55	110	110	110	5934	385	6319	219 <sup>+</sup>		

\* Also includes the time interval between the termination of the execution of a node and the beginning of the execution of its succeeding node.

+ Experimental value approximated.

# Couldn't be determined accurately due to clock skew.

CCr Critical Circuit

Ctrl Control

**Table 4.10 (c) TBO Measurements for Figure 4.16.**

CCr Node	Tokens & Timing (ms)	Output Ctrl Tokens	Input Ctrl Tokens	Input Data Tokens	Output Data Tokens	Node Time (ms)	Comm. Ov'hd. (ms)	Total Time (ms)	Idle Time (ms)	Min TBO (ms)	Actual TBO (ms)
0	Tokens	1	2	1	2					6605	7253
	Min.	55	55	55	110	2000	275	2275	0		
	Actual	55	55	55	110	1813	275	2088	550 <sup>+</sup>		
2	Tokens	1	3	1	3					6605	7253
	Min.	55	55	55	165	4000	330	4330	0 <sup>+</sup>		
	Actual	55	110	110	165	4011	440	4451	164 <sup>+</sup>		

In Figure 4.14 it apparently appears that node zero (iteration ten) fires before node one (iteration nine) terminates. The critical circuit for the underlying AMG dictates that node zero may be fired only upon the termination of node one (since node zero depends on a data token from node one). The anomalous behavior seen in Figure 4.14 is due to a clock skew between the independent timers maintained by individual processors. In this specific example, a lack of synchronization between the software timers for processor two (which executes node one for iteration nine) and processor zero (which executes node zero for iteration ten) are responsible for the aberrant execution pattern. Also note that a processor may experience some delay in between two consecutive node executions. This delay is largely caused by the time taken by predecessor nodes to produce data tokens and the time taken by successor nodes to produce control tokens. This delay is termed as "idle" time. However, due to the clock skew present in the output shown in Figure 4.14, the idle time could not be measured accurately. The particular TGP frame was specifically chosen to demonstrate the clock skewing effect that may appear in the testbed due to the independent running of timers on peer processors. Note that these timers are synchronized only once at the beginning of testbed operations.

The execution measurements for the AMG in Figure 4.4 is shown in Table 4.10 (b). The self-loop on node zero forms the critical circuit for this AMG. Consequently, ideally the TBO for this example should be the intrinsic execution time for node zero, i.e. 6.0 s (6000 ms). The predicted minimum TBO is 6275 ms. However, due to contention the actual communication delay is 385 ms (instead of the minimum 275 ms). Furthermore an idling time of 219 ms is seen. In other words, processor zero takes 219 ms in between successive iterations of node zero. Note that part of this delay is due to node zero's dependence on a control token from node one. Furthermore the processor takes some amount of time to execute the AMOS functions that need to be performed for task-switching between two successive node executions.

The execution measurements for the eight node AMG in Figure 4.7 is shown in Table 4.10 (c). The critical circuit time is set by node zero and node two, which take 2000 and 4000 milliseconds, respectively. Thus, the theoretical TBO for this example is 6000 ms. The minimum TBO is predicted to be 6605 ms. However, the actual TBO is seen to be 7253 ms. The communication overhead for node zero matches the minimum value of 275 ms. Ideally, node two should begin execution immediately after node zero stops. However due to serial network communication (and resultant contention) an idle time of 550 ms between the termination of node zero and firing of node two is seen. Similarly, an idle time of 164 ms is seen between the termination of node two (iteration six) and firing of node zero (iteration seven).

As might be anticipated, the Time Between Inputs and Output (TBIO) for an AMG also suffers due the communication overhead. The TBIO is set by the time taken by the nodes in the critical path. The deviation of actual TBIO from ideal TBO is due to the communication overheads for each node in the critical path and idle times between node transitions. The TBIO measurements for the AMG in Figure 4.4. is shown in Table 4.11. Figure 4.6 was used to calculate the values for this table. Notice that an actual TBIO of 22913 is seen instead of a minimum TBIO of 22605. However, while considering this value of actual TBIO, the deviation in node execution times should also be taken into account. It may be noted that the total node execution time (21343 ms) for all the three nodes in the critical path is off by 657 ms from the theoretical TBIO of 22000ms. Table 4.11 shows a representative set of calculations. Similar calculations could be performed for the AMGs in Figure 2.3(b) and Figure 4.7.

A final comment shall be made about the initial clock synchronization mechanism. During initialization timers on individual processors are synchronized upon the receipt of a RESET packet from one of the processors (say processor zero). The transmission of RESET packets from processor zero to every other processor is done over the ethernet in a specific sequence. Since each atomic network file transfer takes a minimum of 55ms, it is likely that from the very

**Table 4.11 TBIO Measurements for Figure 4.6.**

CP Node	Tokens & Timing (ms)	Output Ctrl Tokens	Input Ctrl Tokens	Input Data Tokens	Output Data Tokens	Node Time (ms)	Comm. Ov'hd. (ms)	Total Time (ms)	Idle Time (ms)	Min TBIO (ms)	Actual TBIO (ms)
0	Tokens	1	2	1	2					22605	22913
	Min.	55	55	55	110	6000	275	6275	0		
	Actual	55	110	110	110	5934	385	6319	165		
1	Tokens	1	1	1	1						
	Min.	55	55	55	55	14000	220	14220	0		
	Actual	55	55	110	55	13846	275	14121	110		
2	Tokens	1	0	1	0						
	Min.	55	0	55	0	2000	110	2110	-		
	Actual	110	55	110	0	1923	275	2198	-		

Note : In this table the total of actual node times is 21343 ms. Thus the execution time was off by  $(22000-21343) = 657$  ms. This should be taken into account when the value for actual TBIO is used in calculations.

CP Critical Path

beginning processors start their clocks at different relative times. This permanent clock skew could explain the idle time seen between node transitions.

#### **4.5 Summary**

The results pertaining to the distributed execution of a decomposed dataflow algorithm under cyclo-static, block cyclo static and static schedules are presented in this chapter. Graph features such as self loops, forwarded tokens, buffers on CMG control arcs and multiple instantiations of nodes are described through an exhaustive experiment. The execution of an eight node AMG on six processors demonstrates the upper operating limits of the testbed. Communication events occurring in the testbed are quantified using lower bound expressions that describe the minimum time a particular communication event may take. The communication overhead may get further aggravated due to possible contention for network access or collisions due to simultaneous transmissions. It is noted that network activity in the testbed occurs in multiples of an atomic time slot of 55ms. It is also observed that due to a single access ethernet channel, communication for a node grows linearly with the number of data interconnections that the node has with predecessor or successor nodes.

One of the design goals for the ATAMM testbed is to establish an experimental vehicle, that can be used for observing the execution behavior of decomposed large grain dataflow algorithms. The experimental results described in this chapter verify the testbed's adequacy towards meeting this goal.

## CHAPTER FIVE

### CONCLUSION

#### 5.0 Executive Summary

The conceptualization of a strategy for cyclo-static scheduling of deterministic large grain dataflow algorithms and its implementation on an local-area ATAMM multicomputing testbed has been the key objective of this research. It was postulated that the periodic execution of a N node dataflow algorithm in R successive TGP frames could be represented by the periodic repetition of R cyclically shifted threads of a node loop that contains all N nodes arranged in an explicit sequence. Assuming the existence of such node loops, a hypothesis for deterministically mapping dataflow algorithm nodes for relative iteration numbers on processors was advanced. It was proposed that R processors aggregately execute R cyclically shifted threads of a basic node loop to produce the composite execution behavior of a dataflow algorithm. It was observed that the mapping of nodes onto R processors is repeated periodically with a modulo R relationship. Such a N node loop is termed as a cyclo-static schedule loop. It was further established that it may be possible to periodically execute blocks of node loops each of which contain fewer than N nodes. If R such blocks can be identified, the set of R blocks of node loops collectively form a static schedule. If k (where  $1 < k < R$ ) blocks of periodic node loops are identified, the resultant schedule is block cyclo-static.

The postulates of the hypothesis for cyclo-static scheduling were extrapolated to build an information structure that contained details pertaining to CMG attributes, node schedules and initialization information. This information

base was translated into data structures that govern the strategy for distributed AMOS operations.

An ATAMM multicomputing testbed that demonstrates distributed dataflow computations has been developed. A local area personal computer environment was chosen as the hardware for the testbed. Six PC/ATs networked using a 10MBps peer-to-peer ethernet network formed the processors of the multicomputing testbed. The LAN file-management system consisting of local drives of peer processors remapped as globally shared network drives was used to model a distributed shared memory space among processors of the multicomputing testbed. System performance was highly enhanced by utilizing MS-DOS RAM-disks as network drives. Message passing in the system was achieved by communicating files among peer processors.

Experiments that demonstrate each of the three types of scheduling policies, dataflow graph properties and testbed features were conducted. A single AMG was scheduled using cyclo-static, block cyclo-static or static scheduling policies. These scheduling strategies were demonstrated on the testbed. The execution of a graph bearing self loops, forwarded data tokens and control buffers was shown. This particular example also demonstrated the testbed's ability to multiply instantiate nodes for different iterations. A third graph that had eight nodes and required six processors was shown. This graph represents the full limits of the testbed's current capabilities.

The performance of each graph was evaluated by inspecting FDT data on the ATAMM Analysis Tool. Time measurements for FIRE events (such as generation of control tokens, consumption of control tokens and consumption of data tokens), node execution and DONE/DATA events (such as generation of data tokens) were taken. The actual performance of the testbed differed from ideal behavior due to a communication overhead that was generated by the file management system of MS-DOS/network software and single ethernet channel access. The delay caused by this overhead was measured.

Expressions that determine the minimum value of the communication overhead have been specified. It was found that the minimum communication overhead for node operations consists of a 55ms slot for the consumption of all control tokens, another 55 ms slot for the consumption of all data tokens and 55ms slots for every control or data token output. Consequently the minimum communication overhead for a single node comprises of 110 ms for the consumption of control or data tokens and 55ms times the total number of tokens generated by the node. In this sense, the communication overhead depends on the graph topology. Due to contentions and collisions, it was seen that this communication delay got aggravated by additional 55ms slots. However, the communication overhead did not lead to a significant deviation of the testbed operation from expected behavior, since the effect of communication delays manifested in real-time as a "stretching" out of the ideal TGP. This preserved the node-iteration relationships of the AMG, thereby satisfying dataflow and scheduling requirements. As a result the scheduling paradigm was satisfactorily carried out.

Measurements for TBO were taken for each experiment. The deviation of actual TBO values from the minimum were accounted for. It was seen that other than a minimum communication overhead, the time a processor took between two successive node executions and the effect of clock skewing also lead to a deviation in expected measurements.

The LAN system is found to be adequate for establishing the veracity of the distributed scheduling approach. It also forms an economical means to providing an experimental vehicle for future ATAMM research. The modest computational speed of the testbed is not an issue since the strategies for information management and distributed processing in the testbed constitute the ensemble of transportable ideas that can be scaled to adapt to a high performance hardware setting.

## 5.2 Enhancement of Testbed Features

The current implementation of the testbed can be described as a preliminary effort towards defining a strategy for achieving distributed processing on loosely coupled architectures. Several deficiencies of the testbed can be immediately addressed by incorporating features that are available in other embodiments of ATAMM such as the ADM and GVSC. These are:

- [1] At present, the testbed's operations are highly sensitive to faults in system modelling, errors in specification of AMOS data structures, system failures, network errors and AMOS software errors. A method of checking for system errors could be borrowed from the AMOS implementation on the ADM and GVSC. This includes the introduction of a self-examine state in the AMOS state diagram. Such an extended view of AMOS operations is presented in Figure 5.1. The incorporation of this state shall allow a processor to verify its system integrity before reassigning itself for task-execution. The state also provides a means to remove a processor from execution during real-time operation.
- [2] The preliminary experiments run on the testbed use artificial delays to simulate node execution times. The testbed could be used more productively by implementing actual dataflow algorithms.
- [3] The next deficiency of the testbed lies in its usage. Specification files for dataflow problems are currently created manually using a text editor. An enhancement to graph entry tools such as the ATAMM Graph Entry Tool [ANDREWS93] may include the automatic generation of the specification file. Software could be written to investigate the execution characteristics possible for node-sequence that can be identified for a given TGP. This would aid the automatic generation of schedule-loops and other distributed AMOS data.

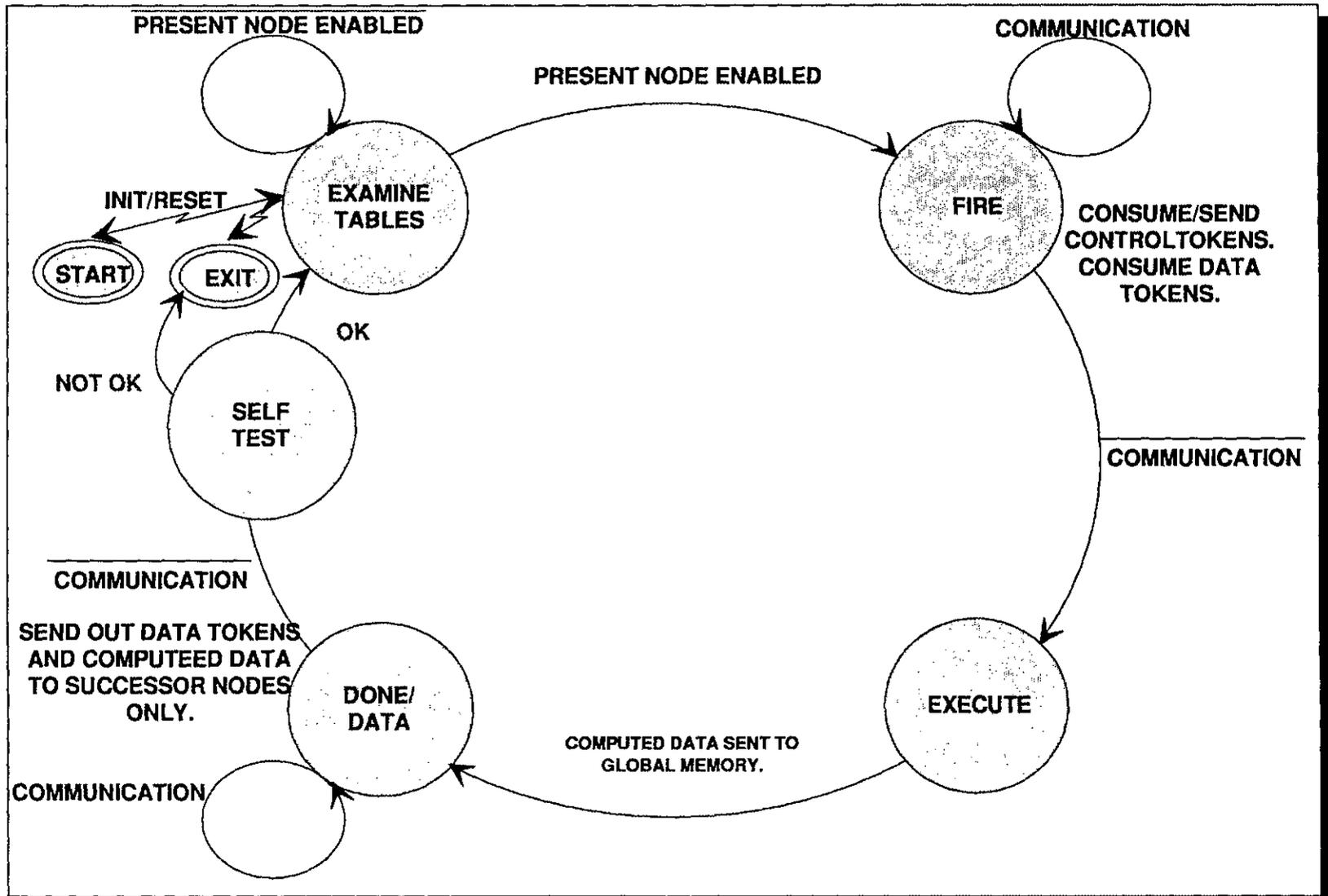


Figure 5.1 Enhanced State-Machine View of Distributed AMOS Graph Manager.

- [4] A certain level of redundancy may be observed in the specifications for distributed AMOS operation. For example, the assignment table could be constructed in software by using the information supplied by the initialization data and next node and next node iteration increment tables. Future versions of testbed AMOS could reduce similar instances of redundant data specification.
- [5] Another physical limitation of the testbed is the size of applications that can be run on it. The current implementation of the testbed incorporates limited size RAM-disks that obviously limit code and data size. This also restricts the amount of FDT data that can be collected for a given problem, thereby limiting the number of iterations for which an algorithm can be repeated. A solution to this problem would be to use powerful processors that offer additional (extended) memory.
- [6] Yet another feature that would make FDT data generation more accurate is a mechanism that periodically synchronizes the individual software timers on peer processors of the system. Mechanisms such as hardware clock triggering, transmission of synchronization packets or time-stamping of control and data tokens could be used to achieve this goal.

### **5.3 Topics for Future ATAMM Research**

The current research may be expanded to encompass new domains of ATAMM enhancement and implementation. The following are identified as potential areas for future research under.

- [1] An adequate proof of the hypothesis for distributed scheduling needs to be constructed. A possible lead in this direction may begin with a proving for a N-node AMG, the existence of  $(N-1)!$  cyclo static schedule loops. It may be further possible to describe the limited

cases of cyclo-stationary scheduling (block cyclo-static and static) as being partitions on a basic cyclo-static node loop.

- [2] CMGs with conditional branches could be handled by the testbed if AMOS is appropriately modified to handle different execution cases arising out of selection of branch conditions. The AMOS specifications file would need to be augmented by the inclusion of an additional sets of data that pertain to each possible execution state.
- [3] A performance bottleneck of the testbed is the single ethernet channel available for communication. It was seen in Chapter Four that simultaneous requests for network access leads to collisions or contention. Consequently Fire and Done states of concurrently executed nodes get delayed while attempting to resolve contention. A network system that could handle shared files and multiple requests is one possible solution to this problem. A method for making IEEE 802.3 Ethernet deterministic is described in [COURT92].
- [4] For certain data intensive applications, it may be more efficient to circulate node code among processors if data size exceeds code size. Future enhancements to the testbed and the ATAMM model in general could incorporate this concept of "code flow".
- [5] The testbed provides an offline means to determining the characteristics of distributed execution of algorithms on the basis of different node schedules. The execution times of algorithm nodes (that are required to be executed on real time dataflow hardware or more powerful computers) may be represented as scaled time delays in the testbed. Thereafter, the testbed provides an economical means to studying the decompositions of dataflow algorithms and their relationships with varying node schedules and processor requirements.

- [6] The continuous processor assignment and deterministic node scheduling approaches suggests the existence of additional marked graph edges which describe the migration of processors between nodes of an AMG. Such edges may be provisionally termed as processor edges and may provide a future enhancement to the ATAMM modelling process.
- [7] The research described in this thesis also suggests a method of using networked computers for distributed dataflow processing. It may be possible to inter-network clusters of networked workstations (that implement a distributed ATAMM algorithm) to form a hierarchy of networked resources that could act as processor nodes of a large distributed dataflow multicomputing system.

In sum, future embodiments of the ATAMM testbed shall be required to improve system performance in all areas of computational activity such as evolving the current ATAMM model, incorporating fault tolerance, improving on communication bottlenecks, including more powerful processors and improving user interfaces for aiding problem specification.

## REFERENCES

- [AGERWALA82] Tilak Agerwala and Arvind, "Data Flow Systems," *IEEE Computer*, February 1982, pp.10-14.
- [ANDREWS93] Asa M. Andrews, CTA Incorporated, VA, *Graph Entry Tool*, Version 2.5.17, 1993.
- [BABB84] R.G. Babb, "Parallel Processing with large grain dataflow techniques," *Computer*, Vol.17, July, 1984.
- [COURT92] R.Court, "Real-time Ethernet," *Computer Communications*, Vol.15, No.3, April 1992.
- [CHASE87] M.Chase, "A pipelined dataflow architecture for signal processing: The NEC  $\mu$ PD7281," in *VLSI Signal Processing*, New York: IEEE Press, 1984.
- [ERCEGOVAC86] Milos D. Ercegovac and Tomás Lang, "General Approaches for Achieving High Speed Computations," *Supercomputers Class VI Systems, Hardware and Software*. S.Fernbach (Editor), Elsevier Science Publishers B.V. (North Holland), 1986, pp.1-28.
- [FURTNEY93] Mark Furtney and George Taylor, "Of Workstations and Supercomputers," *IEEE Spectrum*, May 1993, pp.64-68.
- [HENNESSEY90] John L. Hennessey and David A. Patterson, "Computer Architecture : A Quantitative Approach," Morgan Kaufmann Publishers, Inc., CA 1990.
- [HWANG84] Kai Hwang and F.A.Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, NY, 1984, pp.732-768.
- [JONES90] Robert L. Jones, III, "Diagnostics Software for Concurrent Processing Computer Systems," M. S. Thesis, Old Dominion University, Norfolk, Virginia, April 1990.

- [JONES93] R.L. Jones, III, NASA Langley Research Center, VA, *ATAMM Analysis Tool*, Version 3.1, 1993.
- [LEE87] Edward Ashford Lee and David G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *IEEE Transactions on Computers*, Vol. C-36, No.1., January 1987, pp.24-35.
- [MIELKE88] Roland R. Mielke, John W. Stoughton and Sukhamoy Som, "Modelling and Optimum Time Performance for Concurrent Processing," NASA Technical Paper 4167, Grant NAG1-683, August 1988.
- [MIELKE90] R.R.Mielke, J.W.Stoughton, S.Som, R.Obando, M.Malekpour and B.Mandala, "Algorithm to Architecture Mapping Model (ATAMM) Multicomputer Operating System Functional Specification," NASA Contractor Report 4339, Cooperative Agreement NCC1-136, November 1990.
- [PETERSON81] J.L. Peterson, "Petri Net Theory and the Modelling of Systems," Prentice Hall, Englewood Cliffs, NJ, 1981.
- [RASMUSSEN87] R.D. Rasmussen, G.S. Bolotin, N.J. Dimopoulos, B.F. Lewis, and R.M. Manning, "Advanced General Purpose Multicomputer for Space Applications," *Proceedings of the 1987 International Conference on Parallel Processing*, University Park, PA, USA, August 17-21, 1987, pp.54-57.
- [RISHE91] N.Rishe, D.Tal, S.Navathe and S.Graham, "On Parallel Architectures," *Parallel Architectures*, IEEE Computer Society Press, Los Alamitos, CA, 1991.
- [SCHWARTZ85] D.A.Schwartz and T.P.Barnwell III, "Cyclo-Static Multiprocessor Scheduling for the Optimal Realization of Shift-Invariant Flow Graphs", *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, Tampa, Florida, 1985.
- [SOM88] Sukhamoy Som, "Performance Modelling and Enhancement for ATAMM Data Flow Architectures," Ph.D. Dissertation, Old Dominion University, Norfolk, Virginia, May 1989.

- [SOM90] S.Som, B.Mandala, R.R.Mielke and J.W.Stoughton, "A Design Tool for Computations in Large Grain Real Time Dataflow Architectures," *Proceedings of the IEEE Southeastcon '90*, New Orleans, LA, April 1990.
- [SOM93] S.Som, R.Obando, R.R.Mielke and J.W.Stoughton, "ATAMM: A Computational Model for Real-Time Data Flow Architectures," *International Journal of Mini and Microcomputers*, Vol.15, No.1, 1993, pp.11-22.
- [STOUGHTON86] J.W. Stoughton and R.R. Mielke, "Petri-Net Model for Concurrent Processing of Complex Algorithms," *Proceedings of Government Microcircuit Applications Conference*, San Diego, California, November 1986.
- [STOUGHTON88] J.W. Stoughton and R.R. Mielke, "Strategies for Concurrent Processing of Complex Algorithms in Data Driven Architectures," NASA Technical Paper 181657, Grant NAG1-683, February 1988.
- [STOUGHTON93] Private conversations and technical exchanges with Dr.J.W.Stoughton.
- [TANENBAUM92] Andrew S.Tanenbaum, M. Frans Kaashoek and Henri E. Bal, "Parallel Programming Using Shared Objects and Broadcasting", *IEEE Computer*, August 1992, pp.10-12.
- [TYMCHYSHYN88] William Robert Tymchyshyn, "ATAMM Multicomputer System Design," M. S. Thesis, Old Dominion University, Norfolk, Virginia, August 1988.

