A DATAFLOW MULTIPROCESSOR: PROGRAMMING, SIMULATION AND PERFORMANCE PREDICTION

Pak Kuen Chan

November 1984
CSD-840044

♡ To my Motherland ♡

♡ To my Parents ♡

ACKNOWLEDGEMENTS

Many thanks to the members of the "FP group" at UCLA: Professor Milos Ercegovac, Professor Tomas Lang, Dorab Patel, Jose Arabe, T.M. Ravi, Shih-lien Lu, Paul Tu, Martine Schlag and John Worley for participating in constructive and seemingly endless discussions throughout the development of this thesis.

I wish to express my gratitude to members of my thesis committee, in particular to my adviser Professor Milos D. Ercegovac, for their relentless and patient pursuit of perfection during the preparation of this thesis.

Last but not the least, the work described in this thesis was largely sponsored by the NASA Lewis Research Center grant NAG 3-132 "Dataflow Computing Approaches in High-Speed Digital Simulation", I deeply appreciate their support.

TABLE OF CONTENTS

LIST OF FIGURES

page

ABSTRACT OF THE THESIS

A Dataflow Multiprocessor: Programming,
Simulation and Performance Prediction

by
Pak Kuen Chan
Master of Science in Computer Science
University of California, Los Angeles, 1984
Professor Milos D. Ercegovac, Chair

The research described in this thesis investigates a programming methodology and a dataflow multiprocessor intended primarily for digital continuous-system simulation. The programming approach is based on the functional style language that is good for hierarchical specification of concurrent algorithms, high-level to machine-level code translation, and task partitioning. While functional style languages are conceived by others as a way to organize non-von Neumann type computers, we are primarily interested in using a functional programming language to generate efficient machine task code executed on a network of conventional microprocessors; parallelism is exploited at the machine-level by sequencing the tasks according to the dataflow principles. Restricting the application domain to continuous-system simulation makes implementation of the multiprocessor both pragmatic and feasible. A simulator of the multiprocessor is built to facilitate architecture fine tuning, to experiment with heuristics for partitioning and allocation, and to study various concurrent algorithms. This simulator, together with some analytical performance analysis, helps to justify the programming methodology and the dataflow multiprocessor design.

CHAPTER 1
Introduction

1.1 Problem Statement

This thesis investigates a programming methodology and a multiprocessor system intended for applications such as continuous-system simulation. Continuous-system simulation implements models of *dynamic* systems described by a set of differential equations [Korn78]. The large volume of computation involved makes continuous-system simulation a possible candidate for multiprocessing. In examining the classical numerical methods of solving differential equations, various researchers [Fran78] indicate that parallelism can be both attained at the *equation* level and the *sub-equation* level where arithmetic operations may also be executed in parallel. This suggests that a multiprocessor with a two-level hierarchy may be well suitable for the application.

Contemporary programming methodology [Bueh82, Mako83] for continuous-system simulation may be summarized by Fig. 1.1. The user specification is in some *equation-oriented* continuous-system simulation language such as CSSL. Their efforts involve a *pre-compiler* which converts user specification into FORTRAN code. This is followed by a *parallelizer* which transforms the sequential FORTRAN code into a dependency graph. The dependency graph is further analyzed by a *scheduler* to allocate object code to a multitude of processors.

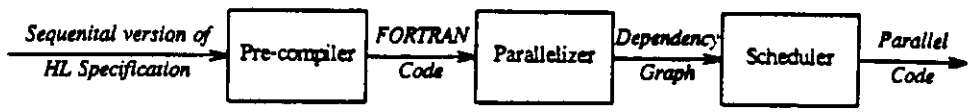

Figure 1.1. Contemporary Programming Methodology in Digital Continuous-system Simulation

Chapter 2 discusses a different programming methodology for the high-level specification of computational algorithms. While the focus is still on continuous-system simulation, it will be illustrated that our approach is applicable to a broader range of numerical problems such as Fast Fourier Transform (FFT) and matrix inversions.

Our approach involves using *Functional Programming Language* (FPL) as the high-level specification of an algorithm. FPL is chosen for its semantic elegance, ease of expressing implicit and explicit concurrency, expandability, and modularity [Back78, Erce83]. The high-level specification is fed into a *symbolic interpreter* [Schl83] to generate a computational graph. The computational graph is subsequently analyzed by a *task partitioning and allocation program* to generate the object code for a proposed multiprocessor. The task partitioning and allocation program attempts to optimize the communication and computation time by partitioning nodes in a computational graph into sets to localize information flow. Fig. 1.2 depicts the methodology.

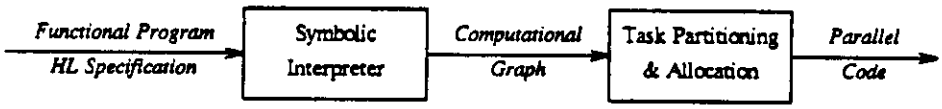

Figure 1.2. A New Programming Methodology

Chapter 3 describes a multiprocessor which executes the computational graphs according to the *dataflow model* [Denn83]. This dataflow model presents no *explicit* constraints on sequencing of execution. Sequencing is done intrinsically and asynchronously by the arrival of data. Accordingly there is no need for a centralized control, and at least in principle it is possible to exploit all the parallelism inherent in the computational graph.

The multiprocessor system is tailored for continuous-system simulation using low-cost implementation. It consists of a network of *processing nodes*, providing concurrent operations at the equation-level. Within each processing node, there are one or more processors to provide parallelism at the sub-equation level. Essentially, each processing node has two sections: one for handling *internode communication* and the other for the execution of object code. These two sections run autonomously thus permitting overlaps between communication and computation. Extension of the system to larger number levels of parallelism, and implementation with off-the-shelf components are also discussed in Chapter 3.

Chapter 4 presents two approaches for the performance analysis of the multiprocessor system. The first approach involves the construction of a *machine-instruction level simulator* to emulate the behavior of the machine. In the second approach, a simple *functional model* based on mean-value arguments is presented to analyze the performance of the machine.

Chapter 5 concludes with a summary on the performance of the machine and suggests directions for further research.

1.2 Assumptions

The work presented in this thesis is supported by a Functional Programming Language symbolic interpreter, a task partitioning and allocation system, and a microscopic simulator. These supporting software are either operational or in the final stage of development at University of California, Los Angeles

[Schl83, Ravi84].

The proposed programming methodology is restricted to *structurally determinate** and computational intensive type problems. Problems that involve data (symbolic) manipulations to be determined by input *values* are out of the scope of this context.

The proposed multiprocessor is tailored for continuous-system simulation. It is not intended to compete with general-purpose dataflow machines such as the Manchester, Dennis' or Arvind's machines [Gurd80, Denn83, Arvi78]. There is also no provision for handling large data bases or data structures; tokens are assumed to be scalars.

At the implementation level, the proposed multiprocessor requires large (compare with von Neumann type computers) amount of random-access memories and dual-port memories to achieve high speed computation. It is therefore assumed that the cost of memories is economical enough to make the implementation of the proposed multiprocessor system feasible.

Study has shown that machine performance is largely influenced by the quality of task partitioning and allocation strategy [Demi82]. Therefore, any conclusion drawn about machine performance while ignoring task partitioning and allocation strategies may be deceptive. Since this thesis does not cover task partitioning and allocation strategies, the performance analysis presented in Chapter 4 is by no means extensive or complete. The intention of Chapter 4 is to provide an estimate of the expected performance and to set up an environment for further investigation and study of the proposed system.

---

* Informally, structurally determinate programs have the property that the sizes (form) of data structures used in the program depend solely on the *form* or *structure* of inputs and not on the *value* of inputs. An example of structurally indeterminate programs is *factorial(n)*. When recursively defined, the stack size required to evaluate *factorial(n)* depends on the *value* of the input *n*.

CHAPTER 2
A Programming Methodology

2.0 Overview

This chapter presents a methodology for generating the object code for a multiprocessor. The source code is given in a Functional Programming Language (FPL) [Back78]. The object code consists of concurrent sequential tasks represented in machine-level code; the tasks are synchronized according to the dataflow principles [Denn83]. The translation from the source language to the machine-level code is done in two steps. First, by a symbolic interpreter which exercises the algebraic constructs, detects parallelisms and dependencies, and extracts primitives defined at the user-defined level of abstraction [Schl83]. Second, the interpreted source which corresponds to a directed graph whose nodes are primitive operations, is passed to a task partitioning and allocation program [Ravi84] to produce machine-level code to be subsequently run on the dataflow multiprocessor.

2.1 Basic Assumptions

This section outlines the assumptions that we make with respect to the application environment of the system, the high-level language, and the mapping from the high-level language to the machine-level language.

This methodology is best suited for scientific or numerical type computations which are characterized by having high computational requirements and *static* program and data structures. Particularly for an application such as continuous-system simulation, the simulation model of the system, which is a set

of differential equations, is well specified before hand. It is usually the case to study the effect of system outputs upon variations in the initial conditions and system parameters. Consequently, the same simulation model is rerun (multiple runs) a number of times with different initial model parameters and/or initial conditions. Only in rare occasions will the *structure* of the system model be changed from simulation run to simulation run [Erce84, Korn78]. The machine-level code generated by this methodology is assumed to be run frequently enough so that the overhead in the extensive analysis of the source language can be offset by the benefits of the efficient machine-level code. It is our belief that in continuous-system simulation the functional style programming makes the analysis more manageable.

The implementation of the proposed multiprocessor system is based on commercially available microprocessor components. The machine-level language of the multiprocessor is therefore of conventional type. The system organization is discussed in detail in Chapter 3.

2.2 The High-Level Language

We employ Functional Programming Language (FPL) as the high-level language. It is essentially the one proposed by Backus, known as Backus' FP. The FPL that we use differs from Backus' FP in minor syntactic details. FPL is chosen for its semantic elegance, ease of expressing implicit and explicit concurrency, expandability, and modularity [Back78, Frie78]. Here, we give an informal description of the language, details can be found in [Worl84].

Functions and Functional Forms	Description
k	selects the k'th element of a sequence.
id	returns the object itself.
tail	returns the tail of a sequence, (all but head).
front	returns the front of a sequence, (all but last element).
last	returns the last object of a sequence.
atom	is true if the object is an atom, false otherwise.
null	is true if the sequence is the null sequence, false otherwise.
length	returns the length of the sequence.
+	returns y+z.
−	returns y−z.
*	returns y*z.
/	returns y/z.
and	returns y AND z.
or	returns y OR z.
not	returns NOT x.
transp	returns the transpose.
eq	is true if y=z, false if not.
gt	is true if y>z, false if not.
lt	is true if y<z, false if not.
distl	distribute left element to all elements in x.
distr	distribute right element to all elements in x.
apndl	appends first element to front of sequence.
apndr	appends last element to front of sequence.
concat	concatenates sequences to form one sequence.
pair	forms sequence into sequence of pairs.
split	splits a sequence into two halves.
rotatel	rotates the sequence circularly to the left.
rotater	rotates the sequence circularly to the right.
reverse	reverses the order of objects in a sequence.
@	routes outputs of function into the input of the next.
[ ]	forms sequence of results of many functions applied to x.
if then	conditionally returns the result of one of two functions.
%y	returns y.
!	recursively applies a function into a sequence x.
|	recursively applies a function into binary tree x.
&	applies the function to all objects in x.

Table 2.1. Brief Description of FPL

A functional program consists of objects, functions, functional forms, definitions and a single operator called "application".

a. **Objects**

Objects are numbers, symbols, or sequences. A sequence $<x_1,x_2, \cdots ,x_n>$ of objects consists of $x's$ which are either numbers, symbols, or sequences. Notice that the notion of "object" is recursively defined; we refer to objects which appear at the first level of recursion as *top level elements*. Objects which are not sequences are referred to as *atoms* (except the empty sequence < > which is considered both as an atom and a sequence).

b. **Application**

This is the operation of applying a function to an object. For example, to apply the addition function (+) to the object <1,2> we write +:<1,2> = 3.

c. **Functions**

Functions are applied to objects and they produce objects as results. Most of the functions operate on sequences. Other functions operate directly on atoms.

Table 2.1 contains an informal description of each of the functions used in FPL.

d. **Functional Forms**

Functional forms are function-forming expressions. They are used to combine existing functions to form new functions. Examples of functional forms are *composition* (@), *construction* ([ ]), *conditional*(**if then**), *constant*

(%), *insert* (!), and *apply-to-all* (&) (see Table 2.1). Functional forms offer explicit parallelism to the language which makes it attractive for the high-level specification of algorithms.

1. *Construct functional form*

   allows parallel application of several functions across the same object, i.e.

$$[f_1,f_2, \cdots ,f_n]:x = <f_1:x,f_2:x,f_n:x>$$

2. *Apply-to-all functional form*

   allows parallel application of a function f to each top level element in an object, i.e.

$$\&f:<x_1,x_2, \cdots ,x_n> = <f:x_1,f:x_2, \cdots ,f:x_n>$$

3. *Linear insert functional form*

   allows a simple representation of pipelined operation on a homogeneous structure, i.e.

$$!f:<x_1,x_2, \cdots ,x_n> = f:<x_1, !f:<x_2, \cdots ,x_n>>$$

4. *Associative insert functional form*

   allows a simple representation of parallel operations by a binary tree, i.e.

$$|f:<x_1,x_2, \cdots ,x_n> = f:<|f:<x_1, \cdots ,x_{\lceil n/2 \rceil}> , |f:<x_{\lceil n/2 \rceil}, \cdots ,x_n>>$$

e. **Definitions**

Definitions define new functions in terms of existing ones. Example,

define *f(object names)* $g@h$ **end**

means that the atomic function f is to stand for the composition (@) of g and f, namely, $g@h$. The *object names* is a list of zero or more mnemonic names. They allow the user to give a mnemonic name to each object in the input sequence.

2.3 Functional Program Languages and Dataflow

Even though we are using FPL as the high-level language, we are not building an FP machine [Mago80, Trel80, Pate81, Kell83] to directly execute the language. Instead, we first translate the FPL specification to a computational graph; then we represent the computational graph using conventional instruction sets (according to the dataflow principles). The reasons for *not* employing a direct execution approach on the FPL specification are discussed next.

It is observed that when a user specifies an algorithm in FPL, he somehow unavoidably uses a large amount of *routing functions* and *functional forms*, along with arithmetic operations, to describe the algorithm [Worl83]. In FPL, *functional forms* are used to express dependency or explicit concurrency. They are useful for analysis and exploitation of parallelism. *Routing functions* are functions which do not change the data values of the object operands, but alter the objects' structure. They are used to channel data to the processing sites, where the data values are processed.

In view of the above facts, if the nature of the computation problem is relatively static, *routing functions* and *functional forms* seem to be superfluous at run time. In the context of *structural* and *computational* behavior, routing functions and functional forms portray the structural information (the connectivities among nodes of the computational graph), while the arithmetic primitives exhibit

the behavior of the computational problem.

In order to achieve the goal of generating efficient run-time machine-level code from the FPL specification, structural information is identified and analyzed before run-time by a technique we term *symbolic interpretation*. This is possible if the application environment (e.g. production type) and the data structures of the input object are static.

2.4 Code Generation Through Symbolic Interpretation

It is assumed that the structure of the input object to the FPL specification is known. The FPL specification is symbolically interpreted on the symbolic input to obtain a graph representation of the computation. This symbolic interpretation effectively resolves the routing functions and functional forms employed in specifying the algorithm. The result of interpreting the routing primitives is manifested in the topology of the graph, while the arithmetic (computational) primitives which can not be symbolically interpreted are retained as nodes in the graph. The actual computation of the primitives is deferred until the values of the arguments are known at run-time. In essence, *most* non-numerical operations are pre-processed by the symbolic interpreter to achieve run-time efficiency. In the context of symbolic interpretation, the FPL primitives can be classified into the following categories.

a. *Computational Primitives*

A computational primitive is represented as a node in the computation graph. Any function can be defined as a computational primitive, thus allowing the computation to be represented at different levels of abstraction. This provides a facility for hierarchical decomposition and partitioning, and enables primitives to match the level of computation supported by

the machine architecture.

b. *Routing and Control Primitives*

This type of primitive can be symbolically exercised since it affects the structure of the FPL object to which it is applied and this effect is determined solely by the structure of that object. Although these primitives do not appear as "nodes" in the graph, they are represented in the topology of the graph whenever they affect the relative positions of the atoms within an FPL object.

Example

As a simple example of symbolic interpretation, consider the following FPL program to evaluate inner product:

define *IP*(*vectors*) (!+)@(&*)@*transp*@*vectors* **end**

To see the symbolic interpretation in action, take the following symbolic input as an example and apply *IP* to this FPL object (with functions * and + tagged as primitives):

$$<<a_1,a_2,a_3> <b_1,b_2,b_3>>$$

1. Composition gives:

$$(!+):((\&*):(transp:<<a_1,a_2,a_3>,<b_1,b_2,b_3>>$$

2. Transpose gives

$$(\&+)@(\&*):<<a_1,b_1>,<a_2,b_2>,<a_3,b_3>>$$

3. Apply-to-all gives

$$(!+):<*:<a_1,b_1>,*:<a_2,b_2>,*:<a_3,b_3>>$$

4. Insertion gives

$$(!+):<g_1,g_2,g_3>$$

then

$$+:<g_1,+:<g_2,g_3>>$$

another addition gives,

$$+:<g_1,g_{23}>$$

and, finally,

$$g_{123}$$

where g_1, g_2, g_3, g_{23} and g_{123} are intermediate variables assigned by the symbolic interpreter.

opcode	*operand*$_1$	*operand*$_2$	*destination*
*	a_1	b_1	g_1
*	a_2	b_2	g_2
*	a_3	b_3	g_3
+	g_2	g_3	g_{23}
+	g_1	g_{23}	g_{123}

Table 2.2. Intermediate Machine-Level Code for Inner Product Example

The intermediate machine-level code generated by the interpreter is shown in Table 2.2. The final machine-level code, which depends on the allocation strategy and the number of available processors, has to be generated by the task partitioning and allocation program.

Notice that after the symbolic interpretation, five simple instructions are sufficient to represent the inner product algorithm. In the context of dataflow operation, the variables appear in the instructions as tokens; instructions are nodes and the connectivity of the graph is implicitly defined by the flow of tokens. Functional forms and routing functions have been exercised by the interpreter to achieve run-time efficiency.

2.5 Levels of Abstraction, Recursion and Parallelism

This section highlights some of the features that are offered by our approach besides efficiency. We demonstrate that any function can be defined as a computational primitive, thus allowing the computation to be represented at a desired level of abstraction. Particularly, it is demonstrated that algorithms which are coded in a recursive programming style are completely unfolded (provided that the predicate of the recursion is based on the structure of the FPL object). This methodology encourages good and efficient program structure and permits effective analysis and transformation before the machine-level code is executed.

a. *Levels of Abstraction*

Fig. 2.1 shows the high-level specification for an 8-point Fast Fourier Transform (FFT). Fig. 2.2.a depicts the computational graph generated with the functions *, /, + and − tagged as computational primitives; Fig. 2.2.b depicts the computational graph generated with the functions *butterfly* being tagged as computational primitives.

As far as task partitioning and allocation is concerned, it is obvious that Fig. 2.2.b offers a much less complicated graph than Fig. 2.2.a. Task partitioning can be performed on the function *butterfly*, and then on Fig. 2.2.b. Though Fig. 2.2.a is fully decomposed, it is unstructured and

complicated, whereas, Fig. 2.2.b reflects the modularity that can be attained from the FFT algorithm.

Comparing Fig. 2.1 with Fig. 2.2.b, we point out that the routing primitives (select, transp, split) and functional forms (constructs, compose) which appeared in the high-level specification have been mapped into the topology of the computational graph.

b. *Recursive Style Programming*

Recursive style programming enables a compact, easy to analyze and elegant specification of an algorithm. However, a direct execution of the program might result in loss of potential concurrency and incur overhead in the dynamic invocation of the functions. As a matter of fact, most reduction machines execute recursive definitions by unfolding them at run-time. In some sense, recursive definitions are treated iteratively [Pate81].

Fig. 2.3 shows a recursive specification of an algorithm for the forward triangularization of a system of linear equations. Fig. 2.4 depicts the computational graph resulting from symbolic interpretation of the recursive definition on a 5x5 matrix. We can see that implicit computations and function invocations have been explicated.

Notice that symbolic unfolding of the recursive definition is possible only if the predicates, which determine when to terminate the recursion, are structurally dependent. That is, the predicates must consist of one of the functions: *atom*, *null* or *length*. Incidentally, value-dependent recursive definitions are being treated iteratively by the symbolic interpreter. For example, if we have a recursive definition like

define $f1()$ **if** p **then** $f2$ **else** $f1 \,@\, f3$ **end**

where p is a value-dependent predicate, then p cannot be symbolically interpreted. It will be retained as a predicate in the computational graph, as depicted in Fig. 2.5.

```
Fast Fourier Transform (FFT)
input structure: pairs of complex numbers
e.g. <<x1 y1> <x2 y2> ... <x8 y8>>
```

```
define  fft( )
        Bitrv @ fftstages
        end

define  fftstages( )
        if eq@[ length,%2 ] then W
        else (& fftstages)@split@concat@Bfly@concat@(&W)@Bfly
        end

define  Bfly( )
        concat@1@[[shuffle@[1,3],shuffle@[2,4]]@concat@(&trans)@split@pair]
        end

define  Bitrv( )
        if eq@[ length,%2] then id else (&Bitrv)@trans@pair
        end

define  W( )
        [cadd,csub]@[1,cmul@[2,[%w,%w]]]
        end

define  cadd( )
        (&+) @ transp
        end

define  csub( )
        (&−) @ transp
        end

define  cmul( )
        [−@[*@[1@1,1@2],*@[2@1,2@2]],+@[*@[1@1,2@2],*@[2@1,1@2]]]
        end
```

Figure 2.1. 8-point FFT in FPL

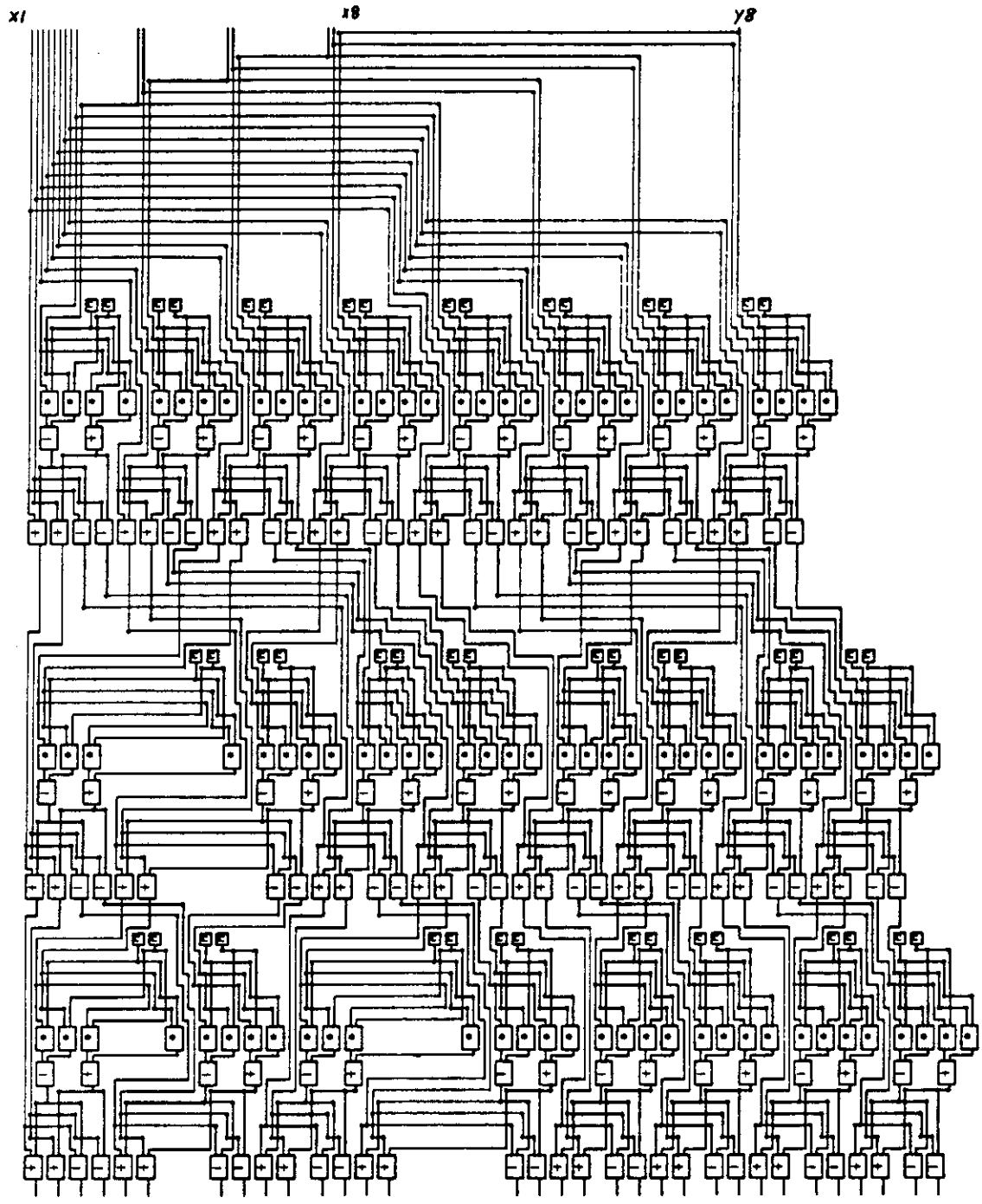

Figure 2.2.a. Computational Graph of FFT Algorithm (Low-level)

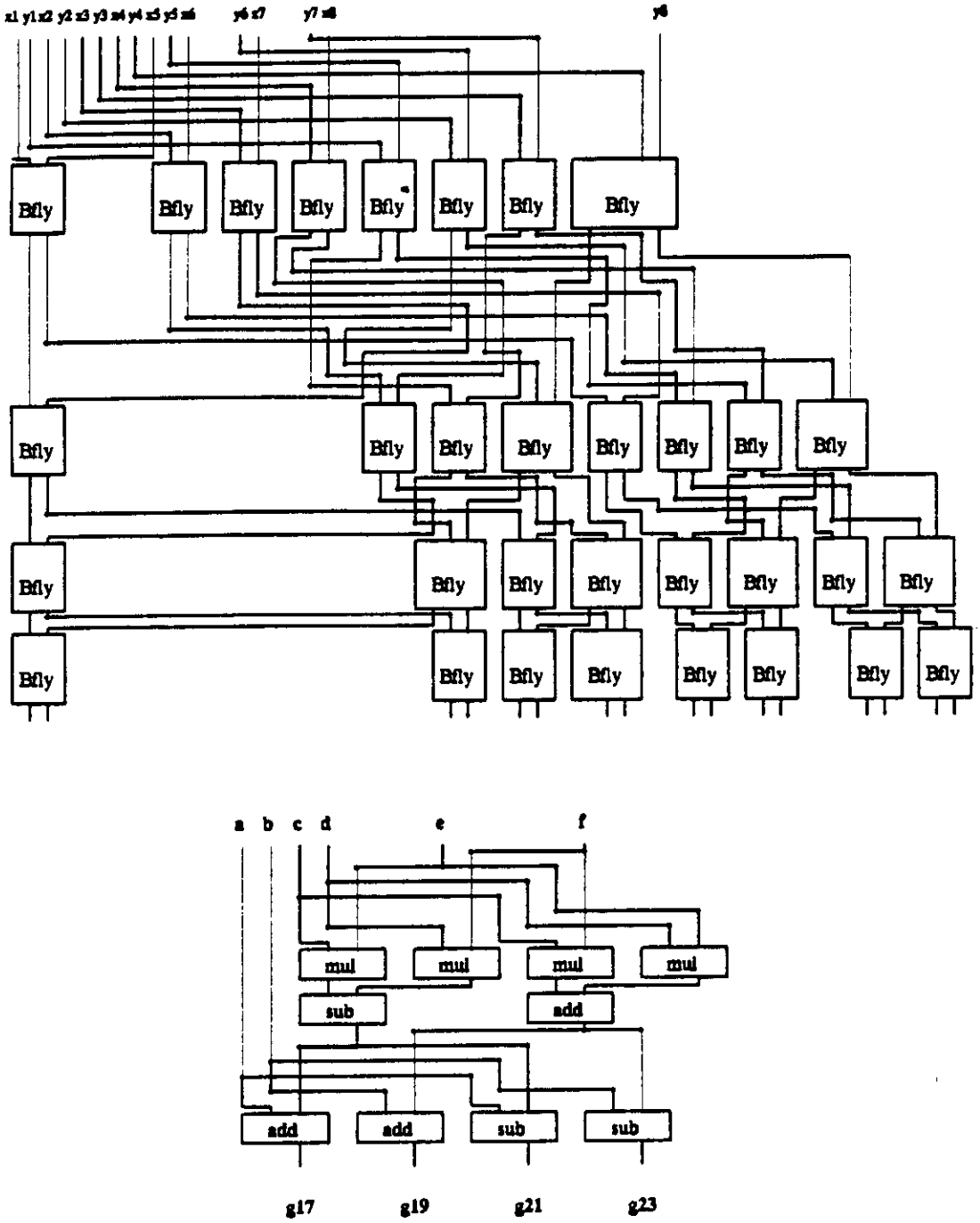

Figure 2.2.b. Computational Graph of FFT Algorithm (High-level)

Forward Triangularization in FPL
input structure: martrix eg. 2x2

<<a11 a12 b1> <a21 a22 b2>>

```
define  gel( )
        if (eq @ [length, %2])
                then    [[/@1]]
                else    tf2@tf1
        end

define  tf1( )
        [norm@1,rte@tail]
        end

define  norm( )
        &/@distl@[1,tail]
        end

define  rte( )
        &(distl@[1,tail])
        end

define  tf2( )
        apndl@[1,gel@&((&elim)@transp)@distl]
        end

define  elim( )
        -@[2@2,*@[1,1@2]]
        end
```

Figure 2.3. Forward Triangularization in FPL

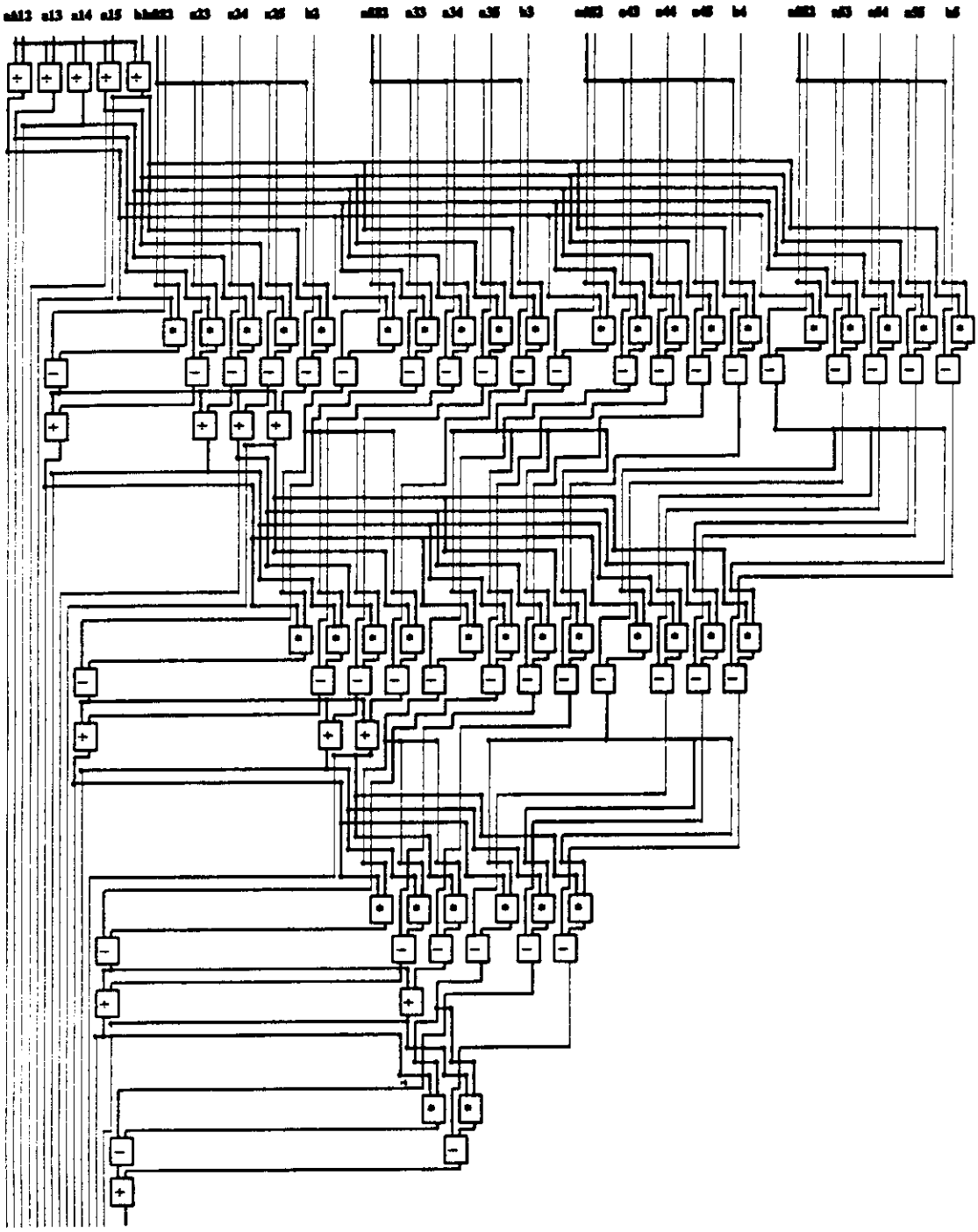

Figure 2.4. Computational Graph of Forward Triangularization

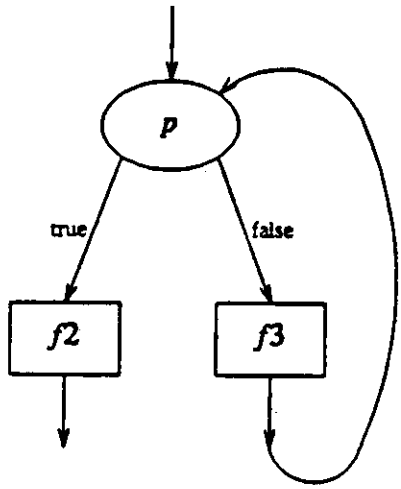

Figure 2.5. Treatment of Value-dependent Conditional

2.6 A Continuous-system Simulation Example

As an example of application of FP to continuous-system simulation, we consider a dynamic system consisting of two coupled pendulums, as exemplified in Fig. 2.6. The mathematical model of the system is described by two ordinary differential equations (ODE's), eqs. (2.1) and (2.2). This system will serve as an example for the remainder of the presentation.

$$\frac{M_1}{\mu}\cdot L^2\frac{d^2\theta_1}{dt^2} + M_1\cdot G\cdot S_1\cdot\sin\theta_1 + SLEN\cdot K\cdot(y-X) + D\,\frac{d\theta_1}{dt} = 0 \qquad (2.1)$$

$$\frac{M_2}{\mu}\cdot L^2\frac{d^2\theta_2}{dt^2} + M_2\cdot G\cdot S_2\cdot\sin\theta_2 - SLEN\cdot K\cdot(y-X) + D\,\frac{d\theta_2}{dt} = 0 \qquad (2.2)$$

where,

G :	the gravitational constant
W_1 :	M_1·G (M_1 : mass of the right pendulum)
W_2 :	M_2·G (M_2 : mass of the left pendulum)
θ_1 :	the displacement of W_1 from the normal
θ_2 :	the displacement of W_2 from the normal
S_1 :	the length from pivot to the center of gravity of right pendulum
S_2 :	the length from pivot to the center of gravity of left pendulum
SLEN :	the distance from ceiling to spring
L :	the distance from ceiling to the bottom of the pendulum
D :	the distance between two pendulums
X :	the unstretched length of the spring
y :	the length while in motion
K :	spring constant
μ:	the friction coefficient

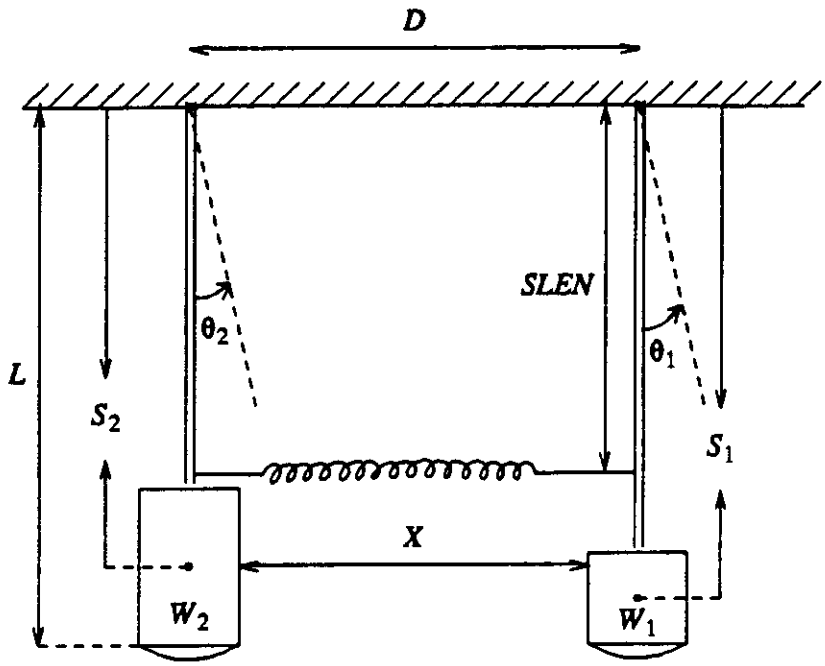

Figure 2.6. Physical set up of the Coupled Pendulums

The solution of the ODE's is obtained using Euler's method. The algorithm, specified in FPL, is given in Appendix A. The computational graph resulting from the high-level symbolic interpretation of the FPL program is depicted in Fig. 2.7, while the low-level symbolic interpretation is shown in Fig. 2.8. More will be said on this example when we come to Chapter 4.

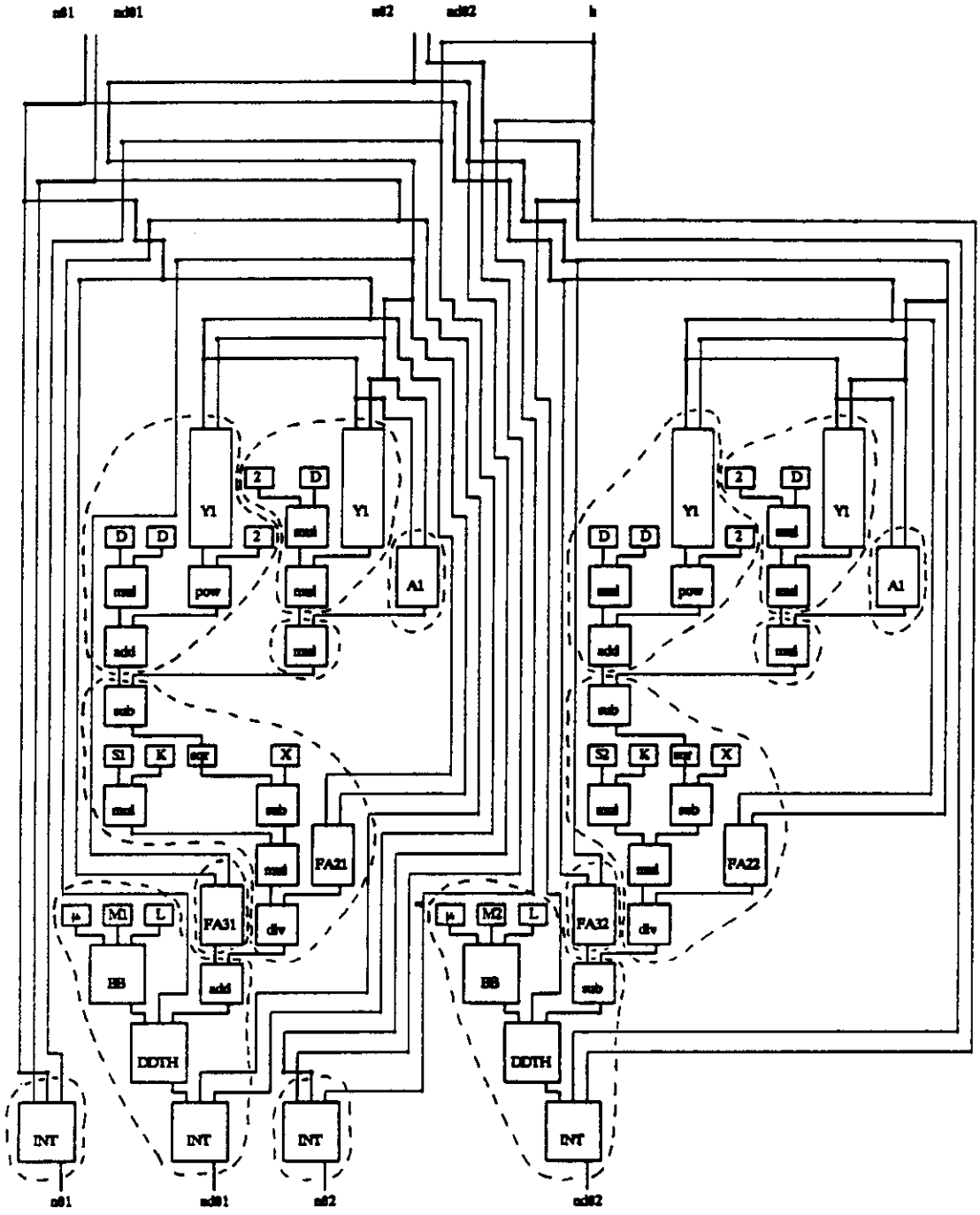

Figure 2.7. Coupled Pendulums Example (High-level Computational Graph)

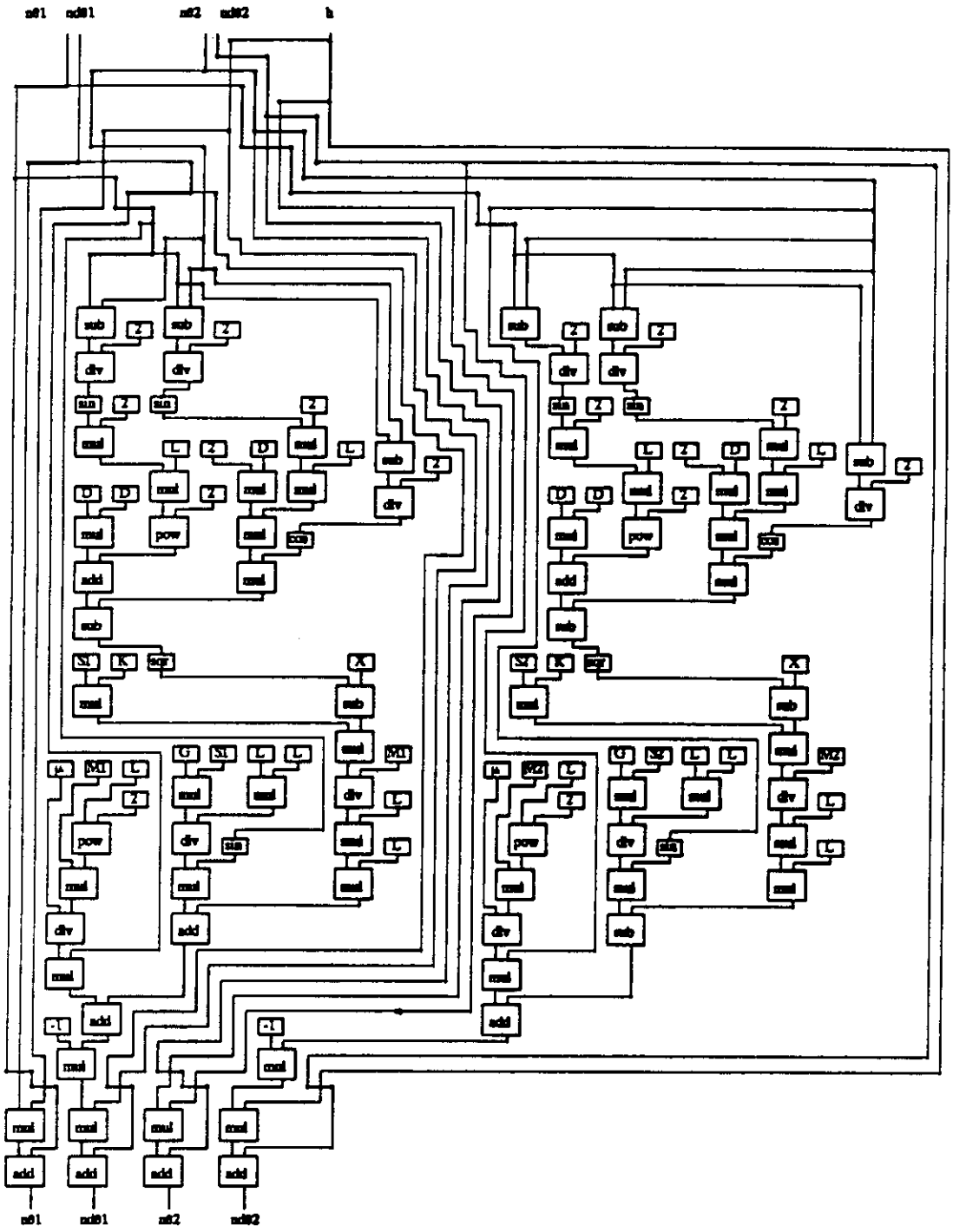

Figure 2.8. Coupled Pendulums Example (Low-level Computational Graph)

CHAPTER 3
The Machine Organization

3.0 Overview

In Chapter 2 we described a methodology for translating the high-level language into computational graphs. This chapter describes a multiprocessor which executes the computation graphs according to the dataflow model.

The design of the multiprocessor system is influenced by two factors: first, the characteristics of the continuous-system simulation, and second, the goal of practical implementation. Since the nature of the simulation problem is relatively static, static partitioning and allocation of tasks are possible and desirable. Processors in the system are connected via a broadcasting interconnection network. This interconnection network is readily implementable using standard components. Its use, however, is limited to a relatively small number of processing nodes (8-16). Section 3.3 describes a hierarchical interconnection scheme to incorporate a larger number of processing nodes. The goal of practicality is achieved by using largely off-the-shelf components such as standard microprocessors and memory modules to realize the system.

3.1 The Dataflow Task Sequencing Model

The basic notion in the this model is (computation) dataflow graphs. In a dataflow graph, the nodes represent functions and the directed arcs represent data dependence between functions. Values are represented as tokens on the arcs. A node can be fired as soon as all the tokens required by the node are

available. This basic model has two important consequences. First, sequencing is done by the flow of data in an asynchronous manner; there is no need for a centralized control. Second, in principle, it is possible to exploit all the parallelism inherent in the computation graph.

The programming methodology presented in Chapter 2 essentially compiles a high-level specification into a dataflow graph. Computational primitives and recursions are the only constructs left after compilation. Structurally dependent recursive definitions are statically unfolded whereas value-dependent recursive definitions are manifested as loops (see Fig. 2.5).

There are two ways to handle *loop* structures in the context of dataflow: the *static* dataflow and *dynamic* dataflow approach [Toko83, Gajs82]. In the static dataflow approach, computations are represented by directed *acyclic* graphs, each iteration being described by a separate graph. In principle, this approach can exploit the maximum parallelism that exists in the algorithm. The main problems with the static dataflow approach is that it demands the iteration depths to be known before run-time, and code has to be replicated for each iteration. For the dynamic dataflow approach, computations are represented by directed *cyclic* graphs. That is, loop structures (schema) do exist during run-time. A common way to differentiate tokens from different iterations is by appending instantiation labels to tokens [Arvi78, Gajs82]. This permits the use of reentrant code and enables an iteration to proceed before the preceding one has finished. However, this tagged token approach presents difficulties for implementations, particularly in relabelling and matching of tokens during run-time.

In attempting to balance the difficulties and the benefits of the static and dynamic approaches, we combine the two approaches and apply them in different domains. We use the concept of dynamic dataflow in the *logical* domain: tokens are labelled and reentrant code is used. Consequently, there is no need to replicate code for each iteration. Each token is represented by a triple $v.x.j$; where v is the data value, x is the unique identifier for the token and j is the generic

instantiation number [Arvi82]. When j is negative, the token belongs to a task which is outside a loop, otherwise the task is in a loop. Nested instantiations are incorporated by appending more than one instantiation number to the token. Hence, a token may look like $v.x.i.k.l$ where i, k and l are instantiation numbers. An instantiation number is immediately appended to a token when it first enters a loop (by an "A" operator), and is automatically detached when it exits from the loop (by a "D" operator), see Fig. 3.1.

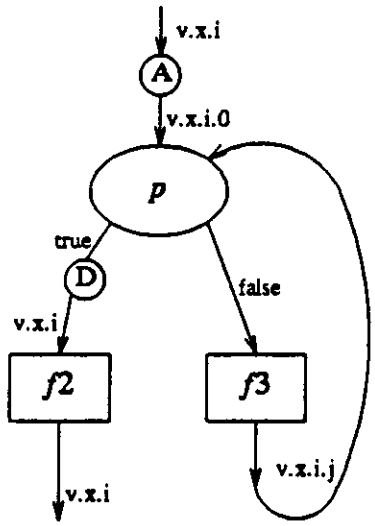

Figure 3.1. Tokens Labelling/Relabelling in a Loop

In the *physical* domain, we use the static dataflow approach. We take advantage of the static nature of our main application - continuous-system simulation, where iteration depths are known before run-time. Consequently, instead of matching tokens in an associative fashion we reserve predetermined and dedicated memory locations to deposit each token, and use dedicated counters for each task to record the arrivals of tokens. Section 3.2.1 will elaborate this point.

A static partitioning and allocation scheme is used to analyze the dataflow graphs before run-time. This scheme attempts to optimize the execution and communication time by partitioning nodes in a dataflow graph into sets so as to localize information flow [Ravi84]. The partitioned nodes are referred to as *tasks*. A task consists of a series of machine-level instructions to be executed sequentially. Static partitioning and allocation scheme enables tasks to be pre-allocated and distributed to the processing nodes.

3.2 Specification of the Architecture

The multiprocessor system has the organization shown in Fig. 3.2, consisting of several identical processing nodes (*Ps*). Each processing node has a Communication Interface Section (CIS), an Execution Section (ES) and an Output Section (OS), as shown in Fig. 3.3.

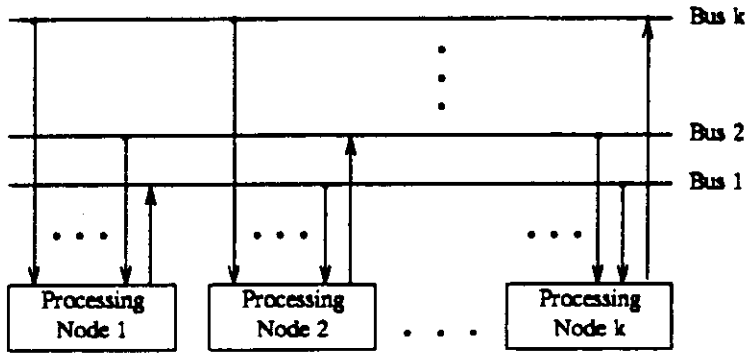

Figure 3.2. The Multiprocessor Organization

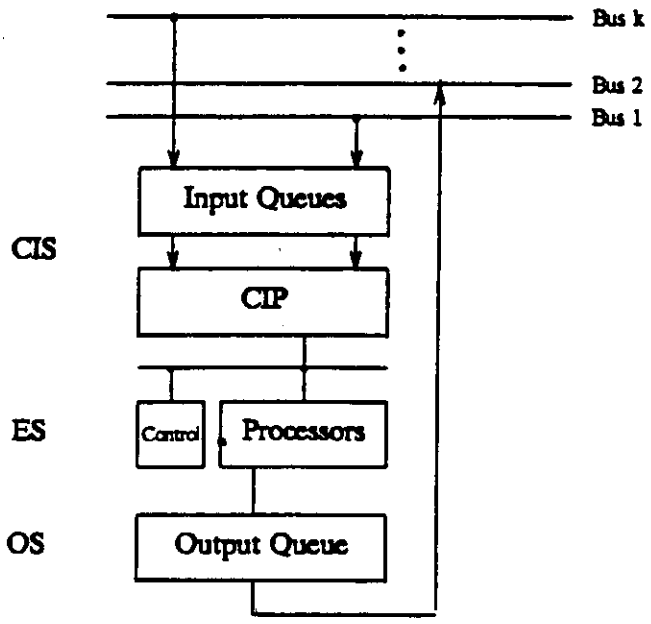

Figure 3.3. Processing Node Organization

A key feature of the multiprocessor organization is that source-dedicated, unidirectional broadcasting buses are used to provide global inter-processor communications. As a consequence of using such broadcasting buses, there is no need for explicit "copy" actions to replicate tokens to more than one processing site. A design choice is whether a dedicated- or a shared-bus structure is appropriate. For a system with N processing nodes, a dedicated-bus structure requires one bus for each processing node, whereas for a system with a shared-bus structure, the number of buses, M, is less than N. A bus control unit is also needed to arbitrate the use of shared-buses among the processing nodes. The performance of the system under a shared- and dedicated- bus structure will be discussed in Chapter 4.

Briefly, a processing node collects tokens from the buses, initiates the execution of a task when all the tokens are available, and broadcasts the results to other processing nodes. We now discuss in detail the main sections of a processing node.

3.2.1 The Communication Interface Section

The broadcasting buses transmit identical tokens to all processing nodes in the system. However, not every token is needed by the processing nodes in the cluster. We consider two possible ways to discriminate the tokens:-

Scheme A - Discrimination by sink processing nodes

This scheme requires each processing node to determine whether to accept or reject tokens. Since each token carries a unique identifier, based on a local linkage table, a processing node may accept or ignore a token.

Scheme B - Discrimination by source processing node

This scheme depends on the sending processing node to determine where the tokens should go. The sending processing node broadcasts a token in the first bus cycle. The token is buffered in the input registers of all processing node. A bit mask is then broadcasted, in the next bus cycle, to remove the undesirable tokens.

The effectiveness of these schemes are studied in Chapter 4. In both cases, Input Queues (IQs) are used in a processing node to buffer the tokens. An Input Queue consists of a number of registers organized as First Come First Serve (FCFS) queues. The number of queues in each processing node is equal to the number of buses, M, in the system. In other words, there is one IQ for each bus.

The IQs are passive devices that are controlled by the Communication Interface Processor (CIP), as shown in Fig. 3.4. The CIP serves the IQs based on a Longest-Queue-First discipline. That is, the IQ with the largest amount of queued items will be served next. The CIP polls the IQs to determine which IQ currently contains the most queued items.

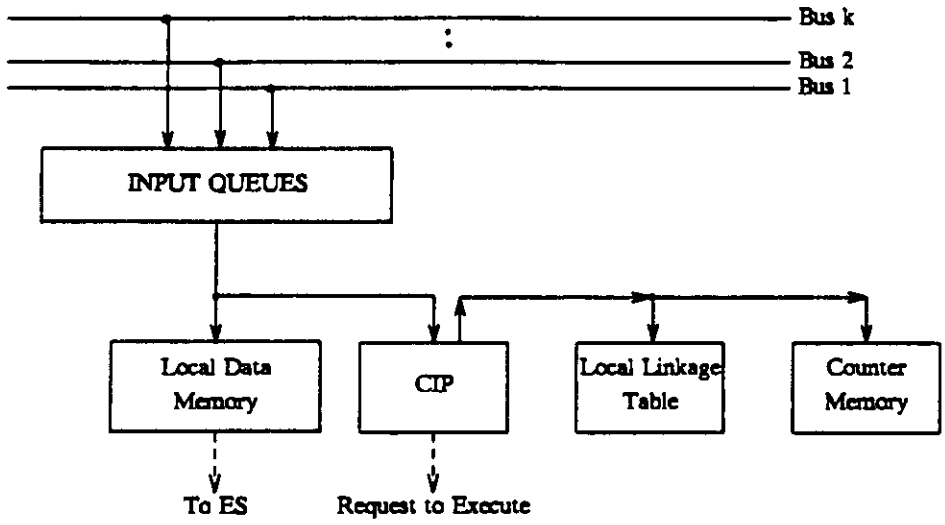

Figure 3.4. The CIS Configuration

The CIP owns a Local Linkage Table (see Fig. 3.4) which converts the global information carried by the unique identifier (x) and the instantiation number (j) of the token into information local to the processing node. Each token in the system has its corresponding entry in the linkage table.

The linkage table is indexed by tokens' identifiers so that tokens with the same identifier but different instantiation numbers require only one entry in the table. All entries in the linkage table are predetermined by the static partitioning and allocation system.

token	f1	f2	f3	f4	f5	f6	f7	f8
..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..

Table 3.1. Linkage Table Format
(see also Table 3.2)

The value (v) portion of an accepted token is transferred into a predetermined region of a Local Data Memory, is to be read later by the ES. Consequently, the Local Data Memory is dual-port. Referring to Tables 3.1 and 3.2, field [1] in the linkage table keeps the base address in the Local Data Memory for depositing the v portion of the token. The effective address is the base address plus the instantiation number, j, of the token. The arrival count of arguments for the tasks (stored in a Counter Memory) which need the token are also updated. Fields [2], [3] and [4]* in the linkage table contain the base addresses in the Counter Memory which is reserved for keeping the arrival counts. Once the arrival count of the task has reached its threshold, the starting locations of the tasks in an Instruction Memory, which are kept in fields [5], [6] and [7] of the linkage table, and the j portion of the token are transmitted to the ES. The last field is only used in scheme A. This flag informs the CIP whether or not to accept the token.

The linkage table enables the tokens, which have to be transmitted through the buses, to carry as little information as possible. Consequently, the table helps to reduce the bandwidth requirement of the buses.

*We assume the maximum number of tasks in a processing node which need the same token, is 3. This assumption can be easily relaxed by adding more fields to the linkage table.

Field	Description
1	local memory *base* address to deposit the token
2	*base* address of the arrival counter in counter memory for task #1
3	*base* address of the arrival counter in counter memory for task #2
4	*base* address of the arrival counter in counter memory for task #3
5	starting address for task #1
6	starting address for task #2
7	starting address for task #3
8	flag

Table 3.2. Field Description of the Linkage Table

3.2.2 The Execution Section

The Execution Section consists of a control unit, a scratch pad memory, an instruction memory module, and one or more processors, as shown in Fig. 3.5.

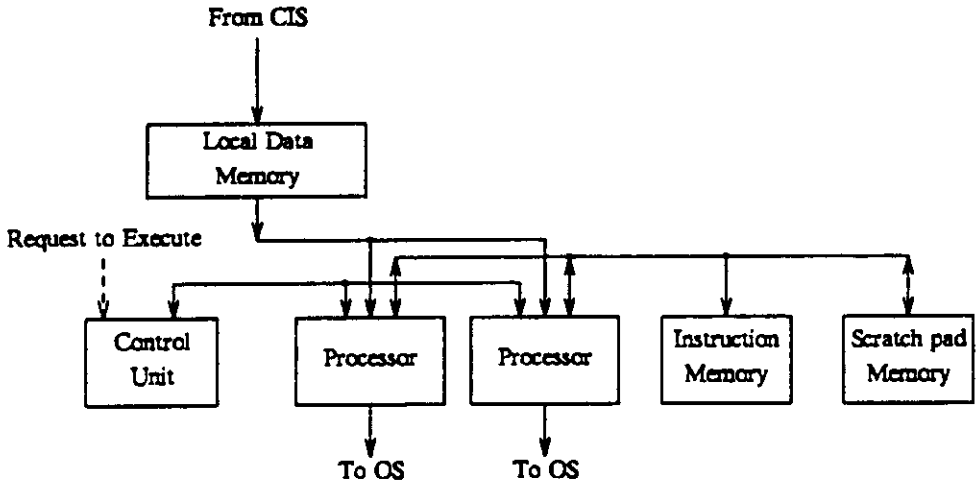

Figure 3.5. The ES Configuration

The control unit intercepts requests from the CIP and invokes the tasks by transferring control to one of the free processors. A task consists of a series of machine-level instructions. There is an instruction memory module which stores the machine-level code of the tasks. An instruction counter is used to keep track of the address of the next instruction. The processor starts execution by fetching the first instruction, with the starting address supplied by the CIP, from the instruction memory. The operands may come from three different sources: 1) immediate values which are encoded within the instructions, 2) the tokens in the dual-port memory, and 3) the scratch pad memory. As soon as a result is generated by a task, the control unit moves the result to the Output Section (OS). Along with that, a new instantiation number j and the result token's identifier, are also transferred to the OS.

A scratch pad memory is used to hold the currently needed data values. This scratch pad memory is useful when the dataflow graph exhibits a certain degree of sequentiality among the tasks. It supports higher-level of granularity. The inter-node communication of routing tokens, through the global buses, the IQs, and the CIP, is reduced by keeping the intermediate data values in the scratch pad memory. However, location assignments of the scratch pad memory have to be handled carefully at compile time to preserve the functionality of the tasks and avoidance of possible side effects [Requ83].

3.2.3 The Output Section

The OS is responsible for converting results from the ES into token format, and sending tokens out via the broadcasting buses. In essence, it does the following simple functions:

a. Obtain results, the new instantiation number j and the new token identification number x from the ES.

b. Pack the result into the token format $v.x.j$.

c. Broadcast tokens from the OS queue through the broadcast bus.

d. For scheme B only, the OS broadcasts the bit mask to clear up the tokens which are still in the input register of the potential recipients.

3.3 System Extension

The interconnection structure shown in Fig. 3.2 is limited to a small number of processing nodes. In that case, the number of Input Queues (IQs) is N^2, where N is the number of processing nodes. Fig. 3.6 suggests a way to incorporate a larger number of processing nodes with the basic structure, for a dedicated-bus system. In this case, the number of IQs is $(\frac{N}{2})^2$ and we can have shorter buses. Processing nodes are combined into clusters (groups). There is a processing node within a cluster responsible for intra-cluster communications. The solid lines drawn in Fig. 3.6 denote inter-cluster buses, while the dashed lines are for intra-cluster communications. The inter-cluster buses support tight interaction among the processing nodes in a cluster, while the intra-cluster buses are for loose interaction among the clusters.

Inter-cluster communications requires one step (bus cycle) while intra-cluster communications may take two or three steps. For example, if P_1 needs to send tokens to P_5. The tokens have to go through:

$$P_1 \rightarrow P_4 \rightarrow P_8 \rightarrow P_5$$

which takes three steps, whereas the transfer of tokens from P_1 to P_8 requires only two steps.

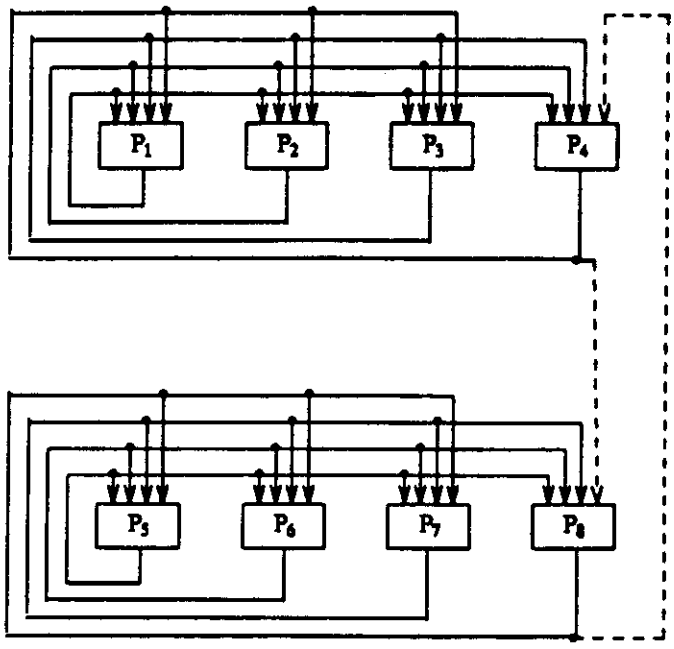

Figure 3.6. System Extension

3.4 Implication of the Arrival Counts - Pseudo Dependence

Since this machine sequences the tasks solely based on the arrival of tokens, there is no centralized control mechanisms to exert an execution sequence. In this section, we investigate the possibility of exploiting the *arrival counts* to introduce pseudo dependence among the tasks.

Pseudo dependence is advantageous in two aspects. First, it enables one task to be dependent on the completion of another task so to exert the execution of a particular task to take place only after a certain point in the computation. Fig. 3.7 depicts one occasion where enforced execution is beneficial. Tasks α_1, α_2 and α_3 are allocated to P_1, whereas tasks α_4 and α_5 are allocated to P_2, the critical path is $\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3$. Let us assume that x_1 and x_2 arrive simultaneously at both

processing nodes, and there is only one processor at each processing node. It is also assumed that the inter-task communication time is negligible compared with the execution time of the tasks. If P_2 fires α_5 before α_4, then P_1 has to wait for the completion of α_5 and α_4 before it can fire α_2. On the other hand, if α_4 is scheduled before α_5, then P_1 can run smoothly without pauses. This scheduling can be accomplished by introducing a *pseudo arc* from α_4 to α_5. A pseudo arc is different from a regular dependence arc in two ways: first, the data produced by the parent is not consumed by its dependent(s). Second, the arrival counts (thresholds) of the dependent tasks are affected by the arrivals of tokens belonging to the parental task. So, α_4 has an arrival count of one while the arrival count for α_5 is two: α_4 can be fired as soon as x_2 arrives, whereas α_5 has to depend on the arrivals of x_1 and x_2.

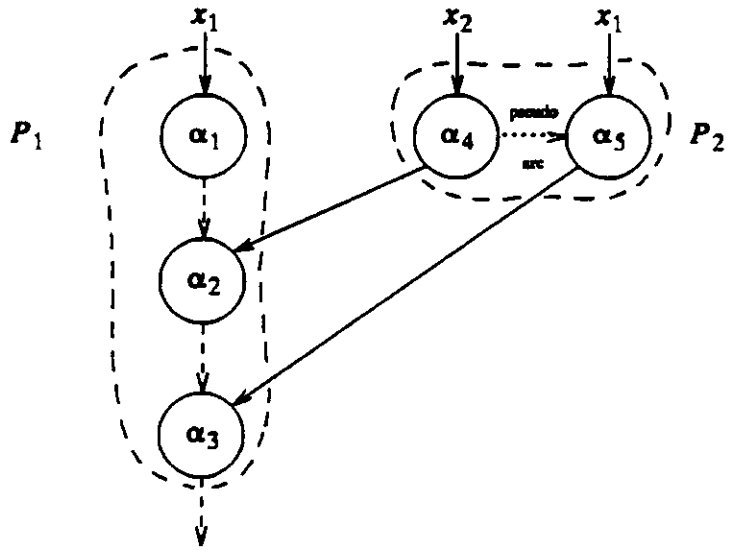

Figure 3.7. A Scheduling Problem

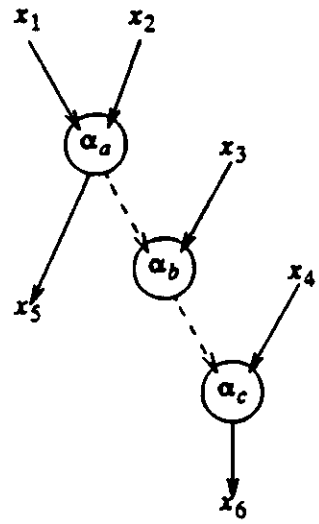

Figure 3.8. Segment of a Task Graph

Pseudo dependence also aids to reduce inter-task (token) traffic. Fig. 3.8 depicts a segment of a task graph from a tree-reduction algorithm. Let's suppose that this segment is allocated to a single processing node. Tokens x_1 and x_2 are associated with task α_a, while task α_b needs the result from task α_a and token x_3, and task α_c requires result from task α_b and token x_4. We assume the order of arrival for the tokens is not known. We see that this segment of the graph exhibits strong sequentiality among the tasks. An attempt to execute this task graph at this level of resolution is inappropriate, since each inter-task token has to go through the Output Section, the buses, and then the CIP before the tasks α_b and α_c can be fired.

Gaudiot and Ercegovac [Gaud84] have considered variable-level granularity for the execution of dataflow graphs to reduce the overhead in inter-task communications. According to their scheme, the three tasks are combined together to form a macro-task. This macro-task has to wait for the arrivals of tokens x_1, x_2, x_3, and x_4 before it can be fired. However, this scheme does not exploit

the possibility that tokens x_1 and x_2 might arrive considerably earlier than the rest of the tokens. If this happens, task α_a can be fired and can send out token x_5. In fact, it is not hard to see that if the variance of the arrival times of the tokens belonging to a macro-task is large, some tokens have to wait a considerable amount of time before they are used; even though some of the sub-tasks may be firable during the waiting period.

We suggest that by imposing pseudo dependence, precedence among the tasks can be honored, parallelism can be exploited, and yet inter-task communications can be reduced. To accomplish this, first, a threshold (stored in the Counter Memory, see also Table 3.2) is assigned to a task as follows:

Task	Threshold	Meaning
α_a	2	fan-in to α_a
α_b	3	threshold of α_a + *external* fan-in to α_b
α_c	4	threshold of α_b + *external* fan-in to α_c

When either x_1 or x_2 arrives, the arrival counts for *all tasks* are decremented by one. A task can be fired as soon as the corresponding arrival counter becomes zero. The threshold for task α_a is associated with tokens x_1 and x_2. Similarly, the threshold for task α_b is affected by tokens x_1, x_2, and x_3 (but *not* x_4!). However, the association between tokens x_1, x_2 and task α_b is only virtual. It merely serves to insure that α_b is executed after the completion of α_a. In the same manner, the threshold for task α_c is affected by tokens x_1, x_2, x_3 and x_4. By using arrival counters to keep track of the thresholds, precedences among the

tasks can be honored*. Further, to reduce the inter-task communications due to the sequential nature of the graph, the intermediate tokens generated by tasks α_a, and α_b are kept in the scratch pad memory. Subsequent operand access for tasks α_b and α_c will be to the scratch pad memory. This eliminates the overhead of routing the intermediate tokens through the global buses, the IQs, and the CIP and hence reduces the inter-task communications.

3.5 Implementation Considerations

The bus-oriented interconnection between the processing nodes renders a practical implementation of the system. The IQs can be readily implemented using commercial FIFO queue controller (e.g., The Signetics 8X60 RAM controller or Zilog Z-FIFO Buffer Unit). The R/W memory between the CIS and ES should present no particular problem since dual-port access RAM modules have been available since 1977 (e.g., AMD 29705). If dual-port RAMs are not commercially available at low-cost, the R/W memory can be implemented with some Direct Memory Access (DMA) schemes.

We consider the arithmetic requirements involved in continuous-system simulations: logical operations, transcendental functions, additions and multiplications which require that the processors in the ES be floating-point ALUs.

The availability of low-cost microprogrammable processors (e.g., AMD 2900 series) particularly offers an advantage: it enables the user to tailor the microcode for an instruction set to match the desired level of abstractions coded in the high-level language. The user does this by adding instructions to the instruction set to perform operations that otherwise would require two or three primitive instructions to be fetched from memory and executed.

*For the special case of simultaneous arrivals of tokens, (say α_a and α_b are firable at the same instance) the tie can be broken by properly arranging the local linkage table. The starting address for α_a must be placed in the second position of the linkage table, preceded by the starting address of α_b. In this way, the CIP scans (and delivers to the ES) the starting address for α_a before α_b.

CHAPTER 4
Performance Evaluation

4.0 Overview

This chapter presents two approaches used in the performance study of the multiprocessor system described in Chapter 3. First, we discuss a simulation model which aims at providing an environment for 1) fine tuning of the system architecture, 2) experimentation with heuristics for task partitioning and allocation, and 3) study of various parallel algorithms. Second, a functional model based on mean-value arguments is devised to approximate a number of mean performance figures.

4.1 Definitions

The following symbols denote the parameters of the multiprocessor system:

P_i : i-th processing node.

N : number of processing nodes.

M : number of buses.

t_b : cycle time of the bus.

t_c : average time the CIP spends on a needed token.

u_c : average time the CIP spends on a unused token.

t_o : average overhead time ES spends on invoking a task.

η_i : $\dfrac{\text{Total Number of Tokens Used by } P_i}{\text{Total Number of Tokens Queued in } P_i}$.

η : $\dfrac{\sum_{i=1}^{N} \eta_i}{N}$, the equilibrium average token acceptance ratio.

nt_i : number of tasks allocated to P_i.

$\beta_{j,i}$: fan-out of task j in P_i, defined as the number of outgoing arcs from the task.

$\gamma_{j,i}$: fan-in of task j in P_i, defined as the number of incoming arcs to the task.

β : $\dfrac{1}{N} \sum_{i=1}^{N} \dfrac{\sum_{j=1}^{nt_i} \beta_{j,i}}{nt_i}$, average fan-out per task, per processing node.

γ : $\dfrac{1}{N} \sum_{i=1}^{N} \dfrac{\sum_{j=1}^{nt_i} \gamma_{j,i}}{nt_i}$, average fan-in per task, per processing node.

W : mean number of tokens the CIP inspects to fire a task.

$U_{c,i}$: equilibrium mean utilization of CIP in P_i, the fraction of time that the CIP is busy.

U_c : $\dfrac{\sum_{i=1}^{N} U_{c,i}}{N}$, equilibrium mean utilization over all CIPs.

$U_{e,i}$: equilibrium mean utilization of ES in P_i, the fraction of time that the ES is busy.

U_e : $\dfrac{\sum_{i=1}^{N} U_{e,i}}{N}$, equilibrium mean utilization over all ESs.

$U_{b,i}$: equilibrium mean utilization of bus in P_i, the fraction of time that bus is busy.

U_b : $\frac{\sum_{i=1}^{N} U_{b,i}}{N}$, equilibrium mean utilization over all buses.

$T_{e,i}$: average execution time per task in P_i.

T_e : $\frac{\sum_{i=1}^{N} T_{e,i}}{N}$, average execution time per task per processing node.

T_c : processing time required by the CIP before a task can be fired.

S : $\frac{\text{Time steps taken with a sequential processor}}{\text{Time taken with } N \text{ Processing Nodes}}$, speedup; it is assumed that communication cost is zero on the sequential processor.

E : $\frac{\mathbf{S}}{N}$, efficiency.

RT : mean response time per task, per processing node.

NRT : $\frac{RT}{T_e}$ normalized mean response time per task, per processing node.

4.2 The Simulation Environment

A simulator has been developed to simulate the multiprocessor system at the *physical* level [Ferr83]. The principal objectives of the simulator are to facilitate 1) fine tuning of the proposed system architecture, 2) to experiment with heuristics for partitioning and allocation, and 3) to study various parallel algorithms. To accomplish these objectives, some details of implementation at the *machine instruction level* of the system are incorporated. The level of detail is

sufficient to model the interactions between the software and hardware resources, the contention between the hardware resources, and the timing of a computation. However, simulation of the system at the microinstruction level is not considered.

The simulator characterizes the multiprocessor system by a deterministic state model. Inputs to this model are the machine-level instructions compiled from the FPL specification, the time delays for each instruction, and the linkage tables for each processing node.

There are three main reasons for adopting a deterministic model instead of a probabilistic model. First, being deterministic the simulated numerical results can be compared with the results obtained by running the high-level specification on an FPL *value** interpreter. This would partially validate the simulation. Second, the effect of word lengths (eg. 16-bit/32-bit) on the accuracy of continuous-system simulations can also be studied. Third, as mentioned in Chapter 2, the application environment is assumed to be relatively static; it is therefore inappropriate to introduce random quantities, as in probabilistic models, to characterize the workloads of the system.

Activities in a processing node are mapped into states. The behavior of the multiprocessor is identified by the transition of states. Each state is assumed to take 10 cycles of the simulation time (see Table 4.1).

Fig. 4.1.a illustrates the state transition diagram for the CIP employing *scheme A* (see Chapter 3) to discriminate the tokens. The states are defined as:

State 1: CIP polls the IQs to determine which IQ currently has the most queued items.

State 2: CIP reads the IQ with the most queued items.

*The FPL *value* interpreter accepts and computes numerical inputs, whereas the FPL *symbolic* interpreter that we mentioned in Chapter 2 accepts symbolic input and does no numerical computations.

State 2: CIP reads the IQ with the most queued items.

State 3: CIP looks up the linkage table to determine whether to accept or reject the token.

State 4: CIP writes the data portion of the token into the dual-port memory.

State 5: CIP decrements the arrival counts for the corresponding tasks.

State 6: CIP notifies ES that a task is ready to be executed.

In *scheme B*, the transition from state 3 to state 1 is omitted, since all tokens read from the IQs are needed by the processing node.

Activities in the ES is modelled with three states. State 1 intercepts requests from the CIP, State 2 checks whether any free processors are available, and State 3 is the state for executing the tasks, as depicted in Fig. 4.1.b. Similarly, Fig. 4.1.c illustrates the state transition diagram for the Output Section (OS): State 1 of the OS obtains result from the ES, and State 2 broadcasts the result to different processing nodes. In *scheme A*, the result is broadcasted to all processing nodes, whereas in *scheme B* the result is broadcasted to some selected processing nodes. It is assumed that t_b is 10 cycles for *scheme A* and 20 cycles for *scheme B* in the simulation.

The simulator consists of approximately 1500 lines of C code [Ritc78] and runs under Unix Operating System. The simulator is divided into three sections: initialization, simulation, and statistics collection. The initialization section reads the machine-level code, the linkage tables and the timing parameters from an input file. The simulation section controls the transition of states by keeping track of an *event* list and a *resource* list. The statistics collection section gathers data at each increment of the simulation clock and reports the statistics when the simulation terminates.

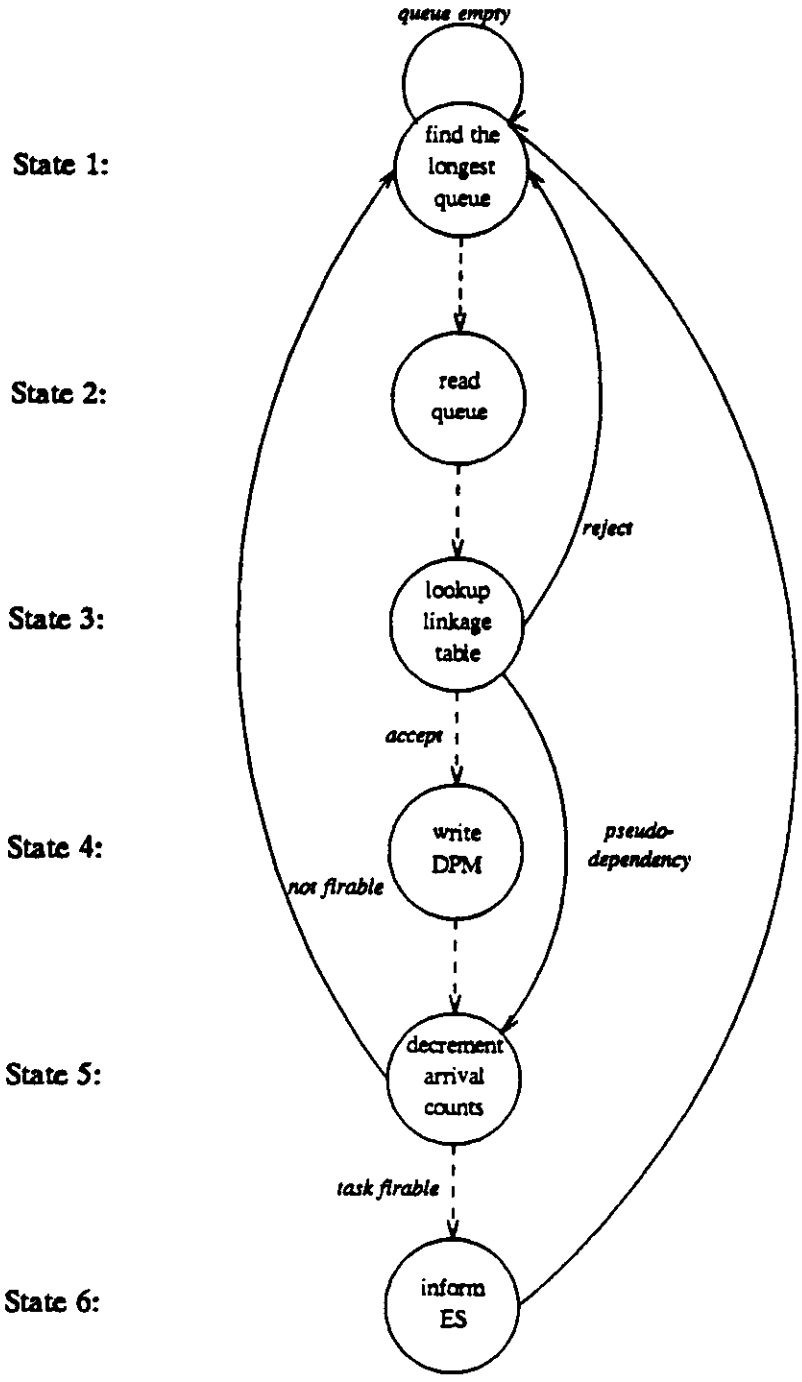

Figure 4.1.a. State Transition Diagram
***(scheme A)* for the CIP**

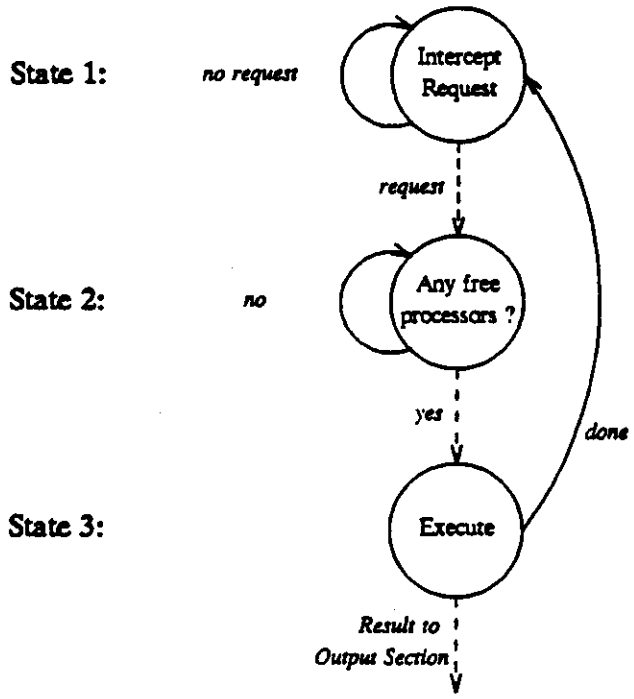

Figure 4.1.b. State Transition Diagram
***(scheme A & B)* for the ES**

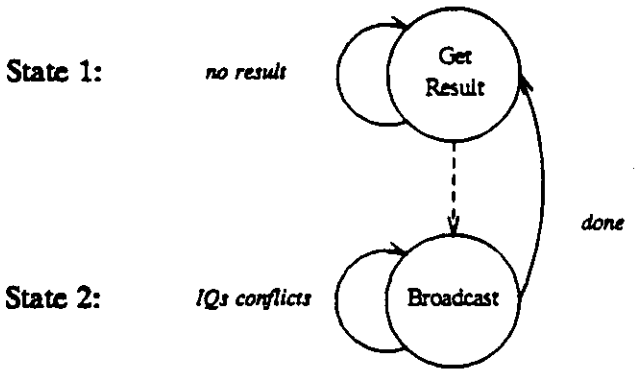

Figure 4.1.c. State Transition Diagram
***(scheme A & B)* for the OS**

The main issues in the construction of the simulator are 1) the simulation of concurrent processes and 2) the flexibility in accommodating different design alternatives. In order to accomplish these goals, first, we need to maintain a workspace (data structures + register images) for each process. Second, each section in the processing node is presented as a separate procedure. To execute a particular process, the pointers in the procedure are set to reference the workspace for the process. The simulation time will not advance untill all the processes which are firable, within the simulation time interval, are executed. Concurrent processes are kept mutually exclusive (no resource conflicts) from each other: contentions of resources within the same simulation time interval are resolved by pre-assigning priorities to processes. The simulator is believed to be structural and flexible enough to adopt to arbitrary number of processing nodes.

Along with the numerical results of the computation, the system-related performance statistics - the lengths of the interface queues between sections in the multiprocessor system, the waiting times of the tokens, the utilizations of the bus, CIP and ES are also reported by the simulator (see Table 4.2).

A Simulation of Continuous-system Simulation

To see the simulator in action, consider the coupled pendulums example that we discussed in section 2.6. Recall that the FPL program to model the coupled pendulums is given in Appendix A, and the computational graph resulting from the symbolic interpretation of the program is shown in Fig. 2.7. In Fig. 2.7, the dotted lines outline one possible (manual) partitioning of the graph into tasks. The machine-level code and the linkage tables generated based on such a partitioning, assigned to *six* processing nodes, are given in Appendix B and C, respectively.

The timing parameters used in this simulation are tabulated in Table 4.1. Fig. 4.2.a depicts the Gantt chart for the tasks scheduled without imposing pseudo dependences, for *scheme A*. The heavy lines indicate the ESs are in execution (state 3). The scenario of execution with pseudo dependences is shown in Fig. 4.2.b. It is clear that by imposing pseudo dependences, we obtain more overlaps between computation and communication.

Tables 4.2 and 4.3 give the performance statistics gathered by the simulator for *scheme A* and *B* (with pseudo dependences), respectively. We see that as far as the processing node utilizations is concerned, there is apparently no differences between the two schemes. P_1 is the most heavily used because all the tasks along the critical path are allocated to P_1. P_2 and P_3 are the supporting processing nodes intended to sacrifice their utilizations so as to keep P_1 busy. As expected, the CIP utilizations in *scheme B* are lower than in *scheme A*. This is due to the fact that in *scheme A*, the CIPs have to process both the needed and undesirable tokens, whereas in *scheme B* the CIPs only have to process the needed ones. Tables 4.2 and 4.3 also show that the buses are under-utilized in both *scheme A* and *B*. The bus utilizations are higher in *scheme B* mainly because we set $t_b = 10$ in *scheme A* and $t_b = 20$ in *scheme B*. This suggests that a shared-bus structure may as well be suitable for this application.

Parameter	Assumed time steps	No. of instructions per iteration (if applicable)
t_b	10	(*scheme A*)
t_b	20	(*scheme B*)
t_{add}	20	10
t_{sub}	20	10
t_{mul}	80	28
t_{div}	80	8
t_{cos}	160	2
t_{sin}	160	6
t_{pow}	160	2
t_{sqr}	80	2

Table 4.1. Timing Parameters used in the Simulation

P_i	$U_{e,i}$ ES Utilization(%)	$U_{b,i}$ bus Utilization(%)	$U_{c,i}$ CIP Utilization(%)	η_i Token Acceptance Ratio(%)
1	92.60	0.75	24.64	50.67
2	50.47	1.50	23.99	50.67
3	37.48	1.55	18.74	20.00
4	92.45	0.75	24.59	50.00
5	50.47	1.50	23.94	50.00
6	37.48	1.55	18.74	20.00
average	60.1	1.3	22.5	40.00

Total token accepted : 906
Average token flows : 0.045277 tokens/cycle
Speedup : 3.61

Table 4.2. Performance Statistics *(scheme A)* Reported by the Simulator

P_i	$U_{e,i}$ ES Utilization(%)	$U_{b,i}$ bus Utilization(%)	$U_{c,i}$ CIP Utilization(%)	η_i Token Acceptance Ratio(%)
1	93.58	1.45	16.15	100
2	49.74	2.95	15.47	100
3	36.69	2.95	6.00	100
4	93.68	1.45	16.12	100
5	49.74	2.95	15.45	100
6	36.69	2.90	6.00	100
average	60.02	2.44	12.53	100

Total token accepted : 712
Average token flows : 0.017796 tokens/cycle
Speedup : 3.60

Table 4.3. Performance Statistics (*scheme B*) Reported by the Simulator

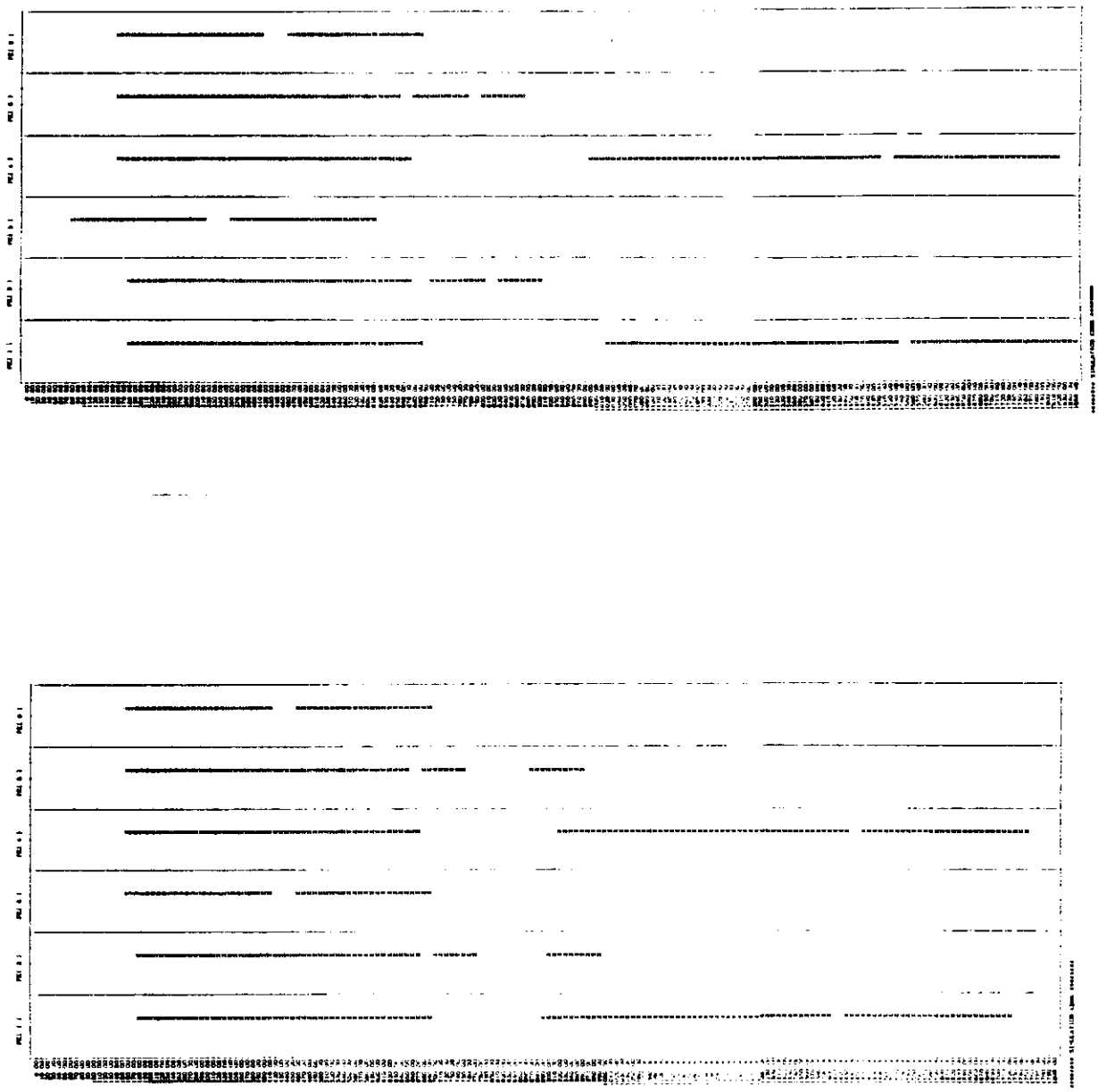

Figure 4.2. Gantt Charts

4.3 An Equilibrium Model based on Mean-value Arguments

Our goal in this section is to derive a functional model to approximate the performance of the multiprocessor system at equilibrium. We believe that an approximate functional model, based on mean-value arguments, will suffice to provide some insights and better understanding on the system performance. This functional model makes no assumptions on the implementation details. Only the functional behavior of the multiprocessor system is considered. The word "mean-value" means that we are not seeking for the performances of individual processing nodes. Rather, we are interested in the overall speedup, and the mean utilizations of the CIPs, ESs and buses.

The key inputs to this model are:

$$\beta = \text{average fanout per task per processing node} \tag{4.1}$$

and

$$T_e = \text{average execution time per task} \tag{4.2}$$

which attempts to characterize the *average* effect of task partitioning and allocation. The way that β varies with T_e depends on the nature of computational graphs and the partitioning strategy. For example, we expect a binary tree to exhibit an almost *linear* relationship, whereas in a computational graph which is sequential, β is independent of T_e (see Fig. 4.3).

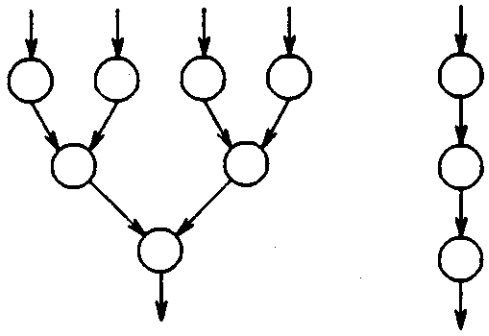

Figure 4.3. A Binary Tree and A Sequential Task Graph

The basic observation is that, at equilibrium, the number of tokens consumed by the tasks is equal to the number of (result) tokens generated by the tasks. That is, γ equals β. If this is not the case, the number of tokens will either diminish or grow. In the former case, the system will be idle while in the latter case, the system will be saturated.

The following analysis applies only to *scheme A*. At equilibrium, each processing node in the system is envisioned as possessing a *single* task with processing demand T_e and fan-out (which is also the fan-in) β. The result token produced by this task is needed by β processing nodes. Hence, the average token acceptance ratio, η, is related to the average fan-out by,

$$\eta = \frac{\beta}{N} \tag{4.3}$$

Next, the task activation overhead (time required by the CIP before a task can be fired) is given by,

$$T_c = (W - \beta) \cdot u_c + \beta \cdot t_c \tag{4.4}$$

where W is the mean number of tokens that the CIP has to inspect before the task is fired. In general, W is not equal to β since each processing node in the system

broadcasts the result token to the rest of the processing nodes. In the worst case, we expect the CIP would inspect a total of N tokens before it collects all the desirable tokens for the task. On the other extreme, the CIP only needs to inspect the *first* β tokens before it fires the task. Therefore, the mean value of W is expected to be:

$$W = \frac{\sum_{i=\lceil\beta\rceil}^{N} j}{N - \lceil\beta\rceil + 1}$$

$$= \frac{N + \lceil\beta\rceil}{2} \tag{4.5}$$

The average response time and the normalized average response time for a task are respectively,

$$RT = T_c + T_e + t_b + t_o \tag{4.6a}$$

$$NRT = \frac{RT}{T_e} \tag{4.6b}$$

where t_b is the bus cycle time and t_o is the average *overhead* time spent on invoking a task, by the ES.

Following Eq. (4.6), the equilibrium mean utilization for the ES is given by

$$U_e = \frac{T_e}{RT} \tag{4.7}$$

Note that U_e corresponds to the efficiency $\mathbf{E} = \frac{\mathbf{S}}{N}$ of the system.

Regarding the CIP, since it collects a total of N tokens from all processing nodes at each (task) cycle, it then spends $\beta \cdot t_c$ amount of time on the needed tokens and $(N-\beta) \cdot u_c$ time steps on the rest of the tokens, this yields

$$U_c = \frac{(N-\beta) \cdot u_c + \beta \cdot t_c}{RT} \tag{4.8}$$

Finally, the equilibrium mean bus utilization and the speedup can be expressed as

$$U_b = \frac{t_b}{RT} \tag{4.9}$$

$$\mathbf{S} = \frac{N \cdot T_e}{RT} \tag{4.10}$$

since a sequential processor would take N times the amount of time for the multiprocessor system.

By the same arguments, T_c, RT, U_e, U_c, U_b, and the speedup S for *scheme B* can be shown to be:

$$T_c = \beta \cdot t_c \tag{4.11}$$

$$RT = T_c + T_e + t_b + t_a \tag{4.12}$$

$$U_e = \frac{T_e}{RT} \tag{4.13}$$

$$U_c = \frac{T_c}{RT} \tag{4.14}$$

$$U_b = \frac{t_b}{RT} \tag{4.15}$$

$$S = \frac{N \cdot T_e}{RT} \tag{4.16}$$

Numerical Example

Eqs. (4.1) to (4.16) are the main results for the performance prediction. To see how accurate they are, we compare them with the result from simulation. The following data are obtained from the coupled pendulums example (see Fig. 4.2).

P_i	$T_{e,i}$	Number of task per iteration nt_i	Number of tokens per iteration per task
1	446	3	2.0
2	226	3	2.0
3	250	2	1.5
4	446	3	2.0
5	226	3	2.0
6	250	2	1.5

Table 4.3. Partitioned Tasks Data from the Coupled Pendulum Example

with $N=6$, $t_c=40$ (see Fig. 4.1), $u_c=20$, $t_b=10$, and $t_o=20$, we get W = 4. and from Table 4.3 we obtain,

$$\beta = 1.8333 \tag{4.17}$$

and

$$T_e = 308 \tag{4.18}$$

so,

$$T_c = (4-1.8333)\cdot 20 + 1.8333\cdot 40 = 116.667 \tag{4.19}$$

which, in turn, give us,

Performance indexes	Result from the Functional Model		Result from Simulation	
	scheme A	*scheme B*	*scheme A*	*scheme B*
η	31.2%	0.0%	40%	0%
U_e	67.7%	73.1%	60.1%	60.1%
U_c	34.5%	17.4%	22.5%	12.5%
U_b	2.2%	4.7%	1.3%	2.4%
S	4.1	4.4	3.6	3.6
E	0.67	0.73	0.6	0.6

Table 4.4. Comparison of Results

In essence, results from the two models do approximately agree with each other, and the performance values from the functional model are higher than from simulation. This may be due to the fact that the functional model is devised for equilibrium behavior, whereas the simulation accounts both the transient and equilibrium performances. Nevertheless, we see that the functional model captures the essential features of the proposed architecture.

4.4 Performance Projections

This section uses the functional model to study the performance of the machine, under *scheme A*. We basically are concerned with the performance indexes U_e, U_c, U_b, and S and the normalized response time as a function of N, T_e and β. The performance measures: Eqs. 4.1 to 4.10 form the basis for this study.

In attempting to quantify a relationship between β and T_e, we consider the simple case that β is a linear function of T_e, namely,

$$\beta = r \cdot T_e \tag{4.20}$$

where r is a constant expressing the unit increment of β w.r.t. T_e. Substituting Eq. (4.14) into Eqs. (4.5) to (4.10) gives us the expressions for the performance

of the machine w.r.t. t_o, t_b, N, T_e and r. Moreover, we select N=8 and r=*1/150*, and as before we set t_o=20 and t_b=10. Fig. 4.4.a,b,c and d illustrate how the normalized average response time, CIP, ES and bus utilizations change through the variation of T_e, respectively. Clearly, these figures indicates that scaling up the level of granularity (larger T_e and β) yields higher utilization of the ES and smaller normalized average response time.

It is also illustrative to see the change of N on the performance of the machine. We keep the assumption that $\beta = r \cdot T_e$. Here, we select N as the independent variable. Fig. 4.5.a,b,c and d demonstrate the effect of scaling up N on the speedup, normalized average response time, CIP and ES utilizations, respectively. Fig. 4.5.a and b suggest that increasing the number of processing nodes in the multiprocessor system has the undesirable effect of saturating the system as indicated by the gradual deviation of the achieved speedup from the ideal case. This may be due to the fact that as the number of processing nodes increases, the number of tokens that has to be processed by the CIPs subsequently will increase. This in turn increases the inter-task communication time between task activations and thus degrade the performance of the system.

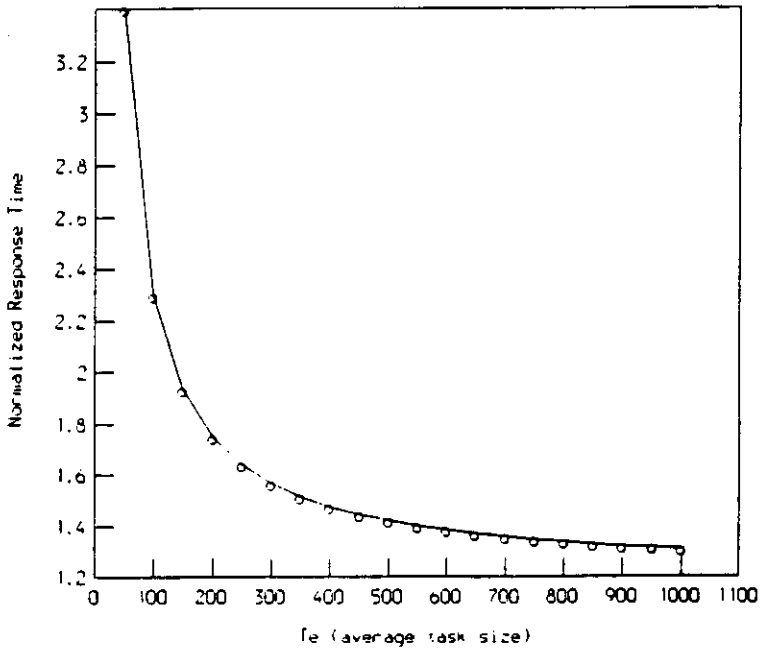

Figure 4.4.a. ***NRT*** **vs.** T_e

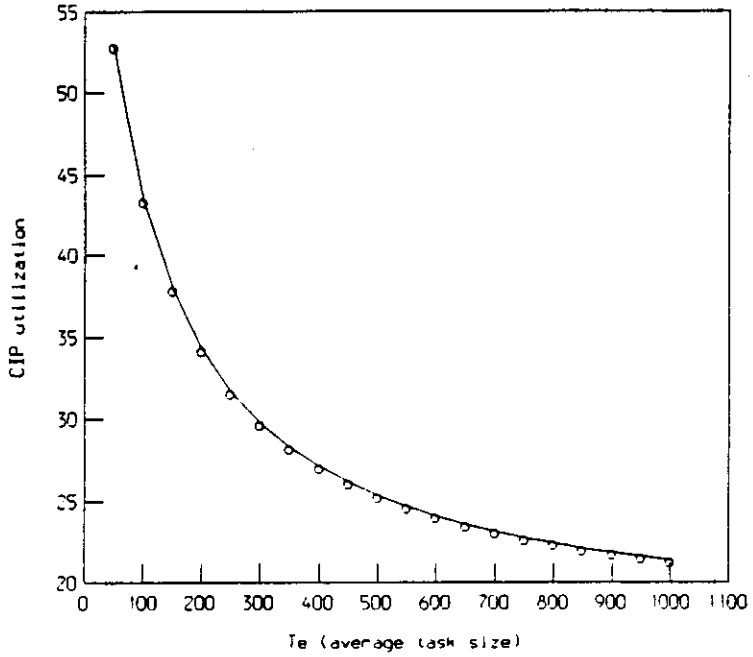

Figure 4.4.b. U_c **vs.** T_e

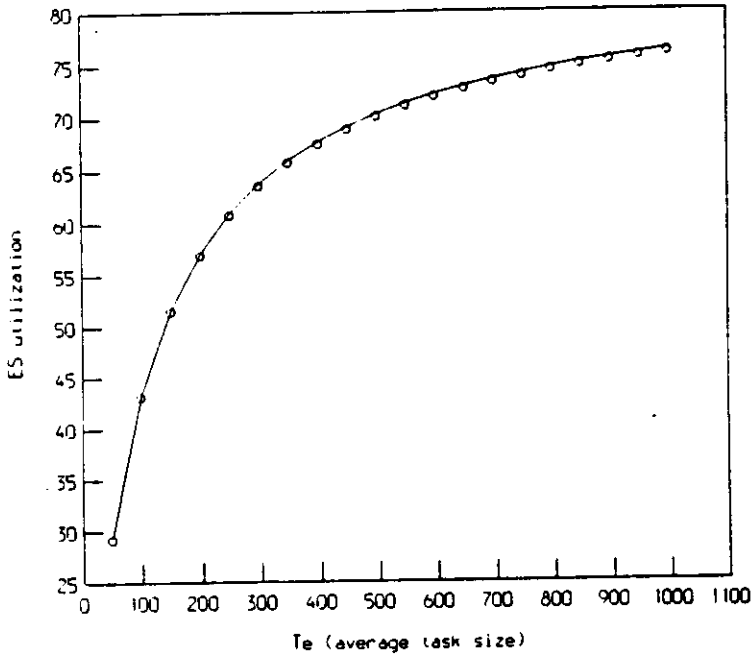

Figure 4.4.c. U_e vs. T_e

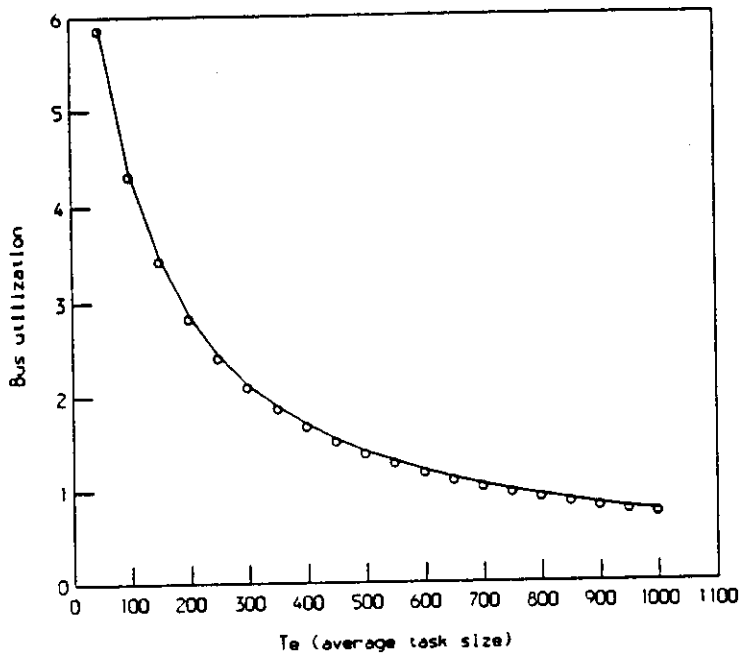

Figure 4.4.d. U_b vs. T_e

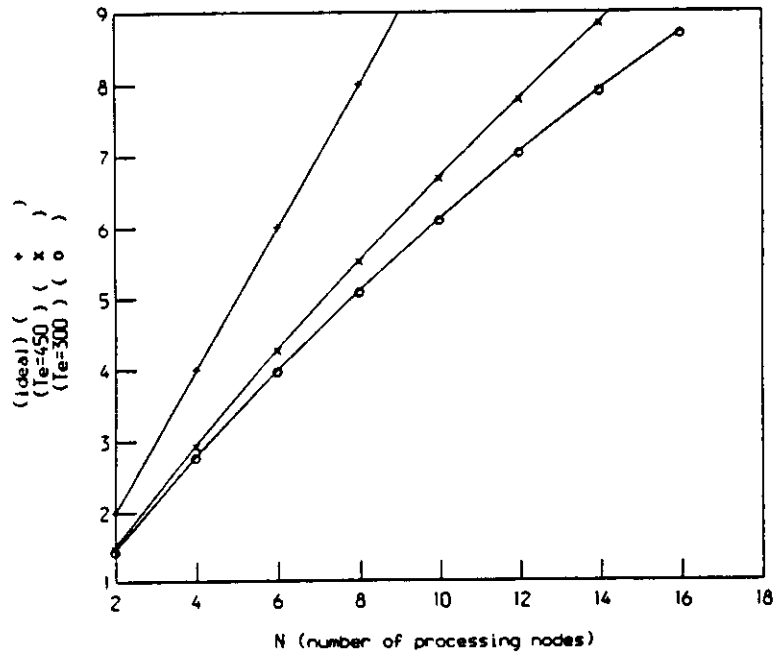

Figure 4.5.a. S vs. *N*

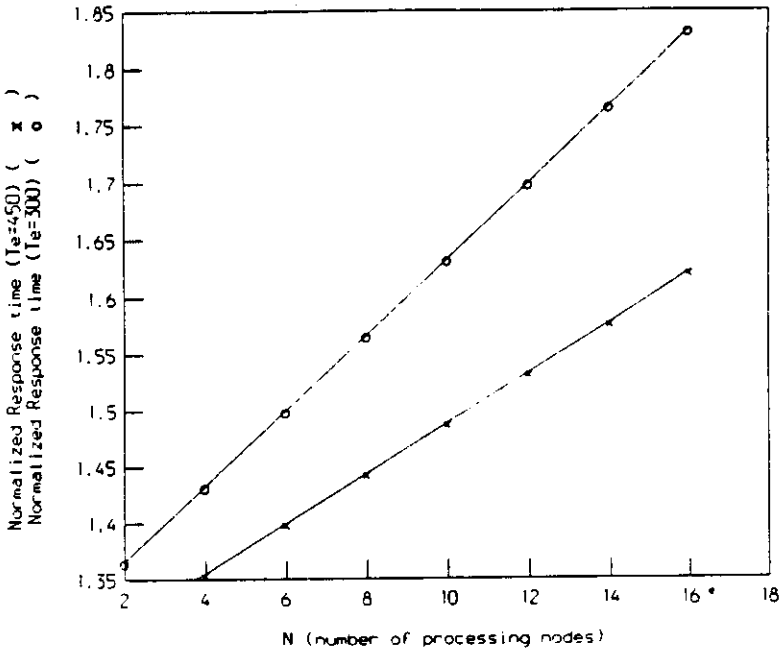

Figure 4.5.b. *NRT* vs. *N*

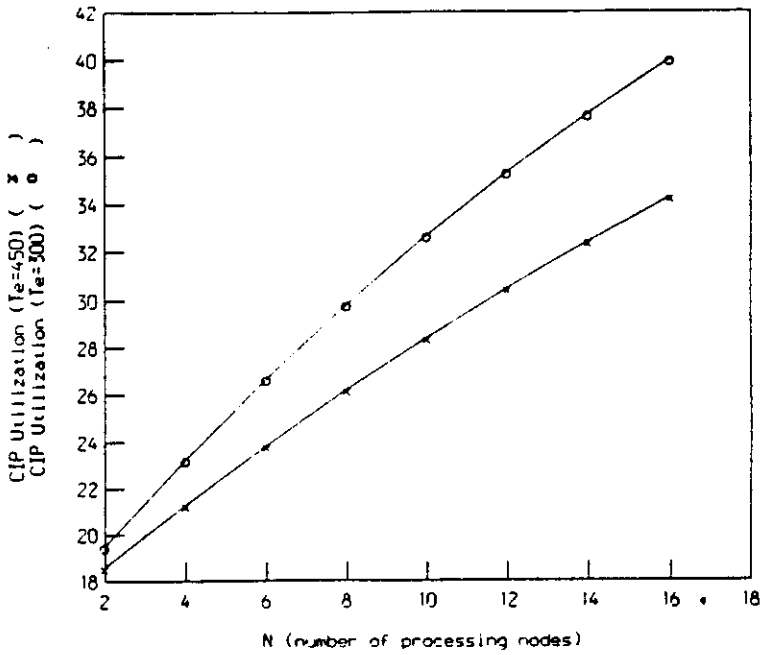

Figure 4.5.c. U_c vs. N

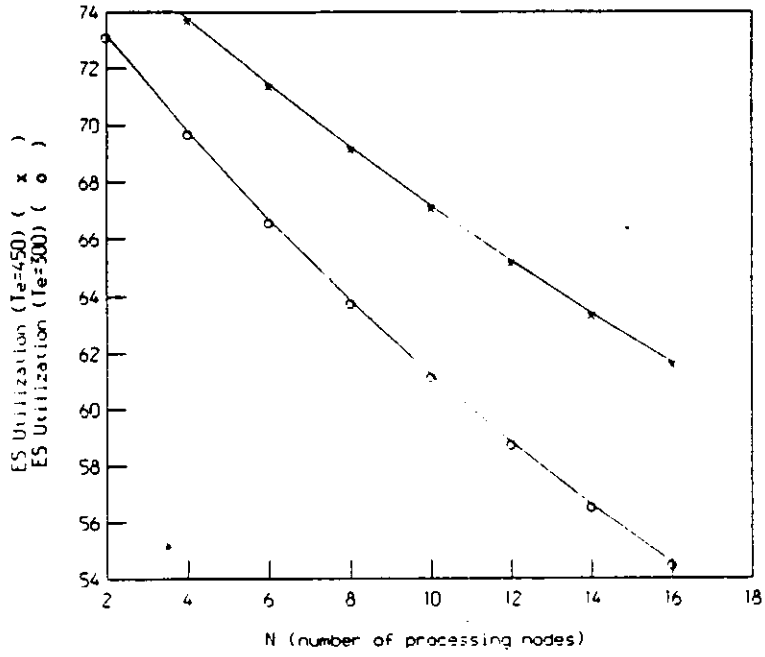

Figure 4.5.d. U_c vs. N

CHAPTER 5
Conclusion

We have presented a functional language based programming methodology and a dataflow multiprocessor for digital continuous-system simulation. We have also developed an environment for detailed simulation and approximate performance analysis. The programming methodology and the multiprocessor, together with the support tools provide a useful environment for the formulation, experimentation, and validation of continuous-system models.

The interaction between the user and the tools in continuous-system modelling is organized in our environment as follows. A user begins with a dynamic system model coded in FPL augmented with a user-defined level of abstractions. The symbolic interpreter, and partitioning and allocation program will then generate the machine-level code according to the FPL specification and architectural parameters such as the number of available processing nodes. The user can either run a detailed simulation to obtain performance estimations, or use Eqs. (4.1) to (4.16) to predict the performance. If the performance is satisfactory, the user may experiment with the dynamic system model using different initial conditions and system parameters on the dataflow multiprocessor system. On the other hand, if the performance is not satisfactory, he may either reformulate the dynamic system model, or adjust the partitioning and allocation strategies to match the application, or under extreme circumstances to refine the numerical methods used to solve the differential equations.

We have stated in Chapter 1 and demonstrated in Chapter 2 that the programming methodology is suitable for *structurally determinate* type problems. Structurally determinate algorithms, coded in FPL, enable functional forms and routing functions to be symbolically executed before run-time. In the domain of numerical computations, the distinction between computational functions and routing functions is often clear. This may not be the case in applications where the storage characteristics (size and form) depend on the *values* of input or intermediate variables.

We admit that the *syntax* of FPL described in this thesis is not as appealing as CSSL in which users are supplied with operators such as INTEGRATE. However, it is our belief that the users would easily exploit the modular nature of FPL to customize compact functions and establish their own library functions. Incidentally, syntactic enhancement of FPL has been discussed in [Lu84].

A general-purpose machine is difficult to design and to build. However, by trading versatility and feasibility we are able to come up with a low-cost dataflow multiprocessor design to support the programming methodology.

Simulation of the architecture allows us to discover both bottlenecks and under-utilizations of various components. The deterministic simulation, described in Chapter 4, demonstrates that the dedicated-buses are under-utilized. This suggests that a shared-bus structure is preferrable.

The question of which scheme is better for token discrimination: *scheme A* or *scheme B* was not resolved by the simulation. Tables 4.2 and Table 4.3 show that both schemes are competitive to each other in terms of speed up and processors utilizations. However, Eqs. (4.4) and (4.11) show that as the number of processing nodes in the system grows, *scheme A* is subjected to a higher CIP utilization than *scheme B*. For a shared-bus system or system with slow buses, *scheme A* may be preferrable mainly because the amount of bus traffic required is less.

Several areas for further research were uncovered as a result of this work:

a. The performance analysis in Chapter 4 was presented under a specific set of architectural parameters and an arbitrary (manual) partitioning strategy. It requires more work to review the sensitivity of architectural performances to changes in the parameterization of the architecture and graph partitioning strategies.

b. Variable level granularity for the execution of dataflow graphs is proved in Chapter 4 (and by other researchers) to be desirable [Gaud84]. However, the desired level of granularity for a given set of architectural parameters remains to be investigated and formulated.

APPENDIX A

Coupled Pendulum Example

Continuous-system Simulation in FPL The Coupled Pendulum Example input structure: <<nθ1 ndθ1> <nθ2 ndθ2> h>
Initializations Defining Constants for the Parameters

```
define  zero( )
        %0
        end
define  two( )
        %2
        end
define  minusone( )
        %-1
        end
define  G( )
        %980.0
        end
define  X( )
        %44.0
        end
define  L( )
        %59.0
        end
define  D( )
        %40.0
        end
define  SLEN( )
        %48.0
        end
define  M1( )
        %10.0
        end
```

```
define  M2( )
        %10.0
        end
define  S1( )
        %48.0
        end
define  S2( )
        %48.0
        end
define  MU( )
        %0.03
        end
define  K( )
        %1.5
        end
```

Run Section
Integration of the ODE's

```
define  ddth( )
        *@[minusone,+@[*@[1,2],3]]
        end
define  ff1( )
        +@[*@[/@[*@[G,S1],*@[L,L]],sin@ 1],
        /@[*@[*@[SLEN,K],-@[y,X]],*@[*@[M1,L],L]]]
        end
define  ff2( )
        -@[*@[/@[*@[G,S2],*@[L,L]],sin@ 2],/@[*@[*@[SLEN,
        K],-@[y,X]],*@[*@[M2,L],L]]]
        end
define  y( )
        sqr@-@[+@[*@[D,D],pow@[*@[*@[sin@/@[-@[1,2],
        two],two],L],two]],*@[*@[*@[two,D],*@[*@
        [sin@/@[-@[1,2], two],two],L]],cos@/@[-@[1,2],two]]]
        end
define  bb( )
        /@[1,*@[2,pow@[3,two]]]
        end
define  tminush( )
        -@[time,h]
        end
define  Integral( )
        +@[*@[1,3],2]
        end
define  main( )
        [next, h, tminush]
        end
```

```
define  next( )
        [[nθ1,ndθ1],[nθ2, ndθ2]]
        end
define  timeup( )
        <@ [3 , zero]
        end
define  h( )
        2
        end
define  time( )
        3
        end
define  ndθ1( )
        Integral @ [ddθ1,dθ1,h]
        end
define  ndθ2( )
        Integral @ [ddθ2,dθ2,h]
        end
define  nθ1( )
        Integral @ [dθ1,θ1,h]
        end
define  nθ2( )
        Integral @ [dθ2,θ2,h]
        end
define  ddθ1( )
        ddth@[b1, dθ1, f1]
        end
define  ddθ2( )
        ddth@[b2, dθ2, f2]
        end
define  dθ1( )
        2@1@1
        end
define  dθ2( )
        2@2@1
        end
define  θ1( )
        1@1@1
        end
define  θ2( )
        1@2@1
        end
define  f1( )
        ff1@ [θ1, θ2]
```

```
        end
define  f2( )
        ff2@ [θ1, θ2]
        end
define  b1( )
        bb@ [MU, M1, L]
        end
define  b2( )
        bb@ [MU, M2, L]
        end
```

APPENDIX B
Machine Code for Processing Nodes

opcode	addressing mode	first operand	second operand	destination operand
sub	0	1	100.0	1
div	1	1	2.0	1
sin	1	1	0.0	1
mul	1	1	2.0	1
mul	1	1	59.0	1
pow	1	1	2.0	1
add	1	1	1600.0	1
eot				
sub	2	1	200.0	2
sqr	1	2	0.0	2
sub	1	2	44.0	2
mul	1	2	72.0	2
div	1	2	10.0	2
mul	1	2	59.0	2
mul	1	2	59.0	2
eot				
add	2	2	400.0	3
mul	3	300	0.0000008618	4
add	4	4	3.0	3
mul	1	3	-1.0	3
mul	7	3	501.0	3
add	2	3	300.0	9
inc	5			
out	1	1		
eot				

Machine-code for Processing Element # 1

opcode	addressing mode	first operand	second operand	destination operand
sub	0	1	100.0	1
div	1	1	2.0	1
sin	1	1	0.0	1
mul	1	1	2.0	1
mul	1	1	59.0	1
mul	1	1	80.0	1
eot				
mul	2	1	200.0	9
out	1	5		
eot				
mul	6	300	401.0	3
add	2	3	1.0	9
inc	5			
out	1	0		
eot				

Machine-code for Processing Element # 2

opcode	addressing mode	first operand	second operand	destination operand
sub	0	1	100.0	1
div	1	1	2.0	1
cos	2	1	0.0	9
out	1	9		
eot				
sin	0	1	1.0	2
mul	1	2	13.5	9
out	1	6		
eot				

Machine-code for Processing Element # 3

opcode	addressing mode	first operand	second operand	destination operand
sub	0	1	100.0	1
div	1	1	2.0	1
sin	1	1	0.0	1
mul	1	1	2.0	1
mul	1	1	59.0	1
pow	1	1	2.0	1
add	1	1	1600.0	1
eot				
sub	2	1	200.0	2
sqr	1	2	0.0	2
sub	1	2	44.0	2
mul	1	2	72.0	2
div	1	2	10.0	2
mul	1	2	59.0	2
mul	1	2	59.0	2
eot				
add	2	2	400.0	3
mul	3	300	0.000008618	4
add	4	4	3.0	3
mul	1	3	-1.0	3
mul	7	3	501.0	3
add	2	3	300.0	9
inc	5			
out	1	3		
eot				

Machine-code for Processing Element # 4

opcode	addressing mode	first operand	second operand	destination operand
sub	0	1	100.0	1
div	1	1	2.0	1
sin	1	1	0.0	1
mul	1	1	2.0	1
mul	1	1	59.0	1
mul	1	1	80.0	1
eot				
mul	2	1	200.0	9
out	1	7		
eot				
mul	6	300	401.0	3
add	2	3	100.0	9
inc	5			
out	1	2		
eot				

Machine-code for Processing Element # 5

opcode	addressing mode	first operand	second operand	destination operand
sub	0	100	1.0	1
div	1	1	2.0	1
cos	2	1	0.0	9
out	1	11		
eot				
sin	0	1	1.0	2
mul	1	2	13.5	9
out	1	8		
eot				

Machine-code for Processing Element # 6

Addressing mode	first operand	second operand
0	index base	index base
1	scratch-pad	direct
2	scratch-pad	index base
3	direct	index base
4	scratch-pad	scratch-pad
5	not used	
6	index base	direct
7	scratch-pad	index

Addressing Modes in the Multiprocessor

APPENDIX C

Linkage Tables

token	f1	f2	f3	f4	f5	f6	f7	f8
nθ1	1	1	100	200	0	7	15	1
ndθ1	300	200			15			1
nθ2	100	1	100	200	0	7	15	1
ndθ2								
h	500	200			15			1
071	200	100	200		7	15		1
015	400	200			15			1
175								
119								
068								
007								
172								

Linkage Table for Processing Node # 1

token	f1	f2	f3	f4	f5	f6	f7	f8
nθ1	1	1	100	200	0	7	10	1
ndθ1	300	200			10			1
nθ2	100	1	100		0	7		1
ndθ2								
h	400	200			10			1
071		200			10			1
015								
175								
119								
068	200	100			7			1
007								
172								

Linkage Table for Processing Node # 2

token	f1	f2	f3	f4	f5	f6	f7	f8
n01	1	1	100		0	5		1
nd01								
n02	100	1	100		0	5		1
nd02								
h								
071								
015								
175								
119								
068								
007								
172								

Linkage Table for Processing Node # 3

token	f1	f2	f3	f4	f5	f6	f7	f8
n01	1	1	100	200	0	7	15	1
nd01								1
n02	100	1	100	200	0	7	15	1
nd02	300	200			15			
h	500	200			15			1
071								
015								
175	200	100	200		7	15		1
119	400	200			15			1
068								
007								
172								

Linkage Table for Processing Node # 4

token	f1	f2	f3	f4	f5	f6	f7	f8
n01	1	1	100	200	0	7	10	1
nd01								
n02	100	1	100		0	7		1
nd02	300	200			10			1
h	400	200			10			1
071								
015								
175		200			10			1
119								
068								
007								
172	200	100			7			1

Linkage Table for Processing Node # 5

token	f1	f2	f3	f4	f5	f6	f7	f8
n01	100	1	100		0	5		1
nd01								
n02	1	1	100		0	5		1
nd02								
h								
071								
015								
175								
119								
068								
007								
172								

Linkage Table for Processing Node # 6

REFERENCES

[Arvi78] Arvind, K.P. Gostelow, and W. Plouffe, "An Asynchronous Programming Language and Computing Machine," Dept. of Information and Computer Science, University of California at Irvine, Tech. Rep. TR 114a, Dec. 1978.

[Arvi82] Arvind and Gostelow K.P. , "The U-Interpreter," *IEEE Computer* , Feb. 1982, pp. 42-49.

[Back78] J. Backus, "Can Programming be Liberated from the von Neumann Style ? A Function Style and Its Algebra of Programs," *Communications of the ACM*, Vol. 21, No. 8, Aug. 1978, pp. 613-641.

[Bueh82] R.E. Buehrer et al., "The ETH-Multiprocessor EMPRESS: A Dynamically Configurable MIMD System," *IEEE Trans. on Computers*, Vol. C-31, No. 11, Nov. 1982.

[Demi82] J. Deminet, "Experience with Multiprocessor Algorithms," *IEEE Trans. on Computers*, Vol. C-31, No. 4, Apr. 1982, pp. 278-288.

[Denn83] J.D. Dennis, W.Y.P. Lim, and W.B. Ackerman, "The MIT Data Flow Engineering Model," *Proceedings of the IFIP*, 1983, pp. 553-563.

[Erce83] M.D. Ercegovac and S.L. Lu, "A Functional Language Approach in High-Speed Digital Simulation," *Summer Computer Simulation Conference*, 1983, pp. 383-387.

[Erce84] M.D. Ercegovac and W.J. Karplus, "A Data Flow Approach in High-Speed Simulation of Continuous Systems," *Proc. Intl. Workshop on High-Level Computer Architecture 84*, May. 1984, pp. 2.1-2.8.

[Ferr83] D. Ferrari et al., *Measurement and Tuning of Computer Systems*: Prentice-Hall, Inc., 1983.

[Requ83] J.E. Requa, "The Piecewise Data Flow Architecture Control Flow and Register Management," *The 10th Annual Intl. Symp. on Computer Architecture*, Vol. 11, No. 3, Jun. 13-17, 1983, pp. 84-89.

[Ritc78] D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "UNIX Time-Sharing System: The C Programming Language," *Bell Sys. Tech. J.*, Vol. 57, No. 6, 1978, pp. 1991-2019.

[Schl83] M.D.F. Schlag, "Extracting Geometry from FP Expressions," University of California, Los Angeles, Los Angeles, CA, Tech. Rep. (Internal Report), Dec. 1983.

[Toko83] M. Tokoro, J.R. Jagannathan, and H. Sunahara, "On the Working Set Concept for Data-Flow Machines," *The 10th Annual Intl. Symp. on Computer Architecture*, Vol. 11, No. 3, Jun. 13-17, 1983, pp. 90-97.

[Trel80] P. Treleaven and G. Mole, "A Multi-Processor Reduction Machine for User-Defined Reduction Languages," *Proc. 7th Annual Symp. on Computer Arch.*, 1980, pp. 121-130.

[Worl83] J. Worley, K.G. Tu, and J. Arabe, *The Architecture and Design of the Functional Programming Machine*, UCLA Computer Science Department internal memorandum (Dec. 1983).

[Worl84] J. Worley, *The UCLA T-FP User Manual*, Los Angeles, CA: University of California, Los Angeles, Jun. 1984.

[Fran78] M.A. Franklin, "Parallel Solution of Ordinary Differential Equations," *IEEE Trans. on Computers*, Vol. C-27, No. 5, May. 1978, pp. 413-420.

[Frie78] D.P. Friedman and D.S. Wise, "Aspects of Applicative Programming for Parallel Processing," *IEEE Trans. on Computers*, Vol. C-27, No. 4, Apr. 1978, pp. 289-296.

[Gajs82] D.D. Gajski, D.A. Padua, D.J. Kuck, and R.H. Kuhn, "A Second Opinion on Data-Flow Machines and Languages," *IEEE Computer*, Feb. 1982, pp. 58-69.

[Gaud84] J.-L. Gaudiot and M.D. Ercegovac, "Performance Analysis of a Data Flow Computer with Variable Resolution Actor," *Proc. 4th Intl. Conf. on Distributed Computing*, 1984.

[Gurd80] J. Gurd and I. Waton, "Data Driven System for High Speed Parallel Computing - Part 2: Hardware Design," *Computer Design*, Jul. 1980, pp. 97-106.

[Kell83] J.N. Kellman, "Parallel Execution of Functional Programs," University of California, Los Angeles, Los Angeles, CA, Tech. Rep. CSD-830114, Jan. 1983.

[Korn78] G.A. Korn and J.V. Wait, *Digital Continuous-system Simulation*: Prentice-Hall, Inc., 1978.

[Lu84] S.L. Lu, "A Compiler for A Functional Programming System," Univeristy of California, Los Angeles, Los Angeles, CA, Tech. Rep. Master Thesis, Apr. 1984.

[Mago80] G.A. Mago, "A Cellular Computer Architecture for Functional Programming," *Proceedings of the COMPCON Fall 1980*, 1980, pp. 179-187.

[Mako83] A. Makoui and W.J. Karplus, "Data Flow Methods for Dynamic System Simulation : A CSSL-IV Microcomputer Network Interface," *SCSC*, Jul. 1983, pp. 376-382.

[Pate81] D.R. Patel, "A System Organization for Applicative Programming," University of California, Los Angeles, Los Angeles, CA, Tech. Rep. CSD-810302, Mar. 1981.

[Ravi84] T.M. Ravi, "Partitioning and Allocation of Functional Programs for Data Flow Processors," University of California, Los Angeles, Los Angeles, CA, Tech. Rep. (in preparation), 1984.