An Abstract of the Thesis of

Vincent R. Freytag for the degree of Masters of Science in Electrical and Computer Engineering presented on March 18, 1993.

Title: Program Allocation for Hypercube Based Dataflow Systems

Abstract approved: _____
Ben Lee

The dataflow model of computation differs from the traditional control-flow model of computation in that it does not utilize a program counter to sequence instructions in a program. Instead, the execution of instructions is based solely on the availability of their operands. Thus, an instruction is executed in a dataflow computer when all of its operands are available. This asynchronous nature of the dataflow model of computation allows the exploitation of fine-grain parallelism inherent in programs. Although the dataflow model of computation exploits parallelism, the problem of optimally allocating a program to processors belongs to the class of NP-complete problems. Therefore, one of the major issues facing designers of dataflow multiprocessors is the proper allocation of programs to processors.

The problem of program allocation lies in maximizing parallelism while minimizing interprocessor communication costs. The culmination of research in the area of program allocation has produced the proposed method called the Balanced Layered Allocation Scheme that utilizes heuristic rules to strike a balance between computation time and communication costs in dataflow multiprocessors. Specifically, the proposed allocation scheme utilizes Critical Path and Longest Directed Path heuristics when allocating instructions to processors. Simulation studies indicate that the proposed scheme is effective in reducing the overall execution time of a program by considering the effects of communication costs on computation times.

Program Allocation for Hypercube Based Dataflow Systems

by

Vincent R. Freytag

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed March 18, 1993

Commencement June 1993

Approved:

Redacted for Privacy

Assistant Professor of Electrical and Computer Engineering in charge of major

Redacted for Privacy

Head of department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented March 18, 1993

Thesis written and typed by Vincent R. Freytag

Acknowledgements

The success of this research would not have been possible without the effort of others before me and the direction given to me by my colleagues, friends, and family. My special thanks go to Dr. Ben Lee for laying out much of the ground work in this project. Dr. Lee's guidance, patience, and expertise have been extremely valuable to my studies at Oregon State University. Special thanks to my committee members Dr. Sayfe Kiaei, Dr. Shih-Lien Lu, and Professor Robert Schultz. I would also like to thank my family for their unconditional support and especially my wife Lupita for her encouragement and support. This thesis is dedicated to my father who has inspired my pursuit of knowledge.

Table of Contents

List of Figures

Page

List of Tables

Page

List of Algorithms

Program Allocation for Hypercube Based Dataflow Systems

1. Introduction

The need for high-speed computing has lead to the development of general-purpose multiprocessor systems whose primary goal is the exploitation of parallelism. The shift from single processor machines to multiprocessor machines has been facilitated by performance limits imposed by fundamental electrical properties, such as switching speeds and propagation delays [20]. The advent of multiprocessor machines has made it possible to reduce program execution times by concurrently executing programs on several processors.

Conventional multiprocessors are based on the von Neumann model of computation (*i.e.*, control-flow model of computation). The simplicity offered by the von Neumann model of computation is characterized by a single, global addressable program/data memory and a program counter (PC) that threads the execution of tasks in a sequential manner. When the control-flow model is extended to multiple processors, computations communicate through the shared memory and parallelism is achieved by allowing more than one thread of control to be active at any instance on the system.

Although several successful multiprocessors based on the von Neumann model of computation have been constructed, their performance is limited by the constraints of the control-flow execution model [6, 10, 20]. One serious problem with distributing work over several von Neumann processors is the implied global, shared memory. A single processor can mask the time to fetch an item from memory with a variety of techniques such as registers, caches, pipelines, etc. However, when there are multiple processors in a system, parallel tasks may require simultaneous access to a shared memory cell or one task may require the result of another task. A number of *synchronization* methods are available to enforce the correct sequencing of processes and to ensure the mutual

exclusion of shared data [6, 7]. However, synchronization introduces computational overhead and increases the complexity of program development. Due to the large overhead of synchronization, high performance can be achieved only when the program is partitioned into long threads with few synchronization operations. Thus, synchronization imposes a limit on the exploitation of concurrency.

A more subtle concern with a von Neumann multiprocessor is the PC. The single locus of control in a PC implies a bottleneck in the scheduling of instructions. A processor that uses a PC to access the next executable instruction must decode instructions sequentially to guarantee the logical correctness of the results. Once decoded, the instructions may be pipelined or executed out of order depending on the availability of operands (*e.g.*, superscalar processors). A somewhat higher performance can be obtained by decoding several instructions simultaneously, however it is still necessary to sequentially scan the relative order of the instructions [2]. Thus, parallelism can only be exploited around the current point of control (*i.e.*, the PC). Due to the sequential nature of the control-flow model, parallelism must be explicitly defined by the programmer. This places an overwhelming burden on the programmer to detect embedded parallelism and to ensure the logical correctness of program execution.

Another method of exploiting parallelism in a von Neumann multiprocessor focuses on the programming languages for these machines. Over the years, attempts have been made to design compilers that optimize programs from conventional languages (*e.g.*, vectorizing compilers for FORTRAN). How effectively a computer handles the combination of vector and scalar operations is the key to its performance [20]. The percentage of code that is inherently scalar ranges from ten to ninety percent across a broad range of scientific applications [2]. The code that cannot be vectorized must run on the scalar portion of the system, resulting in an increase in overall execution time, and thus a bottleneck on a vector machine.

Although the von Neumann model of computation has become the standard for uniprocessor machines, many experts agree that this model does not carry over to multiprocessors [2, 3, 13, 27]. An alternative to the von Neumann model of computation is the dataflow model of computation. The dataflow model of computation can maximally exploit parallelism in a program. In addition, the functional and asynchronous characteristics of the dataflow model of computation overcome many of the problems associated with the control-flow method of exploiting parallelism. First, there is no concept of a shared storage. Instead, operands are communicated as *tokens* of values rather than addresses of variables. Thus, the dataflow model of computation is functional in the sense that its operations do not produce side-effects such as the inadvertent modification of a shared variable. Second, there is no concept of a PC. Instead, the execution of instructions are based solely on the availability of their operands. It is through this property that dataflow operations are asynchronous.

Although the dataflow model of computation has several advantages, some problems remain to be solved before dataflow computers become a practical alternative to the more traditional methods of parallel computing. One challenge yet to be resolved is the efficient partitioning and mapping of a dataflow program to a dataflow computer [2]. The aim of our research is to develop an allocation scheme that partitions a program graph and assigns the program modules to processors while controlling and supporting large amounts of interprocessor communication. The present work began in 1991 and extends the work done by Lee *et al.* in the area of processor allocation for dataflow systems [15].

To describe the work of our research, basic dataflow principles are discussed in Chapter 2. The discussion elaborates on the dataflow model of computation and the importance of dataflow graphs. Additionally, this chapter includes a brief summary of dataflow languages and the difficulties of handling data structures in a dataflow environment. Chapter 3 discusses two approaches used to realize the dataflow model of

computation, namely, the static dataflow model and the dynamic dataflow model. Three dataflow computers and their respective architectures are then presented. The classical Tagged-Token Dataflow Architecture is discussed first followed by a discussion of a more recent dataflow computer, the Monsoon. This chapter concludes with the discussion of a hybrid dataflow computer, the Epsilon-2. Chapter 4 discusses why hypercubes are an excellent topology for a multicomputer system, including a dataflow environment. In Chapter 5 we turn to the issue of program allocation. The allocation problem and a brief discussion of prior work in this area is presented. We then propose an allocation scheme that efficiently maps dataflow programs onto dataflow computers while controlling large amounts of interprocessor communication. The effectiveness of the proposed allocation scheme is then analyzed through simulation results. Finally, we provide a brief conclusion and give direction for further study in Chapter 6.

2. Dataflow Principles

The dataflow model of computation deviates from the more traditional control-flow model in two fundamental principles: asynchrony and functionality. First, dataflow operations are asynchronous in that an operation *fires* (executes) only when all the required operands are available. In contrast to the control-flow model of computation, the dataflow model of computation does not utilize a program counter (PC) to sequence instructions. Instead, the execution of instructions is based solely on the *availability* of their operands. Thus, the dataflow model of computation has decentralized the locus of control from a PC to the parallelism embedded within an application program.

Second, the dataflow model of computation is functional in that it does not produce any side-effects. Two or more instructions may be executed in any order or concurrently if their operands are available. Thus, synchronization takes place at the instruction level. Additionally, the dataflow model of computation has no concept of a shared memory variable. Instead, operands are communicated as *tokens* of *values* rather than addresses of variables. Since a token is value oriented, a result token of an operation can be applied to several different functions without the possibility of side-effects.

2.1 Dataflow Graphs

In a dataflow computer, the machine-level program is represented by a *dataflow graph*. In a dataflow graph, nodes represent instructions and arcs represent data dependencies between the instructions. Thus, an arc from one node to another node implies that the successor instruction can not fire until the predecessor instruction has fired. Since dataflow graphs encode the data dependencies of a program, they can be interpreted as a machine language for dataflow computers. More concisely, dataflow computers are stored program computers in which the stored program is in the form of a

dataflow graph [1]. The importance of the dataflow graph can be seen by examining a simple sequence of assignment statements typical of high-level programming languages. For example, consider the following generic code segment:

```
10    A = X + Y
20    B = A * X
30    C = A / Y
40    D = B + C
50    E = B * C
60    RESULT = D / E
```

A traditional execution of this code would execute the first statement to completion, then the second statement, etc., until the **RESULT** is computed. However, further analysis of this program fragment will reveal that several of these instructions can be executed concurrently. For example, one possible sequence would be **10**, **20** and **30** simultaneously, **40** and **50** simultaneously, **60**. These sequencing constraints can be represented by a dataflow graph as shown in Figure 2.1.

From Figure 2.1 it can be seen that line **10** of the sample code depends on the values of **X** and **Y**. Therefore, line **10** can execute once the values of **X** and **Y** are available. Similarly, line **20** depends on the completion of line **10** and the availability of **X**. Line **30** also depends on the completion of line **10** and the availability of **Y**. Since lines **20** and **30** do not depend on each other, but require the values **X**, **Y**, and the outcome of line **10**, they may execute concurrently. Likewise, lines **40** and **50** can execute concurrently. Since line **60** depends on the values generated from lines **40** and **50**, line **60** can not execute until both line **40** and line **50** have computed their results.

More formally, a dataflow program is represented as a directed graph $G \equiv G(N,A)$, where the nodes, N, represent instructions or functions (*i.e.*, macro-actors) and the arcs, A, represent the data dependencies between nodes. Operand values are passed along the arcs in the form of data packets, called tokens. Theoretically, the arcs are assumed to be FIFO (first in first out) queues with unlimited capacity. A node fires when it has received all the necessary input tokens to perform its operation. When a node

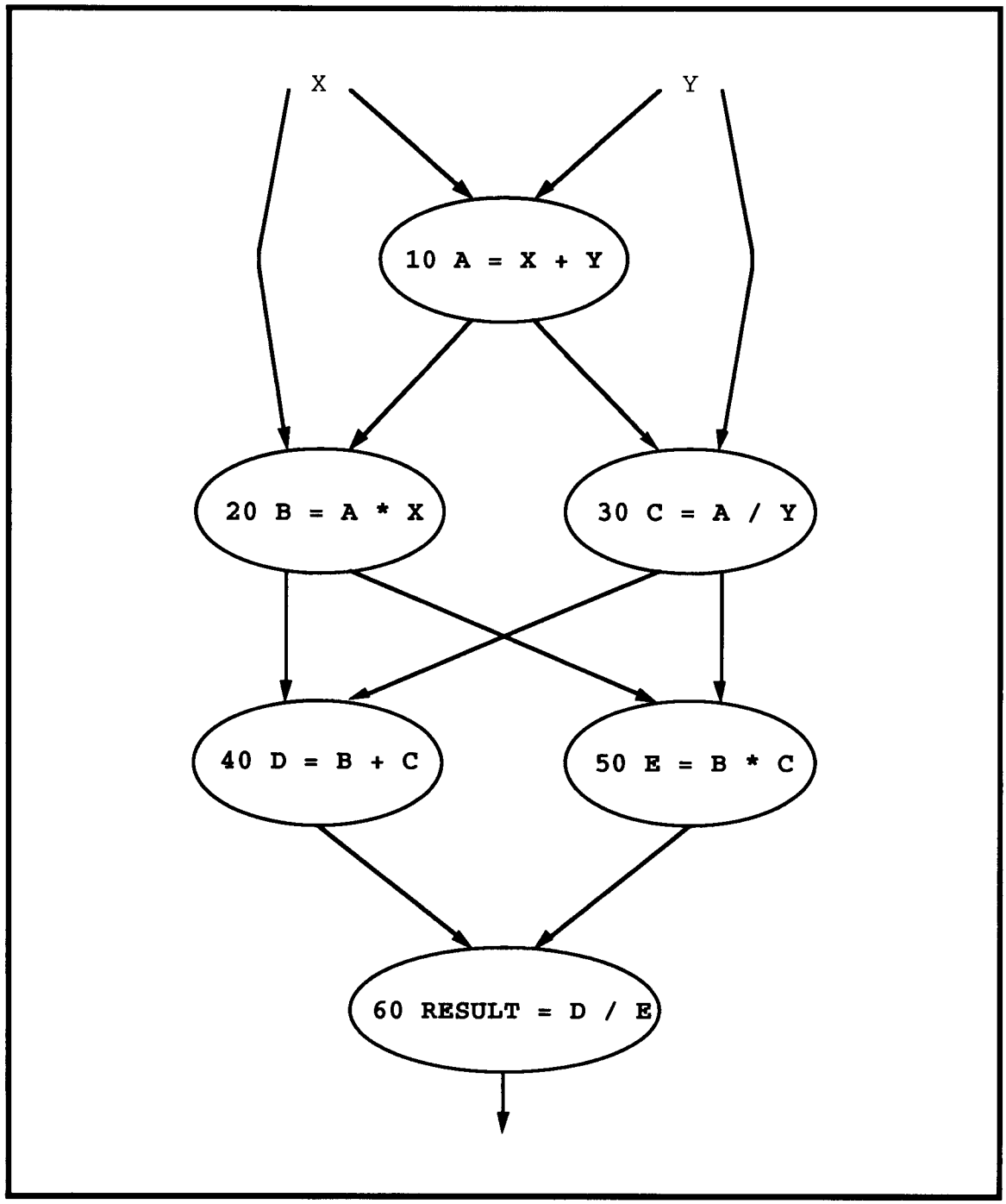


Figure 2.1 A dataflow graph illustrating sequencing constraints in assignment statements.

fires, the input tokens are consumed and one or more output tokens are generated. Instructions executed on dataflow computers do not have any set sequencing constraints except the data dependencies implicit in the program itself. Since the data dependencies are encoded in a dataflow graph, explicit synchronization is not needed. Thus, a dataflow graph can exploit all possible parallelisms of a given program.

A node that is enabled by the availability of all its operands will fire by consuming all its input tokens, computing a result value, and producing a result token on each of its output arcs. This dictates a basic instruction cycle for a dataflow computer. The instruction cycle consists of:

- Detecting when an operation is enabled (*i.e.*, all input tokens are available).
- Determining the operation to be performed (*i.e.*, fetch the instruction).
- Computing the result.
- Generating the result tokens.

The arcs in a dataflow graph may transmit either numerical values or boolean values. The set of dataflow primitives is shown in Figure 2.2. A numerical value is produced by an *Operator* as a result of some operation, *f*. Boolean control values of either **TRUE** or **FALSE** are generated by a *Decider* that generates a result from applying a predicate *P* to the values consumed on its input arcs. The special primitives *Switch* and *Merge* are used to direct data values and handle conditional statements and loops within a dataflow graph. The Switch primitive directs an input data token to either its **TRUE** or **FALSE** output arc depending on the boolean control value it receives. For example, when a **TRUE** value is received on the Switch control arc, the Switch primitive transfers the input token to its **TRUE** output arc and places nothing on its **FALSE** output arc. Similarly, when a **FALSE** value is received on its control arc, the input token is transferred to the **FALSE** output arc and nothing is placed on the **TRUE** output arc. The Merge primitive is the inverse of the Switch primitive. The Merge primitive transfers an input token from the **TRUE** or the **FALSE** input arc to its output arc, according to the boolean value on the

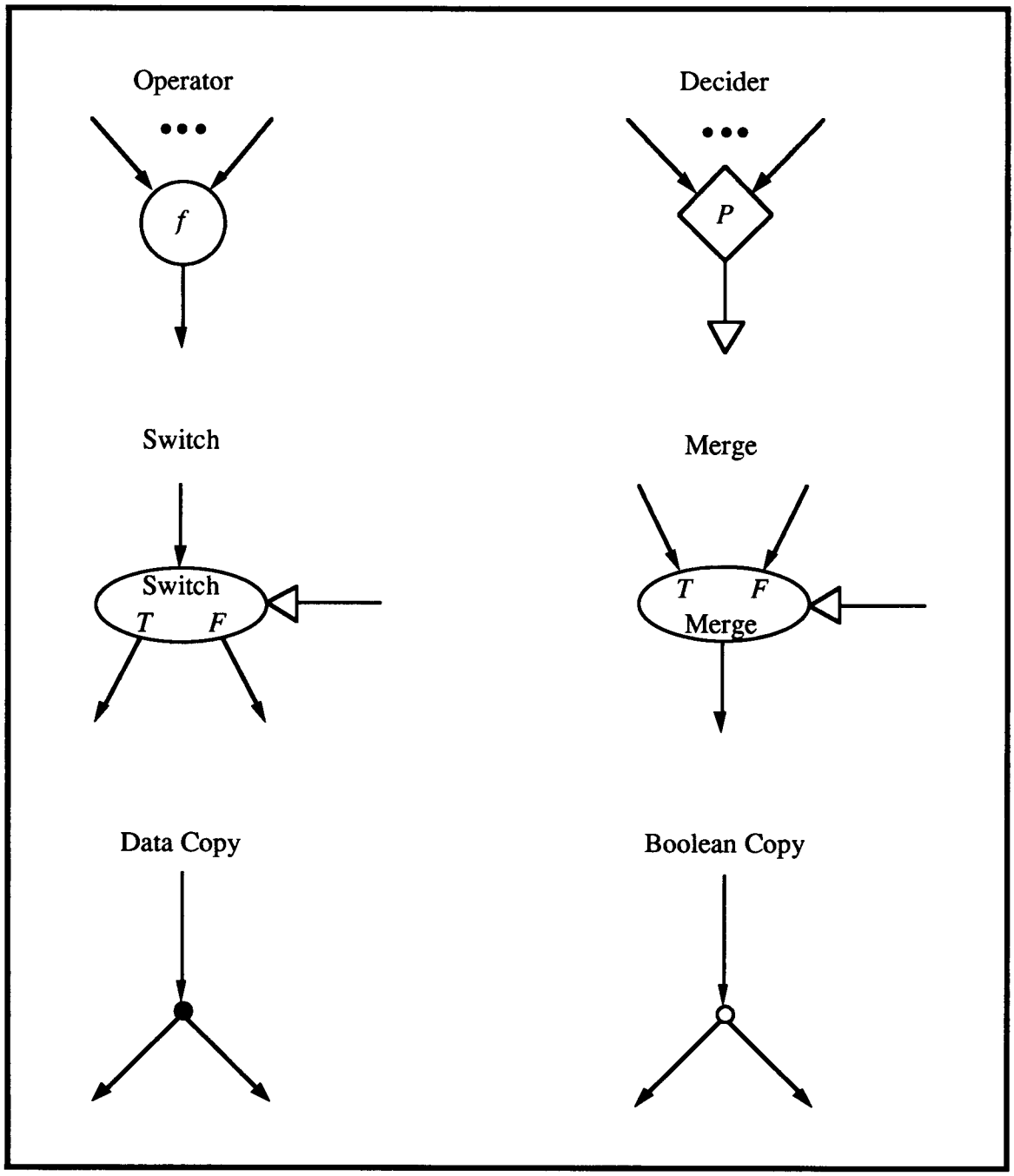


Figure 2.2 The set of dataflow primitives used to construct dataflow graphs.

control arc. Finally, the *Data Copy* primitive duplicates integer, real, or complex numbers while the *Boolean Copy* duplicates boolean values.

An example from Arvind and Culler [3] demonstrates how these primitives are used to implement a dataflow graph for the conditional statement:

```
IF (X < Y)  Output = X + Y;
ELSE Output = X - Y;
```

The dataflow graph for this conditional statement is shown in Figure 2.3.

A dataflow graph is considered *well-behaved* if a single wave of tokens on the input arcs produces a single wave of tokens on the output arcs [3]. In other words, if a wave of inputs is placed on the input tokens, then a wave of appropriate outputs should appear on the output arcs of the dataflow graph. It can be shown that the dataflow graph of Figure 2.3 is well-behaved. To show this, consider the case where X < Y is the first wave of input tokens to the dataflow graph and X > Y is the second wave of input tokens to the dataflow graph. Suppose that delays in the plus and minus operators cause the token for the **FALSE** side of the Merge to arrive before the tokens on the **TRUE** side. The sequence of control tokens at the Merge will restore the output tokens to their correct order.

The two main properties of dataflow can now be summarized. First, instructions may execute in parallel unless data dependencies do not allow it. Second, results do not depend on the relative order in which potentially parallel nodes execute [3].

2.2 Dataflow Languages

There is a special need to provide a high-level language for dataflow computers since the dataflow graph (*i.e.*, machine-level program representation) is not an efficient programming method. One method considered by researchers is to program dataflow computers in conventional *imperative* languages. With this method, a program is mapped

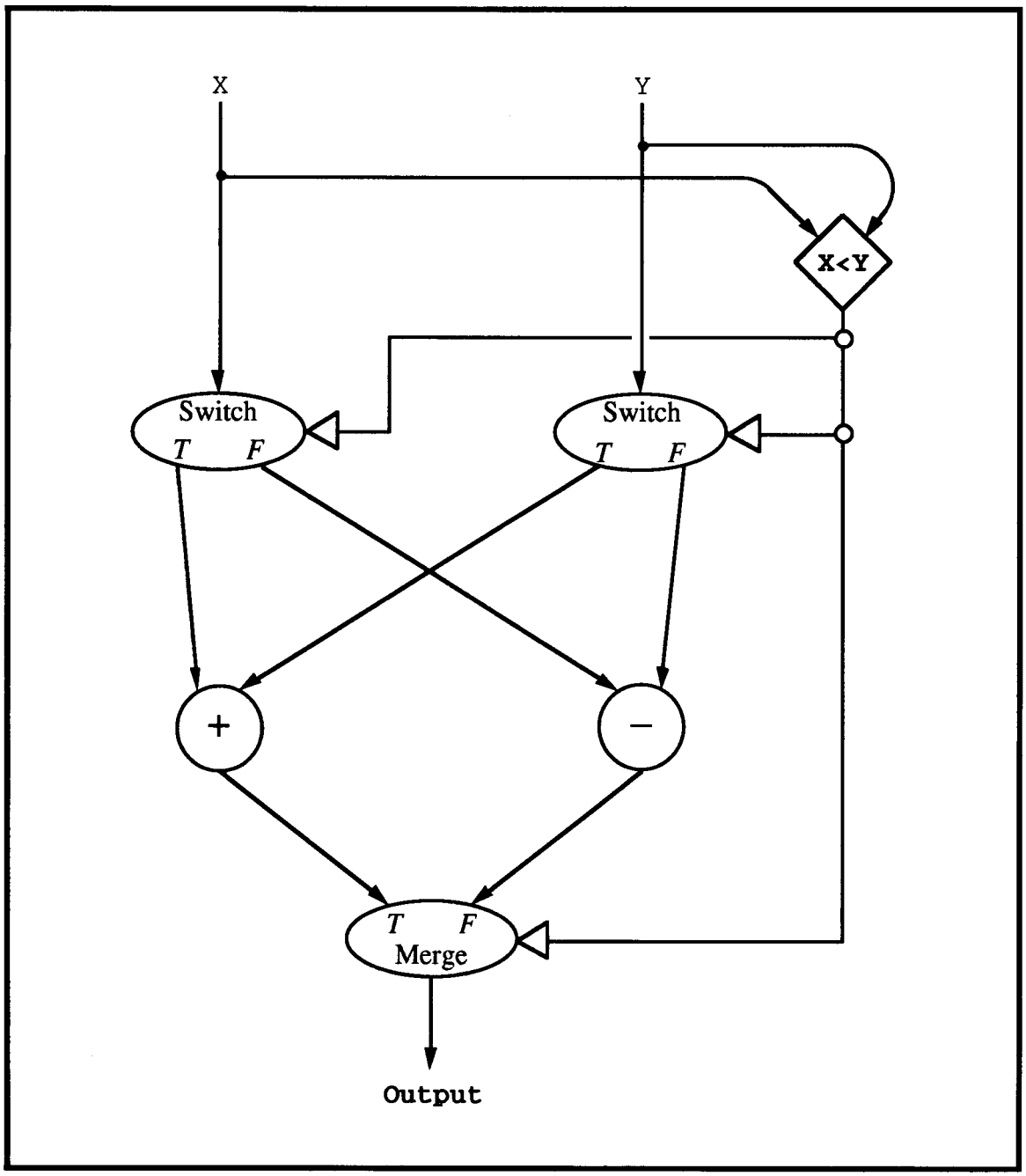


Figure 2.3 A conditional dataflow graph.

from a conventional language to a directed dataflow graph by using dataflow analysis. However, this approach is inefficient since conventional imperative languages are sequential in nature [1]. Like other parallel computers, dataflow computers are best programmed in special languages. Therefore, a number of dataflow or *applicative* languages have been developed to provide a more efficient way of expressing parallelism for dataflow computers. Examples of dataflow languages include the Value Algorithmic Language (VAL), Irvine Dataflow language (Id), and Stream and Iteration in a Single-Assignment Language (SISAL) which have been proposed by dataflow projects at MIT, the University of California at Irvine, and Lawrence Livermore National Laboratories, respectively [1].

Languages suitable for dataflow computers can be very elegant and have several useful properties [1]. The discussion of functionality that guarantees freedom from side-effects has already been presented. However, for the sake of completeness, the relevant properties to keep in mind when discussing dataflow are:

- *Freedom from side-effects.* This property is necessary to ensure that the data dependencies are consistent with the sequencing constraints [1, 12]. A side-effect free program guarantees that there is no possibility of corrupted memory (*i.e.*, variable) locations. The most common side-effects are caused from procedures that modify variables in the calling code. However, since dataflow computers are value-driven, dataflow languages do not allow the modification of variables. This restriction eliminates the possibility of side-effects in a dataflow computer.
- *Locality of effect.* Dataflow languages generally exhibit considerable locality [1]. This property means that variables do not have far reaching dependencies. That is, variables have a limited scope for which they are valid. Outside this range they are no longer valid and preferably no longer used.

- *Single assignment rule.* The single assignment rule provides a method to promote parallelism. Simply stated, a variable can only appear to the left side of an assignment statement once. This restriction eliminates the problem of *aliasing*. An aliased variable is a variable that is used for two unrelated computations. In the past, the practice of variable aliases was common to save memory space. Alleviating variable aliases and enforcing a single assignment rule aids the compiler in identifying embedded parallelism.

2.3 Data Structures

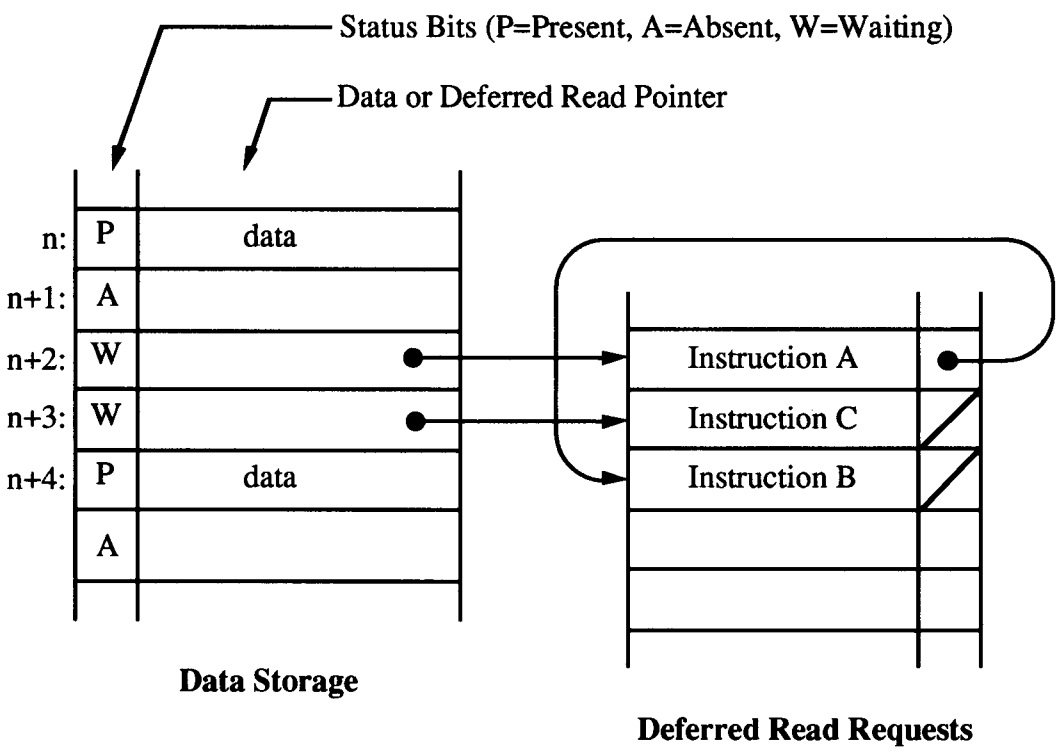
The ability for dataflow languages to be side-effect free resides in the dataflow model of computation where operations consume input tokens and generate one or more output tokens. However, if tokens are allowed to carry arrays or other complex structures, the absence of side-effects implies that an operation on a structure element must result in an entirely new structure. This solution is acceptable theoretically, but would create excessive system-level overhead. A data dependency problem may also exist when data structures are implemented on a parallel machine. If function "F" fills an array with values one at a time and then passes the array onto function "G," that reads the elements one at a time, "G" cannot begin until "F" completes (*i.e.*, strict access).

A technique to handle data structures while preserving the functionality of dataflow operations is called I-structures [1, 3]. From a programmer's perspective, an I-structure is an array of slots that are initially empty, and that can be written at most once. An I-structure is asynchronous in the sense that the construction of structures is not strictly ordered; therefore, it is possible for a process to attempt to select an element before that element has been written (*i.e.*, non-strict access). Thus, if a *read request* arrives for a storage cell that does not have valid data, the controller defers the read until

a write to that particular storage cell arrives. To implement this concept, each storage cell contains status bits to indicate that the cell is in one of three possible states:

- PRESENT: The cell contains valid data that can be read as in conventional memory.
- ABSENT: No data has been written into the cell since it was last allocated and no attempt has been made to read the cell.
- WAITING: No data has been written into the cell, but at least one attempt has been made to read it. Such read requests are deferred to a *deferred read request list* that must be satisfied when the status bit changes from WAITING to PRESENT.

An example of I-structure storage is illustrated in Figure 2.4.



Possible sequence of instructions producing this structure:

- Attempt to **`READ(n+2)`** for instruction A
- **`WRITE(n+4)`**
- Attempt to **`READ(n+3)`** for instruction C
- **`WRITE(n)`**
- Attempt to **`READ(n+2)`** for instruction B
- **`READ(n)`**

Figure 4. An example of I-structure storage.

Figure 2.4 An example of I-structure storage.

3. Dataflow Architectures

In the abstract dataflow model of computation, data values are carried on tokens which travel along arcs that connect data dependent instructions in a program graph. It is assumed that the arcs are unbounded FIFO queues. However, a direct implementation of this model is impossible since an unbounded FIFO queue can not be realized. Instead, two different approaches have been developed for handling token transmission and storage. A *static* approach allows storage for only one token per arc. With this approach, the firing rule is modified so that an operator is executed when all its tokens are available on its input arcs and no tokens exist on any of its output arcs. Conversely, a *dynamic* approach allows storage for multiple tokens on an arc. In this approach, a *tag* is associated with each token that identifies the instance in which it was generated. Dataflow operators are then executed when its input arcs contain a set of tokens with matching tags.

3.1 Static and Dynamic Dataflow Models

The *static dataflow model* was proposed by Dennis and his colleagues at MIT [3]. The basic organization of the static dataflow model is illustrated in Figure 3.1. The Program Memory holds *instruction templates* that represent nodes in a dataflow graph. Each instruction template has slots for an opcode, operands, destination address(es), and acknowledge address(es). The operand slots have *presence flags* to determine the availability of the operands. Addresses of enabled instructions reside in the Instruction Queue. The Fetch Unit removes the first address from the Instruction Queue, and fetches the corresponding instruction from the Program Memory. The Fetch Unit clears the presence flags of the fetched instruction and forms an *operation packet* containing the opcode, operands, and destination list. The operation packet is forwarded to an operation

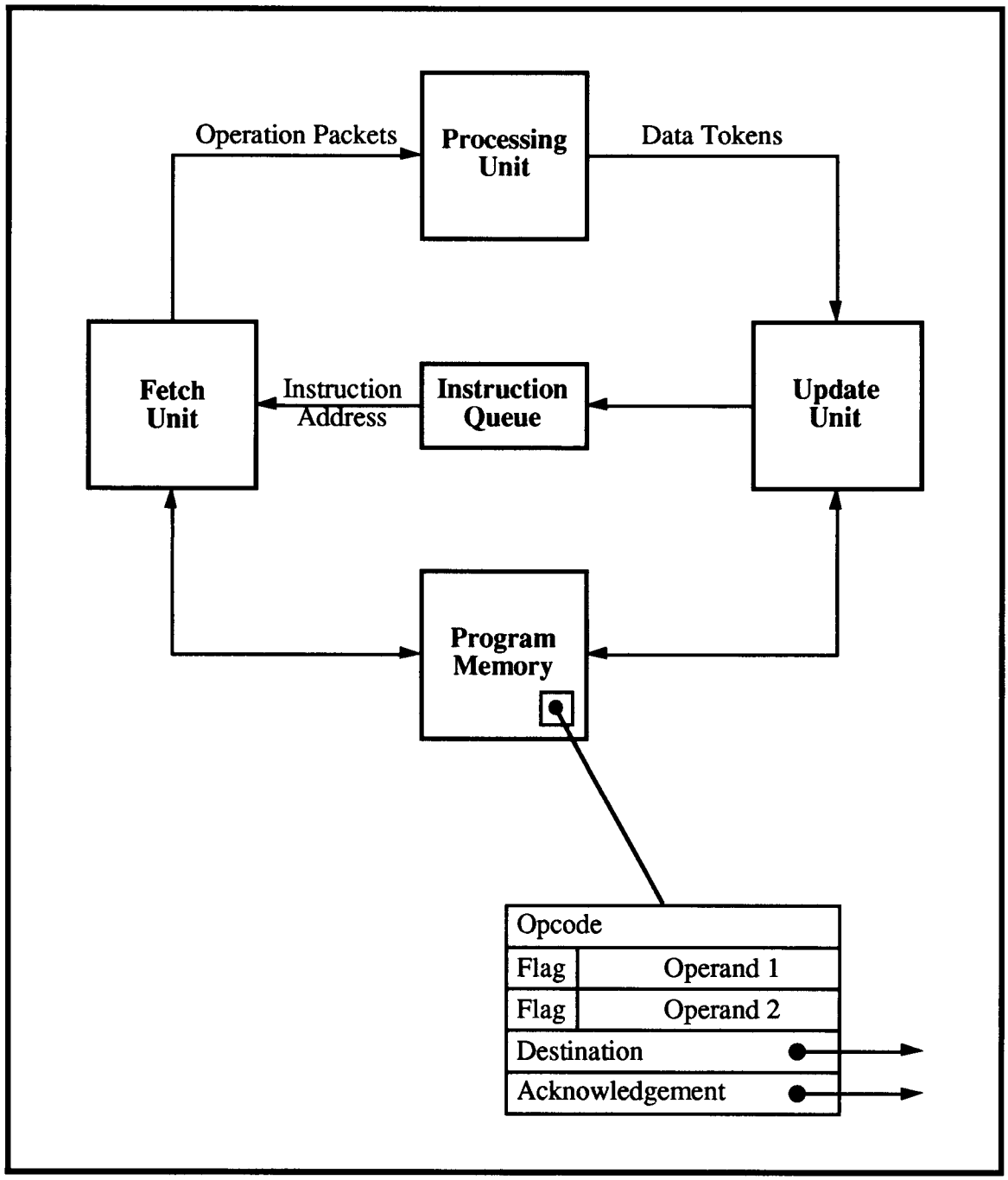


Figure 3.1 The basic organization of the static dataflow model.

unit available within the Processing Unit. The Processing Unit computes a result and generates tokens for each destination. These result tokens are then sent to the Update Unit. The Update Unit stores the results in the Program Memory, and checks the presence flags to determine if the corresponding instruction is enabled. If the instruction is enabled, the address of the instruction is placed in the Instruction Queue.

The static dataflow model is characterized by the one-token-per-arc restriction. This is enforced by requiring all output arcs of a node to be empty before it can fire (*i.e.*, static firing rule). Thus, data dependent nodes communicate with a send-acknowledge protocol. Even with these assumptions, multiple tokens belonging to the same arc may coexist in the machine since there may be buffering in the units and communication network. If multiple tokens can exist on an arc, then two firings of a single node can execute on different operation units and the instruction that is logically second in the queue may finish first [3]. Although the result tokens will be sent to the same destination node, they will arrive in the wrong order. Thus, buffering within the system may ultimately lead to nondeterminacy.

If the one-token-per-arc restriction can be enforced, then problems due to the incorrect sequence of arriving tokens will not arise. Additionally, this restriction allows a fixed amount of storage to be allocated at compile-time since the number of arcs in a program graph remain constant. The one-token-per-arc restriction can be achieved by a simple transformation of the program graph: for each arc in the program graph, an *acknowledgement arc* is added in the opposite direction. A token on an acknowledgement arc indicates that its corresponding input arc is empty. Thus, a node may fire when a token is present on each input arc and on each incoming acknowledgement arc.

The transformation of a program graph to enforce the one-token-per-arc restriction is not completely satisfactory. Even though many of the acknowledgement arcs in a program graph can be eliminated, the amount of token traffic increases by a

factor of 1.5 to 2, the time between successive firings of a node increases drastically, and most importantly, the amount of parallelism that can be exploited in a program is reduced [3]. Additionally, the dynamic unfolding of loops is hindered with the static dataflow model due to the strict enforcement of the static firing rule. To illustrate this, consider the loop of Figure 3.2. It should be possible to pipeline four distinct computations through the body of the loop, but with the static approach the second initiation must wait until the divide node fires, clearing the input arc to the divide operator. These shortcomings motivated work on the dynamic dataflow model.

The dynamic dataflow model was proposed by Arvind at MIT [3]. The basic organization of the dynamic dataflow model is illustrated in Figure 3.3. In this model, tokens are received by the Matching Unit, which is a memory containing a pool of waiting tokens. The function of the Matching Unit is to identify tokens with identical tags. If a match exists, the token pair is extracted from the Matching Unit and passed on to the Fetch Unit. If no match is found, the token is stored in the Matching Unit to await a partner. In the Fetch Unit, the tags of the token pair uniquely identify an instruction to be fetched from the Program Memory. The instruction, combined with the token pair, forms an enabled instruction and is sent to the Processing Unit. The Processing Unit produces result tokens that are sent to the Matching Unit.

In the dynamic dataflow model, tokens may coexist on a single arc. No control tokens are needed to acknowledge the transfer of tokens among instructions. Instead, multiple tokens are distinguished by their tags. In addition to specifying the destination node, a tag also specifies a particular firing of the node. In the dynamic dataflow model, a node may fire if and only if two tokens have the same tag. Thus, tags guarantee the correct firing sequence of nodes in a system where multiple tokens may coexist on an arc. Tags have four parts: invocation ID, iteration ID, code-block, and instruction address [3]. The first pair identify a particular firing of the instruction and the latter pair identify the destination instruction. The iteration ID distinguishes between different iterations of a

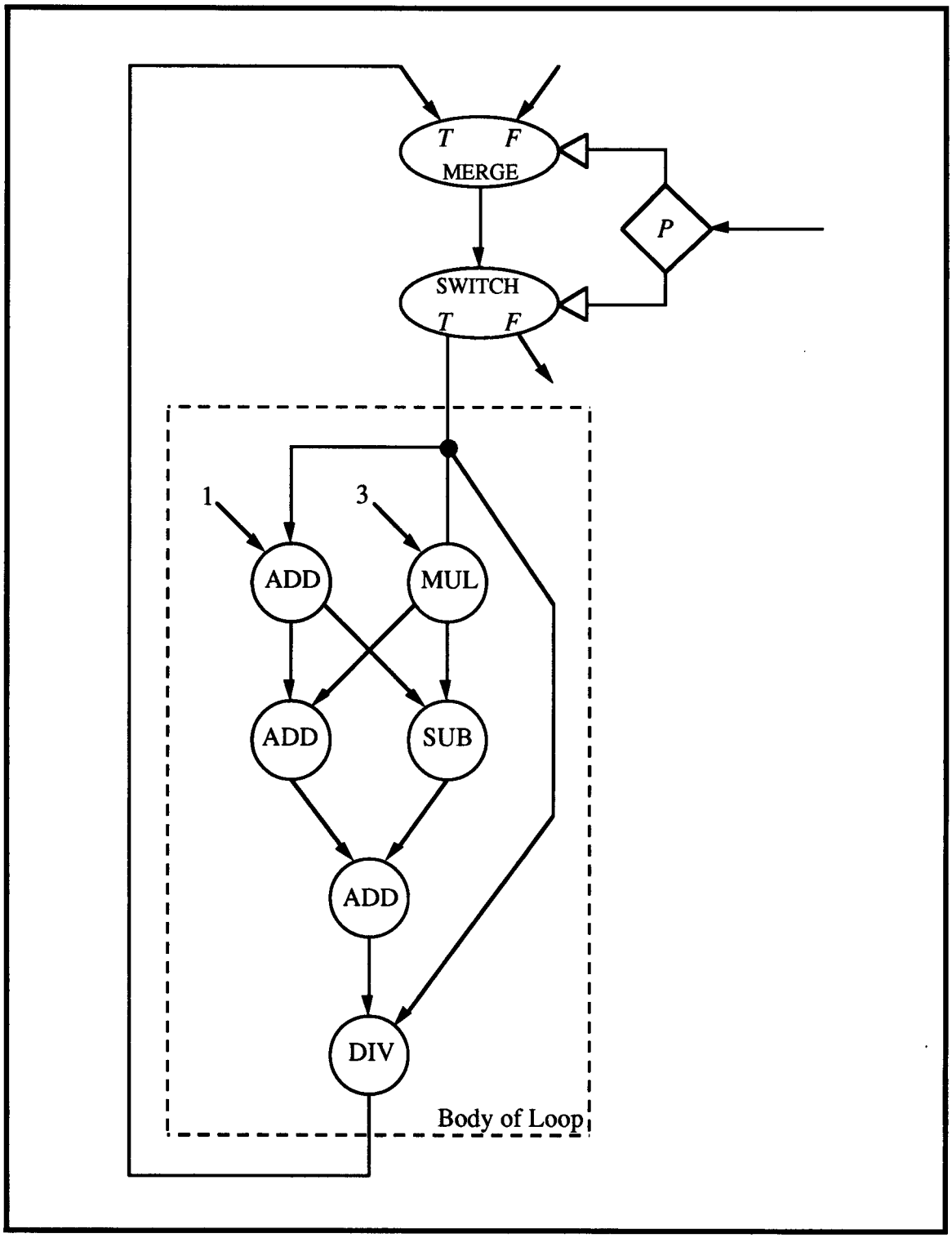


Figure 3.2 A loop structure hindered by the static firing rule.

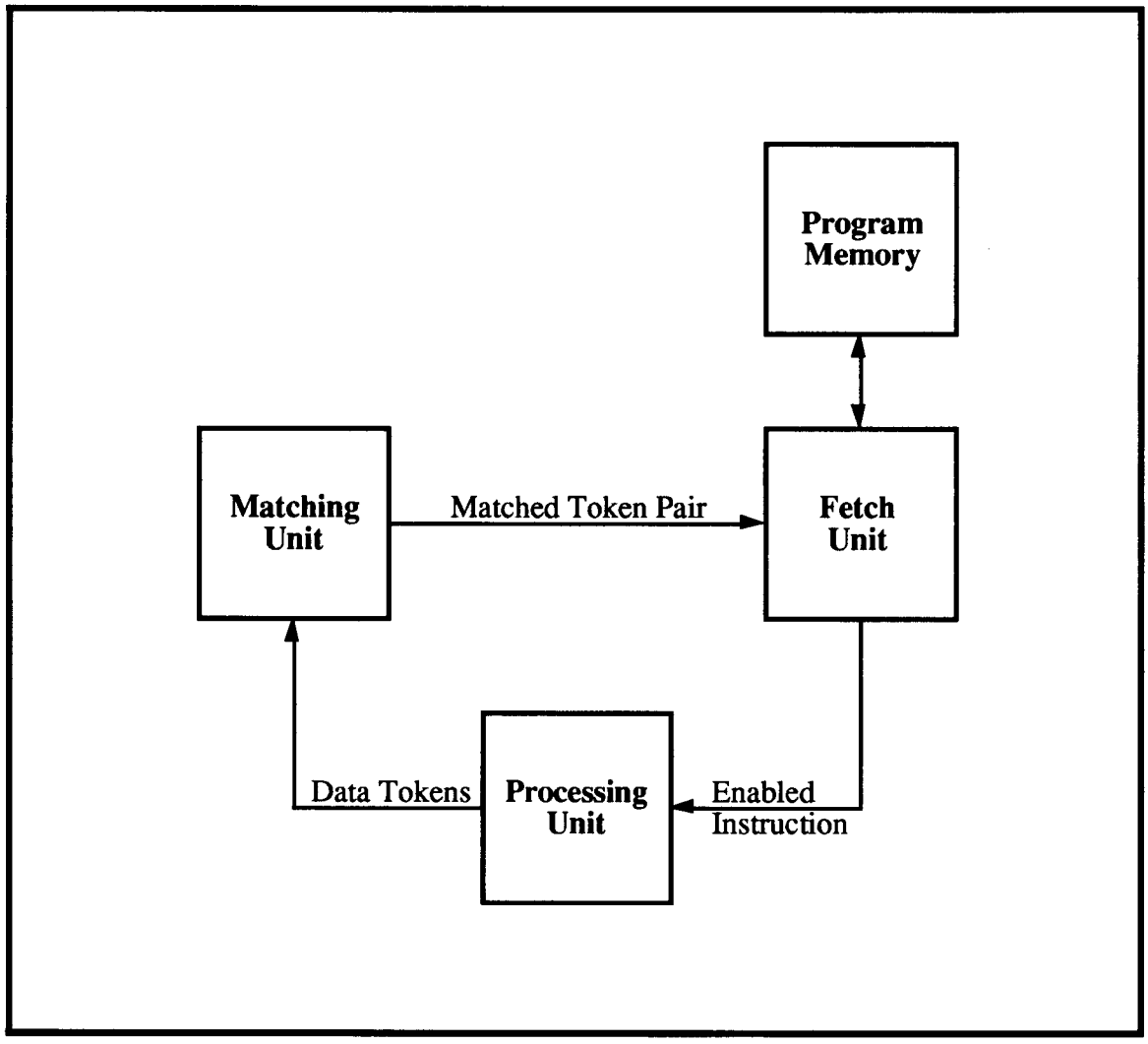


Figure 3.3 The basic organization of the dynamic dataflow model.

particular invocation of a loop code-block, while the invocation ID distinguishes between different invocations.

The dynamic dataflow model more closely models the pure dataflow model of computation by eliminating the need to maintain FIFO queues on the arcs and thus offers more parallelism than the static dataflow model. In the following sections, we will focus on three dynamic dataflow architectures that represent the spectrum of dataflow computers. First, the classical Tagged-Token Dataflow Architecture proposed by Arvind at MIT is discussed [3, 4]. Second, a more recent dataflow computer, the Monsoon machine, proposed by Papadopoulos and Culler is presented [17, 18]. Finally, the Epsilon-2 hybrid dataflow architecture proposed by Grafe and Hoch is discussed [8, 9].

3.2 Tagged-Token Dataflow Architecture

The Tagged-Token Dataflow Architecture (TTDA) is a dynamic dataflow machine developed at MIT under the direction of Arvind *et al.* [3, 4]. The TTDA is composed of a collection of PEs connected by a packet communication network. A single PE constitutes a complete dataflow computer as shown in Figure 3.4.

A token entering a PE is first routed to the Waiting-Matching unit where its tag is compared with the tags resident in the memory. If a match is found, the matched tokens are removed from the Waiting-Matching unit and forwarded to the Instruction Fetch unit. If a match does not occur, the incoming token is added to the Waiting-Matching store until its partner arrives. TTDA instructions are restricted to at most two operands, so a single match enables an instruction [3]. Monadic instructions (*i.e.*, instructions that require only one operand) bypass the Waiting-Matching unit. In the Instruction Fetch unit, the token pair uniquely identifies an instruction and any required constants to be fetched from the Program Memory. The opcode and data values are passed to the arithmetic logic unit (ALU) for processing. Concurrent with the ALU, the Compute Tag

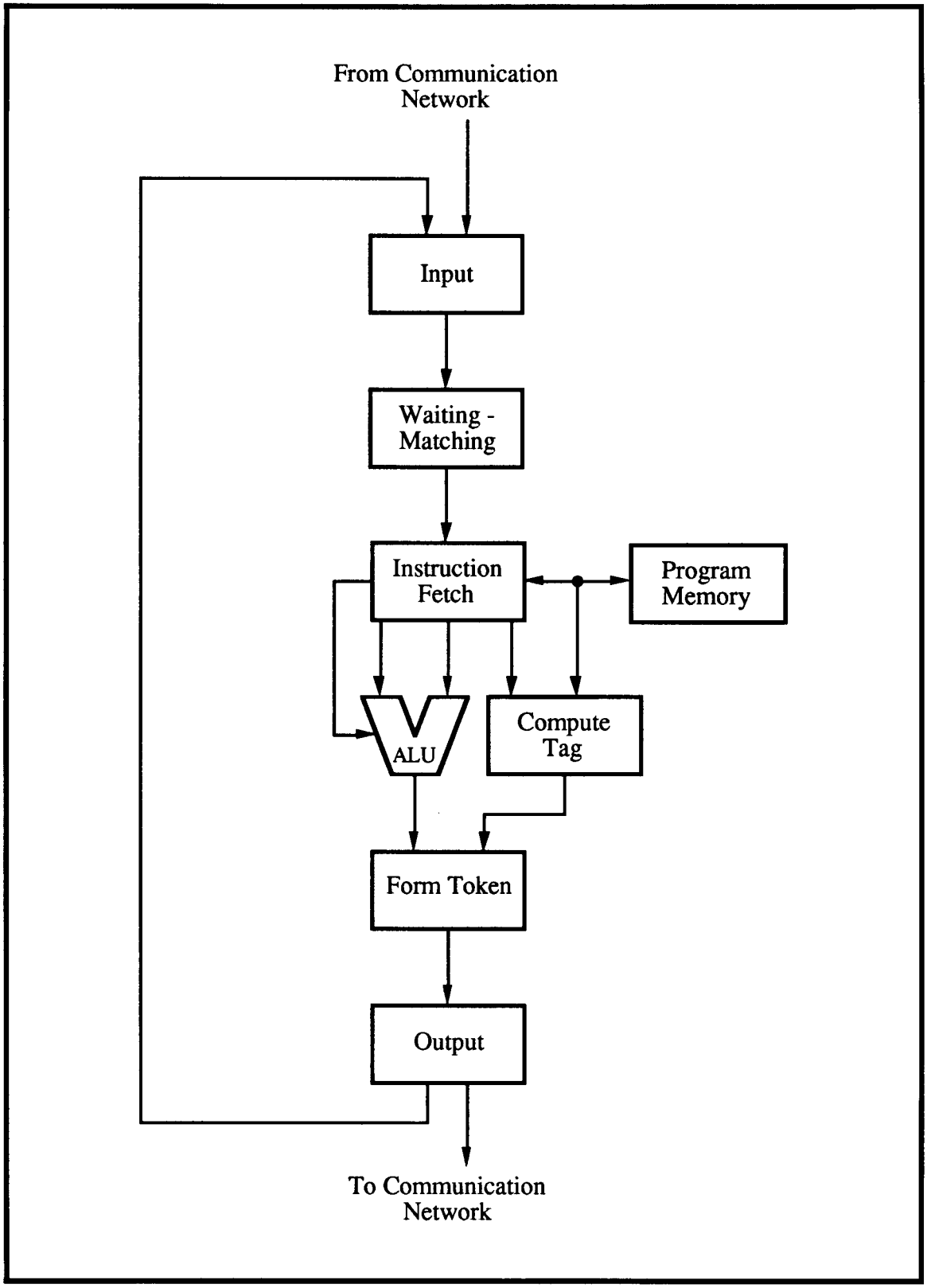


Figure 3.4 A processing element of the TTDA.

unit accesses the destination list of the instruction and the current tag to prepare a result tag. Result tags and values are concatenated to form new tokens which are then routed to the appropriate PE.

The Waiting-Matching unit is the key to the success of the TTDA. If this unit reaches its maximum storage capacity, the system will deadlock [3]. It is important to realize that the Waiting-Matching unit will implicitly allocate token store resources when the unit fails to find a match. To help alleviate the problem of deadlock, *token buffers* can be placed in a variety of places, including the Input unit and the Output unit, depending on the relative speed of each unit. The token buffer(s), combined with the size of the Waiting-Matching store, must be large enough to make the probability of overflow reasonably small.

The TTDA is the classical dynamic dataflow machine. The ideas that have emerged from the TTDA have led to several dataflow projects [3, 4, 17, 22, 27]. One of these projects is the Monsoon dataflow machine which will be discussed next.

3.3 Monsoon

Monsoon is a recent dataflow computer developed by Papadopoulos at MIT and Motorola Corporation [17, 18]. Monsoon is a highly pipelined, general purpose dataflow computer that evolved from the TTDA at MIT [17]. Like the TTDA, Monsoon is a dynamic dataflow computer; however, the Waiting-Matching unit is implemented more efficiently by utilizing the Explicit Token Store (ETS) model (*i.e.*, direct matching).

The basic idea of direct matching is to eliminate the complex and expensive task of performing an associative search to match token pairs. The direct matching scheme used in Monsoon is illustrated in Figure 3.5. In this scheme, a token store (called an *activation frame*) is dynamically allocated for all the tokens generated by a code-block, with detailed usage of locations determined at compile-time. Tokens are composed of

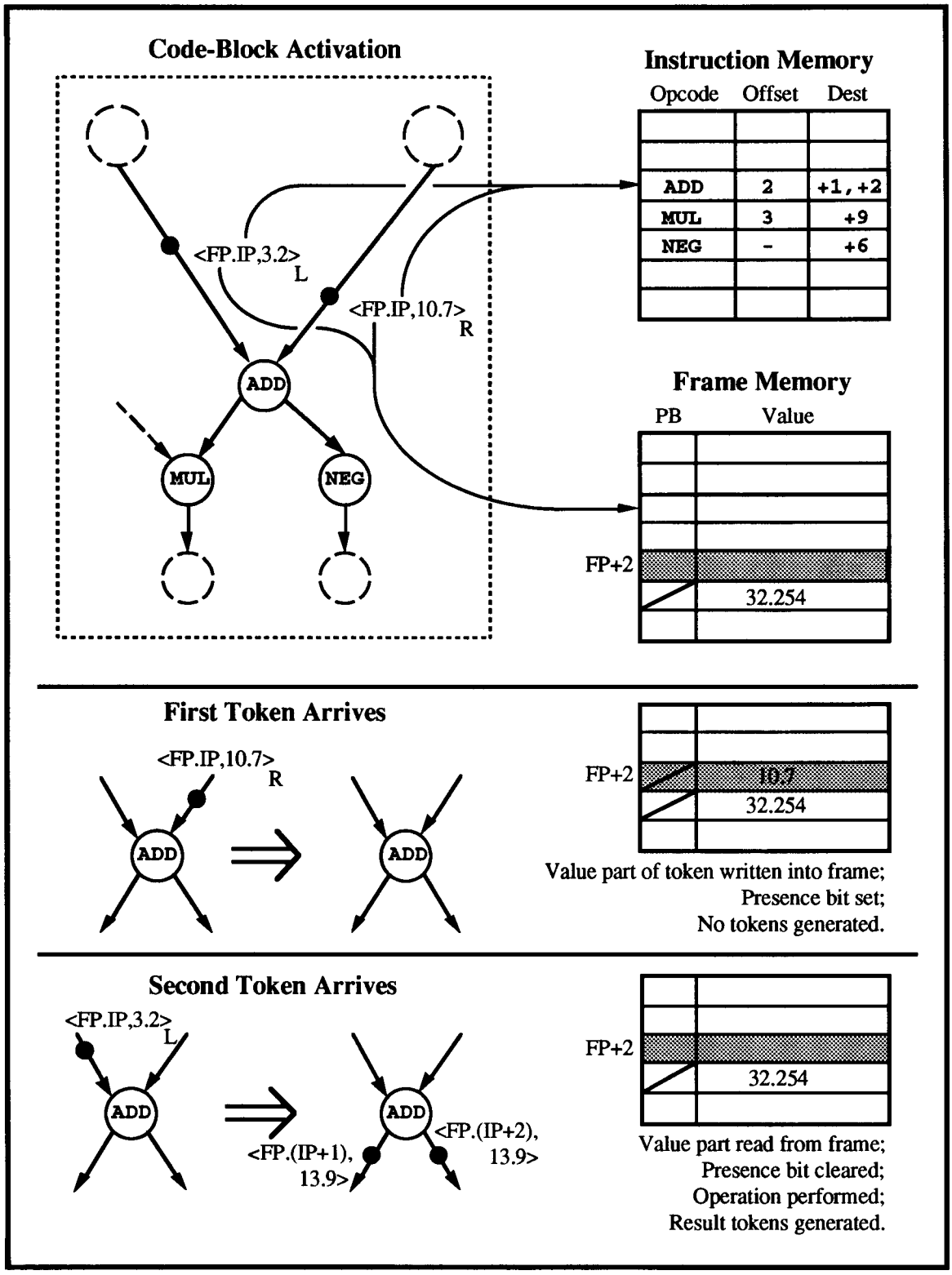


Figure 3.5 A direct matching representation of an executing dataflow program.

three components: a pointer to an activation frame (FP), a pointer to the instruction to execute (IP), and a value. The pair of pointers <FP.IP> comprise the token *continuation* or tag. The instruction fetched from location IP specifies an opcode (*e.g.*, ADD), an offset in the activation frame where the match is to take place (*e.g.*, FP + 2), and destination instructions that will receive the results of the current operation (*e.g.*, IP + 1, IP + 2). Each destination is also accompanied by an input port (left/right) that specifies the appropriate input port for a destination instruction.

Each slot in an activation frame has an associated presence bit used to determine the state of the slot. When a new token arrives, the match location is determined with a simple address calculation of the FP plus an offset. If the frame slot is empty, the value on the token is placed in the slot and the presence bit is asserted. No further processing of the instruction takes place. If the frame slot is full, the value is extracted and the presence bit is cleared. The instruction is then executed, producing one or more new tokens. After the completion of a code-block, all the slots in an activation frame are returned to their original empty state.

The Monsoon consists of a collection of pipelined PEs connected to each other and to a set of interleaved I-structure memory modules by a multistage packet switch network. Each PE has eight pipeline stages as shown in Figure 3.6. The Instruction Fetch stage fetches a local instruction from the instruction memory as specified by the IP in the incoming token. The Effective Address stage computes the address of a slot (*i.e.*, FP + offset) in the frame where the match is to take place. The presence bit associated with the frame memory location is then read, modified, and written back to the same location. Depending on the status of the presence bit, the Frame Store Operation stage ignores, reads, writes, or exchanges the value part of the specified frame store location with the value on the current token. The ALU has three stages and operates in parallel with the Compute Tag stage. In the first stage of the ALU, the value from the current token and the value extracted from the frame memory are sorted into left and right values

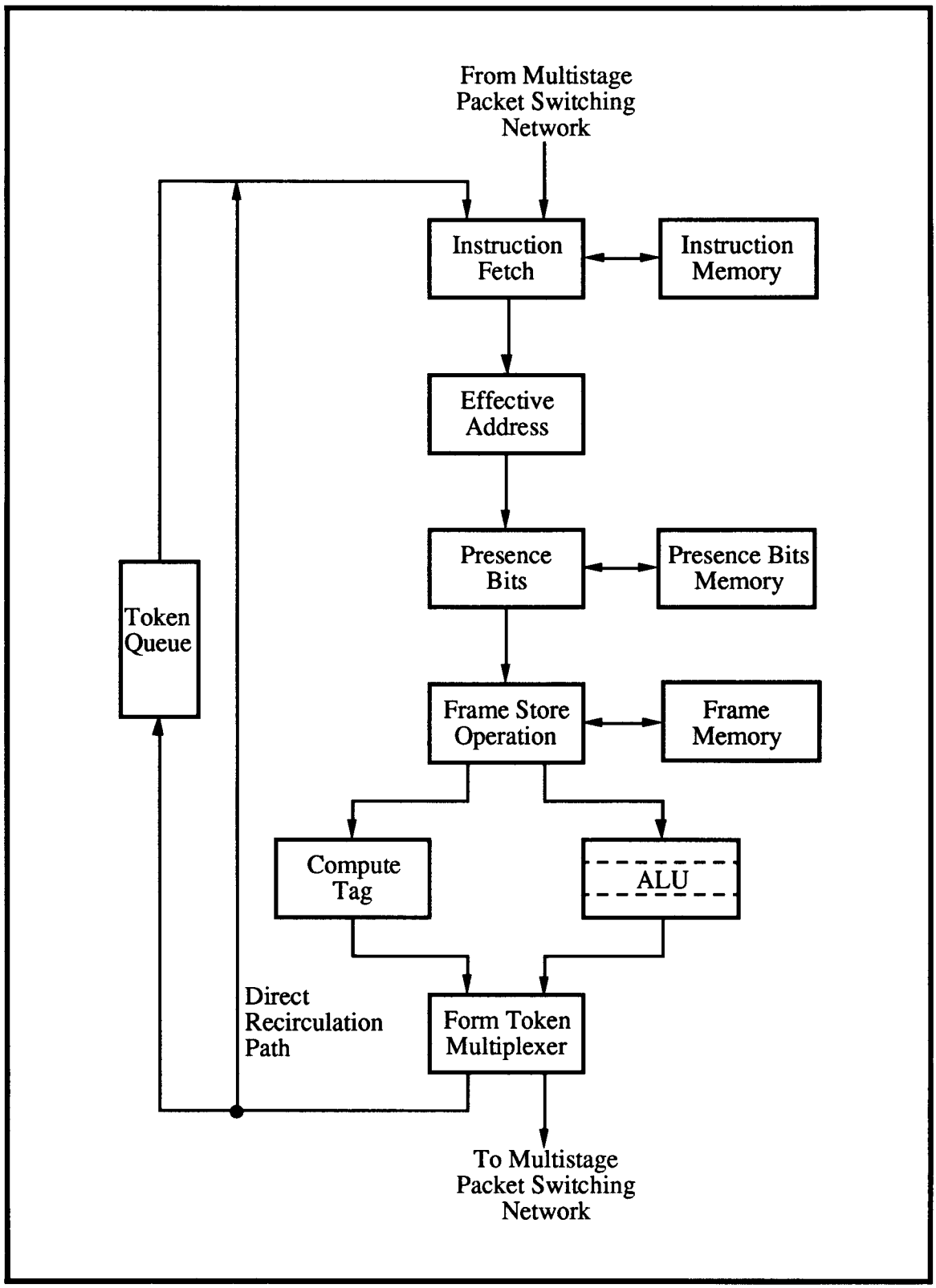


Figure 3.6 The organization of the Monsoon processing elements.

according to the input port indicator of the incoming token. In the last two stages, the operands are processed by one of the functional units. Finally, the Form Token stage creates result tokens by concatenating the computed tags with the results from the ALU.

A recent trend in dataflow computers is to combine the sequential efficiency of von Neumann computing with the fine-grain parallelism of pure dataflow computing. Monsoon accomplishes this with a technique called *multithreading* and a simple *recirculation scheduling* paradigm [18]. Multithreading incorporates control-flow sequencing in the dataflow model of computation with a simple manipulation of continuations. Recall that a computation is completely described by the continuation <FP.IP>, where IP represents a pointer to the current instruction. Therefore, the successive instruction can be described by <FP.IP + 1>. In addition to this simple computation, the hardware must also support the immediate re-insertion of tokens into the execution pipeline. Figure 3.6 shows how this is achieved by bypassing the Token Queue using the Direct Recirculation Path.

One potential problem with the recirculation method is successive tokens are not generated until the last stage of the pipeline [17]. Therefore, the execution of the next instruction in the computational thread will experience a delay that is equal to the number of stages in the pipeline. On the other hand, the recirculation method allows up to m independent threads to be interleaved in an m-stage pipeline.

3.4 Epsilon-2

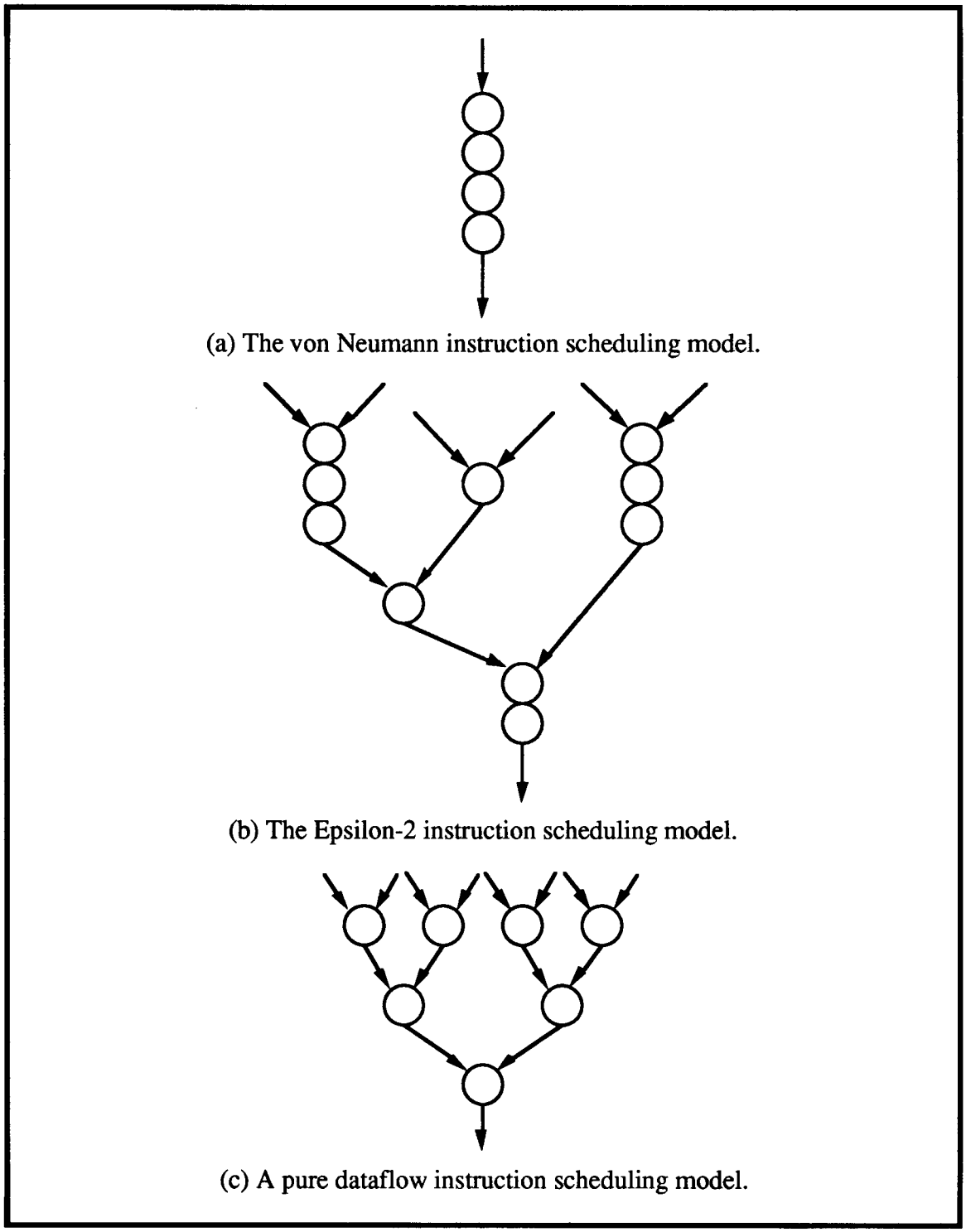
The Epsilon-2 dataflow multiprocessor was developed at Sandia National Laboratories under the direction of Grafe and Hoch [8, 9]. The design of Epsilon-2, a direct descendant of the Epsilon processor, is based on the dynamic dataflow model. Two prototypes of the Epsilon processor have been built which have demonstrated

sustained uniprocessor performance comparable to that of commercial mini-supercomputers [8, 9].

The Epsilon-2 is based on a *macro-actor* concept that allows the integration of the control-flow concept into the dataflow concept. In this scheme, instructions are grouped into larger grains where instructions within a grain can be scheduled in a control-flow fashion and the grains themselves are scheduled in a dataflow fashion. The spectrum of instruction scheduling models is illustrated in Figure 3.7. There are several advantages of a hybrid control-flow/dataflow computer. First, a simple control-flow pipeline can be utilized within a grain. Second, the instruction cycle can be reduced by utilizing a register file to store the tokens in a grain. This eliminates the overhead associated with constructing and transferring result tokens within a grain.

The architecture of the Epsilon-2 is illustrated in Figure 3.8. Each processing module is composed of a Processing Element, an I/O port, a Structure Memory, and a 4x4 Crossbar Switch that connects the various modules by a Global Interconnect. Tokens are fixed length and composed of a *target* and *data* section. The target section consists of a frame pointer and instruction pointer pair (*i.e.*, <FP.IP>) that represent the tag while the data section contains the token value. Both the target and data section contain type fields that identify the type of information contained in the remainder of the token section.

Tokens arriving from the local 4x4 switch are buffered in the Token Queue. A Token is read from the Token Queue and the IP is used to access the Instruction Memory. A match offset from the current instruction identifies a synchronization location in the Match Memory. The selected location in the Match Memory is read and compared with the match count encoded in the current instruction. If the match count equals the value from the Match Memory, the instruction fires and the match location is initialized to zero. If a match does not occur, the match value is incremented by one and the operand is stored in the Frame Memory as specified by the frame pointer and an opcode offset.



(a) The von Neumann instruction scheduling model.

(b) The Epsilon-2 instruction scheduling model.

(c) A pure dataflow instruction scheduling model.

Figure 3.7 The spectrum of instruction scheduling models.

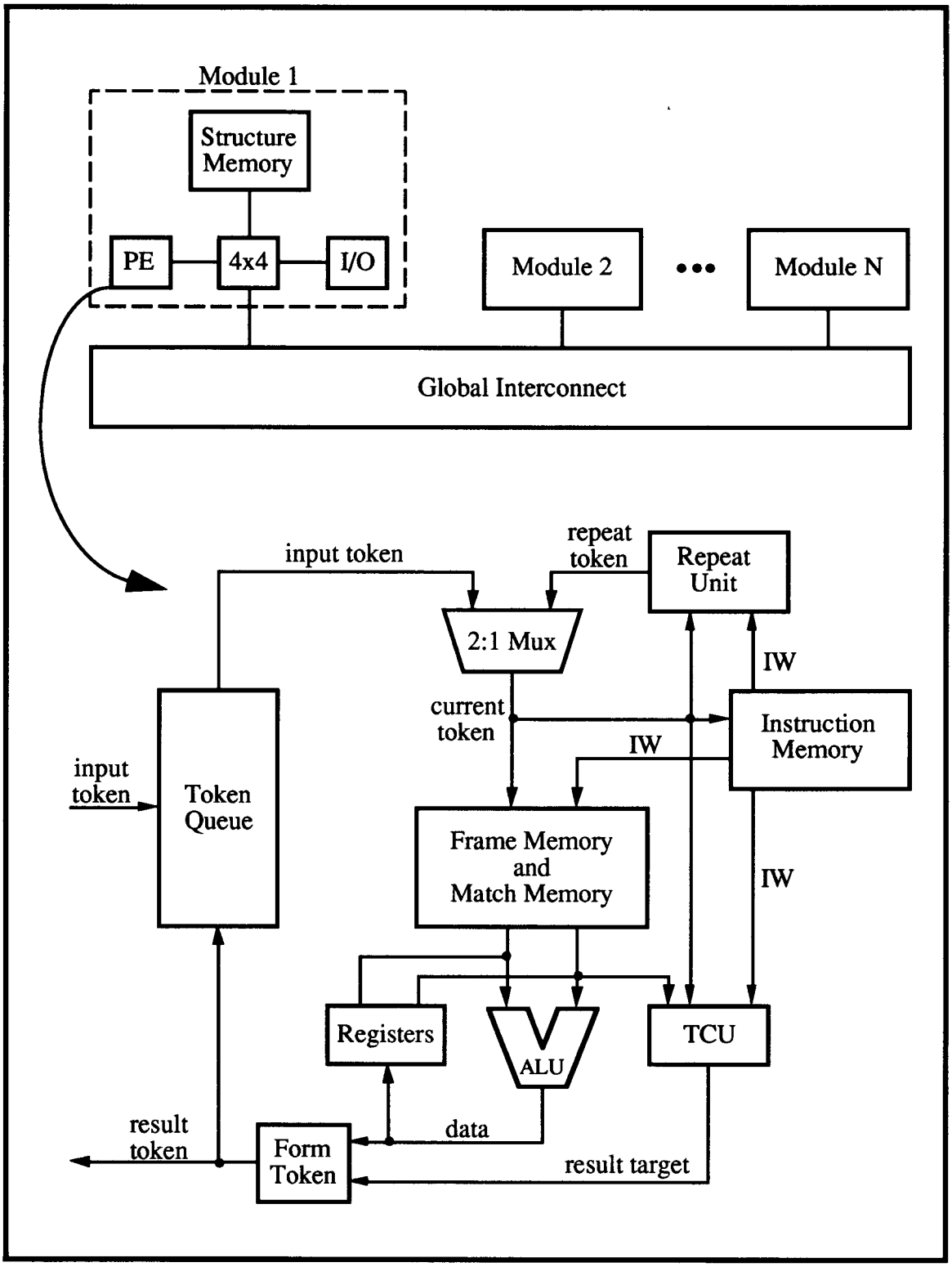


Figure 3.8 The architecture of the Epsilon-2 multiprocessor.

Thus, the value in the match location represents a count of the number of tokens that have arrived at the synchronization point.

The ALU operates on the operands of enabled instructions as specified by the opcode. Results generated from operations are written into registers that can be referenced by succeeding instructions within a grain. The Target Calculation Unit (TCU) is responsible for generating the target section of output tokens. Targets local to the current activation retain the current FP and a new IP is generated by adding a target offset to the current IP.

Sequential scheduling is implemented with the Repeat Unit [8, 9]. The Repeat Unit generates repeat tokens that efficiently reduces data fanout. A repeat token is generated by adding the repeat offset in the instruction word to the current instruction pointer to generate a new instruction pointer. The current token's frame pointer and data section are not modified when producing a repeat token. A 2:1 multiplexer controls whether the next token is a repeat token or a new token from the Token Queue. In effect, repeat offsets in the instruction word are used to build a linked thread of instructions that utilize registers to buffer the results between operations.

4. Hypercubes

Now that several dataflow architectures have been discussed, we turn our discussion to the topological organization of dataflow systems. A dataflow system can be classified as either *centralized* or *distributed.* The classification is based on the organization of instruction memory.

In a centralized organization, memory is shared by all the processors. A centralized memory organization requires expensive hardware and/or software protocols for arbitration among processors. Furthermore, since memory references tend to be a large fraction of a programs execution, access to a centralized memory must be kept small [10, 11]. These two properties alone will ultimately limit the number of processors that can economically be connected.

A distributed memory system has a separate memory for each processor and information is exchanged by messages between processors. If each processor has most of the data it needs, the number of messages passed between processors will remain relatively low, and the number of processors in the system can be scaled upward. Any message that must be sent to a nonadjacent processor must be passed through an intermediate processor. Due to the limiting nature of centralized organizations, recent trends have adopted the more scalable, distributed organization for concurrent computer designs [10, 14, 20, 21].

As can be seen, a topology for a distributed architecture should have many features. These features should include, among others, a high fault-tolerance, regularity of structure, scalability, simple deadlock-free routing algorithms, high I/O bandwidth, and a small diameter. Additionally, a topology should be able to embed important topologies such as rings, meshes, and trees so applications written for these specific topologies can be ported to a new system with ease. One topological organization that satisfies all these requirements is the *hypercube.* We have chosen the hypercube topology

as a representative processor organization for our research due to their powerful interconnection features and wide use [19]. Although the culmination of our research (Chapter 5) is applicative to any processor organization, we demonstrate our results on a hypercube. Thus, this chapter is devoted to the discussion of important properties of a hypercube topology.

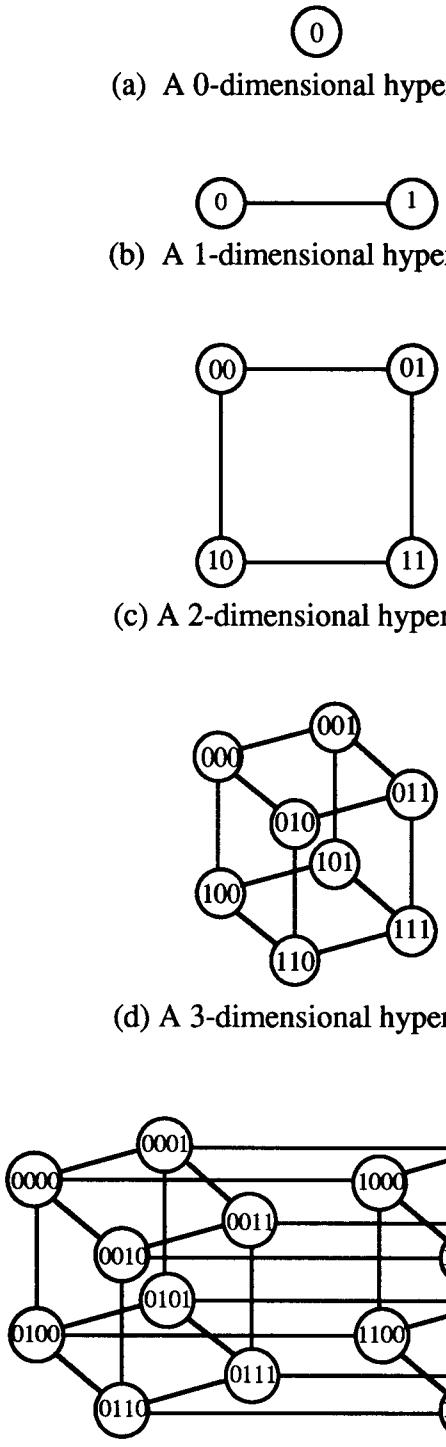
4.1 Fundamental Hypercube Properties

In the following discussion, the hypercube is regarded as a graph and the terms *vertices* or *nodes* are synonymous for the processing elements they represent. A hypercube of dimension k is a multicomputer system with 2^k processors. Nodes are adjacent and connected by an *edge* if their addresses differ in only one bit position. Therefore, a node in a k-dimensional hypercube has k adjacent nodes. The hypercube is formally defined in *Definition 4.1* below.

***Definition 4.1*:** A k-dimensional hypercube is an undirected graph of 2^k nodes having addresses between 0 and $2^k - 1$ such that there is an edge between any two nodes if and only if the binary representations of their addresses differ by one and only one bit.

***Property 4.1*:** A k-dimensional hypercube can be constructed recursively using two $(k-1)$-dimensional hypercubes [21].

Consider two identical $(k-1)$-dimensional hypercubes whose nodes have unique addresses from 0 to $2^{(k-1)} - 1$. A k-dimensional hypercube is then formed by joining each node in the first $(k-1)$-dimensional hypercube to the corresponding node in the second $(k-1)$-dimensional hypercube having the same address. It then suffices to renumber the nodes of the first $(k-1)$-dimensional hypercube as $0 \wedge a_i$ and those of the second by $1 \wedge a_i$ where a_i represents the binary address of a node and $\wedge$ represents the concatenation of binary bits. Figure 4.1 illustrates a 0-dimensional, a 1-dimensional, a 2-dimensional, a 3-dimensional, and a 4-dimensional hypercube.



(a) A 0-dimensional hypercube.

(b) A 1-dimensional hypercube.

(c) A 2-dimensional hypercube.

(d) A 3-dimensional hypercube.

(e) A 4-dimensional hypercube.

Figure 4.1 Hypercubes of dimension k=0, 1, 2, 3, 4.

Conversely, consider a k-dimensional hypercube that has been separated into two subgraphs such that one subgraph contains all the nodes whose leading bit is 0 and the other subgraph contains all the nodes whose leading bit is 1. If the edges connecting the two subgraphs and the leading bits are removed, then two distinct $(k-1)$-dimensional hypercubes are formed. This procedure of forming two $(k-1)$-dimensional hypercubes from a single k-dimensional hypercube is referred to as *tearing* [21]. More generally, tearing may be implemented by separating a k-dimensional hypercube into two subgraphs such that one subgraph contains all the nodes whose i^{th} bit is 0 and the other subgraph contains all the nodes whose i^{th} bit is 1. This forms two $(k-1)$-dimensional hypercubes and is referred to as tearing along the i^{th} direction. The notion of tearing leads to the following properties.

***Property 4.2*:** A k-dimensional hypercube can be torn in k different directions. It suffices to state that since nodes in a k-dimensional hypercube have a k-bit address, then there are also k directions in which to tear the hypercube.

Property 4.3: There are $k!2^k$ different ways in which the 2^k nodes can be numbered in a k-dimensional hypercube to conform with *Definition 4.1*.

Proof by induction: The result is trivial for $k=0$, therefore assume *Property 4.2* is true for a $(k-1)$-dimensional hypercube. Consider numbering the nodes in a k-dimensional hypercube. To perform this numbering, tear the nodes into two $(k-1)$-dimensional hypercubes. By *Property 4.2* there are k different ways to tear the hypercube. Next, assign addresses to the nodes of the first $(k-1)$-dimensional hypercubes (of which there are $(k-1)!2^{(k-1)}$ different ways from our proposition). After having assigned addresses to one of the $(k-1)$-dimensional hypercubes, prefix their addresses with a leading zero. Assign addresses to nodes in the second $(k-1)$-dimensional hypercube such that there is a one-to-one correspondence with nodes in the first $(k-1)$-dimensional hypercube and prefix the addresses with a leading one. A

second ordering can be obtained by reversing the most significant bit in the address. Thus, there are a total of

$$\begin{aligned} & k\left[(k-1)!2^{k-1}+(k-1)!2^{k-1}\right] \\ &= 2k(k-1)!2^{k-1} \\ &= k(k-1)!2^{k} \\ &= k!2^{k} \end{aligned}$$

different numberings of the nodes of the k-dimensional hypercube.

There is always a path between two nodes N_i and N_j of a k-dimensional hypercube; however, one may wish to know the total number of edges and the minimum distance between the two nodes in a k-dimensional hypercube. This leads to *Property 4.4* and *Property 4.5*.

Property 4.4: The total number of edges in a k-dimensional hypercube is $k2^{k-1}$. If the number of nodes in a k-dimensional hypercube is denoted by n, where $n = 2^k$, then *Property 4.4* can be rephrased as the total number of edges in a k-dimensional hypercube with n nodes is $(n/2)\log_2 n$.

Proof by induction: The result is trivial for $k = 0$, therefore assume *Property 4.4* is true for a $(k-1)$-dimensional hypercube. Therefore, a $(k-1)$-dimensional hypercube will have $(k-1)2^{k-2}$ edges. From Property 4.1 a k-dimensional hypercube can be constructed from two $(k-1)$-dimensional hypercubes by placing an edge from each node in the first $(k-1)$-dimensional hypercube to its corresponding node in the second $(k-1)$-dimensional hypercube. This requires the addition of 2^{k-1} edges for a total of

$$\begin{aligned} & 2(k-1)2^{k-2}+2^{k-1} \\ &= 2(k-1)2^{(k-1)-1}+2^{k-1} \\ &= 2^{k-1}[(k-1)+1] \\ &= k2^{k-1} \end{aligned}$$

edges in a k-dimensional hypercube.

Property 4.5: The minimum distance between two nodes N_i and N_j is equal to the number of bits that differ between the address of node N_i and node N_j, *i.e.*, the Hamming distance $H(N_i,N_j)$ [21].

Property 4.5 implies that the minimum distance between the two most distant nodes is k. That is, a hypercube with dimension $k = \log_2 n$ has a minimum path distance of k between the two furthest nodes, N_i and N_j, where $N_i = \overline{N_j}$, such that the distance is a logarithmic function of the number of nodes, n, in the hypercube (*i.e.*, the hypercube exhibits a logarithmic diameter).

4.2 Hypercube Message Passing

One important question that needs to be addressed is whether there are different paths between nodes N_i and N_j in a hypercube. The availability of such paths would help speedup the inherent communication latency in a distributed multicomputer system. Additionally, the availability of multiple paths would imply that the hypercube has fault tolerance with respect to the communication paths. For fault tolerance, the multiple paths from node N_i to node N_j must truly be *parallel paths,* and thus, the two paths must not have any nodes in common except the nodes N_i and N_j. Therefore, in determining a path from N_i to N_j one may look at the source address and change the most significant bit (MSB) that differs from the destination address of N_j. Since only one bit is being changed at a time, the new address formed is guaranteed to be adjacent to the present source address by *Definition 4.1*. Therefore, the minimum distance traveled from N_i to N_j is $H(N_i,N_j)$. This is illustrated in Figure 4.2 for a 3-dimensional hypercube with $N_i = 000$ and $N_j = 111$.

However, we need not always change the MSB that differs between the source address and the destination address. In fact, we may change any of the $H(N_i,N_j)$ bits that differ between the addresses of N_i and N_j. This leads to *Property 4.6*.

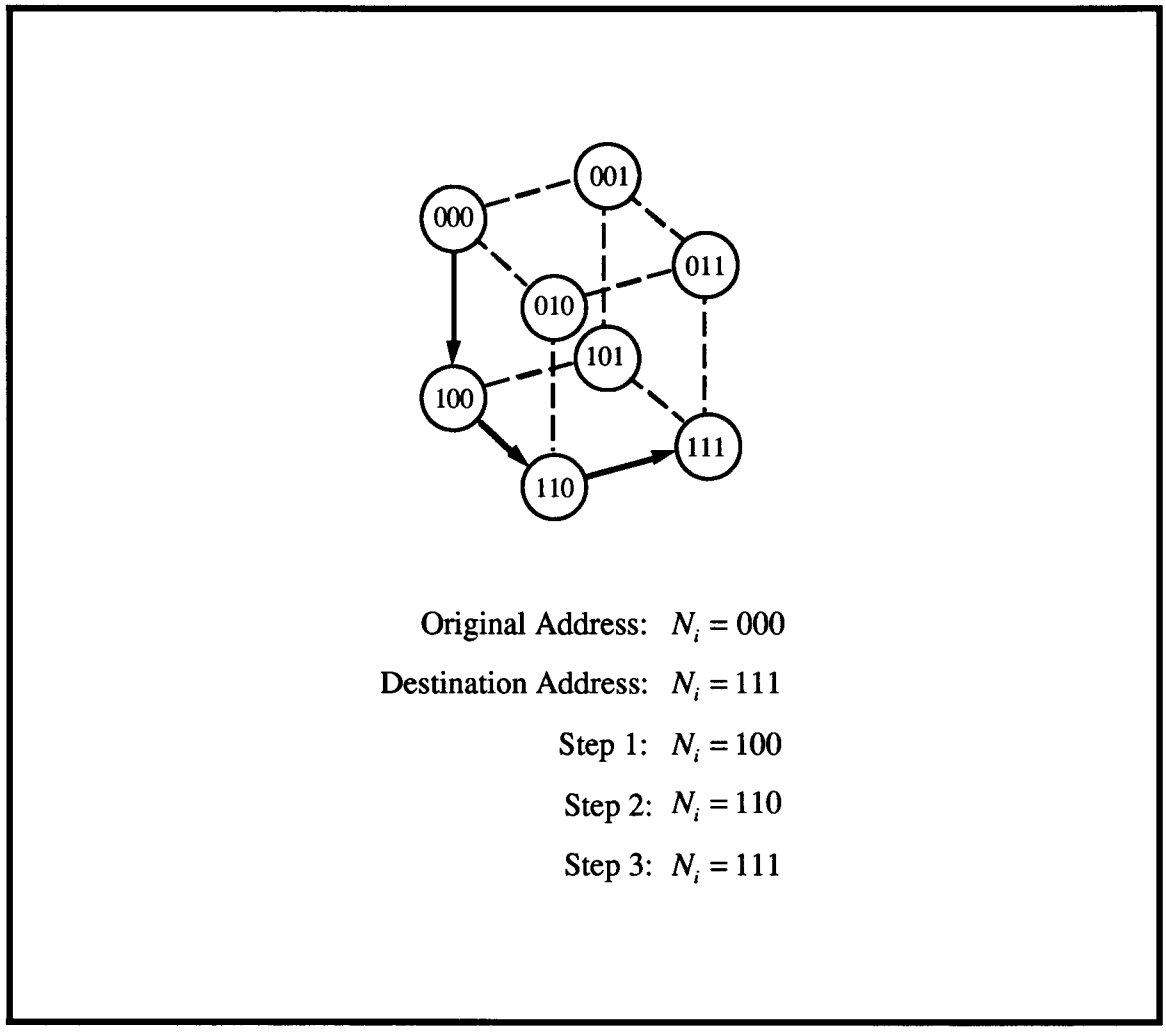


Figure 4.2 A possible message routing algorithm for a 3-dimensional hypercube.

Property 4.6: If N_i and N_j are two nodes in a k-dimensional hypercube, then there are $H(N_i,N_j)$ parallel paths of length $H(N_i,N_j)$ between the nodes N_i and N_j.

Property 4.6 states that the number of parallel paths between two nodes N_i and N_j is equal to the number of bits that differ between the two addresses. However, this method only identifies minimum length parallel paths. This result can be improved by relaxing the restriction that the path length must be $H(N_i,N_j)$. In doing so, as many as k parallel paths can be found in a k-dimensional hypercube, even for the case when $H(N_i,N_j)<k$. This leads to *Property 4.7* that exploits the full communication bandwidth of a k-dimensional hypercube multicomputer system.

Property 4.7: If N_i and N_j are two nodes in a k-dimensional hypercube, then there are k parallel paths of length at most $H(N_i,N_j)+2$ between the nodes N_i and N_j [21].

A simple routing algorithm that uses *Property 4.6* can execute on every node in a k-dimensional hypercube. This algorithm numbers an edge in a hypercube relative to the two nodes that it connects. Specifically, an edge has *edge number i* if it connects two nodes whose addresses differ in the i^{th} bit (where the least significant bit is designated as bit 0). For example, if an edge connects two nodes whose addresses are 001 and 011, then the edge connecting these nodes is referred to as edge number 1 since the addresses differ in bit 1. Therefore, each node has edges labeled from edge number 0 to edge number $k-1$. This is illustrated for a 2-dimensional hypercube in Figure 4.3. The corresponding routing algorithm that is executed on each node of the hypercube is described in *Algorithm 4.1*.

When using a distributed multicomputer system, like a hypercube, one processor may need to broadcast a single message to all other processors. If the broadcast scheme can send the message to every processor only once, then the message is said to have been transmitted by a *nonredundant broadcast algorithm*. *Algorithm 4.2* describes such a nonredundant broadcast algorithm for a k-dimensional hypercube that allows the

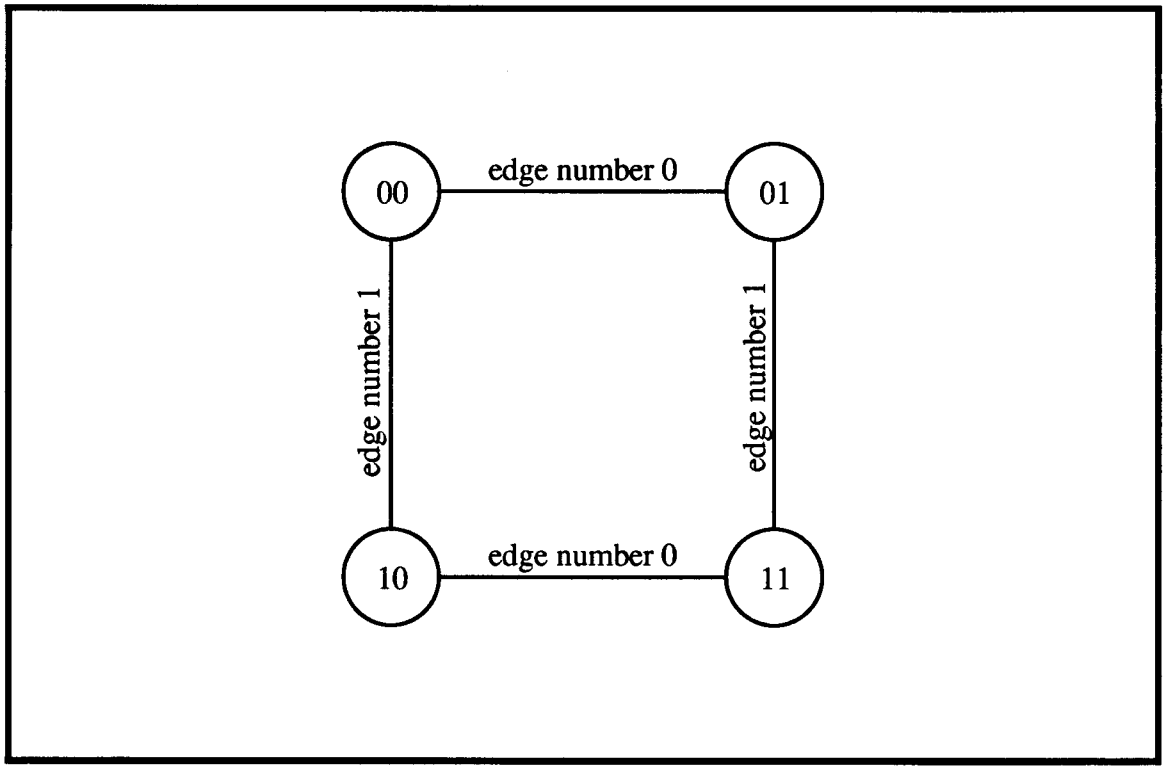


Figure 4.3 Edge numbering for a 2-dimensional hypercube.

Algorithm 4.1 Send/forward a message from N_i to N_j in a k-dimensional hypercube.

```
IF (SOURCE == DESTINATION)
     message has arrived at destination;
ELSE
     RESULT = SOURCE ⊕ DESTINATION;
     EDGE_NUMBER = K - 1;
     WHILE (the MSB of RESULT ≠ 1) {
          RESULT = leftshift(RESULT);
          EDGE_NUMBER = EDGE_NUMBER-1;
     }
     send_message_on_edge(EDGE_NUMBER);
```

Algorithm 4.2 Broadcast a message from any node in a k-dimensional hypercube.

```
Start the algorithm at any node with WEIGHT set to K.

FOR (each edge I from the current node with I < WEIGHT) DO
     send_message_on_edge(EDGE_NUMBER with weight I);
```

communication bandwidth of a multicomputer system to be used to its fullest. This algorithm sends the message in k steps (*i.e.*, $\log_2 n$) and works by sending a weight along with each message that indicates how the message should be broadcasted from the receiving node.

4.3 Embedding Properties of the Hypercube

One of the most important features of hypercubes is their capability of embedding other important topologies [10, 21, 19]. For example, the hypercube can embed trees, rings, and meshes of all dimensions [19]. Additionally, the communication structures used in the Fast Fourier Transform and the bitonic sort algorithm can be embedded in a hypercube [10]. In general, a connected graph G can be embedded into a hypercube with dimension k if and only if it is possible to label the edges of G with the integers $\{1, 2, \cdots, k\}$ such that

- Edges incident with a common node have different labels;
- In every path of G at least one label appears an odd number of times; and
- In every cycle of G no label appears an odd number of times [19].

One of the most important embeddings is that of meshes into hypercubes. In performing such an embedding, binary sequences called *Gray codes* are used. A Gray code is a binary sequence such that two successive bit sequences differ by one and only one bit. An example of a Gray code sequence is 000, 001, 011, 010, 110, 111, 101, 100. In performing the mapping, each dimension of the mesh is assigned an encoded Gray code string such that traversing the mesh in that dimension yields a complete Gray code cycle. A node in the mesh is then assigned to a node in the hypercube with an address specified by the concatenation of its binary coordinates. Figure 4.4 illustrates the embedding of an 8 x 4 grid into a 32-node hypercube. From Figure 4.4, nodes A, B, and

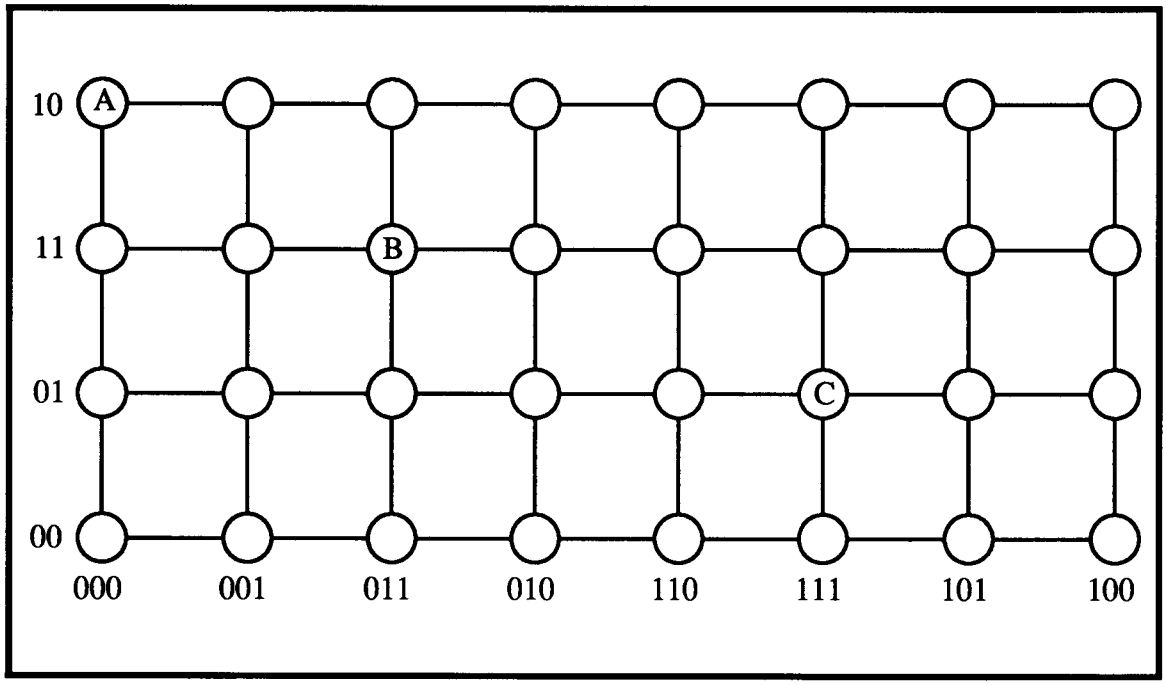


Figure 4.4 Embedding a 2-dimensional mesh into a 32-node hypercube.

C map to nodes 00010, 01111, and 11101, respectively in a 32-node (4-dimensional) hypercube.

Thus, from general topological arguments it can be seen that the hypercube has high fault tolerance, logarithmic diameter, and exhibits regularity of structure. Additionally, the hypercube balances node connectivity with algorithm embeddability and programming ease. This balance makes the hypercube a suitable processor topology in a wide variety of multiprocessor systems, including dataflow systems.

5. The Balanced Layered Allocation Scheme

Although the dataflow model of computation offers many attractive properties for parallel processing, there are still several problems that need to be addressed [2, 3, 12]. One of these concerns is the issue of program allocation in a dataflow environment. The proper allocation of tasks to processing elements (PEs) has an immediate effect on the overall performance of the system. Maximizing the embedded parallelism of an allocated program while minimizing interprocessor communication is a main objective of any allocation scheme [2, 15]. This chapter proposes a method called the *Balanced Layered Allocation Scheme* (BLAS) that utilizes heuristic rules to find a balance between computation and communication costs in the allocation of programs to processors. Simulation studies indicate that the BLAS is effective in reducing communication overhead as well as overall execution times of dataflow programs.

5.1 The Allocation Problem

Despite the architectural differences in dataflow multiprocessor systems, they all share a common goal in the allocation of programs: maximizing the inherent concurrency of a program while minimizing the contention for processing resources. Yet, if the execution times of instructions vary, or if the number of processors is larger than two, the problem of optimally allocating a program to processors is NP-complete [15, 19]. The allocation problem is further complicated by the fact that communication costs exist between instructions assigned to different processors. Therefore, heuristics are generally used to solve the allocation problem [15, 23, 24]. One such approach is the Vertically Layered (VL) allocation scheme proposed by Lee *et al.* for Multistage Interconnection Network (MIN) based dataflow systems [15].

The VL allocation scheme is a compile-time method based on two underlying philosophies: (1) assign concurrently executable instructions to separate PEs and (2) assign data dependent instructions to the same PE [15]. Thus, the goal of these two philosophies is to minimize execution times and communication costs. The VL allocation scheme is implemented with a separation phase and an optimization phase. During the separation phase, the dataflow graph representation of a program is separated into vertical *layers* that have a one-to-one correspondence with processors in the dataflow computer. The vertical layers are determined by first identifying the *critical path* (CP) of the program graph. The CP identifies the most time consuming path, from root node to exit node, of a given dataflow graph. Ideally, the CP dictates the maximum total execution time of a program, and as a result, the CP is given the highest priority for processor assignment (*i.e.*, the CP is assigned to the most central processor). The remaining vertical layers are found by recursively determining the *Longest Directed Path* (LDP) from nodes that have already been allocated by using the same approach as when the CP was found. These LDPs are then assigned to processors based on the density of allocated nodes (*i.e.*, load balancing). Note that the separation phase is carried out without considering communication costs. Once the relative assignment of the nodes to processors is known, an optimization phase attempts to minimize inter-PE communication behaviors.

Performance studies indicate that the VL allocation scheme is very effective in reducing the communication overhead; however, one shortcoming of the VL allocation scheme is its poor performance when the number of available PEs is much less than the maximum parallelism of the dataflow graph [15]. A reason for the poor performance in such a case is the VL allocation scheme neglects the effects of communication delays when LDPs are assigned to layers in the separation phase. Furthermore, the VL allocation scheme assumes constant interprocessor communication delays among all pairs of PEs limiting its application to a small class of interconnection networks. In light of the

above discussion, we extend the work done by Lee *et al.* and propose an alternative method for allocating programs to dataflow computers.

5.2 The Proposed Allocation Scheme

Our proposed scheme assumes that the underlying architecture incorporates the macro-actor concept (see section 3.4) since performance is not limited by the number of independent computational threads in an application program. Although the proposed algorithm is applicable to any processor organization, we demonstrate our algorithm on a hypercube processor organization. We have selected the hypercube topology to discuss our algorithm due to its wide use and powerful interconnection features as demonstrated in Chapter 4.

The proposed allocation scheme is based on three general objectives: (1) assign concurrently executable nodes to separate processing elements, (2) assign data dependent nodes to the same processing element, and (3) assign nodes to PEs that lead to an earlier completion time of the program. Objective (1) encourages the exploitation of parallelism on dataflow processors, while objective (2) minimizes communication costs. However, these are two conflicting objectives; therefore, objective (3) is used to provide a compromise by considering the effects of node execution times and interprocessor communication costs. In doing so, local communication and parallelism are encouraged while poor allocations that yield large communication costs and long program execution times are discouraged.

The proposed allocation scheme utilizes Critical Path (CP) and Longest Directed Path (LDP) heuristics to initially separate the dataflow graph representation of a program into modules. Ideally, the CP dictates the maximum execution time of a program and thus the CP is given the highest priority for processor assignment. All other program

modules in G are found recursively by determining the LDP emanating from the nodes that have already been assigned to processing elements.

The heart of the BLAS is directed dataflow graphs. Consider an arbitrary directed dataflow graph $G \equiv G(N,A)$ where N represents the set of instructions and A represents the dependencies between these instructions. Thus, a directed path between node n_i and node n_j implies that n_i precedes n_j (*i.e.*, $n_i \rightarrow n_j$). Also associated with each node n_i is an execution time t_i. It is assumed that these execution times are known in advance at compile-time. An example of a directed dataflow graph with the node execution times, t_i, is shown in Figure 5.1. Additionally, the proposed algorithm requires a dataflow graph to have a root node and a single exit node. If a root node or an exit node does not exist, a *dummy* root node or exit node is introduced with negligible execution times and negligible communication delays between PEs.

The proposed heuristic algorithm is based on two parameters: execution times and communication costs. It is assumed that the execution times, t_i, are known in advance at compile-time. In addition, a communication delay, C_{ij}, exists between adjacent processing elements PE_i and PE_j. These communication costs are a function of the target architecture. If α denotes the time required to initiate a packet, and if β denotes the time required to send the packet, then the total time required to send a single token to an adjacent processor is $\alpha+\beta$. Thus, communication between adjacent processors, PE_i and PE_j, requires a constant time $C_{ij} = \alpha+\beta$. It is assumed that the communication delays among all adjacent processors, C_{ij}, in a hypercube based system are identical. It is also assumed that communication between non-adjacent processors in a hypercube requires an integral multiple of C_{ij} (*i.e.*, store-and-forward routing).

A dataflow graph is partitioned into program modules based on execution times and communication delays. Program modules are allocated to processors only after iteratively considering these parameters on all PEs. Each program module, containing a set of serially connected nodes, is rearranged into separate layers such that each layer has

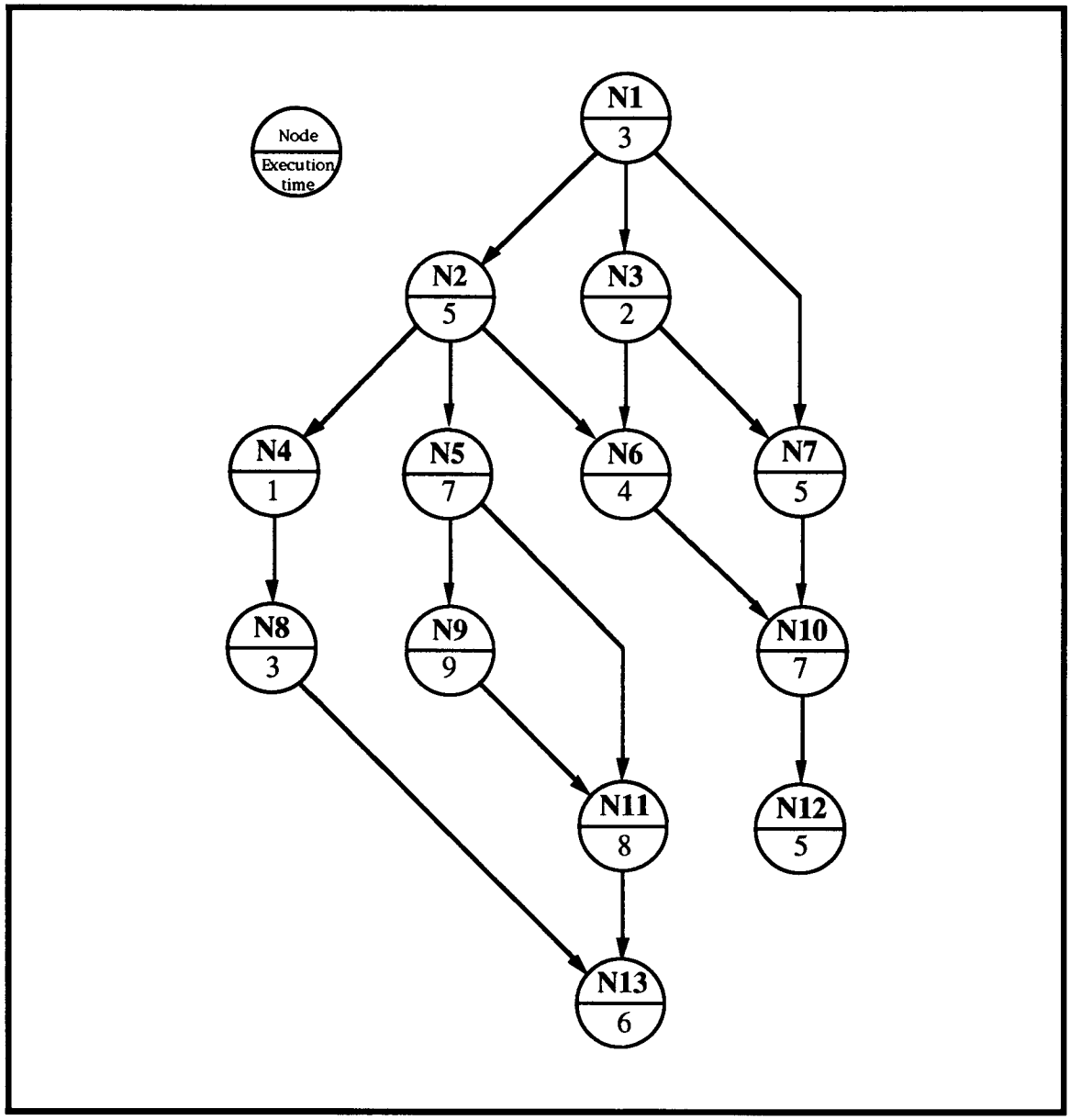


Figure 5.1 A directed dataflow graph.

a one-to-one correspondence with a processor in the hypercube. Thus, a k-dimensional hypercube with 2^k different processors has 2^k unique layers numbered 0, 1, ..., $2^k - 1$. However, before the critical path of a directed graph G can be determined, special provisions for conditional nodes, loops, and recursive functions must be considered [15].

Conditional nodes in a dataflow graph direct tokens to only one output arc depending on a boolean input token. Since the boolean input tokens are only known at run-time, the critical path through a conditional node cannot be determined at compile-time. An example of a dataflow graph with a conditional node is shown in Figure 5.2. The expected execution time of nodes in a dataflow graph with conditional nodes is determined by assigning a probability p_{ij} to the arc $a(n_i, n_j)$ for all $a_{ij} \in A$. If node n_i is not a conditional node, then $p_{ij} = 1$. Otherwise, every conditional node must satisfy

$$\sum_{j \in S_i} p_{ij} = 1,$$

where S_i is the set of index values for all the immediate successor nodes of n_i, *i.e.*,

$$S_i = \left\{ j \middle| (n_i, n_j) \in N \text{ and } (n_i \rightarrow n_j) \in A \right\}.$$

Therefore, the expected execution time of the successive nodes of n_i is defined as

$$\bar{t}_j = p_{ij} t_j \quad \forall j \in S_i.$$

The number of iterations performed in a loop can be either deterministic or random. Figure 5.3 depicts an example of a dataflow loop schema. In the case of a deterministic loop, the number of iterations is fixed before compile-time. Hence, the execution time of the loop can be determined from the number of iterations to be performed. For random loops, the number of iterations is not known in advance. In this case, a probability p_{loop} assigned to the conditional node indicates whether the execution of the loop continues. A typical loop branch is taken with about 90% probability [11]; however, more specific probabilities can be obtained through profiles of application programs. Once p_{loop} is determined we can calculate the expected number of iterations $\bar{I}$ of a random loop.

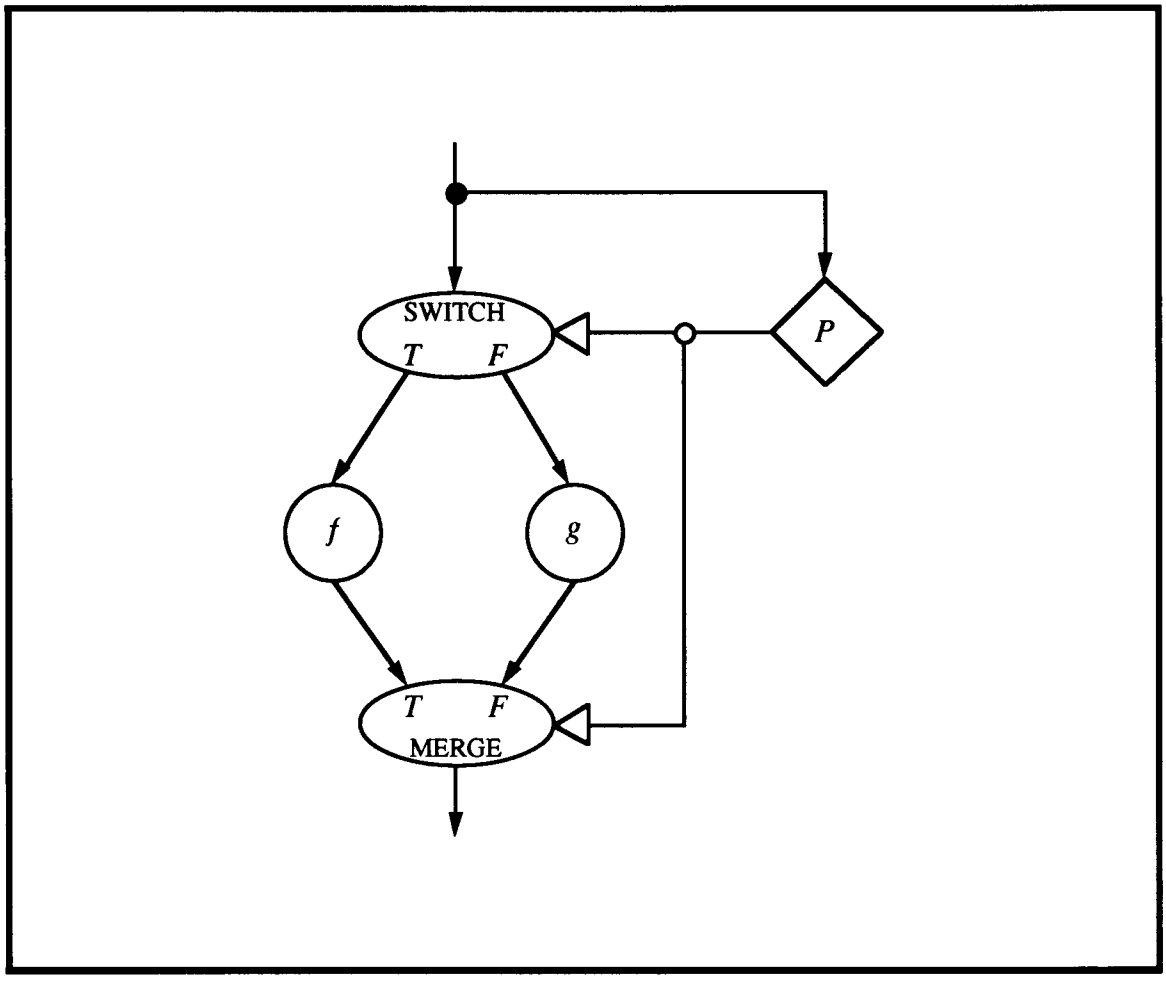


Figure 5.2 A dataflow graph with a conditional node.

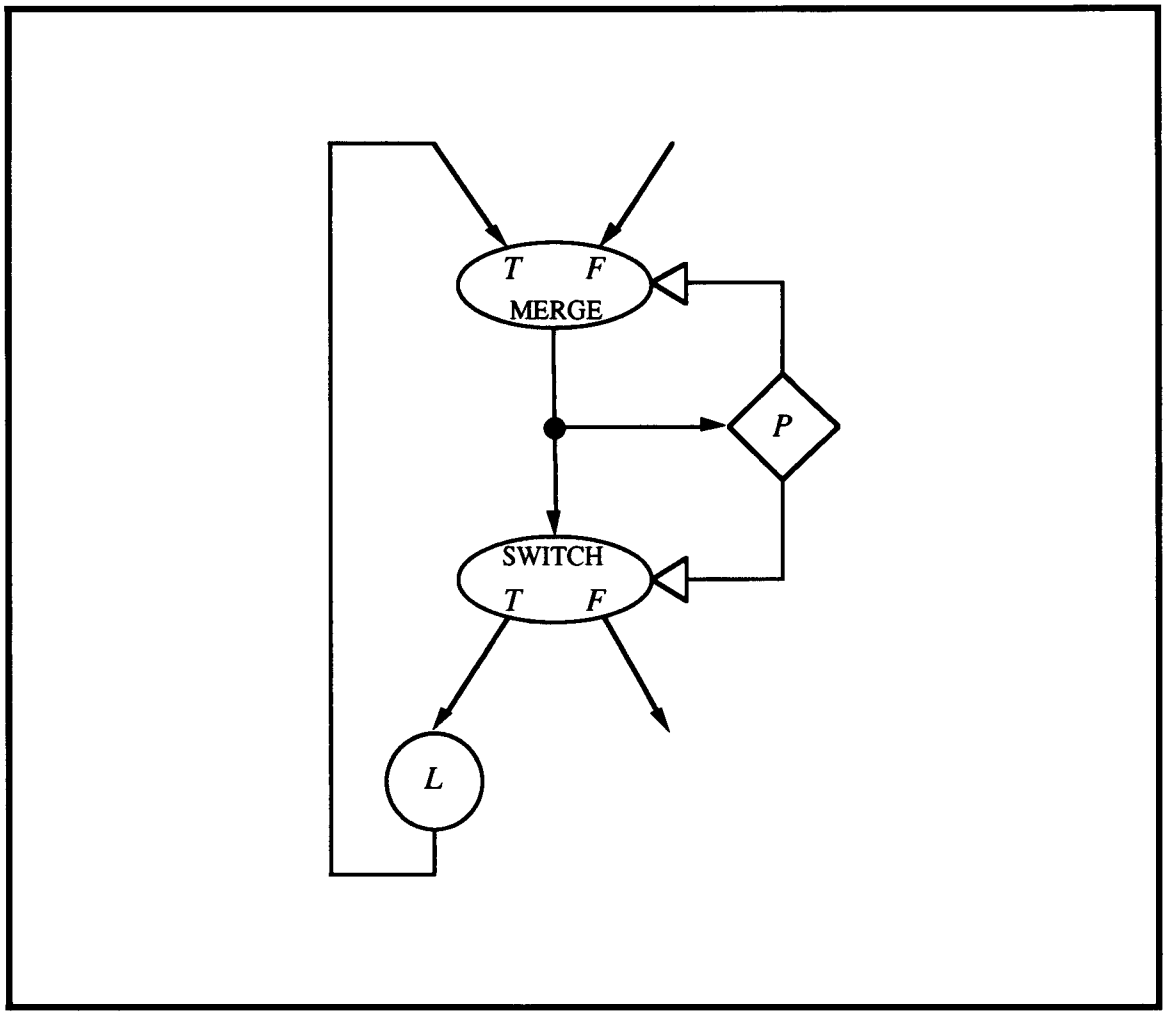


Figure 5.3 An example of a dataflow loop schema.

$$E[I] = \bar{I} = \sum_{i=1}^{\infty} i(p_{loop})^i = \frac{p_{loop}}{(1-p_{loop})^2} \quad \text{for } 0 \le p_{loop} < 1.$$

The expected recursion depth of a function is also unknown at compile-time. This case is similar to the problem of finding the expected number of iterations for a random loop. Thus, a probability p_{rec} can be assigned to the conditional node that determines whether the function is recursively invoked. As above, we can calculate the expected recursion depth $\bar{R}$ of a recursively invoked function

$$E[R] = \bar{R} = \sum_{i=1}^{\infty} i(p_{rec})^i = \frac{p_{rec}}{(1-p_{rec})^2} \quad \text{for } 0 \le p_{rec} < 1.$$

These probabilities for the control nodes can be obtained in a number of ways. They may be provided by the programmer, derived by the compiler, or preferably obtained from profile information [11, 23, 24].

After the expected execution times, t_i, have been determined and assigned to their respective nodes, the allocation process starts by locating the critical path, CP, of a directed graph G. Since the critical path defines the longest path from the root node to the exit node, assignment of the critical path to a single layer minimizes interprocessor communication associated with the critical nodes. When determining the critical path, only the expected execution times, t_i are considered. The critical path of G is found by determining the *earliest time*, e_i, and the *latest time*, l_i, a node can finish executing [15]. The critical path is found with two passes of the dataflow graph G, a *forward pass* and a *backward pass*. During the forward pass, the earliest time for each node is computed using the formula

$$e_j = t_j + \max\{e_i\} \quad \text{if } n_i \rightarrow n_j$$

where t_j is the execution time associated with n_j. During the backward pass, the latest time for each node is computed. Initially, the exit node's latest time, l_n, is assigned $\max\{\text{all } e_i\}$. The remaining l's are calculated in a backward manner using the formula

$$l_i = \min\{l_j - t_j\} \quad \text{if } n_i \rightarrow n_j.$$

Node n_i lies on the critical path if $e_i = l_i$. If a unique critical path does not exist, the algorithm arbitrarily chooses one.

Using the method outlined above, the critical path of the dataflow graph in Figure 5.1 is N1→N2→N5→N9→N11→N13. The set of nodes defining the critical path, N_{CP}, is assigned to an arbitrary layer. The nodes that comprise N_{CP} are then marked and queued into a First-In-First-Out (FIFO) queue Q while maintaining their precedence constraints. All remaining nodes are selected iteratively for allocation as follows: Let N_{marked}^{S-1} represent the set of nodes assigned to a layer at step S-1. Initially, $N_{marked}^{0} = N_{CP}$. A node, n_j, is removed from Q and the set of nodes N_{LDP}^{S} comprising the longest directed path emanating from n_j is formed such that

$$N_{marked}^{S-1} \cap N_{LDP}^{S} = \{\varnothing\} \text{ and}$$

$$N_{marked}^{S} = N_{marked}^{S-1} \cup N_{LDP}^{S}.$$

The method used in finding the longest directed path emanating from n_j involves the same method as finding the critical path without considering the marked nodes.

To determine the placement of the remaining program modules, each set of nodes N_{LDP}^{S} is assigned in an iterative fashion to every possible layer. In this manner, the effects of execution times and communication costs can be weighed against the different layer assignments for N_{LDP}^{S}. Thus, after weighing the effects of N_{LDP}^{S} in each layer, the set of nodes comprising N_{LDP}^{S} is assigned to the layer favoring the objectives of the BLAS. For the iterative assignment of N_{LDP}^{S} to each layer L, the algorithm considers two properties:

- Let T_i^L represent the *completion time of layer i* when N_{LDP}^{S} is assigned to layer L. Then, T^L is the set of T_i^L, *i.e.*,

$$T^L = \left\{ T_i^L \mid i = 0,\ 1,\ \ldots,\ 2^k - 1 \right\}.$$

- Let T_{graph}^{L} represent the *completion time of graph G, i.e.*,

$$T_{graph}^{L} = \max\left\{ T^L \mid L = 0,\ 1,\ \ldots,\ 2^k - 1 \right\}.$$

Each set of nodes N_{LDP}^{S} is then assigned to the layer L that yields the *lowest completion time*, T_{min}, where

$$T_{min} = \min\left\{T_{graph}^{L} \mid L = 0,\ 1,\ \ldots\ 2^{k} - 1\right\}.$$

If T_{min} yields more than one minimum, then a layer is chosen arbitrarily. The set of nodes N_{LDP}^{S} is then marked and queued into Q while maintaining precedence constraints. The allocation phase is completed when the queue Q is empty.

To illustrate our algorithm, consider the allocation of the dataflow graph in Figure 5.1 to a two-dimensional hypercube processor organization. For illustrative purposes, we arbitrarily assume that communication costs between adjacent layers are twice the average execution time of an instruction, *i.e.*, $C_{ij} = C_{ji} = 10$ units of time since the average execution time of the nodes in the graph of Figure 5.1 is 5 units of time. Thus, in our two dimensional hypercube, $C_{01} = 10$, $C_{02} = 10$, $C_{13} = 10$, $C_{23} = 10$, $C_{03} = 20$, and $C_{12} = 20$. The set of nodes comprising the critical path N_{CP}={N1, N2, N5, N9, N11, N13} is arbitrarily assigned to layer 2 as illustrated in Figure 5.4. After the arbitrary assignment of N_{CP} to layer 2, the nodes comprising N_{CP} are then marked and queued. Node N1 is then removed from Q and the set of nodes N_{LDP}^{1}={N3, N7, N10, N12} is formed. The set of nodes N_{LDP}^{1} is then iteratively assigned to every possible layer to determine which assignment yields the lowest completion time. For example, assigning the set of nodes N_{LDP}^{1} to layer 0 yields:

$$T_0^0 = 32,\ T_1^0 = 0,\ T_2^0 = 38,\ T_3^0 = 0,$$

$$T^0 = \{32,\ 0,\ 38,\ 0\}, \text{ and } T_{graph}^{0} = 38.$$

Assigning the set of nodes N_{LDP}^{1} to layer 1 yields:

$$T_0^1 = 0,\ T_1^1 = 42,\ T_2^1 = 38,\ T_3^1 = 0,$$

$$T^1 = \{0,\ 42,\ 38,\ 0\}, \text{ and } T_{graph}^{1} = 42.$$

Assigning the set of nodes N_{LDP}^{1} to layer 2 yields:

$$T_0^2 = 0,\ T_1^2 = 0,\ T_2^2 = 57,\ T_3^2 = 0,$$

$$T^2 = \{0,\ 0,\ 57,\ 0\}, \text{ and } T_{graph}^{2} = 57.$$

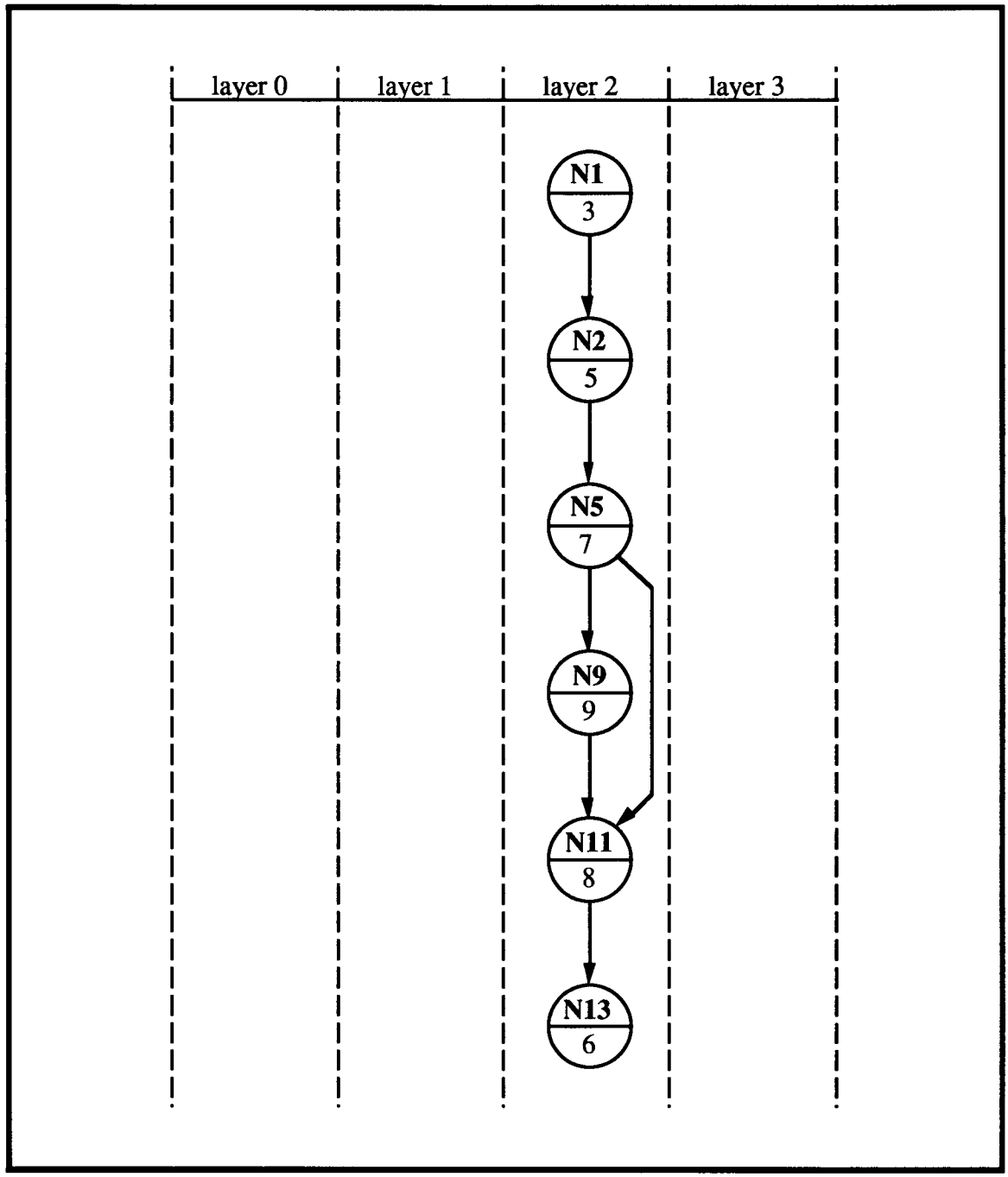


Figure 5.4 Arbitrary layer assignment of the critical path.

Assigning the set of nodes N^1_{LDP} to layer 3 yields:

$$T^3_0 = 0,\ T^3_1 = 0,\ T^3_2 = 38,\ T^3_3 = 32,$$

$$T^3 = \{0,\ 0,\ 38,\ 32\}, \text{ and } T^3_{graph} = 38.$$

According to these results, N^1_{LDP} is assigned to the layer that yields the lowest completion time, *i.e.*,

$$T_{\min} = \min\left\{T^0_{graph} = 38,\ T^1_{graph} = 42,\ T^2_{graph} = 57,\ T^3_{graph} = 38\right\}.$$

Therefore, N^1_{LDP} can be assigned to either layer 0 or layer 3. Layer 3 is chosen arbitrarily as illustrated in Figure 5.5. After all the nodes of Figure 5.1 have been allocated, a layered graph similar to Figure 5.6 is generated. A formal description of the Balanced Layered Allocation Scheme is presented in Algorithm 5.1.

A rudimentary worst-case complexity analysis for the BLAS can be derived as follows. The maximum number of arcs A on N nodes is $N(N\text{-}1)$. Therefore, a cyclic to acyclic transformation has complexity

$$O(N^2).$$

Determining the set of nodes that comprise the critical path can be determined in

$$O(N + A) = O(N^2) \text{ [23].}$$

Once the critical path has been identified, the complexity of determining the LDPs decreases with each iteration. In the worst case this occurs one node at a time. With the removal of each node, the complexity of determining the remaining LDPs decreases by N with each iteration. Therefore, the total time required to determine all the LDPs in G is

$$O\left(\sum_{i=1}^{N-2} N^2 - iN\right) = O\left(\frac{1}{2}\left(N^3 - N^2 - 2N\right)\right) = O(N^3).$$

Note that the upper limit of the summation is N-2 since a graph consisting of two or fewer nodes will not have an LDP, just a CP (*i.e.*, an LDP can only occur on a directed, connected graph with three or more nodes). Finally, the completion time of every layer can be determined in a similar manner to finding the CP. This is done p times for each LDP where p is the number of processors. Thus each LDP requires a total time of

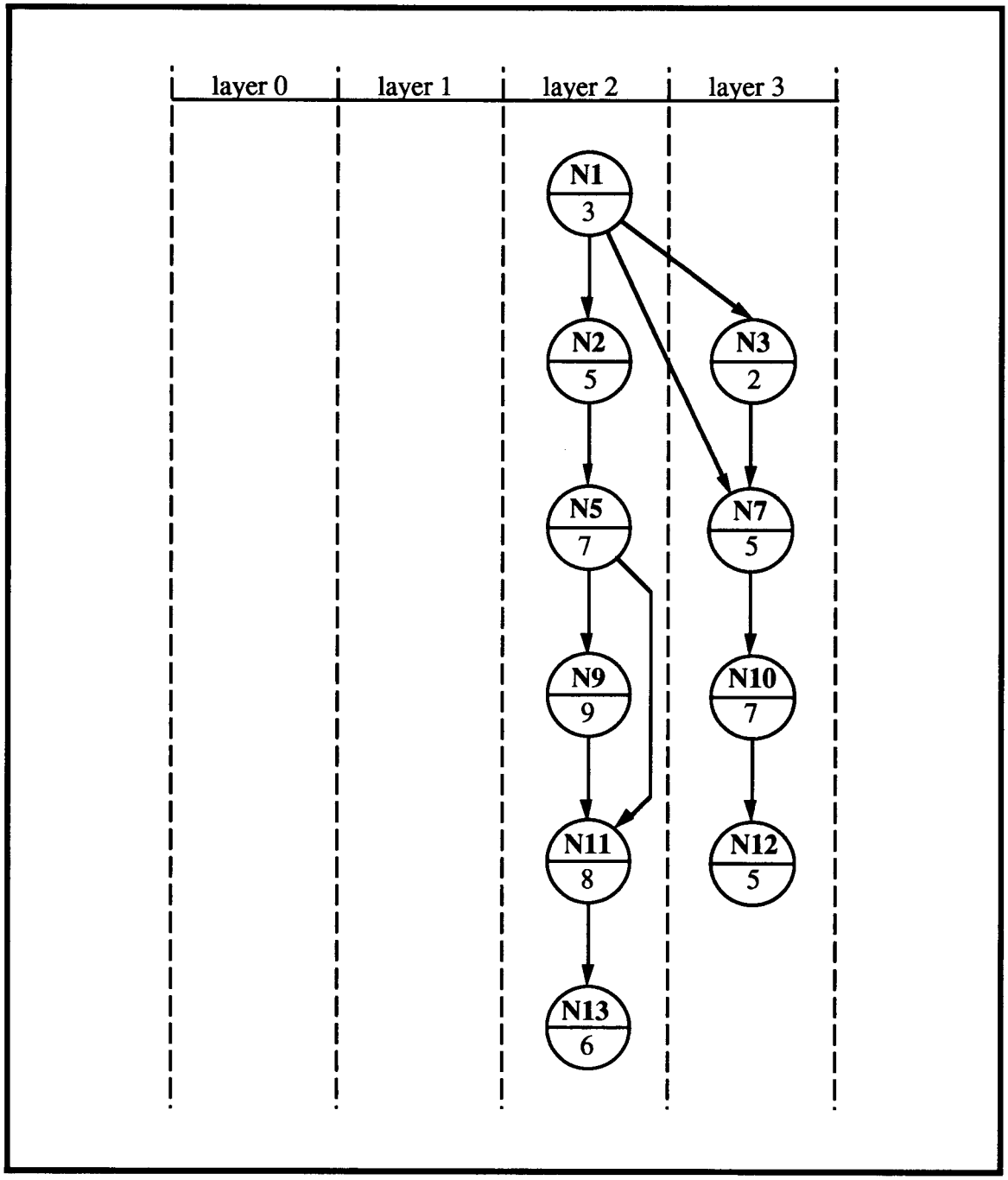


Figure 5.5 The state of the layers after the assignment of N^1_{LDP} = {N3, N7, N10, N12}.

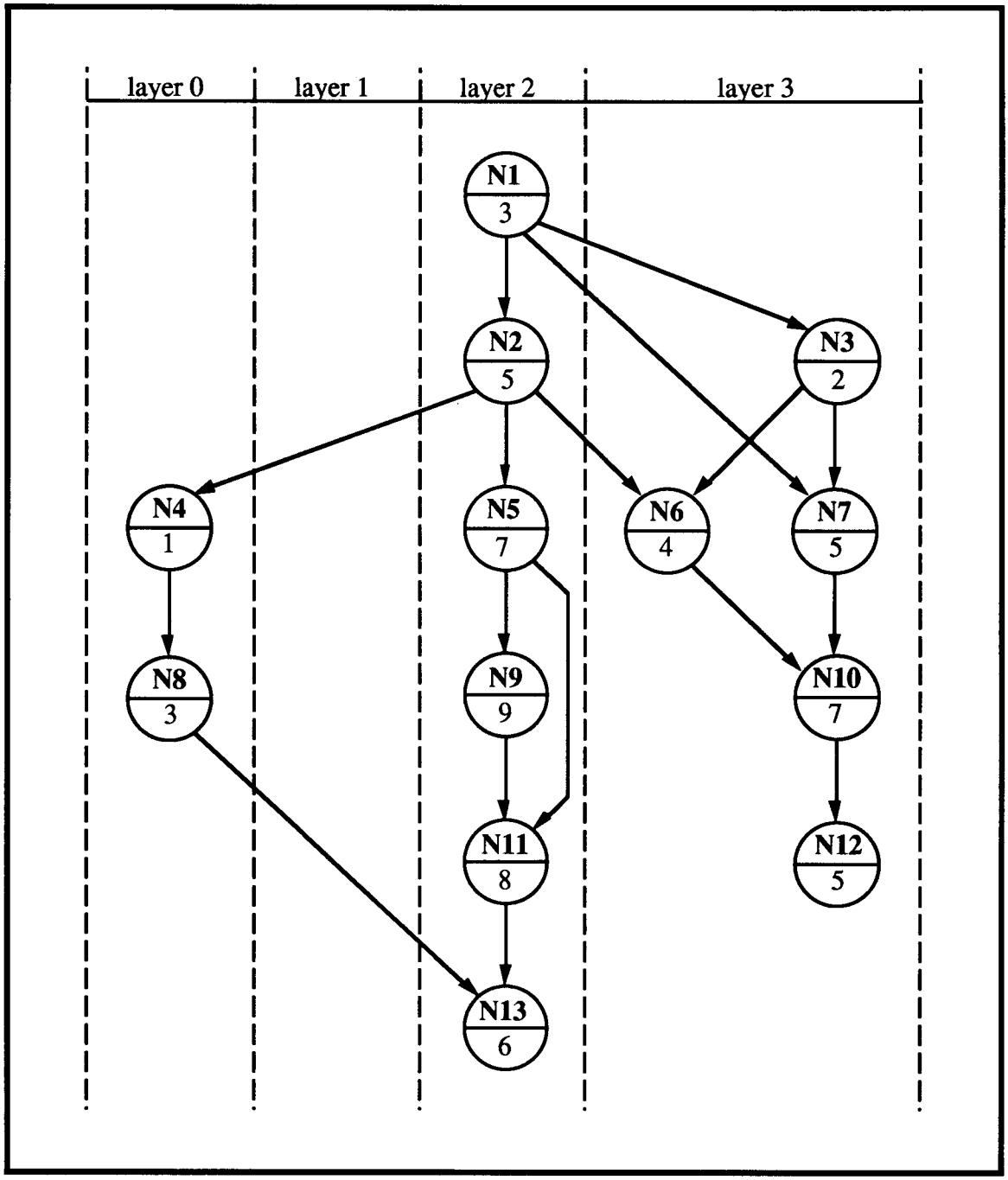


Figure 5.6 The balanced layered graph of Figure 5.1.

Algorithm 5.1 The Balanced Layered Allocation Scheme.

Input: An arbitrary dataflow graph $G(N,A)$, where all $n_i \in N$ have an expected execution time t_i.

Output: A set of 2^k layers having a one-to-one correspondence with processors in a k-dimensional hypercube.

Allocation

- Determine the set of nodes belonging to the critical path and assign these nodes to an arbitrary layer.
- Queue the set of nodes comprising the critical path into a FIFO queue Q while maintaining precedence constraints.

WHILE (Q is not empty) **DO**

- **CALL ALLOCATE**

END

PROCEDURE ALLOCATE
BEGIN

- Remove the node n_i from the front of the queue Q.

FOR (All the arcs emanating from n_i) **DO**

- Determine the longest directed path N_{LDP}^{S} emanating from n_i such that:

$$N_{marked}^{S-1} \cap N_{LDP}^{S} = \{\varnothing\} \text{ and}$$
$$N_{marked}^{S} = N_{marked}^{S-1} \cup N_{LDP}^{S}.$$

ENDFOR

- Insert the set of nodes N_{LDP}^{S} into the queue Q.

FOR L=0 TO $2^k - 1$ **DO**

FOR i=0 TO $2^k - 1$ **DO**

- Find the completion time of layer i, T_i^L, if N_{LDP}^{S} is assigned to layer L.

ENDFOR

- Let T^L represent the set of T_i^L, *i.e.*,

$$T^L = \left\{ T_i^L \mid i = 0,\ 1,\ \ldots,\ 2^k - 1 \right\}.$$

- Find the completion time of graph G, T_{graph}^{L}, *i.e.*,

$$T_{graph}^{L} = \max\left\{ T^L \mid L = 0,\ 1,\ \ldots,\ 2^k - 1 \right\}.$$

ENDFOR

- Assign the set of nodes N_{LDP}^{S} to the layer that yields the lowest completion time, T_{min}, where

$$T_{min} = \min\left\{ T_{graph}^{L} \mid L = 0,\ 1,\ \ldots,\ 2^k - 1 \right\}.$$

- Queue the set of nodes N_{LDP}^{S} into the queue Q while maintaining precedence constraints.
- Mark each node in N_{LDI}^{S} to indicate that these nodes have been allocated.

END

$$pO(N^2).$$

Since assigning, queuing, and marking nodes can all be done in $O(N)$ time, the total algorithm complexity of the Balanced Layered Allocation Scheme is

$$O(N^2)+O(N^2)+O(N^3)+pO(N^2)+O(N)=O(N^3) \text{ for } p \le N.$$

Note that $O(N^3)$ is the worst case complexity of the BLAS, and in reality we can expect a much more optimistic time complexity. For example, the number of arcs A is much less than N^2 for most dataflow graphs. Additionally, we assumed that the time complexity of determining LDPs decreases incrementally with each iteration. However, we can expect the complexity to decrease at a much faster rate since, in general, LDPs are composed of more than a single node. Moreover, determining an LDP uses the same approach as determining the CP, yet on a smaller subset of nodes. It is then reasonable to assume that the time to calculate a single LDP is less than the time to determine the CP. Without loss of generality, we can assume that the time to determine an LDP falls in the range of $O(N)$ to $O(N^2)$. Our simulation studies indicate that, because of the size of threads, the time complexity to determine an LDP should be slightly larger than $O(N)$ and much smaller than $O(N^2)$. Therefore, a more realistic time complexity for determining all the LDPs would be

$$\gamma O(N^{4/3})$$

where γ is the total number of LDPs and $O(N^{4/3})$ is a realistic time to find each LDP. Our simulation studies show values of γ in the range of 7 to 38 for graphs with 16 to 192 nodes, respectively. This suggests a $\gamma = N^{2/3}$ relationship that reduces the time of determining all the LDPs to

$$N^{2/3}O(N^{4/3}) = O(N^2).$$

This reduces the overall time complexity of the BLAS to

$$O(N^2)+O(N^2)+O(N^2)+pO(N^2)+O(N)=pO(N^2) \text{ for } p \le N.$$

5.3 Modified BLAS

The time complexity analysis of the BLAS illustrates that a large number of PEs may drastically increase the time necessary to allocate a program to processors. This occurs because each LDP is assigned iteratively to *every* layer in order to determine the assignment that yields the lowest completion time.

The time to allocate a program to processors can be reduced efficiently by considering the allocation of each set of nodes N_{LDP}^{S} to a layer that is at most one processor away from a parent thread of N_{LDP}^{S}. That is, let L_{adj}^{S} represent the set of layers and all adjacent layers of any assigned parent thread to N_{LDP}^{S} at step S. Each set of nodes N_{LDP}^{S} is assigned in an iterative fashion to every layer in L_{adj}^{S}. Thus, after weighing the effects of N_{LDP}^{S} in each of the identified layers, the nodes comprising N_{LDP}^{S} are assigned to the layer that yields the lowest completion time of the program. This slight adjustment to the BLAS is referred to as the *Modified BLAS*. Only a slight modification to **PROCEDURE ALLOCATE** in the existing BLAS algorithm is necessary for the implementation of the Modified BLAS. The refinements to **PROCEDURE ALLOCATE** are depicted in Algorithm 5.2.

The time complexity of the Modified BLAS is similar to the BLAS except for the iterative assignment of LDPs to layers. The completion time of every layer can be determined in a similar manner to finding the CP which has complexity

$$O(N^2).$$

This is performed $1+\log_2 p$ times for each LDP, where p is the number of processors in the hypercube. Thus, the total time required to assign an LDP to a layer has complexity

$$(1+\log_2 p)O(N^2) = \log_2(2p)O(N^2).$$

The total complexity of the Modified BLAS is

$$O(N^2)+O(N^2)+O(N^2)+\log_2(2p)O(N^2)+O(N) = \log_2(2p)O(N^2) \quad \text{for } p \le N.$$

Algorithm 5.2 Refinements to Procedure Allocate for the Modified Balanced Layered Allocation Scheme.

PROCEDURE ALLOCATE
BEGIN
- Remove the node n_i from the front of the queue Q.

FOR (All the arcs emanating from n_i) **DO**
- Determine the longest directed path N_{LDP}^{S} emanating from n_i such that:

$$N_{marked}^{S-1} \cap N_{LDP}^{S} = \{\varnothing\} \text{ and}$$
$$N_{marked}^{S} = N_{marked}^{S-1} \cup N_{LDP}^{S}.$$

ENDFOR
- Insert the set of nodes N_{LDP}^{S} into the queue Q.
- Let L_{adj}^{S} represent the set of layers and all adjacent layers of any assigned parent thread to N_{LDP}^{S}.

FOR L=0 TO $2^k - 1$ **DO**
  IF L$\in L_{adj}^{S}$ **THEN**
    FOR i=0 TO $2^k - 1$ **DO**
    - Find the completion time of layer i, T_i^L, if N_{LDP}^{S} is assigned to layer L.

    ENDFOR
  ELSE
    FOR i=0 TO $2^k - 1$ **DO**
    - $T_i^L = \infty$

    ENDFOR
  ENDIF
- Let T^L represent the set of T_i^L, *i.e.*,

$$T^L = \left\{ T_i^L \mid i = 0,\ 1,\ \ldots,\ 2^k - 1 \right\}.$$

- Find the completion time of graph G, T_{graph}^{L}, *i.e.*,

$$T_{graph}^{L} = \max\left\{ T^L \mid L = 0,\ 1,\ \ldots,\ 2^k - 1 \right\}.$$

ENDFOR
- Assign the set of nodes N_{LDP}^{S} to the layer that yields the lowest completion time, T_{min}, where

$$T_{min} = \min\left\{ T_{graph}^{L} \mid L = 0,\ 1,\ \ldots,\ 2^k - 1 \right\}.$$

- Queue the set of nodes N_{LDP}^{S} into the queue Q while maintaining precedence constraints.
- Mark each node in N_{LDI}^{S} to indicate that these nodes have been allocated.

END

5.4 Simulation Results

To analyze the effectiveness of the BLAS, seven dataflow graphs were chosen for our simulation studies. The first dataflow graph, GRAPH1, is the graph of Figure 5.1 used to illustrate the BLAS. The second dataflow graph, entitled EX1, is a 12-node graph utilized in [15] to illustrate the VL allocation scheme. The final five dataflow graphs were also obtained from [15]. These graphs consist of: (1) a 16-node dataflow graph entitled QUICKSORT for the implementation of the quicksort algorithm, (2) a 32-node dataflow graph entitled NWP32 representing a numerical weather prediction program, (3) an 82-node graph entitled 82V that performs assignment and sequencing operations, (4) a 146-node graph entitled NWP147 which is a more complex version of the numerical weather prediction problem, and (5) a 193-node graph entitled L2 which is a more complex version of the assignment and sequencing program.

Our simulation studies are based on the following assumptions:

- The execution time t_i of each node is assigned randomly from a uniform distribution with an average of five time units.
- The inter-PE communication delays are varied based on a ratio of communication to execution time, C/t.
- The inter-PE communication delays associated with non-adjacent processors is an integral multiple of the fundamental communication delay C_{ij}.
- The intra-PE communication delay is negligible when compared to the inter-PE communication delay.

To illustrate the results of the simulation studies, L2 was selected as a representative dataflow graph. The simulation studies were performed on two different processor topologies to illustrate the broad effectiveness of the proposed allocation scheme. First, simulation results were obtained for a Multistage Interconnection Network (MIN) based dataflow system where a constant interprocessor communication delay

exists between all pairs of PEs. This topology was picked for simulation studies because the VL allocation scheme showed promising results on a MIN topology [15]. Second, simulation results were obtained for a more widely used processor topology, namely, the hypercube.

5.4.1 Simulation Results for a MIN Topology

As mentioned earlier, the primary motivation for simulating the BLAS on a MIN topology is to compare the proposed allocation scheme on the targeted topology of the VL allocation scheme. To compare these allocation schemes, several studies have been performed. Figure 5.7 depicts the total execution time of L2 versus the number of PEs when the C/t ratio is 0/5 (*i.e.*, inter-PE communication delays are negligible). As expected, the total execution time for the BLAS decreases as the number of PEs increases. Also included in Figure 5.7 is the total execution time of the VL allocation scheme. In some allocations (*i.e.*, 8 PEs) the VL allocation scheme made a poor initial allocation that can not be improved in the optimization phase. Note that the total execution time for the VL allocation scheme starts to saturate to the lower bound as the number of PEs reaches 17. This is because the L2 graph has a maximum parallelism of 17. Also note that the BLAS saturates before the VL allocation scheme. This is because the BLAS considers communication costs in its initial assignment of nodes to layers. The lower bound in the simulation results represents the ideal minimum execution time of a program executed on a parallel computer. That is, the lower bound represents the total execution time dictated by the critical path of the application program.

To analyze the effectiveness of the BLAS with communication delays, the execution times of L2 have been simulated for a C/t ratio of 10/5 and a C/t ratio of 20/5 as depicted in Figure 5.8 and Figure 5.9, respectively. As can be seen, the proposed

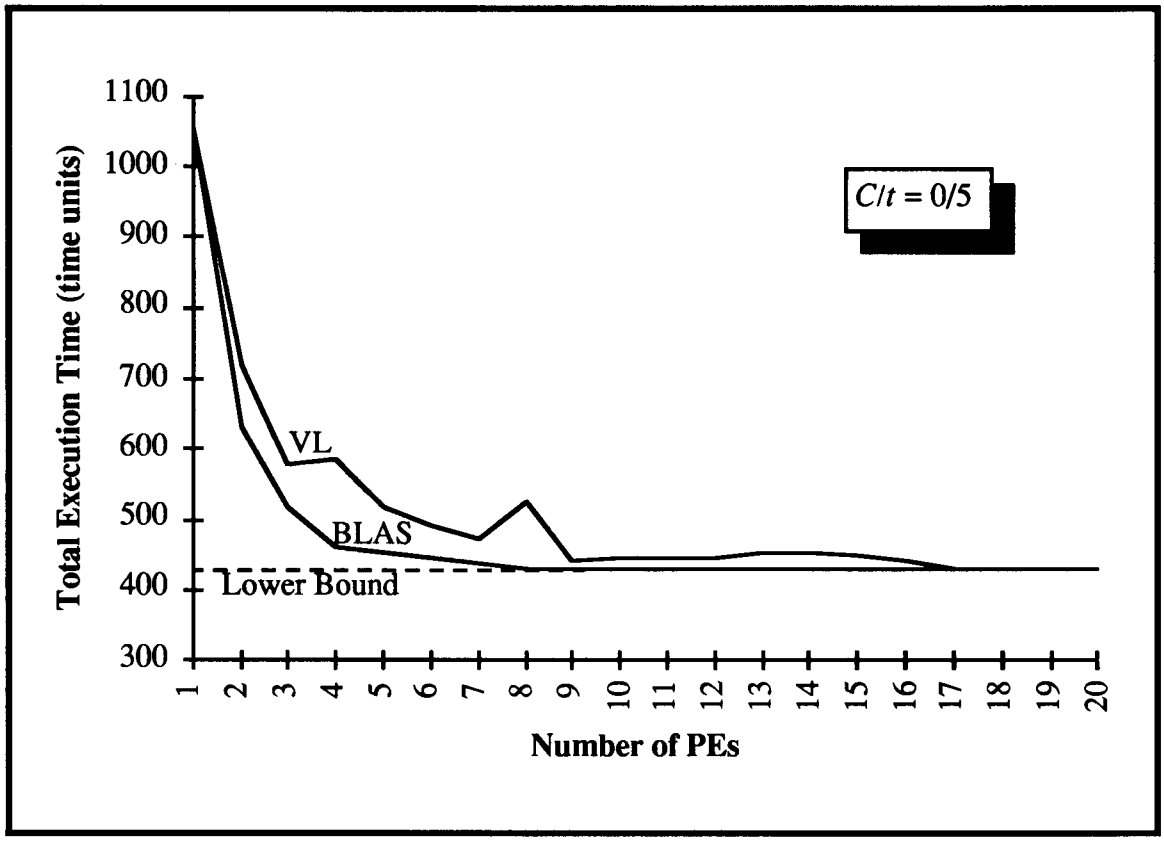


Figure 5.7 A plot of total execution times versus number of PEs for $C/t = 0/5$ on a MIN topology.

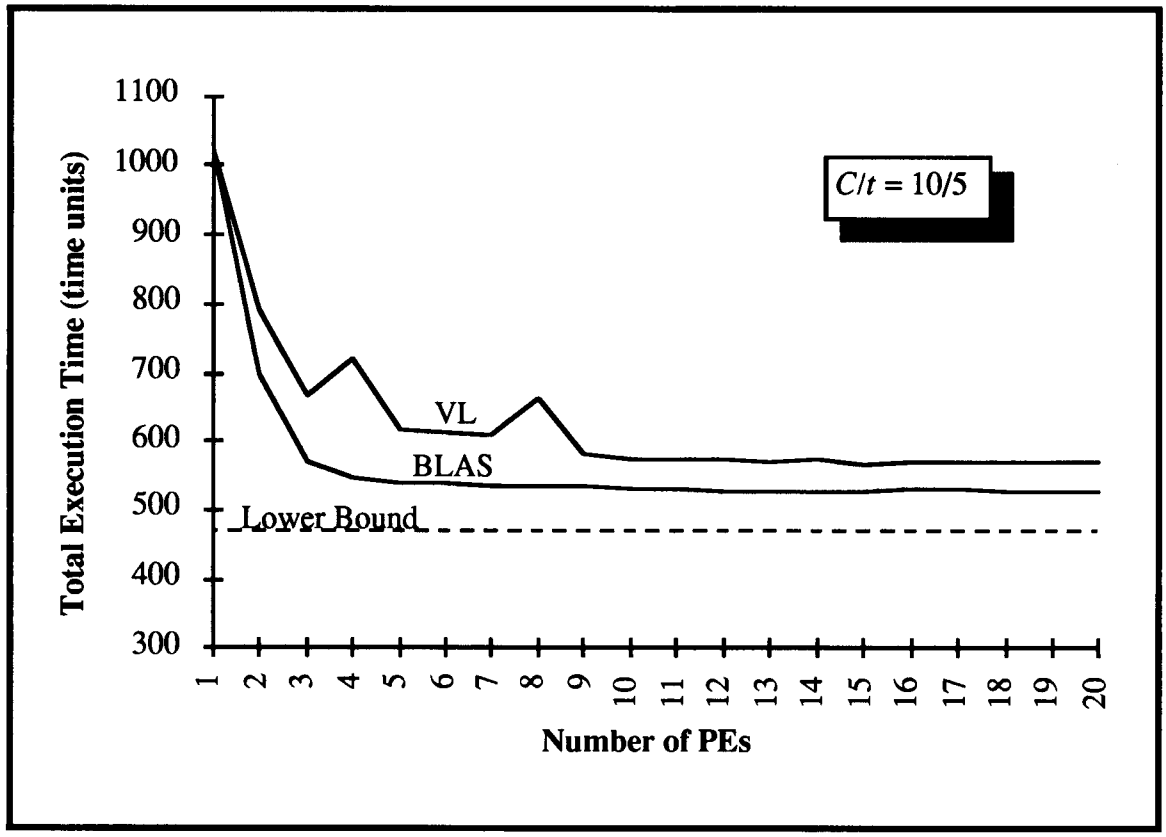


Figure 5.8 A plot of total execution times versus number of PEs for $C/t = 10/5$ on a MIN topology.

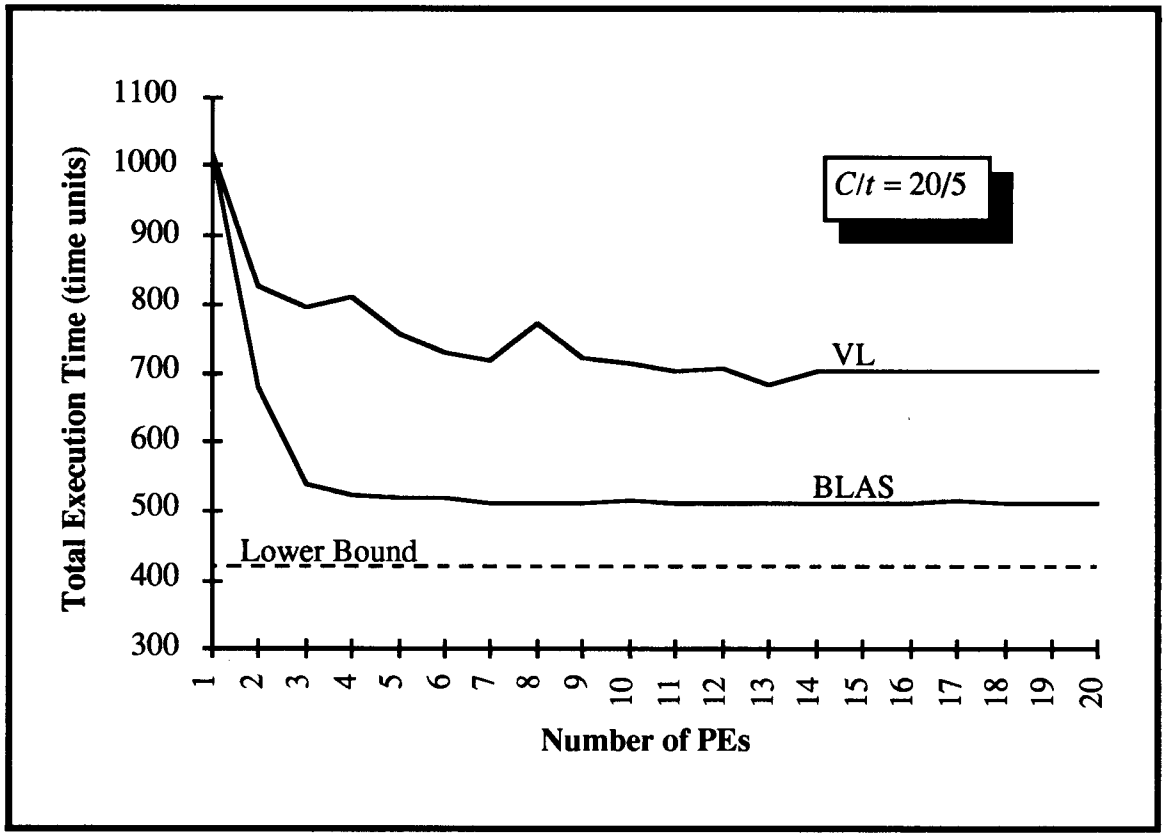


Figure 5.9 A plot of total execution times versus number of PEs for $C/t = 20/5$ on a MIN topology.

method significantly improves, relative to the VL allocation scheme, as the C/t ratio increases.

As another indication of the BLAS performance, *speedup* for varying numbers of PEs and C/t ratios is illustrated in Figure 5.10. Included in the speedup plot of Figure 5.10 is an indication of the ideal speedup, called *average parallelism*, T_{avg}. That is,

$$T_{avg} = \frac{T_s}{T_{CP}}$$

where T_s is the serial execution time of the program and T_{CP} is the critical path execution time. As can be seen in Figure 5.10, the BLAS saturates to the average parallelism faster than the VL allocation scheme when the C/t ratio is 0/5. Also note that for a large communication overhead (*i.e.*, C/t = 25/5), the BLAS achieves a significantly better speedup than the VL allocation scheme.

Finally, Table 5.1 shows the average performance improvement of the BLAS relative to the VL allocation scheme for a MIN topology. Our findings indicate that the BLAS shows significant improvement over the VL allocation scheme for most of the seven dataflow graphs studied. Only the extremely small 12-node graph entitled EX1 experienced a negative performance improvement. The BLAS performs well because it attempts to find the best allocation for each LDP by considering the effects of communication costs on execution times in the initial assignment of nodes to processors. Conversely, the VL allocation scheme uses load balancing techniques that do not consider communication costs when initially assigning LDPs to processors. It is only after the initial allocation of nodes to processors that the VL allocation scheme attempts to minimize the execution time of a program by considering communication costs along the critical path. In the next section we analyze the effectiveness of the BLAS on the widely used hypercube topology.

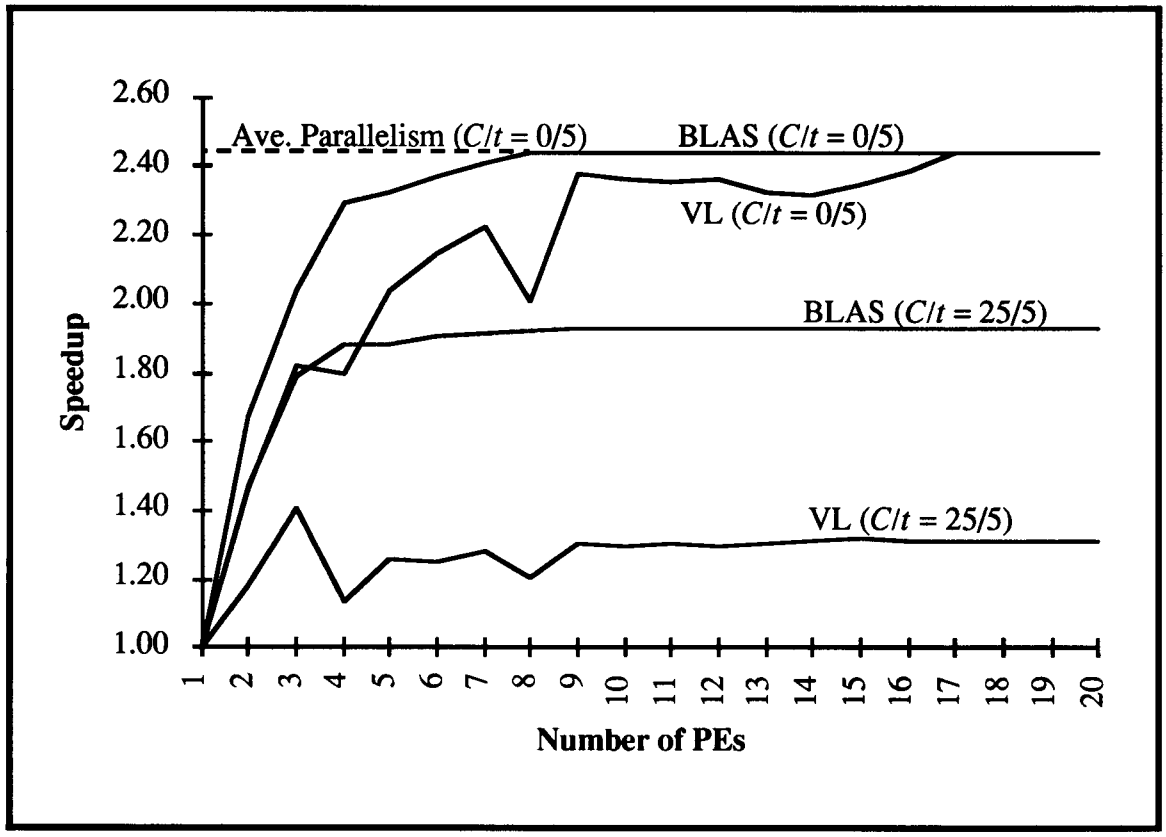


Figure 5.10 A plot of speedup versus number of PEs for a spectrum of *C/t* ratios on a MIN topology.

Table 5.1 Average performance improvement of the BLAS relative to the VL allocation scheme for a MIN topology.

C/t	EX1	GRAPH1	QUICKSORT	NWP32	82V	NWP147	L2
0/5	7.5	8.7	11.1	19.8	10.1	10.0	8.2
2/5	17.7	4.7	8.2	12.5	12.5	5.8	8.0
5/5	13.8	5.6	2.9	9.8	11.2	5.6	8.6
10/5	-2.1	1.5	7.6	7.6	17.2	4.6	11.9
15/5	0.0	4.3	12.5	2.8	7.7	7.0	11.9
20/5	-1.2	1.1	7.0	5.8	27.8	8.0	37.6
25/5	-1.1	2.4	9.7	7.1	39.4	7.2	44.4

5.4.2 Simulation Results for a Hypercube Topology

To analyze the effectiveness of the BLAS on a hypercube topology, L2 is used again as a representative dataflow graph. Figure 5.11 illustrates the total execution time of L2 versus the number of PEs when the *C/t* ratio is 0/5 on a hypercube topology. As can be seen, the execution time of the BLAS decreases at a faster rate than the VL allocation scheme.

To analyze the effectiveness of the BLAS with communication delays, the execution times of L2 have been simulated for a *C/t* ratio of 2/5 and a *C/t* ratio of 10/5 as depicted in Figure 5.12 and Figure 5.13, respectively. The effects of communication delays become noticeable in the VL allocation scheme for a *C/t* ratio of 2/5 and an allocation of 16 or more PEs. The increase in the overall execution time occurs because the VL allocation scheme does not consider communication costs in the initial allocation of nodes to processors. Instead, the initial allocation of nodes is based on a load balancing technique that neglects communication costs of distant PEs (see section 5.1). When the *C/t* ratio is increased from 2/5 to 10/5, as shown in Figure 5.13, the effects of communication costs become very detrimental to the execution times of the VL allocation scheme. However, the BLAS demonstrates a gradual decrease in the execution time of the program as the number of PEs increases. This gradual decrease in the execution time of a program to a given saturation level occurs for all seven of the dataflow graphs studied. Simulation results have indicated that the BLAS is very successful in minimizing communication overhead in the initial allocation of nodes to layers, therefore, an optimization phase, similar to the one used in the VL allocation scheme, is not necessary.

Figure 5.14 illustrates the speedup of L2 as a function of *C/t* ratios. Note that as the *C/t* ratio increases, the speedup decreases as would be expected. In addition, the

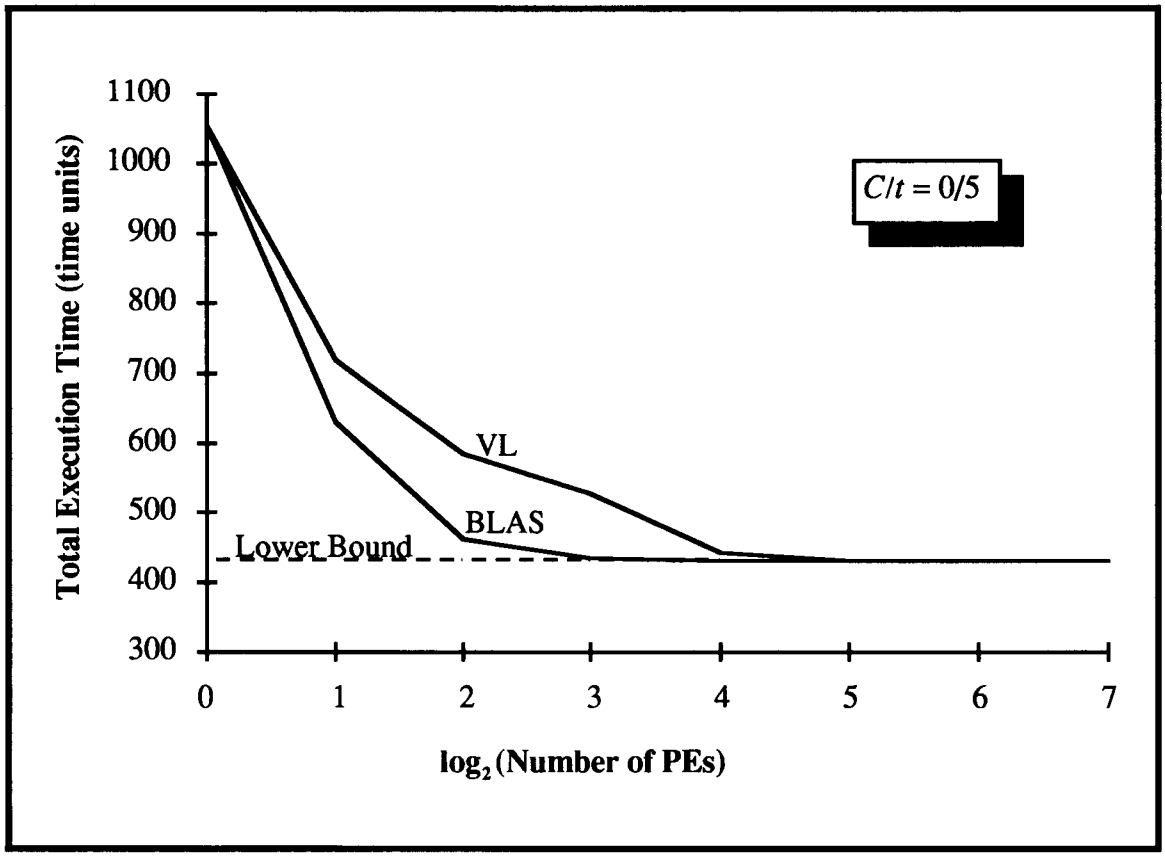


Figure 5.11 A plot of total execution times versus number of PEs for $C/t = 0/5$ on a hypercube topology.

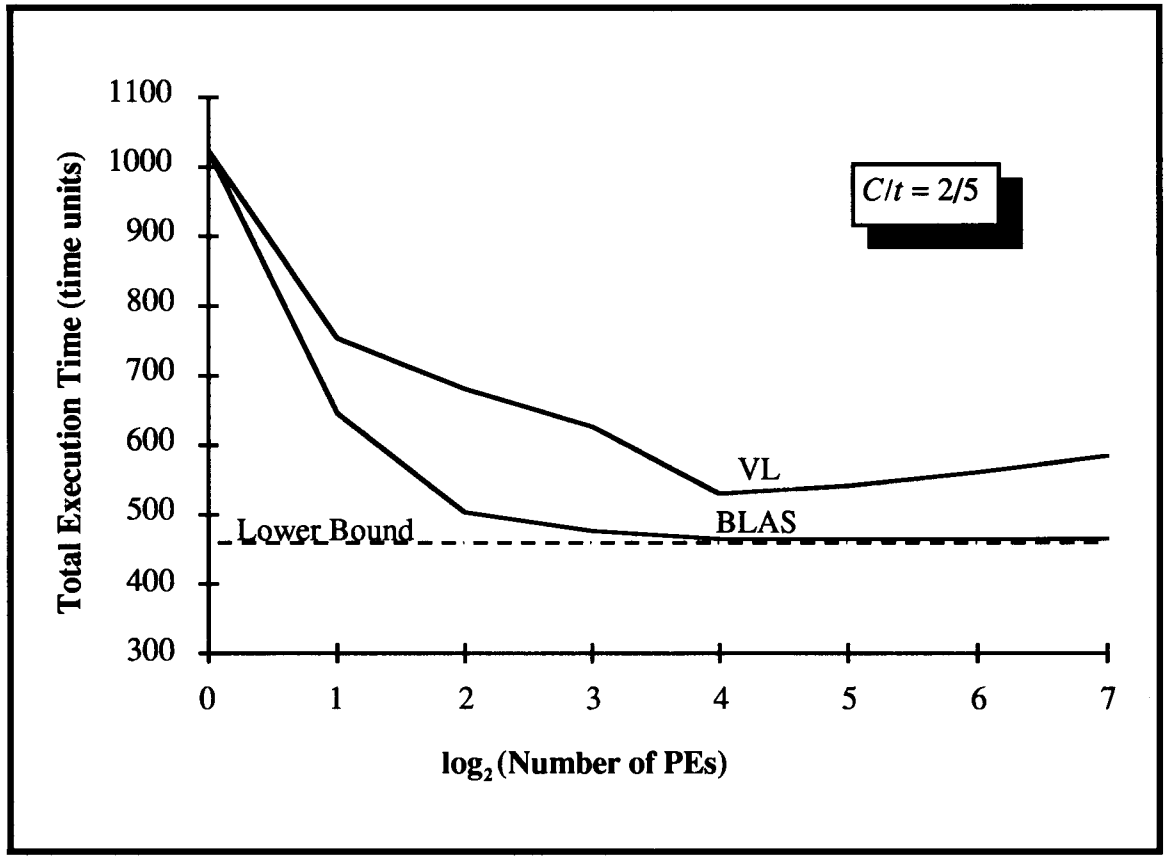


Figure 5.12 A plot of total execution times versus number of PEs for C/t = 2/5 on a hypercube topology.

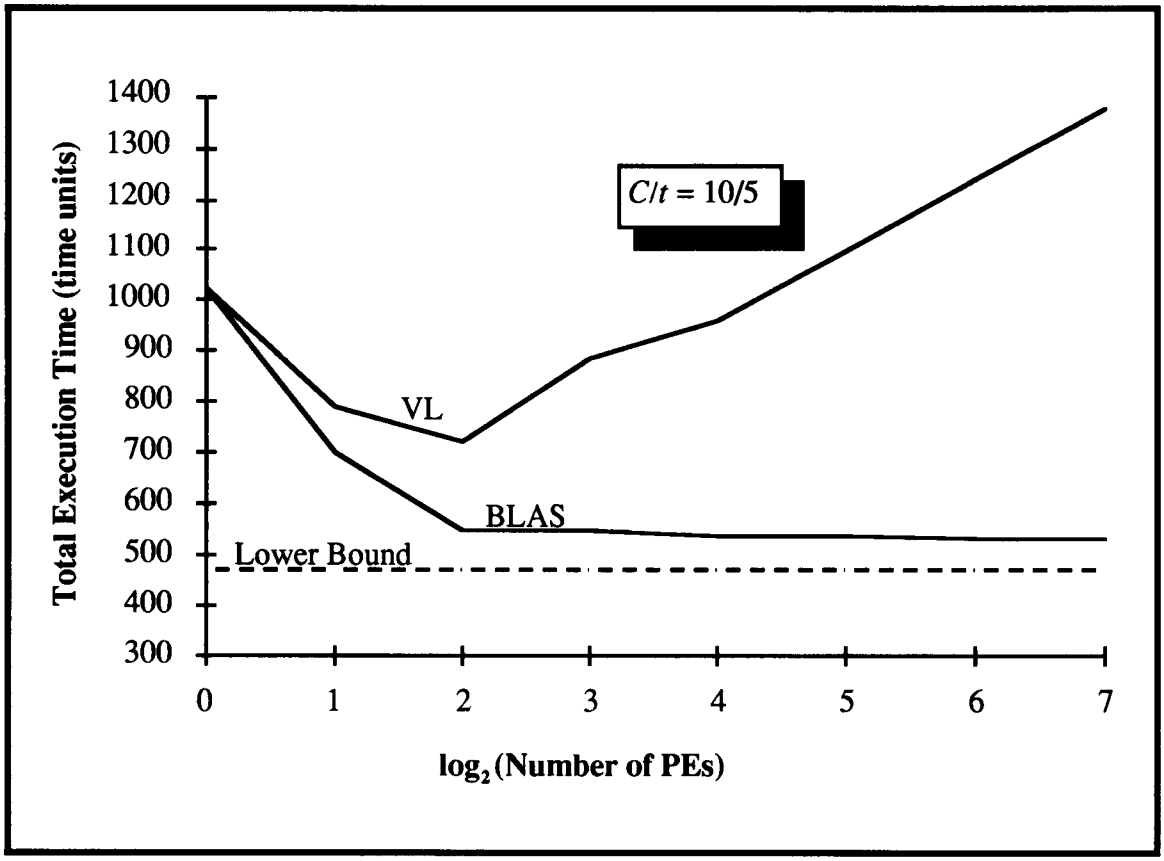


Figure 5.13 A plot of total execution times versus number of PEs for C/t = 10/5 on a hypercube topology.

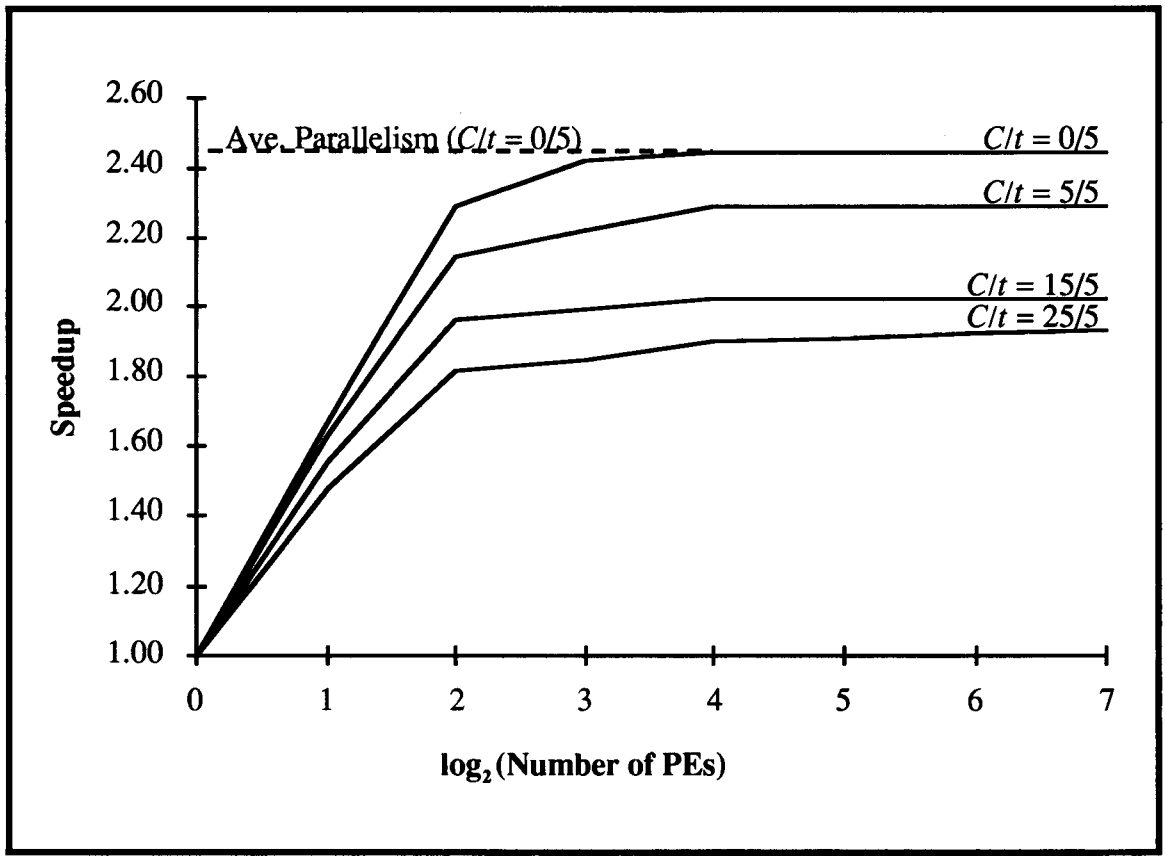


Figure 5.14 A plot of speedup for varying *C/t* ratios on a hypercube topology.

speedup curve for the *C/t* ratio of 0/5 reaches the ideal speedup indicated by the average parallelism.

The effectiveness of the BLAS becomes more apparent as the *C/t* ratio increases. Figure 5.15 is a plot of the performance improvement of the BLAS with respect to the VL allocation scheme for varying *C/t* ratios. It is interesting that a performance increase occurs in Figure 5.15 when the number of available PEs is small and communication delays are negligible. This is an area where the VL allocation scheme has performed poorly [15].

Finally, Table 5.2 presents the overall performance improvements of the BLAS relative to the VL allocation scheme for all seven dataflow graphs Our findings indicate that the BLAS outperforms the VL allocation on all but the smallest (*i.e.*, 13-node) dataflow graphs. On all the dataflow graphs that have been reconstructed from real applications, the BLAS outperforms the VL allocation scheme substantially and, in general, the performance improvement of the BLAS increases with an increase in the *C/t* ratio.

5.4.3 Simulation Results for the Modified BLAS

Several studies have been performed to analyze the effectiveness of the Modified BLAS on a hypercube topology. In these simulation studies, L2 is used as a representative dataflow graph. Figure 5.16 illustrates the total execution time versus the number of PEs when the *C/t* ratio is 0/5. As expected, the total execution time of the Modified BLAS decreases as the number of PEs increases. Also included in Figure 5.16 is the total execution time of the VL allocation scheme and the BLAS. The Modified BLAS performs better than the VL allocation scheme when the number of PEs is less than eight and it performs slightly worse when the number of PEs exceeds eight. This occurs because the VL allocation scheme performs load balancing in its separation phase

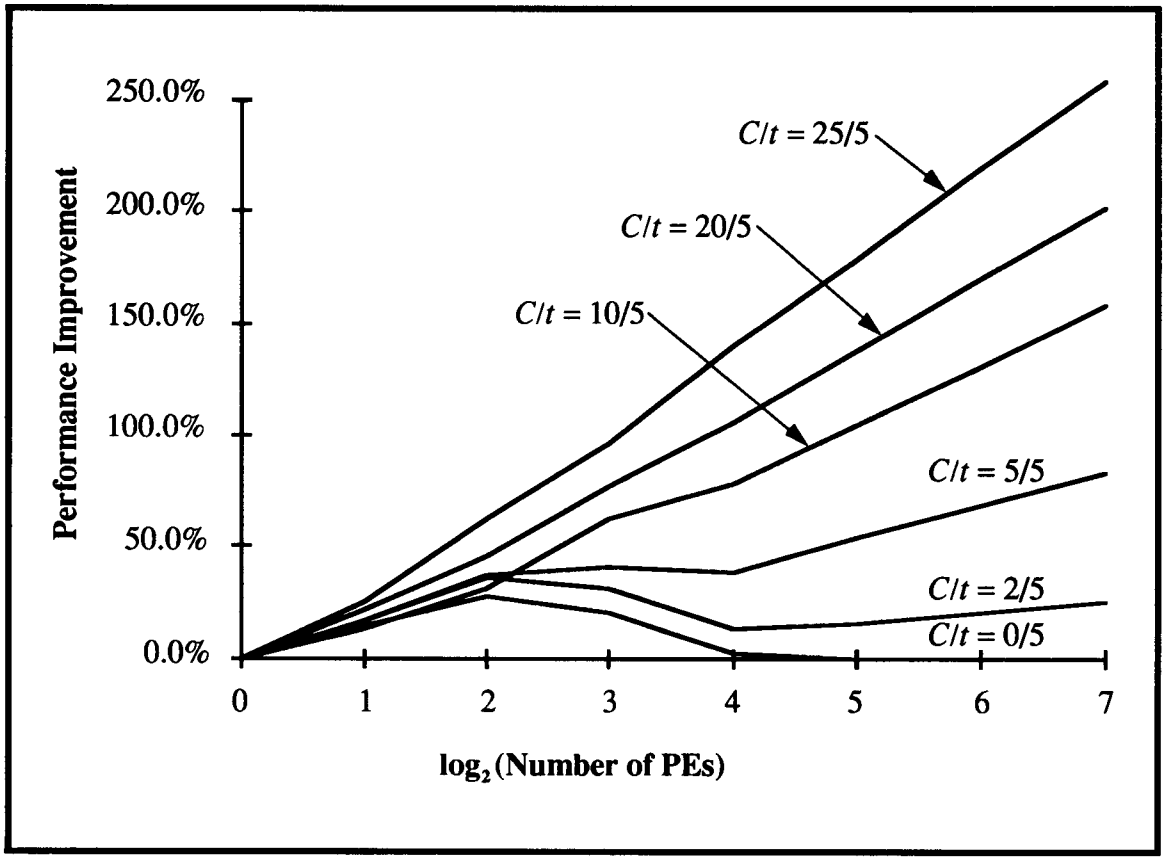


Figure 5.15 Performance improvement of the BLAS relative to the VL allocation scheme for varying *C/t* ratios.

Table 5.2 Average performance improvement of the BLAS relative to the VL allocation scheme for a hypercube topology.

C/t	EX1	GRAPH1	QUICKSORT	NWP32	82V	NWP147	L2
0/5	7.5	4.3	14.8	10.6	8.8	13.4	10.8
2/5	8.4	0.0	11.8	18.2	16.3	16.2	18.9
5/5	13.8	0.0	11.0	9.4	19.0	22.4	31.1
10/5	-1.4	3.3	18.3	13.9	41.1	42.6	48.1
15/5	4.0	9.6	46.8	23.1	14.5	50.9	33.5
20/5	5.8	15.9	35.1	23.7	63.2	74.7	64.8
25/5	-1.1	-4.4	28.9	36.8	80.1	69.0	83.9

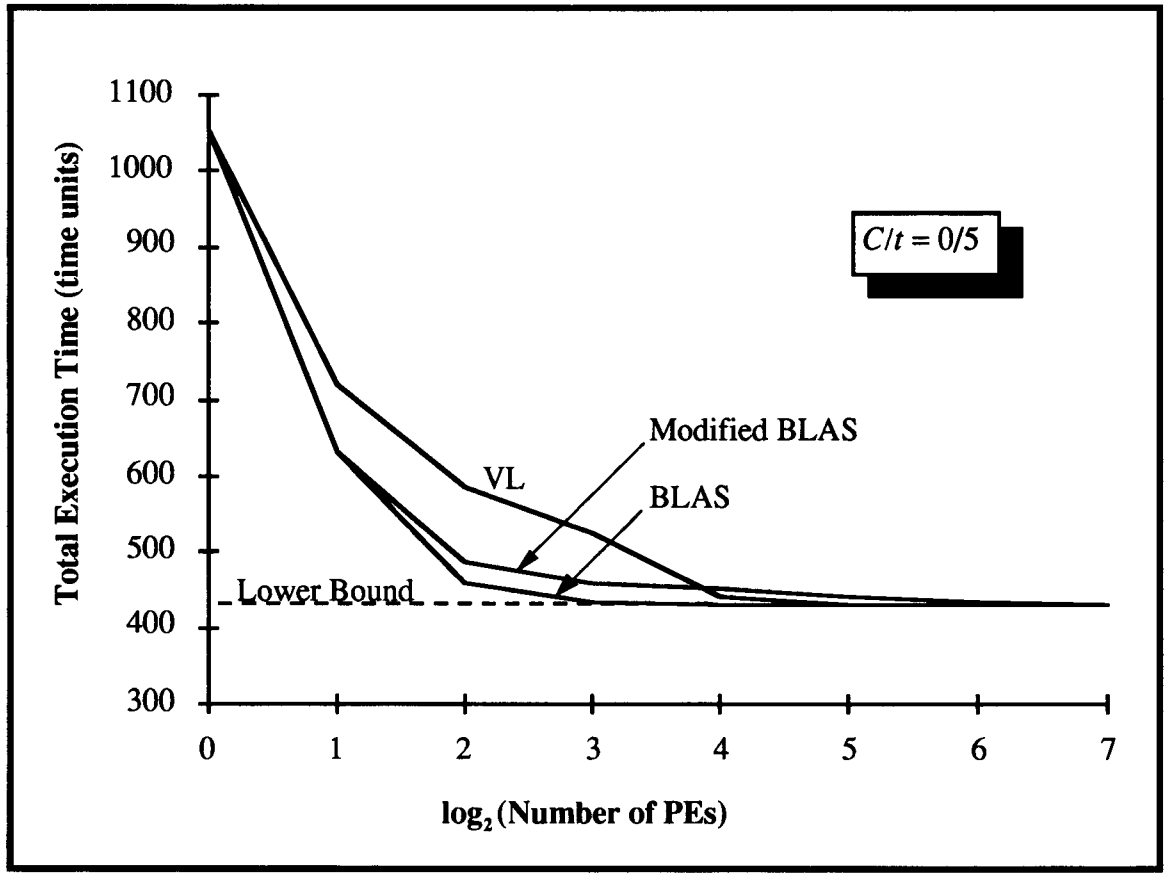


Figure 5.16 A plot of total execution times versus number of PEs for C/t = 0/5 on a hypercube topology.

and the *C/t* ratio of Figure 5.16 is 0/5. Since there is no penalty for communication, the VL allocation scheme takes advantage of initially allocating nodes to PEs based on load balancing techniques. In contrast, the Modified BLAS only allocates nodes to adjacent PEs. Also note that the BLAS performs as well, if not better, than either the VL allocation scheme or the Modified BLAS.

To analyze the effectiveness of the Modified BLAS with communication delays, the execution times of L2 have been simulated for a *C/t* ratio of 5/5 and a *C/t* ratio of 20/5 as depicted in Figure 5.17 and Figure 5.18, respectively. In both cases, the Modified BLAS performs better than the VL allocation scheme when communication costs are considered. Additionally, the Modified BLAS does not perform as well as the BLAS for small *C/t* ratios, but performs as well as the BLAS for large *C/t* ratios.

Finally, Table 5.3 shows the average performance improvement of the BLAS relative to the Modified BLAS for a hypercube topology. Our findings indicate that the BLAS outperforms the Modified BLAS in most situations. This makes sense since the BLAS is allowed to allocate threads to any layer while the Modified BLAS is restricted to allocating threads to layers that are at most one layer from a parent thread. However, Table 5.3 illustrates that the Modified BLAS is comparable to the BLAS and performs exceptionally well for large *C/t* ratios.

5.5 Summary

In this chapter, the problem of allocating dataflow graphs to a dataflow computer has been discussed. The concept of the Balanced Layered Allocation Scheme, which utilizes CP and LDP heuristics to handle the allocation problem, was introduced. The central idea of this method is to arrange the nodes of a dataflow graph into layers that have a one-to-one correspondence with processors in a given topology. During the allocation, CP and LDP heuristics determine the set of nodes that are to be assigned to

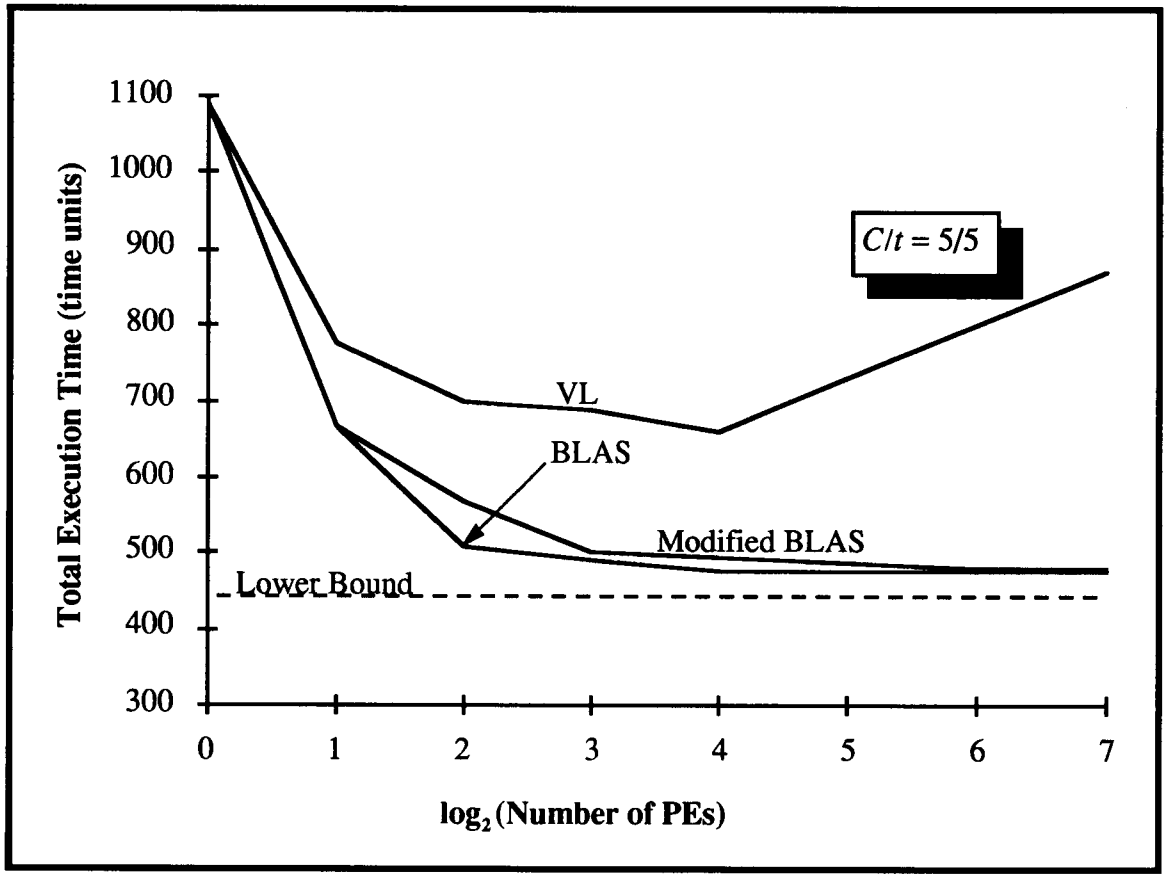


Figure 5.17 A plot of total execution times versus number of PEs for C/t = 5/5 on a hypercube topology.

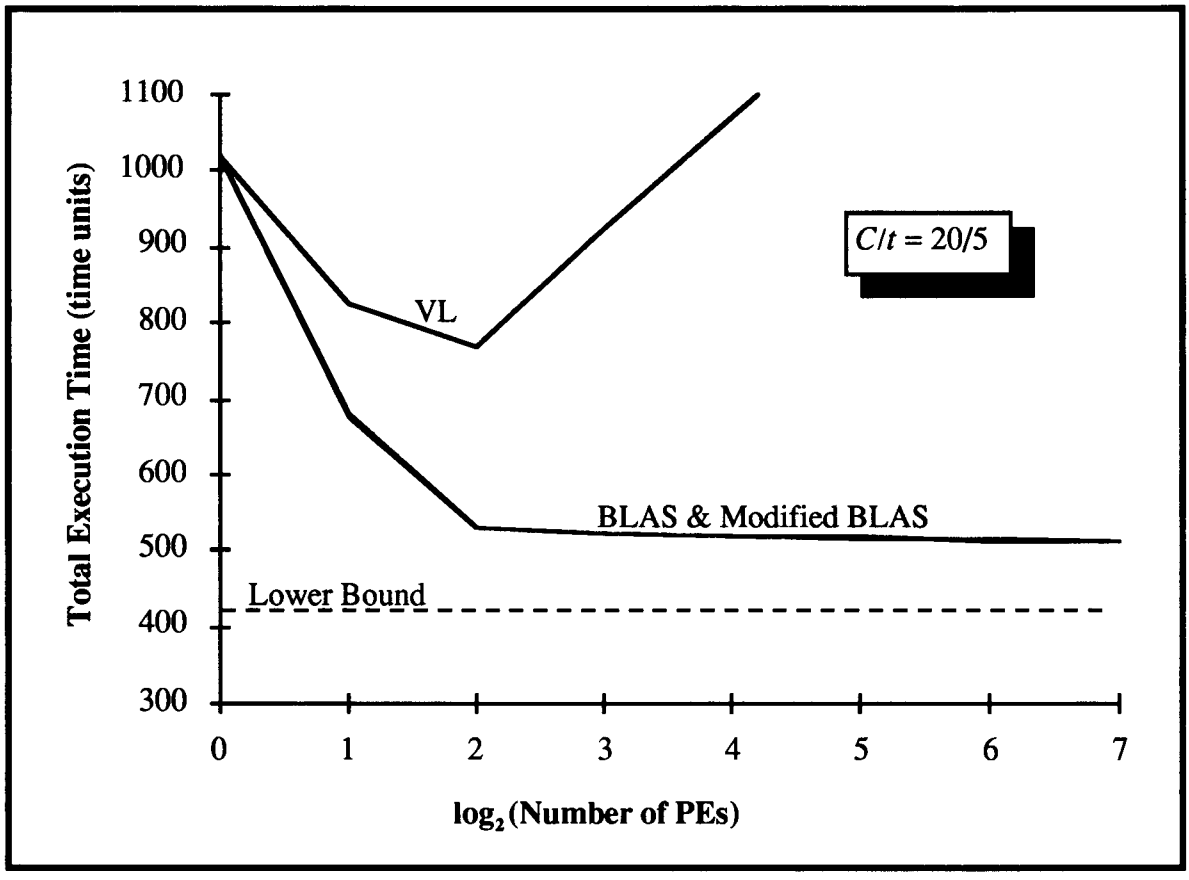


Figure 5.18 A plot of total execution times versus number of PEs for C/t = 20/5 on a hypercube topology.

Table 5.3 Average performance improvement of the BLAS relative to the Modified BLAS for a hypercube topology.

C/t	EX1	GRAPH1	QUICKSORT	NWP32	82V	NWP147	L2
0/5	0.0	0.0	0.0	9.5	3.0	29.4	3.0
2/5	0.0	-1.0	0.8	16.2	2.7	16.4	2.0
5/5	0.0	1.5	0.0	5.8	1.2	9.5	3.3
10/5	0.0	0.0	0.0	-3.3	1.9	19.7	0.3
15/5	0.0	0.0	0.0	0.4	1.5	8.2	0.5
20/5	0.0	0.0	0.0	0.7	0.5	3.7	0.3
25/5	0.0	0.0	0.0	0.0	0.4	-3.5	0.1

processors. Each set of nodes is assigned in an iterative fashion to every possible layer. In the case of the Modified BLAS, each set of nodes is assigned in an iterative fashion to a layer that is at most one processor away from its parent thread. In this manner the effects of execution times and communication costs can be weighed against each other for every possible layer assignment. Sets of nodes are then assigned to the layer that yields the earliest completion time of the program.

Simulation studies indicate that the proposed allocation scheme is effective in reducing communication overhead and thus the overall execution time of a program distributed on a MIN or hypercube dataflow computer. Overall, the BLAS showed promising improvements over the VL allocation scheme. In addition, the proposed algorithm need not be restricted to the allocation of dataflow graphs to MIN or hypercube based dataflow multiprocessors. Since the method of assigning nodes to processors considers execution times as well as communication delays, the Balanced Layered Allocation Scheme is general enough to be applied to any multiprocessor system.

6. Conclusion and Further Study

In this work, we have discussed the control-flow versus dataflow approach for a high speed multiprocessor environment. Due to its asynchronous and functional characteristics, the dataflow model of computation was suggested as an alternative to the more traditional control-flow approach. However, before dataflow can become a viable alternative to the control-flow model of computation, problems such as program allocation must be resolved.

The importance of an allocation scheme that properly maps tasks to processing elements was discussed. A scheme for partitioning and mapping dataflow graphs to general dataflow multiprocessors was introduced. During the allocation scheme, CP and LDP heuristics are used to determine the set of instructions that are assigned to processors. The effects of communication costs, execution times, and total program execution time are weighed against each other in an iterative manner to determine the final allocation of instructions to processors. Finally, a simulation study was performed to analyze the effectiveness of the proposed allocation scheme.

Further study of the Balanced Layered Allocation Scheme can be pursued in many directions. First, an in-depth study can be performed on the communication behaviors of programs allocated by the BLAS. This study should focus on the distances of assigned program modules with respect to a targeted processor topology. The study may find that program modules need only be iteratively placed on layers that are within a predetermined distance of their parent module, as in the case for the Modified BLAS. Perhaps, for certain topologies (*e.g.*, meshes), an iterative placement on adjacent layers would be sufficient. If this modification occurs, the use of an optimization phase, similar to the one utilized by the VL allocation scheme, may be used to minimize other inter-PE communication behaviors.

Second, identify cases where a course-grain approach is more efficient for the allocation of programs to processors than a fine-grain approach. For example, a conditional statement will either execute the **TRUE** operations or the **FALSE** operations, not both. Treating both the **TRUE** and **FALSE** nodes separately will most likely yield an allocation to two different processors. Instead, it becomes more relevant to determine the execution time of the entire conditional statement and place the entire conditional on one PE. That is, treat the conditional nodes as a macro-actor that can then be assigned to one processor. Several cases similar to this one should be identified and incorporated into the BLAS.

Third, investigate how to group multiple threads into a single node to aid in the pipeline of multithreading computers. For example, Monsoon may process up to eight threads simultaneously in the pipeline of one PE. Therefore, it seems reasonable that eight independent threads should be grouped together and assigned to one PE in order to keep its pipeline full. Incorporating multithreading capabilities into the BLAS will also require a knowledge of the communication behaviors between all the threads for a particular application program.

Fourth, the simulation needs to be modified to more accurately model the communication costs of a hypercube. Our simulation assumed that communication between adjacent processors, PE_i and PE_j required a constant time C_{ij} and communication between non-adjacent processors required an integral multiple of C_{ij}. These assumptions were made to simplify the simulation and to determine the effectiveness of the BLAS. Now that the allocation scheme has shown its effectiveness, it is necessary to more accurately model the communication behaviors of a targeted topology based on loading constraints and interprocessor communication delays.

Bibliography

[1] Ackerman, W. B., "Data Flow Languages," *Computer*, Feb. 1982, pg. 15-25.

[2] Agerwala, T. and Arvind, "Data Flow Systems," *Computer*, Feb. 1982, pg. 10-13.

[3] Arvind and Culler, D. E., "Dataflow Architectures," *Annual Review of Computer Science,* 1986, Vol. 1, pg. 225-253.

[4] Arvind and Nikhil, R. S., "Executing a Program on the MIT Tagged-Token Dataflow Architecture," *IEEE Transactions on Computers*, March 1990, Vol. 39, No. 3, pg. 300-318.

[5] Buehrer, R. and Ekanadham, K., "Incorporating Data Flow Ideas into von Neumann Processors for Parallel Execution," *IEEE Transactions on Computers*, December 1987, Vol. C-36, No. 12, pg. 1515-1522.

[6] Dinning, A., "A Survey of Synchronization Methods for Parallel Computers," *Computer,* July 1989, pg. 66-77.

[7] Dubois, M., Scheurich, C., and Briggs, F. A., "Synchronization, Coherence, and Event Ordering in Multiprocessors," *Computer*, Feb. 1988, pg. 9-21.

[8] Grafe, V. G. and Hoch, J. E., "The Epsilon-2 Multiprocessor System," *Journal of Parallel and Distributed Computing*, 1990, Vol. 10, pg. 309-318.

[9] Grafe, V. G. and Hoch, J. E., "The Epsilon-2 Hybrid Dataflow Architecture," *Proceedings of Compcon90,* March 1990, pg. 88-93.

[10] Hayes, J. P., Mudge, T., Stout, Q. F., Colley, S., and Palmer, J., "A Microprocessor-based Hypercube Supercomputer," *IEEE Micro*, Oct. 1986, Vol. 6, No. 5, pg. 6-17.

[11] Hennesy, J. L. and Patterson, D. A., *Computer Architecture: A Quantitative Approach.*, San Mateo, California: Morgan Kaufmann Publishers, Inc. 1990.

[12] Hwang, K. and Briggs, F. A., *Computer Architecture and Parallel Processing*, McGraw-Hill, Inc. 1984.

[13] Iannucci, R. A. "Toward a Dataflow/von Neumann Hybrid Architecture," *Proceedings of the 15th International Symposium on Computer Architecture,* 1988, pg. 131-140.

[14] Katseff, H. P., "Incomplete Hypercubes," *IEEE Transactions on Computers*, May 1988, Vol. 37, No. 5, pg. 604-608.

[15] Lee, B., Hurson, A. R., and Feng, T. Y., "A Vertically Layered Allocation Scheme for Data Flow Systems," *Journal of Parallel and Distributed Computing*, 1991, Vol. 11, pg. 175-187.

[16] Lubeck, O. M., "A User's View of Dataflow Architectures," *Proceedings of Compcon90,* March 1990, pg. 84-87.

[17] Papadopoulos, G. M. and Culler, D. E., "Monsoon: An Explicit Token Store Architecture," *1990 IEEE 17th Annual International Symposium on Computer Architecture*, May 1990, pg. 82-91.

[18] Papadopoulos, G. M. and Traub, K. R., "Multithreading: A Revisionist View of Dataflow Architectures," *18th Annual International Symposium on Computer Architecture,* 1991, pg. 342-351.

[19] Quinn, M. J., *Designing Efficient Algorithms for Parallel Computers*, 2nd ed., New York: McGraw-Hill Book Company, 1991.

[20] Rattner, J., "Concurrent processing: A new direction in scientific computing," *National Computer Conference*, 1985, pg. 157-166.

[21] Saad, Y. and Schultz, M. H., "Topological Properties of Hypercubes," *IEEE Transactions on Computers,* July 1988, Vol. 37, No. 7, pg. 867-872.

[22] Sakai, S., Yamaguchi, Y., Hiraki, K., Kodama, Y., and Yuba, T., "An Architecture of a Dataflow Single Chip Processor," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989, pg. 46-53.

[23] Sarkar, V. and Hennessy, J., "Partitioning Parallel Programs for Macro-Dataflow," *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, 1986, pg. 202-211.

[24] Sarkar, V. and Hennessy, J., "Compile-time Partitioning and Scheduling of Parallel Programs," *Proceedings of the Symposium on Compiler Construction*, 1986, pg. 17-26.

[25] Seitz, C. L., "The Cosmic Cube," *Communications of the ACM*, January 1985, Vol. 28, No. 1, pg. 22-33.

[26] Squire, J. and Palais, S. M., "Programming and design considerations of a highly parallel computer," *Proceedings of the AFIP Spring Joint Computer Conference*, 1963, Vol. 23, pg. 395-400.

[27] Yamaguchi, Y., Sakai, S., Hiraki, K., Kodama, Y., and Yuba, T., "An Architectural Design of a Highly Parallel Dataflow machine," *Information Processing*, 1989, pg. 1155-1160.

APPENDIX A

Verification of the Simulation Studies

The simulation studies of the Balanced Layered Allocation Scheme were performed on a Macintosh IIsi in the programming language C. The simulation program was originally designed for the VL allocation scheme and was modified to simulate the Balanced Layered Allocation Scheme.

A number of steps were taken to verify the correctness of the simulation program. First, a trace facility was incorporated into the simulation program to verify the correctness of the results. The trace facility aided in the initial debugging process and allowed monitoring of the allocation process. For example, the trace facility allowed monitoring of each longest directed path (LDP) to determine the proper execution of the algorithm. Second, the obtained results are logically acceptable in the sense that they are consistent with what one would have expected them to be (*i.e.*, there was no evidence of gross miscalculations in the results). Finally, the simulation program was tested with two independent sets of data for which we had manually calculated the results. In both cases, the simulation results agreed with the calculated results.