PIPELINED DATA PARALLEL ALGORITHMS —

CONCEPT, DESIGN, AND MODELING

By

Chung-Ta King

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science
Michigan State University
East Lansing, MI 48824

1988

ABSTRACT

PIPELINED DATA PARALLEL ALGORITHMS — CONCEPT, DESIGN, AND MODELING

By

Chung-Ta King

As multiprocessors become more and more popular, the need for efficient parallel algorithms becomes more and more imminent. In this thesis, the concept, design, implementation, and modeling of a class of parallel algorithms, called *pipelined data parallel algorithms*, is discussed. When running on distributed-memory multiprocessors, pipelined data parallel algorithms exploit processor level macro-pipelining to achieve a balance between computation and communication.

The concept of pipelined data parallel computations is first introduced by contrasting with other styles of parallel computations. Various considerations in designing efficient parallel algorithms, especially data parallel algorithms, are discussed. A new paradigm, called *large-grain pipelining*, to facilitate the design of pipelined data parallel algorithms on distributed-memory multiprocessors is introduced.

To estimate the performance of pipelined data parallel algorithms, an analytic model is introduced. The accuracy of the analytic model is studied by comparing the analytic results with experimental results from a 64-node NCUBE multiprocessor. The close match between these two sets of results indicates that the analytic model can also be used to determine the optimal design parameters, such as the granularity, for a given problem instance.

A systematic approach for designing pipelined data parallel algorithms on distributed-memory multiprocessors is presented. Starting from a nested-loop program, the approach restructures the program through a series of transformations. From data dependencies between the loops, data and operations in the program are grouped together. The grouping technique allows the control over the granularity so that a balanced computation and communication results in the generated pipelined data parallel program.

Finally, major contributions of this thesis are summarized and future work is outlined.

To my parents and my wife

ACKNOWLEDGEMENTS

I wish to thank my advisor, Dr. Lionel M. Ni, for his guidance over the years. Without the research environment that he provided and his intellectual advice and inspiration, this dissertation would not have been possible.

I would also like to thank Dr. Anthony S. Wojcik for his valuable suggestions and comments on this dissertation and careful reading the manuscript. I am grateful to Dr. A. Esfahanian, who is always willing to answer my questions and provide me with valuable references. I also wish to express my appreciation to Dr. D. Feldman for his encouragement and support.

A person cannot accomplish anything without the help from others. I would like to thank all the people who helped me during my stay at Michigan State. In particular, I wish to acknowledge my friends T. Gendreau, E. Wu, Y.H. Liu, S.W. Chen, J. Liao, W.H. Chou, T.H. Pan, B. McMillin, Youran Lan, D. Ra, X.H. Sun, F. Fotouhi, and T. Znati.

Finally, I would like to thank my parents and my wife for their constant support, patience, and love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A *multiprocessor* is a computer system containing multiple processors which are capable of communicating and cooperating at different levels in order to solve a given problem [HwBr84]. A *distributed-memory multiprocessor* (DMM) is a multiprocessor in which each memory module (not cache memory) is physically associated with each processor and an interprocessor communication network provides a mechanism for communication between processors. Centered around the development of efficient parallel algorithms on DMMs, this thesis focuses on the design, implementation, and modeling of a special class of parallel algorithms, called *pipelined data parallel algorithms*, on DMMs.

DMMs are different from *shared-memory multiprocessors* (SMMs), in which all memory modules are equally accessible to all processors through a processor-memory interconnection network. Both architectures have commercial implementations [Kwan87]. A brief review and comparison of SMMs and DMMs are presented in Section 1.1. This discussion serves to motivate our study of DMMs. A model of DMMs is given in Section 1.2. Characteristics of DMMs are discussed in more detail, which form the basis for the following discussion. As will be seen in Section 1.2, the major bottleneck in current DMMs is communication. The problems caused by the non-negligible communication overhead are discussed, and, from a user's point of view, the best way to get around with the communication problem is to carefully design the

algorithm. The work presented in this thesis is devoted entirely to achieving this end.

Pipelined data parallel algorithms are a class of parallel algorithms which uses pipelining and data level partitioning to achieve a very high degree of parallelism. Special features of pipelined data parallel algorithms make them very suitable for execution on DMMs. As a result, pipelined data parallel algorithms are the primary subject of this thesis. In Section 1.3, the basic concept of pipelined data parallel algorithms is introduced by contrasting them with other styles of parallel computations.

Finally, a formal statement of the problems addressed in this thesis is given in Section 1.4, followed by an overview of the thesis in Section 1.5.

1.1. SHARED VERSUS DISTRIBUTED MEMORY MULTIPROCESSORS

The multiprocessors discussed in this thesis are MIMD machines, in which multiple instruction streams simultaneously operate on multiple data streams. As previously mentioned, to provide such a concurrent processing environment, SMMs treat the memory as a shared resource and use a processor-memory interconnection network to allow the memory to be accessed equally from any processor. The simplest such interconnection network is a single high speed bus to which multiple processors, memory modules, network interfaces, and device controllers are all tied. Most minisupercomputers, such as Alliant's FX series, Encore's Multimax, or Sequent's Balance and Symmetry series [Hwan87], choose this kind of connection. The advantage is, of course, the simple architecture which performs very well for up to a few tens of processors. However, as the number of processors increases, the contention to access the bus increases, which, in turn, increases memory latency. This limits the scalability and expandability of bus-structured SMMs.

At the other end of the spectrum is the SMMs with a crossbar interconnection network. The crossbar provides a one-to-one link from any processor to any memory

module. The only contention that may occur is when two processors attempt to access the same memory module. The major disadvantage of the crossbar is the high cost in building the network, which, again, limits the scalability of the SMM. Another *direct connect* architecture [Hwan87] is the multistage interconnection network, which is a compromise between the bus and the crossbar. One of the problems with the multistage network is the difficulty in achieving cache coherence and in synchronizing parallel activities.

Popular processor interconnection networks for DMMs include multistage and point-to-point networks. For example, BBN's Butterfly uses a butterfly multistage network to allow processors to access each other's local memories [BBN87]. Among point-to-point interprocessor connections, the hypercube is the most popular topology, as is found in NCUBE's NCUBE/10, Intel's iPSC, and AMTEK's S-14 [Hwan87, HaMS86]. The reason for this is because the hypercube topology has a uniform structure with $\log_2 N$ diameter, where N is the number of nodes in the hypercube, and a rich set of other topologies, such as ring and mesh can be embedded in it [SaSh85].

One of the most significant advantages of DMMs is their modularity and scalability. For example, in the hypercube, when the number of processors doubles (from N to $2N$), the diameter of the network only increases by one, and the number of neighbors a processor has to connect to is also increased by one. Currently, DMMs with a few hundred processors are common. Thus, DMMs hold the promising potential for massive parallelism. It is for this reason DMMs are studied in this thesis.

However, DMMs are not without problems. The main problem is due to the need for communication between processors to exchange data held in each other's local memories. In general, there are two possible synchronization and communication methods for DMMs: *shared-variable* and *message-passing*. In the shared-variable approach, processes communicate by sharing common variables. Though the shared-variable approach provides a unified and friendly environment for users, it takes a lot of

effort to support the logic view of a shared, global memory on top of distributed, local memories. Very high speed communication is necessary to close the performance gap between accessing local and remote memory modules.

In the message-passing approach, processes communicate by passing messages explicitly. Message-passing is much simpler to implement on DMMs, because it only uses simple primitives such as send and receive. More importantly, to achieve real massive parallelism, it is necessary to connect hundreds or even thousands of processors together. For such a complex system, message-passing is a natural choice for interprocess(or) communication. Nevertheless, message-passing requires a completely different style of algorithm design and programming than that of the conventional shared-variable approach. Therefore, in this thesis, we will concentrate on developing efficient parallel algorithms on message-passing DMMs.

1.2. A MODEL OF DISTRIBUTED-MEMORY MULTIPROCESSORS

Figure 1.1 is the block diagram of a typical DMM. A set of node processors are interconnected together through a point-to-point network, and each can communicate with a frontend host. The words *processors* and *nodes* will be used interchangeably in the following discussion.

(1) Node structure:

Nodes have homogeneous structures and run in an asynchronous fashion. They communicate with each other by passing messages. Each node consists of a computation unit, a communication unit, and a local memory. The computation is usually performed by the node CPU with some floating-point accelerators. Optional vector processors may be used, such as in Intel's iPSC-VX [Hwan87].

The communication unit is in charge of the communication with the host and other nodes. The task of communication may be performed by the CPU or by dedicated

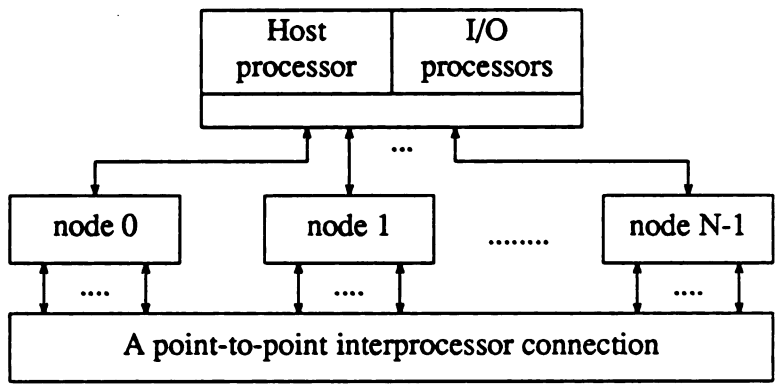


Figure 1.1. The block diagram of a typical DMM

processors through DMA (Direct Memory Access). For example, in many second generation DMMs, nodes contain intelligent routers dedicated to communication [ShFi87]. These routers not only reduce the time for initiating and receiving a message, but also reduce the time in relaying messages. More about communication will be discussed shortly.

Local memory in each node contains space for node operating system, user processes and communication buffers. For many systems, the size of local memory is still a limiting factor. For example, each node in the NCUBE has only 512 Kbytes local memory. In early DMMs, virtual memory is not available at the nodes, because there is no direct access from the nodes to any system peripherals other than through the host. The situation is changing in the second generation DMMs, in which special I/O subsystems are incorporated into the nodes to allow direct access to I/O devices [WiCM88].

(2) Host:

One or more frontend hosts are connected to the nodes to manage the whole system. Popular host to node interconnection topology include one-to-one direct connection, as in NCUBE's NCUBE/10, or bus connection, as in Intel's iPSC/1. In the first

generation DMMs, all peripherals, such as disks, tape drivers, terminals, and network controllers, are all connected to the host. Therefore, hosts provide the primary user interface and perform such functions as program development, performance monitoring, program and data downloading, file system maintenance, and program debugging. With new concurrent I/O subsystems, the role of hosts in future DMMs may be changed or replaced by the nodes.

(3) Node interconnection network:

Most DDMs use a static point-to-point network for node interconnection, such as a hypercube, mesh, tree, or ring [Hwan87]. In earlier implementations, messages are routed through the network using the store-and-forward approach. That is, a message or a packet of a message is stored completely at each intermediate node before it is sent out to the next node. This not only introduces a very high communication delay, which is proportional to the distance of the communication path, but also increases the degree of interference with the intermediate nodes, e.g., the need to allocate communication buffers in these nodes.

In the second generation DMMs, intelligent routers utilizing new techniques, such as *wormhole* routing [DaSe87], are used to reduce the overhead in routing messages. In wormhole routing, as soon as a node examines the header of a message, it selects the next channel and begins forwarding the message down the channel. Thus, only a few control bits are buffered at each intermediate node. As a result, less interference is experienced in the intermediate nodes and message latency remains almost constant, regardless of the distance between the source and destination. The implication is that the network becomes virtually fully connected [ShFi87].

(4) System software:

In a message-passing DMM, the only means of communication and synchronization at the system level, just as at the machine level, is by passing messages. There are no shared variables nor centralized sequence control. One node is dedicated to one job

at a time. A user must gain exclusive access to a set of nodes before the user can use the nodes. Again, this situation will be changed in future generation DMMs. Conventional programming languages are supported with enhanced library routines to handle messages and node management. A more friendly and advance programming environment is needed, which provides intelligent compilers and debuggers for DMMs.

As mentioned earlier, new router designs reduce the latency in routing messages. However, there is still a non-negligible overhead in setting up and receiving a message. Most overhead comes from the system software when invoking send/receive routines and maintaining message buffers. Thus, even though DMA channels and routers can reduce the communication overhead, it is still up to the software — both operating system and algorithm — to minimize the overall communication delay.

From the above discussion, one can see that DMMs achieve modularity and extendibility by minimizing the coupling between nodes. Nevertheless, this also introduces a non-negligible cost of communication. Table 1.1 lists performance figures for three representative first generation DMMs [Duni87].

Table 1.1. Performance figures for three DMMs

	AMETEK S-14	Intel iPSC	NCUBE
8-byte transfer time (μs)	640.0	1120.0	470.0
8-byte multiply time (μs)	33.9	43.0	14.7
Communication/Computation	19	26	32

It can be seen in Table 1.1 that transferring a datum (a 64-bit float-point number) will take a much longer time than will performing a computation. This imbalance is expected to be improved in the second generation DMMs through streamlined communication primitives and intelligent routers. However, as discussed, the distributed nature of DMMs introduces a non-uniform access time in referencing local and remote

memories. Therefore, it is essential not only to minimize the number of messages exchanged in the algorithm, but also to choose an appropriate message size in order to balance the computation with communication.

In general, there are two possible sources of communication bottlenecks in a DMM. One is the *communication bandwidth* between the nodes, which has been discussed above. Another bottleneck is the *I/O bandwidth* between the nodes and external world, e.g., disks. For example, in the current generation DMMs, only the host has access to disks. Thus, at initialization and summing-up stages of the computation, there is the need for the host to download data to the nodes and nodes to upload data back to the host, perhaps sequentially. It follows that the host and host-to-node interconnection (or I/O bandwidth) also tend to be system bottlenecks.

Putting all these considerations together, it can be seen that, from the algorithm designer's point of view, the best way to minimize the communication effect is to carefully partition the algorithm and schedule the operations — a topic which will be further discussed.

1.3. STYLES OF PARALLEL COMPUTATIONS

In general, a parallel computation can be classified from two perspectives: one is according to the way the computation is partitioned and distributed, and the other is based on the way the computation is executed. From the first perspective, we can distinguish between computations that are *data parallel* or *function parallel* [Oste87]. A function parallel computation decomposes the program into segments. Different processors execute different segments in parallel. Function parallelism is suitable for applications that can be implemented using programs with many unique subroutines, e.g., flight simulation.

A data parallel computation divides data among the processors. Processors may be running the same program but working on different subsets of data. Algorithms using this technique are called *data parallel algorithms* [HiSt86]. Data parallelism is appropriate for applications that perform the same operations repeatedly and independently on a large set of data. In terms of programming, programs with loops to handle static and regular data structures are suitable for data parallel decomposition. Typical applications of data parallel algorithms include circuit simulation, network simulation, matrix operations, signal processing, and ray tracing.

From the perspective of scheduling, a parallel computation can be characterized to be either *concurrent* or *pipelined* [HwBr84, Squi86]. Concurrency exploits spatial parallelism by utilizing several processors to execute multiple independent tasks simultaneously. Tasks may be data parallel or function parallel.

Pipelining exploits temporal parallelism in which each processor (called a *stage*) behaves like a filter or transformer which operates on its input data and passes output data to the succeeding processor. Through pipelining, communication occurs only between fixed and neighboring stages; and a datum, once entering the system, will be used or modified repetitively along the pipeline. Thus, a very high ratio of computation to I/O rate will result.

One of the most important characteristics of pipelined computation is the notion of data flows — data flowing in the pipeline as they are processed stage by stage. To see if a computation contains data flows, the following flow test suggested in [NeSn87] seems to be very helpful:

Flow test:

Select an arbitrary edge (not necessarily at the boundary) and "radioactively tag" a single transmission across that edge. As the computation progresses, let all values computed with one or more radioactive values become radioactive. Then, define the "contaminated region" as the set of processors and

edges touched by some radioactive values.

An algorithm passes the flow test if the edge over which the initial value was transmitted is on the boundary of the contaminated region. Note that the flow test is not perfect in the sense that there are algorithms generally called pipelined or systolic which fail the test, e.g., the systolic array for matrix multiplication proposed in [KuLe78]. However, it does capture the essence of the flow concept and serves well in most cases.

From the above discussion, we can see that there are four different styles of parallel computations, as shown in Table 1.2. In the remainder of this section, examples will be used to further explain the basic idea of each style.

Table 1.2. Different styles of parallel computation

		Partition	
		Function	Data
Scheduling	Concurrent	Concurrent function parallel	Concurrent data parallel
	Pipelined	Pipelined function parallel	Pipelined data parallel

(1) Concurrent function parallel computation

In this style of computation, efforts are devoted to identifying concurrently executable operations or program modules. Consider the following program segment:

$$\begin{aligned} &l := 10; \\ &\textbf{for } i := 1 \textbf{ to } k \textbf{ do} \\ &\quad d_i := l \times a_{i-1} + b_i \times c_{i+1} \ ; \end{aligned} \tag{1.1}$$

Note that in the program segment above, $l \times a_{i-1}$ is independent of $b_i \times c_{i+1}$. Therefore, these two operations can be executed in parallel, as shown in Figure 1.2. Here, two processors are used, and it is easy to see that workloads are not always balanced. Note that concurrent function parallel computation can also be applied to other program

constructs, such as a program statement, a function block, or a subroutine. Most scheduling and load balancing algorithms concentrate on scheduling these program modules to achieve concurrent function parallelism [Gonz77].

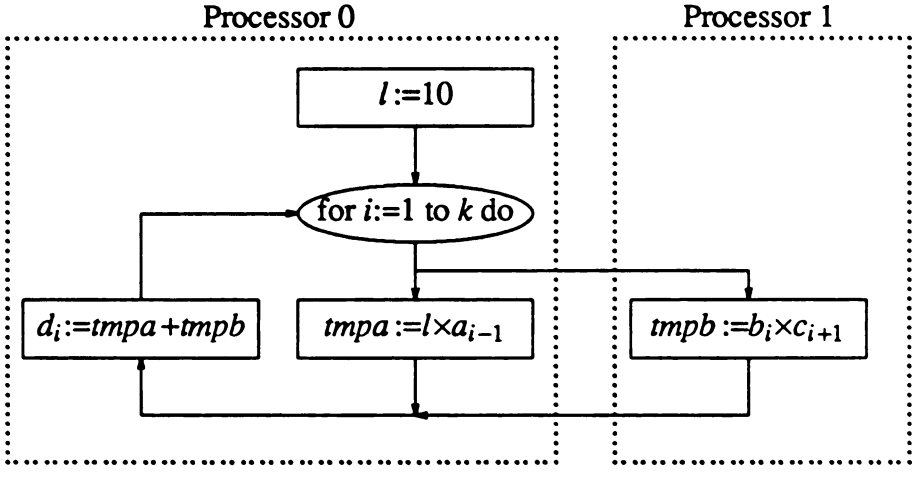


Figure 1.2. A concurrent function parallel computation

(2) Concurrent data parallel computation

If data parallelism is used in decomposing the program segment (1.1), we will obtain an execution scheme similar to that in Figure 1.3. Since there is no data dependency between loops, iterations in (1.1) can be executed on multiple processors in parallel — processors perform the same task but use different data.

A DMM can be used to execute the computation shown in Figure 1.3, in which all array elements are downloaded into corresponding processors beforehand, and then processor 0 (or the host) broadcasts the value of l (10 in (1.1)) to other processors. Note that the number of processors is not necessarily the same as the loop count. We can use n ($<k$) processors and allocate $\lfloor k/n \rfloor$ loops to each processor. Remaining loops are allocated to the first few processors, one for each.

The computation can also be executed on a SMM. The variable l now becomes a shared variable. Again, fewer than k processors can be used. In this case, k tasks are

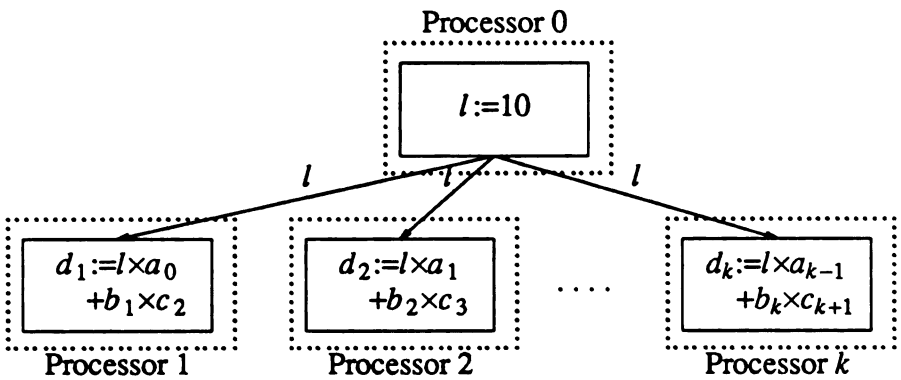


Figure 1.3. A concurrent data parallel computation

created, one for each loop, and are put into a linear list. Each processor fetches the first task in front of the list, executes the loop, and then gets the next task. In this way, load balancing is automatically enforced — no one will be idle if there are still tasks to do.

(3) Pipelined function parallel computation

Traditionally, the term "pipelining" refers to this style of computation. Different stages in the pipeline perform different functions, and, when data flows through the stages, they are modified along the way. An example of pipelined function parallel computation is the parallel compiler shown in Figure 1.4 [Quin87]. Each individual phase of the compiler is assigned to one processor. Other examples include pipelined vector processing [HwBr84], image processing, and speech recognition.

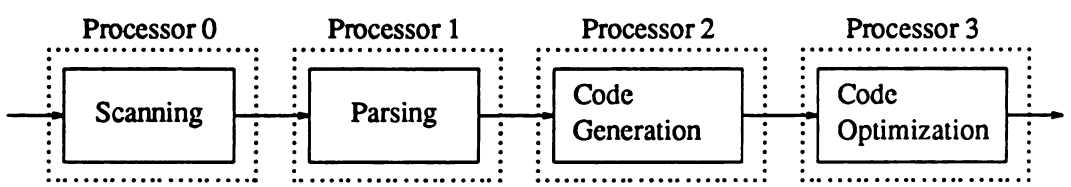


Figure 1.4. A pipelined function parallel computation

The primary consideration in pipelined function parallel computation is to keep the processing speeds of all stages roughly equal. Otherwise the slowest stage will become the bottleneck of the pipeline. Again, this involves equally partitioning the functions among stages.

(4) Pipelined data parallel computation

Systolic arrays are a typical example of pipelined data parallel computation [FoWa87, KuLe78]. Processors in the pipeline perform the same function in a synchronous fashion. To carry out the computation, each processor may take input operands from different data streams flowing in different directions and generate outputs to several other streams. A pipeline in this kind of computation can serve as a transformer, as is exemplified by the following program segment:

```
s := 0;
for i := 1 to k do                                    (1.2)
    s := s + a_i;
```

This program segment can be executed as shown in Figure 1.5, in which each processor accumulates on top of the received s its local value and sends the result (a transformed s) to the next processor. At first thought, this linear reduction scheme is inferior to a binary-tree reduction scheme, because the latter takes only $\log_2 k$ steps. However, from the view point of each individual processor, there is only one addition should be executed in the reduction if the linear scheme is used. This implies that, after one addition, the processor can "forget" and go ahead to perform the next operation.

A pipeline may serve another purpose: data dissemination. Take the program segment (1.1) as an example. Instead of broadcasting the value of l, processor 0 can also send l to processor 1. Processor 1 copies the value, sends l to processor 2 immediately, and, then, executes the loop. In this way, l is "piped" through the pipeline formed by processor 1 to k (see Figure 1.6). If several l values have to be disseminated among the processors, then these data form a data stream. It follows that the communication bandwidth (or I/O bandwidth if processor 0 is the host) between processor 0 and the

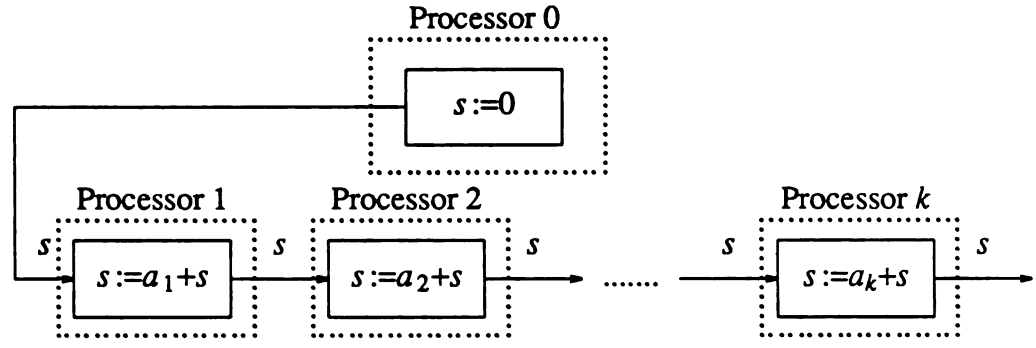


Figure 1.5. A pipelined data parallel computation

remaining processors is reduced.

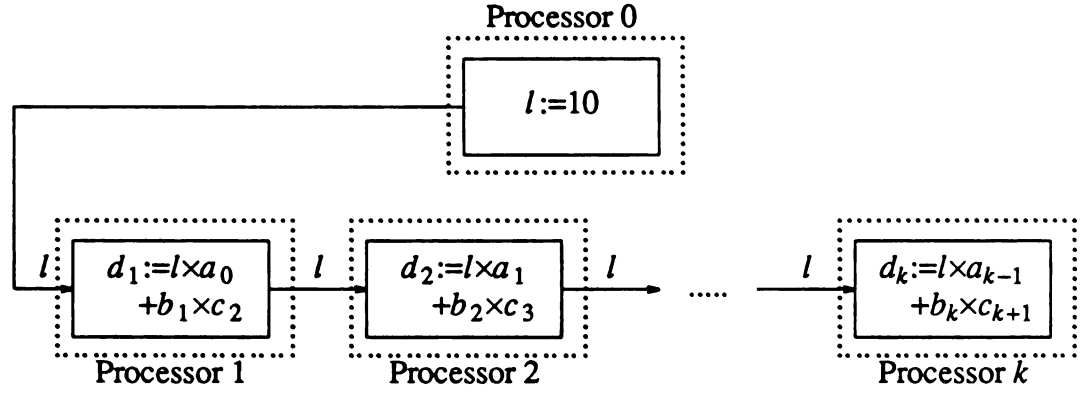


Figure 1.6. Another pipelined data parallel computation

So far, we have discussed the four different styles of parallel computations listed in Table 1.2. Different applications may prefer a different style of computation. However, for most parallel algorithms, a mixture of two or more styles is common. As discussed, pipelined computations allow a very high computation to communication ratio, while data parallelism is more suitable for applications with regular computation patterns and is easier to achieve load balancing. Combining these two factors together,

pipelined data parallel computation is a very suitable style of computation for DMMs. Yet, there still lacks of an extensive and systematic study of pipelined data parallel algorithms on DMMs. Such a study is the major topic of this thesis.

1.4. PROBLEM STATEMENT

As indicated in the beginning of this chapter, our major concern in this thesis is designing efficient parallel algorithms for DMMs. Due to the distributed nature and communication overhead on DMMs, the designer must address the following issues:

- How can communication be balanced with computation?
- How can workloads be balanced and operations be overlapped?
- How can time and space be balanced?

To arrive at a balanced design, the following application specific questions have to be considered:

- How can the problem or data be partitioned?
- How can the partitions be mapped onto the processors?
- What is the granularity of the partitions?
- What is the operation sequence within each processor?

A design methodology or paradigm is needed. Through paradigms, it is possible to answer the above questions and design efficient parallel algorithms in a systematic way. The particular paradigm studied in this thesis is called *large-grain pipelining*, which uses a pipelined data parallel style of computation.

Given a pipelined data parallel algorithm, an analytic model is required, which serves the following two purposes:

- Estimate and predict the performance of the given algorithm;
- Determine optimal design parameters such as the partition size.

To develop an accurate model, one must take into account the characteristics of the underlying architecture as well as the logic behavior of the algorithm. The accuracy of the model will determine how close the estimated design parameters are to the optimal ones. In this thesis an accurate analytic model for pipelined computation will be presented.

Perhaps as important as design paradigms is a systematic way of designing parallel algorithms. Using such a systematic design methodology, a designer can start the algorithm development from the problem definition and follow the procedure step by step until a satisfactory parallel algorithm is obtained. A systematic methodology lends itself to an automatic design tool which can be part of an intelligent compiler for DMMs. In this thesis, a systematic approach for designing pipelined data parallel algorithms on DMMs will be introduced.

1.5. THESIS OUTLINE

The rest of the thesis is organized as follows. Chapter 2 addresses the issues of designing efficient parallel algorithms on DMMs. Considerations for algorithm representation, partitioning, mapping, and scheduling are discussed. A variation of *Petri-nets*, called *P-nets*, will be described as an example of algorithm representation.

In Chapter 3, attention is turned to designing, implementing, and modeling data parallel algorithms. The algorithms are designed to execute on a 64-node NCUBE multiprocessor (a first generation hypercube DMM). Therefore, features of the NCUBE are introduced, followed by a general model of computation and communication on DMMs. This model provides a basis for analyzing the algorithms. Two case studies, array summation and matrix multiplication, are given to illustrate different considerations involved in the design. Performance analysis and comparison of different approaches are presented, which use both analytic results as well as experimental

results.

The basic concept of large-grain pipelining is introduced in Chapter 4. First, various parallel programming paradigms are reviewed. From these paradigms, a new paradigm which applies a pipelined data parallel style of computation on DMMs becomes necessary. The basic idea of large-grain pipelining is introduced through the example of matrix multiplication. Characteristics of pipelined data parallel algorithms are summarized, and experimental results of the pipelined matrix multiplication algorithms executed on the NCUBE are presented.

Chapter 5 covers the analytic model for pipelined data parallel algorithms. The model is introduced first, and important formula for characterizing design parameters are given. The model is then used to analyze the performance of the pipelined matrix multiplication described in Chapter 4. The analytic results are compared with experimental results from the NCUBE. The close match between these two kinds of results indicates the accuracy of the analytic model.

A systematic procedure is described in Chapter 6 to design pipelined data parallel algorithms on DMMs. Starting from a nested-loop program, the procedure restructures the program in various steps until one that is suitable for DMMs is obtained. The most important step in this procedure is to group operations and data elements. Through grouping, the granularity of the algorithm can be controlled, while an appropriate granularity will balance the computation with communication and achieve the a high degree of overlapping.

Finally, conclusions are presented in Chapter 7. Improvements of the work presented here and the impact on second generation DMMs are discussed. Plans for future work is given. The experience gained from this thesis research is also summarized.

CHAPTER 2

CONSIDERATIONS FOR DESIGNING PARALLEL ALGORITHMS

Designing parallel algorithms is much different from designing conventional serial algorithms. For one thing, the designer has to figure out an efficient way of utilizing the available processors. This would require the distribution of workloads evenly among the processors so that all processors keep busy doing useful computation at all times. On DMMs, this requires a balance between computation and communication as well as between time and space. Moreover, the designer has to insure the correctness of the parallel algorithm. The latter would require a proper synchronization and ordering of events between the processors.

To achieve the above goals, one must first identify and be able to express the parallelism inherent in the problem. A suitable representation is necessary to define the data sets and operations used in the algorithm and unveil the concurrency and dependency existing among them. Next, the computation is parallelized according to available concurrency. On DMMs, parallelization involves partitioning the problem into many subproblems, allocating the subproblems to the processors, and scheduling the execution sequence within each processor. The underlying assumption is that the necessary operations to solve the problem are known before the execution, and thus deterministic scheduling is possible [Gonz77]. Most data parallel algorithms possess this

property. In this chapter, we shall discuss these points — representation, partitioning, mapping, and scheduling — in more detail.

The rest of the chapter is organized as follows. A representation of parallelism, called *P-nets*, is presented in Section 2.1. Based on *Timed Petri-nets* [Ramc74], P-nets can express and model general computation, especially the data parallel style of computation. Based on the P-net representation, considerations for algorithm partitioning and mapping will be studied in Section 2.2, followed by discussions for algorithm scheduling in Section 2.3.

2.1. A REPRESENTATION OF PARALLELISM — P-NETS

A good algorithm representation should unveil the dependency and concurrency inherent in the problem and carry useful information, such as time and space, for design decisions and performance analysis. In this section a graph-theoretical representation, called *P-nets*, is presented. The formal definition of P-nets will be presented in Section 2.1.1. Section 2.1.2 will discuss variations of P-nets and show how to analyze P-nets.

2.1.1. Basic Definition of P-nets

A P-net is a directed bipartite graph consisting of two types of vertices: *data vertices* (U) representing data components and *operation vertices* (V) representing operations. If a data vertex u_i is an input operand to an operation v_j, then there is an arc (u_i,v_j) in the P-net. Similarly, if an operation v_i generates a data vertex u_j, then there exists an arc (v_i,u_j) in the P-net. Data vertices which have no preceding (succeeding) vertices are called *initial (resultant) data vertices* and are denoted by U_i (U_r).

Each data vertex u_i in the P-net is associated with a parameter $f_u(u_i)$ to indicate the size of the data component represented by u_i. Each operation vertex v_i is associated with a parameter $f_v(v_i)$ to denote the amount of time to perform v_i. Thus, the static

structure of a P-net can be defined by the five-tuple $G=(U,V,E,f_u,f_v)$, where:

U is a finite nonempty set of *data vertices*,

V is a finite nonempty set of *operation vertices*,

$E \subseteq U \times V \cup V \times U$, is the set of arcs between data vertices and operation vertices,

$f_u : U \rightarrow R^+$, is the *data size function*,

$f_v : V \rightarrow R^+$, is the *operation time function*,

and R^+ is the set of positive real numbers.

<Example 2.1>

Consider the multiplication of two matrices A and B, $A \times B = C$. Figure 2.1(a) describes the task, where circles represent data vertices (the matrices) and squares represent operation vertices (the multiplication). One way of performing the multiplication is to partition A and B into four submatrices as follows:

$$A \times B = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = C$$

where $C_{ij} = C_{ij}^{(0)} + C_{ij}^{(1)} = A_{i0}B_{0j} + A_{i1}B_{1j}$, $0 \le i,j \le 1$. The P-net shown in Figure 2.1(b) defines the necessary operations to calculate the matrix product in this case. The set of initial data vertices is $U_i = \{A_{ij}, B_{ij} \mid 0 \le i,j \le 1\}$, and the set of resultant data vertices is $U_r = \{C_{ij} \mid 0 \le i,j \le 1\}$. Suppose that both matrices A and B have a size 64×64, and that scalar multiplications and additions both take 1 unit of time. Then, each data vertex represents a 32×32 submatrix and has a size $f_u = 1024$. Also, $f_v = 65{,}536$ for each multiplication vertex and $f_v = 1024$ for each addition vertex.

□

The dynamic behavior of a P-net is described by token markings. A *token* in a data vertex u_i indicates the existence of a data element, which occupies a memory space of size $f_u(u_i)$. Data vertices can be viewed as FIFO queues for tokens. A token has two states: *busy* and *ready*. A token is busy when it is first created and its data elements are being generated by the preceding operation vertices. A token becomes ready when its

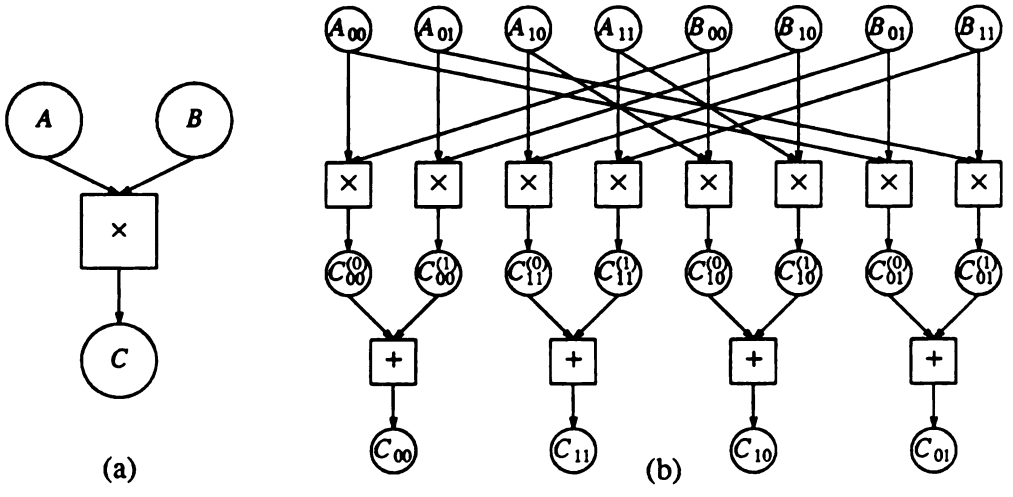


Figure 2.1. Matrix multiplication and its corresponding P-net

data elements are ready for use by the succeeding operation vertices. Also associated with a token is a variable *tag*, which is initialized to the out-degree of the corresponding data vertex and is decremented by 1 whenever one succeeding operation finishes referencing the token. Basic firing rules for an operation vertex of a P-net are as follows:

(1) An operation is *enabled* if and only if each of its input data vertices holds at least one ready token.

(2) An operation can *fire* if it is enabled.

(3) When an operation v_i fires, one busy token is deposited in each of the output data vertices, with the tag initialized accordingly.

(4) The operation will continue firing for $f_v(v_i)$ units of time.

(5) At the end of the firing, the busy tokens in all output vertices become ready and the tags of the ready tokens in all input vertices are decremented by 1. If a tag is reduced to 0, the corresponding ready token is removed.

Figure 2.2 illustrates the executions of the P-net shown in Figure 2.1 at various instants of time, assuming maximal parallelism. It can be seen from Figure 2.2 that the total job execution time is 66,560 and the maximum memory space required is 16,384. Note that for a pure sequential system (e.g., a uniprocessor system) the computation time becomes 528,384, which is simply the sum of all the f_v values.

The execution of a P-net on a DMM can be characterized from two perspectives: one is time (when should an operation vertex be fired?) and the other is space (where should a data vertex reside?) A P-net can be modified to incorporate the effect of partitioning and communication on DMMs. Consider the following example:

<Example 2.2>

Suppose the matrix multiplication in Example 2.1 is to be scheduled on a DMM with three processors. A possible allocation of the P-net among these three processors is shown in Figure 2.3. The effect of communication delay is represented by introducing new operation vertices, called *communication vertices* and denoted by v_c. The buffers for received data in receiving processors are represented by the *buffer vertices*, which are the outputs of communication vertices. The value of $f_v(v_c)$ is the time to transmit the corresponding data elements from the source processor to the destination processor. Note that given a P-net and a mapping of the vertices to processors, it is straightforward to incorporate communication delay and derive the P-net shown in Figure 2.3.

□

In summary, data generations and consumptions in a P-net are represented by operation firings and token movements. Thus, data vertices in a P-net serve as templates of data — a token in a data vertex denotes the existence of a data element. Communication and buffer vertices are introduced to represent communication and partitioning. Space and time parameters are incorporated into the model, which allows the trace of system status and resource requirements. Thus, P-nets are more suitable for algorithm development and performance analysis on DMMs.

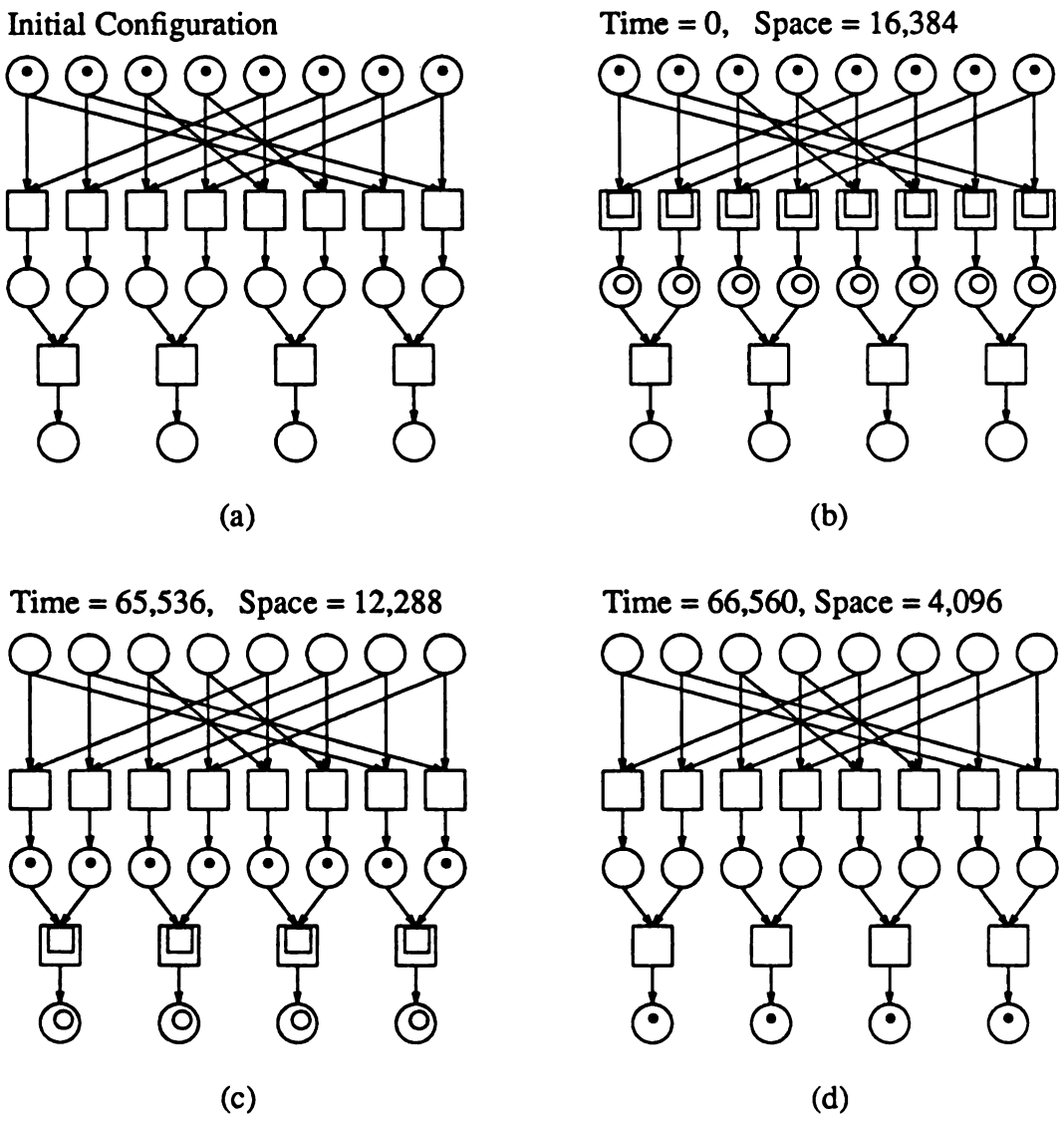


Figure 2.2. Execution of a P-net at various time instants

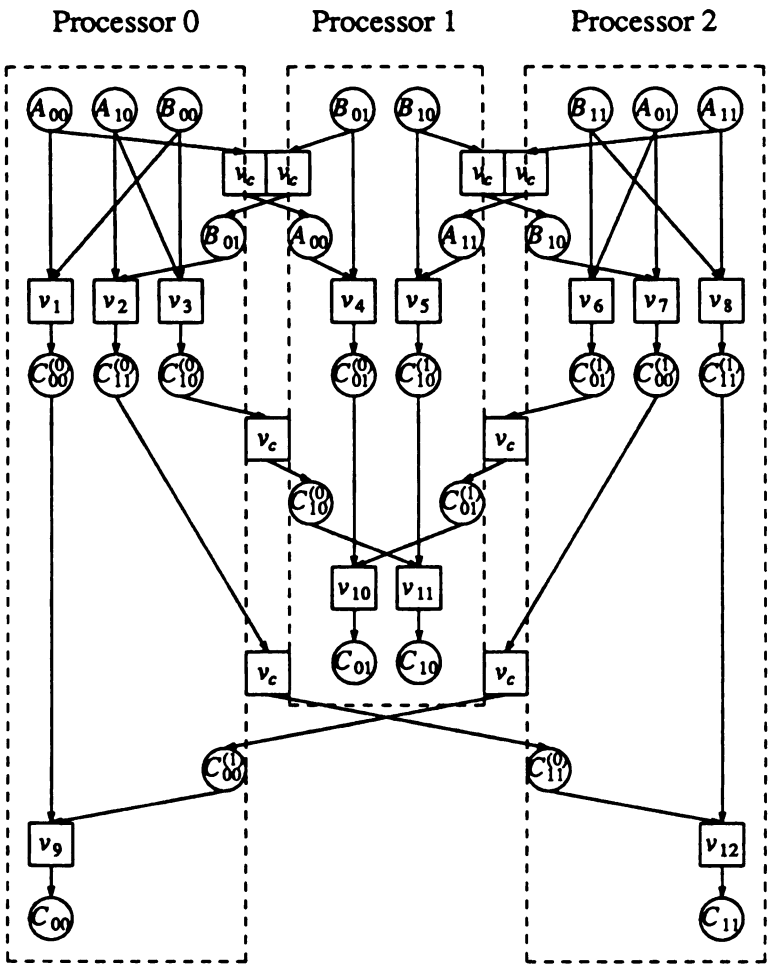


Figure 2.3. Partitioning a P-net among three processors

Note that basic Petri nets [Pete77] are a special case of P-nets, in which $f_v=f_u=0$ for all vertices. This implies that P-nets have at least the same modeling and computing power as Petri nets.

2.1.2. Variations and Analysis of P-nets

Based on the characteristics of data parallel algorithms, we can simplify the discussion by considering a subset of P-nets, which has the following properties:

(1) *acyclic*, i.e., the P-net does not contain a loop,

(2) *decision-free*, i.e., every operation vertex has only one output data vertex,

(3) *safe*, i.e., any data vertex can contain at most one token at any time, and

(4) *functional*, i.e., every data vertex is generated by at most one operation vertex.

From these properties, a much simpler representation can be derived, by noting that there is a one-to-one correspondence between a data vertex and an operation vertex. Thus, corresponding to each such P-net, $G = (U,V,E,f_u,f_v)$, a simplified graph, called a *D-net*, can be defined. A D-net is a four-tuple, $D_G = (U,E',f_u,f'_u)$, where

$(u_i,u_j) \in E'$, if and only if there exists a $v \in V$ such that $(u_i,v) \in E$ and $(v,u_j) \in$ E

$f'_u(u_j) = f_v(v)$, if $(v,u_j) \in E$

That is, $f'_u(u_j)$ is the time to generate the data vertex u_j. Figure 2.4 depicts the corresponding D-net for the P-net shown in Figure 2.1.

If there is only one set of input data to be processed, i.e., there is no pipelining, then each data vertex will be fired only once and will contain at most one token. This case may occur if, for example, we consider only one iteration of a loop program or unfold the whole loop. It follows that, the execution of a D-net on a DMM can be characterized by two state functions. The state function $T_u(u_i)$ denotes the time that data vertex u_i becomes ready, and the function $S_u(u_i)$ denotes the processor that u_i is to be generated. Given these two functions, other state information can be deduced. For

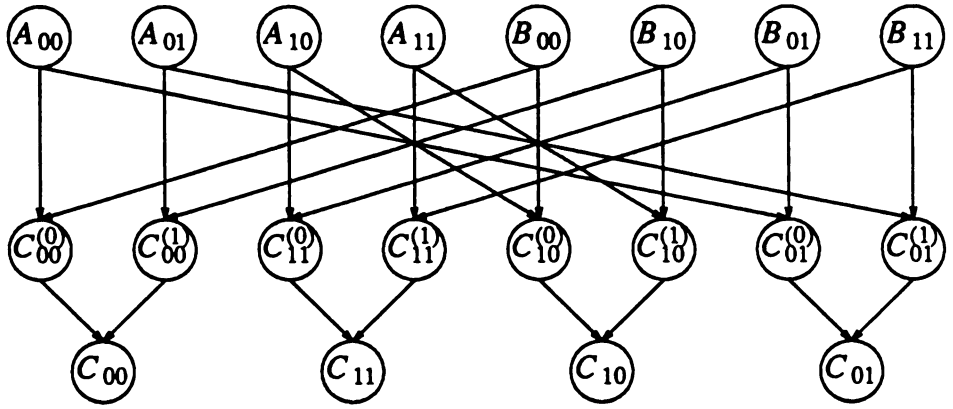


Figure 2.4. The D-net corresponds to the P-net in Figure 2.1

example, the total job execution time, T, is given by

$$T = \max_{u_i \in U_r} T_u(u_i) \tag{2.1}$$

Similarly, the time a data vertex u_i becomes busy is given by

$$T_{u1}(u_i) = T_u(u_i) - f'_u(u_i)$$

and the time u_i can release its token is given by

$$T_{u2}(u_i) = \max_{(u_i,u_j)\in E'} T_u(u_j)$$

It follows that the memory required for the execution is

$$S = \max_{0\le t\le T} \sum_{u_i \in U} \kappa(u_i,t) f_u(u_i) \tag{2.2}$$

where

$$\kappa(u_i,t) = \begin{cases} 1 & \text{if } T_{u1}(u_i) \le t \le T_{u2}(u_i) \\ 0 & \text{otherwise} \end{cases}$$

2.2. CONSIDERATIONS FOR PARTITIONING AND MAPPING

Scheduling in a multiprocessor involves the assignment of operations or data elements to particular processors for execution at particular instants. The multiprocessor scheduling problem has been extensively studied for over 20 years [Gonz77]. The *list scheduling* or *critical path scheduling* algorithm [AhCh74, Hu61, PoBa87] is the most well known algorithm. Unfortunately, this kind of scheduling algorithms does not consider, and is difficult to incorporate into, communication delay and, thus, is mainly applied on SMMs.

On the other hand, scheduling algorithms for DMMs emphasize minimizing inter-processor communication for a given problem. In general, the problem is solved by the min-cut algorithm [RaSH79] or is formulated as variations of the quadratic assignment problem [MaLT82]. One drawback of these scheduling methods is the ignorance of data dependence relationships. This may result in processor idleness due to synchronization.

To take into account both communication delay and data dependency, the scheduling problem should be studied at two levels — *inter-processor* and *intra-processor*. Inter-processor scheduling is concerned with the partitioning and mapping of a given algorithm onto a multiprocessor. Given a mapping, intra-processor scheduling is concerned with the execution sequencing within each processor. Scheduling decisions at one level will affect those made at the other level.

In this section, general considerations for partitioning and mapping algorithms are studied first in Section 2.2.1 and 2.2.2, respectively. Then we will concentrate on partitioning and mapping methods for algorithms represented by P-nets in Section 2.2.3.

2.2.1. Number of Nodes and Partitions

Given an application and the associated D-net, the first problem facing an algorithm designer is the number of processors and the algorithm partitioning scheme that

should be used. The primary goal of algorithm partitioning is to partition the algorithm (i.e., the corresponding D-net) so that processors have an equal amount of the load to work with. The "load" of a processor should include not only the computation time of the assigned task but also the communication time with other processors and the idle time due to synchronization (see Example 2.1). Therefore, if the communication takes more time than the computation, the communication delay between two communicating vertices residing in two different processors will negate the gain due to parallelism. In this case, we should use fewer processors and group several related vertices together to increase the computation to communication ratio. We refer to those vertices assigned to the same processor as a *group* or *partition*. It follows that a good partition scheme, in addition to distributing workloads, should also minimize the communication traffic between groups and match the host capacity.

At the end of the partitioning phase, an application *contracted graph* is obtained. A contracted graph, $M=(V_M,E_M)$, has $k=|V_M|$ vertices, where $k\leq|V_H|$. Each vertex corresponds to one group, and there is an edge between two vertices in M if communication exists between their corresponding groups. The partitioning phase is sometimes referred to as *contraction* [Berm87].

2.2.2. Mapping Considerations

The multiprocessor on which the algorithm is to be executed can be represented by an undirected graph, called the *host graph*, which is denoted by $H=(V_H,E_H)$. Processor i in the multiprocessor is represented by a vertex $v_i\in V_H$, and a communication link between processor i and processor j is represented as an edge denoted by $(v_i,v_j)\in E_H$.

Based on the contracted graph, M, of an algorithm, a suitable processor interconnection pattern is chosen and each group in M is assigned to a different processor in H. The mapping procedure is inconsequential if there is no inter-group communication in

the contracted graph, or the host graph is a complete-connected graph. However, if the communication cost is non-uniform across the system and is a function of the distance between two communicating processors, then one must insure that the communication takes place using the shortest path.

For machines, such as a hypercube multiprocessor which embeds a large set of topologies, the choice of a suitable interconnection pattern is relatively easier and more flexible. In general, mapping is a two phase operation. One phase, the *group mapping phase*, may be defined as $g(v_i)=v_j$, for all $v_i \in V_M$ and any $v_j \in V_H$, which is a one-to-one mapping between vertices in M and processors in the host graph H. Two types of paths in a mapping may be identified: a *logical path* which is an edge in the contracted graph and a *physical link* which is an edge in the host graph. The second phase of the mapping, the *path mapping phase*, is to define a function, h, to map the logical paths onto the physical links. The whole mapping function, f, is then defined as $f=g \circ h$.

The number of physical links that a logical path has to traverse defines the *dilation* of that logical path. The number of logical paths which traverse a particular physical link is referred to as the *sharing* of that link. Since a physical link allows one message to pass at a time, the higher the sharing is, the longer the communication delay will be. Also, the more links which a message must traverse, the greater the communication traffic will be. Thus, a good mapping function, $f=g \circ h$, should try to minimize the average dilation cost and sharing cost.

It can be shown that, for an arbitrary problem topology, both component partitioning and group mapping are NP-complete problems [GaJo79]. Therefore, a good heuristic approach should be used, such as simulated annealing [Berm87]. It is believed that the partitioning and mapping problem must take into account the characteristics of the underlying architecture as well as the behaviors of the algorithms running on such a system.

2.2.3. Inter-Processor Scheduling

As mentioned earlier, the loads of a processor include the computation, communication, and idle time of the processor. Partition and mapping utilize this timing information to distribute the workload evenly among the processors. However, processor idle time cannot be determined unless intra-processor execution sequence is determined. The intra-processor schedule, in turn, determines inter-processor synchronization. On the other hand, an inter-processor partition and mapping scheme must be decided before the intra-processor schedule can be determined.

One approach to break this dilemma is to perform partition and mapping without concern for the processor idle time. Then, given a D-net, (U,E',f_u,f_u'), the inter-processor scheduling problem can be formulated as a quadratic assignment problem as follows. Let d_{kl} denote the distance between processor k and l in H, and $q_k f_u'(v_i)$ be the computation cost of vertex v_i on processor k, where q_k is the speed factor of processor k. Define the *assignment matrix* X to be such that $x_{ik}=1$ if vertex v_i is mapped to processor k, and 0 otherwise. Then, the cost of an inter-processor schedule is

$$Cost(X) = \sum_k \sum_i (q_k f_u'(v_i) x_{ik} + \sum_{(v_i,v_j)\in E'} c f_u(v_i) d_{kl} x_{ik} x_{jl})$$

where c is the cost of transmitting one byte of data between two adjacent processors. Memory limitation requires that, for each processor k,

$$\sum_i f_u(v_i) x_{ik} \le M_k$$

where M_k is the maximum size of the available memory on processor k.

The above formulation can be solved by iterative improvement, which starting from an arbitrary allocation, improves the allocation iteratively through such techniques as that proposed in [KeLi70], simulated annealing [KiGV83], or neuron networks [HoTa85]. Using the iterative method, we may apply the inter- and intra-processor scheduling alternatively in each iteration to obtain a more accurate cost estimation. Note that, through intra-processor scheduling, it is easy to identify the critical path(s) of

the computation. New configurations which involve moving vertices on these critical paths have a better chance of reducing the total execution time of the algorithm. The problem then becomes that of identifying, on the critical paths, the vertex whose reallocation will reduce the cost the most, and determining where the candidate vertex should be moved to.

2.3. CONSIDERATIONS FOR SCHEDULING WITHIN PROCESSORS

In this section, issues for intra-processor scheduling are discussed. Given a D-net $D=(U,E',f_u,f'_u)$ and a partition $(U^{(1)},U^{(2)},...,U^{(k)})$, where $U^{(i)}\cap U^{(j)}=\varnothing$ for all $i\neq j$ and $U^{(1)}\cup U^{(2)}\cup...\cup U^{(k)}=U$, intra-processor scheduling attempts to develop an execution schedule, X, for all $u\in U$. The partition $U^{(i)}$, $1\leq i\leq k$, is assigned to processor i, where k is the number of processors used. Figure 2.5 shows a possible schedule for the D-net shown in Figure 2.4. Dashed arcs are added by the schedule to enforce the total ordering.

In the following discussion, we will not consider the scheduling for initial data vertices, U_i, and buffer vertices, U_b. Initial data vertices are assumed to have been loaded beforehand, while the execution time of a buffer vertex can follow that of the predecessor vertex, if remote data are assumed to be transferred to local nodes as soon as they are generated.

A lower bound for the total execution time T of an algorithm is given by:

$$T = \max_{U^{(i)}} \sum_{u\in U^{(i)}} f'_u(u)$$

However, due to data dependence relationships, a processor may idle waiting for data from other machines. Thus, the above lower bound may not be achievable. To obtain the optimal schedule, we must first determine the cost for a given schedule. Recall that the state of a D-net can be completely described by the state function T_u and S_u (see Section 2.1.2). S_u is given from the partitions (i.e., from an inter-processor scheduling),

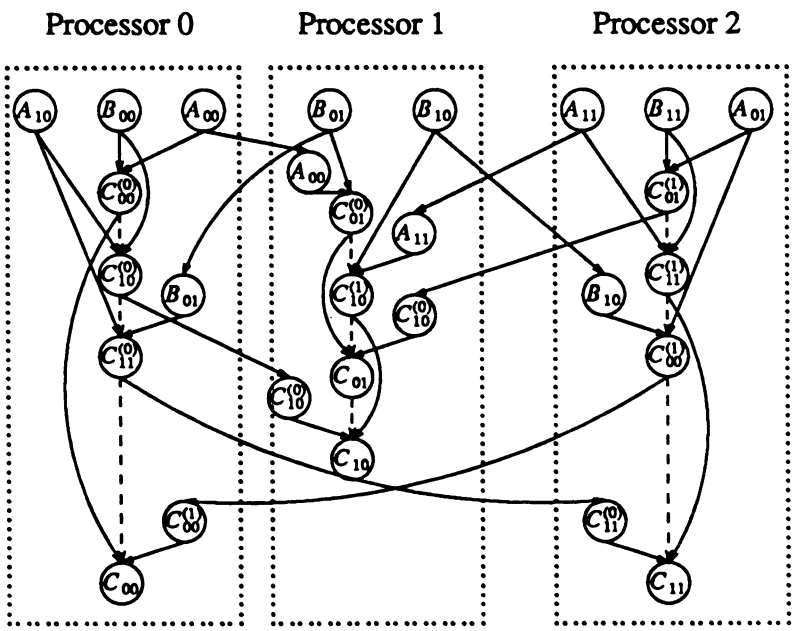


Figure 2.5. A schedule for the D-net in Figure 2.4

and it is easy to see that

$$T_u(u_i) = \begin{cases} f'_u(u_i) + \max\limits_{u \in R(u_i,X)} T_u(u) & \text{if } u_i \notin U_i \\ 0 & \text{if } u_i \in U_i \end{cases} \tag{2.3}$$

where, as defined in Section 2.1.2, $f'_u(u_i)$ is the time to compute u_i and $R(u_i,X)$ is the set of predecessor vertices of u_i in the schedule X. A vertex u_j is a predecessor vertex of another vertex u_i if either $(u_j,u_i) \in E'$ or u_i is generated immediately after u_j in the schedule. From T_u and Equations (2.1) and (2.2), one can obtain the total execution time of the schedule and find out the maximum memory needed for each partition. If a schedule's memory requirement does not exceed that of the maximum available, then the schedule is a feasible schedule.

One way to determine an optimal or near optimal intra-processor schedule for a given D-net is by means of the branch-and-bound method. The cost of a particular search path is expressed as the sum of the cost of the partial solution found so far and the cost from the present partial solution to the final solution (future cost). Possible candidates for expansion in a search path are those data vertices in the D-net that are enabled at that point. For intra-processor scheduling, a simple estimation for the future cost is

$$\max_{1 \le i \le k} \sum_{u \in U_x^{(i)}} f'_u(u)$$

where $U_x^{(i)}$ is the set of data vertices in partition $U^{(i)}$ that have not been scheduled so far. A particular search path can be pruned if it violates space or precedency constraints.

Though branch-and-bound is a powerful way to search for feasible solutions, its complexity is still exponentially proportional to the problem size. One heuristic adapted from critical-path scheduling [AhCh74, PoBa87] works by associating two *level* values with each vertex in a D-net. The *forward level* of vertex u, $\alpha_f(u)$, denotes the maximum "distance" from u to a resultant vertex and is defined as

$$\alpha_f(u) = \max_{P_f} \sum_{u_i \in P_f} f'_u(u_i)$$

where P_f is a path from the vertex u to a resultant data vertex. The *backward level* of a vertex u, $\alpha_b(u)$, denotes the maximum "distance" from u to an initial vertex and is defined similarly. An intra-processor schedule can thus be found by the following steps:

(1) For each partition $U^{(i)}$, $1 \le i \le k$, arrange all vertices in that partition in an nonincreasing order into a linear list L_i according to α_f first, then α_b.

(2) For each partition $U^{(i)}$, $1 \le i \le k$, check space requirement for L_i using Equation (2.1) and (2.2).

(3) For each partition $U^{(i)}$, $1 \le i \le k$, modify L_i if space constraint is violated.

(4) Report the resultant L_i's as the schedule if step (3) succeeded. Otherwise, there is no feasible solution for the set of constraints.

A modification in step (3) may involve swapping the order of generation of data vertices within a processor so that the time interval that a piece of data will occupy the memory is minimized. Another way of increasing memory efficiency is to control the time to transfer a data vertex from one processor to another. If a schedule cannot be obtained at the end of step (4), then we might want to increase the number of processors or use another optimization method such as branch-and-bound introduced earlier.

CHAPTER 3

DESIGNING AND ANALYZING DATA PARALLEL ALGORITHMS

In the previous chapter, we discussed general considerations in designing parallel algorithms. In this chapter, we turn our focus to designing data parallel algorithms for DMMs, with emphasis on implementation and analysis issues.

Designing parallel algorithms cannot be isolated completely from the architectural aspect of the underlying multiprocessor. Design and implementation of a parallel algorithm on a multiprocessor involves the knowledge of the specific language, operating system, and multiprocessor that are used. On DMMs, this may imply the knowledge of how send and receive routines are specified in the language and invoked in the operating system, how messages are maintained and transmitted, and what are the rates of performing computation and communication. The more knowledge one has, the more efficient the resultant program will be.

The same is true for analyzing algorithms. The purposes of analyzing algorithms include estimating algorithm performance to identify inefficiencies in the design and suggesting better scheduling schemes. Again, accurate models of the underlying architecture and algorithm behaviors are needed. These models dictate what timing information is relevant and how timing information can be combined to provide an overall performance figure.

In this chapter, the design, programming, and analysis of data parallel algorithms for two typical examples, array summation and matrix multiplication, are discussed [NiKP87]. These algorithms were implemented on a 64-node NCUBE multiprocessor. Therefore, in Section 3.1, important characteristics of the NCUBE are reviewed first. Then, modeling computation and communication on DMMs is discussed in Section 3.2, including the measured timings on the NCUBE. Based on the model, data parallel algorithms for array summation and matrix multiplication are presented in Section 3.3 and 3.4, respectively.

3.1. NCUBE Multiprocessors

The algorithms and experiments discussed in this thesis are implemented on the NCUBE multiprocessor. Therefore, in this section, the architecture of the NCUBE multiprocessor is reviewed. Section 3.1.1 will concentrate on the hardware aspect of the NCUBE, followed by an introduction in Section 3.1.2 to the software aspect of the NCUBE.

3.1.1. The Hardware

The NCUBE multiprocessor is a DMM with a hypercube interprocessor connection which is capable of supporting up to 1024 nodes [HaMS86]. A hypercube of dimension n consists of $N=2^n$ nodes, each with a direct connection to n other nodes. There is an edge between nodes i and j if i and j have exactly one bit difference in their binary representation, where $0 \le i,j \le N-1$. The most attractive feature of the hypercube topology is the rich set of topologies, including meshes and rings [SaSh85], that can be embedded inside the hypercube. Generally speaking, the hypercube offers a good balance between node connectivity, communication diameter, and algorithm embeddability. That is the reason why most DMMs available today use the hypercube as the

underlying interprocessor connection network [Hwan87].

The processor in each node of the NCUBE is a 32-bit proprietary processor which includes a float-point arithmetic unit, memory management logic, and interprocessor communication mechanism, all on a single VLSI chip. Combined with six memory chips for a total of 512 Kbytes, a node consists of only 7 chips. The instruction set is very similar to that of the VAX, which include arithmetic and logic operations, shift, and jump. Using a 10 *MHz* clock, nonarithmetic instructions can be executed at about 2 MIPS (million instructions per second), single-precision floating-point operations at 0.5 MFLOPS (million floating-point operations per second), and double-precision at 0.3 MFLOPS. The NCUBE used in this thesis research has 64 nodes with a clock rate at 7 *MHz*.

Each node has 22 bit-serial I/O lines which are paired into 11 bidirectional channels. In this way, a node can communicate with up to 10 other neighbors to form a 10-dimensional hypercube and has one more link to connect to the host. Communication with other nodes is handled through asynchronous DMA.

The host board in the NCUBE multiprocessor uses an Intel 80286 as the CPU and has support for various peripherals. Within each host board, there are 16 NCUBE node processors (called the I/O processors) to provide 128 channels to the hypercube, each I/O processor is connected to 8 nodes in the hypercube. The memory spaces of these I/O processors are part of the 4 Mbytes memory space of the host CPU. In this way, memory in the host board is shared between the 80286 CPU and 16 I/O processors.

The hardware design of the NCUBE multiprocessor is excellent. A 1024-node NCUBE has a potential execution rate of 2000 MIPS or about 500 MFLOPS. However, it is up to the software — the operating system and algorithm design — to realize this massive computing power.

3.1.2. The Software

The operating system running on the host of the NCUBE multiprocessor is called AXIS, which is UNIX-like but lacks many of the features of a mature UNIX. To AXIS, the hypercube is nothing more than a device. As with ordinary files, the hypercube can be opened, written to, read from, and closed. Also, AXIS allows a user to allocate disjoint subcubes. Thus, more than one user or application can share the hypercube. Programming on the NCUBE can be done using conventional C or FORTRAN augmented with system primitives to handle messages and cubes.

The node operating system is called VERTEX, which is a small nucleus (less than 4 Kbytes) residing in each node. Its primary function is to provide communication and process management. System primitives available from the VERTEX include:

nread()	receive messages
nwrite()	send messages
ntest()	test for messages arrivals
ntime()	return time since node initialization
whoami()	return node parameters such as node id and subcube dimension

From a node program, *nwrite()* can be called as follows:

nwrite(*msg, msglen, dest, type, status, flag*)

where *msglen* is the length of the message in bytes, *msg* is a linear array holding the message, *dest* is the logical id of the receiving node, *type* indicates the type of the message, and *status* and *flag* are for error checking purposes. Therefore, if data in a message are scattered in different places, the program must first gather data into a linear array before *nwrite()* can be called.

A pool of system buffers is maintained by VERTEX for communication. These buffers are used on both sending and receiving nodes to support non-blocking send and blocking receive operations [MuBA87]. Thus, the caller of *nwrite()* can resume after a

communication buffer has been allocated and the message has been copied into it. Buffers also serve the purpose of queuing messages when the communication channel is busy. When the DMA controlled message transfer has been completed, an interrupt is signaled. The corresponding interrupt service routine will release the communication buffers.

The arguments to *nread()* are similar to those of *nwrite()*. *nread()* examines the unclaimed receiving buffers for a message specified in its arguments. When the requested message does arrive and is located, *nread()* will copy the message to the user process space and release the message buffer. Otherwise, it will wait until an appropriate message arrives. Thus, *nread()* supports blocking receive.

Note that, even though the low level hardware implementation of an NCUBE node supports broadcasting, there is no broadcast primitive provided in the VERTEX. Note also that VERTEX uses a store-and-forward mechanism to relay messages. This mechanism has been shown to be inefficient and abandoned in the second generation DMMs [DaSe87, ShFi87].

Using current AXIS and VERTEX, not only is the programming more difficult than using the shared-variable model, but a very high communication overhead is induced by the inefficiency in the software. It follows that appropriate parallel programming tools and algorithm development methodologies are critical in efficiently utilizing the NCUBE and other DMMs.

3.2. MODELING COMPUTATION AND COMMUNICATION ON DMMS

Modeling an algorithm on a computer system is to identify the most essential characteristics of the algorithm, with a goal to predict the behavior of the algorithm when executed on the machine. To achieve this goal, we need to define *design parameters*, such as the size of partitions, to characterize the behavior of the algorithm.

Furthermore, we need to obtain *system parameters*, such as the message transmission time and primitive computation time, of the underlying architecture to characterize the behavior of the machine. In this section we will show how to model communication and computation in a DMM, and what are the measured system parameters on the NCUBE.

From the view point of programming, communication is nothing more than manipulating system calls to send and receive messages. In general, when invoking these communication primitives, the overhead involved can be classified as one that can be overlapped with the computation processor and one that cannot. For each category, one can further distinguish between the overhead that is invoked per message and the overhead that is paid on a per byte basis. Thus, for overhead that cannot be overlapped with the computation, we have the following estimations [GrRe86]:

$$\sigma_h + \tau_h \times s \quad \text{for communication initiated by the host} \tag{3.1}$$

$$\sigma_n + \tau_n \times s \quad \text{for communication initiated by the nodes} \tag{3.2}$$

where

σ_h = message startup delay in the host

τ_h = single-byte transmission time for messages initiated by the host

σ_n = message startup delay in the node

τ_n = single-byte transmission time for messages initiated by the nodes

s = size of the message (in *bytes*)

On the NCUBE, for example, σ_h and σ_n, will include such overhead as invoking the send and receive primitives, *nwrite()* and *nread()*, and setting up the DMA channels [MuBA87]. On the other hand, τ_h and τ_n will include the overhead to copy data to and from system buffers. Similarly, we can define σ_h', τ_h', σ_n', and τ_n', to characterize the overhead that can be overlapped with the computation processor. Specifically, τ_h' and τ_n' will include the time to transmit the message over the communication link. It follows that, to estimate the time to transmit a message of s bytes from the host to a node,

we can use the following formula:

$$(\sigma_h + \tau_h \times s) + (\sigma_h' + \tau_h' \times s) + \tau_n \times s \tag{3.3}$$

The first term accounts for the time to invoke the *nwrite()* primitive and to copy data to the system buffer, the second term takes care of the overhead involving data transmission over the link, and the final term is the time for the destination node to copy the message to the receiving process. It is assumed here that the receiving process has already issued the *nread()* call and is waiting for the message. Thus, the time σ_n is not included.

There is another important factor which will affect the performance of the resultant program, especially when matrices are involved. Note that matrices are usually stored in the memory in a row-major or column-major fashion. However, when sending a submatrix, the host process must first gather all elements of the submatrix into a consecutive linear array before it passes the starting address of that array to the send primitive (see Figure 3.1(a)). Similarly, when receiving a submatrix, the system must copy the message into a linear array before the receive primitive returns. Then, the receiving process scatters the data to the corresponding locations in the matrix (see Figure 3.1(b)). Note that the gathering/scattering operation is performed on the host sequentially, which cannot be parallelized. Therefore, it is a very expensive operation.

The host communication overheads with and without data gathering/scattering are both measured. To measure the host overhead in sending a message, the host sends r rounds of s-byte messages to n nodes. Similarly, the host overhead in receiving a message is measured by using n nodes, each sends r messages of s bytes to the host. The time, T, to perform these operations is measured on the NCUBE using the VERTEX system call, *ntime()*. To do this, the host designates one node, say node 0, as the time server. Immediately before the operation begins, the host sends one message to inform node 0 to start counting the time. Then, immediately after the operation finishes, the host sends another message to node 0 to stop the clock, and processor 0 sends back the

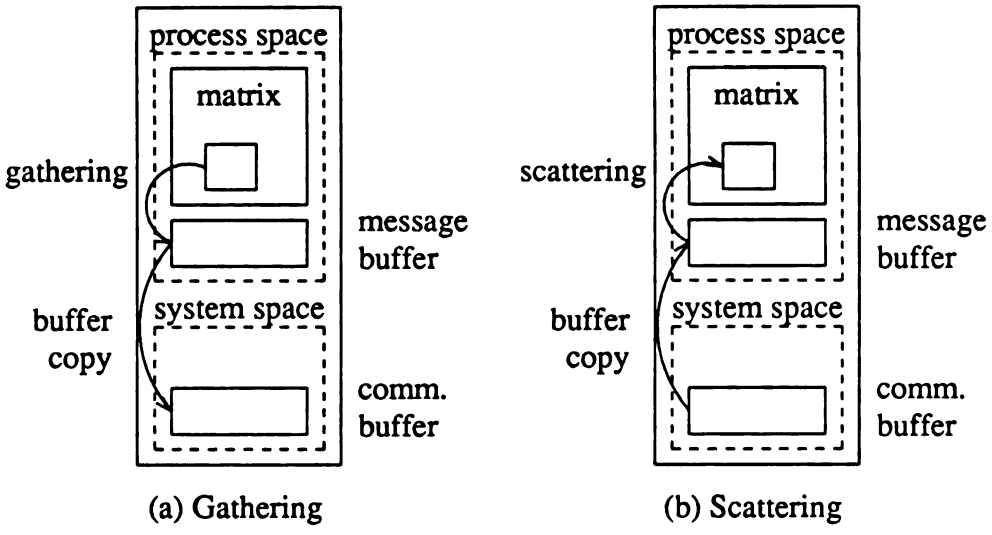


Figure 3.1. Gathering/scattering operations in the NCUBE

elapsed time. When the number of messages in the experiment is large, the overhead involved in sending these control messages can be ignored.

The value returned by *ntime()* is measured in *ticks* where 1 *tick* is equal to 1024/*processor clock rate* (in *Hz*). Thus, if the processor clock rate is 7 *MHz*, then 1 *tick* will equal to 146.2 μ*sec*. From above experiments, the host overhead in sending or receiving a message is given by T/nr. The average time in sending one message from the host to a node is depicted in Figure 3.2. Data gathering/scattering is simulated in the experiments by copying the message from one array to another in the host before it is sent or received.

It is easy to see from Figure 3.2 that the measured data can be fit into straight lines. In fact, if data gathering/scattering is included, then the measured host communication overhead can be modeled by

$$\sigma_h + \tau_h \times s = 8.20 + 0.068 \times s \quad \text{for sending a message} \tag{3.4}$$

$$\sigma_{hr} + \tau_h \times s = 4.55 + 0.068 \times s \quad \text{for receiving a message} \tag{3.5}$$

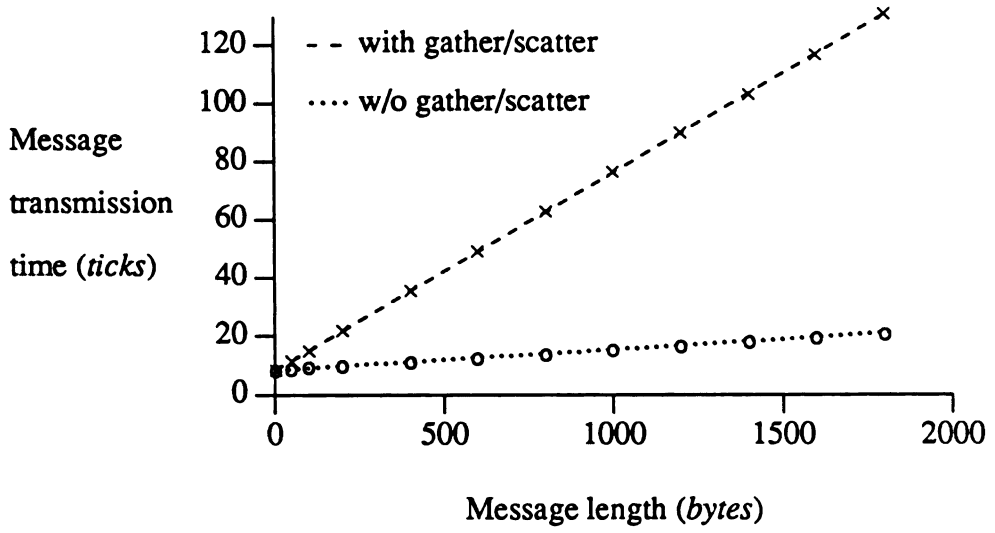


Figure 3.2. Measured host overhead in sending a message

On the other hand, if the computation does not involve data gathering/scattering, then the host communication time can be modeled by

$$\sigma_h + \tau_h \times s = 8.08 + 0.007 \times s \quad \text{for sending a message} \tag{3.6}$$

$$\sigma_{hr} + \tau_h \times s = 4.21 + 0.007 \times s \quad \text{for receiving a message} \tag{3.7}$$

Note that (3.4), (3.5), (3.6), and (3.7) all conform to (3.1). Also, from the experiments, we observe that, although there are some overlappings between computation and communication, e.g., the average message transmission time, T/nr, decreases as the number of rounds, r, increases, the net effect is too small. Therefore, in analyzing algorithms run on the NCUBE, we will ignore the parameters σ_h' and τ_h', which represent the communication overhead that can be overlapped with the computation processor.

Node communication overhead on the NCUBE can be measured by arranging n nodes into a ring. Again, r rounds of message exchanges are performed. In each round, a node sends one message of s bytes to the next node in the ring and receives one

message from the previous node. The measured average message transmission time ($T/2nr$) is plotted in Figure 3.3 against message sizes. Again, the message transmission time can be modeled linearly by

$$\sigma_n + \tau_n \times s = 3.52 + 0.017 \times s \tag{3.8}$$

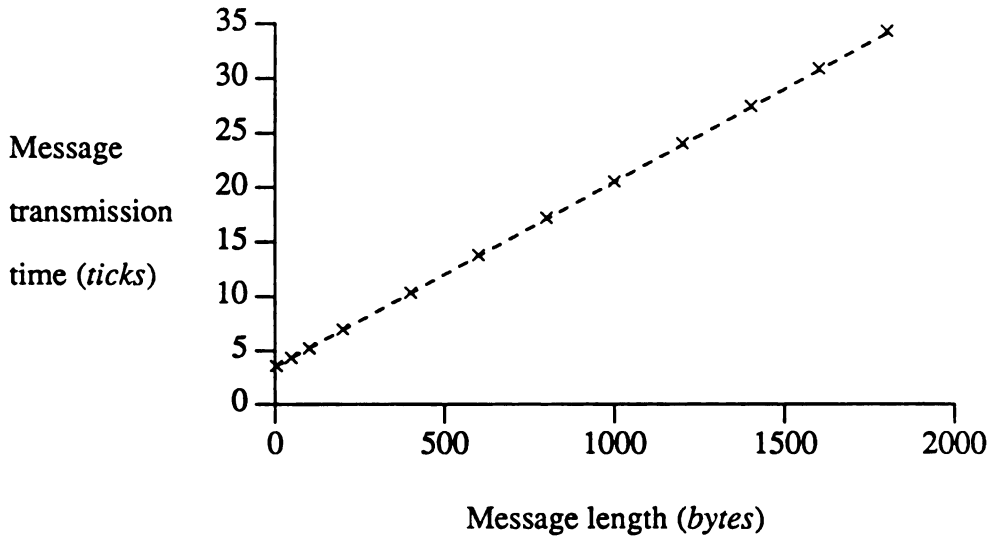


Figure 3.3. Measured node to node communication time

Note that (3.8) applies to both send and receive operations. Also, the overlapping between computation and communication is small, e.g., (3.8) is independent of the number of receiving nodes. Thus, σ_n' and τ_n' can be ignored in the modeling. Note that Expression (3.8) does not include data gathering/scattering operations, because in the algorithms studied in this thesis, nodes do not perform any gathering/scattering operations. Finally, the time (τ_c) to execute the operation $a \times b + c$ and associated loop manipulation overhead is measured, where a, b, and c are floating-point numbers. Similarly, the time (τ_b) for $a+b$ and associated overhead is measured. The following table summarizes the system parameters measured on the NCUBE:

Table 3.1. System parameters on the NCUBE

	σ_h	σ_{hr}	σ_n	τ_h	τ_n	τ_c	τ_b
with G/S	8.20	4.55	3.52	0.068	0.017	0.24	0.15
w/o G/S	8.08	4.21	3.52	0.007	0.013	0.24	0.15

<Note>: all quantities are in *ticks*

3.3. CASE STUDY — ARRAY SUMMATION

In this and the next section, we present two examples, array summation and matrix multiplication, to see how different factors, such as computation, communication, time, and space, may affect the performance of the algorithms. Algorithms were implemented on a 64-node NCUBE, which allows the discussion to relate to a more realistic environment.

Given a large set of numbers, $a_1, a_2, ..., a_L$, the array summation is to find out the sum, $A = a_1 + a_2 + ... + a_L$. To parallelize this operation, the array is partitioned into k subarrays, which are allocated to k processors of a hypercube computer. Each processor will receive one subarray and calculate the partial sum. The partial sums are then accumulated to give the total. The primary considerations here are the optimal number of processors to be used and the size of each partition.

Based on the way the partial sums are accumulated, we consider two different array summation schemes in Section 3.3.1. In the *centralized accumulation method*, the host collects all the partial sums from the processors and evaluates A. Another approach, the *tree-structured accumulation method*, uses a binary-tree reduction among all the nodes to accumulate the partial sums. In Section 3.3.2, performance of the algorithms will be modeled using the analytic model introduced in Section 3.2, and, in Section 3.3.3, the analytic results will be studied and compared with experimental results from the NCUBE.

3.3.1. The Algorithms

In the following discussion, let L denote the size of the array and x_i denote the size of the subarray loaded to node i, where $L = x_0 + \cdots + x_{k-1}$. The execution time (including local computation and communication) at node i is denoted by $t(x_i)$ and the size of the resultant data is denoted by $s(x_i)$. Suppose further that each data element takes b bytes. The P-net describing the array summation using centralized accumulation method is shown in Figure 3.4, where X_i, $0 \le i \le k-1$, is the set of array elements that are sent to processor i.

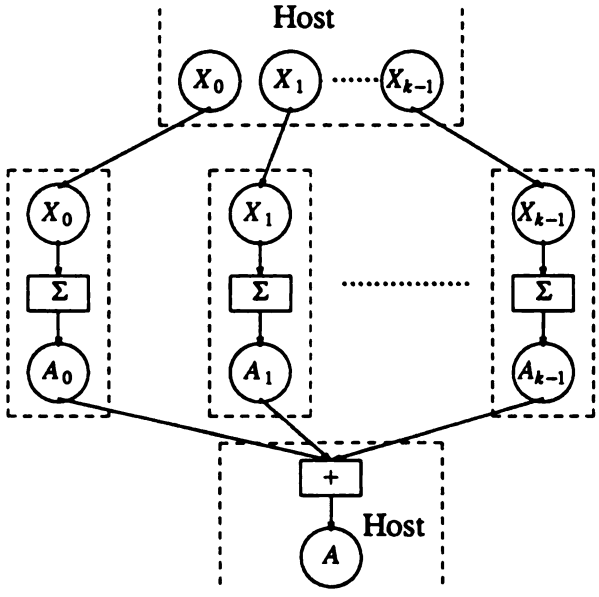


Figure 3.4. Array summation with centralized accumulation

Since there is no communication between nodes, any one-to-one mapping from the subarrays to the processors will generate the same result. A straightforward way of partitioning the array is the *equal partitioning* scheme, in which subarrays have the same size. However, this scheme does not take into consideration the balance between computation and communication. Thus, another partitioning scheme, *ratioed partitioning*, is

proposed, in which the array is divided in such a way that, when node i has just finished its uploading, node $i+1$ is ready to upload. By doing so, we can prevent the host from being idle between two adjacent data uploadings. In Section 3.3.2, we will show how to use analytic model to obtain the optimal partition sizes for ratioed partitioning.

The P-net which describes array summation using the tree-structured accumulation method is shown in Figure 3.5, where $k=4$ nodes are used with node 0 as the root of the reduction tree. The root of the reduction tree is responsible for returning A to the host. Note that this tree-structured reduction operation can be executed on a hypercube in such a way that data movements between nodes in each step are only one hop away [SaSh85], i.e., all nodes only have to communicate with their direct neighbors. Again, either the equal partitioning scheme or the ratioed partitioning scheme can be used.

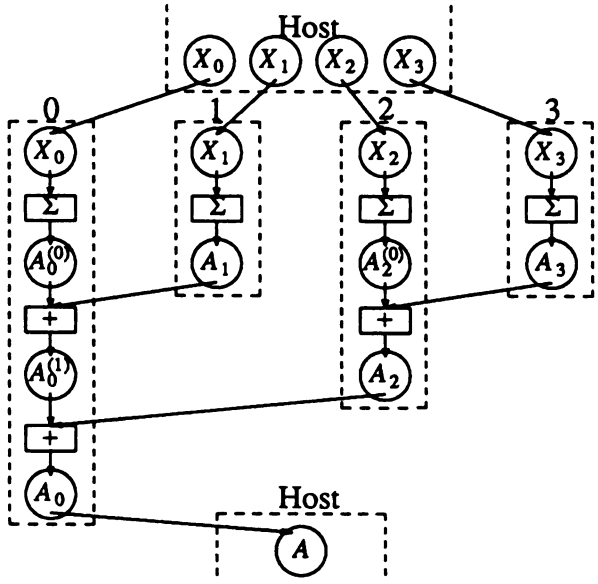


Figure 3.5. Array summation with tree-structured accumulation

3.3.2. Analytic Modeling

In this subsection, we will illustrate how to derive the execution time (T) for array summation. Consider first the centralized accumulation method using the ratioed partitioning scheme. The timing diagram of this method is shown in Figure 3.6, where C is the host idle time. From the timing relationship in Figure 3.6, any pair of x_i and x_{i+1}, $i = 0, ..., k-2$, can be related as follows:

$$(\tau_n b x_i + \tau_b x_i) + (\sigma_n + \tau_n b + \tau_h b) + \tau_b + \sigma_{hr} = (\sigma_h + \tau_h b x_{i+1} + \tau_n b x_{i+1}) + \tau_b x_{i+1} + (\sigma_n + \tau_n b)$$

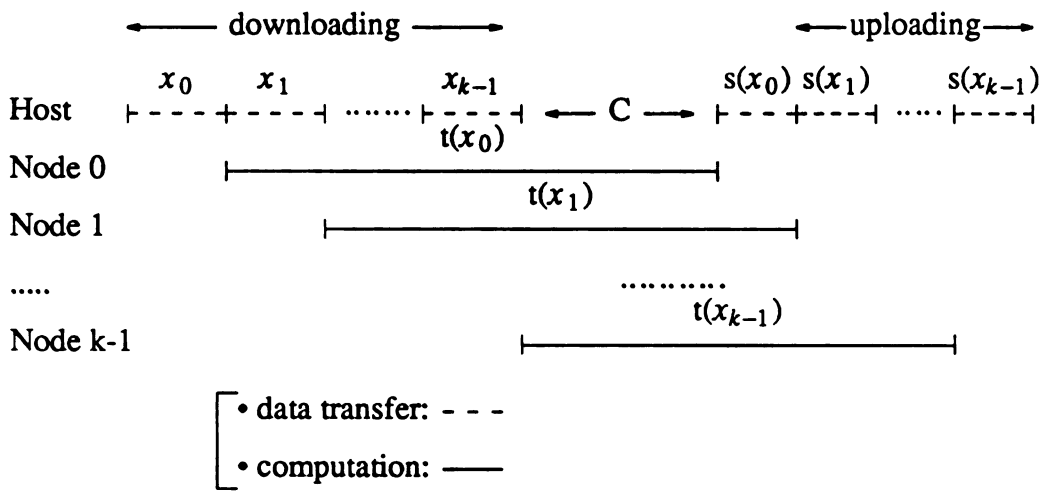


Figure 3.6. Timing diagram of array summation using centralized accumulation

In the left-hand side, $t(x_i)=(\tau_n b x_i+\tau_b x_i)$ comprises the local execution time in node i, including copying data from message buffer and calculating partial sum, $(\sigma_n+\tau_n b+\tau_h b)$ is the uploading time (see Equation (3.3)), and τ_b and σ_{hr} are the times during which the host accumulates the sum and invokes the next receive primitive, respectively. Quantities in the right-hand side can be interpreted similarly. Define

$$\alpha = \frac{\tau_n b + \tau_b}{\tau_h b + \tau_n b + \tau_b} \quad \text{and} \quad \beta = \frac{\tau_h b + \tau_b + \sigma_{hr} - \sigma_h}{\tau_h b + \tau_n b + \tau_b}$$

Then we have

$$x_{i+1} = \alpha x_i + \beta \quad i = 0, \ldots, k-2 \tag{3.9}$$

It is easy to derive from (3.9) that

$$x_i = \alpha^i x_0 + 1 - \alpha^i \quad i = 0, \ldots, k-1$$

Combined with the fact that $L = x_0 + \ldots + x_{k-1}$, we have

$$x_0 = \frac{(1-\alpha)L - n\beta}{1-\alpha^k} + \frac{\beta}{1-\alpha} \tag{3.10}$$

Let T_d denote the downloading time of the last $k-1$ subarrays, i.e.,

$$T_d = \sum_{i=1}^{k-1} (\sigma_h + \tau_h b x_i) = (k-1)\sigma_h + \tau_h b (L - x_0)$$

Then, the host waiting time C can be obtained from the timing relationship between the host and node 0 in Figure 3.6 as

$$C = \rho(\tau_n b x_0 + \tau_b x_0 + \sigma_n + \tau_n b - T_d + \sigma_{hr})$$

where

$$\rho(y) = \begin{cases} y & \text{if } y > 0 \\ 0 & \text{otherwise} \end{cases}$$

Note that $C=0$ implies that the host is a bottleneck, because, before the host finishes downloading to node $k-1$, node 0 has generated the first partial sum and is waiting for uploading. In other words, the host cannot keep up with the computing speed of the nodes. This also implies that we use too many processors that the communication dominates the computation. From above expressions, the whole job execution time can be found as:

$$T = \sum_{i=0}^{k-1} (\sigma_h + \tau_h b x_i) + C + \sum_{i=0}^{k-1} (\sigma_{hr} + \tau_h b + \tau_b)$$

$$= k(\sigma_h + \sigma_{hr} + b\tau_h + \tau_b) + b\tau_h L + C \tag{3.11}$$

For the equal partitioning scheme, the derivation of the total execution time is easier. Note that in this case we have

$$x_0 = x_1 = \cdots = x_{k-1} = \frac{L}{k}$$

If the host is the bottleneck, then we need only consider operations in the host, which is equal to

$$T_1 = k(\sigma_h + \tau_h b\frac{L}{k}) + k(\sigma_{hs} + \tau_h b + \tau_b)$$

On the other hand, if node computation dominates the array summation, then we need only consider the operations in node $k-1$, which take

$$T_2 = k(\sigma_h + \tau_h b\frac{L}{k}) + (\tau_n b + \tau_b)\frac{L}{k} + (\sigma_n + \tau_n b + \tau_h b) + \tau_b$$

It follows that the total execution time can be approximated by

$$T = \text{Max}\{T_1, T_2\} \tag{3.12}$$

Derivations for tree-structured accumulation methods are similar. The timing diagram of the tree-structured accumulation method with ratioed partitioning is shown in Figure 3.7. Using ratioed partitioning scheme, we can identify the following relationship between adjacent nodes from Figure 3.7:

$$\tau_n b x_i + \tau_b x_i = (\sigma_h + \tau_h b x_{i+1} + \tau_n b x_{i+1}) + \tau_b x_{i+1}$$

Defining

$$\alpha = \frac{\tau_n b + \tau_b}{\tau_h b + \tau_n b + \tau_b} \quad \text{and} \quad \xi = \frac{\sigma_h}{\tau_h b + \tau_n b + \tau_b}$$

we have

$$x_{i+1} = \alpha x_i - \xi \tag{3.13}$$

$$x_0 = \frac{(1-\alpha)L - n\xi}{1-\alpha^k} - \frac{\xi}{1-\alpha}$$

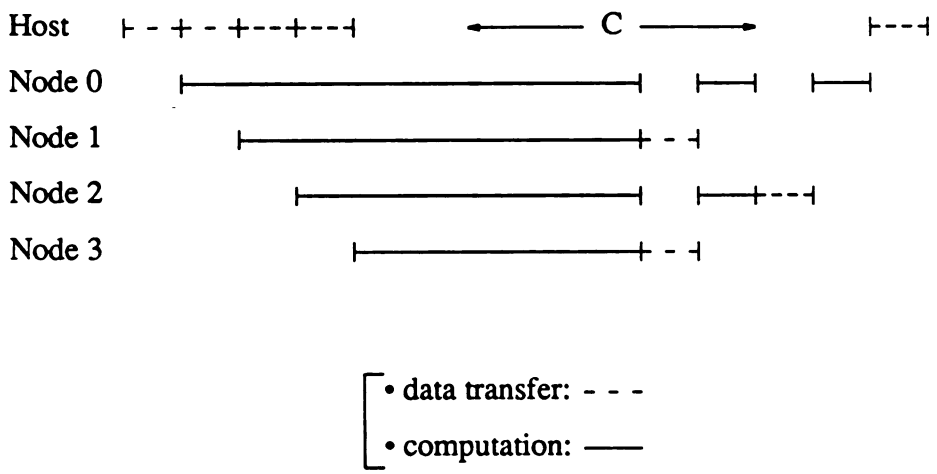


Figure 3.7. Timing diagram of array summation using tree-structured accumulation

$$C = (\tau_n b+\tau_b)x_{k-1} + \log_2 k(\sigma_n+2\tau_n b+\tau_b) + (\sigma_n+\tau_n b)$$

$$T = k\sigma_h + (L+1)\tau_h b + C \qquad (3.14)$$

Similarly, the execution time for the equal partitioning scheme is given by

$$T = (k+1)\sigma_h + (L+1)\tau_h b + (\tau_n b+\tau_b)\frac{L}{k} + \tau_n b + \log_2 k(\sigma_n+2\tau_n b+\tau_b) \qquad (3.15)$$

From (3.11), (3.12), (3.14), and (3.15), performance of algorithms running on the NCUBE can be estimated using the system parameters given in Table 3.1. Optimal design parameters, k and x_i, $0 \le i \le k-1$, can also be determined by optimizing these expressions.

3.3.3. Performance Analysis

In this subsection, performance of the array summation algorithms presented in Section 3.3.1 will be studied. First of all, the centralized accumulation method using the ratioed and equal partitioning schemes are compared (see Figure 3.8). The size of the

array is 8192 elements. The partition sizes, x_i, $0 \le i \le k-1$, for the ratioed partitioning scheme are obtained from Equations (3.9) and (3.10), and the analyzed execution times are obtained from (3.11) and (3.12).

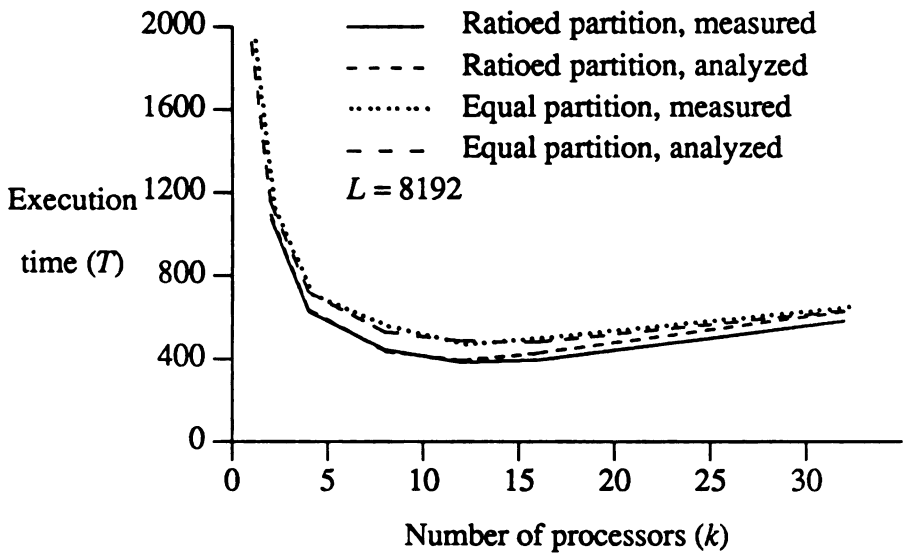


Figure 3.8. Comparison of ratioed and equal partition

The accuracy of the analytic models is evident from Figure 3.8 by observing the close match between the curves representing analytic and experimental results. From Figure 3.8, the optimal number of processors for each scheme can be obtained at the "valley" of the corresponding curve — k=12 for both schemes. Note that the optimal number of processors is the dividing point between host-bound or node-bound execution. Beyond this point, the host becomes the bottleneck. It can also be seen from Figure 3.8 that the ratioed partitioning scheme improves the performance, because a better balance between communication and computation can be achieved by adjusting data partition sizes.

Comparison of centralized and tree-structured accumulation is presented in Figure 3.9 for both the equal and the ratioed partitioning schemes. Again, the partition sizes of

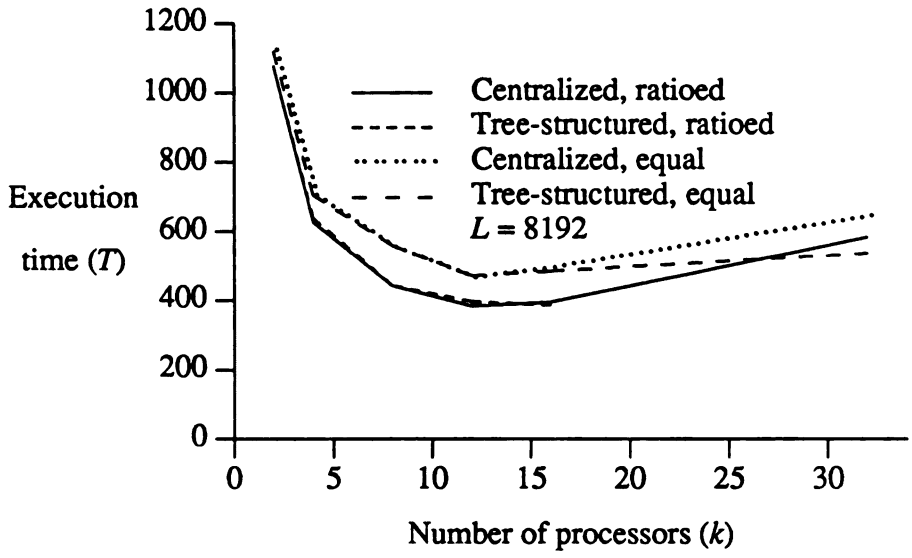


Figure 3.9. Comparison of tree-structured and centralized accumulation

the ratioed partitioning schemes are derived from the analysis. It can be seen from Figure 3.9 that the performance of the tree-structured approach is almost identical to that of the centralized approach, for all k up to the optimal value. This is because, for array summation, the accumulation phase involves only the transmission and addition of a single data element, which incurs only a small portion of the total execution time. However, when k becomes large, the difference between these two approaches becomes apparent. This is evident from the curve corresponding to the equal partitioning, tree-structured accumulation in Figure 3.9, which extends below the curve of the ratioed partitioning, centralized accumulation method when $k \geq 27$.

For the tree-structured accumulation method, the optimal partition sizes obtained from (3.13) will become negative when k becomes large. This implies that the given number of processors is too large for efficient execution. This is also the reason why, in Figure 3.9, there is no data beyond $k=16$ for the tree-structured accumulation method using ratioed partition.

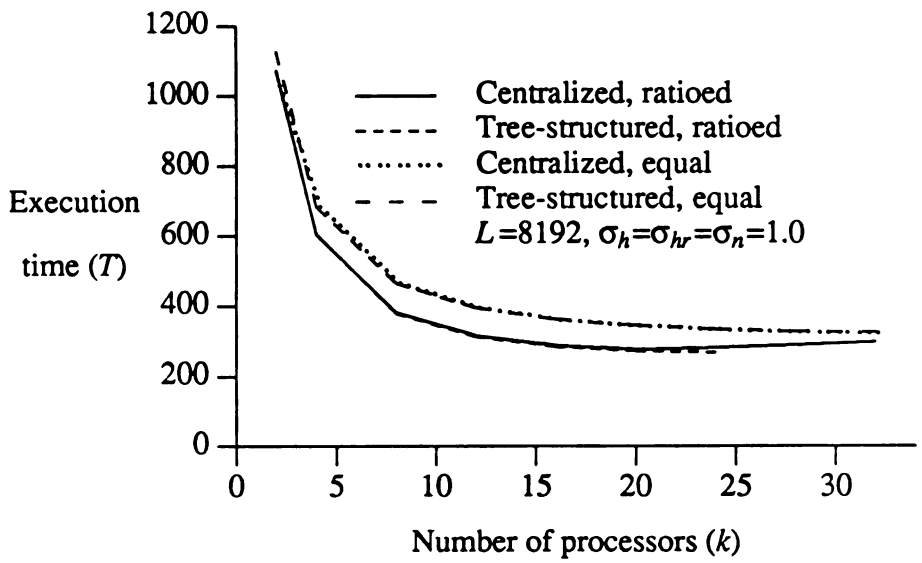


Figure 3.10. Predicted performance of various methods

Next, let us consider the influence of system parameters on algorithm design. Suppose the message startup delay can be improved from 8.08 *ticks* on the host and from 4.21 *ticks* on the nodes to 1 *tick*. Figure 3.10 illustrates the resultant total execution time, predicted by means of the analytic models. Again, tree-structured and centralized accumulations perform almost the same. Compared with Figure 3.9, the improvement due to decreased message startup delay is about 25% for the optimal execution time. In addition, the host can support more processors to improve the performance.

3.4. CASE STUDY — MATRIX MULTIPLICATION

Matrix multiplication on a hypercube multiprocessor has been studied in different contexts [ChSm87, FoOH87]. In this section, the same problem is studied again. The purpose is not to propose any novel algorithm, but to show the tradeoffs between different partitioning and mapping schemes and to illustrate ways of arriving at good

design decisions. It is assumed that the multiplication of the two $m \times m$ matrices, $A \times B = C$, is to be performed on a hypercube with a fixed dimension, $n = \log_2 N$. Each element in the matrices has a size of b bytes. Matrices A and B are initially stored in the host and C will be stored back to the host. It follows that the overhead involved in data downloading and uploading must also be taken into account.

Various considerations in designing the matrix multiplication algorithm will be discussed in Section 3.4.1, followed by analytic modeling of the algorithm in Section 3.4.2. Using both experimental and analytic results of the matrix multiplication algorithm, we then discuss in Section 3.4.3 different tradeoffs in arriving efficient parallel algorithm designs.

3.4.1. The Algorithms

A straightforward approach to perform the matrix multiplication on a hypercube machine is to partition matrix B into N, $m \times (m/N)$ submatrices, $B_0, B_1, ..., B_{N-1}$. Each node is loaded with one submatrix B_i together with the whole matrix A. The resultant products $C_i = A \times B_i$ are then transferred back to the host to form a complete matrix C. The mapping is trivial because there is no interprocessor communication. However, each node will require a large amount of local memory to hold matrices A, B_i, and C_i. With matrices of size 512×512 running on a 5-cube and having 4 bytes per element, each node will require more than 1 Mbyte of local memory.

To relax the memory requirement, we can partition both matrices A and B. Consider the strip partition shown in Figure 3.11(a), in which matrix A is partitioned along the rows and B along the columns. Each processor is allocated with one submatrix of A and one submatrix of B. In each iteration, the processor performs a multiplication on the two local submatrices, and then shifts the B submatrix to the next processor in a ring fashion (as indicated by the arrows in Figure 3.11(b)). Repeating this circulation and

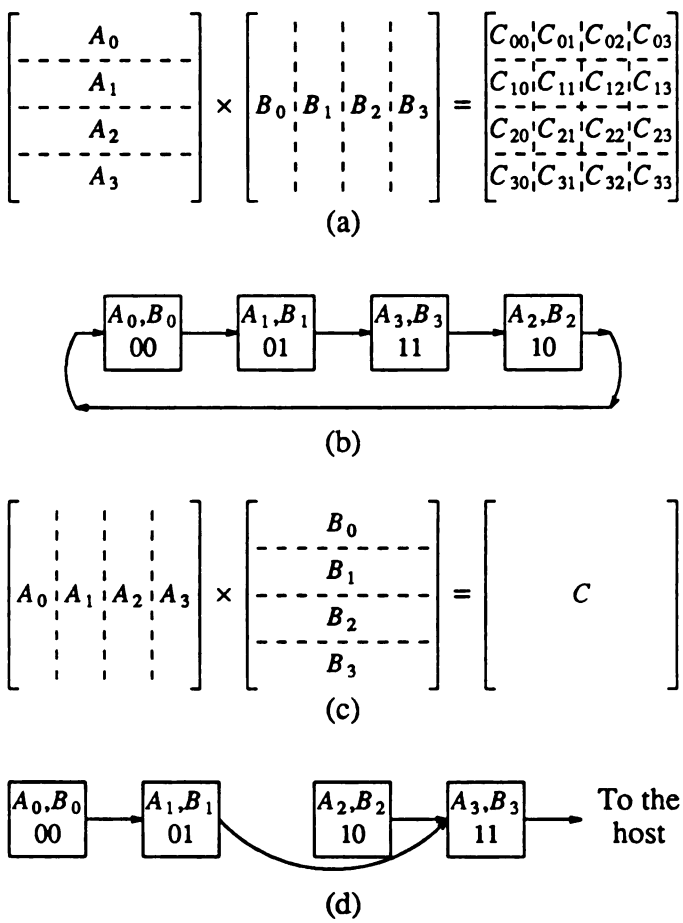


Figure 3.11. Matrix Multiplication with strip partition

multiplication, we can evaluate all submatrices of C. Note that by using the grey code sequence as shown in Figure 3.11(b), all B submatrix circulations are only one hop away. Also, all submatrices of C can be sent back to the host as soon as they are generated.

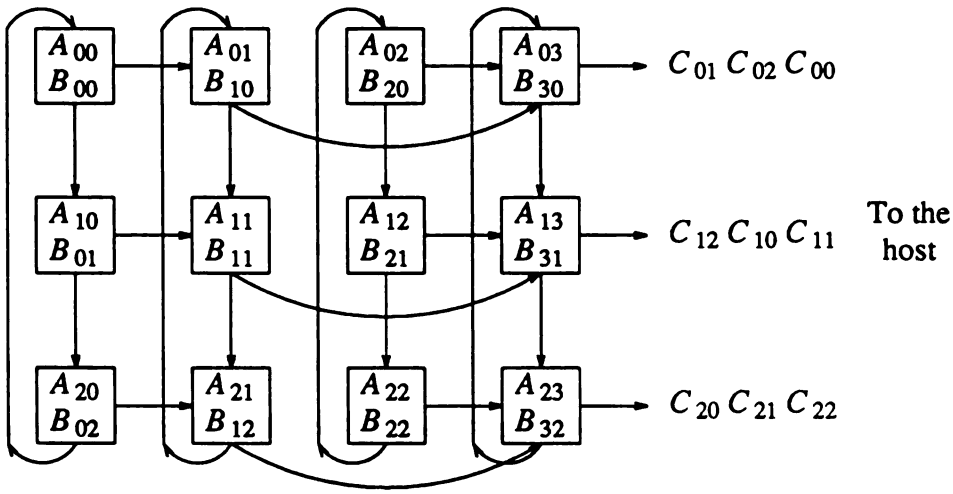
Another strip partition method is to divide matrix A along the columns and B along the rows, as shown in Figure 3.11(c). Again, each pair of submatrices A_i and B_i are loaded into one node, $0 \le i \le 3$. in this case, we obtain at each node a portion of the whole matrix C, $C^{(i)} = A_i \times B_i$. One way to accumulate C, where $C = C^{(0)}+C^{(1)}+C^{(2)}+C^{(3)}$, from all nodes is the tree-structured accumulation method (see Figure 3.11(d)). The root of the reduction tree is responsible for uploading the resultant matrix C back to the host. Note that all data transfers in the reduction tree are only one hop away.

In fact, the above two methods are only special cases of the block partition scheme. In general, we can partition the matrices along both columns and rows, and use two parameters, n_1 and n_2, to denote the number of partitions along rows and columns, respectively (see Figure 3.12 for n_1=3 and n_2=4). In each iteration, every node performs a submatrix multiplication and accumulates the resultant C submatrices with all nodes in the same row using a binary-tree reduction. B submatrices are then transmitted down the column in a ring fashion. Assume that n_1 and n_2 are integer powers of two. Let $node_{ij}$ denote the processor located at row i and column j of the mesh. Then the matrix multiplication algorithm can be taken as a combination of the ring and tree operations presented above. The algorithm is given in Figure 3.13.

There are n_2 rings involved in the multiplication. Each ring contains n_1 nodes. Tree reduction is performed across the rings between corresponding nodes. Note that in Algorithm 3.1 the roots of the reduction trees are shifted at each iteration so that the corresponding processors will not be overloaded by having to upload every submatrix of C. The program for performing matrix multiplication using Algorithm 3.1 is listed in

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} & B_{02} \\ B_{10} & B_{11} & B_{12} \\ B_{20} & B_{21} & B_{22} \\ B_{30} & B_{31} & B_{32} \end{bmatrix} = \begin{bmatrix} C_{00} & C_{01} & C_{02} \\ C_{10} & C_{11} & C_{12} \\ C_{20} & C_{21} & C_{22} \end{bmatrix}$$

(a)



(b)

Figure 3.12. Matrix multiplication with block partition

<Algorithm 3.1: matrix multiplication>
Host:
load A_{ij} and B_{ji} to $node_{ij}$, $0 \le i \le n_1-1$, $0 \le j \le n_2-1$;
receive C_{ij} from $node_{ik}$, $0 \le i,j \le n_1-1$, $k = j \bmod n_2$;
$C \leftarrow [C_{ij}]$, $0 \le i,j \le n_1-1$.
Node: ($node_{ij}$, $0 \le i = n_{1-1}$, $0 \le j \le n_2-1$)
receive A_{ij} and B_{ji} from the host;
for $l \leftarrow i$ to n_1-1 then 0 to $i-1$ do
$C_{il}^{(j)} \leftarrow A_{ij} \times B_{jl}$;
perform a tree reduction with $node_{ik}$, $0 \le k \le n_2-1$, using $node_{ir}$,
$r = l \bmod n_2$ as the root;
if ($j = l \bmod n_2$) then send C_{il} to the host;
if ($j \neq i-1$) then
send B_{jl} to $node_{kj}$, $k = (i-1) \bmod n_1$;
receive $B_{j,l+1}$ from $node_{kj}$, $k = (i+1) \bmod n_1$;

Figure 3.13. The algorithm for matrix multiplication

the Appendix.

3.4.2. Analytic Modeling

We now proceed to analyze the performance of Algorithm 3.1. Before this can be done, it should be noted that in Figure 3.12 the host has to gather data before it downloads submatrices and scatter data after it receives submatrices. Therefore, we should use (3.4) and (3.5) to model the host communication overhead.

Now consider the matrix multiplication algorithm. Due to the operation of ring shifting, nodes in the same column will be synchronized by the slowest one — the one that is loaded the last by the host. Downloading submatrices of A and B from the host to all the corresponding nodes will take time

$$T_d = 2N(\sigma_h + \frac{m^2}{N}\tau_h b)$$

At each iteration, the following operations will be performed in the nodes:

(1) Receive a submatrix of B, which takes time $\tau_n bm^2/N$;

(2) Perform a submatrix multiplication, which takes time $(\tau_c m^3)/(n_1^2 n_2)$;

(3) Perform a tree reduction with all nodes in the same row, which takes time $\log_2 n_2(\sigma_n+(2\tau_n b+\tau_b)m^2/n_1^2)$;

(4) The root of the reduction tree sends the resultant submatrix of C to the host, which takes time $\sigma_n+\tau_n bm^2/n_1^2$;

(5) Send submatrix of B to the next node, which takes time $\sigma_n+\tau_n bm^2/N$;

(6) Prepare to receive the next submatrix of B, which takes time σ_n.

It follows that the iteration time at each node is equal to

$$T_i = 2(\sigma_n+\tau_n b\frac{m^2}{N}) + \tau_c\frac{m^3}{n_1^2 n_2} + \log_2 n_2(\sigma_n+(2\tau_n b+\tau_b)\frac{m^2}{n_1^2}) + (\sigma_n+\tau_n b\frac{m^2}{n_1^2})$$

Now consider the host. During each iteration the host will receive n_1 submatrices of C, i.e.,

$$T_u = n_1(\sigma_{hr}+\tau_h b\frac{m^2}{n_1^2})$$

The total execution time of the algorithm depends on which event takes longer. Let

$$T_m = 2\sigma_n+\tau_n b\frac{m^2}{N}$$

We have

$$T = T_d + (T_i - T_m) + T_u + (n_1-1)\text{Max}\{T_i, T_u\} \qquad (3.16)$$

3.4.3. Performance Study

In this subsection, analytic results of Algorithm 3.1 are compared with experimental results obtained from the NCUBE to gain more insight into design considerations. Note that the execution time given in (3.16) is a function of both n_1 and n_2. However,

if the problem is running on a particular hypercube, the size of the hypercube N is known. Then, we need only consider n_1 ($N=n_1 n_2$).

Figure 3.14 illustrates the relationship between n_1 and the execution time (T) for various sizes of the hypercube. The experiment uses 64×64 matrices. Again, results from the analytic model (Equation (3.16)) match closely with those measured from the NCUBE. Therefore, the analytic model can be used to predict the optimal partitions, n_1 and n_2.

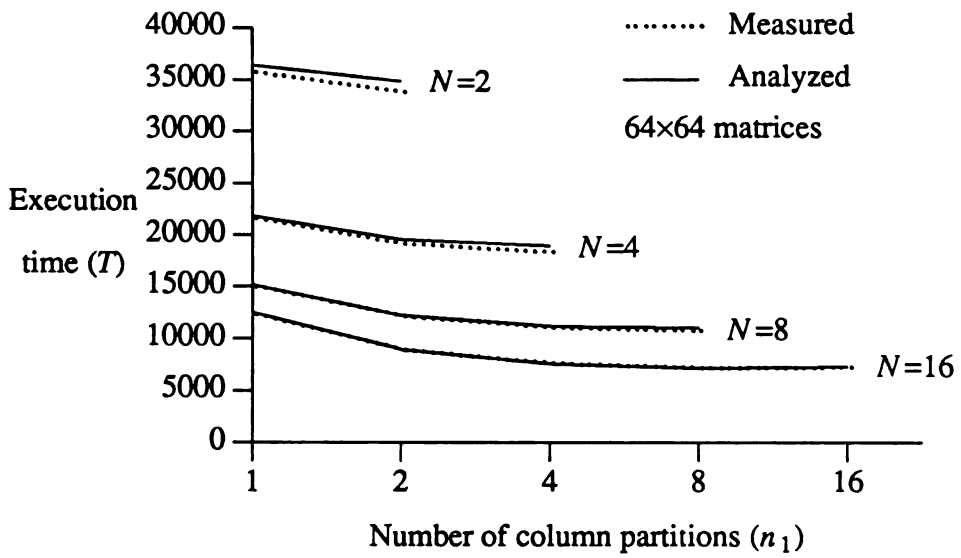


Figure 3.14. Execution time of the matrix multiplication

From Figure 3.14 it can be seen that performance degrades when n_1 is small. Recall that there are n_1 tree reductions ($\log_2 n_2$ steps each) and n_1-1 ring circulations. The smaller that n_1 is, the fewer number of data transfers there are. However, each step in a tree reduction requires the transmission of a matrix $C_{ij}^{(j)}$, which has $(m/n_1)^2$ elements. Thus, the total size of data to be transferred in the tree reduction phase is

$$n_1\left(\frac{m}{n_1}\right)^2 \log_2 n_2 = \frac{m^2}{n_1}\log_2\frac{N}{n_1} \tag{3.17}$$

The smaller that n_1 is, the larger the data size given in (3.17) will be, and the longer it will take to transmit the data. This results in a longer execution time as shown in Figure 3.14. The analytic model presented in Section 3.4.2 is a very useful tool in choosing the optimal n_1 and arriving at a balanced design.

The *speedup* of an algorithm is defined to be

$$\text{speedup} = \frac{\text{execution time using } N \text{ nodes}}{\text{execution time using one node}} \tag{3.18}$$

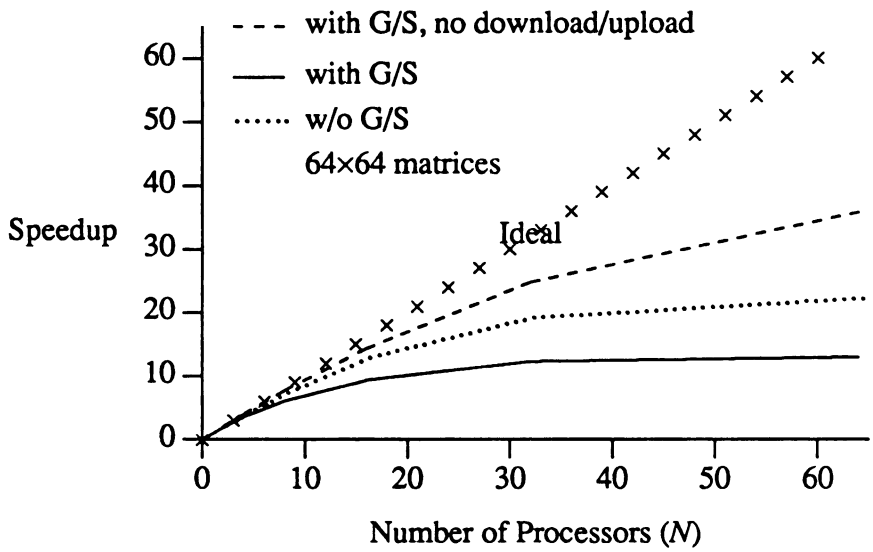


Figure 3.15. Predicted speedup of the matrix multiplication

The optimal speedups for different sizes of the hypercube is depicted in solid line in Figure 3.15. It can be seen in Figure 3.15 that the speedup levels off very fast. Now suppose the gathering/scattering operations can be eliminated by, say, allowing the send primitive to access the matrix directly, then the speedup would nearly double (see the dotted line in Figure 3.15). Furthermore, if matrices reside in the nodes before and after the multiplication [ChSm87, FoOH87], then there will be no data downloading and uploading. The dashed curve in Figure 3.15 shows that the performance in this case is

much better than the original one (almost triple). Therefore, it is desirable to keep operations at the nodes in the hypercube as much as possible, avoiding unnecessary data downloading and uploading.

Finally, assuming $n_1 \leq n_2$, then, the memory requirement at each node for Algorithm 3.1 is

$$2 \times b \times \frac{m^2}{N} + b \times (\frac{m}{n_1})^2$$

Again, considering the case where matrices have a size of 512×512 running on a 5-cube, the local memory required is about 131 Kbytes using the partition n_1=4 and n_2=8, which improves over the 1 Mbyte memory needed in the first approach described earlier in Section 3.4.1.

CHAPTER 4

LARGE-GRAIN PIPELINING

One of the reasons why parallel algorithm design is considered difficult is the lack of design guidelines. Without general methodologies, it is very hard for a beginner to start with parallel algorithm development. Over the years, as we are gaining in experience with parallel programming, several problem solving strategies are identified as effective means in structuring efficient parallel algorithms. These strategies are called *programming paradigms*.

As pointed out in [NeSn87] programming paradigms not only provide a means for programmers to begin when designing a new algorithm, but they also relay solutions for other related problems to the problem at hand. Nevertheless, paradigms for parallel programming are different from those for conventional serial computations. Parallel programming paradigms must address the issues of partitioning, cooperation, sharing, and communication.

In Section 4.1, we will first review important parallel programming paradigms. Based on the characteristics of DMMs and pipelined data parallel computations, a new programming paradigm, called *large-grain pipelining* [KiCN88a, KiNi88], is introduced in Section 4.2. Large-grain pipelining exploits the development of pipelined data parallel algorithms on DMMs. Using processor level macro-pipelining in the DMM, the resultant algorithm can achieve a computation rate that balances the communication rate. As an example, a pipelined matrix multiplication is presented. Again, the purpose

is not to introduce any new algorithm, but to illustrate the basic idea behind large-grain pipelining. In Section 4.3, important characteristics of large-grain pipelining will be summarized and compared with other programming paradigms. Finally, in Section 4.4, experimental results of the pipelined matrix multiplication algorithm on the NCUBE are presented. Various factors that might affect the performance are discussed.

4.1. PARALLEL PROGRAMMING PARADIGMS

In general, there is no precise definition of what a paradigm should or should not be. In fact, different researchers might use different names to describe similar ideas and fail short, in one way or another, in accounting for some other aspects of the ideas [NeSn87, Fink87, Quin87]. Therefore, the discussion in this section is subjective and is by-no-means exhaustive and complete. Note that different paradigms may sometimes overlap with each other. Note also that a parallel algorithm might use a combination of several paradigms.

(1) Divide-and-conquer

In a broad sense, all parallel problem-solving strategies use some sort of divide-and-conquer technique. However, the divide-and-conquer paradigm discussed here is more oriented toward a recursive and usually logarithmic style of computation. The problem is divided into smaller subproblems. The subproblems are further divided into even smaller subproblems until the sizes of the subproblems are manageable, where these subproblems are solved independently. The solutions are then combined together recursively until the final solution is obtained.

Due to its recursive nature, the divide-and-conquer paradigm can easily generate parallel algorithms with a logarithmic time complexity. It is also obvious that divide-and-conquer is most suitable for developing concurrent data parallel algorithms, in which data are partitioned recursively and processed on multiple processors. Several

algorithms belong to this category, including bitonic-merge sort and fast Fourier transform (FFT).

(2) Master-slave

In this paradigm, one or more *masters* generate and deal out tasks to a set of *slaves*. Slaves carry out the tasks (usually independently), and the results are gathered at the masters. The master-slave paradigm is suitable for developing concurrent algorithms, either data or function parallel.

A variation of the master-slave paradigm is the *compute-aggregate-broadcast* (CAB) paradigm [NeSn87], in which processors perform some computations independently, resultant data are combined into one or a few global values, and these global values are broadcast back to each processor. The processors that aggregate and broadcast the global values play the role of masters. Algorithms using the CAB paradigm are usually engaged in a loop with each sequence of compute, aggregate, and broadcast constituting one iteration. The CAB paradigm is more suitable for concurrent data parallel algorithms due to the broadcast operation.

Another variation of the master-slave paradigm is the *client-server* style of computation found in many distributed systems. Clients (the masters) direct their requests to a number of servers (the slaves) which perform services on behalf of the clients. Many parallel program developing tools for SMMs, such as Force [Jord87, Oste87], adopt this paradigm in scheduling the computation. One advantage of the client-server paradigm is dynamic load balancing among the servers. Also, the computation is independent of the number of servers available. This paradigm is suitable for developing concurrent algorithms.

(3) Election

Election is the operation in which one processor from among a group of processors is singled out to perform some special functions. Election finds its application mostly in distributed systems, including replicated data updating, crash recovery, and mutual

exclusion [KiGN88]. Parallel algorithms on multiprocessors also exhibit the behavior of election, e.g., finding the pivot column in a parallelized simplex method [Fink87]. Sometimes, election is the only way to perform reliable computation in an unreliable environment.

(4) Relaxation

In relaxation, each processor works with the most recently available data [Quin87] — no processor has to wait for another processor to provide it with data. The computation proceeds in an asynchronous and iterative fashion. Usually, during each iteration, a processor needs only exchange its most recent values with neighboring processors. A typical example of relaxation is solving partial differential equations (PDEs). However, the asynchronous nature of this paradigm causes difficulties in performance prediction and requires extra effort to detect the terminate conditions. In general, relaxation is very suitable for data parallel algorithms.

(5) Pipelined and systolic computation

Pipelining, in its most common sense, refers to the operation involving an ordered set of stages in which the output of one stage is the input of the next stage. Each stage behaves like a filter; and data, when passed along the pipeline, are modified along the way. Traditionally, pipelining is used to develop pipelined function parallel algorithms (see Section 1.3). Examples include multi-pass transformers such as compilers, scene analyzers, and vector processors.

Systolic algorithms are a special kind of pipelined data parallel algorithm which centered around VLSI computations [FoWa87, KuLe78]. Systolic algorithms are characterized by regular and rhythmic data flows and a set of identical and simple processors. Communication with the outside world occurs only at the boundary processors. Thus, whenever a datum enters the system, it is used repetitively in the pipelines.

In general, systolic algorithms are synchronous (except wavefront arrays [KuLJ87]). Therefore, timing is very important in systolic arrays. Furthermore, since

processing cells in a systolic array are simple and primitive, systolic algorithms use a very fine granularity (at the level of single data elements). Systolic algorithms are suitable for developing data parallel algorithms and have applications in signal processing, matrix arithmetic, image processing, and pattern matching, and dynamic programming.

From the above discussion, one can see that most of the parallel programming paradigms focus on concurrent computation. In view of the high communication overhead in DMMs, pipelining seems to be a more attractive programming paradigm for these machines. However, traditional pipelining techniques concentrate on function parallelism, while systolic algorithms aim at VLSI computation, which uses very fine granularity. Neither is oriented at developing efficient data parallel algorithms on DMMs. In the next section, a new paradigm, called *large-grain pipelining* (LGP), is introduced. LGP is specifically directed at developing parallel data parallel algorithms on DMMs.

4.2 BASIC CONCEPT OF LARGE-GRAIN PIPELINING

The basic idea of LGP is to use node level macro-pipelining in a DMM to regulate the information flows in the system so that data can be processed and transmitted in a pipelined fashion. Nodes in the multiprocessor are organized into pipelines, and data are partitioned into *blocks* to form data streams. The sizes of data blocks determine the partition sizes of the problem, and the directions of data flows determine the execution sequence. Note that although each node may be equipped with pipelined arithmetic units, the major concern of LGP is the pipelining between nodes. Through LGP, the degree of overlapping between computation and communication can be maximized, which minimizes the effect of communication overhead. The net result is a balance between the computation rate and the communication bandwidth.

As an example of LGP, consider again the multiplication of two matrices, $A \times B = C$, where A, B, and C are $m \times m$ matrices. As in Section 3.4.1, A and B are supposed to be stored in the host initially, and C will be stored back to the host. Thus, input data must be downloaded from the host to the nodes and results must be uploaded back to the host.

Assume that the multiplication is to be performed on a hypercube DMM with N nodes. The following partitioning scheme will be used: The hypercube is configured into an $n_1 \times n_2 = N$ mesh, and matrices A and B are partitioned along columns and rows into $n_1 \times n_2$ and $n_2 \times n_3$ submatrices, respectively.

One way to perform the multiplication is to load submatrices of A into the corresponding processors in the mesh, then pipe submatrices of B into the hypercube from the host. Let $node_{ij}$ denote the processor located at row i and column j of the mesh. The algorithm is described in Figure 4.1. The corresponding data flows in the algorithm are shown in Figure 4.2.

<Algorithm 4.1>:

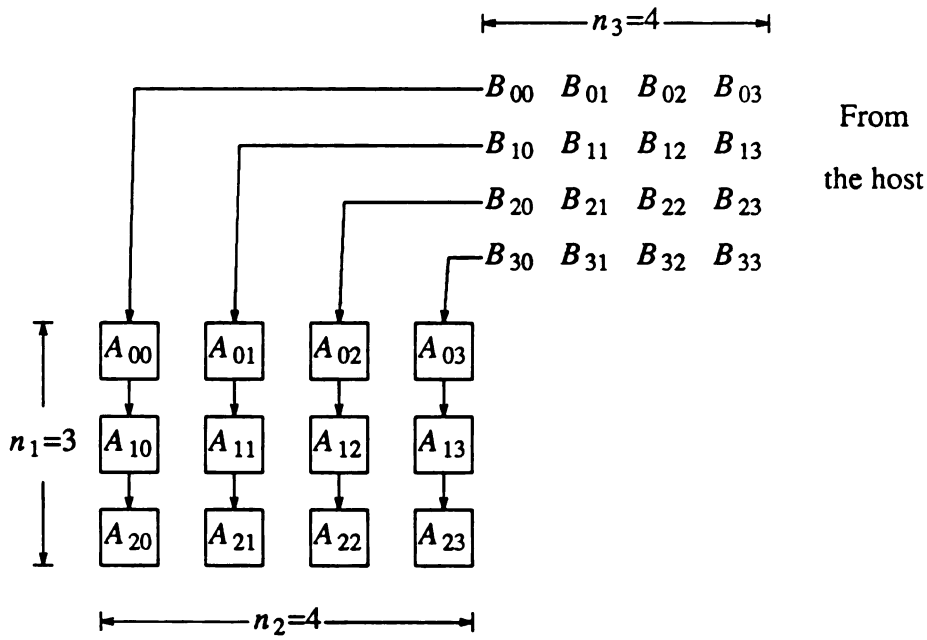
Host:

1.1. send A_{ij} to $node_{ij}$, $0 \le i \le n_1-1$, $0 \le j \le n_2-1$;
1.2. send B_{jk} to $node_{0j}$, $0 \le j \le n_2-1$, $0 \le k \le n_3-1$;
1.3. receive C_i from $node_{i,n_2-1}$, $0 \le i \le n_1-1$.

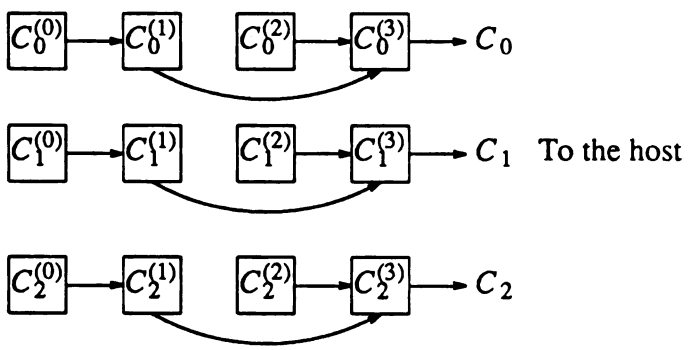
Node: ($node_{ij}$, $0 \le i \le n_1-1$, $0 \le j \le n_2-1$)

2.1. receive A_{ij} from the host;
2.2. for $k := 0$ to n_3-1 do
2.3. if ($i = 0$) then receive B_{jk} from the host
2.4. else receive B_{jk} from $node_{i-1,j}$;
2.5. if ($i \neq n_1-1$) then send B_{jk} to $node_{i+1,j}$;
2.6. $C_{ik}^{(j)} := A_{ij} \times B_{jk}$;
2.7. perform a binary-tree reduction with $node_{il}$, $0 \le l \le n_2-1$, to obtain $C_i := C_i^{(0)} + \ldots + C_i^{(n_2-1)}$ in $node_{i,n_2-1}$, where $C_i^{(l)} := [C_{i0}^{(l)}, \ldots, C_{i,n_3-1}^{(l)}]$;
2.8. if ($j = n_2-1$) then send C_i to the host.

Figure 4.1. Algorithm 4.1 for pipelined matrix multiplication



(a) submatrix multiplication



(b) submatrix addition

Figure 4.2. Data flows in Algorithm 4.1

It can be seen from Figure 4.2(a) that each pipeline will receive a stream of B submatrices from the host, and each stream consists of n_3 data blocks, i.e., submatrices of B. After all submatrices of B have passed through the pipelines, each processor will have collected a portion of a submatrix of C with size $(M/n_1)\times M$. To obtain the final result, a binary-tree reduction is performed among the processors in the same row (Figure 4.2(b)). Thus, there are two phases in each iteration: *submatrix multiplication* for evaluating $A_{ij}\times B_{jk}$ and *submatrix addition* for accumulating C_{ik}, where $0\le i<n_1$, $0\le j<n_2$, and $0\le k<n_3$. In the submatrix multiplication phase, each column of nodes forms one pipeline to disseminate submatrices of B. Thus, there are n_2 pipelines in the system, and each has n_1 stages.

The pipelining concept can be further applied in the reduction, in which each $C_i^{(l)}$ is further partitioned into n_4 smaller sub-submatrices. Then these small sub-submatrices are piped through the reduction tree to accumulate the final results. Note that the granularity (the size of data blocks) plays a very important role. The smaller the granularity, the higher the degree of overlapping. However, since there is a fixed overhead associated with the setup of each message, a small granularity results in a large number of messages and a high communication cost. The choice of a suitable granularity is a very important issue and will be discussed in Chapter 5.

Note that when downloading, the host can feed all the pipelines in a round-robin fashion. In this way, even though the downloading is performed sequentially, operations among the processors can overlap. However, the major problem with Algorithm 4.1 is that each processor has to store a large amount of intermediate data, i.e., submatrices $C_{i0}^{(l)}$, ..., $C_{i,n_3-1}^{(l)}$. To reduce the amount of memory storage, we can intermix the submatrix multiplication and the submatrix addition phase, as in Algorithm 4.2 shown in Figure 4.3. In Algorithm 4.2, as soon as a C submatrix is generated, a binary-tree reduction is invoked among the row processors to accumulate the submatrix. This complete C submatrix is then uploaded back to the host. No intermediate submatrices of C

<Algorithm 4.2>:

Host:

1.1. send A_{ij} to $node_{ij}$, $0 \le i \le n_1-1$, $0 \le j \le n_2-1$;
1.2. send B_{jk} to $node_{0j}$, $0 \le j \le n_2-1$, $0 \le k \le n_3-1$;
1.3. receive C_{ik} from $node_{i,n_2-1}$, $0 \le i \le n_1-1$, $0 \le k \le n_3-1$.

Node: ($node_{ij}$, $0 \le i \le n_1-1$, $0 \le j \le n_2-1$)

2.1. receive A_{ij} from the host;
2.2. for $k := 0$ to n_3-1 do
2.3. if ($i = 0$) then receive B_{jk} from the host
2.4. else receive B_{jk} from $node_{i-1,j}$;
2.5. if ($i \neq n_1-1$) then send B_{jk} to $node_{i+1,j}$;
2.6. $C_{ik}^{(j)} := A_{ij} \times B_{jk}$;
2.7. perform a binary-tree reduction with $node_{il}$, $0 \le l \le n_2-1$, to obtain $C_{ik} := C_{ik}^{(0)} + \ldots + C_{ik}^{(n_2-1)}$ in $node_{i,n_2-1}$;
2.8. if ($j = n_2-1$) then send C_{ik} to the host.

Figure 4.3. Algorithm 4.2 for pipelined matrix multiplication

need to be stored in the processors. The corresponding data flows are described in Figure 4.4.

Note that, in Algorithm 4.2, the size of the data blocks involved in the tree reduction is smaller than that in Algorithm 4.1. Therefore, the balance between granularity and communication overhead becomes even more important. If the communication cost is too high, then we can combine two or more submatrices of C into a larger data block in the reduction. In other words, the tree reduction is performed every two or more iterations instead of one. By increasing the granularity, we can reduce the effect of communication overhead in setting up messages.

Binary-tree reduction is not the only way to accumulate submatrices of C. The algorithm using a linear reduction scheme is described in Figure 4.5. After calculated a C submatrix, each processor sends that C submatrix to its right neighbor. The accumulation progresses from left to right in a linear fashion until the complete C submatrix is obtained in the right-most processor, where it is uploaded back to the host. The corresponding data flows are illustrated in Figure 4.6.

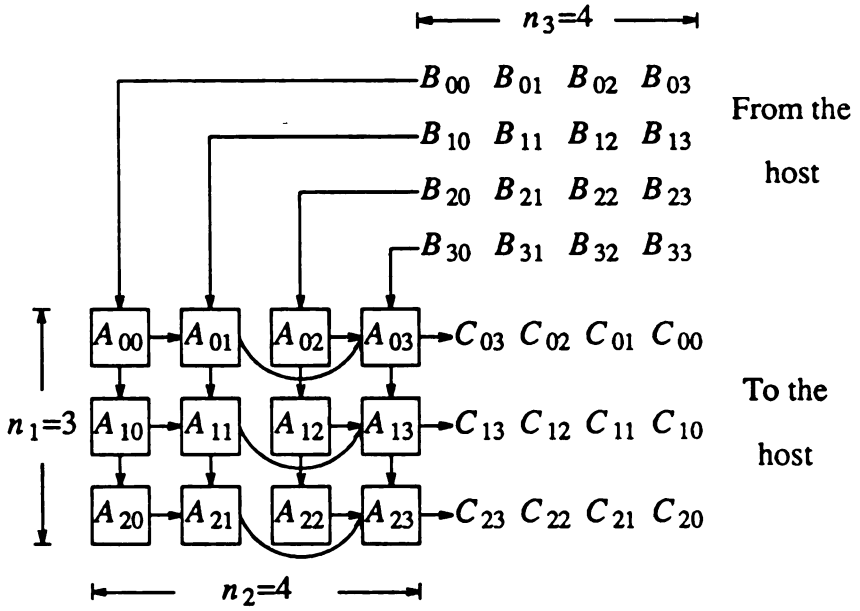


Figure 4.4. Data flows in Algorithm 4.2

Note that in Algorithm 4.3 a processor will perform a submatrix multiplication before it pauses to wait for the C submatrix from the left neighbor (Statement 2.6 - 2.9 in Figure 4.5). This is because submatrix multiplication is a computation intensive operation ($O(n^3)$). If every processor waits until it receives the C submatrix from the left neighbor to perform its multiplication, then the delay within each stage will be very long. Performance analyses of these algorithms are given in the Section 4.4, where considerations for designing pipelined data parallel algorithms will be discussed further.

4.3. CHARACTERISTICS OF LARGE-GRAIN PIPELINING

The most important characteristic of pipelined data parallel computation on DMMs is the concept of large-grain data flows. Through this concept, LGP exploits pipelined data parallel computation from the following two aspects. First, data flows,

<Algorithm 4.3>:

Host:

1.1. send A_{ij} to $node_{ij}$, $0 \le i \le n_1-1$, $0 \le j \le n_2-1$;
1.2. send B_{jk} to $node_{0j}$, $0 \le j \le n_2-1$, $0 \le k \le n_3-1$;
1.3. receive C_{ik} from $node_{i,n_2-1}$, $0 \le i \le n_1-1$, $0 \le k \le n_3-1$.

Node: ($node_{ij}$, $0 \le i \le n_1-1$, $0 \le j \le n_2-1$)

2.1. receive A_{ij} from the host;
2.2. for $k := 0$ to n_3-1 do
2.3. if ($i = 0$) then receive B_{jk} from the host
2.4. else receive B_{jk} from $node_{i-1,j}$;
2.5. if ($i \neq n_1-1$) then send B_{jk} to $node_{i+1,j}$;
2.6. $C_{ik}^{(j)} := A_{ij} \times B_{jk}$;
2.7. if ($j \neq 0$) then
2.8. receive $C_{ik}^{(j-1)}$ from $node_{i,j-1}$
2.9. $C_{ik}^{(j)} := C_{ik}^{(j)} + C_{ik}^{(j-1)}$;
2.10. if ($j = n_2-1$) then send $C_{ik}^{(j)}$ to the host
2.11. else send $C_{ik}^{(j)}$ to $node_{i,j+1}$.

Figure 4.5. Algorithm 4.3 for pipelined matrix multiplication

data are partitioned into *blocks*. The flows of data blocks along the processors form data streams. The size of data blocks determines the granularity of the algorithm and is usually large on DMMs to offset the communication overhead. The data blocks are the most basic unit of processing. In Figure 4.2, each submatrix of B or C forms a data block.

Second, the flows of data connect the nodes in the system into a network. The network consists of multiple pipelines operating on multiple data streams. Each node may participate in several pipelines during the course of computation. The basic mode of operations in a processor is to receive one data block from each of the input streams and generate one data block to each of the output streams. This multiple pipelining scheme results in both spatial (multiple units) and temporal (pipelines) overlapping. In Figure 4.2, there are two kinds of data streams flowing in the system: one along the columns to disseminate submatrices of B, and the other along the rows to accumulate submatrices of C.

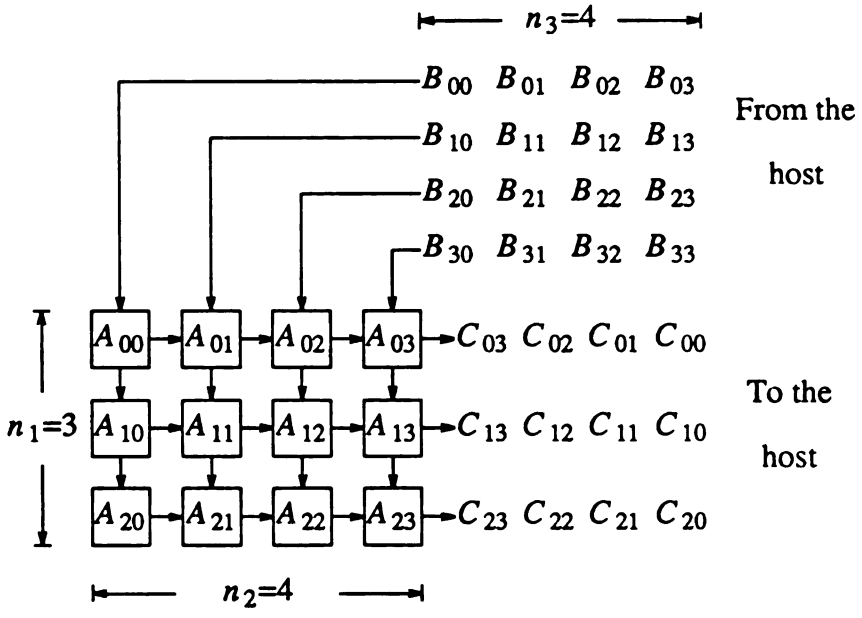


Figure 4.6. Data flows in Algorithm 4.3

From the viewpoint of data flows and the characteristics of DMMs, we can summarize the following important features of pipelined data parallel computations:

(1) **Optimized performance**: Performance of pipelined data parallel algorithms can be fine-tuned by controlling such design parameters as the number of pipelines, the number of stages in the pipelines, and the size of data blocks. In the next chapter, an analytic model will be introduced to serve this optimization purpose. Note that, due to the communication overhead in DMMs, granularity in pipelined data parallel algorithms is usually large.

(2) **Regular and local communication**: Pipelined data parallel algorithms usually have regular data flows. Each node in the system only has to interact with its neighbors (i.e., predecessors and successors in the pipeline).

(3) **Asynchronous and data-driven operations**: Due to the autonomous nature of DMMs, no global synchronization should be used in pipelined data parallel

algorithms. Operations in the system are initiated by the arrivals of data blocks. In this regard, pipelined data parallel algorithms on DMMs are very similar to *wave-front arrays* [KuLJ87]. Note that, to control the flow between two successive nodes (stages) in the pipeline, local memory in the nodes can serve as the buffer between the producer and the consumer.

(4) **Problem size independence**: In DMMs, each node, being a general-purpose processor with local memory, is able to handle data sets with different sizes. Thus, given a fixed node configuration, one can change the size of data blocks to accommodate variable problem sizes.

(5) **Reduced I/O bandwidth**: The number of active channels that the host has to maintain in data downloading and uploading is reduced by using pipelining. For example, in Figure 4.4, the host only has to communicate with nodes in the top row when downloading and with nodes in the right-most column when uploading.

(6) **Identical node programs**: Nodes in a pipelined data parallel algorithm usually perform the same function but on different data sets. This makes the development of pipelined programs easy.

Given the basic idea of LGP, we will study in the next section the performance of the pipelined matrix multiplication algorithm, when they are executed on the NCUBE.

4.4. PERFORMANCE OF PIPELINED MATRIX MULTIPLICATION

The design parameters n_1 and n_2 determine the configuration of the pipelines, where n_2 gives the number of pipelines in the submatrix multiplication phase and n_1 gives the number of stages in each pipeline. If the number of nodes (N) is fixed, then we only have to consider either n_1 or n_2, because $N=n_1n_2$. Figure 4.7 illustrates the relationship between the column partition (n_1) and the execution time (T) for the three algorithms introduced in Section 4.2. Multiplications of 64×64 matrices are studied

using 2 to 32 nodes. The optimal configuration for a given N can be found from the corresponding curve by choosing its minimum point.

It can be seen that Algorithms 4.2 and 4.3 have the same performance around the optimal regions, and both are superior to Algorithm 4.1. This is because the latter has a reduction tree which involves large submatrices. Note that Algorithm 4.3 is expected to perform worse when n_2 is large, due to a long linear path to accumulate C submatrices. However, from Figure 4.7, we can see that Algorithm 4.3 performs as good as Algorithm 4.2 does. A reasonable explanation is that operations in the processors can be fully overlapped in Algorithm 4.3.

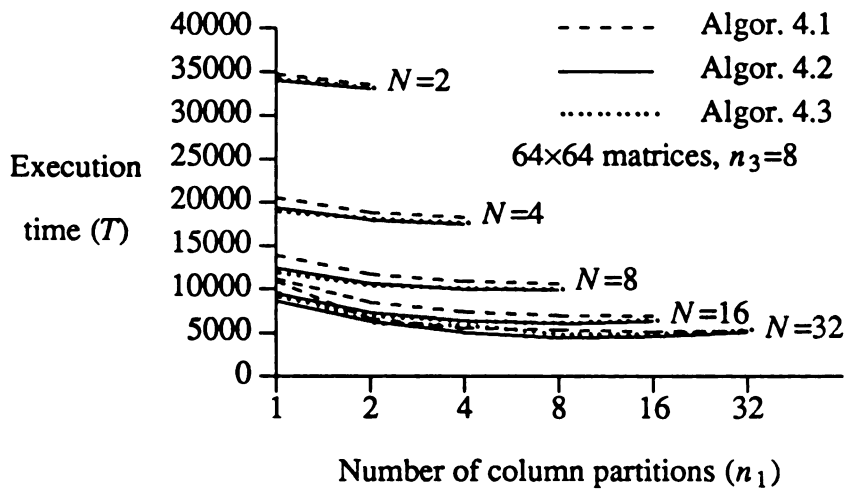


Figure 4.7. Comparison of pipelined matrix multiplication

It can also be seen in Figure 4.7 that a close to square partition always results in good performance [FoOH87]. However, due to the host downloading and uploading operations, the algorithm favors partitions with few pipelines, i.e., a large n_1. Note that, when n_1 is small, C submatrices will be large, and so will be the reduction tree. In this case, the communication overhead is too high for efficient execution.

Parameter n_3 determines the size of data blocks (B_{ij}). The effect of block sizes on the performance of the algorithm is illustrated in Figure 4.8 using Algorithm 4.2. It is evident from Figure 4.8 that granularity does have a bearing on the algorithm performance. A large granularity forces the operations in the nodes to be executed nearly sequentially, which reduces the effect of pipelining between nodes. On the other hand, a small granularity can increase the degree of pipelining, but, due to the fixed overhead in transmitting messages, communication cost will also rise. Though it is not shown, the best partition size in this case is around n_3=16.

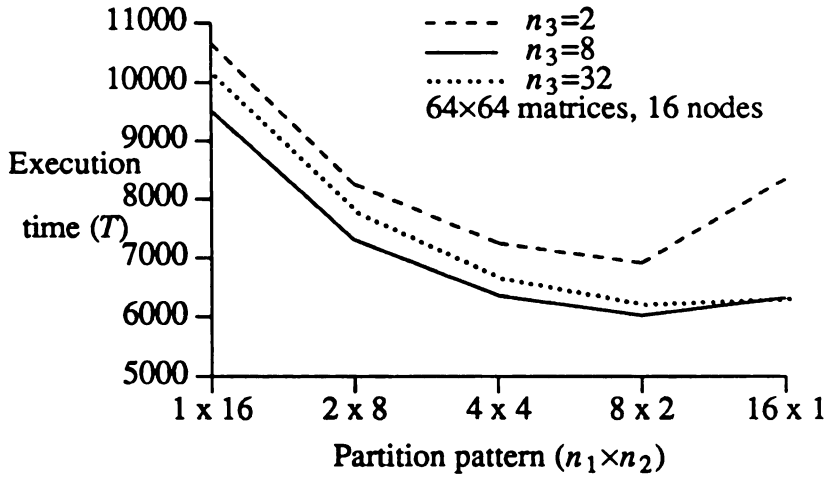


Figure 4.8. Effects of partition sizes

Next, we compare the pipelined matrix multiplication algorithm (Algorithm 4.2) with Algorithm 3.1 introduced in Section 3.4, which was developed without using the concept of pipelining. In Algorithm 3.1, both submatrices of A and B are initially loaded into the nodes. Due to the operation of the ring shifting in Figure 3.12, nodes in the same column are synchronized to the slowest one. The measured speedups are plotted in Figure 4.9, where the speedup of an algorithm is defined in (3.18).

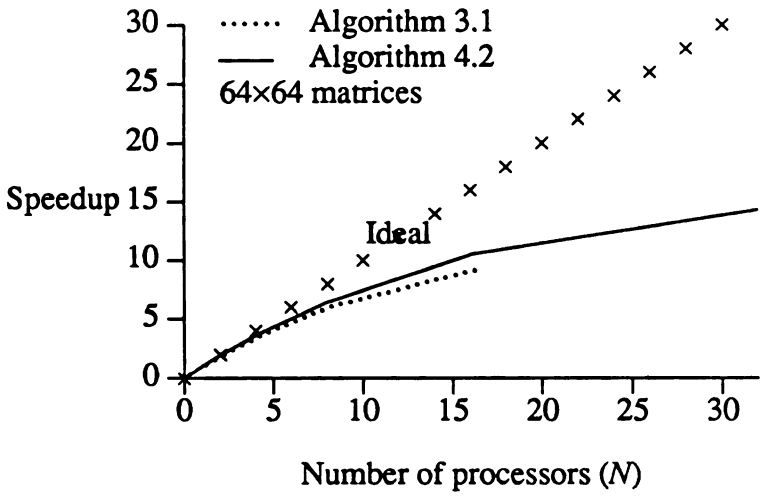


Figure 4.9. Comparison of speedups

Note that the pipelined algorithm performs better than the non-pipelined algorithm does. This is because pipelined data parallel algorithms can better utilize the overlapped operations and balance the computation with communication through the choice of optimal design parameters. However, the improvement is not as significant as expected. The major reason is because of the communication overhead in current generation hypercube multiprocessors, e.g., NCUBE. The overhead in setting up a message is still too high, during which time the host cannot perform any useful computation (see the discussion in Section 3.2).

Given the basic concept of pipelined data parallel computation, the most important issue, then, is to characterize its performance. An analytic model of pipelined data parallel algorithms will be introduced in Chapter 5, including performance analysis of the pipelined matrix multiplication from the analytic perspective.

CHAPTER 5

MODELING PIPELINED DATA PARALLEL ALGORITHMS

The primary strength of large-grain pipelining lies in its ability, through pipelining, to balance computation with communication in the resultant pipelined data parallel algorithms. Nevertheless, this strength cannot be unleashed unless there is a way of deciding the optimal design parameters of the given algorithm, such as

- The number of pipelines used
- The number of stages in each pipeline
- The number of data blocks in each data stream
- The size of data blocks

In general, analytic modeling of an algorithm serves two purposes. First, through the model, the performance of the algorithm can be predicted and studied under various environments. Second, by optimizing the estimated performance, one can determine the optimal set of design parameters. In this way, no expensive trial-and-error is required in determining the parameters, and the design can cope with changes in problem size or even the underlying machine architecture. We have presented in Chapter 3 a concise model for DMMs in general and the NCUBE multiprocessor in particular. Important system parameters of the NCUBE are given in Table 3.1. Therefore, in this chapter we will concentrate on modeling the logical behavior of a pipelined data parallel algorithm.

In Section 5.1, a model of pipelined computation is presented which focuses on a single stage in the pipeline [KiCN88b]. Throughput of a stage is analyzed in Section 5.2, and the model is extended in Section 5.3 to analyze the throughput of linear arrays and 2-dimensional meshes. As an illustration, in Section 5.4, the model is used to analyze the performance of the pipelined matrix multiplication algorithm (Algorithm 4.2). Accuracy of the analytic model is studied by comparing the predicted performance with that measured on the NCUBE.

5.1. A MODEL OF PIPELINED COMPUTATION

Multiple pipelining is a general pattern of computation in pipelined data parallel algorithms. Following the flow concept, nodes in a pipelined data parallel computation can be modeled as a *computational unit* which has multiple input and output streams (see Figure 5.1). Each data stream consists of many data blocks. From this point of view, a pipelined data parallel algorithm can thus be modeled as a network of computational units linked together by data streams as exemplified in Figure 4.6. To analyze such a network, the relationship between input and output streams has to be studied first.

An *m−input computational unit* in a pipelined data parallel computation can be viewed as a process, which takes inputs from m data streams, ids_i $(0 \le i \le m-1)$, performs a function, $ods = h(ids_0, ..., ids_{m-1})$, and generates another data stream, *ods*, as an output. Here only a single output data stream is considered. The result can be easily applied to the case of multiple output data streams by adding constant offsets. Suppose each input stream contains n data blocks. Then, the computational unit has to repeat a *receive-compute-send* sequence n times to complete the task. This iterative process is described in Figure 5.2.

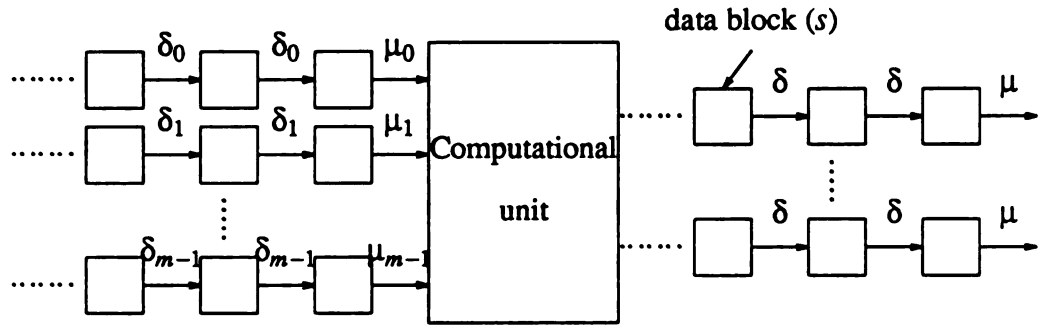


Figure 5.1. Model of a computational unit

```
function h( ) {
   /* I: begin of computation */
   start_up( );  /* initialization time: α */
   for j:=0 to n−1 do {
      recv (ids0, data0, .....);
      P0j: data_handling (data0);  /* handling time: f0 */
        ....
      recv (idsm−1, datam−1, .....);
      Pm−1,j: data_handling (datam−1);  /* handling time: fm−1 */
      Gj: data=computation (data0, ..., datam−1);  /* computation time: c */
      send (ods, data, .....);
   }
   /* E: end of computation */
}
```

Figure 5.2. Functional description of the computational unit

Given the functional description of a computational unit, we are interested in the evaluation of its throughput. Before we can proceed, the following notions should be clarified.

<Definition 5.1>

A *data stream*, *ds*, is characterized by a two-tuple (μ, δ), where

μ = the arrival time of the first data block

δ = the interarrival time between two consecutive data blocks.

□

μ is an absolute time value measured from a fixed reference time instant, usually the instant the whole computation begins. Note that, although one can identify other parameters of data streams, only μ and δ affect the modeling discussed here. Note also that δ is, in general, a function of the size of data blocks and the delay induced by nodes upstream. Again, for the purpose of modeling, δ is taken to be a constant by, say, averaging over the interarrival time between each pair of consecutive data blocks.

<Definition 5.2>

A computational unit, *cu*, is characterized by

α — the initialization time;

f_i, $0 \le i \le m-1$ — the time required to handle a data block from input stream ids_i;

c — the time spent in computation and invoking the *send()* request in each iteration.

□

Note that α is an absolute time value, while f_i and c are time intervals.

Given the set of parameters for the m input streams, (μ_i, δ_i), $0 \le i < m$, and assuming that (α, c, f_i) are known, we are interested in the throughput of the m-input computational unit, i.e., (μ, δ) of its output stream. Note that μ is the time the first output data block arrives at the subsequent computational unit, and δ is equal to the loop iteration time of the function $h(.)$. Detailed analysis is given in the next section.

5.2. THROUGHPUT ANALYSIS OF A COMPUTATIONAL UNIT

The timing sequence of a computational unit can be described by a Timed Petri-net, as illustrated in Figure 5.3. Each *place* in the Petri-net represents one time instant,

and each *transition* in the Petri-net represents an event which takes a certain amount of time to finish. Note that, due to the way a computational unit is modeled, the corresponding Timed Petri-net is acyclic and deterministic.

The following time instants (or places in Figure 5.3) are of interest to us, where $0 \le i \le m-1$, $0 \le j \le n-1$:

I — The starting instant of the computational unit.

X_i — The starting instant of sending data stream ids_i.

P_{ij} — The instant that *recv()* for the j-th data block from data stream ids_i is issued.

G_j — The instant that *computation()* in the j-th iteration starts.

R_{ij} — The instant that the j-th data block from data stream ids_i arrives.

D_{ij} — The instant that data stream ids_i is ready to deliver the $(j+1)$-th data block.

E — The instant that the whole computation is done.

Thus, for example, the time instant P_{ij} indicates that at the j-th iteration, the data block from stream ids_{i-1} has been handled, and that the computational unit is ready to receive the data block from ids_i. The procedure, *data_handling* ($data_i$), can start depending on which event (P_{ij} or R_{ij}) occurs last.

From the above discussion, we can observe that the time a place, $v \in \{P_{ij}, G_j, E \mid 0 \le i \le m-1, 0 \le j \le n-1\}$, will be reached is determined by the longest path among all paths from $\{X_i, I \mid 0 \le i \le m-1\}$ to v. The *length* of a path is defined to be the sum of the transition intervals on the path. For example, the time the place P_{20} will take place is equal to

$$\text{Max}\{\alpha+f_0+f_1,\ \mu_0+f_0+f_1,\ \mu_1+f_1\}$$

Let $\phi_i=f_i+\ ...+f_{m-1}$ denote the length of the path from P_{ij} to G_j, where $0 \le i \le m-1$ and $0 \le j \le n-1$. Let T_j be the time the place G_j, $0 \le j \le n-1$, can take place, i.e., the time in the j-th iteration that all input blocks have been received and that the function *computation()* is to be executed. We have

$$T_j = \max_{0 \le k \le j, 0 \le i \le m-1} \{\alpha+\phi_0+j(\phi_0+c),\ \mu_i+k\,\delta_i+\phi_i+(j-k)(\phi_0+c)\}$$

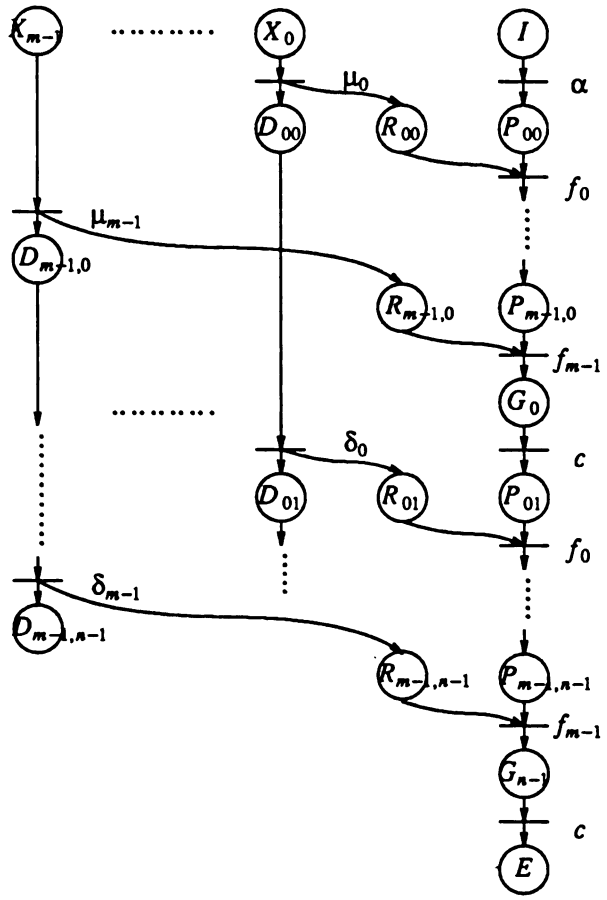


Figure 5.3. Timed Petri-net of a computational unit

$$= \max_{0\le k\le j, 0\le i\le m-1} \{\alpha+\phi_0,\ \mu_i+\phi_i+k(\delta_i-(\phi_0+c))\}+j(\phi_0+c)$$

However, in the above expression, in order to obtain the maximum value for T_j, k can only take two possible values: If $\delta_i \le (\phi_0+c)$, then $k=0$; otherwise, $k=j$. Define

$$\kappa(x) = \begin{cases} 1 & x \ge 0 \\ 0 & x < 0 \end{cases}$$

Consequently, we have the following lemma.

<Lemma 5.1>

The time the place G_j, $0\le j\le n-1$, will be reached is equal to

$$T_j = \max_{0\le i\le m-1} \{\alpha+\phi_0,\ \mu_i+\phi_i+j\,\kappa(\delta_i-(\phi_0+c))\} + j(\phi_0+c) \tag{5.1}$$

□

From Lemma 5.1, the parameters (μ, δ) of the output data stream can be determined. For μ, we have the following theorem:

<Theorem 5.1>

Let ψ be the communication delay for sending one output data block to the subsequent computational unit. Then,

$$\mu = \max_{0\le i\le m-1} \{\alpha+\phi_0,\ \mu_i+\phi_i\}+c+\psi \tag{5.2}$$

<Proof>

The time that the subsequent computational unit will receive the j-th output data block is $T_j+c+\psi$. From (5.1) we can see that, when j is 0, $T_0 = \max_{0\le i\le m-1} \{\alpha+\phi_0,\ \mu_i+\phi_i\}$. Thus, we obtain (5.2).

□

Unfortunately, δ cannot be represented in such a simple form as μ. If we take $\Delta_j = T_j - T_{j-1}$, $1\le j<n$, then Δ_j would not be a constant — instead Δ_j is (ϕ_0+c) plus some arbitrary delay time. Thus we can only estimate a constant δ to approximate Δ_j. Since our major concern is the completion time of a computational unit, we will use

$$\delta \equiv \frac{T_{n-1}-T_0}{n-1} \tag{5.3}$$

In summary, given the parameters, $(\alpha, f_i, c, \mu_i, \delta_i)$, $0 \le i \le m-1$, the throughput of an m-input computational unit can be determined from (5.2) and (5.3). Note that, in a pipelined data parallel computation, the output data of a computational unit become the input data of the subsequent computational units. Thus, by tracing through the flow of data streams and applying (5.2) and (5.3) repetitively, the throughput and execution time of the whole computation can be derived. As an illustration of the analytic model, let us consider the execution of a linear pipeline with only one input stream, as shown in Figure 5.4.

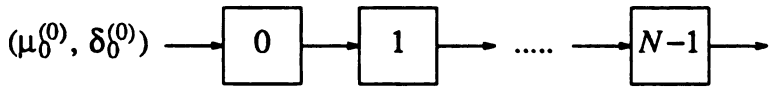


Figure 5.4. A linear array with single input stream

<Example 5.1>

Assume that there are N stages in the linear array. The single input stream to stage 0 has an arrival time of $\mu_0^{(0)}$ and a block interarrival time of $\delta_0^{(0)}$. There are n data blocks in the stream. Each stage of the array is viewed as a computational unit which has parameters α, f_0, and c, for process initialization time, block handling time, and block computation time, respectively (see Definition 5.2). Consider stage 0. From (5.1), we have

$$T_0^{(0)} = \text{Max}\{\alpha+\phi_0,\ \mu_0^{(0)}+\phi_0\}$$

$$T_{n-1}^{(0)} = \text{Max}\{\alpha+\phi_0,\ \mu_0^{(0)}+\phi_0+(n-1)\kappa(\delta_0^{(0)}-(\phi_0+c))\} + (n-1)(\phi_0+c)$$

The throughput of this stage can be derived from (5.2) and (5.3), which give the following parameters for the output stream:

$$\mu_0^{(1)} = T_0^{(0)} + c + \psi = \text{Max}\{\alpha, \mu_0\} + \phi_0 + c + \psi$$

$$\delta_0^{(1)} = \text{Max}\{\phi_0 + c,\ \delta_0^{(0)} - \frac{\kappa(\alpha - \mu_0^{(0)})}{n-1}\}$$

The analysis for other stages are similar. In general, for stage i, $0 \le i \le N-1$, we have

$$T_0^{(i)} = \text{Max}\{\alpha+\phi_0,\ \mu_0^{(i)}+\phi_0\} = \text{Max}\{\alpha, \mu_0\} + i(\phi_0 + c + \psi) + \phi_0$$

$$T_{n-1}^{(i)} = \text{Max}\{\alpha+\phi_0,\ \mu_0^{(i)}+\phi_0+(n-1)\kappa(\delta_0^{(i)} - (\phi_0+c))\} + (n-1)(\phi_0+c)$$

$$= \mu_0^{(i)} + \phi_0 + (n-1)\delta_0^{(i)}$$

because $\delta_0^{(i)} \ge (\phi_0 + c)$. Thus, the output stream from stage i has the following parameters

$$\mu_0^{(i+1)} = \text{Max}\{\alpha, \mu_0\} + (i+1)(\phi_0 + c + \psi)$$

$$\delta_0^{(i+1)} = \delta_0^{(i)} = \delta_0^{(1)}$$

It follows that the first output data block will be generated by stage $N-1$ (and, thus, by this linear array) at

$$\text{Max}\{\alpha, \mu_0\} + (N-1)(\phi_0 + c + \psi) + (\phi_0 + c)$$

with an output interval of $\delta_0^{(1)}$. The last output block will be generated by stage $N-1$ at

$$\text{Max}\{\alpha, \mu_0\} + (N-1)(\phi_0 + c + \psi) + (\phi_0 + c) + (n-1)\delta_0^{(1)}$$

which equals the total execution of this linear array. The above expression conforms to the general execution time of a pipeline [KwBr84],

$$(N + n - 1) \times \text{clock time}$$

where N is the number of stages in the pipeline and n is the number of data elements passed through the pipeline.

□

Note that the model presented in this section only describes the behavior of a single computational unit. However, as can be seen in Example 5.1, the behavior of computational units in regular arrays are very similar. Thus, it is possible to derive a

closed-form to characterize the performance of such regular arrays. The next section expands the analytic model presented here to study the throughput of linear arrays and 2-dimensional meshes.

5.3. THROUGHPUT ANALYSIS OF PIPELINE ARRAYS

Suppose computational units are connected in a linear array or 2-dimensional mesh. In this section, we are interested in obtaining closed-form expressions for the performance of the whole array. Note that the throughput at each single stage can be derived using the techniques presented in the previous section. The throughput of a linear array with only one input stream has been studied in Example 5.1. Let us consider linear arrays with two input streams, as shown in Figure 5.5.

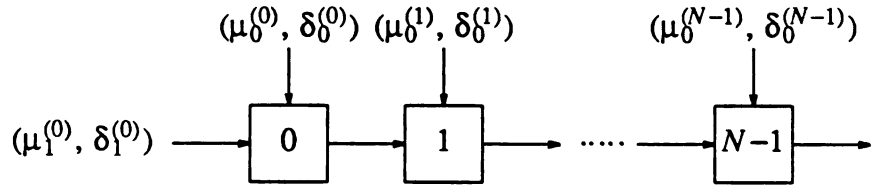


Figure 5.5. A linear array with two input streams

(1) Linear arrays with two input streams

Again, there are N stages in the array. Each stage has two input streams: one from the outside world and one from the previous stage (except stage 0). Assume that the arrival times of data streams have the following relation: $\mu_0^{(i)}=\mu_0^{(i-1)}+\delta$, and that $\delta_0^{(i)}=\delta_0^{(0)}$, for $0<i\leq N-1$. Again, there are n data blocks in each data stream. The events that occur in the stages are depicted in Figure 5.6 using the Timed Petri-net with $N=2$ and $n=2$.

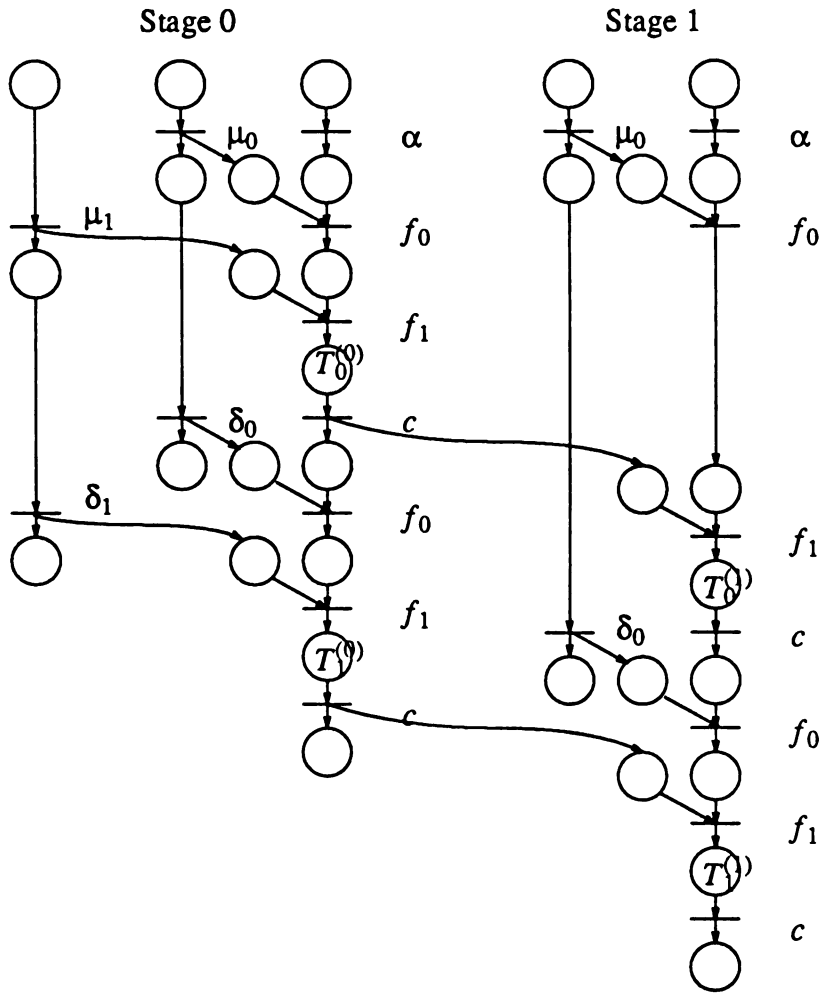


Figure 5.6. The Timed Petri-net of a linear array with two stages

Consider $T_k^{(i)}$, $0 \le i \le N-1$, $0 \le k \le n-1$ — the time the i-th stage has received both input data blocks in the k-th iteration (see Figure 5.6). Again, $T_k^{(i)}$ is dependent upon the longest path from any initial place in the Timed Petri-net to its corresponding place. By inspecting the interaction between stage i and $i-1$ (see Figure 5.6), we can identify the following relation:

$$T_k^{(i)} = \max_{0 \le j \le k} \{ \alpha + \phi_0 + k(\phi_0 + c),\ \mu_0^{(i)} + \phi_0 + k \max\{\delta_0^{(i)}, (\phi_0 + c)\},\ T_j^{(i-1)} + (c + \phi_1 + \psi) + (k-j)(\phi_0 + c) \} \tag{5.4}$$

The first two terms are obtained directly from (5.1). The remaining terms in (5.4) can be simplified by noting that $T_j^{(i)} \ge T_{j-1}^{(i)} + \phi_0 + c$. This is true because from stage 0 in Figure 5.6, it can be seen that $T_0^{(0)} + \phi_0 + c$ is just the length of one of the paths incident to the place labeled $T_1^{(0)}$. However, $T_1^{(0)}$ is equal to the maximum of the lengths of all paths incident to its corresponding place. It follows that $T_k^{(i)}$ can be rewritten as follows:

$$T_k^{(i)} = \max\{ \alpha + \phi_0 + k(\phi_0 + c),\ \mu_0^{(0)} + i\delta + \phi_0 + k \max\{\delta_0^{(0)}, (\phi_0 + c)\},\ T_k^{(i-1)} + (c + \phi_1 + \psi) \}$$

where $\mu_0^{(i)}$ is replaced by $\mu_0^{(0)} + i\delta$. To simplify the expressions, we introduce the following symbols:

$$Z = \alpha + \phi_0 + k(\phi_0 + c)$$
$$U = \mu_0^{(0)} + \phi_0 + k \max\{\delta_0^{(0)}, \phi_0 + c\}$$
$$X = \phi_1 + c + \psi$$

Thus, we have

$$\begin{aligned} T_k^{(i)} &= \max\{Z,\ U + i\delta,\ T_k^{(i-1)} + X\} \\ &= \max\{Z + X,\ U + i\delta,\ U + (i-1)\delta + X,\ T_k^{(i-2)} + 2X\} \\ &= \max_{0 \le l \le i-1} \{Z + (i-1)X,\ U + (i-j)\delta + jX,\ T_k^{(0)} + iX\} \end{aligned}$$

However, note that l can only take two values in order to maximize $T_k^{(i)}$: 0 or $i-1$. Also, from (5.1), we have

$$T_k^{(0)} = \text{Max}\{\alpha+\phi_0+k(\phi_0+c),\ \mu_0^{(0)}+\phi_0+k\text{Max}\{\delta_0^{(0)},(\phi_0+c)\},$$
$$\mu_1^{(0)}+\phi_1+k\text{Max}\{\delta_1^{(0)},(\phi_0+c)\}\}$$

Combined together, we obtain an expression for $T_k^{(i)}$:

$$T_k^{(i)} = \text{Max}\{Z+iX,\ U+i\text{Max}\{\delta,X\},\mu_1^{(0)}+\phi_1+k\text{Max}\{\delta_1^{(0)},(\phi_0+c)\}\}$$
$$= \text{Max}\{\alpha+\phi_0,\ \mu_0^{(0)}+\phi_0+k\,\kappa(\delta_0^{(0)}-(\phi_0+c))+i\,\kappa(\delta-(\phi_1+c+\psi)),\qquad (5.5)$$
$$\mu_1^{(0)}+\phi_1+k\,\kappa(\delta_1^{(0)}-(\phi_0+c))\} + k(\phi_0+c) + i(\phi_1+c+\psi)$$

Notice the similarity between (5.5) and (5.1). In fact, if $i=0$, then Equations (5.1) and (5.5) are identical. From (5.5), the total execution time of the whole array is equal to $T_{n-1}^{(N-1)}+c$.

(2) 2-dimensional mesh

Figure 5.7 shows the computational units that are connected into a 2-dimensional mesh. Each stage has two inputs, and input streams have the following relations:

$$\mu_0^{(0j)} = \mu_0^{(00)} + j\,\delta_0' \qquad \delta_0^{(0j)} = \delta_0^{(00)}$$
$$\mu_1^{(i0)} = \mu_1^{(00)} + i\,\delta_1' \qquad \delta_1^{(i0)} = \delta_1^{(00)}$$

The corresponding Timed Petri-net is shown in Figure 5.8.

From Figure 5.8, we can see that

$$T_k^{(ij)} = \text{Max}\{\alpha+\phi_0+k(\phi_0+c),\ T_k^{(i-1,j)}+(\phi_0+c+\psi),\ T_k^{(i,j-1)}+(\phi_1+c+\psi)\} \qquad (5.6)$$

The reason that $T_l^{(i-1,j)}$ and $T_l^{(i,j-1)}$, $0\le l<k$, are not considered in (5.6) has been explained as in (1) above. Again, to simplify the expressions, we use the following symbols:

$$X = \phi_0+c+\psi$$
$$Y = \phi_1+c+\psi$$
$$Z = \alpha+\phi_0+k(\phi_0+c)$$
$$U_0 = \mu_0^{(0)}+\phi_0+k\text{Max}\{\delta_0^{(0)},\ \phi_0+c\}$$
$$U_1 = \mu_1^{(0)}+\phi_1+k\text{Max}\{\delta_1^{(0)},\ \phi_0+c\}$$

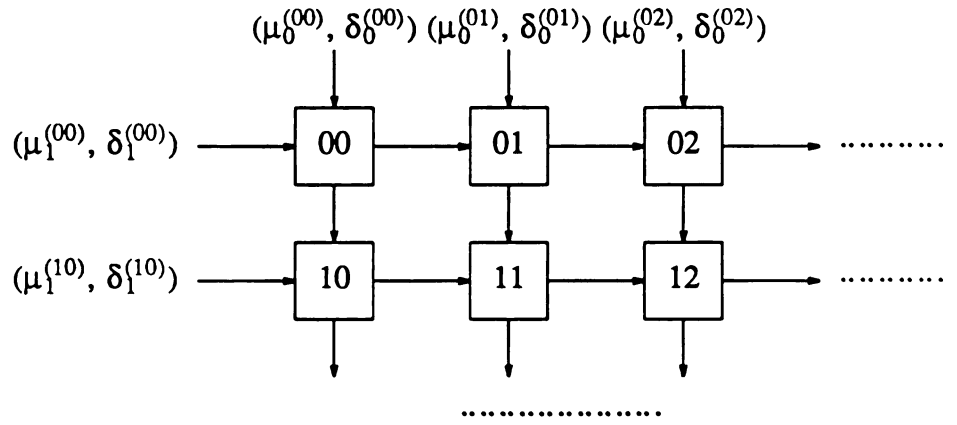


Figure 5.7. Computational units connected in a 2-dimensional mesh

Thus, we have

$$T_k^{(ij)} = \text{Max}\{Z,\ T_k^{(i-1,j)}+X,\ T_k^{(i,j-1)}+Y\}$$

$$= \text{Max}\{Z+Y,\ T_k^{(i-1,j)}+X,\ T_k^{(i-1,j-1)}+X+Y,\ T_k^{(i,j-2)}+2Y\}$$

$$= \max_{0\le l\le j-1}\{Z+(j-1)Y,\ T_k^{(i-1,j-l)}+X+lY,\ T_k^{(i0)}+jY\}$$

However,

$$T_k^{(i0)} = \text{Max}\{Z,\ T_k^{(i-1,0)}+X,\ U_1+i\delta_1'\}$$

Thus, we have

$$T_k^{(ij)} = \max_{0\le l\le j}\{Z+jY,\ T_k^{(i-1,j-l)}+X+lY,\ U_1+i\delta_1'+jY\}$$

$$= \max_{0\le l\le j}\{Z+X+jY,\ T_k^{(i-2,j-l)}+2X+lY,\ U_1+i\delta_1'+jY,\ U_1+(i-1)\delta_1'+X+jY\}$$

$$= \max_{0\le l\le j}\{Z+(i-1)X+jY,\ T_k^{(0,j-l)}+iX+lY,\ U_1+\delta_1'+(i-1)\kappa(\delta_1'-X)+(i-1)X+jY\}$$

However,

$$T_k^{(0j)} = \text{Max}\{Z,\ U_0+j\delta_0',\ T_k^{(0,j-1)}+Y\}$$

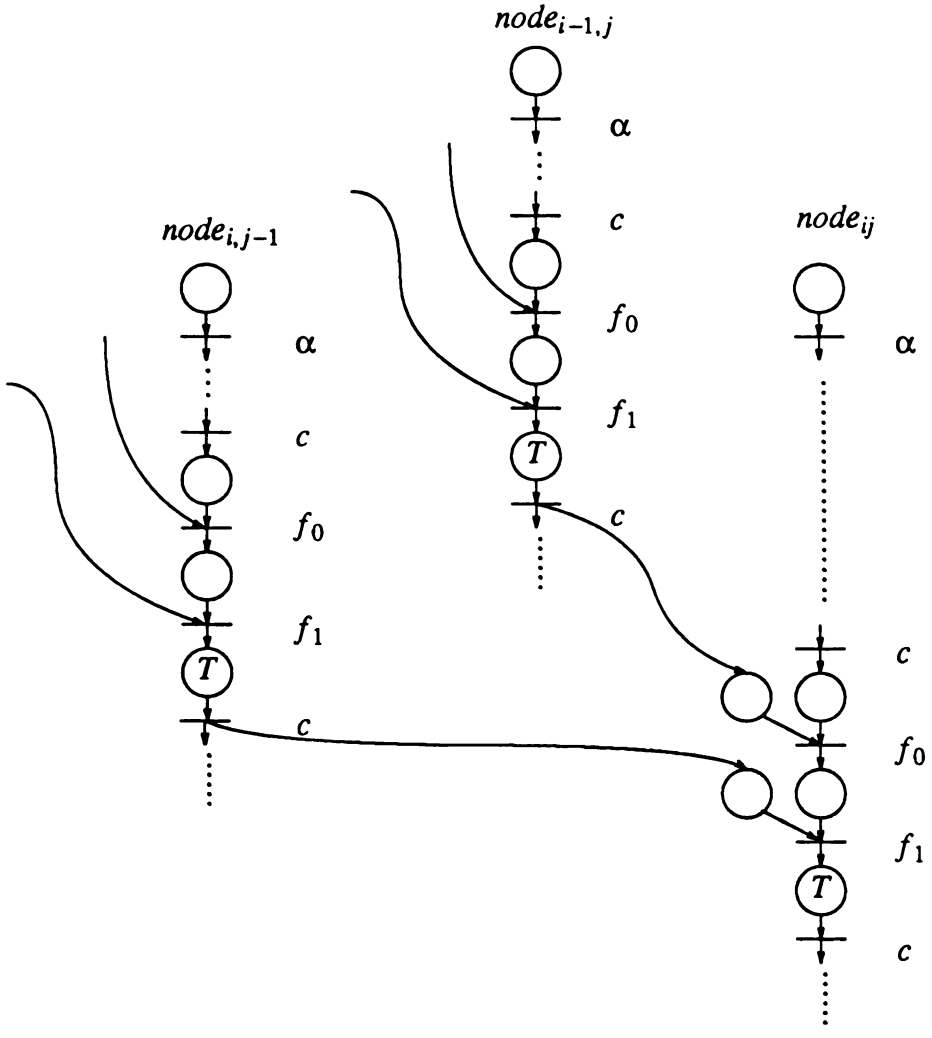


Figure 5.8. The Timed Petri-net of the 2-dimensional mesh

and

$$T_k^{(00)} = \text{Max}\{Z,\ U_0,\ U_1\}$$

It follows that $T_k^{(ij)}$ can be expressed as

$$T_k^{(ij)} = \text{Max}\{\alpha+\phi_0,\ \mu_0^{(0)}+\phi_1+k\,\kappa(\delta_0^{(0)}-(\phi_0+c))+j\,\kappa(\delta_0'-(c+\psi+\phi_1)), \tag{5.7}$$

$$\mu_1^{(0)}+\phi_1+k\,\kappa(\delta_1^{(0)}-(\phi_0+c))i\,\kappa(\delta_1'-(c+\psi+\phi_0))\} + k\,(\phi_0+c)$$

$$+ i\,(\phi_0+c+\psi) + j\,(\phi_1+c+\psi)$$

The total execution time of the 2-dimensional array is then given by $T_{n-1}^{(N-1,N-1)}$. For computational units with other interconnection structures, the technique developed in this section can also be applied to analyze their performance.

5.4. ANALYSIS OF PIPELINED MATRIX MULTIPLICATION

The analytic model presented in the previous section can be directly applied to Algorithm 4.3, which has a 2-dimensional mesh structure. However, due to the tree reduction performed in Algorithm 4.2, the analysis derived in Section 5.3 cannot be used directly. Therefore, in this section, the execution time (T) of Algorithm 4.2 (see Figure 4.3) is derived using the analysis techniques for a single computational unit.

Assume that matrices A, B, and C are all square matrices with size $M\times M$. The analysis can be easily extended to non-square matrices. Matrices A and B are partitioned into $n_1\times n_2$ and $n_2\times n_3$ submatrices, respectively. Assume that n_1, n_2, and n_3 can divide M evenly, and that $N=n_1 n_2$ nodes are used to perform the matrix multiplication. Thus, we have (see Figure 4.4)

$$\text{size of } A_{ij} \text{ is } \frac{bM^2}{n_1 n_2} \text{ for } 0 \le i < n_1, 0 \le j < n_2$$

$$\text{size of } B_{ij} \text{ is } \frac{bM^2}{n_2 n_3} \text{ for } 0 \le i < n_2, 0 \le j < n_3$$

$$\text{size of } C_{ij} \text{ is } \frac{bM^2}{n_1 n_3} \text{ for } 0 \le i < n_1, 0 \le j < n_3$$

where $b = 4$ *bytes* is the size of a floating-point number on the NCUBE. Define the following notations:

$$t_0 = \tau_c \frac{M^3}{n_1 n_2 n_3} \qquad t_1 = \tau_b \frac{M^2}{n_1 n_3}$$

$$\theta_{12} = \tau_h \frac{bM^2}{n_1 n_2} \qquad \psi_{12} = \tau_n \frac{bM^2}{n_1 n_2}$$

$$\theta_{23} = \tau_h \frac{bM^2}{n_2 n_3} \qquad \psi_{23} = \tau_n \frac{bM^2}{n_2 n_3}$$

$$\theta_{13} = \tau_h \frac{bM^2}{n_1 n_3} \qquad \psi_{13} = \tau_n \frac{bM^2}{n_1 n_3}$$

where t_0 is the time to perform a submatrix multiplication ($A_{ij} \times B_{jk}$), and t_b is the time to perform a submatrix addition ($C_{ik}^{(j)} + C_{ik}^{(j+1)}$). Also, following the definitions of θ_{23} and ψ_{23}, the time to transmit one submatrix of B from the host to a node will be $\sigma_h + \theta_{23}$ and that between nodes will be $\sigma_n + \psi_{23}$ (see (3.1) and (3.2)). Other parameters can be similarly interpreted.

The first step in analyzing the performance of a pipelined data parallel algorithm is to identify the critical path of the computation. Note that in Figure 4.4, all nodes at the right-most column are the ones that are loaded from the host the latest and will start the last among all nodes at the same row. Thus, we need only focus on the operations in $node_{i,n_2-1}$, $0 \le i \le n_1 - 1$. Consider $node_{0,n_2-1}$ first. Operations in this node depend on two events: the arrival of a B submatrix and the setup of the receive routine for that submatrix. Note that, before the host can download $B_{n_2-1,0}$ to $node_{0,n_2-1}$, it must send all submatrices of A as well as B_{00}, ..., $B_{n_2-2,0}$ to the corresponding nodes. Thus, $B_{n_2-1,0}$

will arrive at $node_{0,n_2-1}$ at

$$\mu_0^{(0)} = N(\sigma_h+\theta_{12}) + n_2(\sigma_h+\theta_{23})$$

The interval between successive B submatrix arrivals is

$$\delta_0^{(0)} = n_2(\sigma_h+\theta_{23})$$

which is the time the host loads all nodes in the top row exactly once. On the other hand, $node_{0,n_2-1}$ will setup the receive routine for $B_{n_2-1,0}$ only after it has received A_{0,n_2-1} from the host. Thus, this node will be ready for $B_{n_2-1,0}$ at

$$\alpha^{(0)} = n_2(\sigma_h+\theta_{12}) + \psi_{12} + \sigma_n$$

The second term, ψ_{12}, accounts for the time to copy the received A submatrix (A_{0,n_2-1}) from the system buffer, and σ_n is the time to invoke the receive routine for $B_{n_2-1,0}$. From Equation (5.1), the node can start the submatrix multiplication phase at

$$T_0^{(0)} = \text{Max}\{\mu_0^{(0)}, \alpha^{(0)}\} + \phi_0^{(0)}$$

where $\phi_0^{(0)}=f_0^{(0)}=\psi_{23}$ is the time to copy $B_{n_2-1,0}$ from the system buffer. From now on, in each iteration, $node_{0,n_2-1}$ will

(1) Send the received B submatrix to $node_{1,n_2-1}$, which takes time $\sigma_n + \psi_{23}$;

(2) Perform a submatrix multiplication, which takes time t_0;

(3) Perform a tree reduction with all nodes in the top row, which takes time $\log_2 n_2(\sigma_n+2\psi_{13}+t_1)$;

(4) Send the resultant C submatrix to the host, which takes time $\sigma_n + \psi_{13}$;

(5) Issue the *recv()* request to receive a B submatrix from the host, which takes time σ_n.

Let

$$c^{(0)} = (\sigma_n+\psi_{23})+t_0+\log_2 n_2(\sigma_n+2\psi_{13}+t_1)+(\sigma_n+\psi_{13})+\sigma_n$$

The iteration time, $\delta^{(0)}$, at this node can be obtained from Equation (5.6). Specifically,

if $\phi_0^{(0)}+c^{(0)} \geq \delta_0^{(0)}$, we have

$$\delta^{(0)} = \phi_0^{(0)} + c^{(0)}$$

Note that $\delta^{(0)}$ is also the interval between the sending of successive output data blocks (i.e., C submatrices) from $node_{0,n_2-1}$ to the host. The time instant that the first block of this C submatrix stream arrives at the host is equal to

$$\mu^{(0)} = T_0^{(0)} + c^{(0)} - \sigma_n$$

In general, for $node_{i,n_2-1}$, $0 \leq i \leq n_1-1$, the time, $T_0^{(i)}$, that the submatrix multiplication can start is dependent upon (1) when the first B submatrix, $B_{n_2-1,0}$, arrives, i.e.,

$$\mu_0^{(i)} = T_0^{(0)} + (i-1)(\sigma_n+2\psi_{23})$$

and (2) when it receives the A submatrix and is ready for $B_{n_2-1,0}$, i.e.,

$$\alpha^{(i)} = (i+1)n_2(\sigma_h+\theta_{12}) + \psi_{12} + \sigma_n$$

Then, $T_0^{(i)}$, $\mu^{(i)}$, and $\delta^{(i)}$ can be obtained from (5.1), (5.2), and (5.3), respectively. Note that the interarrival time ($\delta_0^{(i)}$) of adjacent input data blocks to $node_{i,n_2-1}$ is equal to the iteration time of $node_{i-1,n_2-1}$ ($\delta^{(i-1)}$).

Now consider the host. The host will finish downloading submatrices of A and B and be ready for the uploading at

$$\alpha = N(\sigma_h+\theta_{12}) + n_2n_3(\sigma_h+\theta_{23}) + \sigma_{hr}$$

When uploading, the host can also be viewed as a computational unit, which has n_1 input streams of C submatrices with parameters ($\mu^{(i)}$, $\delta^{(i)}$), $0 \leq i \leq n_1-1$. After receiving one C submatrix, the host needs an interval of $f_i=\theta_{13} + \sigma_{hr}$ to copy the submatrix from the system buffer and set up the next receive routine. From (5.1) we can see that the host will finish the uploading (which is also the time the job finishes) at

$$T = \max_{0 \leq i \leq n_1-1} \{\alpha+\phi_0, \mu^{(i)}+\phi_i+(n_3-1)(\delta^{(i)}-n_1(\theta_{13}+\sigma_{hr}))\} + (n_3-1)n_1(\theta_{13}+\sigma_{hr})$$

where $\phi_i = (n_1-1)(\theta_{13}+\sigma_{hr}) + \theta_{13}$ and $c = \sigma_{hr}$.

5.5. A COMPARATIVE STUDY OF THE ANALYTIC MODEL

In this section, the execution time derived in Section 5.4 is validated through experimental results obtained from the NCUBE. The close match between analytic and experimental results indicate the accuracy of the model. Note that the analytic results are obtained by using the system parameters listed in Table 3.1.

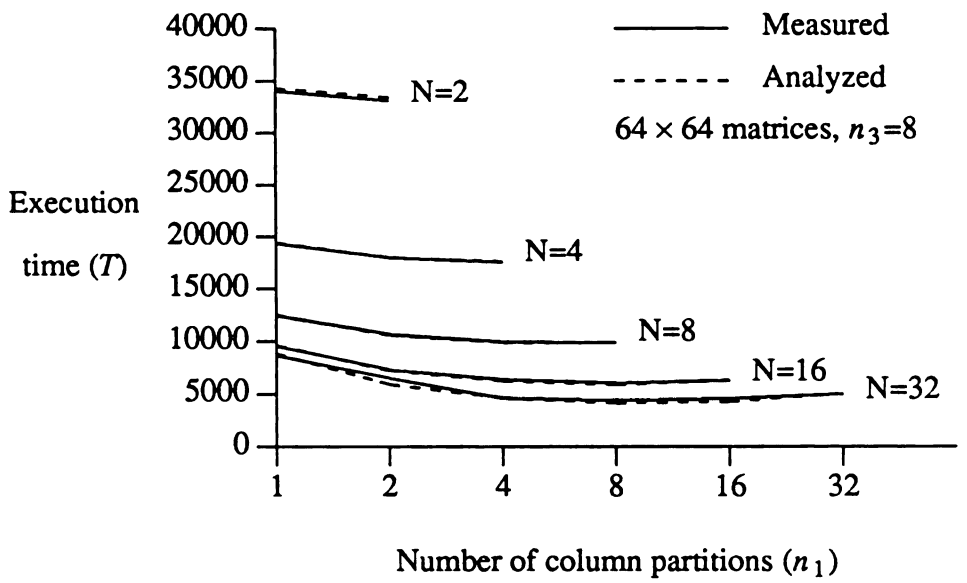


Figure 5.9. Measured and analyzed performance for various pipeline configurations

In Figure 5.9, the total execution times of the pipelined matrix multiplication algorithm (Algorithm 4.2), both measured on the NCUBE and predicted by the analytic model, are plotted. Again, multiplication of 64×64 matrices are studied using 2 to 32 nodes. The block size of submatrices of B is fixed (by setting n_3=8), and the pipeline configuration is changed by choosing the parameter n_1. It can be seen from Figure 5.9 that our analytic model predicts the execution of pipelined matrix multiplication correctly and that errors between analyzed and measured data are within 5%. The same accuracy of the analytic model can also be observed if we change the partition size, as shown in Figure 5.10.

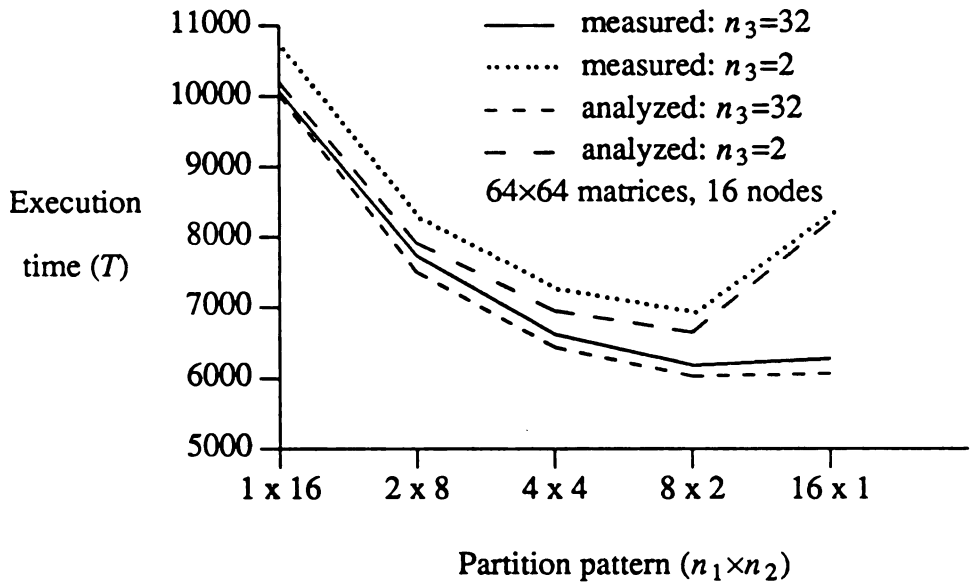


Figure 5.10. Measure and analyzed performance for various partition sizes

The optimal partitions predicted by the analytic model and obtained from the NCUBE are listed in Table 5.1. Again, we can observe a perfect match between these two sets of data. This indicates that the analytic model can be used to choose the best partition for a given problem instance.

Table 5.1. Optimal partitions for pipelined algorithm

	(n_1, n_2, n_3)				
N	2	4	8	16	32
measured	(2,1,16)	(4,1,16)	(8,1,16)	(8,2,16)	(8,4,8)
analyzed	(2,1,16)	(4,1,16)	(8,1,16)	(8,2,16)	(8,4,8)

Having established the accuracy of the analytic model, we then use the model to predict the performance of Algorithm 4.2 using 1024×1024 matrices and up to 1024 nodes. Figure 5.11 shows the predicted speedups for various environments. From Figure 5.11, it can be seen that, if the underlying architecture remains unchanged, i.e., we use the set of systems parameters in Table 3.1, then the speedup of the algorithm levels off when the number of processors increases to more than 600 (see the dashed line in Figure 5.11). As discussed, the communication overhead between nodes and between the host and nodes all contribute to this inefficiency.

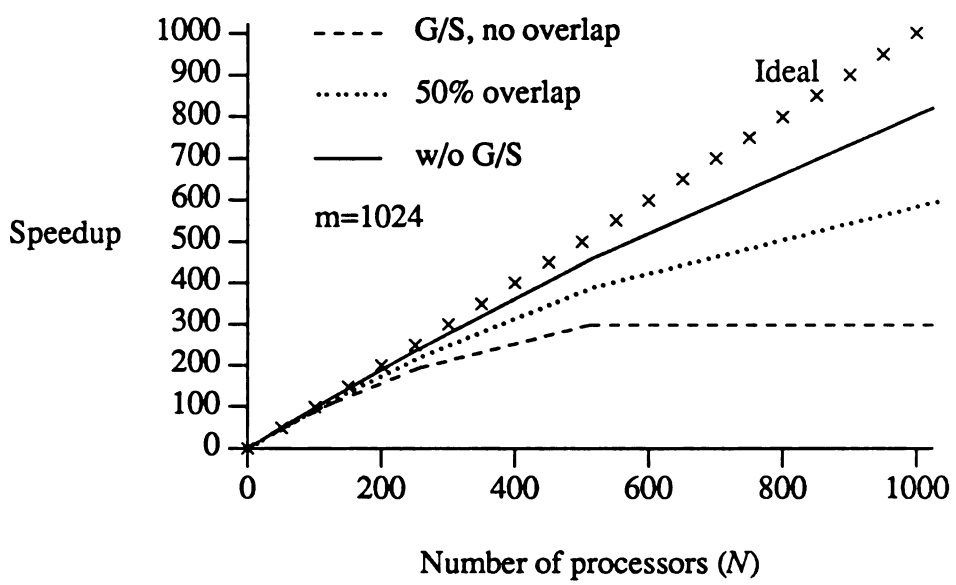


Figure 5.11. Predicted speedups of pipelined matrix multiplication

Now suppose that 50% of the communication overhead can be overlapped with the computation processor. Thus, for example, the single-byte transmission time on the host, which is τ_h=0.068 and τ_h'=0 in Table 3.1, now becomes τ_h=τ_h'=0.034, where τ_h' is the transmission time that can be overlapped with the computation processor. The dotted line in Figure 5.11 shows the speedup corresponding to this case. It can be seen that, by increasing the overlapping between computation and communication

processors, the speedup of the algorithm can be doubled (at N=1024 nodes). Using the new techniques as described in Section 1.2, such a degree of overlapping is not difficult to achieve in future generation DMMs.

In Section 3.2, we discussed the operation of gathering/scattering in the host and noted that this operation is very expensive. The solid line in Figure 5.11 shows the speedup that would obtain if the gathering/scattering operation takes no time. We can see that the speedup becomes almost linear in this case. The major reason is because we eliminate the sequential operation of gathering/scattering, which cannot be parallelized. Therefore, it worthes the effort to reduce the overhead involved in this kind of operation by either system designs (e.g., better complier or host hardware) or programming techniques (e.g., using pointers instead of indices to reference matrix elements).

CHAPTER 6

DESIGNING PIPELINED DATA PARALLEL ALGORITHMS

For many, designing algorithms is an art. However, as the technology advances, systematic approaches for developing algorithms have been discovered for solving a certain class of problems. This is possible because, in essence, the process of program development is nothing more than a series of transformations: from problem specification, through algorithm design, down to program coding and testing.

In previous chapters, we presented the basic concept of large-grain pipelining and the modeling techniques to analyze pipelined data parallel algorithms. However, the power of large-grain pipelining cannot be fully exploited if the development of pipelined data parallel algorithms is very difficult. In this chapter, a systematic approach for designing pipelined data parallel algorithms on DMMs is described.

The key to such a design procedure lies in the realization that systolic algorithms are a special class of pipelined data parallel algorithms (see Section 1.3). Therefore, many techniques for synthesizing systolic arrays can be adopted here [FoFW85, FoWa87]. In general, these techniques start with a nested-loop program or a set of linear recurrent equations, go through a series of transformations, and end up with a systolic array design. A brief review of these techniques will be presented in Section 6.1.

The major difference between pipelined data parallel algorithms running on DMMs and systolic algorithms is the granularity. Processing cells of systolic arrays operate on and exchange one data element at a time. However, this mode of operation is inefficient on DMMs due to the communication overhead in invoking messages. It follows that several data or operations should be grouped together to run on one processor in the DMM, and data exchange between processors uses one group as a unit. The grouping problem thus becomes the unique feature and primary consideration in the design procedure. In particular, one has to address the following issues:

- Which data and operations can be grouped together?
- What is the size of a group?
- How is the communication from one group to other groups minimized?

A general procedure for designing pipelined data parallel algorithms on DMMs is outlined in Section 6.2. Then, in the remainder of this chapter, the grouping problem is defined, assumptions are stated, and techniques for grouping data and operations are elaborated. An example will be given at the end of the chapter (Section 6.7) to illustrate the application of the grouping techniques.

6.1. SYNTHESIZING SYSTOLIC ARRAYS

Over the past few years, there has been a significant amount of work on synthesizing systolic arrays [FaSh87, FoFW85, MoFo86]. These techniques target at problems which can be expressed as *shift-invariant* nested-loop programs, i.e., programs whose loop data dependencies do not change with loop indices [KuLJ87]. Without being stated explicitly, all programs referenced in the following discussion will be assumed to be shift-invariant. In this section, the basic ideas of these synthesizing techniques are reviewed, which form the foundation of our design procedure.

Underlying all these synthesizing techniques is a representation which expresses data dependencies between loops of the given program. In general, a nested-loop program with n levels is usually expressed as follows:

```
for q_0 := l_0 to u_0 do
    for q_1 := l_1 to u_1 do
        .....
            for q_{n-1} := l_{n-1} to u_{n-1} do

                (loop body)
```

The above program can be transformed into a directed graph, Q, in an n-dimensional space. Each vertex in Q represents one loop instance and has a coordinate $(q_0, ..., q_{n-1})$ in the space if the corresponding loop instance has a loop index $(q_0, ..., q_{n-1})$. There is an arc from one vertex $\mathbf{v_i}$ to the other $\mathbf{v_j}$ if the loop corresponding to $\mathbf{v_j}$ references a variable which is generated in the loop corresponding to $\mathbf{v_i}$. Also, there is an arc from an *input variable* to every loop that references that variable. A variable is an input variable if it is referenced before or without being assigned a value in the program. Such a graph will be called the *computational structure* of the loop program [MiWi84]. The arcs are expressed as the difference vector of the two end vertices and will be called *dependence vectors* of the structure.

However, the derivation of a computational structure from a given loop program is not straightforward. Consider the program for matrix multiplication:

```
for i := 0 to M-1 do
    for j := 0 to M-1 do                    (6.1)
        for k := 0 to M-1 do
            c_ij := c_ij + a_ik×b_kj ;
```

There is an implicit data dependency between adjacent iterations in the inner-most loop via the variable c_{ij}, because c_{ij} is updated in each of the M iterations and the value of c_{ij} will depend on its own value in the previous iteration. To enforce an explicit dependencies between the iterations, we should require that the the single-assignment rule be followed. Note also that a_{ik} will be used in each of the M iterations with the same j

index. If one iteration is executed on one processor (as in systolic arrays), then this is equivalent to a broadcast of a_{ik} to all processors corresponding to the M vertices. However, broadcast is undesirable and is difficult to handle in systolic arrays [FoMo84].

Thus, to eliminate broadcast and to enforce single-assignment rule, we must rename variables in program (6.1) as follows (with decorations and initializations omitted for clarity):

$$\begin{array}{l} \text{for } i := 0 \text{ to } M-1 \text{ do} \\ \quad \text{for } j := 0 \text{ to } M-1 \text{ do} \\ \quad\quad \text{for } k := 0 \text{ to } M-1 \text{ do} \\ \quad\quad\quad a_{ijk} := a_{i,j-1,k}\ ; \\ \quad\quad\quad b_{ijk} := b_{i-1,j,k}\ ; \\ \quad\quad\quad c_{ijk} := c_{i,j,k-1} + a_{ijk} \times b_{ijk}\ ; \end{array} \tag{6.2}$$

The corresponding computational structure is shown in Figure 6.1. The numbers in the circles indicate the loop index (i,j,k) of the corresponding iterations. Note that, since the program is shift-invariant, the set of dependence vectors out of each vertex is same for all vertices. In Figure 6.1, there are three dependence vectors for each vertex:

$$D = \{\mathbf{d_0}, \mathbf{d_1}, \mathbf{d_2}\} = \{[0\ 1\ 0], [1\ 0\ 0], [0\ 0\ 1]\}$$

where D is called the *dependence set* of the computational structure. The process of eliminating broadcast and enforcing the single-assignment rule will eventually transform each variable in the loop into a *pipelined variable*, which depends explicitly on all the loop indices [MiWi84].

Given the computation structure, Q, of a program, we can map Q onto an array of processors by assigning each vertex (an loop instance) in Q to one processor. The dependence vectors in Q define the necessary interprocessor communication. However, in this implementation, each processor only executes one iteration and idles otherwise. As a result, there are no continuous data flows in the system and, the advantages of systolic arrays are not fully exploited. It follows that a second level transformation needs to be done.

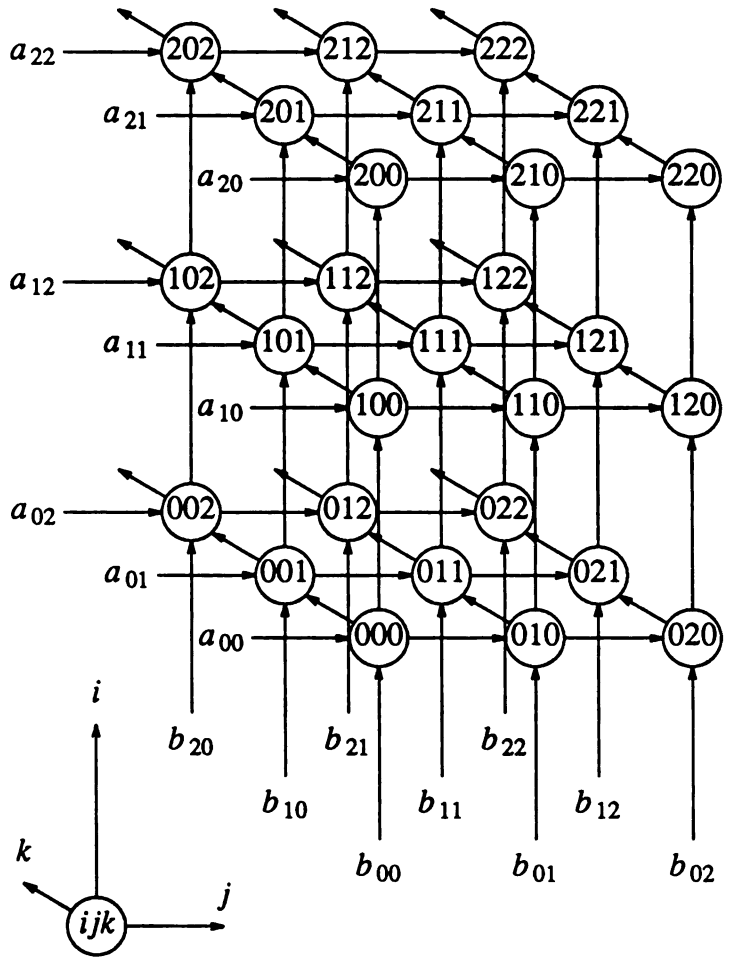


Figure 6.1. The computational structure of the matrix multiplication

The basic idea is to project all vertices in Q along a particular direction, and thus transform Q from an n-dimensional structure into an $(n-1)$-dimensional structure Q'. The projection direction represents the time axis, which indicates the progress of time. The projected structure Q' represents the final spatial layout of the systolic array. Given a projection direction **z**, an *index transformation matrix* **H** can be derived [FaSh87]. Each vertex **v** in Q is projected onto a vertex $\mathbf{Hv}^T$ in Q'. Similarly, each arc **d** in Q is projected onto an arc $\mathbf{Hd}^T$ in Q'. Thus, if vertices in the computational structure in Figure 6.1 are projected along $[i,j,k]=[1,1,1]$, we will have an index transformation matrix

$$\mathbf{H} = \begin{bmatrix} -1 & 1 & 0 \\ -1 & -1 & 2 \end{bmatrix}$$

The corresponding projected systolic array is depicted in Figure 6.2(a), which is the same as that proposed in [KuLe78]. The numbers in the circles of Figure 6.2(a) denote the indices of the loops that are projected onto it. Similarly, a projection along [0,1,0] will generate an index transformation matrix of

$$\mathbf{H} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The corresponding systolic array is shown in Figure 6.2(b), which has the same structure as that in Figure 4.6.

In summary, systolic array synthesizing techniques adopt an intelligent representation to express data dependencies within a nested loop program. From this representation, a systematic transformation of the program into a systolic array is possible. Nevertheless, since VLSI circuits have only 2-dimensional layouts, these techniques usually concentrate on computational structures in 2- or 3-dimensional spaces, i.e., n equals to 2 or 3. Although much work was developed with n being a general parameter, less known are the situations when n is greater than 3 or there are more than one time axis.

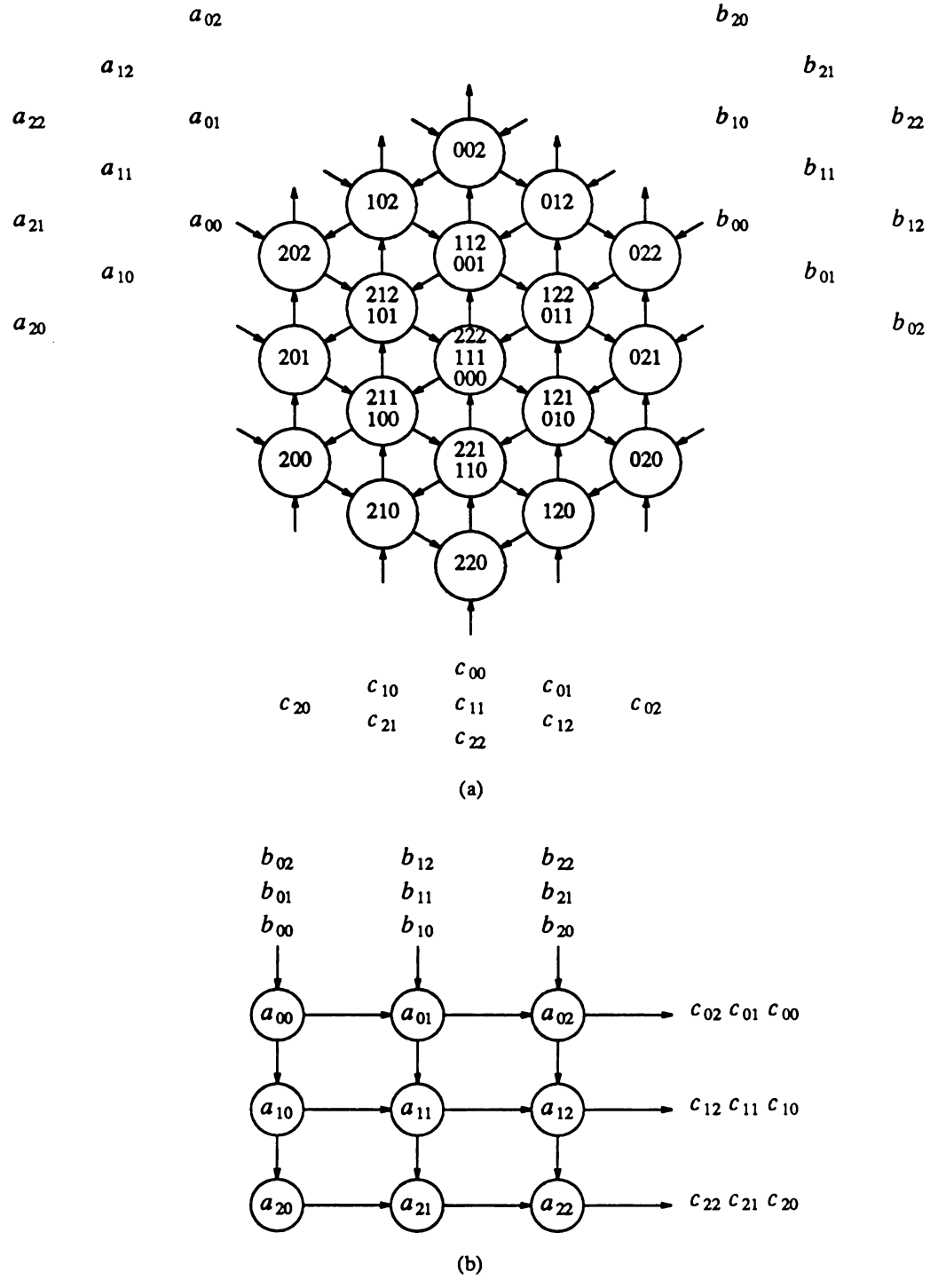


Figure 6.2. Projected computational structures of the matrix multiplication

6.2. THE DESIGN PROCEDURE

Based on the discussion presented in the previous section, a general procedure for designing pipelined data parallel algorithms on DMMs is outlined in this section. Again, we start with shift-invariant nested-loop programs. The initial stages of this design procedure are very similar to those of systolic array synthesizing techniques. However, due to the communication overhead on DMMs, the design procedure will require additional stages to group data and operations so as to control the granularity.

(1) Derive the computational structure of the program

Transformations are necessary to restructure the program, for instance, to eliminate broadcast and to enforce the single-assignment rule. Then, each iteration in the program is represented by one vertex in the computational structure, Q. Data dependencies between loops are identified and are represented as arcs in Q.

(2) Map the computational structure onto the space-time coordinate system

One direction is designated as the time axis, along which all vertices in Q are projected. The purpose of projection is to increase processor utilization and to introduce systolic effects. However, the projection along a particular direction is equivalent to the grouping of all vertices along that direction. Thus, this step is optional and can be combined with the next step.

(3) Group vertices in the computation structure

In this step, adjacent vertices in Q' are merged together to form larger vertices. By controlling the size of the groups, the granularity of the algorithm can be adjusted to balance the computation and communication on a DMM. Nevertheless, the grouping problem is non-trivial. For many computational structures, certain grouping schemes will introduce extra dependence vectors between the groups. This implies extra communication in the resultant algorithm. Details of the grouping will be discussed in the following sections.

(4) Assign the groups to the processors of the DMM

In this step, we allocate all vertices (i.e., loop instances) in a group to one processor in the DMM. Since the resultant computational structure has simple interconnections, the mapping process is straightforward. Besides, as discussed in Chapter 1, using next generation DMMs [ShFi88], the mapping problem is no longer critical.

(5) Determine the optimal design parameters

Design parameters such as the exact size of groups and the number of pipelines are determined. The analytic model introduced in Chapter 5 can be used here. The accuracy of the model and its associated parameters have a bearing on the final performance.

Note that due to the asynchronous nature of large-grain pipelining, the design procedure presented here does not have to consider timing. On the contrary, since systolic arrays are synchronous, timing is very important in synthesizing systolic arrays to decide the time when a datum should be sent to another processor. In this regard, the concept of large-grain pipelining is similar to that of *wavefront arrays* [KuAG82].

In the remaining sections, the core of the above design procedure — grouping — will be discussed more formally and in detail.

6.3. THE GROUPING PROBLEM

Let Z, I, and I^+, denote the set of integers, non-negative integers, and positive integers, respectively. The input to the grouping problem is a computational structure obtained from a shift-invariant nested-loop program with or without projection. Therefore, each vertex in the structure represents one loop instance (in the unprojected case) or a set of instances (in the projected case).

<Definition 6.1>

The computational structure, Q, is a two-tuple, $Q=(V,D)$, where $V=\{(q_0, ..., q_{n-1}) \mid l_i \le q_i \le u_i,\ 0 \le i \le n-1\}$ is the set of vertices in Q, l_i and u_i, $0 \le i \le n-1$,

are the lower and upper bounds of the i-th coordinate, and $D=\{\mathbf{d_0}, ..., \mathbf{d_{m-1}}\}$ is the set of dependence vectors in Q.

□

Suppose that V is a subset of Z^n, the set of integer n-tuples. Thus, all coordinates referred to will be integers. Also, without loss of generality, the upper and lower bounds in V, i.e., u_i and l_i, $0 \le i \le n-1$, are assumed to be constant, and $l_i \le 0$ and $u_i > 0$ so that the vertex with coordinate (0, ..., 0) is defined in Q.

In this study, only *acyclic* computational structures are considered. In other words, if there exist $\mathbf{d_0}, ..., \mathbf{d_{k-1}} \in D$ and $a_0, ..., a_{k-1} \in I$, $k \le m$, such that

$$a_0\mathbf{d_0} + a_1\mathbf{d_1} + \cdots + a_{k-1}\mathbf{d_{k-1}} = 0$$

then $a_0 = ... = a_{k-1} = 0$. Cycles will result in a lock-step synchronization between adjacent nodes for each iteration, which is undesirable for DMMs. More details of acyclic computational structures will be given later.

<Definition 6.2>

Let $Q(V,D)$ be a computational structure.

(1) A vertex $\mathbf{v_j} \in V$ is *dependent* on a vertex $\mathbf{v_i} \in V$ along a vector $\mathbf{d}$ if $\mathbf{v_j} - \mathbf{v_i} = \mathbf{d}$

(2) A vertex $\mathbf{v_j} \in V$ is *reachable* from a vertex $\mathbf{v_i} \in V$ via the set of vectors $D' = \{\mathbf{d_0}, ..., \mathbf{d_{k-1}}\}$ if $\mathbf{v_j} = \mathbf{v_i} + a_0\mathbf{d_0} + ... + a_{k-1}\mathbf{d_{k-1}}$, where $a_i \in Z$, $0 \le i \le k-1$.

(3) The set of vertices, $U(\mathbf{v},D')$, which is *spanned* from a vertex $\mathbf{v} \in V$ by a set of vectors, $D' = \{\mathbf{d_0}, ..., \mathbf{d_{k-1}}\}$, is the set of vertices in Q that are reachable from $\mathbf{v}$ via D', i.e.,

$$U(\mathbf{v},D') = \{\mathbf{w} \mid \mathbf{w} \in V,\ \mathbf{w} = \mathbf{v} + a_0\mathbf{d_0} + ... + a_{k-1}\mathbf{d_{k-1}},\ a_i \in Z,\ 0 \le i \le k-1\}$$

□

Note that any vertex in $U(\mathbf{v},D')$ is reachable from any other vertex in $U(\mathbf{v},D')$. Therefore, it is sometimes convenient to omit where the spanning starts and to speak only of the set of vertices that is spanned by D', which is denoted $U(D')$.

<Definition 6.3>

The *grouping* $G_{\mathbf{d},r}(Q)$ of a computation structure Q along a direction $\mathbf{d}$ of size r is to partition all vertices in Q into disjoint subsets, P_0, ..., P_{k-1}, such that

(1) $|P_0| = ... = |P_{k-1}| = r$

(2) For each subset P_i, $0 \le i \le k-1$, there exists an ordering for all vertices in P_i, i.e., $(\mathbf{v}_0, ..., \mathbf{v}_{r-1})$, such that $\mathbf{v}_{j+1} - \mathbf{v}_j = \mathbf{d}$, $0 \le j \le r-2$.

Each subset P_i is called a *group* of $G_{\mathbf{d},r}(Q)$, and the first vertex, $\mathbf{v}_0$, in the ordering in (2) is called the *base vertex* of P_i. A group P_i is *dependent* on another group P_j along $\mathbf{d}$ if there is a vertex $\mathbf{u}_k \in P_i$ which depends on a vertex $\mathbf{v}_k \in P_j$ along $\mathbf{d}$.

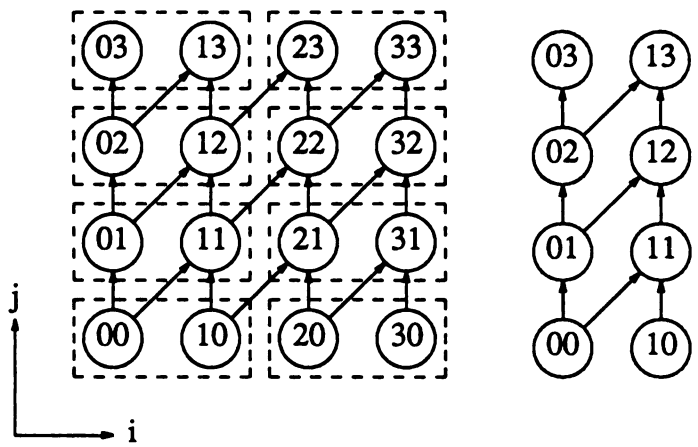
□

The physical meaning of a grouping is the same as that of a projection, i.e., all the loops (represented by vertices) enclosed in the same group will eventually be assigned to one processor and be executed according to their data dependence relationships. Figure 6.3(a) shows a computational structure $Q(V,D)$ with $V=\{(i,j) \mid 0 \le i,j \le 3\}$ and $D=\{[1\ 1], [0\ 1]\}$. A grouping $G_{\mathbf{d},r}(Q)$ along $\mathbf{d}=[1\ 0]$ of size $r=2$ is illustrated in dashed boxes in Figure 6.3(a).

Note that a grouping might not divide all vertices in a computational structure evenly. In this case, we can add dummy vertices at the boundary to make it even or include those extra vertices into boundary groups. Thus, for simplicity, we will assume the boundary conditions are satisfied in all cases. Since groups are disjoint sets, we can immediately identify the following relation in any grouping:

$$r \times k = \prod_{i=0}^{n-1} (u_i - l_i + 1)$$

Note also that a grouping may proceed along multiple directions with different sizes along each direction. In this case, it is not possible to find a total ordering of all vertices in a group.



(a) (b)

Figure 6.3. Grouping of a computational structure along [1 0]

<Definition 6.4>

The *contracted structure*, Q', of a computation structure $Q(V,D)$ with respect to the grouping $G_{\mathbf{d},r}(Q)$ is a directed graph, where

(1) Each vertex in Q' corresponds to one group in $G_{\mathbf{d},r}(Q)$;

(2) If P_i and P_j are the groups in $G_{\mathbf{d},r}(Q)$ which correspond to the vertices $\mathbf{v}_i'$ and $\mathbf{v}_j'$ in Q', respectively, then there is an arc from $\mathbf{v}_i'$ to $\mathbf{v}_j'$, if P_j depends on P_i.

Q' is *dependence preserving* if Q' is an acyclic computational structure, (V',D'), with $|D'| \leq |D|$.

□

The graph shown in Figure 6.3(b) is a contracted structure of that shown in Figure 6.3(a).

The idea of grouping is very similar to that of *emulation* in [FiFi82]. Note that, for a given grouping, the corresponding contracted structure may not be unique, depending

on how the base vertices are chosen. The significance of dependence preservance is that there is no extra dependence relationship introduced in the grouping. In other words, the number of processors with which a processor needs to communicate does not increase when loops are grouped. This is particularly important in DMMs with non-negligible communication overhead. The contracted structure shown in Figure 6.3(b) is dependence preserving. In fact, it has the same structure as the original computational structure.

As mentioned earlier, we only consider acyclic computational structures in this study. The implication of acyclic computational structures needs further explanation. In general, any computational structure derived directly from a nested-loop program is acyclic. Otherwise, there will be a deadlock in the execution of the program — a loop needs a value from another loop which is waiting for the value generated in the first loop. Cycles will be introduced only after a computational structure is projected or grouped. For example, consider the following nested-loop program:

```
for i := 0 to M−1 do
    for j := 0 to M−1 do
        b_ij := b_i,j−1 + b_i−1,j+1 ;
```

The corresponding computational structure for M=4 is shown in Figure 6.4(a). The projection of the computational structure along the direction [1 0] will results in a contracted structure as shown in Figure 6.4(b). Cycles are introduced. Note that each vertex, k, $0 \le k \le 3$, in Figure 6.4(b) represents a program segment as follows:

```
for i := 0 to 3 do
        b_ik := b_i,k−1 + b_i−1,k+1 ;
```

Suppose each vertex in Figure 6.4(b) is executed on one processor. Then, each processor must exchange data with its two neighbors at every iteration. In other words, processor k cannot execute two consecutive loops and produce, say, both b_{0k} and b_{1k} together without communicating with others. It follows that, even though several vertices are grouped together, the granularity of the algorithm does not increase at all!

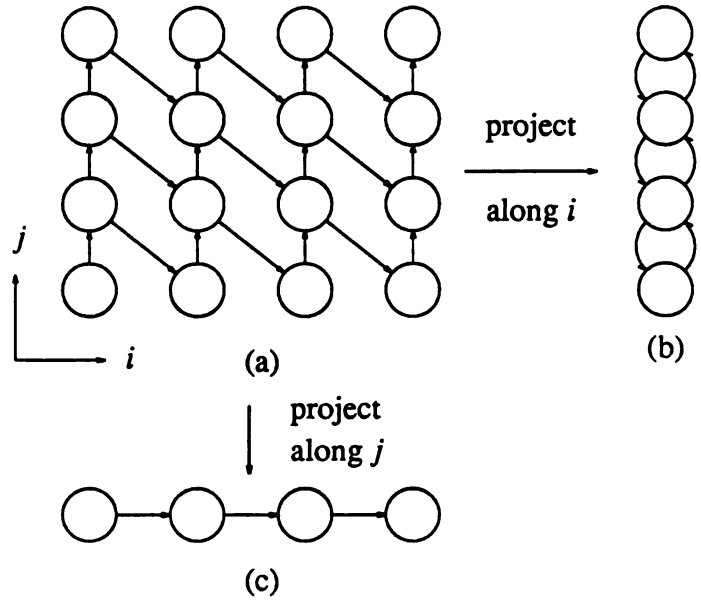


Figure 6.4. A computational structure with possible cycles

On the other hand, the grouping of the computational structure in Figure 6.4(a) along [0 1] will result in an acyclic contracted structure as shown in Figure 6.4(c). Each vertex, k, $0 \le k \le 3$, in Figure 6.4(c) represents a program segment as follows:

for j := 0 to 3 do
    $b_{kj} := b_{k,j-1} + b_{k-1,j+1}$;

Again, if each vertex is assigned to one processor, then processors do not have to exchange data at every iteration. The execution in processor k depends only on the outputs from processor $k-1$. Thus, for example, processor k can execute two loops, $j=0$, 1, before it sends b_{k0} and b_{k1} together to processor $k+1$. In this way, the granularity can be adjusted by changing the grouping size. Thus, cyclic computational structures may exist in systolic arrays (see Figure 6.2(a)), but they are not desirable in pipelined data parallel algorithms when executed on DMMs.

From what we have discussed so far, the grouping problem considered in this paper can thus be stated as follows:

<Grouping problem>

Given a computational structure Q, the grouping problem is to determine those groupings of Q which will result in dependence preserving contracted structures.

□

In the following discussions, we will concentrate on computational structures in 2-dimensional spaces. For higher dimensional spaces, the results obtained here may be extended. We will consider computational structures with one, two, three, and more than three dependence vectors, respectively. For each case, we study the conditions under which the computational structure can be partitioned into independent sets of vertices. We also present the necessary conditions that a grouping will result in dependence preserving contracted structures.

6.4. GROUPING WITH ONE OR TWO DEPENDENCE VECTORS

Suppose the computational structure $Q(V,D)$ has only one dependence vector, i.e., $D=\{\mathbf{d_0}\}$. Figure 6.5(a) depicts such a computational structure with $\mathbf{d_0}$=[0 1]. Since there is only one dependence vector in a 2-dimensional space, vertices in Q are divided into a number of *independent sets*. Each independent set is spanned by the vector $\mathbf{d_0}$, and vertices in different independent sets cannot reach each other, i.e., there is no data dependency between these set. Thus, these sets can be executed independently of each other.

<Theorem 6.1>

The grouping $G_{\mathbf{d_0},r}$, $r \in I^+$ of a computational structure $Q(V,D)$ with $D=\{\mathbf{d_0}\}$ is dependence preserving.

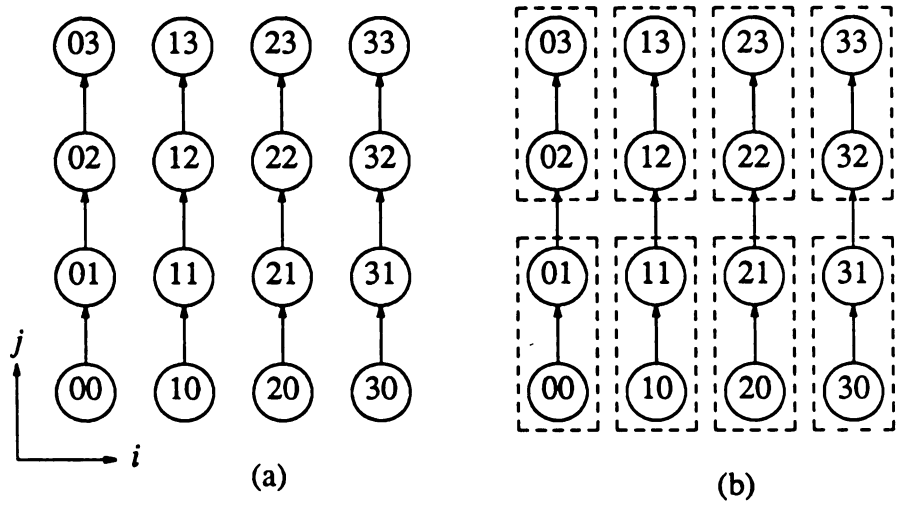


Figure 6.5. A computational structure with one dependence vector

<Proof>

Consider any group P in $G_{\mathbf{d}_0,r}$. Since there is only one dependence vector in the system, there exists a linear ordering of all vertices in P, say, $(\mathbf{v}_0, ..., \mathbf{v}_{r-1})$, where $\mathbf{v}_{j+1}-\mathbf{v}_j=\mathbf{d}_0$, $0 \le j < r-1$. The only vertex which has an arc to another vertex $\mathbf{v} \notin P$ is $\mathbf{v}_{r-1}$, and the only vertex which depends on a vertex $\mathbf{v} \notin P$ is $\mathbf{v}_0$. It follows that the number of group dependencies from P to other groups is only 1. Thus, the resultant contracted structure is dependence preserving.

□

Figure 6.5(b) shows a grouping along $\mathbf{d}_0=[0\ 1]$ with $r=2$, and it can be seen that the contracted structure is indeed dependence preserving.

Now, suppose the grouping is along $\mathbf{d}_1 \neq a_0\mathbf{d}_0$, $a_0 \in Z$. Then, this case is equivalent to a computational structure with two dependence vectors, $D=\{\mathbf{d}_0, \mathbf{d}_1\}$. An example of such a computational structure is shown in Figure 6.6(a), where $D=\{[1\ 0], [1\ 2]\}$. It can be seen from Figure 6.6(a) that the vertex (0,1) cannot be reached from the vertex (0,0). In other words, the vertices are divided into two independent sets (see Figure

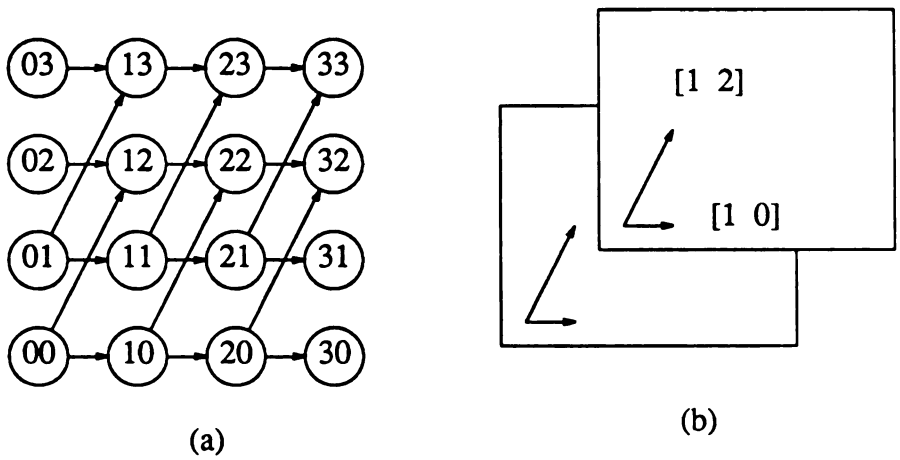


Figure 6.6. A computational structure with two dependence vector

6.6(b)).

Define the *base set*, W, of the computational structure $Q(V,D)$ to be

$$W = \{ \mathbf{w} \mid \mathbf{w} \in Z^2, \mathbf{w} = x\mathbf{d_0} + y\mathbf{d_1}, 0 \le x,y < 1 \}$$

Then, we have the following theorem:

<Theorem 6.2>

In a computational structure $Q(V,D)$ with $D=\{\mathbf{d_0}, \mathbf{d_1}\}$, the number of independent sets spanned by $\mathbf{d_0}$ and $\mathbf{d_1}$ is equal to $|W|$.

<Proof>

Let $\mathbf{v_i} \in V$ be a vertex in Q such that the vertices $\mathbf{v_j}=\mathbf{v_i}+\mathbf{d_0}+\mathbf{d_1}$, $\mathbf{v_k}=\mathbf{v_i}+\mathbf{d_0}$, and $\mathbf{v_l}=\mathbf{v_i}+\mathbf{d_1}$, are all in Q. Define the set

$$W_x = \{ \mathbf{w} \mid \mathbf{w} \in Z^2, \mathbf{w} = \mathbf{v_i} + x\mathbf{d_0} + y\mathbf{d_1}, 0 \le x,y < 1 \}$$

Then, obviously, $|W| = |W_x|$. Note that vertices in W_x cannot reach each other via $\mathbf{d_0}$ and $\mathbf{d_1}$. To prove the theorem, we have to show first that $W_x \subseteq V$. This implies that there are at least $|W|$ independent sets. Next, we have to prove that all vertices in V are

reachable from a vertex in W_x. In other words, the number of independent sets in Q is at most $|W|$. It follows that vertices in Q are divided into exactly $|W|$ independent sets.

Now, recall the definition of V (Definition 6.1). On a 2-dimensional space, we have

$$V = \{(q_0,q_1) \mid l_0 \le q_0 \le u_0,\ l_1 \le q_1 \le u_1\}$$

Since $\mathbf{v_l}$, $\mathbf{v_j}$, $\mathbf{v_k}$, $\mathbf{v_l} \in V$, all points with integer coordinates and inside the parallelogram enclosed by $\mathbf{v_l}$, $\mathbf{v_j}$, $\mathbf{v_k}$, and $\mathbf{v_l}$ must also be in V. It follows that $W_x \subseteq V$.

On the other hand, since Q is acyclic, $\mathbf{d_0}$ and $\mathbf{d_1}$ are independent. Any vertex $\mathbf{v} \in V$ in Q can be expressed as

$$\mathbf{v} = \mathbf{v_l} + (a_0+x)\mathbf{d_0} + (a_1+y)\mathbf{d_1} = \mathbf{v_l} + a_0\mathbf{d_0} + a_1\mathbf{d_1} + \mathbf{w'}$$

where a_0, $a_1 \in Z$ and $\mathbf{w'} = x\mathbf{d_0} + y\mathbf{d_1} \in Z^2$, $0 \le x,y < 1$. By choosing appropriate a_0 and a_1, we can make $\mathbf{w'} \in W$. This implies that $\mathbf{v}$ can be reached from a vertex, $\mathbf{v_l}+\mathbf{w'}$ in W_x. Thus, the theorem is proved.

□

From Figure 6.6, it can be seen that $W=\{(0,0), (0,1)\}$. Therefore, there are two independent sets in the computational structure. The significance of independent sets is not only that each set can be executed independently, but also that the grouping can be formed across the independent sets in any arbitrary manner.

Let us now consider the grouping along either $\mathbf{d_0}$ or $\mathbf{d_1}$.

<Theorem 6.3>

The grouping $G_{\mathbf{d},r}(Q)$ of a computational structure $Q(V,D)$ with $D=\{\mathbf{d_0}, \mathbf{d_1}\}$ is dependence preserving if $\mathbf{d} \in D$, $r \ge 1$, and the base vertices of the groups are chosen along $\mathbf{d_0}$ and $\mathbf{d_1}$.

<Proof>

Since the grouping in one independent set will not affect that in other sets, we can consider each independent set separately. Thus, without loss of generality, we will

assume $|W|=1$, $\mathbf{d}=\mathbf{d_0}$, and $\mathbf{d'}=\mathbf{d_1}$.

Let P_i and P_j be two groups in $G_{\mathbf{d_0},r}$. Order vertices in P_i and P_j according to their dependencies in $\mathbf{d_0}$ as follows:

For P_i: $(\mathbf{u_0}, ..., \mathbf{u_{r-1}})$, where $\mathbf{u_k}, \mathbf{u_{r-1}} \in P_i$ and $\mathbf{u_{k+1}}-\mathbf{u_k}=\mathbf{d_0}$, $0 \le k < r-1$

For P_j: $(\mathbf{v_0}, ..., \mathbf{v_{r-1}})$, where $\mathbf{v_k}, \mathbf{v_{r-1}} \in P_j$ and $\mathbf{v_{k+1}}-\mathbf{v_k}=\mathbf{d_0}$, $0 \le k < r-1$

Now, suppose P_j is dependent on P_i along $\mathbf{d_0}$. Since the grouping is along $\mathbf{d_0}$, then, among all vertices in P_i and P_j, only $\mathbf{v_0}$ depends on $\mathbf{u_{r-1}}$ along $\mathbf{d_0}$. Thus, no extra dependency will be introduced along $\mathbf{d_0}$.

On the other hand, suppose P_j depends on P_i along $\mathbf{d_1}$. Since base vertices are chosen along $\mathbf{d_0}$ and $\mathbf{d_1}$, we have $\mathbf{v_0}-\mathbf{u_0}=\mathbf{d_1}$. For any two vertices $\mathbf{u_k} \in P_i$ and $\mathbf{v_k} \in P_j$, where $0 \le k \le r-1$, we have $\mathbf{u_k}=k\mathbf{d_0}+\mathbf{u_0}$, and $\mathbf{v_k}=k\mathbf{d_0}+\mathbf{v_0}$, and $\mathbf{v_k}-\mathbf{u_k}=\mathbf{v_0}-\mathbf{u_0}=\mathbf{d_1}$. It follows that each vertex in P_j depends on a vertex in P_i along $\mathbf{d_1}$. Thus, there is no extra dependency introduced along $\mathbf{d_1}$, and the grouping is dependence preserving.

□

The above theorem can easily be expanded to allow groupings along both directions.

<Corollary 6.1>

The grouping of a computational structure $Q(V,D)$, where $D=\{\mathbf{d_0}, \mathbf{d_1}\}$, along $\mathbf{d_0}$ with a size $r_0 \ge 1$ and along $\mathbf{d_1}$ with a size $r_1 \ge 1$ is dependence preserving if the base vertices are chosen along $\mathbf{d_0}$ and $\mathbf{d_1}$.

□

Figure 6.7 illustrates a grouping along both directions, where groups are enclosed in the dashed lines. It is easy to check that the contracted structure Q' is dependence preserving. In fact, Q' has the same structure as the original computation structure. Again, the grouping along $\mathbf{d_2}$, where $\mathbf{d_2} \neq a_0\mathbf{d_0}$ and $\mathbf{d_2} \neq a_1\mathbf{d_1}$, $a_0, a_1 \in Z$, is equivalent to that on computational structures with three dependence vectors, $D=\{\mathbf{d_0}, \mathbf{d_1}, \mathbf{d_2}\}$. The latter will be discussed in the next section.

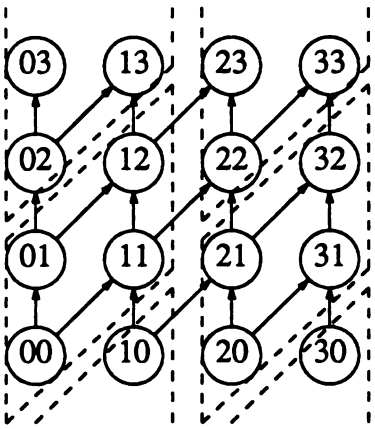


Figure 6.7. Grouping along two directions

6.5. GROUPING WITH THREE DEPENDENCE VECTORS

In this section, the grouping on computational structures with three dependence vectors is discussed. Let $Q(V,D)$ be a computational structure with $D=\{\mathbf{d_0}, \mathbf{d_1}, \mathbf{d_2}\}$. Assume further that

$$a_2\mathbf{d_2} = a_0\mathbf{d_0} + a_1\mathbf{d_1} \tag{6.3}$$

where a_0, a_1, and a_2 are the smallest positive integers to satisfy (6.3). A typical example is shown in Figure 6.8(a), where $\mathbf{d_0}$=[1 0], $\mathbf{d_1}$=[1 2], $\mathbf{d_2}$=[1 1], and 2[1 1] = [1 0] + [1 2].

Again, we start with the study of independent sets in Q. It turns out that a_2 plays a very important role in determining the independent sets.

<Lemma 6.1>

Let W be the base set with respect to $\mathbf{d_0}$ and $\mathbf{d_1}$ in Q. Then, $|W| \geq a_2$.

<Proof>

Since $|W| \geq 1$, the case for $a_2=1$ is trivial. Now, suppose $a_2 \geq 1$. Due to the fact that a_0, a_1, and a_2 are the smallest positive integers to satisfy (6.3), there does not exist any integer solution, x_i, y_i, $1 \leq i \leq a_2-1$, for each of the following a_2-1 equations:

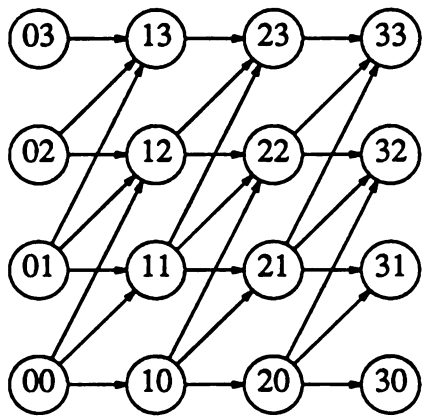


Figure 6.8. A computational structure with three dependence vectors

$$\mathbf{id_2} = x_i\mathbf{d_0} + y_i\mathbf{d_1}$$

In other words, for any vertex $\mathbf{v} \in V$ the vertices $\mathbf{v_1}=\mathbf{d_2}+\mathbf{v}$, ..., $\mathbf{v}_{a_2-1}=(a_2-1)\mathbf{d_2}+\mathbf{v}$, are not reachable from $\mathbf{v}$ via $\mathbf{d_0}$ and $\mathbf{d_1}$. Furthermore, none of these vertices can reach each other via $\mathbf{d_0}$ and $\mathbf{d_1}$. It follows that each of $\mathbf{v}$, $\mathbf{v_1}$, ..., $\mathbf{v}_{a_2-1}$ lies in a different independent set spanned by $\mathbf{d_0}$ and $\mathbf{d_1}$. Thus, $|W| \geq a_2$.

□

Let W' denote the base set with respect to $\mathbf{d_0}$, $\mathbf{d_1}$, and $\mathbf{d_2}$. Then, W' can be defined as

$$W' = \{\mathbf{w} \mid \mathbf{w} \in Z^2,\ \mathbf{w}=x\mathbf{d_0}+y\mathbf{d_1},\ 0 \leq x,y < 1,\ \text{and } \mathbf{w} \neq z\mathbf{d_2},\ z \in Z\}$$

We have the following theorem to relate the independent sets spanned by $\mathbf{d_0}$ and $\mathbf{d_1}$ and those spanned by $\mathbf{d_0}$, $\mathbf{d_1}$, and $\mathbf{d_2}$.

<Theorem 6.4>

Given a computational structure $Q(V,D)$ with $D=\{\mathbf{d_0}, \mathbf{d_1}, \mathbf{d_2}\}$, the number of independent sets spanned by $\mathbf{d_0}$, $\mathbf{d_1}$, and $\mathbf{d_2}$ is equal to $|W'|$ and $|W| = |W'| \times a_2$.

<Proof>

The proof of the first part is very similar to the proof of Theorem 6.2. Let **v** be a vertex in V such that all vertices in

$$W_x' = \{\mathbf{w} \mid \mathbf{w} \in Z^2,\ \mathbf{w}=\mathbf{v}+x\mathbf{d_0}+y\mathbf{d_1},\ 0 \le x,y<1,\ \text{and}\ \mathbf{w} \ne \mathbf{v}+z\mathbf{d_2},\ z \in Z\}$$

are defined in V. Then, $|W_x'|=|W'|$. Note that

$$W_x'=W_x-U(\mathbf{v},\{\mathbf{d_2}\})$$

where W_x is defined as in the proof of Theorem 6.2. Now, W_x contains vertices in V that cannot reach each other via $\mathbf{d_0}$ and $\mathbf{d_1}$. $U(\mathbf{v},\{\mathbf{d_2}\})$ is the set of vertices which can be reached from **v** via $\mathbf{d_2}$. It follows that the vertices in W_x' cannot reach each other via $\mathbf{d_0}$, $\mathbf{d_1}$, as well as $\mathbf{d_2}$. In addition, any vertex $\mathbf{u} \in V$ is reachable from a vertex in W_x', because

$$\mathbf{u} = a_0\mathbf{d_0}+a_1\mathbf{d_1}+a_2\mathbf{d_2}+\mathbf{w}+\mathbf{v}$$

where $a_i \in Z$, $0 \le i \le 2$, $\mathbf{w} \in W'$. Thus, $|W'|$ gives the total number of such independent sets.

Now consider each independent set spanned by $\mathbf{d_0}$, $\mathbf{d_1}$, and $\mathbf{d_2}$. Following the proof in Lemma 6.1, it can easily be seen that, within each set, there are a_2 independent sets spanned only by $\mathbf{d_0}$ and $\mathbf{d_1}$. It follows that $|W|=|W'|\times a_2$.

□

In Figure 6.8, since $a_2=2$, there are two independent sets spanned by [1 0] and [1 2]. Furthermore, we have $W=\{(0,0), (0,1)\}$ and $|W|=2$. This implies that $|W'|=1$ and that all vertices can be reached from (0,0) via those three dependence vectors.

As mentioned in Section 6.4, independent sets spanned by $\mathbf{d_0}$ and $\mathbf{d_1}$ can be considered as forming independent "planes". Then, the third vector, $\mathbf{d_2}$, can be viewed as forming a thread from one plane to the other. From Theorem 6.4, we can see intuitively that the grouping along $\mathbf{d_2}$ with a size $r_2=a_2$ will be dependence preserving, because we are linking a_2 independent sets together. However, any grouping along $\mathbf{d_2}$ with a

size larger than a_2 will include vertices which belong to the same independent set into one group. As a result, cycles are introduced into the contracted structure, and the resultant structure is not dependence preserving. Consider the example shown in Figure 6.9(a) where where $\mathbf{d_0}$=[1 0], $\mathbf{d_1}$=[0 1], $\mathbf{d_2}$=[1 1], and [1 1] = [1 0]+[0 1].

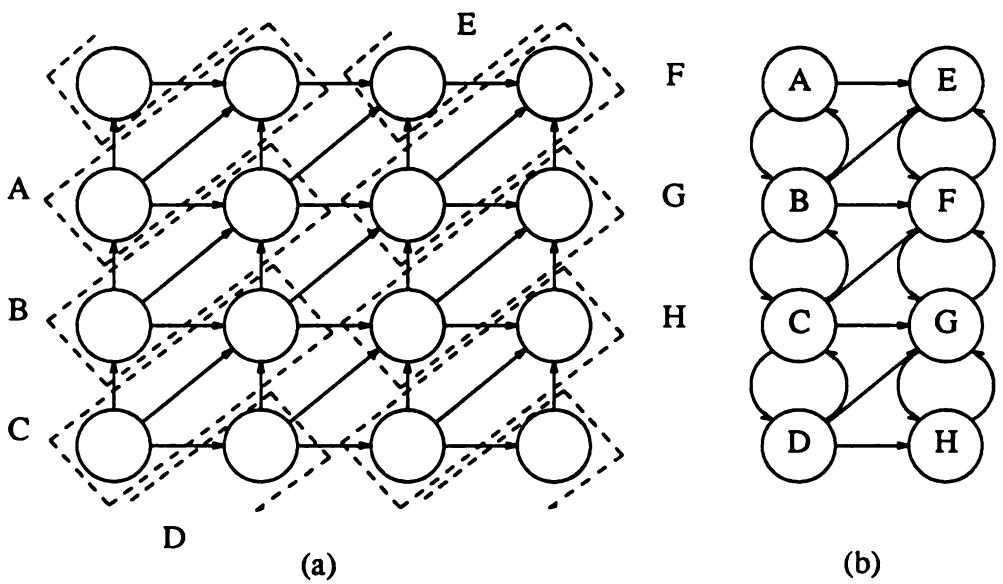


Figure 6.9. A grouping which generates a non-dependence-preserving structure

In Figure 6.9(a), the grouping along $\mathbf{d_2}$=[1 1] is depicted in dashed boxes. The labels besides the boxes denote the group ids. Figure 6.9(b) shows the corresponding contracted structure with the group ids indicated in the circles. It can be seen that an extra dependence vector, [0 -1], is introduced. Thus, the contracted structure is not dependence preserving. To prove the above points in a more formal way, we need the following Lemma:

<Lemma 6.2>

Let $Q(V,D)$ be a computational structure with $D=\{\mathbf{d_0}, \mathbf{d_1}, \mathbf{d_2}\}$, where $a_2\mathbf{d_2} = a_0\mathbf{d_0} + a_1\mathbf{d_1}$. Let P_i and P_j be two groups in the grouping $G_{\mathbf{d}_2,r}$, $r \in I^+$, such

that P_j is dependent on P_i along $\mathbf{d} \in \{\mathbf{d_0}, \mathbf{d_1}\}$. If $G_{\mathbf{d}_2,r}$ is dependence preserving, then the base vertex of P_j must depend on the base vertex of P_i along $\mathbf{d}$.

<Proof>

Let $\mathbf{u_0}$ and $\mathbf{v_0}$ be the base vertex of P_i and P_j, respectively. If $\mathbf{v_0}$ depends on $\mathbf{u_0}$ along $\mathbf{d}$, then there is a one-to-one dependence between the vertices in P_i and P_j along $\mathbf{d}$. As a result, no extra dependence vector will be generated along $\mathbf{d}$ in the contracted structure. On the other hand, assume that P_j depends on P_i along $\mathbf{d}$, but $\mathbf{v_0}-\mathbf{u_0} \neq \mathbf{d}$. Suppose further that P_j also depends on P_l along $\mathbf{d_2}$. The relationship between P_i, P_j, and P_l is illustrated in Figure 6.10. Let the vertices in P_i, P_j, and P_l be ordered according to the dependencies in $\mathbf{d_2}$ as follows:

For P_i: $(\mathbf{u_0}, ..., \mathbf{u_{r-1}})$, where $\mathbf{u_k}, \mathbf{u_{r-1}} \in P_i$ and $\mathbf{u_{k+1}}-\mathbf{u_k}=\mathbf{d_2}$, $0 \leq k < r-1$

For P_j: $(\mathbf{v_0}, ..., \mathbf{v_{r-1}})$, where $\mathbf{v_k}, \mathbf{v_{r-1}} \in P_j$ and $\mathbf{v_{k+1}}-\mathbf{v_k}=\mathbf{d_2}$, $0 \leq k < r-1$

For P_l: $(\mathbf{w_0}, ..., \mathbf{w_{r-1}})$, where $\mathbf{w_k}, \mathbf{w_{r-1}} \in P_l$ and $\mathbf{w_{k+1}}-\mathbf{w_k}=\mathbf{d_2}$, $0 \leq k < r-1$

Let $\mathbf{d}'$ denote the other vector which is different from $\mathbf{d}$ in $\{\mathbf{d_0}, \mathbf{d_1}\}$. Let $\mathbf{u_k} \in P_i$, $0 < k \leq r-1$, be the vertex that $\mathbf{v_0}-\mathbf{u_k}=\mathbf{d}$. If such a $\mathbf{u_k}$ cannot be found, then we consider P_i' instead, where P_i depends on P_i' along $\mathbf{d_2}$. The vertex $\mathbf{u_k}$ can be found in P_i'.

Consider the vertex $\mathbf{u_{k-1}}$. It can easily be seen from Figure 6.10 that $\mathbf{w_{r-1}}-\mathbf{u_{k-1}}=\mathbf{d}$. That is, both P_j and P_l depend on P_i along $\mathbf{d}$, and we introduce one extra dependence along $\mathbf{d}$ in the contracted structure. To keep $G_{\mathbf{d}_2,r}$ dependence preserving, we must make P_j or P_l also depend on P_i along $\mathbf{d}'$.

Suppose there exists a $\mathbf{v_g} \in P_j$ such that $\mathbf{v_g}-\mathbf{u_k}=\mathbf{d}'$ (see Figure 6.10). Then, from the dependence relationship between $\mathbf{v_g}$ and $\mathbf{v_0}$, we can see that

$$\mathbf{d} + g \times \mathbf{d_2} = \mathbf{d}'$$

which is a violation of (6.3). The same is true for the case where $\mathbf{w_g} \in P_l$ and $\mathbf{w_g}-\mathbf{u_k}=\mathbf{d}'$. Thus, there is no way to make P_j or P_l depend on P_i along $\mathbf{d}'$ while keeping $G_{\mathbf{d}_2,r}$ dependence preserving. It follows that we must have $\mathbf{v_0}-\mathbf{u_0}=\mathbf{d}$ to allow $G_{\mathbf{d}_2,r}$ be

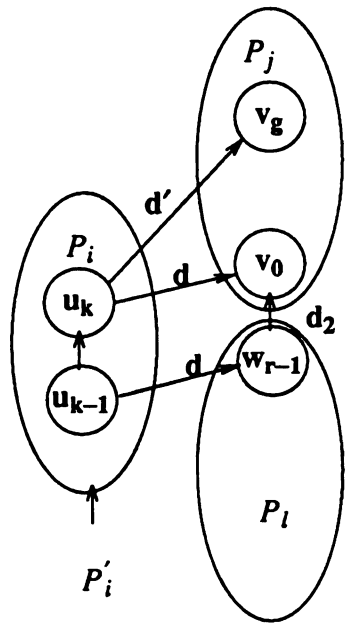


Figure 6.10. Dependence relationships between groups

dependence preserving.

□

<Theorem 6.5>

Let $Q(V,D)$ be a computational structure with $D=\{\mathbf{d_0}, \mathbf{d_1}, \mathbf{d_2}\}$, where $a_2\mathbf{d_2} = a_0\mathbf{d_0} + a_1\mathbf{d_1}$. Then, the grouping $G_{\mathbf{d_2},r}(Q)$ with a size $r>a_2$ is not dependence preserving.

<Proof>

Suppose the group is dependence preserving, and P is a group in the grouping. If we move from the base vertex of P along $\mathbf{d_0}$ for a_0 steps, then along $\mathbf{d_1}$ for a_1 steps, then in each step we should visit a base vertex of a group (Lemma 6.2). However, since $r>a_2$ and the relation in (6.3) holds, in the last step, we will visit a vertex which belongs to P and is not a base vertex. Thus, from Lemma 6.2, we arrive at a contradiction and prove that the grouping is not dependence preserving.

□

Following the same argument, we can arrive at the following result:

<Corollary 6.2>

Let $Q(V,D)$ be a computational structure with $D=\{\mathbf{d_0}, \mathbf{d_1}, \mathbf{d_2}\}$, where $a_2\mathbf{d_2} = a_0\mathbf{d_0} + a_1\mathbf{d_1}$. Then, the grouping $G_{\mathbf{d_2},r}$ with $r<a_2$ is dependence preserving if r can divide a_2 evenly.

<Proof>

The proof is similar to that of Theorem 6.5. Let P be a group in $G_{\mathbf{d_2},r}$. Then, as we move from the base vertex $\mathbf{v_0}$ of P along $\mathbf{d_0}$ for a_0 steps and then along $\mathbf{d_1}$ for a_1 steps, we should visit base vertices along the way until the last step where we arrive at a vertex $\mathbf{v_k}$. From (6.3), the following relation holds: $\mathbf{v_k}-\mathbf{v_0}=a_0\mathbf{d_0}+a_1\mathbf{d_1}=a_2\mathbf{d_2}$. However, Lemma 6.2 requires that $\mathbf{v_k}$ be a base vertex in order for $G_{\mathbf{d_2},r}$ to be dependence preserving. This condition will be satisfied only when the group size r is a factor of a_2.

□

Now, let us consider the grouping along $\mathbf{d_0}$ and $\mathbf{d_1}$ and see how $\mathbf{d_2}$ may affect this grouping. We study the dependence relationships within a contracted structure first.

<Lemma 6.3>

Let $Q(V,D)$ be a computational structure with $D=\{\mathbf{d_0}, \mathbf{d_1}, \mathbf{d_2}\}$, where $a_2\mathbf{d_2} = a_0\mathbf{d_0} + a_1\mathbf{d_1}$. Then, there exists a dependence preserving grouping along $\mathbf{d_i}$ with size a_i, where $i\in\{0,1,2\}$. Furthermore, the dependence vectors in the resultant contracted structure $Q'(V',D')$ have a relationship of

$$a_2'\mathbf{d_2}' = a_0'\mathbf{d_0}' + a_1'\mathbf{d_1}' \tag{6.4}$$

where $D'=\{\mathbf{d_0}', \mathbf{d_1}', \mathbf{d_2}'\}$, and $a_j'=a_j$, if $\mathbf{d_j}\neq\mathbf{d_i}$; $a_j'=1$, otherwise, $0\leq j\leq 2$.

<Proof>

Let $\mathbf{d_x}$ and $\mathbf{d_y}$, $x,y\in\{0,1,2\}$ denote the two vectors in D other than $\mathbf{d_i}$, and W' be the base set with respect to $\mathbf{d_0}$, $\mathbf{d_1}$, and $\mathbf{d_2}$. The grouping which is dependence preserving along $\mathbf{d_i}$ can be formed as follows:

For each vertex in W' perform the following operations:

(1) Form a group along $\mathbf{d}_l$ with size a_i;

(2) From the base vertex of the current group, choose the base vertex of the next group along $a_i\mathbf{d}_l$, $\mathbf{d}_x$, or $\mathbf{d}_y$.

until all vertices in the current independent set are grouped.

First, it is necessary to show that the above procedure does produce a grouping which is dependence preserving. Consider any independent set spanned by $\mathbf{d}_0$, $\mathbf{d}_1$, and $\mathbf{d}_2$, which contains $\mathbf{w} \in W'$. Let $U(\mathbf{w},\Delta)$ be the set of vertices which are spanned from $\mathbf{w}$ via $\Delta=\{a_i\mathbf{d}_l, \mathbf{d}_x, \mathbf{d}_y\}$. Note that vertices in $U(\mathbf{w},\Delta)$ can reach each other via Δ. From step (2) above, it can be seen that all base vertices in this independent set are contained in $U(\mathbf{w},\Delta)$. In addition, $U(\mathbf{w},\Delta)$ contains only base vertices. Suppose that this is not the case, and that $\mathbf{u}_k \in U(\mathbf{w},\Delta)$ is not a base vertex. Assume further that $\mathbf{u}_k \in P_i$ and $\mathbf{u}_0$ is the base vertex of P_i. Then,

$$\mathbf{u}_k = \mathbf{w} + za_i\mathbf{d}_l + x\mathbf{d}_x + y\mathbf{d}_y = \mathbf{u}_0 + k\mathbf{d}_l \;\rightarrow\; \mathbf{u}_0 = \mathbf{w} + (za_i - k)\mathbf{d}_l + x\mathbf{d}_x + y\mathbf{d}_y$$

where $x, y, z \in Z$. However, $k<a_i$. Thus, $\mathbf{u}_0$ cannot be reached from $\mathbf{w}$ via Δ, which contradicts the operation in step (2). It follows that moving away from any base vertex along any vector in Δ will reach only base vertices. Therefore, there will be no extra dependency introduced in the contracted structure (see the proof of Lemma 6.2), and the grouping is dependence preserving.

We now prove (6.4). Note that if we start from the base vertex $\mathbf{u}_0$ of any group P_i and move along $\mathbf{d}_x$ for a_x steps and then along $\mathbf{d}_y$ for a_y steps (actually, the move is along $-\mathbf{d}_2$, if $\mathbf{d}_x=\mathbf{d}_2$ or $\mathbf{d}_y=\mathbf{d}_2$), we will arrive at the base vertex $\mathbf{v}_0$ of another group P_j, where $\mathbf{v}_0=\mathbf{u}_0+a_i\mathbf{d}_l$. Now, all vertices between $\mathbf{u}_0$ and $\mathbf{v}_0$ along $\mathbf{d}_l$ are in P_i, because the grouping size of P_i is a_i. Thus, P_j depends on P_i along $\mathbf{d}_l$, and we obtain the dependence set satisfying (6.4) in the resultant contracted structure.

□

<Lemma 6.4>

Let $Q(V,D)$ be a computational structure with $D=\{\mathbf{d_0}, \mathbf{d_1}, \mathbf{d_2}\}$, where $\mathbf{d_2} = \mathbf{d_0} + \mathbf{d_1}$. Any grouping along $\mathbf{d} \in \{\mathbf{d_0}, \mathbf{d_1}\}$ with a size $r \in I^+$ is dependence preserving.

<Proof>

Without loss of generality, assume $\mathbf{d}=\mathbf{d_0}$. Consider any group P in the grouping. Order all vertices in P according to the dependencies in $\mathbf{d_0}$ as follows:

$$(\mathbf{v_0}, ..., \mathbf{v_{r-1}}), \text{ where } \mathbf{v_k}, \mathbf{v_{r-1}} \in P \text{ and } \mathbf{v_{k+1}}-\mathbf{v_k}=\mathbf{d_0}, \ 0 \le k < r-1$$

Since $\mathbf{v_k}+\mathbf{d_2}=\mathbf{v_k}+\mathbf{d_0}+\mathbf{d_1}=\mathbf{v_{k+1}}+\mathbf{d_1}$, $0 \le k < r-1$, only the following $r+1$ vertices will depend on a vertex in P along either $\mathbf{d_1}$ or $\mathbf{d_2}$: $\mathbf{v_0}+\mathbf{d_1}$, ..., $\mathbf{v_{r-1}}+\mathbf{d_1}$, and $\mathbf{v_{r-1}}+\mathbf{d_2}=\mathbf{v_{r-1}}+\mathbf{d_0}+\mathbf{d_1}$. Note that all these $r+1$ vertices are linked by $\mathbf{d_0}$ in a chain. Thus, they can only be grouped into two groups. It follows that P has only three dependence vectors to other groups, and that the grouping is dependence preserving.

□

<Theorem 6.6>

Let $Q(V,D)$ be a computational structure with $D=\{\mathbf{d_0}, \mathbf{d_1}, \mathbf{d_2}\}$, where $a_2\mathbf{d_2} = a_0\mathbf{d_0} + a_1\mathbf{d_1}$. Then, the grouping along $\mathbf{d_0}$ with a size of ia_0, along $\mathbf{d_1}$ with a size of ja_1, and along $\mathbf{d_2}$ with a size of a_2, where $i,j \in I^+$, is dependence preserving.

<Proof>

From Lemma 6.3, we can see that, if Q is grouped along $\mathbf{d_0}$ with a size a_0, $\mathbf{d_1}$ with a size a_1, and $\mathbf{d_2}$ with a size a_2, then the resultant contracted structure Q' is dependence preserving and has a set of dependence vectors with the following relationship

$$\mathbf{d_2}' = \mathbf{d_0}' + \mathbf{d_1}' \tag{6.5}$$

Therefore, the grouping of vertices in Q' along $\mathbf{d_0}'$ with a size i and along $\mathbf{d_1}'$ with a size j, $i,j \in I^+$ is dependence preserving (Lemma 6.4). However, the grouping in Q' along $\mathbf{d_0}'$ with a size i is equivalent to the grouping along $\mathbf{d_0}$ with a size ia_0 in Q.

Similarly, the grouping in Q' along $\mathbf{d}_1'$ with a size j is equivalent to the grouping along $\mathbf{d}_1$ with a size ja_1 in Q. Thus, the theorem follows.

□

The strategy used in Theorem 6.6 is to transform the computational structure into the one with a hexagonal structure (described by (6.5)). This kind of structure is referred to as the *universal planner array*, which is the most general systolic array in 2-dimensional spaces [MiWi84]. The same structure will be used in the next section to construct grouping schemes for computational structures with three or more dependence vectors.

6.6. GROUPING WITH FOUR OR MORE DEPENDENCE VECTORS

When computational structures have four or more dependence vectors, the relationships between the dependence vectors become very complicated, which make it difficult to analyze individual vectors separately. Furthermore, the chance that there are independent sets spanned by all vectors is rare. As a result, we concentrate in this section only on grouping schemes but not on independent sets.

Given an acyclic computational structure $Q(V,D)$ on a 2-dimensional space with $D = \{\mathbf{d}_0, ..., \mathbf{d}_{m-1}\}$, where $m>2$, we can always find (from linear algebra theory) two vectors, say, $\mathbf{d}_0$ and $\mathbf{d}_1$, such that

$$c_i\mathbf{d}_i = a_i\mathbf{d}_0 + b_i\mathbf{d}_1 \tag{6.6}$$

where a_i, b_i, $c_i \in I^+$, $2 \le i < m$. Define

$$a_{\max} = \underset{2\le i<m}{\text{Max}} \left\{ \left\lceil \frac{a_i}{c_i} \right\rceil \right\} \qquad b_{\max} = \underset{2\le i<m}{\text{Max}} \left\{ \left\lceil \frac{b_i}{c_i} \right\rceil \right\}$$

We have the following lemma concerning the "range of influence" of a vertex in Q.

<Lemma 6.5>

Let $Q(V,D)$ be a computational structure with $D=\{\mathbf{d_0}, ..., \mathbf{d_{m-1}}\}$, where $m>2$, $c_i\mathbf{d_l} = a_i\mathbf{d_0} + b_i\mathbf{d_1}$, and $2\le i<m$. For any vertex $\mathbf{v}\in V$, define

$$X(\mathbf{v}) = \{\mathbf{w} \mid \mathbf{w}\in V,\ \mathbf{w}=\mathbf{v}+xa_{\max}\mathbf{d_0}+yb_{\max}\mathbf{d_1},\ 0\le x,y\le 1\}$$

Then, any vertex $\mathbf{u}\in V$ which is dependent on $\mathbf{v}$ is in $X(\mathbf{v})$.

<Proof>

For any vertex $\mathbf{u}\in V$, if $\mathbf{u}$ is dependent on $\mathbf{v}$, then there exists a dependence vector $\mathbf{d_l}$, $0\le i<m$, such that

$$\mathbf{u}-\mathbf{v} = \mathbf{d_l} = \frac{a_i}{c_i}\mathbf{d_0} + \frac{b_i}{c_i}\mathbf{d_1} \le a_{\max}\mathbf{d_0} + b_{\max}\mathbf{d_1}$$

Thus, $\mathbf{u}\in X(\mathbf{v})$.

□

<Theorem 6.7>

Let $Q(V,D)$ be a computational structure with $D=\{\mathbf{d_0}, ..., \mathbf{d_{m-1}}\}$, where $m>2$, $c_i\mathbf{d_l} = a_i\mathbf{d_0} + b_i\mathbf{d_1}$, and $2\le i<m$. Then, the grouping along $\mathbf{d_0}$ with a size $r_0\ge a_{\max}$ and along $\mathbf{d_1}$ with a size $r_1\ge b_{\max}$ is dependence preserving.

<Proof>

Consider any group P with base vertex $\mathbf{v_0}$. Let P_i, P_j, and P_k be the three neighboring groups of P as shown in Figure 6.11. It suffices to show that, for any vertex $\mathbf{v}$ in P, $X(\mathbf{v})$ is covered by P, P_i, P_j, and P_k. The immediate consequence is that there will only be three dependence vectors in the resultant contracted structure, and these three dependence vectors satisfy the relation in (6.6). In other words, the contracted structure has a structure of the universal planner array.

Any vertex $\mathbf{w}$ falls within P, P_i, P_j, and P_k, can be expressed as follows:

$$\mathbf{w} = \mathbf{v_0} + xa_{\max}\mathbf{d_0} + yb_{\max}\mathbf{d_1} \qquad 0\le x,y<2 \tag{6.7}$$

Any vertex $\mathbf{v}$ in P can be expressed as

$$\mathbf{v} = \mathbf{v_0} + x_v a_{\max}\mathbf{d_0} + y_v b_{\max}\mathbf{d_1} \qquad 0\le x_v,y_v<1$$

Now, from the definition of $X(\mathbf{v})$, any vertex $\mathbf{v}' \in X(\mathbf{v})$ can be expressed as

$$\mathbf{v}' = \mathbf{v} + x_{v'} a_{\max} \mathbf{d_0} + y_{v'} b_{\max} \mathbf{d_1} \quad 0 \le x_{v'}, y_{v'} \le 1$$

$$= \mathbf{v_0} + (x_v + x_{v'})\mathbf{d_0} + (y_v + y_{v'})\mathbf{d_1}$$

where $0 \le (x_v + x_{v'}), (y_v + y_{v'}) < 2$. Thus, $\mathbf{v}'$ satisfies (6.7) and is in the region covered by P, P_i, P_j, and P_k. It follows that, for any $\mathbf{v}$ in P, $X(\mathbf{v})$ falls within the above four groups (see the dashed box in Figure 6.11). Therefore, the grouping is dependence preserving. □

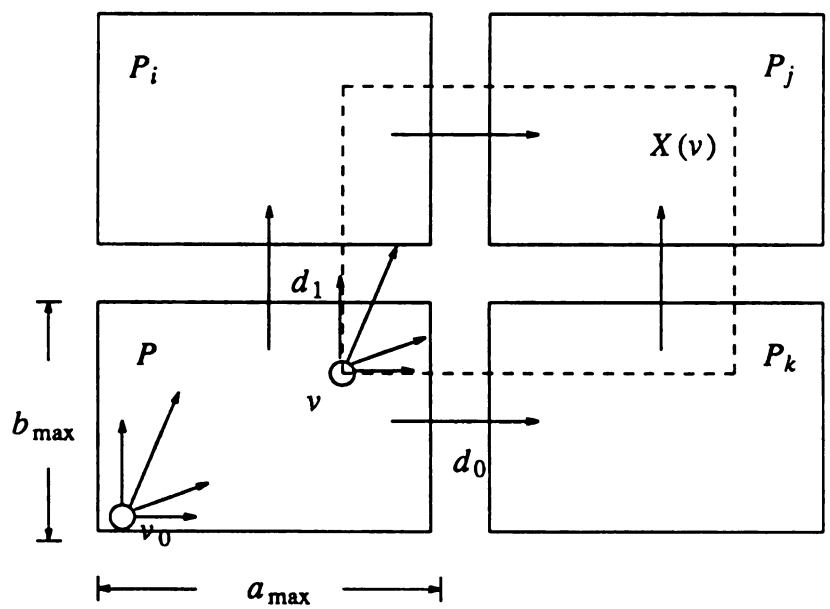


Figure 6.11. Relationship between groups in a universal planner array

Theorem 6.7 presents a powerful grouping method for computational structures. Specifically, as long as the group size is large enough, then there always exist dependence preserving groupings along $\mathbf{d_0}$ and $\mathbf{d_1}$. In fact, $\mathbf{d_0}$ and $\mathbf{d_1}$ need not be dependence vectors of the computational structure, as long as the relations in (6.6) are satisfied.

By applying Theorem 6.7, the resultant contracted structure has only three dependence vectors. It follows that, in the final implementation, each processor only has to communicate with at most three other processors. The communication bandwidth is reduced, and the partition task is simplified, because any partition with a size greater than a_{max} and b_{max} will be dependence preserving. The remaining task is to match the number of partitions (i.e., the number of groups) with the number of processors available.

Note that Theorem 6.7 can be extended to computational structures on higher dimensional spaces. In a 3-dimensional space, for example, a cube is adjacent to 7 other cubes in the first octant. Thus, any computational structure in 3-dimensional space can be grouped along three directions. As long as the group size is large enough, the grouping is dependence preserving and each resultant group is adjacent to at most 7 other groups.

6.7. AN EXAMPLE OF GROUPING

An example is given in this section to illustrate the application of grouping and to summarize what has been discussed so far. Consider the development of a pipelined data parallel algorithm from the following "artificial" nested-loop program:

$$\begin{array}{l} \text{for } i := 0 \text{ to } M-1 \text{ do} \\ \quad \text{for } j := 0 \text{ to } M-1 \text{ do} \\ \quad\quad \text{for } k := 0 \text{ to } M-1 \text{ do} \\ \quad\quad\quad c_{ij} := c_{ij} + b_{ik} \times e_{j,k-2} / g_{i-1,k} \text{ ;} \end{array} \tag{6.8}$$

(1) Derive the computational structure of the program

It can be seen from the above program that b_{ik}'s and $g_{i-1,k}$'s, $0 \le i,k \le M-1$, are broadcast along the loops indexed by j, and $e_{j,k-2}$'s, $0 \le j,k \le M-1$, are broadcast along i. By eliminating broadcast and enforcing the single-assignment rule, the above program can be transformed into the following program:

$$\text{for } i := 0 \text{ to } M-1 \text{ do}$$

$$
\begin{array}{l}
\text{for } j := 0 \text{ to } M-1 \text{ do} \\
\quad \text{for } k := 0 \text{ to } M-1 \text{ do} \\
\quad\quad b_{ijk} := b_{i,j-1,k}; \\
\quad\quad e_{ijk} := e_{i-1,j,k-2}; \\
\quad\quad g_{ijk} := g_{i-1,j-1,k}; \\
\quad\quad c_{ijk} := c_{i,j,k-1} + b_{ijk} \times e_{ijk} / g_{ijk}
\end{array}
\tag{6.9}
$$

The necessary initializations of b_{ijk}, e_{ijk}, and g_{ijk} are omitted for clarity.

From the transformed program, the dependence vectors can be identified as follows: $D=\{[0\ 1\ 0], [1\ 0\ 2], [1\ 1\ 0], [0\ 0\ 1]\}$. The variables which introduce the dependence vectors are listed below:

$$b_{ijk} \rightarrow [0\ 1\ 0] \qquad e_{ijk} \rightarrow [1\ 0\ 2] \qquad g_{ijk} \rightarrow [1\ 1\ 0] \qquad c_{ijk} \rightarrow [0\ 0\ 1]$$

Figure 6.12 depicts the computational structure of the program in (6.9). Due to the complexity of the figure, we only show the projected vertices on each of the three planes. Thus, for example, in Figure 6.12, the plane labeled (1) contains the 16 vertices when the computational structure is project onto the ij-plane. Each vertex in Figure 6.12 represents one loop instance in the original program.

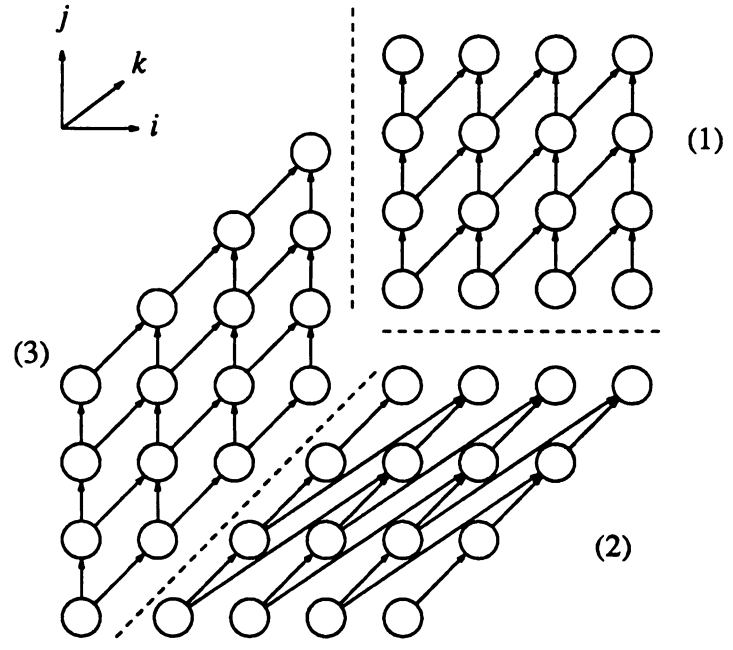


Figure 6.12. The computational structure of the example program

(2) Map the computational structure onto the space-time coordinate system

To take advantage of pipelining and further increase the granularity, a projection is sought to transform the 3-dimensional computational structure into a 2-dimensional one. For example, if we project the vertices along the j-axis, we will obtain a projected computational structure as shown in Figure 6.13. Now, each vertex in the projected structure represents M loops. In fact, the vertex at position (i',k') represents the following program segment:

$$\begin{aligned} &\text{for } j := 0 \text{ to } M-1 \text{ do} \\ &\qquad c_{i'j} := c_{i'j} + b_{i'k'} \times e_{j,k'-2} / g_{i'-1,k'}; \end{aligned} \tag{6.10}$$

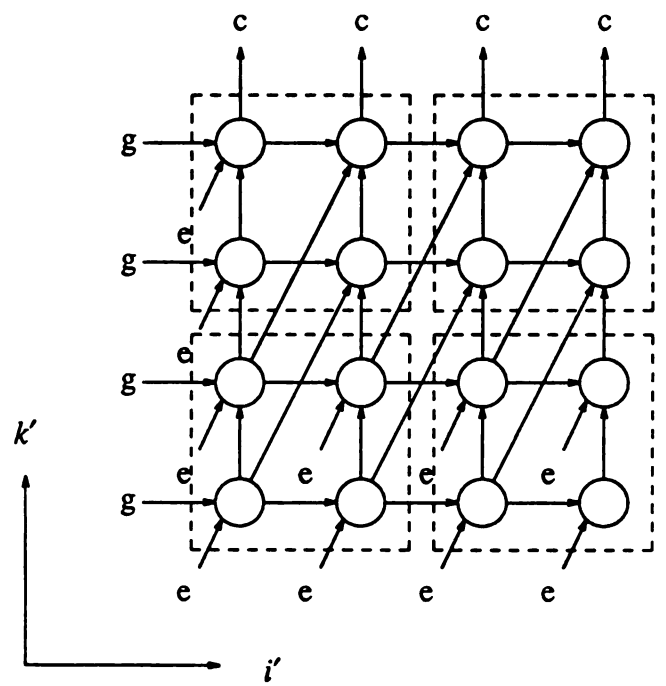


Figure 6.13. The projected computational structure of the example

Due to this projection, the variables b_{ijk}, $0 \le j \le M-1$, are projected onto the same vertex in the projected computational structure. This means that these variables are kept stationary in the corresponding processor, and that they have to be downloaded to the

corresponding processors before the execution can start.

(3) Group vertices in the computational structure

Note that the resultant dependence vectors in the projected structure are: $\mathbf{d_0}$=[1 0], $\mathbf{d_1}$=[0 1], and $\mathbf{d_2}$=[1 2], where

$$a_2\mathbf{d_2} = [1\ 2] = [1\ 0] + 2\times[0\ 1] = a_0\mathbf{d_0} + a_1\mathbf{d_1} \tag{6.11}$$

Thus, we have a_0=a_2=1 and a_1=2. Since a_2=1, from Theorem 6.5, we can conclude that any grouping along $\mathbf{d_2}$=[1 2] is not dependence preserving. To derive groupings which are dependence preserving, either Theorem 6.6 or Theorem 6.7 can be used. For example, if Theorem 6.7 is used, then note that

$$a_{\max} = \frac{a_0}{a_2} = 1 \qquad \text{and} \qquad b_{\max} = \frac{a_1}{a_2} = 2$$

Thus, any grouping along [1 0] with a size $r_0 \geq 1$, and along [0 1] with a size $r_1 \geq 2$, is dependence preserving. The dashed boxes in Figure 6.13 enclose groups of vertices corresponding to the grouping with r_0=r_1=2. On the other hand, if Theorem 6.6 is used, then we can first group vertices in Figure 6.13 along [0 1] with a size 2. From Lemma 6.3, we can see that the resultant contracted structure will have a set of dependence vectors which satisfies

$$\mathbf{d_2}' = \mathbf{d_0}' + \mathbf{d_1}'$$

Figure 6.14(a) depicts the grouping. The corresponding contracted structure is shown in Figure 6.14(b). The contracted structure has $\mathbf{d_0}'$=[1 0], $\mathbf{d_1}'$=[0 1], and $\mathbf{d_2}'$=[1 1]. Now, from Lemma 6.4, any grouping along $\mathbf{d_0}'$ and $\mathbf{d_1}'$ is dependence preserving. Thus, for example, the grouping along [1 0] with a size 2 will generate a dependence preserving grouping which is same as that shown in Figure 6.13.

Finally, as is noted in Section 4.3, we can exploit pipelining in pipelined data parallel algorithms not only from multiple pipelines but also from partitioning data to form data streams. Thus, we can further partition the M loops indexed by j in program

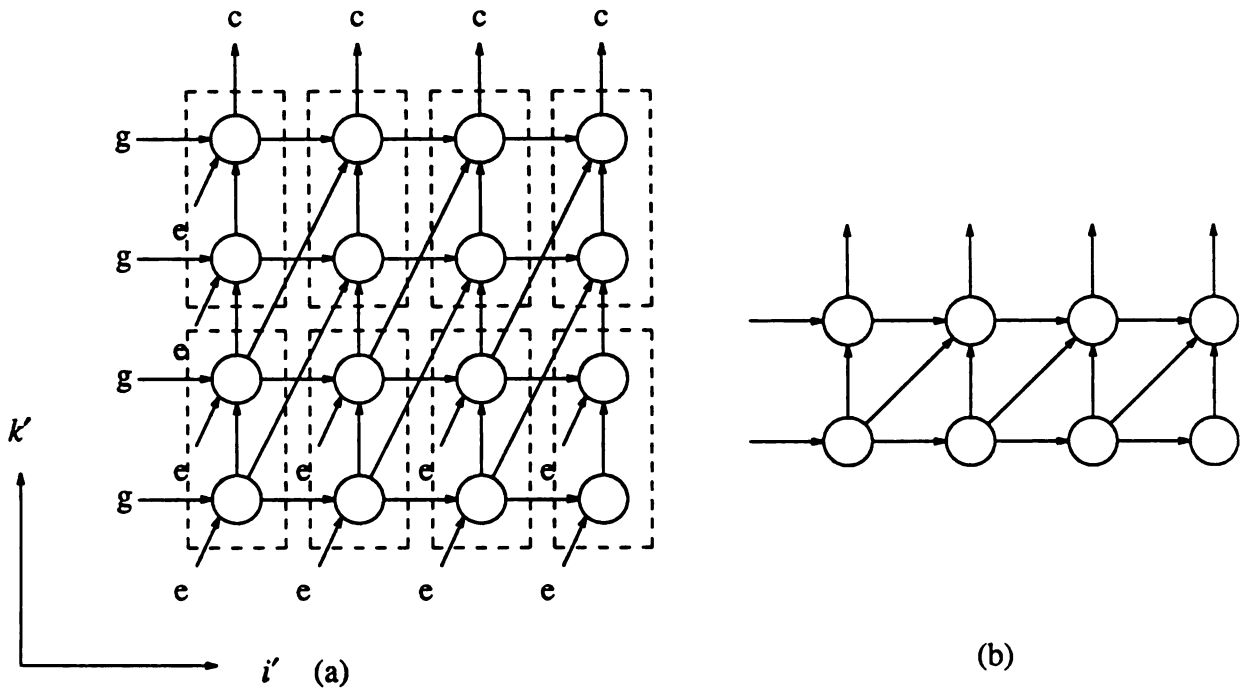


Figure 6.14. The grouping along [0 1] with a size of 2

(6.10) into a number of blocks, each has $r_2 \leq M$ loops. Assume that r_2 divides M evenly. Then, the program executed in the vertex (i',k') of the resultant contracted structure becomes the following:

```
for l := 0 to M−1 by r2 do
    /* communicate with neighboring processors */
    for j := l to l+r2−1 do
        for i := i'×r0 to (i'+1)×r0−1 do                    (6.12)
            for k := k'×r1 to (k'+1)×r1−1 do
                cij := cij + bik×ej,k−2/gi−1,k;
```

where r_0 is the grouping size along [1 0] and r_1 is the grouping size along [0 1]. The implicit assumption here is that the loops can be interchanged [PaWo86]. Thus, the loops indexed by j are moved to the outer-most level. In general, loop interchanging is possible if there is no cycle in the computational structure. Thus, each processor can execute large chunks of data (e.g., the three inner-most loops) before it needs to

communicate.

(4) Assign the groups to the processors of the DMM

Suppose the underlying DMM has a hypercube interprocessor connection topology. Then, there does not seem to have a perfect embedding of a hexagon onto a hypercube with dilation 1. A simple embedding with dilation 2 is to configure the hypercube into a 2-dimensional mesh, and map the hexagon onto the mesh, with each diagonal link, [1 1], in the hexagon going through a pair of edges, [1 0] and [0 1], in the mesh. In new generation DMMs, this mapping should not induce too much communication delay.

(5) Determine the optimal design parameters

The exact values of r_0, r_1, and r_2 can be determined by the analytic model presented in Chapter 5.

Note that, from a grouping, the following parameters can be obtained:

- With which other processors does a processor need to communicate?
- Which data does the processor need to exchange?
- How large should each message be?
- How often does the processor need to communicate?

Referring back to the program in (6.12), we can see that the processor (i',k') needs to communicate every r_2 iterations (indexed by j) with its neighbors. In each data exchange, there will be r_1 elements of g_{ijk} sent to processor $(i'+1,k')$, r_1 elements of e_{ijk} and r_0 elements of c_{ijk} sent to processor $(i',k'+1)$, and r_1 elements of e_{ijk} sent to processor $(i'+1,k'+1)$. The same amount of data should also be received from processors $(i'-1,k')$, $(i'-1,k'-1)$, and $(i',k'-1)$. Finally, a total of $r_0 r_1$ elements of b_{ijk} should be loaded into the processor before program (6.12) can start.

CHAPTER 7

CONCLUSION AND FUTURE WORK

We have presented in this thesis the design, implementation, and modeling of pipelined data parallel algorithms on DMMs. In this chapter, we summarize what we have discussed and highlight major contributions of the work reported here. Next, improvements to the current work will be suggested, impacts of new techniques and architectures will be assessed, and plans for future work will be outlined.

7.1. SUMMARY

In Chapter 1, features of current DMMs were reviewed, from which important system bottlenecks were identified. Special considerations are required when one attempts to design efficient parallel algorithms on DMMs. Different styles of parallel computations were discussed, and pipelined data parallel algorithms were singled out to be the primary subject of this thesis. Then, in Chapter 2, starting with a representation of parallel computation, the P-nets, we discussed various considerations in designing parallel algorithms on DMMs. The implementation and modeling of two data parallel algorithms were given in Chapter 3 to illustrate the tradeoffs in designing parallel algorithms in terms of computation and communication as well as time and space.

With these tradeoffs in mind, a new parallel algorithm design paradigm called large-grain pipelining was introduced in Chapter 4. Large-grain pipelining exploits

node level macro-pipelining to increase the degree of overlapping between computation and communication. Examples of pipelined matrix multiplication algorithms were given to illustrate the application of large-grain pipelining.

As discussed in Section 4.3, the unique feature of large-grain pipelining is the focus on balancing computation with communication by forming data blocks with appropriate sizes. These data blocks in turn form data streams which flow in the system to produce pipeline effects. Operations in the system are scheduled according to the flows of data, i.e., data-driven scheduling. Furthermore, the macro-pipelining in pipelined data parallel algorithms increases the degree of overlapping between nodes while reducing the I/O bandwidth in accessing data. The net effect is a highly parallel computation.

In Chapter 5, a model for analyzing pipelined data parallel computation on DMMs was presented. The model views a pipelined computation as consisting of elementary computational units, which are interconnected through data flows. The throughput of individual computational units as well as linear arrays and 2-dimensional meshes is analyzed by studying the time events represented as Timed Petri-nets. The analyzed results of the pipelined matrix multiplication algorithms were compared with experimental results obtained from a 64-node NCUBE multiprocessor. The close match between these two sets of data indicates the accuracy of the analytic model.

The accuracy stems from two sources: (1) the capability of the analytic model in describing the behavior of a computational unit, and in modeling asynchronous and deterministic events; and (2) the accuracy in estimating system parameters, such as the message transmission time and primitive computation time. Note that the analytic model can be automated and be applied to a more general class of computations.

In Chapter 6, a systematic approach for designing pipelined data parallel algorithms on DMMs was introduced. The design procedure takes a shift-invariant nested-loop program as its input and produces a pipelined data parallel algorithm through a

series of transformations. From the relations between loop dependence vectors, we studied properties of grouping vertices in the computational structure. The major result obtained is that it is possible to group vertices in a 2-dimensional computational structure in such a way that the resultant contracted structure is a universal planner array (see Section 6.6). The extension to computational structures with higher dimension is possible.

The major contribution of the grouping concept and related theorems is, of course, to lead the way to a systematic approach for designing pipelined data parallel algorithms on DMMs. The grouping concept is unique in that it takes a totally different view of algorithm transformation from that in synthesizing systolic arrays, i.e., grouping versus projection. However, the results obtained in this thesis are also applicable to the latter, since projection is a special case of grouping. For example, the knowledge of grouping may assist in predicting the performance of systolic arrays derived from different projections.

Overall, it can be seen that we have presented a systematic and thorough study of pipelined data parallel algorithms — from the basic concept, performance modeling, to design and synthesis. Through the techniques introduced in this thesis, the resultant algorithms can achieve a balance between computation and communication. Furthermore, the systematic approaches used in these techniques hold the key to automated algorithm and program development. Various possibilities that stem from the research developed in this thesis will be probed further in the next section.

7.2. FUTURE WORK

From the discussion throughout this thesis, we can identify at least the following areas that need to be studied further:

(1) Study other applications of large-grain pipelining

We have presented pipelined matrix multiplication algorithms in this thesis. Due to their generality, it is expected that many problems can be solved by directly applying techniques developed here, especially those problems which have systolic array solutions. It is interesting to see how well the resultant algorithms perform compared to the existing algorithms. Another interesting topic would be the application of large-grain pipelining to problems which do not have a static and regular data structures such as those found in artificial intelligence.

(2) Study pipelined data parallel algorithms under different environments

We have studied pipelined data parallel algorithms mainly under the environment of message passing DMMs. It is interesting to see how and in what form pipelined data parallel algorithms can be run on a shared-memory multiprocessor. More importantly, since the shared-variable approach is a more natural way of programming, we should also study how large-grain pipelining can be incorporated into such an environment, perhaps in terms of reducing the amount of remote memory accesses.

(3) Assess the impact of future generation DMMs

Important trends in future generation DMMs have been discussed in Chapter 1, including larger memory, higher computing capability, improved computation to communication ratio, and concurrent I/O subsystems. It is expected that large-grain pipelining can benefit from these novel designs. For example, increasing the communication speed will decrease the block size and increase the number of blocks in a data stream. This is equivalent to increasing the vector length in a pipelined vector processor, which results in higher pipeline efficiency [HwBr84].

With reduced communication overhead, large-grain pipelining can fully exploit the overlapping between computation and communication, which is not possible in current implementations. In addition, more sophisticated communication primitives can be supported, which alleviate, say, the overhead in data gathering/scattering operations

(see Section 3.2). Examples include the modification of send and receive primitives to allow the user to specify the access patterns within a block of data, or the implementation of the broadcast primitive.

Pipelined data parallel algorithms can also take advantage of concurrent I/O subsystems, through which several streams of data can be directed into the system simultaneously. This can further reduce the bottleneck induced by the host. However, depending on how nodes access the disks, new issues may arise concerning how data is distributed on the disks and how access conflicts are resolved.

(4) Refine the grouping techniques

Most theorems presented in Chapter 6 are aimed at characterizing the properties of grouping, e.g., grouping along which direction will result in a dependence preserving grouping. We have not yet addressed the issue of choosing which data elements to pack together into one message, that of defining and determining the optimal grouping scheme, and the issue of implementing the ideas discussed. Mathematical formulations are also needed to determine the computation and communication requirements of a particular grouping. This information can be used to determine the granularity of the algorithm.

One important issue is to study the grouping of computational structures in higher dimensional space. Intuitively, the results presented in this thesis can be applied to higher dimensional spaces. However, a unified and rigorous mathematical treatment of all cases will be desirable, which should allow the extension of the theorems more easily.

It is interesting to study other program constructs which differ from shift-invariant nested-loop programs. Examples include the IF-THEN-ELSE construct and anything not enclosed within a loop. We should also consider the problem of how to incorporate pipelined data parallel computations into other styles of computation. This includes the interface with other nested loops in the program.

Finally, it is important to have a complete understanding of the relations between grouping and projection. The idea is to use grouping techniques to assist in synthesizing systolic arrays, and the goal is to predict the performance of the final systolic array from the knowledge of grouping.

(5) Implement the techniques

Most techniques presented in this thesis can be computer implemental. These tools should facilitate the development of parallel algorithms on DMMs.

(6) Develop an intelligent compiler for DMMs

The ultimate goal of the work presented in this thesis is to develop intelligent compilers on DMMs. The intelligent compiler takes a sequential program as its input and generates a parallel version suitable for DMMs. As a first step, a pre-processor to the compiler may be developed. The user provides information necessary for the parallelization as directives. The pre-processor attempts to parallelize the given program as much as possible and generates a transformed source program for compilation. Using the grouping techniques, many nested loops in the program can be parallelized and transformed into pipelined data parallel forms.

Intelligent compilers are complex and difficult to implement. However, the techniques presented in this thesis point to a new direction in achieving this goal. Additional research is needed to design more efficient parallel algorithms and to further advance our knowledge and capability in parallel processing.

BIBLIOGRAPHY

[AdCD74] T.L. Adam, K.M. Chandy, J.R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," *Comm. ACM*, Dec. 1974, pp. 685-690.

[BBN87] BBN Advanced Computers, Inc., *Butterfly Products Overview*, Oct. 1987.

[Berm87] F. Berman, "Experience with an Automatic Solution to the Mapping Problem," in *The Characteristics of Parallel Algorithms*, L.H. Jamieson, D.B. Gannon, R.J. Douglass, Eds., MIT Press, 1987.

[ChSm87] V. Cherkassky, R. Smith, "Efficient Mapping and Implementation of Matrix Algorithms on a Hypercube," *Technical Report*, Department of Electrical Engineering, Univ. of Minn., 1987.

[DaSe87] W.J. Dally, C.L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks, "*IEEE Trans. on Computers*, May, 1987, pp. 547-553.

[Duni87] T.H. Duningan, "Hypercube Performace," *Hypercube Multiprocessors 1987*, Ed., M.T. Heath, SIAM, 1987, pp. 178-191.

[FaSh87] N. Faroughi, M.A. Shanblatt, "Systematic Generation and Enumeration of Systolic Arrays from Algorithms," *Proc. of the 1987 Int'l Conf. on Parallel Processing*, Aug. 1987, pp. 844-847.

[Fink87] R.A Finkel, "Large-Grain Parallelism — Three Case Studies," in *The Characteristics of Parallel Algorithms*, L.H. Jamieson, D.B. Gannon, R.J. Douglass, Eds., MIT Press, 1987.

[FiFi82] J.P. Fishburn, R.A. Finkel, "Quotient Networks," *IEEE Trans. on Computers*, Apr., 1982, pp. 288-295.

[FoFW85] J.A.B. Fortes, K.S. Fu, B.W. Wah," Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays," *Proc. of 1985 Int'l Conf. on Acoustics, Speech, and Signal Processing*, 1985, pp. 300-303.

[FoMo84] J.A.B. Fortes, D.I. Moldovan, "Data Broadcasting in Linearly Scheduled Array Processors," *Proc. of Symp. on Computer Architecture*, 1984, pp. 224-231.

[FoWa87] J. Fortes, B.W. Wah, "Systolic Arrays — From Concept to Implementation," *IEEE Computer*, July, 1987, pp. 12-17.

[FoOH87] G.C. Fox, S.W. Otto, A.J. Hey, "Matrix Algorithms on a Hypercube I: Matrix Multiplication," *Parallel Computing*, Jan. 1987, pp. 17-31.

[GaJo79] M.R. Garey, D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman & Company, 1979.

[Gonz77] M.J. Gonzalez, Jr., "Deterministic Processor Scheduling," *Computing Survey*, Sep. 1977, pp.173-204.

[GrRe86] D.C. Grunwald, D.A. Reed, "Benchmarking Hypercube Hardware and Software," *Technical Report*, UIUCDCS-R-86-1303, Department of Computer Science, University of Illinois at Urbana-Champaign, 1986.

[HaMS86] J.P. Hayes, T.N. Mudge, Q.F. Stout, S. Colley, J. Palmer, "Architecture of a Hypercube Supercomputer," *Proc. Int'l Conf. on Parallel Processing*, Aug. 1986, pp. 653-660.

[HiSt86] W.D. Hillis, G.J. Steele, Jr., "Data Parallel Algorithms," *Comm. ACM*, Dec. 1986, pp. 1170-1183.

[HoTa85] J.J. Hopfield, D.W. Tank, "Neural Computation of Decisions in Optimization Problems," *Biological Cybernetics*, 1985, pp. 141-152.

[Hu61] T.C. Hu, "Parallel Sequening and Assembly Line Problems," *Management Science*, Nov. 1961, pp. 841-848.

[HwBr84] K. Hwang, F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill Book Co., 1984.

[Hwan87] K. Hwang, "Advanced parallel processing with supercomputer architectures," *Proc. of the IEEE*, Oct. 1987, pp. 1348-1379.

[Jord87] H.F. Jordan, "The Force," in *The Characteristics of Parallel Algorithms*, L.H. Jamieson, D.B. Gannon, R.J. Douglass, Eds., MIT Press, 1987.

[KeLi70] B.W. Kernighan, S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal*, Feb. 1970, pp. 291-307.

[KiCN88a] C.T. King, W.H. Chou, L.M. Ni, "Large-Grain Pipelining on Distributed-Memory Multiprocessors," *Proc. of the Third Int'l Conf. on Supercomputing*, May, 1988.

[KiCN88b] C.T. King, W.H. Chou, L.M. Ni, "Pipelined Data Parallel Algorithms — Concept and Modeling," *Proc. of the 1988 ACM Int'l Conf. on Supercomputing*, July, 1988.

[KiGN88] C.T. King, T.B. Gendreau, L.M. Ni, "Reliable Elections in Broadcast Networks," to appear in *Journal of Parallel and Distributed Computing*.

[KiNi88] C.T. King, L.M. Ni, "Large grain Pipelining on Hypercube Computers," *Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications*, 1988.

[KiGV83] S. Kirkpatrick, C.D. Gelatt, Jr., M.P Vecchi, "Optimization by Simulated Annealing," *Science*, May 1983., pp. 671-680.

[KuLe78] H.T. Kung, C.E. Leiserson, "Systolic Arrays (for VLSI)," *Sparse Matrix Proc.*, 1978, pp. 32-63.

[KuLJ87] S.Y. Kung, S.C. Lo, S.N. Jean, J.N. Hwang, "Wavefront Array Processors — Concept to Implementation," *IEEE Computer*, July 1987, pp. 18-33.

[MaLT82] P.R. Ma, E.Y. Lee, M. Tsuchiya, "A Task allocation Model for Distributed Computing Systems,"*IEEE Trans. Computers*, Jan. 1982, pp. 41-47.

[MiWi84] W.L. Miranker, A. Winkler, "Spacetime Representations of Computational Structures," *Computing*, 1984, pp. 93-114.

[MoFo86] D.I. Moldovan, J.A.B. Fortes, "Partitioning and Mapping Algorithms into Fixed-Size Systolic Array," *IEEE Trans. on Computers*, Jan. 1986, pp. 1-12.

[MuBA87] T.N. Mudge, G.D. Buzzard, T.S. Abdel-Rahman, "A High Performance Operating System for the NCUBE," *Hypercube Multiprocessors 1987*, Ed., M.T. Heath, SIAM, 1987.

[NeSn87] P.A. Nelson, L. Snyder, "Programming Paradigms for Nonshared Memory Parallel Computers," in *The Characteristics of Parallel Algorithms*, L.H. Jamieson, D.B. Gannon, R.J. Douglass, Eds., MIT Press, 1987.

[NiKP87] L.M. Ni, C.T. King, P. Prins, "Parallel Algorithm Design Considerations for Hypercube Multiprocessors," *Proc. of 1987 Int'l Conf. on Parallel Processing*, 1987, pp. 717-720.

[Oste87] A. Osterhaug, *Guide to Parallel Programming on Sequent Computer Systems*, Sequent Computer Systems, Inc., 1987.

[Pete77] J.L. Peterson, "Petri Nets," *Computing Survey*, Sep. 1977, pp. 223-252.

[PaWo86] D.A. Padua, M.J. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Comm. of ACM*, Dec., 1986, pp. 1184-1201.

[PoBa87] C.D. Polychronopoulos, U. Banerjee, "Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds," *IEEE Trans. Computers*, Apr. 1987, pp. 410-420.

[Quin87] M.J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill Book Co., 1987.

[Ramc74] C. Ramchandani, "Analysis of Asynchronous Concurrent Systems by Timed Petri Nets," Ph.D. Dissertation, Project MAC, MAC-TR-120, MIT, 1974.

[RaSH79] G.S. Rao, H.S. Stone, T.C. Hu, "Assignment of Tasks in a Distributed Processor System with Limited Memory,"*IEEE Trans. Computers*, Apr. 1979, pp. 291-298.

[SaSc85] Y. Saad, M.H. Schultz, "Topological Properties of Hypercubes," *Technical Report*, YALEU/DCS/RR-389, Department of Computer Science, Yale University, June 1985.

[ShFi87] Y. Shih, J. Fier, "Hypercube Systems and Key Applications," in *Parallel Processing for Supercomputing and AI*, K. Hwang, D. DeGroot, Eds., 1987.

[Squi86] M.L. Squires, "Applying Parallel Processing," *COMPCON Spring 86*, 1986, pp. 394-396.

[WiCM88] A. Witkowski, K. Chandrakumar, G. Macchio, "Concurrent I/O System for the Hypercube Multiprocessor," *The Third Conf. on Hypercube Concurrent Computers and Applications*, Jan., 1988.

APPENDIX

```
/*
 * Array summation using tree-structured accumulation method with ratioed partitioning
 */

#include <stdio.h>
#define MAXNODE 64      /* Maximum number of processors used */
#define MAXARRAY 100000 /* Maximum size of the array */

log2(n)  /* Calculate the integer logrithm of an integer number */
int n;
{  int x=1;
   n--;   while(n = n>>1) x++;
   return(x);
}

main(argc,argv)  /* The program running in the host */
int argc; char *argv[];
{  int i,j,k,*ipt;
   int m;      /* Size of the array */
   int n,d;    /* Number of nodes used and the dimension */
   float a[MAXARRAY];  /* The array */
   float sum,time,*fpt;
   int x[MAXNODE];     /* The size of the subarray in each processor */
   int ch,type,nd,status,nflags;       /* Communication parameters */
   FILE *fopen(),*fp1,*fp2;

   /* Initialization */
   m=atoi(argv[1]);    n=atoi(argv[2]);    d=log2(n);

   /* Input the partition sizes from the file 'part' */
   fp1=fopen("part","r");
   for(i=0,j=0,ipt=x; i<n; i++) {
       scanf(fp1,"%d ",ipt);    j += *(ipt++);}

   /* Initialize the cube and load node program from the file 'node' */
   ch=nopen(d);
   status=nloadm(ch,"node",(-1)*n,nflags,a,1024*sizeof(float));

   /* Input the array from the file 'data' */
   fp2=fopen("data","r");
   for(i=0,fpt=a; i<m; i++) fscanf(fp2,"%f",fpt++);

   /* Inform processor 0 to start couting the time */
   nwrite(ch,a,sizeof(float),0,90);

   /* Load the subarrays */
   for(i=0,fpt=a,ipt=x; i<n; i++,ipt++) {
       status=nwrite(ch,fpt,(*ipx)*sizeof(float),i,10);
       fpt += (*ipt); }
```

```
  /* Receive the resultant total sum */
  type=40;    nd=0;    nread(ch,&sum,sizeof(float),&nd,&type);

  /* Inform processor 0 to stop counting the time */
  nwrite(ch,a,sizeof(float),0,95);

  /* Receive the total execution time from processor 0 */
  type=50;    nd=0;    k=nread(ch,&time,sizeof(float),&nd,&type);
  printf("total sum is %7.1f  total execution time is %f ticks\n",sum,time);
}




/*
 * Array summation using tree-structured accumulation with ratioed partitioning --- node program
 */

#define MAXARRAY 4096  /* Maximum size of a subarray */

main()
{  int i,j,k,diff,ll;
  int m;            /* Size of the subarray */
  int root=0;         /* Root of the reduciton tree */
  int nd,id,dim,pid,host,type,flag;   /* Parmeters for communication */
  int time;           /* Execution time */
  float ai[MAXARRAY];  /* The subarray */
  float sum=0,tmp;

  /* Get environment information */
  whoami(&id,&pid,&host,&dim);

  /* If processor 0, then start couting the time */
  if (id == 0) {
      type=90;    nread(ai,sizeof(float),&host,&type,&flag);
      time=ntime(); }

  /* Receive the subarray */
  type=10;    m=nread(ai,MAXARRAY*sizeof(float),&host,&type,&flag);
  m = m/4;

  /* Subarray summation */
  for(i=0; i<m; i++) sum += ai[i];

  /* Tree reduction to perform the final accumulation */
  diff=id^root;
  for(i=1,ll=1; i<d; i++) ll = ll << 1;
  while(ll>diff) {
      nd=(-1);    type=30;
      nread(&tmp,sizeof(float),&nd,&type,&flag);
      sum += tmp;   ll = ll>>1; }
  if(ll != 0) {
      nd=id^ll;   nwrite(&sum,sizeof(float),nd,30,&flag); }
  else  /* I am the root, send back to the host */
```

```
      nwrite(&sum,sizeof(float),host,40,&flag);

  /* If processor 0, then stop counting the time */
  if (id == 0 ) {
      type=95;   nread(ai,sizeof(float),&host,&type,&flag);
      ai[0]=ntime() - time;
      nwrite(ai,sizeof(float),host,50,&flag); }
}



/*
 * Matrix multiplication using Algorithm 3.1
 */

#include <stdio.h>
#define MAXMTX 64   /* Maximum size of the matrices */

gray(n)                    /* Gray code of n */
int n;
{ return((n>>1)^n); }

ginv(n)                    /* Inverse of the gray code n */
int n;
{ int x=n;
  while(n = n>>1) x = x^n;
  return(x); }

log2(n)             /* Calculate the integer logrithm of an integer */
int n;
{ int x=0;
  while(n = n>>1) x++;
  return(x); }

main(argc,argv)
int argc; char *argv[];
{ int i,j,k,l,u,v,x,y;
  int m,n;          /* Size of the matrices and number of processors */
  int n1,n2,n3;     /* Number of partitions */
  int d1,d2;        /* Dimension of the subcubes */
  int s12,s13,s1,s2,s3;          /* Size of submatrices */
  int ch,type,nd,status,nflags;  /* Communication parameters */
  float a[MAXMTX][MAXMTX],b[MAXMTX][MAXMTX];      /* Matrices A and B */
  float buf[4097];               /* Buffer for loading */
  FILE *fopen(),*fp1;

  /* Initialization */
  m=atoi(argv[1]);  n1=atoi(argv[2]);  n2=atoi(argv[3]);  n3=atoi(argv[4]);
  n=n1*n2;  d1=log2(n1);  d2=log2(n2);
  s1=m/n1; /* Number or rows in one partition in A */
  s2=m/n2; /* Number or columns in one partition in A */
  s3=m/n3;
  s12=s1*s2;       /* Size of the submatrix in A: (n1,n2) partition */
  s13=s1*s1;       /* Size of the submatrix in C: (n1,n1) partition */
```

```
/* Read the input matrices */
fp1=fopen("data","r");
for(i=0; i<m; i++)
   for(j=0; j<m; j++) fscanf(fp1,"%f",&a[i][j]);
for(i=0; i<m; i++)
   for(j=0; j<m; j++) fscanf(fp1,"%f",&b[i][j]);

/* Initialize the cube */
ch=nopen(d1+d2);
status=nloadm(ch,"node",(-1)*n,nflags,buf,1024*sizeof(float));

/* Load initialization parameters */
buf[0]=m;   buf[1]=n1;   buf[2]=n2;   buf[3]=n3;
for(i=0; i<n; i++) nwrite(ch,buf,4*sizeof(float),i,10);

/* Inform processor 0 to start counting the time */
nwrite(ch,buf,sizeof(float),0,90);

/* Load matrix A and B to the cube, size of s12 each */
for(u=0; u<n1; u++)
  for(v=0; v<n2; v++) {
    i=(gray(u)<<d2)+v;       /* Node id to which the submatrix is loaded */
    x=(u+1)*s1; y=(v+1)*s2;
    for(j=x-s1,l=0; j<x; j++)
       for(k=y-s2; k<y; k++,l++) buf[l]=a[j][k];
    status=nwrite(ch,buf,s12*sizeof(float),i,15);
    for(k=x-s1,l=0; k<x; k++)
       for(j=y-s2; j<y; j++,l++) buf[l]=b[j][k];
    status=nwrite(ch,buf,s12*sizeof(float),i,20);
}

/* Receive results */
for(i=0; i<n1*n1; i++) {
   type=40;   nd=-1;   nread(ch,buf,(s13+1)*sizeof(float),&nd,&type);
   x=(ginv(nd>>d2)+1)*s1;  y=(buf[s13]+1)*s3;
   for(j=x-s1,l=0; j<x; j++)
      for(k=y-s3; k<y; k++,l++) a[j][k]=buf[l];
}

/* Inform processor 0 to stop counting the time */
nwrite(ch,buf,sizeof(float),0,95);

/* Print out matrix C */
printf("The resultant matrix C\n");
for(i=0; i<m; i++) {
   for(j=0; j<m; j++) printf("%3.0f ",a[i][j]);
   printf("\n"); }

/* Receive the execution time from processor 0 */
type=-1;   nd=0;   k=nread(ch,buf,sizeof(float),&nd,&type);
printf("Execution time is %f ticks\n",buf[0]);
}
```

```
/*
 * Matrix multiplication using Algorithm 3.1 --- node program
 */

gray(n)             /* Gray code of n */
int n;
{ return((n>>1)^n); }

ginv(n)             /* Inverse gray code of n */
int n;
{ int x=n;
  while(n = n>>1) x = x^n;
  return(x); }

log2(n)       /* Calculate the integer logrithm of an integer */
int n;
{ int x=0;
  while(n = n>>1) x++;
  return(x); }

main()
{ int i,j,k,l,ll,idd,diff,root,max;
  int m,n1,n2,n3,d,d1,d2,s12,s13,s1,s2;
  int id,id1,id2,su,nd,pid,host,type,flag;
  float ai[2048],bi[2048],ci[2049],buf[2048]; /* Buffers */

  /* Receive the initialization parameters from the host */
  whoami(&id,&pid,&host,&d);     /* Get environment info */
  type=10;    nread(ai,4*sizeof(float),&host,&type,&flag);
  m=ai[0];    n1=ai[1];    n2=ai[2];    n3=ai[3];
  d1=log2(n1);    d2=log2(n2);
  s1=m/n1;    s2=m/n2;    s12=s1*s2;    s13=s1*s1;

  /* Find predecessor and successor nodes in the ring */
  id1=id>>d2;  id2=id & (n2-1);
  k = id1 ? gray(ginv(id1)-1) : gray(n1-1);
  su=(k<<d2) | id2;

  /* Processor 0 starts counting time */
  if (id == 0) {
     type=90;    nread(ai,sizeof(float),&host,&type,&flag);
     max=ntime(); }

  /* Receive submatrix A */
  type=15;    nread(ai,s12*sizeof(float),&host,&type,&flag);

  /* Start submatrix multiplication */
  root= (id1<<d2);  /* Root of the reduction tree */
  diff=id^root;    pid=ginv(id1)-1;
  for(l=0; l<n1; l++) {
     /* Receive submatrix B */
     nd = l ? (-1) : host;
     type=20;    nread(bi,s12*sizeof(float),&nd,&type,&flag);
```

```
    /* Submatrix multiplication */
    for(i=0,ll=0; i<s12; i+=s2)
       for(j=0; j<s12; j+=s2,ll++)
          for(k=0,ci[ll]=0; k<s2; k++) ci[ll] += ai[i+k]*bi[k+j];

    /* Shift out submatrx B */
    if(l<n1-1) nwrite(bi,s12*sizeof(float),su,20,&flag);

    /* Tree reduction to find the complete submatrices of C */
    ll = n2 >> 1;
    while(ll>diff) {
       nd=(-1);  type=30+l;    nread(buf,s13*sizeof(float),&nd,&type,&flag);
       for(i=0; i<s13; i++) ci[i] += buf[i];
       ll = ll>>1; }
    if(ll != 0) {
       nd=id^ll;    nwrite(ci,s13*sizeof(float),nd,30+l,&flag); }
    else { /* I am the root, send back to the host */
        ci[s13] = pid = (pid+1) % n1;
       nwrite(ci,(s13+1)*sizeof(float),host,40,&flag); }
  }

  /* Processor 0 sends the execution time bacj to the host */
  if (id == 0) {
    type=95;    nread(ai,sizeof(float),&host,&type,&flag);
    buf[0]=ntime() - max;
    nwrite(buf,sizeof(float),host,50,&flag); }
}



/*
 * Matrix multiplication using Algorithm 4.2
 */

#include <stdio.h>
#define MAXMTX 64      /* Maximum size of the matrices */

log2(n)                    /* Calculate the integer logorithm of an integer */
int n;
{ int x=0;
  while(n = n>>1) x++;
  return(x); }

main(argc,argv)
int argc; char *argv[];
{
  int  i,j,k,l,u,v,x,y;
  int  ch,type,nd,nflags; /* Communication parameters */
  int  m,n;          /* Size of the matrices and number of processors */
  int  n1,n2,n3;    /* Number of partitions */
  int  d1,d2;              /* Dimension of the subcubes for n1, n2 */
  int  s12,s13,s23,s1,s2,s3;          /* Size of submatrices */
  float buf[4096];              /* Message buffer */
  float a[MAXMTX][MAXMTX],b[MAXMTX][MAXMTX];      /* Matrices A and B */
```

```
FILE    *fopen(),*fp1;

/* Initialization */
m=atoi(argv[1]);  n1=atoi(argv[2]);  n2=atoi(argv[3]);  n3=atoi(argv[4]);
n=n1*n2;  d1=log2(n1);  d2=log2(n2);
s1=m/n1;        /* Number or rows in one partition in A */
s2=m/n2;        /* Number or columns in one partition in A */
s3=m/n3;
s12=s1*s2;          /* Size of the submatrix in A */
s13=s1*s3;
s23=s2*s3;          /* Size of the submatrix in B */

/* Read the input matrices */
fp1=fopen("data","r");
for(i=0; i<m; i++)
    for(j=0; j<m; j++) fscanf(fp1,"%f",&a[i][j]);
for(i=0; i<m; i++)
    for(j=0; j<m; j++) fscanf(fp1,"%f",&b[i][j]);

/* Initialize the cube and load the program */
ch = nopen(d1+d2);
nloadm(ch,"node",(-1)*n,nflags,buf,4096*sizeof(float));

/* Load initialization parameters */
buf[0]=m; buf[1]=n1; buf[2]=n2; buf[3]=n3;
for(i=0; i<n; i++) nwrite(ch,buf,4*sizeof(float),i,10);

/* Inform processor 0 to start counting the time */
nwrite(ch,buf,sizeof(float),0,90);

/* Load matrix A to the cube, size of s12 each */
for(u=0; u<n1; u++)
    for(v=0; v<n2; v++) {
        i=(u<<d2)+v;  /* Node id */
        x=(u+1)*s1;  y=(v+1)*s2;
        for(j=x-s1,l=0; j<x; j++)
            for(k=y-s2; k<y; k++,l++) buf[l]=a[j][k];
        nwrite(ch,buf,s12*sizeof(float),i,20); }

/* Load B in a pipeline fashion */
for(u=0; u<n3; u++)
    for(v=0; v<n2; v++) {
        x=(u+1)*s3;  y=(v+1)*s2;
        for(k=x-s3,l=0; k<x; k++)
            for(j=y-s2; j<y; j++,l++) buf[l]=b[j][k];
            nwrite(ch,buf,s23*sizeof(float),v,25); }

/* Receive results */
for(i=0; i<n1*n3; i++) {
    type=40;  nd=(-1);  k=nread(ch,buf,(s13+1)*sizeof(float),&nd,&type);
    /* Assembling C from each submatrix received */
    y=((nd>>d2)+1)*s1;  x=(buf[s13]+1)*s3;
    for(j=y-s1,l=0; j<y; j++)
        for(k=x-s3; k<x; k++,l++) a[j][k]=buf[l]; }
```

```
  /* Inform processor 0 to stop counting the time */
  nwrite(ch,buf,sizeof(float),0,95);

  /* Print out the resultant C */
  for(i=0; i<m; i++) {
       for(j=0; j<m; j++) printf("%3.0f ",a[i][j]);   printf("\n"); }

   /* Receive the execution time from processor 0 */
   type=-1;  nd=0;  k=nread(ch,buf,sizeof(float),&nd,&type);
   printf("Total execution time is %f ticks\n",buf[0]);
}



/*
 * Matrix multiplication using Algorithm 4.2 --- the node program
 */

gray(n)              /* Gray code of n */
int n;
{ return((n>>1)^n); }

ginv(n)              /* Inverse gray code of n */
int n;
{ int x=n;
  while(n = n>>1) x = x^n;
  return(x); }

log2(n)              /* Calculate the integer logorithm of an integer */
int n;
{ int x=0;
  while(n = n>>1) x++;
  return(x); }

main()
{ int i,j,k,l,ll;
  int m,n1,n2,n3,s12,s13,s23,s1,s2,s3,d1,d2;
  int nd,pid,host,type,flag,id,id1,id2,idg,diff,su,pr,maxt;
  float ai[4096],bi[4096],ci[4097];  /* Buffers */

  /* Receive the initialization parameters from the host */
  whoami(&id,&pid,&host,&diff);     /* Get environment info */
  type=10;    nread(ai,4*sizeof(float),&host,&type,&flag);
  m=ai[0];    n1=ai[1];    n2=ai[2];    n3=ai[3];
  d1=log2(n1);       d2=log2(n2);
  s1=m/n1;    s2=m/n2;    s3=m/n3;    s12=s1*s2;    s13=s1*s3;    s23=s2*s3;

  id1=id>>d2;  id2= id & (n2-1);
  su = (id2 == n2-1) ? host : id+1;
  idg=ginv(id1);    k=gray(idg+1);    pr=(k<<d2)|id2;

  /* Processor 0 starts counting the time */
  if (id == 0) {
```

```
    type=90;    nread(ai,4,&host,&type,&flag);
    maxt=ntime(); }

  /* Receive submatrix A */
  type=20;    nread(ai,s12*sizeof(float),&host,&type,&flag);

  /* Start the pipeline: there will be n3 submatrices of B coming in */
  diff=id^(id1 << d2);
  for(l=0; l<n3; l++) {
    /* Receive a submatrix of B */
    nd=(-1);  type=id1 ? 27:25;
    nread(bi,s23*sizeof(float),&nd,&type,&flag);

    if(idg != (n1-1)) nwrite(bi,s23*sizeof(float),pr,27,&flag);

    /* Submatrix multiplication */
    for(i=0,ll=0; i<s12; i+=s2)
      for(j=0; j<s23; j+=s2,ll++) {
        for(k=0,ci[ll]=0; k<s2; k++) ci[ll] += ai[i+k]*bi[k+j];
    }

    /* Accumulate submatrices of C using a binary tree */
    ll = n2 >> 1;
    while(ll>diff) {
      nd=(-1);  type=30+l;    nread(bi,s13*sizeof(float),&nd,&type,&flag);
      for(i=0; i<s13; i++) ci[i] += bi[i];
      ll = ll>>1; }
    if(ll != 0) {
      nd=id^ll;    nwrite(ci,s13*sizeof(float),nd,30+l,&flag); }
    else { /* I am the root, send back to the host */
        ci[s13] = l;   nwrite(ci,(s13+1)*sizeof(float),host,40,&flag); }
  }

  /* Processor 0 stops counting the time */
  if (id == 0) {
    type=95;   nread(ai,4,&host,&type,&flag);
    ai[0]=ntime()-maxt;
    nwrite(ai,sizeof(float),host,50,&flag); }
}
```