On Compiling Regular Loops for Efficient Parallel Execution

Pei Ouyang

A Dissertation Submitted in Partial Fulfillment

of the Requirements for the Degree of

Doctor of Philosophy

Department of Computer Science

New York University

February, 1992

Approved: ____________________________

Zvi M. Kedem, Research Advisor

____

Krishna V. Palem, Research Advisor

To My Wife and Parents

Acknowledgments

I would like to express my deepest gratitude to my advisors Zvi Kedem and Krishna Palem for their guidance, supports, and encouragements over these years. My special thanks are to Ben Goldberg, Allan Gottlieb, and Bud Mishra for being in my Defense Committee, as well as Jiazhen Cai, Bob Paige, and Ken Perlin for being in my Oral Committee. They have helped me not only at the committee, but also throughout my graduate school years. I am also grateful to all of the friends I met in NYU. Their companionship made the stay in New York City a pleasant experience. In particular, my thanks are due to Chih-Hung Hsieh and Peizong Lee for their helpful discussions. The fellowships from the Ministry of Education, Taiwan, R.O.C. and the Courant Institute of Mathematical Sciences are also highly appreciated. Finally, I am indebted to my brothers and sisters for their constant encouragement, as well as my parents for their love and persistent support. Especially I am obliged to my dear wife for her understanding and diligence. Together we went through our enjoyable times and tough roads. In particular, she has been taking care of our lovely daughter for over two years by herself without any complaint. I feel happy to share everything with her in my life.

Abstract

In this thesis, we study the problem of *mapping* regular loops onto multiprocessors. We develop mapping schemes that yield very efficient executions of regular loops on *shared* and *distributed* memory architectures. We also develop novel analysis techniques, using which we argue about the efficiency of these resulting executions. The quality of the execution of these regular loops in the distributed memory setting, relies heavily on implementing *cyclic shifts* efficiently. Effectively, cyclic shifts are used to communicate results between individual processors, to which different interdependent iterations are assigned. Therefore, in order to achieve efficient executions of regular loops on distributed memory architectures, we also develop and analyze algorithms for solving the cyclic shift problem.

In order to analyze the executions of regular loops that result from any specific mapping, we need to characterize the important parameters that determine its efficiency. We formally characterize a basic set of such parameters. These parameters facilitate the analysis of the *memory* and the *processor* requirements of a given execution, as well as its *running time*. Using these parameters, we analyze a *greedy* execution scheme, in the shared memory model. For example, we can determine the limit on the number of processors beyond which no speedup can be attained by the greedy method, for regular loops. The greedy scheme is of interest because it exploits the maximal possible parallelism in a natural way.

We then address the mapping scheme of regular loops onto distributed memory machines. Unfortunately, we show that the problem of finding an optimal mapping is computationally intractable in this case. In order to provide schemes that can be actually applied to regular loops at compile-time, we relax the requirement that the resulting executions be optimum. Instead, we design a heuristic mapping algorithm and validate it through experiments. This heuristic mapping scheme relies heavily on the use of efficient algorithms for realizing cyclic shifts. Therefore, we also study the problem of realizing cyclic shifts on hypercube architectures.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis studies issues concerning the execution of *regular loops* on *multiprocessor computers.*

1.1 Programming Languages and Regular Loops

In the parallel programming community, many languages have been considered for writing programs for parallel machines. These languages include conventional languages (such as Fortran, C, Pascal, Lisp, Prolog), conventional languages with augmented parallel constructs (such as Fortran 8X, Concurrent C, Pascal-m, Concurrent Lisp, Concurrent Prolog), and languages with built-in parallel constructs (such as Ada and CSP) [6] [8] [9] [12] [29] [115] [131].

A program in one of the above languages is compiled into object code for parallel execution. For a language with augmented parallel constructs or with built-in parallel constructs, the compiler only needs to consider the explicit parallel constructs. For a conventional sequential language, the compiler need to identify the source of possible parallelism and then convert the "sequential" constructs into "parallel" constructs. It is well known that loops constitute the major parallelism of a program in a sequential language.

In this thesis, we confine ourselves to the loop constructs in sequential languages. More specifically, the *regular loop* construct is considered in this thesis. Chapter 2 gives the formal definition of regular loops.

The motivation for studying regular loops is twofold. First, many utility programs have been implemented in conventional languages such as Fortran. Hence parallelizing this "sequential" construct can save much work for writing new utilities in parallel languages. Second, it has been shown that many algorithms can be formulated by regular loops [84]. That means the success of this research will facilitate solving a large amount of problems.

1.2 Parallel Computers and Multiprocessors

Computers can be divided into four categories [47]:

- Single Instruction Stream – Single Data Stream (SISD)
- Single Instruction Stream – Multiple Data Stream (SIMD)
- Multiple Instruction Stream – Single Data Stream (MISD)
- Multiple Instruction Stream – Multiple Data Stream (MIMD)

In this thesis, the computers considered are MIMD machines, for which we sometimes use the term multiprocessors.

A multiprocessor consists of a set of interconnected processing elements (PE) working independently and cooperatively. The PE's work independently in the sense that each PE may execute different operations with different operands; the PE's work cooperatively in the sense that sometimes they may have to communicate and synchronize with one another. The communication is achieved by passing messages through an *interconnection network*. Typical interconnection networks include the omega network (in butterfly, shuffle-exchange, Beneš, or banyan connection patterns), tree, mesh, mesh-of-trees, and hypercube [11] [133]. Extensive work has been done on routing algorithms [69] [81] [134], simulations among different networks [5] [18] [19] [27] [34] [75] [76] [85] [89] [107] [117] [119] [136], and architectural supports for routing algorithms [38] [53] [57] [58].

According to their communication mechanism, multiprocessors can be further divided into two classes: shared-memory multiprocessors and distributed-memory multiprocessors. In shared-memory multiprocessors, a global memory can be accessed by all the PE's through an interconnection network. Each PE communicates with other PE's via the shared global memory. For example, in Ultracomputer [53] [118], a region of the shared memory is used to store the ready processes. When a PE becomes free, it will access this region and fetch a process to execute. In distributed-memory multiprocessors, no global memory is available and each PE has its own local memory. PE's communicate with one another by *message passing*. For example, in Ncube/ten, one PE can communicate with another PE by sending out a message (using the library function *nread*, *ntest*, or *nwrite*) to one of its neighboring PE's, which in turn forwards this message to its neighboring PE. Via several forwardings, the message will finally reach the target PE [109].

In this thesis, both shared-memory and distributed-memory multiprocessors are considered. For shared-memory multiprocessors, we study the efficiency of a greedy scheduling scheme (Chapter 4), by applying the a set of properties we derive (Chapter 3). For distributed-memory multiprocessors, we devise an (interconnection) network independent scheduling scheme (Chapter 6), which follows our formal scheduling scheme specification (Chapter 5).

1.3 Parallelizing Compilers

Executing regular loops on multiprocessors is facilitated by *parallelizing compilers* [6] [8]. Generally speaking, a parallelizing compiler consists of four phases:

1. Lexical, syntactic, and semantic analysis
2. Preprocessing for dependence analysis
3. Dependence analysis
4. Code generation and optimization

Phase 1 can be implemented by using traditional compiler techniques [1], and produces an intermediate form of the source program. Phase 2 preprocesses the intermediate form to facilitate the data dependence analysis in phase 3. Typical preprocessing tasks include loop normalization, induction variable recognition, and wraparound variable recognition [6] [98]. Phase 3 plays an important role in parallelizing compilers, since the more data independence among statements can be found, the more probabilities that the program can be parallelized. Typical data dependence tests include the greatest common divisor (GCD) test, Banerjee Test, Shostak Test, and interprocedure tests [13] [14] [15] [20] [21] [51] [86] [110] [125] [139]. Finally, phase 4 actually generates the object codes, which is also optimized for efficient parallel execution. Typical code optimization includes loop interchanging, strip mining, loop collapsing, loop fission, loop fusion, recursive breaking, and others [7] [8] [10] [45] [78] [83] [98] [102] [141] [142].

In this thesis, our concentration is on Phase 4. That is, given a regular loop which has been preprocessed by Phase 1 to 3, and a multiprocessor, we investigate how to *map* the *loop structure* onto the *multiprocessor (interconnection network) structure*, so that the parallel execution is efficient.

1.4 Related Works

Executing loops in parallel has been done extensively for various architectures. Possibly the fastest method to execute a loop is to design special purpose hardware (e.g. a systolic array) to execute the loop [80] [82] [84] [91]. By using several processing elements connected in a specific way, computations of a loop are distributed among these processing elements so that data can flow among them in a rhythmic way without conflict. However, due to the cost of the special purpose hardware, software solutions are usually preferred.

In a vector processor, vector operations are used to execute loops [4] [8] [22] [98] [129]. These kinds of loops are usually written for numerical computations, such as matrix multiplication, that the parallelism can be exploited across *different* iterations. However, the parallelism inside the *same* iterations is hard to exploit by a vector processor. VLIW processors solve this problem by compacting different instructions into a very long instruction word. With such a mechanism, both the parallelism across iterations and inside the same iterations can be exploited [3] [33] [46] [143].

In a multiprocessor environment, more possibilities are available for executing loops [42] [56] [103] [104] [112] [114] [116] [126] [130] [140]. If all the processing elements of a multiprocessor are synchronous, "delay instructions" can be inserted at the beginning of iterations to fulfill data and control dependence requirements and reduce synchronization cost [36]. However, due to memory and interconnection network access conflicts, synchronization instructions are usually needed, even if all of the processing elements work at the same speed. Nevertheless, the number of synchronization instructions inserted into iterations can be reduced, as some synchronization instructions can be implied by some other synchronization instructions [90]. In addition, when synchronization cost gets higher, it is preferable by grouping several iterations into a larger execution unit such that the number of synchronization instructions can be further reduced [74]. In an extreme case where synchronization cost is very high, the synchronization costs can be totally avoided by partitioning all the iterations of a loop into independent execution units, where each execution unit need not synchronize with any other execution units [41] [100] [121].

1.5 Organization of the Thesis

The rest of this thesis is organized as follows. Chapter 2 formally defines regular loops. Chapter 3 characterizes a set of important properties of executing regular loops on multiprocessors. These properties facilitate the design and analysis of the schemes for executing regular loops on multiprocessors, shared-memory or distributed-memory. Chapter 4 addresses issues of executing regular loops on shared-memory multiprocessors. A greedy scheduling scheme is first introduced, which is then analyzed by applying the properties obtained in Chapter 2. In addition, some possible improvements for the greedy scheduling scheme are also proposed. Chapter 5 formulates the "mapping" problem for distributed-memory multiprocessors (i.e., the problem of mapping a regular loop onto a distributed-memory multiprocessor). A mapping scheme can be specified either by a *free mapping representation* or by a *linear mapping representation*. The former representation can always specify an execution scenario with the shorter parallel execution time, yet it needs exponential space with respect to that of the latter representation. Both representations are addressed. In addition, some NP-hard results are also shown. Chapter 6 proposes a method for executing regular loops on distributed-memory multiprocessors. This method is based on the linear mapping scheme, with some modifications so that the parallel execution time can be shortened and the number of processors used can be minimized. One nice property of this method is that it is independent of the structures of interconnection networks – the parallel execution of a regular loop is reduced to the execution of *cyclic shift operations*. Hence for different interconnection networks, only the cyclic shift algorithm needs to be re-implemented. As an example of the cyclic shift algorithms, Chapter 7 is dedicated to the cyclic shift algorithms for hypercube interconnection networks. Finally, Chapter 8 summarizes the contributions of this work.

Chapter 2

Regular Loops

The regular loops that will be considered in this thesis are formally defined in this chapter. Throughout this thesis, a regular loop has the form shown in Figure 2.1. The variables $I_1, I_2, \cdots, I_n$ are called *induction variables*. For each induction variable I_j, its value can range from L_j to U_j. Therefore, it is clear that there are $\prod_{j=1}^{n}(U_j - L_j + 1)$ iterations in the regular loop.

```
DO $I_1$ = $L_1$, $U_1$
   DO $I_2$ = $L_2$, $U_2$
         ⋮
      DO $I_n$ = $L_n$, $U_n$
         /* loop body */
         Statement $S_1$
         Statement $S_2$
            ⋮
         Statement $S_s$
      ENDDO
         ⋮
   ENDDO
ENDDO
```

Figure 2.1: The regular loop.

In the following discussion, an iteration will be represented by the values of its induction variables. That is, when we say iteration $[i_1, i_2, \ldots, i_n]$, it means the iteration with induction variable I_j being equal to i_j, for $1 \leq j \leq n$. Note that for each induction variable, we assume that its *increment value* is also 1. In addition, in our following discussion, we will assume $L_j = 0$ or $L_j = 1$. This does not harm the generality because, for an induction variable, if its increment value is not 1 and its initial value is not 0 or 1, we can always transform the regular loop such that the increment value is 1 and the initial value is 0 or 1. This transformation is called *loop normalization.* As an example, given the following regular loop, we can transform it so that the increment value and the the initial value are both 1.

```
DO $I$ = $L$, $U$, $C$
    ⋮
  $\cdots = A(I)$
    ⋮
ENDDO
```

If C is positive and $L \leq U$, or C is negative and $L \geq U$, then the loop can be normalized as below:

```
DO $I$ = 1, $\lfloor \frac{U-L}{C} \rfloor + 1$, 1
    ⋮
  $\cdots = A(L + (I-1) * C)$
    ⋮
ENDDO
```

In addition, a set of statements constitutes the *loop body* of the regular loop in Figure 2.1. Each statement has $\prod_{j=1}^{n}(U_j - L_j + 1)$ *instances*, with each instance corresponding to one iteration. Among all of the statement instances, the execution order is stipulated by *data dependencies.* Two statement instances are said to be data dependent if both of these instances want to access the same variable, with at least one instance trying to modify the value of the variable in question. Three types of data dependencies exist, namely, *flow dependence*, *anti-dependence*, and *output dependence* [98]. The distinction among these types is not significant in our discussion here, and we will simply use *data dependence* to represent any one of them. As a concrete example, consider the following program for matrix multiplication:

```
DO $I_1$ = 1, $N$
    DO $I_2$ = 1, $N$
        DO $I_3$ = 1, $N$
            $C(I_1, I_2)$ = $C(I_1, I_2) + A(I_1, I_3) * B(I_3, I_2)$
        ENDDO
    ENDDO
ENDDO
```

Figure 2.2: A regular loop for matrix multiplication.

The statement instance at iteration $[1, 1, 2]$ must be executed after the statement instance at iteration $[1, 1, 1]$, because both of the statement instances want to access the variable $C(1, 1)$; the statement instance at iteration $[1, 2, 3]$ must be executed after the statement instance at iteration $[1, 2, 2]$, because both of the statement instances want to access the variable $C(1, 2)$. Other restrictions for execution orders can be derived similarly. In addition, as an example, note that all statement instances at iteration $[i_1, i_2, 1]$, for $1 \leq i_1 \leq N$, $1 \leq i_2 \leq N$, can be executed in arbitrary order, because all these instances are data independent.

In the above example, all of the data dependencies can be represented by a single *dependence vector*, that is, $[0, 0, 1]$. Throughout this thesis, we will consider only those data dependencies which can be represented by dependence vectors. A lot of loop algorithms can be formulated by dependence vectors, including discrete Fourier transform, string matching, LU decomposition, and so on [84]. By imposing this restriction on our model, more efficiency can be squeezed out from the loops, with some degree of generality being traded off.

In summary, a regular loop is modeled by (S, D), where S is an *iteration space* and D is a set of *dependence vectors.* The iteration space S for the loop in Figure 2.1 is the Cartesian product $[L_1, U_1] \times [L_2, U_2] \times \cdots \times [L_n, U_n]$. Furthermore, in the set of dependence vectors $D = \{d_1, d_2, \ldots, d_m\}$ for the loop in Figure 2.1, each dependence vector $d_i = [d_{i1}, \ldots, d_{in}]$ is used to describe that the statement instance at iteration $[s_1, \ldots, s_n]$ must be executed after the statement instance at iteration $[s_1 - d_{i1}, \ldots, s_n - d_{in}]$. We call iteration $[s_1 - d_{i1}, \ldots, s_n - d_{in}]$ the *dependent predecessor* of iteration $[s_1, \ldots, s_n]$, and iteration $[s_1, \ldots, s_n]$ the *dependent successor* of iteration $[s_1 - d_{i1}, \ldots, s_n - d_{in}]$. In addition, two basic properties of dependence vectors can be observed immediately:

1. For each dependence vector, the leftmost nonzero component must be positive.

2. $|d_{ij}| < U_j - L_j + 1$, for $1 \leq i \leq m$, $1 \leq j \leq n$.

As an example for this model, consider the regular loop in Figure 2.2. The iteration space is $[1, N] \times [1, N] \times [1, N]$, and the only dependence vector is $[0, 0, 1]$, which means that the outer two loops can be executed in parallel without yielding different answers from that obtained by the sequential execution of the same algorithm.

2.1 Dependence Graphs

To study the execution of regular loops, it is also useful to represent a regular loop (S, D) by a *dependence graph* $G = (V, E)$, where each vertex in V corresponds to an iteration in the iteration space S, and an edge $< v_1, v_2 >$ is in E if the iteration corresponding to v_1 is a dependent predecessor of the iteration corresponding v_2. An example of a dependence graph (for the regular loop $([1, 10] \times [1, 10], \{[1, 2], [2, 1]\})$) can be found in Figure 3.1(a). In addition, for convenience, we will represent a vertex $v \in V$ by its corresponding iteration in S. Hence for $< v_1, v_2 > \in E$, we can also say that $v_2 - v_1 \in D$.

Note that the size of a dependence graph is exponential with respect to the size of representing its corresponding regular loop directly. Specifically, consider a regular loop (S, D), where $S = [1, U_1] \times \cdots \times [1, U_n]$, and $D = \{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\}$. Representing this regular loop in a "compact" form needs roughly $(m+1)\log(U_1 U_2 \cdots U_n)$ bits: $\lceil \log U_1 \rceil + \cdots + \lceil \log U_n \rceil$ bits[1] are needed for the iteration space S, and $m(\lceil \log U_1 \rceil + \cdots + \lceil \log U_n \rceil)$ bits are needed for the set of dependence vectors D. On the other hand, in the corresponding dependence graph, the regular loop is "unrolled" completely over the iteration space. In other words, in the corresponding dependence graph, there are $U_1 U_2 \cdots U_n$ vertices and roughly $mU_1 U_2 \cdots U_n$ edges. To represent a vertex in the dependence graph, it is clear that at least $\lceil \log(U_1 U_2 \cdots U_n) \rceil$ bits are needed. Hence the total number of bits required to represent the whole dependence graph is about $(2m + 1)U_1 U_2 \cdots U_n \log(U_1 U_2 \cdots U_n)$. By comparing this value with $(m + 1)\log(U_1 U_2 \cdots U_n)$, it is clear that the size of representing a dependence graph is at least exponential to the size of representing the regular loop directly. We will call the former representation the *unrolled form* and the latter representation the *compact form.*

Throughout this thesis, except when explicitly stated, we will assume the regular loop is represented in compact form.

[1]For convenience, the lower bounds of the induction variables are assumed to be 1 here, and hence need not be represented explicitly.

Chapter 3

Properties of Executing Regular Loops

In this chapter, some basic properties of executing a regular loop shown in Figure 2.1 are considered. Based on these properties, efficient scheduling schemes for regular loops on shared-memory multiprocessors can be designed. In Chapter 4, we will show how these basic properties help in designing and analyzing scheduling schemes.

Previously, properties for more general loops have been studied [16] [61]. In this chapter, the loops are confined to a special yet important case. By doing this, more properties can be exploited.

Throughout this chapter, the regular loop considered is of the form (S, D), where $S = [1, U_1] \times [1, U_2] \times \cdots \times [1, U_n]$, and $D = \{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\}$. In addition, all statements inside an iteration is assumed to be a basic *execution unit*. Note that on MIMD machines, the communication cost is high when compared to that of SIMD machines. Hence each execution unit should be large enough to balance the computation and communication costs. However, sometimes each iteration may contain only a small number of statements. In such a situation, several iterations need to be grouped together into a larger execution unit. Some grouping techniques have been studied in [74]. we will address the grouping problem at more detail in Section 4.3.1. For the time being, each iteration is assumed to be a basic execution unit, which can only be executed by a single processing element.

In our model, an iteration without any dependent predecessors is called an *initial* iteration. Given a regular loop, the number of initial iterations is fixed, that is, this number does not vary during the execution of the regular loop. Hence, the number of initial iterations is called a *static* property of executing a regular loop. In Section 3.1, we will show which iterations are initial iterations, and give an algorithm which can find all of the initial iterations efficiently.

In contrast to the static property, executing a regular loop also has some *dynamic* properties. Note that an iteration can be executed only after all its dependent predecessors have been completed. During the execution of a regular loop, an iteration will pass through four states, namely, *idle, pending, ready*, and *finished* states:

- An iteration is at idle state if none of its dependent predecessors have been completed.

- An iteration is at pending state if some of its dependent predecessors have been completed, but not all of them have.

- An iteration is at ready state if all of its dependent predecessors have been completed, but it itself has not.

- Otherwise, an iteration is at finished state.

For convenience, we will also call an iteration at idle, pending, ready, or finished state as an idle, pending, ready, or finished iteration respectively. Note that an initial iteration can only have ready and finished states. During the execution of a regular loop, the number of iterations at idle states will get less and less, while the number of iterations at finished states will become more and more. When the regular loop is completed, all iterations are at finished states. The *maximal* numbers of idle, pending, ready, and finished iterations at *any* instance during the execution are the *dynamic* properties of executing a regular loop. It is trivial to know that at the beginning of executing a regular loop, we have the maximal number of idle iterations, which is equal to the total number of iterations, i.e., $\prod_{j=1}^{n} U_j$, minus the number of initial iterations. Similarly, at the end of executing a regular loop, we have the maximal number of finished iterations, which is equal to the total number of iterations, i.e., $\prod_{j=1}^{n} U_j$. It is not trivial to determine, at any instance during the execution, the maximal number of ready and pending iterations, which will be addressed in Section 3.2 and 3.3 respectively.

The last property that will be studied for a regular loop is the *length of the longest paths* in the associated dependence graph G. A *longest path* of $G = (V, E)$ is a path $p = v_0 v_1 \ldots v_l$ such that $v_i \in V$ for $0 \leq i \leq l$, $< v_i, v_{i+1} > \in E$ for $0 \leq i \leq l-1$, and l is the maximum over all such paths. Note that the parallel execution time of a regular loop has intimate relation with the length of the longest paths. In Section 3.4, the method of finding the length of the longest paths of G is studied.

As a summary of these basic properties, let us give an example:

Example 3.1 Consider the following regular loop:

```
DO $I_1$ = 1, 10
  DO $I_2$ = 1, 10
    a($I_1$, $I_2$) = a($I_1 - 1$,$I_2 - 2$) + a($I_1 - 2$,$I_2 - 1$)
  ENDDO
ENDDO
```

Figure 3.1: Illustration of a regular loop (S, D), where $S = [1, 10] \times [1, 10]$ and $D = \{[1, 2], [2, 1]\}$.

Figure 3.1: *continued.*

The regular loop will be modeled by (S, D), where the iteration space S is $[1, 10] \times [1, 10]$, and the set of dependence vectors D is $\{[1, 2], [2, 1]\}$. Figure 3.1 (a) shows the corresponding dependence graph, where the horizontal axis corresponds to I_1 and the vertical axis corresponds to I_2. For clearness, the edges of the dependence graph are not drawn in Figure 3.1 (b) to (d). In Figure 3.1 (b), all of the iterations enclosed in the shadow region are initial iterations. In Figure 3.1 (c), an instance of executing the regular loop is shown. The filled circles are idle iterations, the triangles are pending iterations, the rectangles are ready iterations, and the unfilled circles are finished iterations. Finally, in Figure 3.1 (d), two of the longest paths of the dependence graph are shown. The length of the longest paths is 6 here. $\square$

3.1 Initial Iterations

In this section, initial iterations of a regular loop are identified, and an efficient algorithm which generates all of the initial iterations is given.

We first identify which iterations of a regular loop are initial iterations. From Example 3.1, it can be observed that all initial iterations are at the "borders" of the iteration space. For example, consider a regular loop (S, D), where the iteration space $S = [1, 10] \times [1, 10]$, and the set of dependence vectors $D = \{[2, 0], [1, -2]\}$. For the dependence vector $[2, 0]$, any iteration in the set $I_1 = I_{11} \cup I_{12} = \{[e_1, e_2] \mid 1 \leq e_1 \leq 2,\ 1 \leq e_2 \leq 10\} \cup \emptyset$ does not depend on any other iteration via the dependence vector $[2, 0]$. (Please see Figure 3.2 (a).) Note that all iterations in the set I_1 are at the "border" of the iteration space. In addition, for the dependence vector $[1, -2]$, any iteration in the set $I_2 = I_{21} \cup I_{22} = \{[e_1, e_2] \mid 1 \leq e_1 \leq 1,\ 1 \leq e_2 \leq 10\} \cup \{[e_1, e_2] \mid 1 \leq e_1 \leq 10,\ 9 \leq e_2 \leq 10\}$ does not depend on any other iteration via the dependence vector $[1, -2]$. (Please see Figure 3.2 (b).) Note that all iterations in the set I_2 are also at the "border" of the iteration space. As an initial iteration does not depend on any other iteration via *any* dependence vector, the set of initial iterations for the regular loop is $I = I_1 \cap I_2 = (I_{11} \cup I_{12}) \cap (I_{21} \cup I_{22}) = \{[e_1, e_2] \,|\, 1 \leq e_1 \leq 2,\ 1 \leq e_2 \leq 10\} \cap (\{[e_1, e_2] \,|\, 1 \leq e_1 \leq 1,\ 1 \leq e_2 \leq 10\} \cup \{[e_1, e_2] \,|\, 1 \leq e_1 \leq 10,\ 9 \leq e_2 \leq 10\})$. (Please see Figure 3.2 (c).) The following theorem formally states which iterations are initial iterations:

Theorem 3.1 *Consider a regular loop (S, D), where $S = [1, U_1] \times [1, U_2] \times \cdots \times [1, U_n]$, and $D = \{d_i = [d_{i1}, \ldots, d_{in}] \mid 1 \leq i \leq m\}$. Let*

$$I_{ij} = \begin{cases} \{[e_1, \ldots, e_n] \in S | 1 \leq e_j \leq d_{ij}\} & \text{if } d_{ij} > 0 \\ \emptyset & \text{if } d_{ij} = 0 \\ \{[e_1, \ldots, e_n] \in S | U_j + d_{ij} < e_j \leq U_j\} & \text{if } d_{ij} < 0 \end{cases}$$

Then an iteration k is in $I = \bigcap_{i=1}^{m} \bigcup_{j=1}^{n} I_{ij}$ if and only if k is an initial iteration for (S, D).

Proof. If $k = [k_1, \ldots, k_n]$ is in I, we show that it is impossible that k depends on any other iteration in the iteration space S. Suppose on the contrary that k depends on some other iteration, then there must exist an iteration $k' = [k'_1, \ldots, k'_n]$ such that $k = k' + d_s$ for some fixed s. Since $k \in \bigcap_{i=1}^{m} \bigcup_{j=1}^{n} I_{ij}$, k must be in I_{st} for some fixed t. Since $k = k' + d_s$, we have $k_t = k'_t + d_{st}$. However,

- if $d_{st} > 0$, then $k'_t = k_t - d_{st} \leq d_{st} - d_{st} = 0$;
- if $d_{st} = 0$, then $I_{st} = \emptyset$, k cannot be in I_{st};
- if $d_{st} < 0$, then $k'_t = k_t - d_{st} > U_t + d_{st} - d_{st} = U_t$.

These cases imply that k cannot depend on any other iteration in S, contradicting to our assumption. Hence we conclude that k does not depend on any other iteration in S and k is an initial iteration.

(a) Iterations which do not depend on other iterations via the dependence vector $[2, 0]$.

(b) Iterations which do not depend on other iterations via the dependence vector $[1, -2]$.

(c) Iterations which do not depend on other iterations via any dependence vector.

Figure 3.2: Illustration for the regular loop (S, D), where $S = [1, 10] \times [1, 10]$, and $D = \{[2, 0], [1, -2]\}$.

Conversely, if $k \notin I$, then $k \notin \bigcup_{j=1}^{n} I_{sj}$ for some fixed s. In other words, $k \notin I_{sj}$ for all $1 \leq j \leq n$. Let us consider the following cases:

- if $d_{sj} > 0$, then $d_{sj} < k_j \leq U_j$;
- if $d_{sj} = 0$, then $1 \leq k_j \leq U_j$;
- if $d_{sj} < 0$, then $1 \leq k_j \leq U_j + d_{sj}$.

Define $k' = [k_1 - d_{s1}, \ldots, k_n - d_{sn}]$. It is clear that k' must be in the iteration space S and $k = k' + d_s$. That is, k depends on k'. This completes our proof. □

The above theorem identifies all initial iterations in a regular loop. Now we will consider how these initial iterations can be generated in an algorithmic way.

A naive method to generate all of the initial iterations would be sweeping through the iteration space to check for each iteration if it has dependent predecessors. This is described by the following algorithm:

Algorithm 3.1 Consider a regular loop (S, D), where $S = [1, U_1] \times [1, U_2] \times \cdots \times [1, U_n]$, and $D = \{d_i \mid 1 \leq i \leq m\}$. The following program generates all of the initial iterations of (S, D) by sweeping through the iteration space:

```
for (each iteration $I \in S$) {
    $init$ = true;
    for (each dependence vector $d_i \in D$) {
        if ($I - d_i$ is in $S$) {
            $init$ = false;
            break;
        }
        if ($init$ is true) { generate iteration $I$; }
    }
}
```

$\square$

In spite of the simplicity of the algorithm, the time needed to execute the algorithm is $O(m\prod_{j=1}^{n}U_j)$. Much time can be saved if we can avoid scanning non-initial iterations. This motivates us to design an algorithm which *generates* all of the initial iterations directly, instead of *verifying* if an iteration is an initial iteration. This more efficient algorithm can be realized by applying Theorem 3.1.

Consider a regular loop as defined in Theorem 3.1. For any dependence vector d_i, we have:

$$\begin{aligned}
\cup_{j=1}^{n} I_{ij} &= (I_{i1}) \cup ((S - I_{i1}) \cap (\cup_{j=2}^{n} I_{ij})) \\
&= (I_{i1}) \cup ((S - I_{i1}) \cap I_{i2}) \cup ((S - I_{i1} - I_{i2}) \cap (\cup_{j=3}^{n} I_{ij})) \\
&= (I_{i1}) \cup ((S - I_{i1}) \cap I_{i2}) \cup ((S - I_{i1} - I_{i2}) \cap I_{i3}) \\
&\quad \cup((S - I_{i1} - I_{i2} - I_{i3}) \cap (\cup_{j=4}^{n} I_{ij})) \\
&= \cdots
\end{aligned}$$

These equations motivates an algorithm which generates all of the initial iterations in a "dimension by dimension" way.

For clarity, let us describe the algorithm by an example first. Consider a regular loop (S, D), where the iteration space $S = [1, 10] \times [1, 10] \times [1, 10]$ and the set of dependence vectors $D = \{d_1 = [0, 2, 3], d_2 = [1, -1, 2], d_3 = [3, 1, 1]\}$. Let d_i be denoted by $[d_{i1}, d_{i2}, d_{i3}]$ for $1 \leq i \leq 3$. The algorithm recursively divides each dimension into regions until it reaches the last dimension. Initially, dimension 1 is divided into three regions [1,1], [2,3], and [4,10] according to d_{11}, d_{21} and d_{31}. By doing this way, we have:

- For each iteration in the subspace $[1, 1] \times [1, 10] \times [1, 10]$, it may depend on other iterations only via d_1. Dependence vectors d_2 and d_3 need not be considered in "deeper" dimensions, as implied by Theorem 3.1.

dimension 1		dimension 2			dimension 3		
$\{d_{11}, d_{21}, d_{31}\}$	[1,1]	R_1	$\{d_{12}\}$	[1,2]	R_{11}	$\{\}$	[1,10]
				[3,10]	R_{12}	$\{d_{13}\}$	[1,3]
	[2,3]	R_2	$\{d_{12}, d_{22}\}$	[1,2]	R_{21}	$\{d_{23}\}$	[1,2]
				[3,9]	R_{22}	$\{d_{13}, d_{23}\}$	[1,2]
				[10,10]	R_{23}	$\{d_{13}\}$	[1,3]
	[4,10]	R_3	$\{d_{12}, d_{22}, d_{32}\}$	[1,1]	R_{31}	$\{d_{23}\}$	[1,2]
				[2,2]	R_{32}	$\{d_{23}, d_{33}\}$	[1,1]
				[3,9]	R_{33}	$\{d_{13}, d_{23}, d_{33}\}$	[1,1]
				[10,10]	R_{34}	$\{d_{13}, d_{33}\}$	[1,1]

Table 3.1: Generating initial iterations for the regular loop (S, D), where the iteration space $S = [1, 10] \times [1, 10] \times [1, 10]$ and the set of dependence vectors $D = \{d_1 = [0, 2, 3], d_2 = [1, -1, 2], d_3 = [3, 1, 1]\}$.

- For each iteration in the subspace $[2, 3] \times [1, 10] \times [1, 10]$, it may depend on other iterations only via d_1 or d_2. Dependence vector d_3 need not be considered in the next "deeper" dimension, as implied by Theorem 3.1.

- For each iteration in the subspace $[4, 10] \times [1, 10] \times [1, 10]$, it may depend on other iterations via d_1, d_2, or d_3.

These three regions are shown in Figure 3.3 by R_1, R_2, and R_3 respectively. With dimension 1 restricted to the region $[1, 1]$, dimension 2 will be divided, using d_{12} only, into [1,2] and [3,10]. By doing this way, we have:

- For each iteration in the subspace $[1, 1] \times [1, 2] \times [1, 10]$, it may not depend on any other iterations.

- For each iteration in the subspace $[1, 1] \times [3, 10] \times [1, 10]$, it may depend on other iterations only via d_1.

These two regions are shown in Figure 3.3 by R_{11} and R_{12} respectively. Finally, with dimension 1 and 2 restricted to the region [1,1] and [1,2] respectively, initial iterations in the region $[1,1] \times [1,2] \times [1,10]$ are generated. Table 3.1 summarizes the generation procedure for this example. At each dimension, the table lists the dependence vectors that are used to partition the dimension into regions, the resulting regions, and the region names shown in Figure 3.3.

The general procedure for generating all of the initial iterations for a regular loop is described in Algorithm 3.2.

Algorithm 3.2 Let the regular loop be (S, D), where $S = [1, U_1] \times [1, U_2] \times \cdots \times [1, U_n]$, and $D = \{d_i = [d_{i1}, d_{i2}, \ldots, d_{in}] \mid \text{for } 1 \leq i \leq m\}$. The algorithm generates the indices of all initial iteration by recursive calls to the procedure iter(k,E), where k is the depth of the loop under considered, and E is a set containing dependence vectors which must be checked. Let $x_1, \ldots, x_n, y_1, \ldots, y_n$ be global variables. At the main routine, iter(1, D) is called.

```
**procedure** iter($k$, $E$)
{  **if** ($k = n$) {
        $x_n = \mathbf{max}(\{1\} \cup \{U_n + d_{in} + 1 | d_i \in E, d_{in} \leq 0\})$;
        $y_n = \mathbf{min}(\{U_n\} \cup \{d_{in} | d_i \in E, d_{in} \geq 0\})$;
        generate iterations in the Cartesian space $[x_1, y_1] \times \cdots \times [x_n, y_n]$;
   } **else** {
        $T = \{[f(d_{ik}), d_i] \ | \ d_i \in E, d_{ik} \neq 0, |d_{ik}| < U_k\}$ where
            $f(d_{ik}) = \begin{cases} d_{ik} & \text{if } d_{ik} > 0 \\ U_k + d_{ik} & \text{if } d_{ik} < 0 \end{cases}$
        $E' = \{d_i \in E | -U_k < d_{ik} \leq 0 \}$;
        $finished$ = **false**;
        $y_k = 0$;
        **while** (**not** $finished$) {
            $x_k = y_k + 1$;
            **if** ( $T = \emptyset$ )
                $y_k = U_k$;
            **else**
                /* smallest($T$) is the smallest value of all $f(d_{ik})$'s in the elements of $T$. */
                $y_k$ = smallest($T$);
            iter($k$+1, $E'$);
            **if** ( $T = \emptyset$ )
                $finished$ = **true**;
            **else** {
                **for** (each $[f(d_{ik}), d_i] \in$ T that $f(d_{ik}) = y_k$) {
                    **if** ($d_{ik} > 0$) insert $d_i$ into $E'$; **else** /* $d_{ik} < 0$ */ delete $d_i$ from $E'$;
                    remove $[f(d_{ik}),\ d_i]$ from $T$;
                }
            }
        }  /* while */
   }
}
```

□

The above algorithm is faster than Algorithm 3.1 because those iterations which are not initial iterations are not visited. However, Algorithm 3.2 does waste time for sweeping through "empty regions" when the last dimension n is empty. In this case, Algorithm 3.2 can be revised so that the role of dimension n is replaced by a nonempty dimension k where x_k is always less than or equal to y_k. In addition, it depends on applications when to generate all of the initial iterations. Algorithm 3.2 can be used to generate all of the initial iterations at compiling time, or be updated to fit into a run-time self-scheduling scheme [44].

3.2 Ready Iterations

In this section, we consider the maximum number of ready iterations at any instance during the execution of a regular loop. Note that the number of ready iterations (or the iterations at ready states) varies during the execution of a regular loop. The maximum number of ready iterations is interesting because, if each iteration is executed by at most one processor, the number of processors used to execute a regular loop can be upper-bounded by the maximum number of ready iterations. Execution speedup cannot be improved by increasing the number of processors to more than the maximum number of ready iterations. The following theorem gives an upper bound for the maximum number of ready iterations at any instance during the execution of a regular loop:

Theorem 3.2 *Given a regular loop (S, D), where the iteration space $S = [1, U_1] \times [1, U_2] \times \cdots \times [1, U_n]$, and the set of dependence vectors $D = \{d_i = [d_{i1}, \ldots, d_{in}] \mid \text{for } 1 \leq i \leq m\}$. The maximum number of ready iterations at any instance during the execution of the regular loop is less than or equal to $\min\{\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |d_{ij}|) \mid 1 \leq i \leq m\}$.*

Figure 3.3: Initial iterations for the regular loop (S, D), where the iteration space $S = [1, 10] \times [1, 10] \times [1, 10]$ and the set of dependence vectors $D = \{d_1 = [0, 2, 3], d_2 = [1, -1, 2], d_3 = [3, 1, 1]\}$.

Proof. Recall that according to the definition of dependence vectors, we have $|d_{ij}| < U_j$, for $1 \leq i \leq m$ and $1 \leq j \leq n$. Therefore, the value of $U_j - |d_{ij}|$ is at least 1.

Let I_{ij} be the same as that in Theorem 3.1. Then for each dependence vector d_i, $\cup_{j=1}^{n} I_{ij}$ is the set of iterations which do not depend on any other iterations via d_i. In addition, note that when each iteration is completed, at most one iteration can be activated via dependence vector d_i. Therefore, the maximum number of ready iterations at any instances during the execution is less than or equal to $\min\{ \mid \cup_{j=1}^{n} I_{ij} \mid | 1 \leq i \leq m\}$, where $\mid \cup_{j=1}^{n} I_{ij} \mid$ denotes the size of the set $\cup_{j=1}^{n} I_{ij}$.

Without loss of generality, we assume that $d_{ij} \geq 0$ for $1 \leq i \leq m$, and $1 \leq j \leq n$ below in computing the value of $\mid \cup_{j=1}^{n} I_{ij} \mid$. According to the definition of I_{ij}, we have:

$$\begin{aligned}
\cup_{j=1}^{n} I_{ij} &= \cup_{j=1}^{n} \{[e_1, \ldots, e_n] \in S | 1 \leq e_j \leq d_{ij}\} \\
&= \{[e_1, \ldots, e_n] \in S | \vee_{j=1}^{n} (1 \leq e_j \leq d_{ij})\} \\
&= S - \{[e_1, \ldots, e_n] \in S | \wedge_{j=1}^{n} (d_{ij} < e_j \leq U_j)\}
\end{aligned}$$

Therefore, we have $\mid \cup_{j=1}^{n} I_{ij} \mid = \prod_{j=1}^{n} U_j - \prod_{j=1}^{n} (U_j - d_{ij})$. This completes our proof. $\square$

Note that the bound given by the above theorem for the maximum number of ready iterations is tight. That is, there exist regular loops whose maximum numbers of ready iterations are exactly the same as the values computed by the above theorem. As an example, consider the regular loop (S, D), where $S = [1, 17] \times [1, 17]$ and $D = \{[1, 3], [3, 2]\}$. According to Theorem 3.2, the maximum number of ready iterations at any instance of executing the regular loop is 37. Figure 3.4 shows an execution instance of the regular loop. All of the iterations in the shadow region can be at ready states simultaneously and the number of them is exactly 37.

Figure 3.4: An execution instance for the regular loop $([1,17]\times[1,17],\{[1,3],[3,2]\})$. All iterations in the shadow region can be at ready states simultaneously.

3.3 Pending Iterations

In this section, we address the number of pending iterations when executing a regular loop. The following theorem gives an upper bound for the maximal number of pending iterations at any instance during the execution of a regular loop.

Theorem 3.3 *Consider a regular loop (S,D), where the iteration space $S = [1,U_1] \times [1,U_2]\times\cdots\times[1,U_n]$ and the set of dependence vectors $D = \{d_i = [d_{i1},\ldots,d_{in}] \mid 1 \le i \le m\}$. Then the number of pending iterations at any instance during the execution of the regular loop cannot exceed $\sum_{i=1}^{m}(\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |d_{ij}|)\,)$.*

Proof. When an iteration becomes pending, it must be "activated" by one of its dependent predecessors. For a dependence vector d_i, the number of pending iterations that are activated via d_i cannot exceed $\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |d_{ij}|)$, as can be seen from Theorem 3.2. Therefore, the total number of pending iterations cannot exceed

$\sum_{i=1}^{m}(\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |d_{ij}|))$. $\square$

Some other improvement can be imposed for the above theorem. For example, consider the case when $\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |d_{ij}|) < \prod_{j=1}^{n}(U_j - |d_{ij}|)$ for some i. That is, the number of *initially ready* iterations (considering d_i only) exceeds half of the total number of iterations. Then no more than $\prod_{j=1}^{n}(U_j - |d_{ij}|)$ iterations can be "activated" by the dependence d_i. Hence in this case, we can simply replace $\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |d_{ij}|)$ with $\prod_{j=1}^{n}(U_j - |d_{ij}|)$ in Theorem 3.3.

Recall that an iteration is at pending state if *any* of its immediately dependent predecessors has been completed. Therefore, it is natural to expect that the maximal number of pending iterations is greater than the maximal number of ready iterations. Figure 3.5 shows an execution instance of a regular loop, where all iterations in the shadow region can be at pending states simultaneously. The number of pending iterations shown is greater than the maximal number of ready iterations at any instance during the execution of the regular loop (given by Theorem 3.2), and is slight smaller than the bound given in Theorem 3.3.

The bound obtained in Theorem 3.3 may be quite large sometimes, especially when the number of depending vectors are large. However, for most scientific computations, we expect that the number of dependence vectors is small and the components of each dependence vector are small also. In this case, Theorem 3.3 should give reasonable upper bound for the number of pending iterations at any instance of executing a regular loop.

3.4 The Longest Paths

In this section, we discuss the issue of finding the length of the longest paths in the dependence graph associated with a regular loop. Note that in a dependence graph, there may be several longest paths with the same length. However, we are satisfied by finding

Figure 3.5: A regular loop (S, D), where $S = [1, 10] \times [1, 10]$ and $D = \{[0, 1], [1, 0]\}$. All iterations in the shadow region can be at pending states simultaneously.

the *length* of the longest paths, instead of the longest paths themselves.

For any directed acyclic graph, the problem of finding the length of the longest paths can be solved in polynomial time [50]. A simple algorithm to solve this problem is by applying topological sorting and dynamic programming techniques. Since the dependence graph for a regular loop is directed acyclic (see Theorem 3.9), we can find the length of the longest paths in the dependence graph in time polynomial to the size of the dependence graph.

However, our interest is in regular loops represented in "compact" form instead of in "unrolled" form. That is, we are interested in a polynomial time algorithm for the following problem:

P1: Given a regular loop (S, D) represented in "compact" form, what is the length of the longest paths in the corresponding dependence graph?

Unfortunately, We will prove there is no polynomial time algorithm to solve the above problem unless P = NP. That is, we will prove that Problem $P1$ is NP-hard.

To prove that Problem $P1$ is NP-hard, we first define a *path* in terms of the regular loop representation directly as below:

Definition 3.1 Given a regular loop (S, D), where the iteration space $S = [1, U_1] \times \cdots \times [1, U_n]$, and the set of dependence vectors $D = \{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\}$, a *path* with length l is a sequence of iterations $v_0, v_1, \cdots, v_l$, such that

1. for each iteration $v_k = [v_{k1}, \cdots, v_{kn}]$, we have $1 \leq v_{kj} \leq U_j$, for $0 \leq k \leq l$, and $1 \leq j \leq n$, and
2. for $0 \leq k < l$, we have $v_{k+1} = v_k + d_i$ for some d_i, $1 \leq i \leq m$.

$\square$

Instead of proving Problem $P1$ to be NP-hard directly, we first prove the decidability problem of $P1$ is NP-hard. The decidability problem of $P1$ is as below:

P2: Given a regular loop (S, D) represented in "compact" form, and an integer constant B, is there a path in the corresponding dependence graph with length greater than or equal to B?

To prove that Problem $P2$ is NP-hard, we will reduce the Set Packing Problem to Problem $P2$. The Set Packing Problem is as below:

SP: Given a collection C of m finite sets $S_1, \cdots, S_m$, and a positive integer $K \leq |C|$, does C contain at least K mutually disjoint sets?

The complexity of the Set Packing Problem has been studied by Karp [70]:

Lemma 3.4 *The Set Packing Problem SP is NP-complete.*

Now we shall reduce the Set Packing Problem SP to Problem $P2$ in polynomial time so that the answer to SP is true if and only if the answer to $P2$ is true. Hence Problem SP can be solved in polynomial time if there exists a polynomial time algorithm to solve Problem $P2$. In other words, since Problem SP has been proven to be NP-complete, Problem $P2$ is NP-hard.

Theorem 3.5 *Problem $P2$ is NP-hard.*

Proof. Given an instance of Problem SP, an instance of Problem $P2$ is constructed as follows:

- Let n be the total number of distinct elements in finite sets $S_1, \cdots, S_m$. Number these n elements by $\alpha_1, \alpha_2, \cdots, \alpha_n$ respectively. The idea is to "bind" an element to a dimension of the regular loop that will be constructed.
- Construct an iteration space $S = [1, U_1] \times \cdots \times [1, U_n]$, where $U_1 = U_2 = \cdots = U_n = 2$.
- For each finite set S_i in C, a dependence vector $d_i = [d_{i1}, \cdots, d_{in}] \in D$ is defined, where
$$d_{ij} = \begin{cases} 1 & \text{if } \alpha_j \in S_i \\ 0 & \text{otherwise} \end{cases}$$
- Let $B = K$, where B and K are integer constants of the instances of Problem $P2$ and SP respectively.

It is not hard to see the above construction can be completed in time polynomial to the instance size of Problem SP.

Next we prove that the answer to SP is true if and only if the answer to $P2$ is true. Suppose the answer to Problem SP is true, that is, suppose C contains $r \geq K$ mutually disjoint sets. Without loss of generality, let these r sets be $S_1, \cdots, S_r$. Since these r sets are mutually disjoint, an element α_k is contained in at most one of the finite set $S_1, \cdots, S_r$. In terms of the constructed regular loop, each dimension is thus bound by an element at most once. That is, the following sequence is a path satisfying Definition 3.1:

$$\begin{array}{cl} & [1, 1, \cdots, 1] \\ \rightarrow & [1, 1, \cdots, 1] + d_1 \\ \rightarrow & [1, 1, \cdots, 1] + d_1 + d_2 \\ & \vdots \\ \rightarrow & [1, 1, \cdots, 1] + d_1 + d_2 + \cdots + d_r \end{array}$$

Hence there is a path with length $r \geq B$ in the constructed regular loop, that is, the answer to Problem $P2$ is true also.

Conversely, suppose the answer to Problem $P2$ is true, that is, suppose the constructed regular loop contains a path with length $r \geq B$. Since any component of the dependence vectors is either 0 or 1, and the range of any dimension of the iteration space is $[1, 2]$, a dependence vector cannot be "used" more than once in the path. Without loss of generality, let the source iteration of the path be v_0 and the sink iteration of the path be $v_0 + d_1 + d_2 + \cdots + d_r$. By the same argument, for any two dependence vectors d_p and d_q "used" in the path, $1 \leq p, q \leq r$, we cannot have $d_{pj} = d_{qj} = 1$, for any $1 \leq j \leq n$. In other words, there are $r \geq K$ mutually disjoint finite sets $S_1, \cdots, S_r$ contained in C, that is, the answer to Problem SP is true also.

we have reduced Problem SP to Problem $P2$ in polynomial time so that the answer to SP is true if and only if the answer to $P2$ is true. Since Problem SP is NP-complete, we conclude that Problem $P2$ is NP-hard. $\square$

Note that in the proof of the above theorem, we reduced the Set Packing Problem to a special case of Problem $P2$. This special case of Problem $P2$ is certainly NP-hard also.

Corollary 3.6 *Problem $P2$ is NP-hard even if every component of all dependence vectors is restricted to either* 0 *or* 1.

From this corollary, we know that it is very unlikely to find a polynomial time algorithm to solve Problem $P2$, even for the seemingly very simple case where every component of all dependence vectors is restricted to either 0 or 1.

To make the proof of Theorem 3.5 clear, let us see an example of the reduction:

Example 3.2 Let the collection of finite sets C be $\{S_1 = \{a, b\}, S_2 = \{b, c, d\}, S_3 = \{a, c, d\}, S_4 = \{b\}\}$, then the total number of elements in all of the finite sets in C is 4, that is, $\alpha_1 = a, \alpha_2 = b, \alpha_3 = c, \alpha_4 = d$. Therefore, the constructed regular loop is (S, D), where the iteration space S is $[1, 2] \times [1, 2] \times [1, 2] \times [1, 2]$, and the set of dependence vectors D is $\{d_1 = [1, 1, 0, 0], d_2 = [0, 1, 1, 1], d_3 = [1, 0, 1, 1], d_4 = [0, 1, 0, 0]\}$. In addition, C contains 2 mutually disjoint finite sets S_3 and S_4; correspondingly, the constructed regular loop contains a path with length 2: $[1, 1, 1, 1] \rightarrow [1, 1, 1, 1] + d_3 \rightarrow [1, 1, 1, 1] + d_3 + d_4 \equiv [1, 1, 1, 1] \rightarrow [2, 1, 2, 2] \rightarrow [2, 2, 2, 2]$. It is also easy to check that C does not contain more than 2 mutually disjoint finite sets, neither does the constructed regular loop contain a path with length longer than 2. $\Box$

Our goal has been to prove that Problem $P1$ is NP-hard. This is achieved by Turing reduction from Problem $P2$ to Problem $P1$:

Theorem 3.7 *Problem $P1$ is NP-hard.*

Proof. Let the algorithm solving Problem $P1$ be $A(S, D, l)$, where S and D are input parameters representing the regular loop in question, and l is an output parameter representing the length of the longest paths of the regular loop. Then Problem $P2$ can be solved in terms of algorithm $A(S, D, l)$ as follows:

```
Call algorithm $A(S, D, l)$;
if ( $l \geq B$ )
        answer true;
else
        answer false;
```

Therefore, if Algorithm A can solve Problem $P1$ in polynomial time, Problem $P2$ can be solved in polynomial time also. On the other hand, since Problem $P2$ has been proved to be NP-hard, Problem $P1$ is NP-hard also. $\square$

Since finding the exact length of the longest paths of a regular loop is NP-hard, solving this problem, in practice, will take time more than polynomial to the problem size. However, we would be satisfied sometimes if we can find the approximate length of the longest paths. For example, to estimate the parallel execution time of a regular loop as described in the next chapter, finding the approximate length is sufficient.

In the remaining of this section, we will describe an algorithm which finds the *upper bound* of the length of the longest paths. The candidate algorithm should not unroll the whole iteration space, since it will take too much time as we have analyzed. With these ideas in mind, we reduce the problem of "finding the upper bound of the length of the longest paths" to a set of linear programming problems.

Algorithm 3.3 Given a regular loop (S, D), where the iteration space $S = [1, U_1] \times \cdots \times [1, U_n]$, and the set of dependence vectors $D = \{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\}$, this algorithm finds the upper bound of the length of the longest paths of this regular loop.

Step 1. Let us call the first iteration of a longest path the *source iteration.* The first work in finding the upper bound of the length of the longest paths is to identify the possible source iterations. Intuitively, these source iterations should be at the "corners" of iteration space S. Moreover, some "corners" can be excluded since a longest path cannot start from those iterations. For example, consider a regular loop (S, D), where $S = [1,10] \times [1,10]$ and $D = \{[1,-1],[1,2]\}$. There are four iterations at the corners, that is, $[1,1],[1,10],[10,1]$, and $[10,10]$. Since the first dimensions of both dependence vectors are positive (i.e., 1), it is clear that a longest path cannot start from $[10,1]$ and $[10,10]$. Generally, assuming that[1] $\sum_{i=1}^{m} d_{ij}^2 > 0$, for $1 \le j \le n$, then the set of all possible source iterations, C, is

$$\{s_k = [s_{k1}, \cdots, s_{kn}] \mid s_{kj} \in f(j),\ 1 \le j \le n\}$$

where

$$f(j) = \begin{cases} \{1\} & \text{if } \forall\ \ 1 \le i \le m,\ d_{ij} \ge 0 \\ \{U_j\} & \text{if } \forall\ \ 1 \le i \le m,\ d_{ij} \le 0 \\ \{1, U_j\} & \text{otherwise} \end{cases}$$

Step 2. Let $C = \{s_1, s_2, \cdots, s_a\}$ be the set of all possible source iterations obtained in Step 1. Then the length of the longest paths in the dependence graph cannot exceed $\max\{v_1, \cdots, v_a\}$, where v_k is obtained by solving the following linear programming problem:

$$\begin{array}{lll} \textbf{LP:} & \textbf{max} & v_k \equiv x_1 + x_2 + \cdots + x_m \\ & \textbf{subject to} & \\ & \quad 1 \le s_{kj} + \sum_{i=1}^{m} x_i d_{ij} \le U_j & \text{for } 1 \le j \le n \\ & \quad x_i \ge 0 & \text{for } 1 \le i \le m \end{array}$$

□

Now we prove the above algorithm computes the upper bound of the length of the longest paths:

[1]If for some fixed dimension j, all d_{ij}'s are zeros, then the iteration space can be partitioned into U_j independent groups with each group corresponding to a distinct value in the range from 1 to U_j. Hence under such circumstance, dimension j can be regarded as nonexistent in solving the problem.

Theorem 3.8 *Given a regular loop (S, D), the length of the longest paths in the regular loop cannot exceed* $\max\{v_1, \cdots, v_a\}$, *as computed in Algorithm 3.3.*

Proof. Let L_1 denote the exact length of one of the longest paths, $u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_{L_1}$, where $u_k = [u_{k1}, \cdots, u_{kn}]$ for $0 \leq k \leq L_1$. In addition, suppose in the longest path, dependence vector d_i is used by x_i times, for $1 \leq i \leq m$. According to the definition of the longest path, all iterations in a longest path must be within the iteration space S. Specifically, the "sink" iteration of the longest path must be within the iteration space S also. Therefore, L_1 must be less than or equal to L_2 in the following *Integer Programming Problem*:

$$\begin{array}{lll} \textbf{LP2:} & \textbf{max} & L_2 \equiv x_1 + x_2 + \cdots + x_m \\ & \textbf{subject to} & \\ & \quad 1 \leq u_{0j} + \sum_{i=1}^{m} x_i d_{ij} \leq U_j & \text{for } 1 \leq j \leq n \\ & \quad x_i \geq 0 & \text{for } 1 \leq i \leq m \end{array}$$

Now we show that considering only the source iterations in C is sufficient. That is, given a path with both source node u_0 and sink node u_{L_2} in S, it is always possible to "shift" the path so that the new source node is at a corner of S and the new sink node is still in$S$. Mathematically, we show that there exists an source iteration $s_k = [s_{k1}, \cdots, s_{kn}] \in C$ such that L_2 is less than or equal to L_3 in the following *Integer Programming Problem*:

$$\begin{array}{lll} \textbf{LP3:} & \textbf{max} & L_3 \equiv x_1 + x_2 + \cdots + x_m \\ & \textbf{subject to} & \\ & \quad 1 \leq s_{kj} + \sum_{i=1}^{m} x_i d_{ij} \leq U_j & \text{for } 1 \leq j \leq n \\ & \quad x_i \geq 0 & \text{for } 1 \leq i \leq m \end{array}$$

The following cases explain the reason for $L_2 \leq L_3$ (note that $u_0 = [u_{01}, \cdots, u_{0n}]$ is within the iteration space S):

- If $1 < u_{0j} < u_{0j} + \sum_{i=1}^{m} x_i d_{ij} \leq U_j$, then there must exist some d_i such that $d_{ij} > 0$. Consequently, there must exist some $s_k \in C$ such that $s_{kj} = 1$ (i.e., the new source node is at a corner of the iteration space S). In addition, it is obvious that $1 < 1 + \sum_{i=1}^{m} x_i d_{ij} < U_j$ (i.e., the new sink node is still inside the iteration space S.).

- If $1 \leq u_{0j} + \sum_{i=1}^{m} x_i d_{ij} < u_{0j} < U_j$, then there must exist some d_i such that $d_{ij} < 0$. Consequently, there must exist some $s_k \in C$ such that $s_{kj} = U_j$. In addition, it is obvious that $1 < U_j + \sum_{i=1}^{m} x_i d_{ij} < U_j$.

- If $1 < u_{0j} = u_{0j} + \sum_{i=1}^{m} x_i d_{ij} < U_j$, then we have $1 = 1 + \sum_{i=1}^{m} x_i d_{ij} < U_j$ and $1 < U_j + \sum_{i=1}^{m} x_i d_{ij} = U_j$.

All of these cases imply that a feasible solution $(x_1, x_2, \cdots, x_m)$ to Problem $LP2$ is also a feasible solution to Problem $LP3$, hence we have $L2 \leq L3$.

Finally, since all feasible solutions to an integer programming problem are also feasible solutions to the corresponding linear programming problem, it is clear that $L3 \leq \max\{v_1, \cdots, v_a\}$, the value computed by Algorithm 3.3. □

We have reduced the problem of finding the upper bound of the longest paths' length to a set of linear programming problems. Note that there is at least one feasible solution to Problem LP. Since all source iterations in C are also in S, a trivial solution to Problem LP is when $x_1 = x_2 = \cdots = x_m = 0$, which corresponds the case when the length of the longest paths is zero. In addition, the maximum value of the object function in Problem LP (i.e., v_k) is always bounded. The reason for it is because the dependence graph is directed acyclic, as proven by the following theorem:

Theorem 3.9 *The dependence graph for a regular loop* (S, D), *where* $S = [1, U_1] \times \cdots \times [1, U_n]$ *and* $D = \{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\}$, *is directed acyclic.*

Proof. According to the definition of dependence graph, it is clear that the graph is directed.

We prove the dependence graph is acyclic. Suppose that there is a cycle in the dependence graph, then there must exist integers $x_1, \cdots, x_m$ such that $x_1 d_1 + x_2 d_2 + \cdots + x_m d_m = 0$. However, note that according to the definition of dependence vectors, the leftmost nonzero entry of a dependence vector must be positive. Let D_j be the set containing all of those dependence vectors in D with their leftmost nonzero entries at dimension j. Then for all $d_i \in D_1$, the corresponding x_i's must be zeros, since there is no negative component at dimension 1 for all dependence vectors. Furthermore, for all $d_i \in D_2$, the corresponding x_i's must also be zeros, since there is no negative component at dimension 2 for all dependence vectors in $D_2 \cup D_3 \cup \cdots \cup D_n$. Inductively, we have $x_1 = x_2 = \cdots = x_m = 0$. Hence the dependence graph is acyclic. $\square$

The efficiency of Algorithm 3.3 relies heavily on the complexity of solving the linear programming problem. The well known simplex method is very efficient in solving the problem in practice. However, under worst case, the simplex method may take time exponential to the problem size. Other algorithms like ellipsoid method or projection method guarantee solving this problem in polynomial time [94]. However, at the worst case, the number of linear programming problems we need to solve is exponential to the number of dimensions of the iteration space, since the number of "corners" of the iteration space is exponential to the number of dimensions of the iteration space. Fortunately, the number of dimensions of most loops is less than or equal to 3 [123], hence usually no more than 4 linear programming problems need to be solved (Note that for every source iteration, the component at the first dimension is always 1.).

Finally, it is important to consider the difference between the exact length and the upper bound obtained by Algorithm 3.3. For most common cases, where U_j's are large and $|d_{ij}|$'s are small, the difference is expected to be minor. However, Algorithm 3.3 is not suitable for the cases when U_j's and $|d_{ij}|$'s are close. For example, for the regular loop (S, D), where $S = [1, U_1] \times [1, U_2] = [1, 100] \times [1, 6]$ and $D = \{[d_{11}, d_{12}], [d_{21}, d_{22}]\} = \{[1, -5], [1, 4]\}$, the difference between the exact longest paths' length and the upper bound obtained by Algorithm 3.3 is large. The difference mainly results from that $|d_{12}|$ and $|d_{22}|$ are very close to U_2. On the other hand, we show that when U_j's and $|d_{ij}|$'s are not close, the exact length and its upper bound can be quite near for certain cases:

Theorem 3.10 *Consider a two dimensional regular loop (S, D), where $S = [1, U_1] \times [1, U_2]$, $D = \{d_i = [d_{i1}, d_{i2}] \mid 1 \le i \le m\}$, and $|d_{ij}| < \frac{1}{2}U_j$ for $1 \le i \le m$ and $1 \le j \le 2$. If an algorithm for integer programming problem is used to solve Problem LP in Algorithm 3.3, then the upper bound obtained by Algorithm 3.3 is exactly the same as the length of the longest paths in the regular loop.*

Proof. Assuming the upper bound obtained by Algorithm 3.3 is $\sum_{i=1}^{m} x_i$, with dependence vector d_i being used by x_i times, we prove that these $\sum_{i=1}^{m} x_i$ dependence vectors can always be "ordered" to yield a path with all its nodes being inside iteration space S.

Since only positive components are at the first dimensions for all dependence vectors, it is "safe" to set the first dimension of the source node (iteration) to be 1. No matter how these $\sum_{i=1}^{m} x_i$ dependence vectors are ordered, the first dimensions of all nodes in the path are between 1 and U_1, since $d_{i1} \ge 0$ for all $1 \le i \le m$. Therefore, we only need to consider the second dimension to make sure that the whole path is within the iteration space.

Without loss of generality, assume that $\sum_{i=1}^{m} x_i d_{i2} \ge 0$. Set the second dimension of the source node (iteration) to 1. Order these $\sum_{i=1}^{m} x_i$ dependence vectors according to the following algorithm:

```
1.    Let $P$ be the collection of these $\sum_{i=1}^{m} x_i$ dependence vectors;
2.    Let $c = [c_1, c_2]$ be the *current node* of the path;
      Initially, $c$ is the source node we have set, i.e., $c = [1, 1]$;
3.    **while** $(P \neq \emptyset)$ {
4.        retrieve a dependence vector $d_i$ from $P$ so that
              $c + d_i$ is within the iteration space $S$;
5.        $c \leftarrow c + d_i$;
6.        $P \leftarrow P - \{d_i\}$;
      }
```

Note that at line 4, we can always find a dependence vector in P which satisfies the condition mentioned. Consider the second dimension of the iteration space only. Without loss of generality, suppose before executing line 4, c_2 is at the "lower half" of the second dimension of the iteration space, i.e., $c_2 \leq \frac{1}{2}(U_2 + 1)$. If some d_i is chosen such that $c + d_i$ falls outside the iteration space, then we must have $c_2 + d_{i2} < 1$ and $d_{i2} < 0$, since $|d_{i2}| < \frac{1}{2}U_2$. However, since the sink node of the path must be within the iteration space S as required by Algorithm 3.3, we can always find another dependence vector $d_k \in P$ such that $d_{k2} > 0$. As $1 \leq c_2 \leq \frac{1}{2}(U_2 + 1)$ and $0 < d_{k2} < \frac{1}{2}U_2$, we must have $c + d_k$ in the iteration space S.

In summary, given the upper bound obtained by Algorithm 3.3, we can always generate a path with that length and of which all nodes are within the iteration space. Hence the upper bound is tight in this case. $\square$

Note that the difference between the upper bound obtained by Algorithm 3.3 and the real length of the longest paths is due to that at some dimension j, there exist both positive and negative d_{ij}'s. If for every dimension j, either all d_{ij}'s are non-negative or all d_{ij}'s are non-positive, then the upper bound obtained by Algorithm 3.3 are tight, as described by the following theorem:

Theorem 3.11 *Consider a regular loop (S, D), where $S = [1, U_1] \times \cdots \times [1, U_n]$, $D = \{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\}$, and for every dimension j, either all d_{ij}'s are non-negative or all d_{ij}'s are non-positive. If an algorithm for integer programming problem is used to solve Problem LP in Algorithm 3.3, then the upper bound obtained by Algorithm 3.3 is exactly the same as the length of the longest paths in the regular loop.*

Proof. For each dimension j, $1 \leq j \leq n$, the proof is similar to the proof for the first dimension of Theorem 3.10. □

Besides showing the quality of Algorithm 3.3 by Theorem 3.10 and 3.11, we evaluate its quality by using the 25 regular loops in [48, page 329]. Given a regular loop, let L_1 be the length of the longest paths obtained by applying Algorithm 3.3, and L_o be the real length of the longest paths. The values of L_1 and L_o for these 25 loops are shown below:

Loop	1	2	3	4	5	6	7	8	9	10	11	12	13
L_1	3.00	7.20	3.00	12.00	4.50	9.00	54.00	11.25	4.50	18.00	27.00	30.38	27.00
L_o	3	7	3	12	4	9	54	11	4	18	27	29	27
$\frac{L_1}{L_o}$	1.00	1.03	1.00	1.00	1.12	1.00	1.00	1.02	1.12	1.00	1.00	1.05	1.00

Loop	14	15	16	17	18	19	20	21	22	23	24	25
L_1	7.20	12.00	13.50	6.00	31.50	9.00	31.50	5.00	8.00	6.67	12.00	13.20
L_o	7	11	13	6	31	9	31	5	8	6	12	13
$\frac{L_1}{L_o}$	1.03	1.09	1.04	1.00	1.02	1.00	1.02	1.00	1.00	1.11	1.00	1.02

The average ratio of $\frac{L_1}{L_o}$ for these 25 regular loops is only 1.03. Therefore, we expect that the quality of Algorithm 3.3 will be very high in practice.

Chapter 4

Executing Regular loops on Shared–Memory Multiprocessors

In this chapter, we describe a simple scheme for scheduling regular loops on shared-memory multiprocessor computers. The scheme is suitable for asynchronous computers, where the asynchronism may result from different speeds among processing elements, communication links, communication switching boxes, or shared-memory subsystems. The scheme is also suitable for executing certain regular loops on synchronous computers, where different iterations of the loop may require different computation times, or where arranging non-blocking accesses to the shared-memory is difficult.

The scheduling scheme is described in Section 4.1. An analysis of the scheduling scheme, using the properties obtained in last chapter, is presented in Section 4.2. Finally, some improvements over the simple scheduling scheme are introduced in Section 4.3.

4.1 The Scheduling Scheme

The scheduling scheme described here is quite straightforward: each iteration is an execution unit (i.e., each iteration is the unit to be scheduled and will not be broken into smaller pieces) and is scheduled to be executed by a free processor *greedily*.

The scheduling scheme is designed for executing regular loops on shared-memory multiprocessor systems where either the systems are asynchronous or memory-access conflicts in the interconnection networks are hard to be predicted and avoided. Note that some "optimal" schemes such as [3] [37] [143] cannot be used on asynchronous multiprocessor systems. Though the greedy scheduling scheme seems quite naive and inefficient, it is essential in the sense that more advanced schemes are basically evolved from it. Therefore, understanding various properties of the basic scheduling scheme is very important to designing more advanced schemes.

Now let us describe the scheduling scheme. Recall that during the execution of a regular loop, an iteration will pass through four states, namely, idle, pending, ready, and finished states. In addition, certain iterations of a regular loop are called initial iterations, meaning that those iterations have no dependent predecessors. (Please see Chapter 3.) Our greedy scheduling scheme will schedule all initial iterations to be executed first. After all initial iterations have been completed, there must be iterations at ready states, that is, ready to be executed. This property can be observed from the fact that the corresponding dependence graph is directed acyclic (Theorem 3.9). Our scheduling scheme then choose iterations at ready states to execute. Note that an iteration will enter the ready state when it is ready to be executed and leave the ready state when it has been completed. The regular loop is completed totally when no more iterations can be at ready state. In addition, when an iteration has more than one dependent predecessors, it has to stay at pending states for sometimes and enter the ready state when all of its dependent predecessors have been completed. This implies that our scheduling scheme need to maintain information about when an iteration can enter from the pending state to the ready state.

Based on the previous discussion, our scheduling scheme needs the following three "pools" to store iterations:

- INIT: a data structure used to store initial iterations.
- READY: a date structure used to store ready iterations.
- PENDING: a data structure used to store pending iterations.

Note that an iteration can be stored by its induction variables instead of the whole loop body, since the loop bodies are the same for all iterations. The scheduling scheme is as follows. A free processor first tries to fetch an iteration from INIT to execute. If INIT is empty, then an iteration is fetched from READY. Whenever a processor finishes executing a ready iteration *ci*, it "installs" each dependent successor *si* as follows:

```
1.   Try to find the entry for si in PENDING;
2.   if (si is not in PENDING)  {
3.       if (si has no other dependent predecessors)
4.           install si in READY;
5.       else
6.           install si in PENDING;
     } else  {
7.       At the si entry in PENDING, mark ci finished;
8.       if (si has no other unfinished dependent predecessors)
9.           move si from PENDING to READY;
     }
```

4.2 The Analysis

Although the concept of the scheme is simple, several nontrivial issues have to be considered:

1. What is the memory space needed by this scheme?
2. What is the maximum number of processing elements needed by this scheme?
3. How does one determine if the parallel execution is superior to the sequential execution?

Such issues of executing regular loops on shared-memory multiprocessors is important to further refining the basic scheme to more efficient ones. However, no discussion of them can be found in existent literature. This section will address these issues in detail. In our following discussion, the regular loop being considered is (S, D), where $S = [1, U_1] \times \cdots \times [1, U_n]$, and $D = \{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\}$.

We address the first issue first. That is, what is the memory space needed by this scheduling scheme? The space required can be divided into two classes, that is, the space needed by the regular loop, and the space needed by the scheduler. The space needed by the regular loop is used to store variables of the loop. This space requirement depends on the loop body of the regular loop and we will not discuss it here.

The space needed by the scheduler includes the space needed by the data structures INIT, PENDING, and READY. From the discussion in Chapter 3, the maximum spaces required by these three data structures are listed below:

INIT	$c_1 \mid \bigcap_{i=1}^{m} \bigcup_{j=1}^{n} I_{ij} \mid$, where I_{ij} is defined in Theorem 3.1.
PENDING	$c_2 \sum_{i=1}^{m}(\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - \lvert d_{ij}\rvert))$.
READY	$c_3 \min\{\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - \lvert d_{ij}\rvert) \mid 1 \leq i \leq m\}$

In the above table, c_1, c_2, and c_3 represent the spaces required by an iteration entry in data structures INIT, PENDING, and READY respectively. The formulae for INIT, PENDING, and READY are derived from Theorem 3.1, 3.3, and 3.2 respectively.

Note that the value of $| \bigcap_{i=1}^{m} \bigcup_{j=1}^{n} I_{ij} |$ (in the formula for INIT) is always smaller than or equal to the value of $\min\{\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |d_{ij}|) \mid 1 \le i \le m\}$ (in the formula for READY). In fact, the space requirement for INIT can be eliminated totally by generating all initial iterations "on-the-fly", that is, an initial iteration is generated only when a processing element is free to execute a new iteration. The generation algorithm can be obtained easily by revising Algorithm 3.2.

Now we address the second issue. That is, what is the maximum number of processing elements needed by this scheduling scheme? The solution to this can be obtained easily from Theorem 3.2, that is, the maximum number of processing elements needed is $\min\{\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |d_{ij}|) \mid 1 \le i \le m\}$, because only iterations at ready states can be executed. In other words, we cannot get any speedup even if we use more than this number of processing elements to execute the regular loop.

Finally, we address the third issue. That is, how does one determine if the parallel execution is superior to the sequential execution? Because of the synchronization and communication cost, it is well known that the parallel execution of a regular loop is not necessarily faster than the sequential execution of the same loop. Therefore, the compiler should estimate both of the parallel and the sequential times to make the right choice.

Let t denote the average execution time of an iteration, s denote the synchronization time required by the algorithm on page 46, and l denote the number of iterations on any of the longest paths in the dependence graph. Then the sequential execution time is $t \prod_{j=1}^{n} U_j$ and the parallel execution time is $(t + ms)l$. Therefore, the parallel execution is preferred to the sequential execution when $(t + ms)l < t \prod_{j=1}^{n} U_j$, i.e., when

$$\frac{ms}{t} < \frac{\prod_{j=1}^{n} U_j}{l} - 1 \tag{4.1}$$

The value of t can be estimated by examining the object code generated for the loop body; the value of s can be estimated by examining the object code generated for the algorithm on page 46; and the value of l can be estimated by applying Algorithm 3.3. Note that the value of l is hence the upper bound of the real length of the longest path, say l'. Therefore, Equation 4.1 is a *conservative* estimation for the decision of parallelization. That is, when Equation 4.1 is satisfied, it is safe to execute the regular loop in parallel by using our scheduling scheme. Mathematically, we have:

$$\begin{aligned} & l' \leq l \\ \Rightarrow \quad & (t + ms)l' \leq (t + ms)l < t \prod_{j=1}^{n} U_j \\ \Rightarrow \quad & \frac{ms}{t} < \frac{\prod_{j=1}^{n} U_j}{l} - 1 \leq \frac{\prod_{j=1}^{n} U_j}{l'} - 1 \end{aligned}$$

It is also interesting to observe, from Equation 4.1, that under certain situations, parallelizing a regular loop is preferred:

1. when the execution time of an iteration (t) is large,
2. when the synchronization cost (ms) is small,
3. when the total number of iterations ($\prod_{j=1}^{n} U_j$) is large, and
4. when the longest path is short (Recall Amdahl's Law).

These observations also provide the guideline for further evolving the scheduling scheme. For example, Item 1 above indicates the *granularity* [11] [127] should be increased to benefit the parallelization of the regular loop. To increase the granularity, several iterations should be grouped together into a larger execution unit. In addition, Item 2 above indicates the synchronization cost should be reduced as much as possible. Other indirect observations can also guide the improvement of our scheduling scheme. For example, since the longest path dominates the parallel execution time, the iterations on the longest paths should be executed as soon as possible. This idea of scheduling "critical" iterations earlier has also been exploited under other settings [99]. In the next section, we will present several improvement techniques for our scheduling scheme based on these observations.

4.3 Improvement of the Scheduling Scheme

In this section, we will address some possible improvements of the greedy scheduling scheme. Our goal is not in proposing an extremely efficient scheduling scheme. Instead, our intention is to show how those basic properties obtained in Chapter 3 can help optimizing a scheduling scheme for a regular loop.

Three examples of improvement will be shown in this section:

1. Increasing the granularity.
2. Scheduling "critical" iterations earlier.
3. Reducing synchronization cost.

4.3.1 Increasing the Granularity

We first describe the technique of increasing the granularity. The motivation behind increasing the granularity is when the grain size is too small, most of the execution time will be spent on synchronization and communication. Hence by increasing the grain size, the overall execution time can be reduced. In terms of the parallel execution time, we

(a) Grouping along the direction $[0,1]$ with size 2.

(b) Grouping along the direction $[0,1]$ with size 2, and along the direction $[1,0]$ with size 2.

Figure 4.1: Two grouping methods for the regular loop $([1,8]\times[1,8],\{[0,1],[1,1],[1,0]\})$.

want to group several iterations into a larger execution unit so as to minimize the value of $(t+ms)l$ as described in last section. By grouping several iterations together, the average execution time of an execution unit, t, is increased, yet the number of execution units on the longest paths, l, can be reduced, hence it is possible to reduce the parallel execution time $(t+ms)l$.

However, the problem of grouping is not trivial [74]. It involves how many iterations should be grouped together, which iterations should be grouped together, and so on. We will not go into the detail here. Instead, we will give several examples of grouping. Our intention here is to point out a possible way of increasing the performance of the greedy scheduling scheme. Much work still needs to be done.

Consider the regular loop $([1,8]\times[1,8],\{[0,1],[1,1],[1,0]\})$. Figure 4.1 shows two grouping strategies: Figure 4.1(a) shows a grouping along the direction $[0,1]$ with size 2, and Figure 4.1(b) shows a grouping along the direction $[0,1]$ with size 2, as well as along the direction $[1,0]$ with size 2. For clearness, the original dependence vectors are not shown. After the grouping, the iteration space and dependence vectors of the regular loop may be changed:

- Figure 4.1(a) corresponds to the regular loop $([1,8] \times [1,4], \{[0,1],[1,1],[1,0]\})$,
- Figure 4.1(b) corresponds to the regular loop $([1,4] \times [1,4], \{[0,1],[1,1],[1,0]\})$,

Together with the original regular loop, the parallel execution time $(t + ms)l$ of these regular loops are summarized below:

- Original loop: $(t + 3s) \cdot 15$
- The loop in Figure 4.1(a): $(2t + 3s) \cdot 11$
- The loop in Figure 4.1(b): $(4t + 3s) \cdot 7$

Note that the larger the grain size, the shorter is the longest path. Which grouping strategies is better highly depends on the execution time of an iteration t and the synchronization cost s. For example, when $t = s = 1$, the grouping strategy for Figure 4.1(b), whose parallel execution time is 49, is the best among the above three possibilities. When $s = 1$ and $t = 2$, the original loop, whose parallel execution time is 75, is better than the other two loops generated by grouping. Actually, when $\frac{s}{t} < \frac{13}{24}$, the original loop has the least parallel execution time; and when $\frac{s}{t} > \frac{13}{24}$, the loop generated by using the grouping in Figure 4.1(b) has the least parallel execution time. This result is consistent with our intuition. That is, when the s/t ratio is high, grouping is beneficial since it reduces the overall synchronization cost. At one extreme case, when the s/t ratio is infinite, we will group all iterations into one execution unit, i.e., assign all iterations to only one processing element. On the other hand, when the s/t ratio is zero, we will let each execution unit contain only one iteration.

Figure 4.2: Grouping along $[1,1]$ with size 2 for the regular loop $([1,4] \times [1,4],\ \{[0,1],[1,1],[1,0]\})$.

There are many other ways of grouping iterations into an execution unit. However, some grouping methods are infeasible in the sense that it may introduce cycles in the dependence graph. For example, consider the regular loop $([1,4] \times [1,4], \{[0,1],[1,1],[1,0]\})$. Suppose the iterations are grouped together along the direction $[1,1]$ with size 2 (Figure 4.2(a)), the resulted loop will be $([1,5] \times [1,2], \{[1,0],[-1,0],[1,1],[2,1]\})$ (Figure 4.2(b)). Note that the "dependence vectors" $[1,0]$ and $[-1,0]$ in the resulted loop introduce cycles in the dependence graph. Actually, the dependence vector $[-1,0]$ is not allowed in our model. Therefore, this grouping strategy is infeasible.

4.3.2 Scheduling "Critical" Iterations Earlier

Now we discuss another approach to optimize our basic scheduling scheme. From the dependence graph, it can be observed that an iteration near the "border" of the iteration space has less dependent successors than does an iteration at the "interior" of the iteration space. That is, when a border iteration is completed, it can "fire" less iterations than does an interior iteration. If the iterations at ready states are fetched to execute by free processors in arbitrary order, it may result in a situation where a free processor has no iteration to fetch. For example, consider a regular loop $([1,4] \times [1,4], \{[0,1],[1,0]\})$. Suppose two processors are available to execute the loop and each iteration takes 1 unit computation time. A possible execution order would be:

	time = 0	time = 1	time = 2	time = 3	time = 4
Processor 1	$[1,1]$	$[1,2]$	$[1,3]$	$[1,4]$	$[2,2]$
Processor 2		$[2,1]$	$[3,1]$	$[4,1]$	

By this execution order, only one iteration (i.e. $[2,2]$) are available for execution at time 4. This under-utilizes the available processors and hence will increase the total execution time sometimes. On the other hand, another possible execution order would be:

	time = 0	time = 1	time = 2	time = 3	time = 4
Processor 1	$[1,1]$	$[1,2]$	$[1,3]$	$[1,4]$	$[2,4]$
Processor 2		$[2,1]$	$[2,2]$	$[2,3]$	$[3,1]$

In this case, two iterations are available for execution at time 4. Therefore, This motivates us to schedule those "critical" iterations as soon as possible, especially when the number of available processing elements are limited.

The first thing we can do is to merge the data structure INIT to the data structure READY. Note that all of the iterations in INIT are at ready states. Therefore, this saves the space for INIT.

The second thing we can do is to organize READY as a priority queue, where the "priority" for each iteration in the queue is equal to the number of dependent successors of the iteration.

With these modifications, critical iterations at ready states will be assigned higher priorities in the queue READY. In other words, these critical iterations will be executed by free processors as earlier as possible.

There are other views of which iterations are critical. For example, one may regard iterations on the longest paths as more critical, because the parallel execution time is dominated by the longest paths. More works are needed to define "critical" iterations and to evaluate the effects to the parallel execution times.

4.3.3 Reducing Synchronization Cost

Finally, we propose an approach to reduce the synchronization time. Consider the regular loop $([1, U_1] \times \cdots \times [1, U_n], \{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\})$. According to Theorem 3.3, the maximal number of iterations in PENDING is $\sum_{i=1}^{m}(\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |d_{ij}|))$, which is large sometimes. Therefore, for the operation "Try to find the entry for *si* in PENDING" in the algorithm on page 46, it is unacceptable if the *si* in PENDING is searched sequentially.

In the following, we will propose a strategy organizing the data structure PENDING to reduce the access time to PENDING. In brief, the data structure PENDING is organized as a hash table, with $\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |c_j|)$ *buckets*, where $c = [c_1, \ldots, c_n] = [\sum_{i=1}^{m} d_{i1}, \ldots, \sum_{i=1}^{m} d_{in}]$, with the assumption that $|c_j| \leq U_j$ for $1 \leq j \leq n$ (This should account for most cases in practice, where U_j's are large and $|d_{ij}|$'s are small.). The hash function, which maps a given iteration (*key*) to a bucket in the hash table, is described in the following algorithm:

Algorithm 4.1 Consider the regular loop $([1, U_1] \times \cdots \times [1, U_n], \{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\})$. Let $c = [c_1, \ldots, c_n] = [\sum_{i=1}^{m} d_{i1}, \ldots, \sum_{i=1}^{m} d_{in}]$. In addition, assume that $|c_j| \leq U_j$ for $1 \leq j \leq n$. Then an iteration (key) $[x_1, \ldots, x_n]$ will be hashed to a bucket r by the following computation (assume that $c_j \geq 0$ for $1 \leq j \leq n$ for clarity):[1]

```
/* Find the initial iteration $[y_1, \ldots, y_n]$ which is the ancestor of $[x_1, \ldots, x_n]$ via
    the virtual dependence vector $c$ */
```

[1]We define $\sum_{s \in \phi} f(s) \equiv 0$ and $\prod_{s \in \phi} f(s) \equiv 1$.

```
 $a = \min\{\lfloor \frac{x_j - 1}{c_j} \rfloor \mid 1 \le j \le n,\ c_j \ne 0\}$;
 $p = \min\{j \mid \lfloor \frac{x_j - 1}{c_j} \rfloor = a$ for $1 \le j \le n$ and $c_j \ne 0\}$;
 $[y_1, \ldots, y_n] = [x_1 - a * c_1, \ldots, x_n - a * c_n]$;
/* find $s$, the sum of sizes from block 1 to block $p-1$, where block $i$ is the Cartesian
    space $[c_1 + 1, U_1] \times \cdots \times [c_{i-1} + 1, U_{i-1}] \times [1, c_i] \times [1, U_{i+1}] \times \cdots \times [1, U_n]$ */
 $s = \sum_{i=1}^{p-1} (\prod_{j=1}^{i-1} (U_j - c_j) c_i \prod_{j=i+1}^{n} U_j)$;
/* find $t$, the address of $[y_1, \ldots, y_n]$ within block $p$, where block $p$ is the Cartesian
    space $[c_1 + 1, U_1] \times \cdots \times [c_{p-1} + 1, U_{p-1}] \times [1, c_p] \times [1, U_{p+1}] \times \cdots \times [1, U_n]$ */
 let $[z_1, \ldots, z_n] = [y_1 - c_1, \ldots, y_{p-1} - c_{p-1}, y_p, \ldots, y_n]$;
 let $[v_1, \ldots, v_n] = [U_1 - c_1, \ldots, U_{p-1} - c_{p-1}, c_p, U_{p+1}, \ldots, U_n]$;
 $t = \sum_{i=1}^{n} (z_i - 1) \prod_{j=i+1}^{n} v_j$ ;
 $r = s + t + 1$;
```

$\square$

The basic idea of the above algorithm is that, given the iteration space $[1, U_1] \times \cdots \times [1, U_n]$, and a virtual dependence vector $c = [c_1, \cdots, c_n]$, all iterations can be divided into $\prod_{j=1}^{n} U_j - \prod_{j=1}^{n} (U_j - |c_j|)$ independent groups according to the proof of Theorem 3.2. The first step of the algorithm is to identify the group "leader" $[y_1, \cdots, y_n]$ from a given iteration (key) $[x_1, \cdots, x_n]$. Note that the set of all group leaders are equivalent to the set of all initial iterations for the virtual regular loop $([1, U_1] \times \cdots \times [1, U_n], \{c = [c_1, \cdots, c_n]\})$. Hence the second step of the algorithm is to map all initial iterations in the virtual loop to unique integers in the range from 1 to $\prod_{j=1}^{n} U_j - \prod_{j=1}^{n} (U_j - |c_j|)$. This mapping is derived by dividing all initial iterations into at most n n-dimensional "rectangular boxes" and then orders the iterations in each box according to row-major order. Figure 4.3 illustrates an example of such ordering.

Figure 4.3: Order all initial iterations for the regular loop $([1,5] \times [1,6], \{[2,3]\})$.

One advantage of the above access scheme is that the expected number of iterations in each bucket of the hash table is 1 for most cases. This is because for most cases, all of the dependent predecessors of iteration $I + d_1 + \ldots + d_m$ have iteration I as a dependent ancestor. Hence when iteration I is pending, iteration $I + d_1 + \ldots + d_m$ cannot become pending as it requires at least one of its dependent predecessors be finished, which in turn requires iteration I be finished.

A bucket of the hash table may contain more than one iterations only when those iterations near the iteration space boundaries are being executed, and for some dimensions, there are both positive and negative components of dependence vectors at those dimensions. For example, for the regular loop $([1,10] \times [1,10] \times [1,10], \{[0,1,-2],[1,-2,1],[1,0,2]\})$, iteration $[2,2,2]$ and iteration $[4,1,3]$ (which is equal to $[2,2,2] + (\ [0,1,-2] + [1,-2,1] + [1,0,2]\ )$) may be at the pending states simultaneously, as shown below:

For some certain type regular loops, we prove that any bucket of the hash table contains at most one iteration at any instance during the execution:

Theorem 4.1 *Given a regular loop* $([1,U_1] \times \cdots \times [1,U_n], \{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\})$*, any bucket of the hash table contains at most one iteration at any instance during the execution if either one of the following conditions is satisfies:*

1. $n = 2$ *and* $d_{i2} < \frac{1}{2}U_2$*, for all* $1 \leq i \leq m$.
2. *For each* $1 \leq j \leq n$*, either all* d_{ij}*'s are non-negative or all* d_{ij}*'s are non-positive.*

Proof. Assume iteration I and $I' \equiv I + d_1 + \cdots + d_m$ are both in the iteration space, and iteration I is at pending state. In addition, suppose on the contrary that iteration I' is also at pending state. Then by the definition of the pending state, at least one of the iterations $I' - d_k$, for $1 \leq k \leq m$, must be at finished state. However, by similar discussions to that of the proofs for Theorem 3.10 (for Condition 1) and Theorem 3.11 (for Condition 2), iteration $I' - d_k$ must be dependent on iteration I via a "path" totally inside the iteration space. That is, iteration $I' - d_k$ cannot be at finished state, neither can iteration I'. This completes our proof. $\square$

Finally, we prove that the number of buckets in the hash table is less than or equal to the bound we got in Theorem 3.3:

Theorem 4.2 *Consider a regular loop* (S, D)*, where* $S = [1,U_1] \times [1,U_2] \times \cdots \times [1,U_n]$*, and* $D = \{d_i = [d_{i1}, \ldots, d_{in}] \mid 1 \leq i \leq m\}$*. Assume that* $U_j \geq |\sum_{i=1}^{m} d_{ij}|$ *for* $1 \leq j \leq n$*. Then we have* $\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |\sum_{i=1}^{m} d_{ij}|) \leq \sum_{i=1}^{m}(\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |d_{ij}|))$.

Proof. We prove the theorem by considering two cases:

Case 1. $\prod_{i=1}^{m} |d_{ij}| \geq U_j$ for some $1 \leq j \leq n$:

Without loss of generality, assume that $\prod_{i=1}^{m} |d_{i1}| \geq U_1$. Then we have:

$$\begin{aligned}
& \sum_{i=1}^{m}(\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |d_{ij}|)) \\
\geq \;& \sum_{i=1}^{m}(\prod_{j=1}^{n} U_j - (U_1 - |d_{i1}|)\prod_{j=2}^{n} U_j) \\
= \;& (\sum_{i=1}^{m} |d_{i1}|)\prod_{j=2}^{n} U_j \\
\geq \;& \prod_{j=1}^{n} U_j
\end{aligned}$$

Furthermore, since we assumed that $U_j \geq |\sum_{i=1}^{m} d_{ij}|$, for $1 \leq j \leq n$, we have:

$$\begin{aligned}
& \prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |\sum_{i=1}^{m} d_{ij}|) \\
\leq \;& \prod_{j=1}^{n} U_j - \prod_{j=1}^{n} 0 \\
\leq \;& \sum_{i=1}^{m}(\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |d_{ij}|))
\end{aligned}$$

The theorem is proved for this case.

Case 2. $\prod_{i=1}^{m} |d_{ij}| < U_j$ for all $1 \leq j \leq n$:

We first prove the following equation holds:

$$\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |d_{ij}|) = \sum_{j=1}^{n}(\prod_{k=1}^{j-1}(U_k - |d_{ik}|)|d_{ij}| \prod_{k=j+1}^{n} U_k \;) \tag{4.2}$$

This equation can be shown true by induction on n. When $n = 1$, Equation 4.2 is obviously true. Suppose Equation 4.2 is true when $n = l$, then it is true also when $n = l + 1$, because:

$$\prod_{j=1}^{l+1} U_j - \prod_{j=1}^{l+1}(U_j - |d_{ij}|)$$

$$
\begin{aligned}
&= (U_1 - |d_{i1}| + |d_{i1}|)\prod_{j=2}^{l+1} U_j - (U_1 - |d_{i1}|)\prod_{j=2}^{l+1}(U_j - |d_{ij}|) \\
&= |d_{i1}|\prod_{j=2}^{l+1} U_j + (U_1 - |d_{i1}|)(\prod_{j=2}^{l+1} U_j - \prod_{j=2}^{l+1}(U_j - |d_{ij}|)) \\
&= |d_{i1}|\prod_{j=2}^{l+1} U_j + (U_1 - |d_{i1}|)\sum_{j=2}^{l+1}(\prod_{k=2}^{j-1}(U_k - |d_{ik}|)|d_{ij}|\prod_{k=j+1}^{l+1} U_k\ ) \\
&= \sum_{j=1}^{l+1}(\prod_{k=1}^{j-1}(U_k - |d_{ik}|)|d_{ij}|\prod_{k=j+1}^{l+1} U_k\ )
\end{aligned}
$$

Based on Equation 4.2, We further prove the following claim:

if $U_j \geq V_j$ then

$$\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |d_{ij}|) \geq \prod_{j=1}^{n} V_j - \prod_{j=1}^{n}(V_j - |d_{ij}|)$$

This claim is true because:

$$
\begin{aligned}
&\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |d_{ij}|) \\
&= \sum_{j=1}^{n}(\prod_{k=1}^{j-1}(U_k - |d_{ik}|)|d_{ij}|\prod_{k=j+1}^{n} U_k\ ) \qquad \text{(By Equation 4.2)} \\
&\geq \sum_{j=1}^{n}(\prod_{k=1}^{j-1}(V_k - |d_{ik}|)|d_{ij}|\prod_{k=j+1}^{n} V_k\ ) \qquad \text{(Since } U_k \geq V_k\text{)} \\
&= \prod_{j=1}^{n} V_j - \prod_{j=1}^{n}(V_j - |d_{ij}|) \qquad \text{(By Equation 4.2)}
\end{aligned}
$$

Now we prove the theorem. Because of $|\sum_{i=1}^{m} d_{ij}| \leq \sum_{i=1}^{m} |d_{ij}|$, we have:

$$\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |\sum_{i=1}^{m} d_{ij}|) \leq \prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - \sum_{i=1}^{m} |d_{ij}|) \tag{4.3}$$

Let $V_j^i = U_j - \sum_{k=1}^{i-1} |d_{kj}|$ for $1 \leq j \leq n$, $1 \leq i \leq m$. We can reduce the right-hand side of Inequality 4.3 as below:

$$\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - \sum_{i=1}^{m} |d_{ij}|)$$

$$
\begin{aligned}
&= \prod_{j=1}^{n} V_j^1 + \sum_{i=2}^{m}\prod_{j=1}^{n} V_j^i - \sum_{i=1}^{m-1}\prod_{j=1}^{n}(V_j^i - |d_{ij}|) - \prod_{j=1}^{n}(V_j^m - |d_{mj}|) \\
&= \sum_{i=1}^{m}(\prod_{j=1}^{n} V_j^i - \prod_{j=1}^{n}(V_j^i - |d_{ij}|)) \\
&\leq \sum_{i=1}^{m}(\prod_{j=1}^{n} U_j - \prod_{j=1}^{n}(U_j - |d_{ij}|)) \qquad \text{(By the claim above)}
\end{aligned}
$$

By combining Inequality 4.3 and the above inequality, the theorem is proved. $\square$

A different approach for the proof of Theorem 4.2 can be found in [96]. The proof presented here is much simpler than that one.

Chapter 5

The Mapping Problem and its Complexity for Distributed–Memory Multiprocessors

In this chapter, we formulate the problem of mapping regular loops onto distributed-memory MIMD machines, and discuss its complexity. The hypercube structure is used for our discussion whenever the interconnection network needs to be specified. The complexity of executing an *arbitrary program* on a multiprocessor has been studied extensively (please see [50, Section A5.2] as examples). However, because our regular loop is represented in compact form instead of in its unrolled form, the techniques used to discuss the complexity of executing arbitrary programs cannot be applied directly to our case. In Section 5.1, the mapping problem is formally defined and some terminology is introduced. In Section 5.2, the problem complexity is discussed.

5.1 Problem Statement

In this section, we will formally define the mapping problem. We introduce two ways of representing the problem: the *free* mapping and the *linear* mapping. The terms *free* and *linear* were originally used in [71]. Section 5.1.1 defines the mapping problem by unrolling the whole loops. By defining the problem in this way, both the communication costs and communication conflicts can be stated explicitly. However, as discussed in Chapter 2, the size of an unrolled loop can be exponential in the size of its compact representation. This implies that solving a problem with the loop unrolled takes time at least exponential in the size of its compact loop representation. This is not desirable sometimes. In Section 5.1.2, the mapping problem is defined by two linear functions: one is for defining the execution time of an iteration, the other is for defining the execution place (processor) of an iteration. These two functions can be represented in size that is linear in the compact loop representation. However, the communication costs and communication conflicts are not taken care of explicitly in this case.

5.1.1 The Free Mapping Problem

Let the regular loop that will be mapped to a distributed-memory MIMD machine be (S, D), where the iteration space S is $[1, U_1] \times \cdots \times [1, U_n]$ and the set of dependence vectors D be $\{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\}$. In addition, as a concrete example, let us assume the MIMD machine is represented by a graph (V, E) with a hypercube interconnection topology.

The complete definition of a mapping problem should specify:

- the location (processor) where an iteration is executed,
- the starting time when an iteration is executed,

- the path which is used for iteration X to send a message to iteration $X + d_i$, where $X \in S$ and $d_i \in D$, and
- the schedule that a message uses as it traverses from one processor to an adjacent processor over a path.

Mathematically, we have:

- $\sigma : S \rightarrow V$

  $\sigma(X)$ is the hypercube node where iteration $X \in S$ will be executed.
- $\alpha : S \rightarrow Z_0$, where Z_0 is the set of non-negative integers.

  $\alpha(X)$ is the time which iteration $X \in S$ starts to be executed.
- $\beta : S \times D \rightarrow Z_0$, where Z_0 is defined as above.

  $\beta(X, d_i)$ represents the number of links in the hypercube path, which is the image of the edge $X \rightarrow X + d_i$ in the dependence graph for the regular loop. If iteration $X \in S$ and $X + d_i \in S$ are mapped to the same hypercube nodes (i.e., $\sigma(X) = \sigma(X + d_i)$), then $\beta(X, d_i) = 0$. Note that the hypercube path should consist of adjacent hypercube links only. In addition, the domain of the function β should exclude those pairs (X, d_i)'s where the iteration $X + d_i \notin S$. However, for simplicity, we will not formulate this fact explicitly.
- $\gamma : S \times D \times M \rightarrow V$, where M varies depending on different $X \in S$ and $d_i \in D$. Given specific $X \in S$ and $d_i \in D$, define $M = \{1, 2, \cdots, \beta(X, d_i)\}$ when $\beta(X, d_i) > 0$.

  $\gamma(X, d_i, m)$ represents the hypercube node that is the source of the m-th link in the hypercube path, which is the image of the edge $X \rightarrow X + d_i$ in the dependence graph. Note that $\gamma(X, d_i, 0) = \sigma(X)$.

- $\delta : S \times D \times M \to Z_0$, where M and Z_0 are defined as above.

  $\delta(X, d_i, m)$ represents the time that the hypercube node $\gamma(X, d_i, m)$ sends out the message, starting from node $\sigma(X)$ and ending at node $\sigma(X + d_i)$.

To be a valid mapping, the above functions must also satisfy several constraints. For simplicity of our discussion, we will let

- the execution (computation) time of a single iteration be e, and
- the synchronization (communication) time between a pair of adjacent processors be c.

The constraints for the above functions are as follows:

Hypercube Structures For each edge in the dependence graph for (S, D), the image of this edge on the hypercube must use only hypercube links:

For $X \in S$ and $d_i \in D$, if $\beta(X, d_i) > 0$ then

$$\gamma(X, d_i, 1) = \sigma(X)$$
$$count(\gamma(X, d_i, j) \oplus \gamma(X, d_i, j+1)) = 1 \quad \text{for } 1 \leq j < \beta(X, d_i)$$
$$count(\gamma(X, d_i, \beta(X, d_i)) \oplus \sigma(X + d_i)) = 1$$

where $count(x)$ is the number of 1's in the binary representation of x.

Resource Congestion Avoidance

Computation If two iterations are mapped to the same hypercube node, then their computation time cannot be overlapped:

if $\sigma(X_1) = \sigma(X_2)$, then either

$$\alpha(X_1) \geq \alpha(X_2) + e$$

or

$$\alpha(X_1) + e \leq \alpha(X_2)$$

Communication If two messages traverse the same hypercube link, then their communication time cannot be overlapped (it is assumed that all the links connected to a hypercube node can send/receive data independently at the same time):

if $\gamma(X_1, d_i, m_1) = \gamma(X_2, d_j, m_2)$ and $\gamma(X_1, d_i, m_1+1) = \gamma(X_2, d_j, m_2+1)$, then either

$$\delta(X_1, d_i, m_1) \geq \delta(X_2, d_j, m_2) + c$$

or

$$\delta(X_1, d_i, m_1) + c \leq \delta(X_2, d_j, m_2)$$

Remark: When $m_1 = \beta(X_1, d_i)$, let $\gamma(X_1, d_i, m_1 + 1)$ be $\sigma(X_1 + d_i)$. The case is defined similarly for m_2.

Data Dependence Preservation The data dependencies stipulated by the regular loop must be satisfied. That is, an iteration cannot be started until it has received all of the data from its immediately dependent predecessors.

- If an iteration and its immediately dependent predecessor are mapped onto the same hypercube node, then the communication time is assumed to be zero:

  if $\sigma(X) = \sigma(X + d_i)$ then $\alpha(X) + e \leq \alpha(X + d_i)$.

- If an iteration and its immediately dependent predecessor are mapped onto different hypercube nodes, then the communication time must be taken into consideration:

if $\sigma(X) \neq \sigma(X + d_i)$ then

- $\delta(X, d_i, 1) \geq \alpha(X)+e$. That is, an iteration can send out a message only after the computation of this iteration has been finished.
- $\delta(X, d_i, j + 1) \geq \delta(X, d_i, j) + c$ for $1 \leq j < \beta(X, d_i)$. That is, a hypercube node can pass a message to the next node only after it has receive the message from the previous node.
- $\alpha(X + d_i) \geq \delta(X, d_i, \beta(X, d_i)) + c$. That is, an iteration can start its execution only after it has received the message from its immediately dependent predecessors.

Note that the parallel execution time of the mapped regular loop on the hypercube machine is $\max\{\alpha(X) + e \mid \forall X \in S\}$. The optimal mapping problem is to find a mapping (i.e., to compute the values of the above functions which satisfy their constraints) so that the parallel execution time is minimal.

It has been shown that the problem of optimally mapping an *arbitrary program* onto a multiprocessor machine is NP-complete [50]. We will prove our optimal mapping problem (a special case of the more general problem) is NP-hard in Section 5.2.

5.1.2 The Linear Mapping Problem

It can be observed from the last section that even specifying a free mapping takes a large amount of space (and hence time). This is not always desirable. Therefore, *linear mappings* are often used instead.

As before, let us consider the regular loop (S, D), where the iteration space S is $[1, U_1] \times \cdots \times [1, U_n]$ and the set of dependence vectors D is $\{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\}$. A linear mapping consists of two mapping vectors: one vector is $\Pi = [\pi_1, \cdots, \pi_n]$ for determining the starting time to execute an iteration, the other vector is $H = [h_1, \cdots, h_n]$

for determining the location (processor) to execute an iteration. Mathematically, we have:

- $\Pi \cdot X$, where "$\cdot$" is the inner product of two vectors, is the starting time to execute iteration $X \in S$.
- $H \cdot X$, where "$\cdot$" is the inner product of two vectors, is the processor to execute iteration $X \in S$.

Since the path to send a message on the parallel machine is not specified explicitly, an execution scheme must have some way of specifying a path and to solve the communication congestion implicitly. This implicit strategy depends on the target machine's structure. We will address the implicit communication protocol for our mapping strategy in Chapter 6.

Let us assume that the execution time of an iteration is one unit, and the communication cost between two distinct processors is zero. Then a valid mapping must satisfy the following conditions:

1. $\Pi \cdot X \in Z$ and $H \cdot X \in Z$, for all $X \in S$, and Z is the set of all integers.

   This means the times and the processor labels should be integers. An immediate consequence of this condition is that every entry of Π and H must also be an integer. This consequence can be easily proved as follows. Let $X = [x_1, \cdots, x_n]$ and $Y = [y_1, \cdots, y_n]$ be two iterations in S. Suppose $x_1 = y_1 + 1$ and $x_j = y_j$ for $2 \leq j \leq n$. Since $\Pi \cdot X$ and $\Pi \cdot Y$ are both in Z, it is clear that $\Pi \cdot X - \Pi \cdot Y = \Pi \cdot (X - Y) = \Pi \cdot [1, 0, \cdots, 0] = \pi_1$ must be in Z also. Other cases can be proved similarly.

2. $\Pi \cdot d_i > 0$, for all $d_i \in D$.

   For any two dependent iterations X and $X + d_i$ in S, the execution time of iteration $X + d_i$ must be after that of iteration X. Hence we have $\Pi \cdot (X + d_i) - \Pi \cdot X = \Pi \cdot d_i > 0$.

3. If $\Pi \cdot (X_1 - X_2) = 0$, then $H \cdot (X_1 - X_2) \neq 0$, for all $X_1, X_2 \in S$ and $X_1 \neq X_2$.

   This says that two distinct iterations X_1 and X_2 cannot be executed at the same time by the same processor. Note that $\Pi \cdot (X_1 - X_2) = 0$ represents that iterations X_1 and X_2 are executed at the same time, and $H \cdot (X_1 - X_2) \neq 0$ represents that iterations X_1 and X_2 are executed at different processors.

Note that the linear mapping scheme has been used extensively in the context of systolic arrays [84] [91] [105]. In that setting, an architecture is designed to fit a regular loop. In contrast, in our setting, a regular loop is transformed to fit a fixed architecture. Since in the former setting, an architecture is "synthesized" instead of being given, additional conditions than those listed above may be specified. On the other hand, in our setting, the communication paths and the avoidance of communication conflicts must be specified by other rules in the mapping strategy.

It can be observed that the parallel execution time of a linearly mapped regular loop is $\max\{\Pi Y - \Pi Z + 1 \mid \forall Y, Z \in S\}$. It has been shown in [71] that the optimal parallel execution times between a linear mapping and a fast mapping are within a constant of each other. In the next section, we will show that determining the optimal parallel execution time of a linearly mapped regular loop is NP-hard. This is important, as it supports the use of exponential time algorithms [84] [122] to find optimal linear mappings, since we show that even *determining* the optimal execution time is NP-hard!

5.2 The Problem Complexities

In this section, we will prove the following problems to be NP-hard:

Problem Q1: Is there a mapping of a given regular loop onto a MIMD machine of any size so that the parallel execution time is less than or equal to T?

Problem Q2: Is there a mapping of a given regular loop onto a MIMD machine of p processors so that the parallel execution time is less than or equal to T?

Problem Q3: Is there a *linear* mapping of a given regular loop onto a MIMD machine of any size so that the parallel execution time is less than or equal to T?

As before, the regular loop that will be considered is still denoted by (S, D), where $S = [1, U_1] \times \cdots \times [1, U_n]$ and $D = \{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\}$. We first prove that:

Theorem 5.1 *Problem Q1 is NP-hard.*

Proof. Recall that in Theorem 3.5, we proved that Problem P2 is NP-hard. That is, it is NP-hard for the problem of deciding, in a dependence graph (represented in compact form), if there is a path with length greater than or equal to B.

To prove Problem Q1 to be NP-hard, we reduce Problem P2 to Problem Q1. Let the computation cost of each iteration by a processor be 1 unit, and the communication cost between any two processors be zero. Then it is clear that the length of a longest path in the dependence graph is one less than the optimal parallel execution time of the regular loop. That is,

- If there is a mapping so that the parallel execution time is less than or equal to B, then there is no path with length greater than or equal to B.
- If there is no mapping so that the parallel execution time is less than or equal to B, then there is a path with length greater than or equal to B.

Therefore, Problem P2 can be solved by applying Problem Q1. Since Problem P2 is NP-hard, we conclude that Problem Q1 is also NP-hard. $\square$

The above theorem says that it is "hard" to decide the optimal execution time of mapping a regular loop onto a MIMD machine with *any* number of processors. The next theorem says that it is still "hard" if the MIMD machine has p processors, that is:

Theorem 5.2 *Problem Q2 is NP-hard.*

Proof. Note that a regular loop contains $U_1 \times U_2 \times \cdots \times U_n$ iterations. Since each iteration is executed by at most one processor, the optimal execution time cannot be improved any further if the number of processors used is beyond $U_1 \times U_2 \times \cdots \times U_n$. Therefore, Problem Q1 can be solved by applying Problem Q2 with the number of processors p set to $U_1 \times U_2 \times \cdots \times U_n$. Since Problem Q1 is NP-hard, we conclude that Problem Q2 is also NP-hard. $\square$

Now we prove Problem Q3 is also NP-hard. To simplify our proof, we only consider the "time mapping", and neglect the constant "1" in the parallel execution time $\max\{\Pi Y - \Pi Z + 1 \mid \forall\ Y, Z \in S\}$. Mathematically, the optimal linear mapping problem Q3 can be represented as follows:

Q31: **minimize** $\max\{\Pi \cdot Y - \Pi \cdot Z \mid \forall \ \ Y = [y_1, \cdots, y_n],\ Z = [z_1, \cdots, z_n] \in S\}$

subject to:

$$\begin{array}{ll} \Pi \cdot d_i > 0 & \forall\ d_i \in D \\ 1 \le y_j,\ z_j \le U_j & \text{for } 1 \le j \le n \end{array}$$

To simplify the above problem, let $X = [x_1, \cdots, x_n] = Y - Z$ and let $V_j = U_j - 1$ for $1 \le j \le n$. Then the above problem is equivalent to the following problem:

Q32: **minimize** $\max\{\Pi \cdot X \mid \forall\ X\ \}$

subject to:

$$\begin{array}{ll} \Pi \cdot d_i > 0 & \forall\ d_i \in D \\ -V_j \le x_j \le V_j & \text{for } 1 \le j \le n \end{array}$$

Note that when $\Pi = [\pi_1, \cdots, \pi_n]$ is decided, $X = [x_1, \cdots, x_n]$ can be decided immediately. This is because we want to find the maximal value of $\Pi \cdot X = \pi_1 x_1 + \cdots + \pi_n x_n$. Hence if π_j is positive, the value of x_j will be V_j; and if π_j is negative, the value of x_j will be $-V_j$. In either case, the value of $\pi_j x_j$ will be $|\pi_j|\ V_j$. Therefore, the above problem is equivalent to the following one:

Q33: **minimize** $|\pi_1|V_1 + \cdots + |\pi_n|V_n$

subject to:

$$d_{11}\pi_1 + d_{12}\pi_2 + \cdots + d_{1n}\pi_n > 0$$
$$d_{21}\pi_1 + d_{22}\pi_2 + \cdots + d_{2n}\pi_n > 0$$
$$\cdots\cdots$$
$$d_{m1}\pi_1 + d_{m2}\pi_2 + \cdots + d_{mn}\pi_n > 0$$

Instead of proving the above problem to be NP-hard directly, we prove a special case of the above problem to be NP-hard. The special case is by letting $V_1 = V_2 = \cdots = V_n$. Hence the problem of minimizing $|\pi_1|V_1 + \cdots + |\pi_n|V_n = (|\pi_1| + \cdots + |\pi_n|)V_1$ is equivalent to that of minimizing $|\pi_1| + \cdots + |\pi_n|$, because V_i's are constants. The new problem is as follows:

Q34: **minimize** $|\pi_1| + \cdots + |\pi_n|$

subject to:

$$d_{11}\pi_1 + d_{12}\pi_2 + \cdots + d_{1n}\pi_n > 0$$
$$d_{21}\pi_1 + d_{22}\pi_2 + \cdots + d_{2n}\pi_n > 0$$
$$\cdots\cdots$$
$$d_{m1}\pi_1 + d_{m2}\pi_2 + \cdots + d_{mn}\pi_n > 0$$

Lemma 5.3 *If Problem Q34 is NP-hard, then Problem Q33 is NP-hard.*

Proof. Since Problem Q34 is a special case of Problem Q33, We can always use Problem Q33 to solve Problem Q34. The proof follows immediately. $\square$

To facilitate our proof, we change the *optimization* problem Q34 to a *decision* problem:

Q35: If there exist $[\pi_1, \cdots, \pi_n]$ so that $|\pi_1| + \cdots + |\pi_n| \leq K$

subject to:

$$d_{11}\pi_1 + d_{12}\pi_2 + \cdots + d_{1n}\pi_n > 0$$
$$d_{21}\pi_1 + d_{22}\pi_2 + \cdots + d_{2n}\pi_n > 0$$
$$\cdots\cdots$$
$$d_{m1}\pi_1 + d_{m2}\pi_2 + \cdots + d_{mn}\pi_n > 0$$

Since an optimization problem can be reduced to a decision problem, we have the following lemma:

Lemma 5.4 *If Problem Q35 is NP-hard, the Problem Q34 is also NP-hard.*

Proof. It is easy to have a Turing reduction from Problem Q35 to Problem Q34. The proof follows immediately. $\square$

Now we prove Problem Q35 to be NP-hard. Note that there is at least one feasible solution $[\pi_1, \cdots, \pi_n]$ to Problem Q35, because for every dependence vector $d_i = [d_{i1}, \cdots, d_{in}]$, the leftmost nonzero entry must be positive. We will reduce the Hitting Set Problem [50, Problem SP8, page 222] to Problem Q35.

The Hitting Set Problem is defined as follows:

Definition *Let $A = \{a_1, a_2, \cdots, a_n\}$ be a finite set of n elements, $B_1, B_2, \cdots, B_m$ be m subsets of the finite set A, and $K \leq n$ be a positive integer. Is there a subset A' of A with $|A'| \leq K$ so that A' contains at least one element from each subset $B_1, B_2, \cdots, B_m$?*

Lemma 5.5 *Problem Q35 is NP-hard.*

Proof. Since the Hitting Set Problem is known to be NP-complete [50, Problem SP8, page 222], we need only to find a polynomial time reduction from the Hitting Set Problem to Problem Q35 so that the former problem answers "yes" if and only if the latter problem answers "yes".

The reduction is as follows. For each subset B_i, $1 \leq i \leq m$, if $a_j \in B_i$, then let $d_{ij} = 1$; otherwise, let $d_{ij} = 0$. It is easy to see that this transformation can be completed in polynomial time.

Note that since each subset B_i is non-empty, the corresponding dependence vector d_i must have at least one nonzero entry. Furthermore, each entry of d_i can only be either 0 or 1. Hence the requirement of a dependence vector (the leftmost non-zero entry must be positive) is satisfied.

We now prove that the Hitting Set Problem answers "yes" if and only if Problem Q35 answers "yes". Suppose the answer to the Hitting Set Problem is "yes", then we have a hitting set A', where $|A'| \leq K$, which "hits" every subset B_i at least once. If $a_j \in A'$, then let $\pi_j = 1$; otherwise, let $\pi_j = 0$. It is clear that the conditions of Problem Q35 (the "subject to" part) are satisfied and $|\pi_1| + \cdots + |\pi_n| = \pi_1 + \cdots + \pi_n \leq K$. In other words, the answer to Problem Q35 is also "yes".

Suppose on the contrary that the answer to Problem Q35 is "yes", then we have a vector $[\pi_1, \cdots, \pi_n]$ so that the conditions of Problem Q35 are satisfied and $|\pi_1| + \cdots + |\pi_n| \leq K$. Since each of the d_{ij}'s is either 0 or 1, a negative π_j, provided one exists, can always be changed to a positive one without altering the satisfiability of Problem Q35. Hence we can assume $\pi_j \geq 0$, for $1 \leq j \leq n$. If $\pi_j > 0$, then let $a_j \in A'$; otherwise, let $a_j \notin A'$. It is clear that the set A' hits every subset B_i and $|A'| \leq K$. In other words, the answer to the Hitting Set Problem is also "yes". $\square$

Now we are ready to state our main theorem:

Theorem 5.6 *Problem Q3 is NP-hard.*

Proof. By the fact that Problem Q31, Q32, and Q33 are equivalent, and by Lemma 5.3, 5.4, and 5.5, the theorem follows immediately. □

Chapter 6

Executing Regular loops on Distributed–Memory Multiprocessors

In this chapter, we propose a strategy of mapping regular loops onto distributed-memory MIMD machines by applying linear mapping techniques [71] [82] [84] [87] [122]. As shown in the previous chapter, finding an optimal linear mapping is computationally intractable. In other words, finding an optimal linear mapping needs at least exponential time computation in practice. Note that exponential time algorithms [84] [91] are usually used to find optimal linear mappings of regular loops onto systolic arrays. This amount of time is worthwhile because the systolic arrays will be used repeatedly. However, on a distributed-memory MIMD machine, if the regular loops are compiled and executed only a few times, then the compiling time (for finding the mappings) and the computational time (for executing the mapped loops) should be compromised. In this chapter, heuristic algorithms are devised to find mappings of regular loops onto MIMD machines. The time complexity of the heuristic algorithms is only linear in the problem size. In addition, the efficiency of the parallelized loop (generated by our algorithms) is also verified by experiments.

A simple mapping strategy is derived by combining existing work. For example, to map a regular loop onto a distributed-memory hypercube machine, one can "concatenate" the following two methods:

1. Mapping a regular loop onto a systolic array [84] [91] [105].

2. Mapping a systolic array onto a hypercube machine [66].

However, this approach has the following drawback. Suppose iteration X and Y are two interdependent iterations in a dependence graph as shown in Figure 6.1(a). After step 1 of the mapping strategy, the regular loop is mapped onto a systolic array as shown in Figure 6.1(b). Note that the edge $X \rightarrow Y$ in the dependence graph is lengthened into a sequence of links in the systolic array. After step 2 of the mapping strategy, the systolic array is mapped onto a 2-dimensional hypercube as shown in Figure 6.1(c). Note that the sequence of links from X to Y in Figure 6.1(b) is mapped onto the path $00 \rightarrow 01 \rightarrow 11 \rightarrow 10$ in Figure 6.1(c). Therefore, in the 2-dimensional hypercube, iteration X will send a message to iteration Y by passing through the path $00 \rightarrow 01 \rightarrow 11 \rightarrow 10$, instead of the shortest path $00 \rightarrow 10$. This is because step 2 of the mapping strategy does not know that X and Y are two immediately interdependent iterations. Therefore, a more "integrated" strategy is preferred. In the following, we will describe such a mapping strategy.

Throughout the rest of this chapter, the regular loop to be considered is of the form (S, D), where $S = [0, U_1] \times \cdots \times [0, U_n]$, and $D = \{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\}$. The set of integers is denoted by Z, and the set of non-negative integers is denoted by Z_0. In addition, the computation time of each iteration is assumed to be one unit.

Figure 6.1: A two step mapping strategy. Step 1: map a regular loop in (a) to a systolic array in (b). Step 2: map a systolic array in (b) to a 2-dimensional hypercube in (c).

Our heuristic algorithm for mapping the regular loop onto a distributed-memory MIMD machine involves three steps:

1. Find a *time mapping function* $\mathcal{T}$ such that iteration $X \in S$ is executed at time $\mathcal{T}(X)$. Function $\mathcal{T}$ must satisfy the condition that the execution time of iteration $X + d_i$ is greater than that of iteration X, that is, $\mathcal{T}(X + d_i) > \mathcal{T}(X)$, for all $d_i \in D$ and $X, X + d_i \in S$.

2. Find a *processor mapping function* $\mathcal{P}$ such that iteration $X \in S$ is executed on processor $\mathcal{P}(X)$. Function $\mathcal{P}$ must satisfy the condition that for *any* two iterations executed on the same processor, their execution times are different, that is, if $\mathcal{P}(X) = \mathcal{P}(Y)$, then $\mathcal{T}(X) \neq \mathcal{T}(Y)$, for any $X, Y \in S$.

3. If the number of available processors (called *real* processors) is less than the number of processors required by step 2 (called *virtual* processors), then fold the *virtual* processors to fit in the *real* processors.

The following three sections will discuss these three steps at more detail. In addition, the mapped loop exploits *cyclic shift* operations to transmit data among inter-dependent iterations, if these iterations are assigned to different processors. In the next chapter, we will study cyclic shift algorithms on distributed-memory hypercube machines.

6.1 Finding the Time Mapping Function

In this section, we will determine the time mapping function $\mathcal{T}$ so that an iteration $X \in S$ will be executed at time $\mathcal{T}(X)$. The time mapping function has to satisfy certain constraints. In addition, the parallel execution time of the mapped loop should be as small as possible. In the rest of this section, a basic time mapping function is first described, which is based on the linear mapping technique [82] [84] [91] [105]. With some minor modifications, a new time mapping function is then defined, which reduce the parallel execution time of the mapped loop.

6.1.1 A Basic Time Mapping Function

In this section, we will describe a basic time mapping function $\mathcal{T}$ so that an iteration $X \in S$ will be executed at time $\mathcal{T}(X)$. The discussion of this section is in the following order:

- The time mapping function $\mathcal{T}$ is first defined, and the *time mapping vector* Π is introduced.
- The constraint to the time mapping function (or equivalently, the time mapping vector) is stipulated.
- The optimization criteria of the time mapping function is derived.
- A simple algorithm to find the time mapping vector Π is described.
- An improved algorithm to find the time mapping vector Π is devised.

Consider an iteration $X = [x_1, \cdots, x_n]$ in S. The time mapping function is defined as follows:[1]

$$\mathcal{T}(X) = \Pi \cdot X = \pi_1 x_1 + \cdots + \pi_n x_n \tag{6.1}$$

where the vector $\Pi = [\pi_1, \cdots, \pi_n]$ is called a *time mapping vector*.

To satisfy the condition that an iteration will be executed after its immediately dependent predecessors, we have:

$$\mathcal{T}(X + d_i) > \mathcal{T}(X)$$

[1]To make the execution time of an iteration non-negative, the time mapping function $\mathcal{T}$ should be defined as $\mathcal{T}(X) = \Pi \cdot X + b$, where b is an integer to make the smallest $\mathcal{T}(X)$ be zero. However, for clarity of our discussion, we will allow negative execution times.

$$\Rightarrow \quad \Pi \cdot (X + d_i) > \Pi \cdot X$$

$$\Rightarrow \quad \Pi \cdot d_i > 0$$

That is, the time mapping vector must satisfy the condition that

$$\Pi \cdot d_i > 0 \quad \text{for every } d_i \in D \tag{6.2}$$

With the time mapping function $\mathcal{T}$ defined as above, the values of x_j and y_j, $1 \leq j \leq n$, can be computed as follows to get the parallel execution time T:

$$\begin{aligned} T &= \max\{|\mathcal{T}(X) - \mathcal{T}(Y)| \mid \forall\, X, Y \in S\} + 1 \\ &= \max\{|\Pi X - \Pi Y| \mid \forall\, X, Y \in S\} + 1 \\ &= \max\{|\Pi(X - Y)| \mid \forall\, X, Y \in S\} + 1 \\ &= \max\{|\sum_{j=1}^{n} \pi_j (x_j - y_j)| \mid 0 \leq x_j, y_j \leq U_j\} + 1 \end{aligned}$$

where $X = [x_1, \cdots, x_n]$ and $Y = [y_1, \cdots, y_n]$. The value of x_j and y_j can be set as follows to determine the value of T:

- When $\pi_j > 0$, let $x_j = U_j$, $y_j = 0$, hence we have $\pi_j(x_j - y_j) = \pi_j U_j > 0$.
- When $\pi_j = 0$, we have $\pi_j(x_j - y_j) = 0$ for any x_j and y_j.
- When $\pi_j < 0$, let $x_j = 0$, $y_j = U_j$, hence we have $\pi_j(x_j - y_j) = -\pi_j U_j > 0$.

Therefore, given a time mapping vector Π, the parallel execution time is

$$T = \sum_{j=1}^{n} |\pi_j| U_j + 1 \tag{6.3}$$

Our goal is to minimize the value of T.

However, as noted in the previous chapter, it is NP-hard to find an time mapping vector Π so that the constraint in Eq. (6.2) is satisfied and the parallel execution time in Eq. (6.3) is minimized. That is, finding an optimal time mapping vector Π will take at least exponential time in practice. Therefore, we will describe a heuristic algorithm below to find a suboptimal time mapping vector Π. This algorithm was also proposed independently in [84].

Algorithm 6.1 Given a regular loop (S, D), where $S = [0, U_1] \times \cdots \times [0, U_n]$, and $D = \{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\}$, determine a time mapping vector $\Pi = [\pi_1, \cdots, \pi_n]$:

```
1. Let $\phi(d_i)$ be the position of the leftmost nonzero entry of $d_i$, for $1 \leq i \leq m$;
   In addition, let $\psi_j$, $1 \leq j \leq g$, be a number between 1 and $n$ so that $\psi_j = \phi(d_i)$ for
   some $d_i \in D$;
   Furthermore, assume $1 \leq \psi_1 < \psi_2 < \cdots < \psi_g \leq n$;
2. $\Pi = [\pi_1, \cdots, \pi_n] = [0, \cdots, 0]$;
3. for $j = g$ downto 1 {
       $k = \psi_j$;
       $\pi_k = \min\{r \in Z_0 \mid r d_{ik} + \sum_{l=k+1}^{n} \pi_l d_{il} > 0,\ \forall\ d_i \in D$ such that $\phi(d_i) = \psi_j\}$;
   }
```

$\square$

It is obvious that the time mapping vector Π computed by the above algorithm satisfies the constraint in Eq. (6.2):

Theorem 6.1 $\Pi \cdot d_i > 0$ *for all* $d_i \in D$*, where* Π *is computed by Algorithm 6.1.*

Proof. Note that for each dependence vector $d_i \in D$ with $\phi(d_i) = j$, we have $d_{ip} = 0$ for $1 \leq p < j$. The theorem follows immediately. $\square$

Let us see an example for Algorithm 6.1 first:

Example 6.1 Consider the regular loop (S, D), where $S = [0, 10] \times [0, 15] \times [0, 20]$, and $D = \{d_1 = [1, 3, -1], d_2 = [1, 2, 1], d_3 = [0, 0, 2]\}$. Then we have:

- $\psi_1 = \phi(d_1) = \phi(d_2) = 1$
- $\psi_2 = \phi(d_3) = 3$

By using Algorithm 6.1, we obtain $\Pi = [\pi_1, \pi_2, \pi_3] = [2, 0, 1]$, as shown below:

$$\begin{array}{rcl} d_1 & = & [\ \boxed{1},\ \ 3,\ \ \text{-}1\ ] \\ d_2 & = & [\ \boxed{1},\ \ 2,\ \ 1\ ] \\ d_3 & = & [\ 0,\ \ 0,\ \ \boxed{2}\ ] \\ \hline \Pi & = & [\ 2,\ \ 0,\ \ 1\ ] \end{array}$$

With this mapping vector, it can be seen that the parallel execution time $T = 2 \times 10 + 0 \times 15 + 1 \times 20 + 1 = 41$. □

In addition, it is easy to see that the above algorithm can be computed in time $O(mn)$, where m is the number of dependence vectors and n is the number of nested levels of the regular loop. In other words, the above algorithm is very efficient since it only takes time linear to the problem size. Recall that computing the optimal time mapping vector Π takes exponential time in practice.

Note that in Algorithm 6.1, dimension k is not considered if no dependent vector $d_i \in D$ so that $\phi(d_i) = k$. By considering those dimensions also, Algorithm 6.1 can be revised to reduce the parallel execution time:

Algorithm 6.2 Given a regular loop (S, D), where $S = [0, U_1] \times \cdots \times [0, U_n]$, and $D = \{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\}$, determine a time mapping vector $\Pi = [\pi_1, \cdots, \pi_n]$:

1. Let $\phi(d_i)$ be the position of the leftmost nonzero entry of d_i, for $1 \leq i \leq m$;

```
   In addition, let $\psi_j$, $1 \le j \le g$, be a number between 1 and $n$ so that
   $\psi_j = \phi(d_i)$ for some $d_i \in D$;
   Furthermore, assume $\psi_1 < \psi_2 < \cdots < \psi_g < \psi_{g+1} = n + 1$;
2. $\Pi = [\pi_1, \cdots, \pi_n] = [0, \cdots, 0]$;
3. for ($j = g$ downto 1) {
    $\alpha = \infty$;
    for ($k = \psi_{j+1} - 1$ downto $\psi_j$) {
      $A = \{s \in Z \mid s d_{ik} + \sum_{l=\psi_j+1}^{n} \pi_l d_{il} > 0,\ \forall\ d_i \in D$ such that $\phi(d_i) = \psi_j\}$;
      if ($A \ne \emptyset$) {
        $\sigma = r$, where $r \in A$ and $|r| \le |s|\ \ \forall s \in A$;
        /* in case of a tie, let $\sigma$ be the positive one */
        if ( $|\sigma U_k| \le \alpha$) {
          $\alpha = |\sigma U_k|$;  $\beta = k$;   $\gamma = \sigma$;
        }
      }
    }
    $\pi_\beta = \gamma$;
   }
```

□

The correctness of the above algorithm is obvious:

Theorem 6.2 $\Pi \cdot d_i > 0$ *for all* $d_i \in D$*, where* Π *is computed by Algorithm 6.2.*

Proof. Note that for each dependence vector $d_i \in D$ with $\phi(d_i) = j$, we have $d_{ip} = 0$ for $1 \le p < j$. The theorem follows immediately. □

Now let us see an example to show how Algorithm 6.2 improves the parallel execution time:

Example 6.2 Consider the regular loop (S, D), where $S = [0, 10] \times [0, 15] \times [0, 20]$, and $D = \{d_1 = [1, 3, -1], d_2 = [1, 2, 1], d_3 = [0, 0, 2]\}$. The time mapping vector computed by Algorithm 6.2 is $\Pi = [\pi_1, \pi_2, \pi_3] = [0, 1, 1]$, as is shown below:

$$\begin{array}{rcl} d_1 &=& [\ \boxed{1},\ \ 3,\ \ \text{-}1\ ] \\ d_2 &=& [\ \boxed{1},\ \ 2,\ \ 1\ ] \\ d_3 &=& [\ 0,\ \ 0,\ \ \boxed{2}\ ] \\ \hline \Pi &=& [\ 0,\ \ 1,\ \ 1\ ] \end{array}$$

Note that the parallel execution time now is $T = [\pi_1, \pi_2, \pi_3] \cdot [U_1, U_2, U_3] + 1 = 0 \times 10 + 1 \times 15 + 1 \times 20 + 1 = 36$ units. This is less than 40 units, which is required if the time mapping vector obtained by Algorithm 6.1 is used. $\square$

Besides the correctness of the algorithm, as well as the efficiency of the mapped loop, it is also easy to see that the computational complexity of Algorithm 6.2 is still linear to the problem size. That is, with the same computational complexity as that of Algorithm 6.1, Algorithm 6.2 can give better parallel execution time of the mapped loop sometimes.

Unfortunately, Algorithm 6.2 does not always give the shorter parallel execution time than Algorithm 6.1 does. This is because what Algorithm 6.2 does is still "local" optimization instead of "global" optimization. Fortunately, Algorithm 6.1 and 6.2 can be combined together so that the computational complexity (of the algorithm) is still linear in the problem size and the parallel execution time (of the mapped loop) is shorter than those derived by either of Algorithm 6.1 or 6.2.

Algorithm 6.3 Given a regular loop (S, D), where $S = [0, U_1] \times \cdots \times [0, U_n]$, and $D = \{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\}$, determine a time mapping vector $\Pi = [\pi_1, \cdots, \pi_n]$:

```
1. Let $\Pi' = [\pi'_1, \cdots, \pi'_n]$ be the time mapping vector computed by Algorithm 6.1;
2. Let $\Pi'' = [\pi''_1, \cdots, \pi''_n]$ be the time mapping vector computed by Algorithm 6.2;
3. if $(\sum_{j=1}^{n} |\pi'_j|\ U_j < \sum_{j=1}^{n} |\pi''_j|\ U_j)$
     $\Pi = \Pi'$;
   else
     $\Pi = \Pi''$;
```

□

It is clear that the time mapping vector computed by Algorithm 6.3 is better than that computed by Algorithm 6.1 or 6.2:

Theorem 6.3 *Given a regular loop (S, D), where $S = [0, U_1] \times \cdots \times [0, U_n]$, and $D = \{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\}$, let the parallel execution times of the regular loop be T_1, T_2, and T_3 respectively when the time mapping vectors are computed by Algorithm 6.1, Algorithm 6.2, and Algorithm 6.3 respectively. Then T_3 is less than or equal to T_1 and T_2.*

Proof. This theorem follows immediately from Algorithm 6.3 and Eq. (6.3). □

6.1.2 An Improved Time Mapping Function

The parallel execution time can be reduced further by modifying the time mapping function. We will first state the key observation, then a new time mapping function, finally a new algorithm to find the associated time mapping vector.

Note that in Example 6.2, we have:

- $\Pi \cdot d_1 = [0, 1, 1] \cdot [1, 3, -1] = 2$
- $\Pi \cdot d_2 = [0, 1, 1] \cdot [1, 2, 1] = 3$

- $\Pi \cdot d_3 = [0,1,1] \cdot [0,0,2] = 2$

That is, the execution time of iteration $X + d_i$, where $i = 1$, 2, or 3, lags the execution time of X by at least 2 time units. However, this 2 time unit delay is not necessary, because the data required by a dependent succeeding iteration has been available in the previous time step. Therefore, the time mapping function can be redefined to be

$$\mathcal{T}(X) = \lfloor \Pi \cdot X/2 \rfloor$$

without violating the dependence constraint.

In general, let the *displacement* of a time mapping vector Π be as follows:

$$disp(\Pi) = \min\{\Pi \cdot d_i \mid d_i \in D\} \tag{6.4}$$

Then the execution time of an iteration lags the execution time of any of its immediately dependent predecessors by at least $disp(\Pi)$ units. Therefore, an iteration $X \in S$ can be executed at time

$$\mathcal{T}(X) = \lfloor \frac{\Pi \cdot X}{disp(\Pi)} \rfloor \tag{6.5}$$

without violating the dependence constraint. This is because for any $X, Y \in S$, if $\Pi \cdot X \geq \Pi \cdot Y + disp(\Pi)$, then we have $\lfloor \frac{\Pi \cdot X}{disp(\Pi)} \rfloor \geq \lfloor \frac{\Pi \cdot Y}{disp(\Pi)} \rfloor + 1$. The parallel execution time of a regular loop with the new time mapping function $\mathcal{T}$ in Equation 6.5 is:

$$\begin{aligned} T &= \max\{|\mathcal{T}(X) - \mathcal{T}(Y)| \mid \forall\ X, Y \in S\} + 1 \\ &= \max\{|\lfloor \frac{\Pi X}{disp(\Pi)} \rfloor - \lfloor \frac{\Pi Y}{disp(\Pi)} \rfloor| \mid \forall\ X, Y \in S\} + 1 \\ &= \begin{cases} |\lfloor \frac{\sum_{j=1}^{n} |\pi_j| U_j}{disp(\Pi)} \rfloor| & \text{or} \\ |\lfloor \frac{\sum_{j=1}^{n} |\pi_j| U_j}{disp(\Pi)} \rfloor| + 1 & \end{cases} \end{aligned} \tag{6.6}$$

To minimize the parallel execution time (6.6) associated with the new time mapping function (6.5), the algorithm for finding the time mapping vector Π should be revised also:

Algorithm 6.4 Given a regular loop (S, D), where $S = [0, U_1] \times \cdots \times [0, U_n]$, and $D = \{d_i = [d_{i1}, \cdots, d_{in}] \mid 1 \leq i \leq m\}$, determine a time mapping vector $\Pi = [\pi_1, \cdots, \pi_n]$:

```
1. Let $\phi(d_i)$ be the position of the leftmost nonzero entry of $d_i$, for $1 \leq i \leq m$;
   In addition, let $\psi_j$, $1 \leq j \leq g$, be a number between 1 and $n$ so that
   $\psi_j = \phi(d_i)$ for some $d_i \in D$;
   Furthermore, assume $\psi_1 < \psi_2 < \cdots < \psi_g < \psi_{g+1} = n + 1$;
2. $\Pi = [\pi_1, \cdots, \pi_n] = [0, \cdots, 0]$;
   $\delta = \infty$;
3. for ($j = g$ downto 1) {
     $\alpha = \infty$;
     $e = \infty$;
     for ($k = \psi_{j+1} - 1$ downto $\psi_j$) {
       $A = \{s \in Z \mid sd_{ik} + \sum_{l=\psi_j+1}^{n} \pi_l d_{il} > 0, \ \forall\ d_i \in D$ such that $\phi(d_i) = \psi_j\}$;
       if ($A \neq \emptyset$) {
         $\sigma = r$, where $r \in A$ and $|r| \leq |s| \ \ \forall s \in A$;
         /* in case of a tie, let $\sigma$ be the positive value */
         $d = \min\{\sigma d_{ik} + \sum_{l=\psi_j+1}^{n} \pi_l d_{il} > 0 \mid \forall\ d_i \in D$ such that $\phi(d_i) = \psi_j\}$;
         if ( $\frac{|\sigma U_k|}{\min\{d,\delta\}} \leq \alpha$) {
           $\alpha = \frac{|\sigma U_k|}{\min\{d,\delta\}}$;   $\beta = k$;   $\gamma = \sigma$;   $e = d$;
         }
       }
     }
     $\pi_\beta = \gamma$;
     $\delta = \min\{e, \delta\}$;
   }
```

□

Note that the value of $disp(\Pi)$ is computed from Π. Hence by using the time mapping vector Π computed by the above algorithm, the time constraint will not be violated:

Theorem 6.4 *If Algorithm 6.4 is used to compute the time mapping vector, then we have $\mathcal{T}(X + d_i) > \mathcal{T}(X)$, where the time mapping function is defined in Eq. (6.5).*

Proof. From the above discussion and the reason in the proof of Theorem 6.1, the theorem follows immediately. □

Besides the correctness of Algorithm 6.4, it is also obvious that the computational complexity of Algorithm 6.4 is linear in the problem size.

6.1.3 Performance Evaluation

In this section, we show that the parallel execution time of a mapped loop by using our method is "very close" to the optimal parallel execution time. The following quantities of a mapped loop are used for our comparison:

- T_1: the parallel execution time of a regular loop by using the time mapping function (6.1) and the time mapping vector obtained by Algorithm 6.3.
- T_2: the parallel execution time of a regular loop by using the time mapping function (6.5) and the time mapping vector obtained by Algorithm 6.4.
- T_{opt}: the optimal parallel execution time of a regular loop by using the time mapping function (6.5) and the "best" time mapping vector.

We compare T_1, T_2, and T_{opt} by using the 25 algorithms listed in [48, page 329]:

Algorithm	1	2	3	4	5	6	7	8	9	10	11	12	13
T_1	10	10	10	28	10	19	55	28	10	19	28	55	28
T_2	4	10	4	14	10	10	55	14	5	19	28	55	28
T_{opt}	4	8	4	13	5	10	55	12	5	19	28	31	28
$\frac{T_1}{T_{opt}}$	2.5	1.25	2.5	2.15	2	1.9	1	2.33	2	1	1	1.77	1
$\frac{T_2}{T_{opt}}$	1	1.25	1	1.08	2	1	1	1.17	1	1	1	1.77	1

Algorithm	14	15	16	17	18	19	20	21	22	23	24	25
T_1	10	19	19	10	37	10	37	13	9	13	17	21
T_2	10	19	19	10	37	10	37	13	9	13	17	21
T_{opt}	8	13	14	7	32	10	32	6	9	7	13	19
$\frac{T_1}{T_{opt}}$	1.25	1.46	1.36	1.43	1.16	1	1.16	2.17	1	1.86	1.31	1.11
$\frac{T_2}{T_{opt}}$	1.25	1.46	1.36	1.43	1.16	1	1.16	2.17	1	1.86	1.31	1.11

From the above table, it can be seen that:

- The average ratio of T_1/T_{opt} is 1.55.
- The worst case ratio of T_1/T_{opt} is 2.50.
- For T_1, 6 out of 25 regular loops can be finished in optimal parallel execution time.
- The average ratio of T_2/T_{opt} is 1.26.
- The worst case ratio of T_2/T_{opt} is 2.17.
- For T_1, 10 out of 25 regular loops can be finished in optimal parallel execution time.

From this experiment, we expect that our methods achieve good performance in practice.

6.2 Finding the Processor Mapping Function

After deciding the time mapping function $\mathcal{T}$ and the time mapping vector Π, we will decide the processor mapping function $\mathcal{P}$ so that if $\mathcal{P}(X) = \mathcal{P}(Y)$, then $\mathcal{T}(X) \neq \mathcal{T}(Y)$, for any two distinct iterations $X, Y \in S$.

We will use the time mapping function (6.1) instead of the time mapping function (6.5). This is because, although Equation 6.5 gives shorter parallel execution time, more processors are needed. Yet in practice, only limited number of processors are available. That is, the *folding* step described in the next section is needed. However, after the folding step, the parallel execution time is lengthened. In other words, we cannot get better parallel execution time because of the limited number of processors. Furthermore, if the time mapping function (6.5) is used, the processor mapping function gets more complicated and the program on each processor (called *node program*) becomes less efficient. Therefore, the time mapping function (6.1) will be used in the following discussion.

As in the previous section, we will first introduce a simple processor mapping function $\mathcal{P}$ and then give a sequence of improvement to this processor mapping function. In addition, we will also evaluate the number of processors required by our method, as well as sketch the node program on each processor.

6.2.1 A Processor Mapping Function for $(n-1)$-dim Processors

Consider a *projection vector* $Q = [q_1, \cdots, q_n]$ which maps each iteration in the n-dimensional space onto an $(n-1)$-dimensional hyperplane. An example projection vector is by letting $Q = \Pi$, which guarantees that all the iterations on the same *time hyperplane* are projected onto different processors [124]. An example of this projection is shown in Figure 6.2. However, if Π is used as a projection vector, we will use more processors than

Figure 6.2: A regular loop with iteration space $[0,9] \times [0,9]$ and time mapping vector $\Pi = [1,1]$.

necessary. We explain this in more detail below.

Some more notation is first introduced:

- $Q = [q_1, \cdots, q_n]$ denotes a projection vector with $\|Q\| = 1$, where $\|Q\|$ represents the length of vector Q.
- $\hat{Q}$ represents an $(n-1)$-dimensional *projection hyperplane* with normal vector Q and containing the origin.
- $I_j = [i_1, \cdots, i_n]$ is a vector where $i_j = 1$ and $i_k = 0$ for every $k \neq j$.
- $X = [x_1, \cdots, x_n]$ is an iteration in S. It is easy to see that $X = \sum_{j=1}^{n} x_j I_j$.

- $\eta_Q(X)$ is the image of X projected through the projection vector Q onto the projection hyperplane $\hat{Q}$. We call $\eta_Q(X)$ a *projected vector.* Mathematically, we have $\eta_Q(X) = X - (X \cdot Q)Q$. Equivalently, we have $\eta_Q(X) = \eta_Q(\sum_{j=1}^{n} x_j I_j) = \sum_{j=1}^{n} x_j \eta_Q(I_j)$.

One of our goals is to determine the projection vector so that the number of processors used is as small as possible. In other words, we want the number of distinct images on the projection hyperplane to be as small as possible. We will show that if the projection vector is chosen as one of the I_j's, $1 \leq j \leq n$, then the number of distinct images is the smallest. We need a lemma first:

Lemma 6.5 *The rank of* $\{\eta_Q(I_1), \cdots, \eta_Q(I_n)\}$ *is* $n-1$.

Proof. Since $\eta_Q(I_1), \cdots, \eta_Q(I_n)$ are vectors on an $(n-1)$-dimensional hyperplane, it is clear that the rank of $\eta_Q(I_1), \cdots, \eta_Q(I_n)$ is at most $n-1$. We show that the rank cannot be smaller than $n-1$.

<u>Case 1:</u> $Q = I_k$, for some $1 \leq k \leq n$.

Without loss of generality, let $Q = I_1$. Then we have:

- $\eta_Q(I_1) = I_1 - (I_1 \cdot I_1)I_1 = 0$
- $\eta_Q(I_j) = I_j - (I_j \cdot I_1)I_1 = I_j$, for $1 < j \leq n$.

Therefore, it is clear that the rank of $\eta_Q(I_1), \cdots, \eta_Q(I_n)$ is $n-1$.

<u>Case 2:</u> $Q \neq I_j$, for all $1 \leq j \leq n$.

Note that the condition of this case implies that $\eta_Q(I_j) \neq 0$ for $1 \leq j \leq n$. Suppose on the contrary that the rank of $\eta_Q(I_1), \cdots, \eta_Q(I_n)$ is less than $n-1$. Without loss of generality, assume that:

- $\eta_Q(I_{n-1}) = \sum_{j=1}^{n-2} \alpha_j \eta_Q(I_j)$

- $\eta_Q(I_n) = \sum_{j=1}^{n-2} \beta_j \eta_Q(I_j)$.

Then we have:

$$\left\{ \begin{array}{l} I_{n-1} - (I_{n-1} \cdot Q)Q = \sum_{j=1}^{n-2} \alpha_j I_j - \sum_{j=1}^{n-2} \alpha_j (I_j \cdot Q)Q \\ I_n - (I_n \cdot Q)Q = \sum_{j=1}^{n-2} \beta_j I_j - \sum_{j=1}^{n-2} \beta_j (I_j \cdot Q)Q \end{array} \right.$$

$$\Rightarrow \left\{ \begin{array}{l} I_{n-1} - \sum_{j=1}^{n-2} \alpha_j I_j = (I_{n-1} \cdot Q)Q - \sum_{j=1}^{n-2} \alpha_j (I_j \cdot Q)Q = aQ \quad \text{where } a \neq 0 \\ I_n - \sum_{j=1}^{n-2} \beta_j I_j = (I_n \cdot Q)Q - \sum_{j=1}^{n-2} \beta_j (I_j \cdot Q)Q = bQ \quad \text{where } b \neq 0 \end{array} \right.$$

$$\Rightarrow \left\{ \begin{array}{l} Q = \frac{1}{a}(I_{n-1} - \sum_{j=1}^{n-2} \alpha_j I_j) \\ Q = \frac{1}{b}(I_n - \sum_{j=1}^{n-2} \beta_j I_j) \end{array} \right.$$

$$\Rightarrow \left\{ \begin{array}{l} q_{n-1} \neq 0, \quad q_n = 0 \\ q_{n-1} = 0, \quad q_n \neq 0 \end{array} \right.$$

This is a contradiction. The Lemma follows immediately. □

Now we count the number of distinct images on the projection hyperplane. Recall that our goal is to minimize the number of distinct images as much as possible. Based on the previous lemma, the lower bound on the number of distinct images is shown below:

Lemma 6.6 *Given a regular loop with iteration space* $[0, U_1] \times \cdots \times [0, U_n]$ *and a projection vector* Q*, the number of distinct images* L *on the projection hyperplane is lower-bounded as below:*

- *If* $Q = I_k$ *for some* $1 \leq k \leq n$*, then* $L = \frac{\prod_{j=1}^{n}(U_j+1)}{U_k+1}$.
- *If* $Q \neq I_j$ *for all* $1 \leq j \leq n$*, then* $L \geq \frac{\prod_{j=1}^{n}(U_j+1)}{U_k+1}$*, where* $U_k = \max\{U_j \mid 1 \leq j \leq k\}$.

Proof. The lemma is clearly true for the first case, so we prove the second case below.

By Lemma 6.5, it can be seen that any $n-1$ vectors in the set $\{\eta_Q(I_1), \cdots, \eta_Q(I_n)\}$ are linearly independent. Without loss of generality, consider a subset A of the iteration space, where $A = \{[x_1, \cdots, x_{n-1}, 0] \mid 0 \leq x_j \leq U_j \text{ for } 1 \leq j \leq n-1\}$. Then for any two distinct iterations $X = [x_1, \cdots, x_n] \in A$ and $Y = [y_1, \cdots, y_n] \in A$, we have $\eta_Q(X) = \sum_{j=1}^{n-1} x_j \eta_Q(I_j)$ and $\eta_Q(Y) = \sum_{j=1}^{n-1} y_j \eta_Q(I_j)$. Since $\eta_Q(I_1), \cdots, \eta_Q(I_{n-1})\}$ are linearly independent, it is clear that $\eta_Q(X) \neq \eta_Q(Y)$. Because the cardinality of A is $\frac{\prod_{j=1}^{n}(U_j+1)}{U_n+1}$, we have $L \geq \frac{\prod_{j=1}^{n}(U_j+1)}{U_n+1}$. By considering *every* U_j as a denominator, the lemma follows immediately. $\square$

Because of Lemma 6.6, we would like to let $Q = I_k$ for some $1 \leq k \leq n$ so that the number of processors used can be minimized. However, the dimension k cannot be chosen arbitrarily, because we still need to follow the constraint that if two iterations are assigned to the same processor, then their execution times must be different. The constraint will not be violated if $Q = I_k$ where $\pi_k \neq 0$:

Lemma 6.7 *Suppose an iteration is projected to the projection hyperplane through the projection vector $Q = I_k$, then any two iterations with the same projected image will be executed at different times if any only if $\pi_k \neq 0$.*

Proof. Consider any two iterations $X = [x_1, \cdots, x_n]$ and $X' = [x'_1, \cdots, x'_n]$, where $x_k \neq x'_k$ and $x_j = x'_j$ for $j \neq k$. It is clear that these two iterations will be mapped to the same image. Furthermore, we can see that their execution times are the same, i.e., $\Pi \cdot X = \Pi \cdot X'$, if and only if $\pi_k = 0$. The lemma follows immediately. $\square$

Based on Lemma 6.6 and Lemma 6.7, we will choose the projection vector $Q = I_k$, where $\pi_k \neq 0$ and $U_k = \max\{U_j \mid 1 \leq j \leq n,\ \pi_j \neq 0\}$. In terms of the processor mapping function, we define:

$$\mathcal{P}(X) = X - (X \cdot Q)Q \tag{6.7}$$

With this processors mapping function, the number of processors used is exactly $\frac{\prod_{j=1}^{n}(U_j+1)}{U_k+1}$.

In the following section, we will define a 1-dim processor mapping function by mapping the processors on the $(n-1)$-dim processor hyperplane onto a 1-dim processor hyperplane.

6.2.2 A Processor Mapping Function for 1-dim Processors

In this section, we will transform the processor mapping function in (6.7) so that the "mapped" processors lie on a one dimensional processor array. Because our iteration space is a rectangular polytope, and our projection vector $Q = I_k$ is in a coordinate direction, linearizing an $(n-1)$-dimensional processor hyperplane into a one dimensional processor array is exactly the same as linearizing array elements by row or column major order in programming languages. Note that this linearizing process does not increase the number of processors used. That is, the number of processors used is still

$$P = \frac{\prod_{j=1}^{n}(U_j + 1)}{U_k + 1} \tag{6.8}$$

where k is the dimension where $\pi_k \neq 0$ and $U_k = \max\{U_j \mid 1 \leq j \leq n,\ \pi_j \neq 0\}$.

For clearness of our following discussion, the following notations are used:

- $V_k = 1$
- $V_j = U_j + 1$, for $1 \leq j \leq n$ and $j \neq k$

Then for a given iteration $X = [x_1, \cdots, x_n] \in S$, it will be executed on processor

$$\mathcal{P}(X) = H \cdot X = [h_1, \cdots, h_n] \cdot [x_1, \cdots, x_n] \tag{6.9}$$

where

$$h_j = \begin{cases} 0 & \text{if } j = k \\ \prod_{l=j+1}^{n} V_j & \text{otherwise} \end{cases} \tag{6.10}$$

We show that the processor mapping function $\mathcal{P}$ defined in this way satisfies the condition that if $\mathcal{P}(X) = \mathcal{P}(Y)$, then $\mathcal{T}(X) \neq \mathcal{T}(Y)$, for any two distinct iterations X and Y in S.

Theorem 6.8 *Let $\mathcal{P}$ and $\mathcal{T}$ be the function defined in (6.9) and (6.1) respectively. We have if $\mathcal{P}(X) = \mathcal{P}(Y)$, then $\mathcal{T}(X) \neq \mathcal{T}(Y)$, for any two distinct iterations X and Y in S.*

Proof. To prove the theorem, we prove that if $H \cdot X = H \cdot Y$, then $\Pi \cdot X \neq \Pi \cdot Y$. Without loss of generality, suppose the n dimensions are reordered so that $k = n$. That is, assume that $U_n = \max\{U_j \mid \pi_j \neq 0, 1 \leq j \leq n\}$.

Let $X = [x_1, \cdots, x_n]$ and $Y = [y_1, \cdots, y_n]$. Suppose $H \cdot X = H \cdot Y$. Then we must have $x_1 = y_1$, because $|\sum_{j=2}^{n} h_j x_j - \sum_{j=2}^{n} h_j y_j| = |\sum_{j=2}^{n} h_j(x_j - y_j)| \leq \sum_{j=2}^{n-1} h_j(U_j - 0) = \sum_{j=2}^{n-1}(V_j - 1)\prod_{l=j+1}^{n-1} V_l = \sum_{j=2}^{n-1}\prod_{l=j}^{n-1} V_l - \sum_{j=2}^{n-1}\prod_{l=j+1}^{n-1} V_l = \prod_{j=2}^{n-1} V_j - 1 < \prod_{j=2}^{n-1} V_j = h_1$. Similarly, we have $x_j = y_j$, for $1 \leq j \leq n-1$. Because $X \neq Y$, then we must have $x_n \neq y_n$. Since $\pi_n \neq 0$ by our assumption, we have $\Pi \cdot X = \prod_{j=1}^{n-1} \pi_j x_j + \pi_n x_n \neq \prod_{j=1}^{n-1} \pi_j y_j + \pi_n y_n = \Pi \cdot Y$. This completes our proof. $\square$

6.2.3 An Improved Processor Mapping Function for 1-dim Processors

When the time mapping vector $\Pi = [\pi_1, \cdots, \pi_n]$ contains more than one nonzero components, with at least one component greater than 1 or less than -1, then we can map more iterations onto one processor so that the total number of processors used can be further reduced. For clearness of our discussion, we will assume $\pi_j \geq 0$, for $1 \leq j \leq n$. Our discussion below can be extended to the general case easily.

Given a significant dimension r where $\pi_r = 1$, and another significant dimension k where $\pi_k > 1$, let G_i denote a group containing all the iteration $[x_1, \cdots, x_n]$, where

- $0 \leq x_k \leq U_k$. (Dimension k is the projection direction, i.e., $Q = I_k$.)

- $x_r = i$. (Algorithms 6.3 (for finding a time mapping vector Π) guarantees that there exists at least a dimension r such that $|\pi_r| = 1$.)

- $x_j = i_j$, where i_j is a constant between 0 and U_j.

Note that all the iterations in the same group are executed on the same processor because $Q = I_k$. Furthermore, let

$$\sigma = \sum_{\substack{1 \le j \le n \\ j \ne k,\ j \ne r}} \pi_j i_j$$

Then the $U_k + 1$ iterations in group G_i will be executed at time $\sigma + i, \sigma + i + \pi_k, \sigma + i + 2\pi_k, \cdots, \sigma + i + U_k\pi_k$ respectively. Therefore, it can be seen that the execution times of group $G_i, G_{i+1}, \cdots, G_{i+\pi_k-1}$ are interleaved:

$$\begin{array}{rllll} G_i: & \sigma + i & \sigma + i + \pi_k & \cdots & \sigma + i + U_k\pi_k \\ G_{i+1}: & \sigma + i + 1 & \sigma + i + 1 + \pi_k & \cdots & \sigma + i + 1 + U_k\pi_k \\ \vdots & \vdots & \vdots & & \vdots \\ G_{i+\pi_k-1}: & \sigma + i + \pi_k - 1 & \sigma + i + \pi_k - 1 + \pi_k & \cdots & \sigma + i + \pi_k - 1 + U_k\pi_k \end{array}$$

Based on the above observation, we can map several groups onto one processor as follows:

$$\underbrace{G_0\ G_1\ \cdots G_{\pi_k-1}}\ \ \underbrace{G_{\pi_k}\ G_{\pi_k+1}\ \cdots G_{2\pi_k-1}}\ \ \cdots\ \ \underbrace{G_{\lfloor U_r/\pi_k \rfloor \pi_k}\ G_{\lfloor U_r/\pi_k \rfloor \pi_k+1}\ \cdots G_{U_r}}$$

With this grouping, it can be seen that the number of processors used is:

$$P = \frac{\prod_{j=1}^{n}(U_j + 1)\left\lceil \frac{U_r+1}{\pi_k} \right\rceil}{(U_k + 1)(U_r + 1)} \tag{6.11}$$

If we neglect the small rounding error of the ceiling function, then minimizing the number of processors used is equivalent to letting dimension k be the one where $\pi_k U_k = \max\{\pi_j U_j \mid \pi_j > 1,\ 1 \le j \le n\}$. Note that this can be computed in linear time.

With this grouping, the processor mapping function $\mathcal{P}$ needs to be redefined. Again for clarity of the processor mapping function $\mathcal{P}$, we use the following notation:

- $V_k = 1$
- $V_r = \lceil (U_r + 1)/\pi_k \rceil$
- $V_j = U_j + 1$, for $1 \leq j \leq n$, $j \neq k$, and $j \neq r$

Then for a given iteration $X = [x_1, \cdots, x_n] \in S$, it will be executed on processor

$$\mathcal{P}(X) = H \cdot X = [h_1, \cdots, h_n] \cdot [x_1, \cdots, x_{r-1}, \left\lfloor \frac{x_r}{\pi_k} \right\rfloor, x_{r+1}, \cdots, x_n] \tag{6.12}$$

where

$$h_j = \begin{cases} 0 & \text{if } j = k \\ \prod_{l=j+1}^{n} V_j & \text{otherwise} \end{cases} \tag{6.13}$$

Now we prove that no two iterations will be executed at the same time on the same processor:

Theorem 6.9 *If iteration X is executed at time $\mathcal{T}(X)$ (Eq. (6.1)) on processor $\mathcal{P}(X)$ (Eq. (6.12)), then no two iterations will be executed at the same time on the same processor.*

Proof. Consider two iteration $X = [x_1, \cdots, x_n]$ and $Y = [y_1, \cdots, y_n]$ in S, and $X \neq Y$. Suppose $\mathcal{P}(X) = \mathcal{P}(Y)$, we prove that $\mathcal{T}(X) \neq \mathcal{T}(Y)$, i.e., $\Pi \cdot X \neq \Pi \cdot Y$.

Without loss of generality, suppose the n dimensions are reordered so that $r = n - 1$ and $k = n$. That is, assume that $\pi_{n-1} = 1$ and $U_n = \max\{\pi_j U_j \mid \pi_j > 1,\ 1 \leq j \leq n\}$.

By the discussion similar to that in Theorem 6.8, we have:

- $x_j = y_j$, for $1 \leq j \leq n - 2$.
- $\left\lfloor \frac{x_{n-1}}{\pi_n} \right\rfloor = \left\lfloor \frac{y_{n-1}}{\pi_n} \right\rfloor$.
- If $x_{n-1} = y_{n-1}$, then $x_n \neq y_n$.

Now consider $\Pi \cdot X$ and $\Pi \cdot Y$. Since $x_j = y_j$, for $1 \leq j \leq n-2$, we only need to prove that $\pi_{n-1}x_{n-1} + \pi_n x_n \neq \pi_{n-1}y_{n-1} + \pi_n y_n$. In other words, we only need to prove that $\pi_{n-1}x_{n-1} - \pi_{n-1}y_{n-1} \neq \pi_n y_n - \pi_n x_n$:

Case 1 **($x_{n-1} = y_{n-1}$):** trivial.

Case 2 **($x_{n-1} \neq y_{n-1}$):** From the fact that $\left\lfloor \frac{x_{n-1}}{\pi_n} \right\rfloor = \left\lfloor \frac{y_{n-1}}{\pi_n} \right\rfloor$, and the fact that $\pi_{n-1} = 1$, we know that $\pi_{n-1}x_{n-1}$ and $\pi_{n-1}y_{n-1}$ can differ by no more than $\pi_n - 1$. Moreover, if $\pi_n x_n$ is different from $\pi_n y_n$, then the difference must be an integer multiple of π_n.

Therefore, it is clear that we cannot have $\pi_{n-1}x_{n-1} - \pi_{n-1}y_{n-1} = \pi_n y_n - \pi_n x_n$. This completes our proof. $\square$

6.2.4 Performance Evaluation

In this section, we show that the number of processors required by our method is "very close" to the minimal number of processors actually required. We assume that the time mapping vector Π has been computed by Algorithm 6.3.

The quantities compared are:

- P_1: the number of processor required by using the processor mapping function (6.12) and the processor mapping vector H determined in the previous section.

- P_{opt}: the number of minimal processors required. That is, suppose c_k is the cardinality of the set $\{X \mid \forall\ X \in S \text{ such that } \Pi X = k\}$. Then T_{opt} is the maximal value of c_k's for all possible values of k.

By experimenting on the 25 regular loops listed in [48, page 329], we obtain the following table:

Algorithm	1	2	3	4	5	6	7	8	9	10	11	12	13
P_1	10	10	10	5	10	10	2	5	10	10	50	30	50
P_{opt}	10	10	10	5	10	10	2	5	10	10	50	26	50
$\frac{P_1}{P_{opt}}$	1	1	1	1	1	1	1	1	1	1	1	1.15	1

Algorithm	14	15	16	17	18	19	20	21	22	23	24	25
P_1	100	100	100	100	40	100	50	75	125	125	75	75
P_{opt}	100	100	100	100	40	100	50	75	125	95	65	65
$\frac{P_1}{P_{opt}}$	1	1	1	1	1	1	1	1	1	1.32	1.15	1.15

Note that for 21 out of 25 loops, the number of processors required by our method is the minimal number of processors. Even in the worst case, the extra processors used are only 32% more. The average value of P_1/P_{opt} for these 25 loops is 1.03. Therefore, we expect that in practice, this processor mapping function, computable in linear time, is efficient in using processors.

6.2.5 Node Program

In this section, we will describe how the mapped regular loop is executed on each processor, according to the time mapping function (6.1) and the processor mapping function (6.12). We call the program executed on a processor as a *node program*. For ease of discussion, we assume that $\pi_j \geq 0$, for $1 \leq j \leq n$. The node program described below can be easily modified when we have $\pi_j < 0$ for some j's.

Consider the node program on a processor with identifier p_{id}. In the node program, we first need to identify the iterations to be executed, and then execute those iterations sequentially. In addition, in the node program, we also have to accomplish the communication among interdependent iterations on distinct processors. In the following, we will address these issues in this order. In addition, we assume that dimension k and r

are the two significant dimensions used by the processor mapping function (please see Section 6.2.3).

<u>Identify the iterations to be executed:</u>

By Eq. (6.12), an iteration $X = [x_1, \cdots, x_n]$ will be executed on processor p_{id} if and only if

$$\mathcal{P}(X) = H \cdot X = [h_1, \cdots, h_n] \cdot [x_1, \cdots, x_{r-1}, \left\lfloor \frac{x_r}{\pi_k} \right\rfloor, x_{r+1}, \cdots, x_n] = p_{id}$$

For convenience, let $[x_1, \cdots, x_{r-1}, \left\lfloor \frac{x_r}{\pi_k} \right\rfloor, x_{r+1}, \cdots, x_n] = [a_1, \cdots, a_n]$. By the construction of vector $H = [h_1, \cdots, h_n]$, it is easy to see that:

- $0 \leq x_k = a_k \leq U_k$
- $a_j = \left\lfloor \frac{p_{id} - \sum_{i=1}^{j-1} h_i a_i}{h_j} \right\rfloor$, for $1 \leq j \leq n$ and $j \neq k$.
- $x_j = a_j$ for $1 \leq j \leq n$, $j \neq k$, and $j \neq r$.
- $\pi_k a_r \leq x_r \leq \pi_k a_r + \pi_k - 1$.

By using this solution, the iterations to be executed on processor p_{id} can be identified (by a node program) in linear time.

<u>Execute the iterations:</u>

After identifying the iterations to be executed, the node program will execute these iteration in the order stipulated by the time mapping function Eq. (6.1). Note that for all the iterations $X = [x_1, \cdots, x_n]$ to be executed by the processor p_{id}, the induction variables x_j, for $1 \leq j \leq n$, are fixed constants a_j, except that $0 \leq x_k \leq U_k$ and $\pi_k a_r \leq x_r \leq \pi_k a_r + \pi_k - 1$. Since $\pi_r = 1$, these iterations will be executed from time $t_1 = \sum_{\substack{1 \leq j \leq n \\ j \neq k,\ j \neq r}} \pi_j a_j + \pi_r(\pi_k a_r)$ to time $t_2 = \sum_{\substack{1 \leq j \leq n \\ j \neq k,\ j \neq r}} \pi_j a_j + \pi_r(\pi_k a_r + \pi_k - 1) + \pi_k U_k$, with one iteration per time unit. Note that $t_2 - t_1 + 1 = \pi_k(U_k + 1)$, the total number of iterations executed by a single processor. The node program is sketched below:

```
1. wait for $t_1$ unit time;
2. let $x_j = a_j$, for $1 \leq j \leq n$, $j \neq k$, and $j \neq r$;
3. for $(x_k = 0$ to $U_k)$
4.     for $(x_r = \pi_k a_r$ to $\pi_k a_r + \pi_k - 1)$ {
5.         execute iteration $[x_1, \cdots, x_n]$;
6.         for (each $d_i = [d_{i1}, \cdots, d_{in}] \in D$)
               send data to the processor containing iteration $[x_1 + d_{i1}, \cdots, x_n + d_{in}]$;
       }
```

Note that the above node program will not be very succinct if the projection vector is not set to be one of the I_j, $1 \leq j \leq n$. For example, the node program in [124] will be much more complicated, because the time mapping vector Π is used as a projection vector.

Accomplish the communication:

Step 6 of the above node program involves sending a message from processor $\mathcal{P}(X)$ to processor $\mathcal{P}(X + d_i)$, where $X \in S$ and $d_i \in D$. We show that the *shift distance* of the message is independent of the locations of the iterations:

Lemma 6.10 *Given a dependence vector $d_i = [d_{i1}, \cdots, d_{in}]$ and iterations X, $X + d_i \in S$, we have $\mathcal{P}(X + d_i) - \mathcal{P}(X) = \sum_{\substack{1 \leq j \leq n \\ j \neq k,\ j \neq r}} h_j d_{ij} + h_r d'_{ir}$, where $d'_{ir} = \lfloor d_{ir}/\pi_k \rfloor$ or $\lceil d_{ir}/\pi_k \rceil$. (Recall that $h_k = 0$.)*

Proof. Let $X = [x_1, \cdots, x_n]$. Then the difference between $\mathcal{P}(X + d_i)$ and $\mathcal{P}(X)$ can be computed as follows:

$$\begin{aligned}
\mathcal{P}(X + d_i) - \mathcal{P}(X) &= \Big(\sum_{\substack{1 \leq j \leq n \\ j \neq k,\ j \neq r}} h_j (x_j + d_{ij}) + h_r \left\lfloor \frac{x_r + d_{ir}}{\pi_k} \right\rfloor \Big) - \Big(\sum_{\substack{1 \leq j \leq n \\ j \neq k,\ j \neq r}} h_j x_j + h_r \left\lfloor \frac{x_r}{\pi_k} \right\rfloor \Big) \\
&= \sum_{\substack{1 \leq j \leq n \\ j \neq k,\ j \neq r}} h_j d_{ij} + h_r \left(\left\lfloor \frac{x_r + d_{ir}}{\pi_k} \right\rfloor - \left\lfloor \frac{x_r}{\pi_k} \right\rfloor \right)
\end{aligned}$$

Let $d'_{ir} = \left\lfloor \frac{x_r + d_{ir}}{\pi_k} \right\rfloor - \left\lfloor \frac{x_r}{\pi_k} \right\rfloor$, then we have:

Case 1: If $\frac{d_{ir}}{\pi_k} = c \in Z$, then $d'_{ir} = (\ \left\lfloor \frac{x_r}{\pi_k} \right\rfloor + c\ ) - \left\lfloor \frac{x_r}{\pi_k} \right\rfloor = c = \left\lfloor \frac{x_r}{\pi_k} \right\rfloor = \left\lceil \frac{x_r}{\pi_k} \right\rceil$.

Case 2: If $\frac{d_{ir}}{\pi_k} = c + \frac{a}{\pi_k}$, where $a, c \in Z$ and $0 < a < \pi_k$:

- When $b \leq \frac{x_r}{\pi_k} < b + \frac{\pi_k - a}{\pi_k}$, where $b \in Z$, we have $d'_{ir} = (b + c) - b = c = \left\lfloor \frac{d_{ir}}{\pi_k} \right\rfloor$.
- When $b + \frac{\pi_k - a}{\pi_k} \leq \frac{x_r}{\pi_k} < b + 1$, where $b \in Z$, we have $d'_{ir} = (b + c + 1) - b = c + 1 = \left\lceil \frac{d_{ir}}{\pi_k} \right\rceil$.

This completes our proof. $\square$

Given a dependence vector $d_i \in D$, let $d(d_i) = \sum_{\substack{1 \leq j \leq n \\ j \neq k,\ j \neq r}} h_j d_{ij} + h_r \lfloor d_{ir}/\pi_k \rfloor$. Then for every iteration $X \in S$, we have:

- If $\frac{d_{ir}}{\pi_k} \in Z$, then $\mathcal{P}(X + d_i) - \mathcal{P}(X) = d(d_i)$.
- If $\frac{d_{ir}}{\pi_k} \notin Z$, then $\mathcal{P}(X + d_i) - \mathcal{P}(X) = d(d_i)$ or $d(d_i) + 1$.

Therefore, if for every iteration X, the data is sent to its dependent successors in the order $X + d_1, X + d_2, \cdots, X + d_m$, then Step 6 in the node program can be coded as follows:

```
6.          for ($i = 1$ to $m$)
              if ($\frac{d_{ir}}{\pi_k} \in Z$) {
6.1             send data to processor $p_{id} + d(d_i)$
6.2             receive data from processor $p_{id} - d(d_i)$
              } else {
6.3             send data to processor $p_{id} + d(d_i)$
6.4             receive data from processor $p_{id} - d(d_i)$, store it in $tmp$
6.5             send $tmp$ to processor $p_{id} + 1$
6.6             receive data from processor $p_{id} - 1$
              }
```

Certainly, if for some dependence vector $d_i \in D$, $d(d_i) = 0$, then no data communication across the processors is needed for that dependence vector. In addition, note that during the execution of a regular loop, each processor has an iteration to execute. Therefore, the communication of data can be accomplished by exploiting the *all-processor shift algorithms*:

- Step 6.1 and 6.2 is with shift distance $d(d_i)$.
- Step 6.3 and 6.4 is with shift distance $d(d_i)$.
- Step 6.5 and 6.6 is with shift distance 1.

Reduce the communication cost:

Some technique can be used to reduce the communication cost. Recall that in Eq. (6.4), we define:

$$disp(\Pi) = \min\{\Pi \cdot d_i \mid d_i \in D\}$$

which means that the execution time of iteration $X + d_i$, for any $d_i \in D$, must be behind the execution time of iteration X by at least $disp(\Pi)$ unit time. That is, an iteration

X can postpone sending data to its dependent iterations $X + d_i$, $d_i \in D$, by at most $disp(\Pi) - 1$ unit time, without violating the time constraint. Since the communication time involves the startup time and the data transmission time, i.e.,

(communication time) = (startup time) + (data transmission time per byte) × (# of bytes)

we can reduce the overall communication time by increasing the number of bytes per communication. To do this, the node program is changed so that the data is sent out at every $disp(\Pi)$ unit time. That is, a *time_counter* will be used to control whether or not to execute Step 6:

```
6.    if $((time\_counter + 1)/disp(\Pi) \in Z)$
        for $(i = 1$ to $m)$ {
            ...
        }
```

6.3 Folding Virtual Processors

Since the number of available processors (called *real processors*) is usually less than the number of processors (called *virtual processors*) required by the method in previous section, we need to *fold* the virtual processors so that they fit into the real processors. Suppose the number of real processors is P_r. Then we map virtual processor $\mathcal{P}(X)$, where $X \in S$, to the real processor $\mathcal{P}_f(X)$ as follows:

$$\mathcal{P}_f(X) = \mathcal{P}(X) \pmod{P_r} \tag{6.14}$$

After the processor mapping function is changed, the time mapping function $\mathcal{T}$ needs be changed also to $\mathcal{T}_f$ so that the time and processor constraints are not violated:

Time Constraint: $\mathcal{T}_f(X + d_i) > \mathcal{T}_f(X)$, for all $X \in S$ and $d_i \in D$.

Processor Constraint: If $\mathcal{P}_f(X) = \mathcal{P}_f(Y)$, then $\mathcal{T}_f(X) \neq \mathcal{T}_f(Y)$, for all $X, Y \in S$.

There are two ways to define the new time mapping function $\mathcal{T}_f$, that is, the (*virtual*) *processor major* folding method and the *time major* folding method. After the mapping functions $\mathcal{T}$ and $\mathcal{P}$ have been determined, the execution of a regular loop can be represented as a (virtual) processor–time plane, as shown in Figure 6.3. In the following, we will describe these two folding methods.

The (virtual) processor major folding method is by (1) cutting the plane into vertical slates, where each slate is with width P_r, and (2) stacking the slates vertically. The graph representation of this method is shown in Figure 6.3(a). It is not hard to see that the new time mapping function $\mathcal{T}_f$ is:

$$\mathcal{T}_f(X) = \lfloor \mathcal{P}(X)/P_r \rfloor T_v + \mathcal{T}(X) \tag{6.15}$$

where T_v is the virtual parallel execution time, i.e., $T_v = \sum_{j=1}^{n} |\pi_j| \; U_j + 1$.

By using this method, it is clear that the Processor Constraint is not violated. However, the Time Constraint may be violated, if, for example, an iteration X in slate 2 needs to send data to an iteration $X + d_i$ in slate 1. Therefore, to follow the Time Constraint, we must have:

$$\mathcal{P}(X + d_i) \geq \mathcal{P}(X) \qquad \forall\ X \in S \text{ and } \forall\ d_i \in D$$

In other words, if the condition that $\mathcal{P}(d_i) \geq 0$ is satisfied, then the time constraint will be obeyed and hence the processor major folding method can be used. Note that the n dimensions can be reordered sometimes so that the condition $\mathcal{P}(d_i) \geq 0$ will be satisfied.

Alternatively, the new time mapping function $\mathcal{T}_f$ can be defined by using the time major folding method. The method is by (1) cutting the processor–time plane into bricks, where each brick is with width P_r and height no more than $disp(\Pi)$, (2) stacking the bricks in time (row) major order. The graph representation of this method is shown in

Figure 6.3: (a) (Virtual) processor major folding method. (b) Time major folding method.

Figure 6.3(b). It is not hard to see that the new time mapping function $\mathcal{T}_f$ is:

$$\mathcal{T}_f(X) = \mathcal{T}(X)\lceil P_v/P_r\rceil + \lfloor \mathcal{P}(X)/P_r\rfloor \qquad (6.16)$$

where P_v is the number of virtual processors in (6.11). With this new time mapping function, it is also easy to see that both the time constraint and the processor constraint are obeyed.

The two folding methods mentioned above are *work preserving* transformations, that is, we have $P_vT_v = P_rT_r$, where P_v, P_r, and T_v are defined above, and T_r is the parallel execution time on the *real* processors. However, with some minor modification, these transformations can reduce the amount of work sometimes. The key point is that the iterations on the processor–time plane may not necessarily constitute a rectangle. Therefore, the "stacking" of slates or bricks can be more compact sometimes. Figure 6.4 illustrates the idea behind the processor major folding method. The new time mapping function can be easily redefined to accomplish such a modification, hence we omit the detail here.

In addition, the node programs need be modified also to accomplish the folding step. Since the virtual processors and the real processors are related by a module function (Eq 6.14), the modification is quite simple.

Moreover, recall that in the virtual processor setting, an all-processor shift algorithm is needed. Suppose the shift distance is d. Then after the folding step, the algorithm needs to be used is the all-node *cyclic shift* algorithm, with shift distance $d \pmod{P_r}$, where P_r is the number of real processors. As a concrete example of how the cyclic shift algorithm works, the next chapter is devoted to the cyclic shift algorithms on hypercube machines.

As a summary, given a regular loop, we map it onto a distributed-memory MIMD machine in three steps as follows:

Figure 6.4: (a) (Virtual) processor–time plane. The iterations constitute the shadow region. (b) Processor major folding method without compression. (b) Processor major folding method with compression.

1. compute the time mapping function,
2. compute the (virtual) processor mapping function, and
3. fold the virtual processors into real processors.

In addition, the mapped loop exploits cyclic shift operations to accomplish the data communication, which is the only part dependent on the interconnection structure. That is, *on a new machine, we only need to implement a new cyclic shift algorithm, and the three steps in our mapping strategy can be easily implemented by using existing code, since the above three steps are machine independent.*

Chapter 7

Very Efficient Cyclic Shifts on Hypercubes

Our mapping strategy in last chapter utilizes cyclic shift operations to achieve data communications among processors. To get a better insight of the cyclic shift operation, we devote this chapter to the cyclic shift algorithms for hypercube machines. Besides being used by our mapping strategy, cyclic shifts are also intrinsic operations in many parallel algorithms. Therefore, it is important to execute them efficiently. In this chapter, we present and analyze algorithms for the *cyclic shift* operation on n-dimensional (distributed memory) *hypercubes*.

This chapter is organized as follows. Section 7.1 introduces previous work and briefly summarizes our contribution. Section 7.2 formally defines the model and the cyclic shift problem. Section 7.3 presents and analyzes the Shortest Path Algorithm, which always uses the minimal number of links to route a message. Section 7.4 shows the Disjoint Link Algorithm, which always guarantees that all of the routing paths are link disjoint. Finally, Section 7.5 compares the merits of various cyclic shift algorithms.

7.1 Introduction

The cyclic shift algorithm is interesting because, besides being used by our mapping strategy (in Chapter 6), the cyclic shift (or rotation) operation is also used by many algorithms (such as parallel matrix multiplication) in a natural way as a primitive step. This has been recognized in a variety of parallel computational settings [68] [73] [137]. As noted by Johnsson [68], "many linear algebra algorithms can be formulated using rotations as an operator on aggregate data structures". Consequently, a "better" cyclic shift algorithm is instrumental in improving the performance of such problems. Informally, a cyclic shift operation is that by arranging all the nodes in an n-dimensional hypercubes into a linear array, every node will send a message to the node at distance d apart away cyclically. Formal definition of the cyclic shift operation will be introduced in Section 7.2. In this chapter, we present *provably* efficient (sometimes optimal) algorithms for realizing cyclic shifts on distributed memory hypercubes, which are popular commercially and in research [60] [109] [120] [144].

Typically, an n-dimensional hypercube consists of 2^n processors (nodes), interconnected in a hypercube topology [60] . Because of its abundant communication bandwidth, hypercube attracts many attentions: its properties and routing schemes are extensively explored [2] [30] [55] [59] [67] [68] [72] [95] [101] [113], abundant algorithms are designed for it [17] [23] [39] [49] [65] [79] [88] [92] [97], and embedding various data structures, such as tree [19] [40] [85] [138], mesh [24] [25] [26] [54] [63] [64], and others [28] [31] [32] [34] [43] [62] [77] [136], into hypercube topology is widely studied.

An array of data or processes can be embedded onto hypercube nodes either in natural order or in Binary-Reflect Grey Code (BRGC) order. Consider an array $a[0..N-1]$ to be embedded onto an n-dimensional hypercube, where $N = 2^n$. The natural order embedding assigns $a[i]$ to hypercube node i, while the BRGC order embedding assigns

$a[i]$ to hypercube node j, where j is the BRGC representation of i. An array embedded in one order can be transformed into the other order in $n-1$ communication steps [68]. In addition, these two embedding schemes have the following important properties: for natural order embedding, $a[i]$ and $a[i+2^k]$ are assigned to adjacent hypercube nodes, where $i \mathrm{and} 2^k = 0$ and $0 \leq i, i+2^k < N$; while for the BRGC order embedding, $a[i]$ and $a[i+1 \pmod{N}]$ are assigned to adjacent hypercube nodes. Depending on applications and algorithms, one of these embedding schemes is adopted over the other. For example, the natural order embedding is used by algorithms for bitonic merge [93] and convolution [108], while the BRGC order embedding is used by algorithms for matrix manipulations [17] and fast Fourier transforms [23].

On an n-dimensional hypercube where an array is embedded in the natural order, a cyclic shift operation can be performed in n communication steps, with the property that all of the n routing paths are link disjoint [108]. However, no cyclic shift algorithm is known on an n-dimensional hypercube when an array is embedded in the BRGC order. In this chapter, we concentrate on the cyclic shift algorithms for the hypercubes where arrays are embedded in the BRGC order. Note that a cyclic shift algorithm for the BRGC order embedding can be designed by composing the "embedding order transformation algorithms" and the "cyclic shift algorithms for the natural order embedding". However, this composed algorithm is neither efficient ($3n-2$ communication steps are required), nor does it maintain the link disjoint property.

Historically, Johnsson presented two algorithms (henceforth referred to as *J1* and *J2* respectively) for the *cyclic shift* operation, to which he refers as the *rotation* operation. As noted in [68], to route a message from a source node to a sink node on an n-dimensional hypercube, Algorithm *J1* uses at most $2n$ hypercube links, while Algorithm *J2* uses at most n hypercube links. (Throughout this chapter, we will be restricting our attention to routing patterns that are generated by cyclic shifts only.) The intuition behind motivating Algorithm *J1*, even though it is obviously less efficient than *J2* is as follows: *using more communication links reduces the "possibility" that two routing paths will contend for the same hypercube link, thereby reduces the need for local buffering (memory) to route the conflicting messages.* Therefore, the hope is that link congestion can be *traded-off* against more communication steps. However, there is no analysis to substantiate this intuition.

By studying Algorithm *J1* and *J2* carefully, we discover surprisingly that Algorithm *J2* is both faster as well as has negligible congestion. In particular, the amount of congestion that it introduces is never worse than that introduced by Algorithm *J1.* this is contrary to the intuition conveyed by the previous work about Algorithm *J1.* To better understand this, let us refine hypercube architectures into the *synchronous* and *asynchronous* communication cases. This classification is central to the analysis of the congestion, and hence the size of local buffers needed to realize cyclic shifts.

Specifically, we show that on synchronous[1] hypercubes, Algorithm *J2* guarantees that a link is *never* used by two different routing paths during the same communication step (Theorem 7.3). Therefore, Algorithm *J2* can always be used in place of Algorithm *J1* on synchronous hypercubes, with twice the speed. Furthermore, since Algorithm *J2* uses no more than n links for a routing path, it is trivially optimal.

On asynchronous hypercubes, we show that Algorithm *J2* guarantees that any link is used by *at most* two different routing paths (Theorem 7.6). This implies that at most *one word* of local buffer per-link is sufficient, since on any link, at most two messages can conflict during the cyclic shift operation. Note that this leads to a worst-case slowdown of one unit at every link on a routing path for any message. Therefore, even in the worst-case Algorithm *J2* can route messages between hypercube nodes in $2n-2$ steps; this bound is

[1]All the communication and computation steps proceed in *lock-step.*

tight. Furthermore, in this setting, we can show (trivially) that Algorithm *J1* also results in link congestion, and is thus strictly worse than Algorithm *J2* in the asynchronous case as well.

We note that in a completely asynchronous setting, *any* algorithm that shares links across paths has the drawback that messages from different paths are likely to contend for the same link at the same time. Therefore, local buffers are necessary in this case. This need for local buffers can be circumvented if we restrict our attention to message routing schemes that do not share links for *any* shift distance. We present such an algorithm, referred to as the *disjoint link algorithm* (Theorem 7.17). Moreover, we show that on any n-dimensional hypercube, and given any shift distance, the algorithm uses no more than $\frac{4}{3}n$ links[2] between any pair of hypercube nodes (Theorem 7.18). We note that this is no more than 34% overhead relative to the obvious optimum, namely n.

The problem of general *permutation routing* in hypercubes has a rich history; please see [35] [69] [106] [128] [135] as examples. While these algorithms are all asymptotically efficient (or optimal as the case might be), their analysis relies on the traditional "big-Oh" notation. Such analysis does not explicitly account for the constants that multiply the complexities, which are of significance especially when a given operation such as cyclic shift is used repeatedly by other more complex algorithms. It is our intent here to analyze the complexity of our algorithms by explicitly accounting for these multiplicative constants, for the restricted and yet important special case of permutation routing, namely cyclic shifts.

[2]More precisely, when the shift distance is d, the number of links used is no more than $\frac{4}{3}\lceil \log d' \rceil + 1$, where $d' = d$ if $1 \leq d \leq 2^{n-1}$, and $d' = d - 2^{n-1}$ if $2^{n-1} < d < 2^n$.

7.2 Preliminaries

7.2.1 The Hypercube Architecture

An *n-dimensional hypercube* (or *binary n-cube*) is a directed graph $G = (V, E)$, where V contains 2^n nodes labeled from 0 to $2^n - 1$, and E contains all of the links $v_1 \rightarrow v_2$ where the n-bit binary representations of the labels of v_1 and v_2 differ by exactly one bit. A hypercube *link* is at dimension i if v_1 and v_2 differs at bit i (the least significant bit is at position 0). Two kinds of hypercubes are considered in our study. A hypercube is synchronous if the computation and communication are performed in lock-step. Otherwise, a hypercube is asynchronous. In addition, it is assumed that all of the incident links on a hypercube node can send/receive messages concurrently, as the model used in [69].

7.2.2 Problem Statement

Associated intimately with hypercubes is the *Binary Reflected Gray Code (BRGC)* [111]. An n-bit Binary-Reflected Gray Code (BRGC) G_n is a sequence of 2^n n-bit strings defined recursively as below:

$$\begin{aligned} G_1 &= [0, 1] \\ G_n &= [0G_{n-1}, 1G_{n-1}^R] \end{aligned}$$

where $0G_{n-1}$ ($1G_{n-1}^R$) means a 0(1) is inserted at the head of every string of G_{n-1} (G_{n-1}^R), and G_{n-1}^R is the sequence of strings in reversed order of G_{n-1}. For example, $G_2 = [00, 01, 11, 10]$, and $G_3 = [000, 001, 011, 010, 110, 111, 101, 100]$. In addition, let $G_n(i)$ represent the $(i+1)$st n-bit string in G_n. That is, G_n is the sequence of integers $G_n(0)$, $G_n(1)$, $G_n(2)$, $\cdots$, $G_n(2^n - 1)$.

The cyclic shift problem is defined as follows. In an n-dimensional hypercube, the *cyclic shift* problem is that each node $G_n(i)$ will send a message to node[3] $G_n(i + d \pmod{2^n})$ for all i, $0 \leq i \leq 2^n - 1$, and some fixed d, $1 \leq d \leq 2^n - 1$. Please note that when the shift distance is an integer multiple of 2^n, no message shifts are needed at all; and when the shift distance , say d', is not in the range as specified, we can always let d be $d' \pmod{2^n}$ and get the same results. Recall that the reason for considering routing from $G_n(i)$ to $G_n(i + d \pmod{2^n})$ instead of from i to $i + d \pmod{2^n}$ is that it yields a natural way through BRGC for embedding, which has the advantage that adjacent array elements are allocated at neighboring hypercube nodes.

7.3 The Shortest Path Algorithm

In this section, we will introduce and analyze in detail Johnsson's algorithm *J2*, which is referred to as *the shortest path algorithm* because it always uses the minimal number of links to route a message.

7.3.1 The Algorithm

The shortest path algorithm is described below:

Algorithm 7.1 Given a hypercube of dimension n, a node $G_n(i)$, and a shift distance d, where $0 < d < N = 2^n$, the following algorithm routes a message from $G_n(i)$ to $G_n(i+d \pmod{2^n})$ by correcting successive bits that differ in $G_n(i)$ and $G_n(i+d \pmod{2^n})$, from the less significant end to the more significant end:

[3]When $b > 0$, define $a \pmod{b} \equiv a - \lfloor \frac{a}{b} \rfloor \cdot b$. Note that $a \pmod{b}$ will never be negative by this definition.

```
let the binary representations of $G_n(i)$ and $G_n(i + d \pmod{2^n})$ be $g^a_{n-1}g^a_{n-2}\cdots g^a_0$
    and $g^b_{n-1}g^b_{n-2}\cdots g^b_0$ respectively;
let $t_1, \ldots, t_k$ be the numbers such that $g^a_{t_j} = \overline{g^b_{t_j}}$ and $t_1 < t_2 < \cdots < t_k$;
    /* $\overline{x}$ means the complement value of bit $x$ */
$g_{n-1}g_{n-2}\cdots g_0 = g^a_{n-1}g^a_{n-2}\cdots g^a_0$;
$path = g_{n-1}g_{n-2}\cdots g_0$;
for $j = 1$ to $k$ do {
    $g_{t_j} = \overline{g_{t_j}}$;
    $path = path$ ▯ $g_{n-1}g_{n-2}\cdots g_0$;    /* "▯" stands for concatenation here. */
}
```

□

The algorithm is correct because any two hypercube nodes are connected if their node labels differ by exactly one bit. In addition, it is mentioned by Johnsson that *each* such path is subject to at most n routing steps, yet a hypercube link may be congested by more than one paths. Johnsson explain this phenomenon by an example which routes messages from $G_3(i)$ to $G_3(i + 3 \pmod 8)$ for $i = 0$ and 1:

0: $000 \rightarrow 010$

1: $001 \rightarrow 000 \rightarrow 010 \rightarrow 110$

Note that in this example, the link $000 \rightarrow 010$ is congested only in the asynchronous communication environment. When considered in the synchronous communication environment, the link $000 \rightarrow 010$ is not a congested link. In the following section, we will prove that algorithm 7.1 yields no hypercube link congestion at any synchronous communication step.

7.3.2 Properties on Synchronous Hypercubes

In this section, an important property (Theorem 7.3) of executing the shortest path algorithm on synchronous hypercubes is presented. Before describing the theorem, Definition 7.1 to 7.3 are introduced first to facilitate the proof of the theorem.

Definition 7.1 For a positive integer d, define $\overline{d} = 2^{\lceil \log_2 d \rceil}$. □

Definition 7.2 A *region* of G_n is a sequence of consecutive BRGC numbers $S \equiv [G_n(a), G_n(a+1), \ldots, G_n(b)]$. Let $f(S) = a$, $l(S) = b$, and $nf(S) = l(S) + 1 \pmod{2^n}$. Then we have:

- $|S| = b - a + 1$ is the *size* of region S.
- S is a *whole region* if $\overline{|S|}$ divides both $f(S)$ and $nf(S)$.

□

Note that G_n contains 2^{n-k} whole regions with size 2^k each. For any two strings in the same whole region, their most significant $n-k$ bits are same. For any two strings in consecutive whole regions, their most significant $n-k$ bits differ by at most one bit.

Definition 7.3 The *wraparound subtraction* around 2^n for $0 \leq as, bs < 2^n$ is defined as follows:

$$bs \underset{n}{\dot{-}} as = \begin{cases} bs - as & \text{if } bs \geq as \\ 2^n - as + bs & \text{otherwise} \end{cases}$$

□

Using these definitions, we first prove the following two lemmas:

Lemma 7.1 *Consider two n-bit strings $G_n(a) = g^a_{n-1}g^a_{n-2}\cdots g^a_0$ and $G_n(b) = g^b_{n-1}g^b_{n-2}\cdots g^b_0$ in G_n. If $g^a_{n-1}g^a_{n-2}\cdots g^a_k = g^b_{n-1}g^b_{n-2}\cdots g^b_k$, for $k < n$, then a and b are in the same whole region of size 2^k.*

Proof.

From the definition of the binary-reflected gray codes, we have $G_{k+1} = \{0G_k, 1G_k^R\}$, $G_{k+2} = \{00G_k, 01G_k^R, 11G_k, 10G_k^R\}$. By simple induction on $k < n$, G_n can be divided into 2^{n-k} whole regions where all of the 2^k strings in each whole region have the same most significant $n-k$ bits, and no two numbers in different whole regions have the same most significant $n-k$ bits. The lemma follows immediately. $\square$

Lemma 7.2 *Consider two n-bit strings $G_n(as) = g_{n-1}^{as} \cdots g_0^{as}$ and $G_n(bs) = g_{n-1}^{bs} \cdots g_0^{bs}$ in G_n. For all $0 < k < n$ and $0 \leq as, bs < 2^n$, if $bs \underset{n}{\div} as < 2^k$ and $g_{k-1}^{as} g_{k-2}^{as} \cdots g_0^{as} = g_{k-1}^{bs} g_{k-2}^{bs} \cdots g_0^{bs}$, then $bs \underset{n}{\div} as$ is odd.*

Proof. Let us divide G_n into 2^{n-k+1} whole regions S_0, S_1, $\cdots$, $S_{2^{n-k+1}-1}$, with each region containing 2^{k-1} strings. Furthermore, let B_i represent the strings in S_i with each string restricted to the least significant k bits. By the definition of binary reflected Gray code, it is not hard to see that $B_0 = 0G_{k-1}, B_1 = 1G_{k-1}^R, B_2 = 1G_{k-1}, B_3 = 0G_{k-1}^R, B_4 = 0G_{k-1}$, and so on. Therefore, from the conditions that $bs \underset{n}{\div} as < 2^k$ and $g_{k-1}^{as} g_{k-2}^{as} \cdots g_0^{as} = g_{k-1}^{bs} g_{k-2}^{bs} \cdots g_0^{bs}$, we know $G_n(as)$ and $G_n(bs)$ must be in region S_{2i-1} and $S_{2i \,(\text{mod } 2^{n-k+1})}$ respectively for some i. Furthermore, from the reflective property between B_{2i-1} and $B_{2i \,(\text{mod } 2^{n-k+1})}$, $bs \underset{n}{\div} as$ must be odd. This completes our proof. $\square$

Now we are ready to present the main property:

Theorem 7.3 *On a synchronous n-dimensional hypercube, any two messages sent from source nodes $G_n(a)$ and $G_n(b)$ to sink nodes $G_n(a + d \,(\text{mod } 2^n))$ and $G_n(b + d \,(\text{mod } 2^n))$ respectively will never reach the same hypercube node at the same time step provided the Shortest Path Algorithm is used.*

Proof. For convenience, let $as \equiv a + d \,(\text{mod } 2^n)$ and $bs \equiv b + d \,(\text{mod } 2^n)$. In addition, let the n-bit binary representations of $G_n(v)$ be $g_{n-1}^v g_{n-2}^v \cdots g_0^v$ for $v = a, b, c, as, bs$ re-

Figure 7.1: Illustration for the proof of Theorem 7.3.

spectively. We prove by contradiction that it is impossible that the message from $G_n(a)$ to $G_n(as)$ and the message from $G_n(b)$ to $G_n(bs)$ will arrive at a common node $G_n(c)$ during the same communication step.

Suppose on the contrary that at time step p, the messages from nodes $G_n(a)$ and $G_n(b)$ arrive at node $G_n(c)$ simultaneously via the hypercube links at dimension ad and bd respectively (please see Figure 7.1). Without loss of generality[4], assume $ad \neq bd$ and let $k - 1 = \max(ad, bd)$.

Now from the routing scheme, we have

1. $g^a_{n-1} \cdots g^a_k = g^c_{n-1} \cdots g^c_k = g^b_{n-1} \cdots g^b_k$.
2. $g^{as}_{k-1} \cdots g^{as}_0 = g^c_{k-1} \cdots g^c_0 = g^{bs}_{k-1} \cdots g^{bs}_0$.

Without loss of generality, let us assume that $0 \leq a < b < 2^n$. From the fact that $g^a_{n-1}g^a_{n-2} \cdots g^a_k = g^b_{n-1}g^b_{n-2} \cdots g^b_k$ and Lemma 7.1, we know that $b - a < 2^k$. Since $b - a = bs \underset{n}{\div} as$, it follows that $bs \underset{n}{\div} as < 2^k$. From this fact and that $g^{as}_{k-1}g^{as}_{k-2} \cdots g^{as}_0 = g^{bs}_{k-1}g^{bs}_{k-2} \cdots g^{bs}_0$, it follows from Lemma 7.2 that $b - a = bs \underset{n}{\div} as$ is odd.

From the construction of BRGC, $b - a$ being odd implies that the Hamming distance between $G_n(a)$ and $G_n(b)$ is odd also. Furthermore, note that the Hamming distance

[4] If $ad = bd$, we can always find an earlier time step p' such that $ad' \neq bd'$.

between two adjacent hypercube nodes is exactly 1. Therefore, it is impossible that the messages sent from $G_n(a)$ and $G_n(b)$ will arrive at a common node $G_n(c)$ at the same time step when the Hamming distance between $G_n(a)$ and $G_n(b)$ is odd. This contradicts to our assumption that messages sent from $G_n(a)$ and $G_n(b)$ will arrive at the same node $G_n(c)$ simultaneously. $\square$

Since from Theorem 7.3, all of the routing paths use distinct nodes at each communication step, it follows immediately that:

Corollary 7.4 *On a synchronous n-dimensional hypercube, two messages sent from source nodes $G_n(a)$ and $G_n(b)$ to sink nodes $G_n(a+d \pmod{2^n})$ and $G_n(b+d \pmod{2^n})$ respectively, will never use the same hypercube link at the same communication step provided the Shortest Path Algorithm is used.*

From this corollary and the fact that the Hamming distance of $G_n(i)$ and $G_n(i+d \pmod{2^n})$, for any $0 \leq i \leq 2^n - 1$, is less than or equal to n, it is clear that the Shortest Path Algorithm takes no more than n synchronous communication steps and hence is optimal for such an environment.

Moreover, we can give closer upper bound on the number of communication steps needed to route a message, as described below:

Theorem 7.5 *On a synchronous n-dimensional hypercube, a message sent from source node $G_n(i)$ to sink node $G_n(i+d \pmod{2^n})$ needs no more than $1+\log \overline{d'}$ communication steps, where $d' = d$ if $0 < d \leq 2^{n-1}$ and $d' = 2^n - d$ if $2^{n-1} < d < 2^n$, provided the Shortest Path Algorithm is used.*

Proof. Since on synchronous hypercubes, no link congestion occurs during a single communication step, it is sufficient to consider the number of links used by a routing path.

Consider the case for $0 < d \leq 2^{n-1}$ first. Let us divide G_n into a sequence of whole regions of size $\overline{d}$ each. Note that:

- for any two strings in the same whole region, their most significant $n - \log \overline{d}$ bits are same and their least significant $\log \overline{d}$ bits differ by at most $\log \overline{d}$ bits;
- for any two strings in contiguous whole regions, their most significant $n - \log \overline{d}$ bits differ by exact 1 bit and their least significant $\log \overline{d}$ bits differ by at most $\log \overline{d}$ bits.

Furthermore, it is clear the $G_n(i)$ and $G_n(i + d \pmod{2^n})$ must be either in the same whole region or in contiguous whole regions. The theorem follows immediately for this case.

Now consider the case for $2^{n-1} < d < 2^n$. We note that $G_n(i + d \pmod{2^n}) = G_n(i - (2^n - d) \pmod{2^n})$. By similar discussion to the previous case, the theorem follows immediately for this case. $\square$

It can be verified that the largest value of $\overline{d'}$ is 2^{n-1}. Hence on a synchronous hypercube, the number of communication steps needed by any routing path cannot exceed n for any shift distance d.

7.3.3 Properties on Asynchronous Hypercubes

Although being optimal on synchronous hypercubes, the Shortest Path Algorithm does use the same hypercube links more than once during the routing. Hence it may need more communication time on asynchronous hypercubes, due to link congestion. Fortunately, we can show that using the Shortest Path Algorithm, the maximum link congestion on an asynchronous hypercube is not "serious", as described in the following theorem:

Theorem 7.6 *When the Shortest Path Algorithm is used to solve the cyclic shift problem on an n-dimensional hypercube, we have:*

1. A link at dimension 0 *or* $n-1$ *is traversed at most once.*

2. A link at dimension k*,* $0 < k < n-1$*, is traversed at most twice.*

Proof. When routing a message from a source node $G_n(s)$ to its cyclic-shifted sink node $G_n(t)$, Algorithm 7.1 works by correcting successive bits that differ in $G_n(s)$ and $G_n(t)$, from the less significant end to the more significant end. Since all of the source nodes are distinct and all of the sink nodes are also distinct, it is clear that any link at dimension 0 or $n-1$ is traversed at most once.

Now consider a link at dimension k, $0 < k < n-1$, which connects node $G_n(u) = g_{n-1}^u \cdots g_0^u$ to node $G_n(v) = g_{n-1}^v \cdots g_0^v$. By the definition of hypercubes, we have $g_i^u = g_i^v$ when $i \neq k$ and $g_k^u = \overline{g_k^v}$. If when routing a message from the source node $G_n(s) = g_{n-1}^s \cdots g_0^s$ to the sink node $G_n(t) = g_{n-1}^t \cdots g_0^t$, the link $G_n(u) \rightarrow G_n(v)$ is traversed, then we have $g_{n-1}^s \cdots g_k^s = g_{n-1}^u \cdots g_k^u$ and $g_k^t \cdots g_0^t = g_k^v \cdots g_k^v$. Now consider a set of source/sink node pairs $G_n(s_i)$ and $G_n(t_i)$, where routing a message from node $G_n(s_i)$ to node $G_n(t_i)$ will traverse through the link $G_n(u) \rightarrow G_n(v)$. Then by Lemma 7.1, all of the $G_n(s_i)$'s must be within a whole region of size 2^k. Furthermore, by discussion similar to that in the proof of Lemma 7.2, no more than two $G_n(t_i)$'s can be in a region of size 2^k. Since the shift distance is the same over all source/sink node pairs, the theorem follows immediately. □

From this theorem, it can be seen that when the Shortest Path Algorithm is used on an asynchronous n-cube, at most $2n-2$ routing steps are required. In addition, it is trivial to give the buffer requirement on asynchronous hypercubes:

Corollary 7.7 *On an asynchronous hypercube, at most one unit message buffer is needed at each hypercube link when the Shortest Path Algorithm is used to solve the cyclic shift problem.*

It is not hard to see that Algorithm *J1* [68] also needs message buffers on asynchronous hypercubes. Furthermore, since Algorithm *J1* will traverse through $2n-1$ links at the worst case on an n-cube, it is strictly worse than the Shortest Path Algorithm.

Finally, the theorem tantamount to Theorem 7.5 for asynchronous hypercubes is given below:

Theorem 7.8 *On an asynchronous n-cube, a message sent from source node $G_n(i)$ to sink node $G_n(i+d \pmod{2^n})$ needs no more than $2\log\overline{d'}$ communication steps, where $d' = d$ if $0 < d \leq 2^{n-1}$ and $d' = 2^n - d$ if $2^{n-1} < d < 2^n$, provided the Shortest Path Algorithm is used.*

Proof. From the proof of Theorem 7.5, it can be observed that routing a message uses at most one link which is "across" the whole region boundary, and at most $\log\overline{d'}$ links which are "within" a whole region. Among these $1+\log\overline{d'}$ links, we have:

- The link across the whole region boundary is used by at most one routing path, since the Shortest Path Algorithm routes a message by using links from the lower dimension end to the higher dimension end.
- The link at dimension 0 within a whole region is used by at most one routing path, because of part 1 of Theorem 7.6.
- Any link at dimension 1 to $\log\overline{d'}-1$ within a whole region is used by at most two routing paths, because of part 2 of Theorem 7.6.

The theorem follows immediately. $\square$

It can be verified that the largest value of $\overline{d'}$ is 2^{n-1}. Hence on an asynchronous hypercube, the number of routing steps needed by any routing path cannot exceed $2n-2$ for any shift distance d.

7.4 The Disjoint Link Algorithm

Although the shortest path algorithm is optimal on synchronous hypercubes, it does need buffers on asynchronous hypercubes. To avoid the buffer requirement, this section describes a cyclic shift algorithm which yields no link congestion.

7.4.1 The Algorithm

In this section, we describe an algorithm that routes a message from source node $G_n(i)$ to sink node $G_n(i + d \pmod{2^n})$, for $0 \leq i < 2^n$ and a fixed d, in such a way that these 2^n routing paths are link-disjoint. This algorithm routes a message usually by correcting successive bits that differ in $G_n(i)$ and $G_n(i + d \pmod{2^n})$, from the more significant end to the less significant end, with the exception that when link congestion might occur, some extra links are chosen to detour around the points of congestion.

Figure 7.2 shows the routing paths generated by the algorithm for a 5-cube with the shift distance equal to 13. Each row in the figure represents one routing path, with the source node at the left and the sink node at the right. Briefly speaking, the algorithm works by reflecting smaller and smaller subregions (enclosed in frames) as units in smaller and smaller whole regions (please refer to Definition 7.2). The link congestion is avoided by the careful choice of the subregions and the dimensions to be reflected over.

Before describing the algorithm, we need some definitions (Please refer to Definition 7.2 for the notations):

Definition 7.4 Let S be a region in G_n, we define:

- S is an *aligned region* if $\overline{|S|}$ divides either $f(S)$ or $nf(S)$.
- S is a *low-aligned region* if $\overline{|S|}$ divides $f(S)$.
- S is a *high-aligned region* if $\overline{|S|}$ divides $nf(S)$.
- When S is an aligned region, $\overline{S}$ is the whole region containing S and with size $\overline{|S|}$.

node 0:	00000	00100	01100	01000	01001	01011
node 1:	00001	00101	01101	01001		
node 2:	00011	00111	01111	01011	01001	01000
node 3:	00010	10010	11010	11000		
node 4:	00110	10110	11110	11010	11011	11001
node 5:	00111	10111	11111	11011		
node 6:	00101	10101	11101	11001	11011	11010
node 7:	00100	10100	11100	11110		
node 8:	01100	11100	11101	11111		
node 9:	01101	11101				
node 10:	01111	11111	11101	11100		
node 11:	01110	11110	10110	10100		
node 12:	01010	11010	10010	10110	10111	10101
node 13:	01011	11011	10011	10111		
node 14:	01001	11001	10001	10101	10111	10110
node 15:	01000	11000	10000	10010		
node 16:	11000	11100	10100	10000	10001	10011
node 17:	11001	11101	10101	10001		
node 18:	11011	11111	10111	10011	10001	10000
node 19:	11010	01010	00010	00000		
node 20:	11110	01110	00110	00010	00011	00001
node 21:	11111	01111	00111	00011		
node 22:	11101	01101	00101	00001	00011	00010
node 23:	11100	01100	00100	00110		
node 24:	10100	00100	00101	00111		
node 25:	10101	00101				
node 26:	10111	00111	00101	00100		
node 27:	10110	00110	01110	01100		
node 28:	10010	00010	01010	01110	01111	01101
node 29:	10011	00011	01011	01111		
node 30:	10001	00001	01001	01101	01111	01110
node 31:	10000	00000	01000	01010		

Figure 7.2: Routing paths generated by the Disjoint Link Algorithm for a 5-cube with shift distance 13.

□

Definition 7.5 The function r, inv_f, $move_f$, adj, and ADJ are defined as follows:

- Let $S = [G_n(a), \ldots, G_n(b)]$ be a *whole* region, then $r(G_n(a+i), S) = G_n(b-i)$, $\forall G_n(a+i) \in S$.
- Let $S = [G_n(a), \ldots, G_n(b)]$ be an *aligned* region, then $inv_f(G_n(a+i), S) = G_n(b-i)$, $\forall G_n(a+i) \in S$.
- Let S be an aligned region but not a whole region. In addition, let $p = |\overline{S}| - |S|$ if S is a low-aligned region; otherwise, let $p = |S| - |\overline{S}|$. Then define $move_f(G_n(i), S) = G_n(i+p)$, $\forall G_n(i) \in S$.
- Let S_1 be a high-aligned region, S_2 be a low-aligned region, $|S_1| = |S_2|$, and $nf(S_1) = f(S_2)$. Then $adj(G_n(l(S_1)-i), S_1, next) = G_n(f(S_2)+i)$, $adj(G_n(f(S_2)+i), S_2, prev) = G_n(l(S_1)-i)$, for $0 \le i < |S_1|$. Furthermore, define $ADJ(S_1, next) = S_2$ and $ADJ(S_2, prev) = S_1$.

□

In our algorithm described later, *function* inv_f and $move_f$ will be implemented by *procedure inv*() and *move*() respectively. (This is why we use subscript f for these two functions.) The implementation of function r, adj, and ADJ is omitted however, since these functions can be implemented easily by simple mathematical computations, as hinted by the following lemma:

Lemma 7.9 *For an n-dimensional hypercube:*

1. *If S is a whole region in G_n, then for any node $v \in S$, node v and node $r(v, S)$ are connected by a hypercube link.*
2. *If S is a low-aligned region in G_n, then for any node $v \in S$, node v and node $adj(v, S, prev)$ are connected by a hypercube link.*

3. If S is a high-aligned region in G_n, then for any node $v \in S$, node v and node $adj(v, S, next)$ are connected by a hypercube link.

Proof. Let $|S| = k$. Then G_n can be divided into $2^n/\overline{k}$ whole regions. Specifically, let $S(i)$ be $[G_n(i\overline{k}), G_n(i\overline{k}+1), \ldots, G_n(i\overline{k}+\overline{k}-1)]$, for $0 \le i \le 2^n/\overline{k} - 1$. From the construction of BRGC, it can be shown that

- For any node $v_1 = G_n(i\overline{k}+j)$ and $v_2 = G_n(i\overline{k}+\overline{k}-1-j)]$ in $S(i)$, the least significant $\log \overline{k} - 1$ bits of v_1 and v_2 are the same, and the bits at position $\log \overline{k}$ of v_1 and v_2 are complement to each other (the least significant bit is at position zero).
- For any node $v_1 = G_n(i\overline{k}+\overline{k}-1-j)$ in $S(i)$ and $v_2 = G_n((i+1)\overline{k}+j \pmod{2^n})$ in $S(i+1 \pmod{2^n/\overline{k}})$, the least significant $\log \overline{k}$ bits of v_1 and v_2 are the same.
- For any two nodes v_1 and v_2 in $S(i)$, the most significant $n - \log(\overline{k})$ bits of v_1 and v_2 are the same. Let's denote these $n - \log(\overline{k})$ bits by m_i.
- $[m_0, m_1, \ldots, m_{2^n/\overline{k}-1}]$ is equal to $G_{n-\log \overline{k}}$.

The lemma follows immediately from these facts. □

To make the algorithm described below clear, we further define some subregions derived from an aligned region S.

Definition 7.6 Let S be an aligned region[5] and $|\overline{S}| \ge 4$. We define:

$$\ll G_n(a), G_n(b) \gg \;=\; \begin{cases} [G_n(a), G_n(a+1), \ldots, G_n(b)] & \text{if } a \le b \\ [G_n(b), G_n(b+1), \ldots, G_n(a)] & \text{if } a > b \end{cases}$$

$$g \;=\; \begin{cases} f(S) & \text{if } S \text{ is a low-aligned region} \\ l(S) & \text{if } S \text{ is a high-aligned region} \end{cases}$$

[5] When S is also a whole region, S can be regarded as a region, either low-aligned or high-aligned. It does not affect the correctness of our algorithm.

Figure 7.3: The subregion names for an aligned region S, which consists of S_a, S_b, and S_c here.

$$\triangle \;=\; \begin{cases} + & \text{if } S \text{ is a low-aligned region} \\ - & \text{if } S \text{ is a high-aligned region} \end{cases}$$

Furthermore, let $|\overline{S}| = 2^k$, $p = |\overline{S}| - |S|$ and $q = 2^{k-1} - p$. Then a set of subregions is defined as follows (please see Figure 7.3 for clarity):

$$\begin{aligned}
S_a &= \ll G_n(g), G_n(g\triangle(q-1)) \gg \\
S_b &= \ll G_n(g\triangle q), G_n(g\triangle(2^{k-1}-1)) \gg \\
S_c &= \ll G_n(g\triangle 2^{k-1}), G_n(g\triangle(2^{k-1}+q-1)) \gg \\
S_d &= \ll G_n(g\triangle(2^{k-1}+q), G_n(g\triangle(2^k-1)) \gg \\
S_{ar} &= \ll G_n(g), G_n(g\triangle(p-1)) \gg
\end{aligned}$$

$$\begin{aligned}
S_{br} &= \ll G_n(g\triangle p), G_n(g\triangle(2^{k-1}-1)) \gg \\
S_{cr} &= \ll G_n(g\triangle 2^{k-1}), G_n(g\triangle(2^{k-1}+p-1)) \gg \\
S_{dr} &= \ll G_n(g\triangle(2^{k-1}+p), G_n(g\triangle(2^{k}-1)) \gg \\
S_{aq} &= \ll G_n(g), G_n(g\triangle(2^{k-2}-1)) \gg \\
S_{bq} &= \ll G_n(g\triangle 2^{k-2}), G_n(g\triangle(2^{k-1}-1)) \gg \\
S_{cq} &= \ll G_n(g\triangle 2^{k-1}), G_n(g\triangle(3\cdot 2^{k-2}-1)) \gg \\
S_{dq} &= \ll G_n(g\triangle(3\cdot 2^{k-1})), G_n(g\triangle(2^{k}-1)) \gg \\
S_{ab} &= \ll G_n(g), G_n(g\triangle(2^{k-1}-1)) \gg \\
S_{cd} &= \ll G_n(g\triangle 2^{k-1}), G_n(g\triangle(2^{k}-1)) \gg
\end{aligned}$$

In addition, when $|\overline{S}| \geq 8$ and $\frac{1}{2}|\overline{S}| < |S| < \frac{5}{8}|\overline{S}|$, we define:

$$\begin{aligned}
S_{aa} &= \ll G_n(g\triangle(2^{k-2}-q)), G_n(g\triangle(2^{k-2}-1)) \gg \\
S_{bb} &= \ll G_n(g\triangle 2^{k-2}), G_n(g\triangle(2^{k-2}+q-1)) \gg \\
S_{cc} &= \ll G_n(g\triangle(3\cdot 2^{k-2}-q)), G_n(g\triangle(3\cdot 2^{k-2}-1)) \gg \\
S_{dd} &= \ll G_n(g\triangle(3\cdot 2^{k-2})), G_n(g\triangle(3\cdot 2^{k-2}+q-1)) \gg
\end{aligned}$$

$\square$

We are ready to describe the Disjoint Link Algorithm now:

Algorithm 7.2 Given a hypercube of dimension n, a node $G_n(i)$, and a shift distance d, where $0 < d < N = 2^n$, procedure $route(G_n(i), d, n)$ will generate the path from node $G_n(i)$ to node $G_n(i + d \pmod{N})$ by using a contiguous sequence of hypercube links.

```
**procedure** $route(v, d, n)$
{
    **if** $(d \leq N/2)$
        $d' = d$; $dir = next$;
    **else** { $/ * \, d > N/2 * /$
        $d' = N - d$;  $dir = prev$;
    Divide $G_n$ into whole regions
        of size $\overline{d'}$ each;
    Let the whole region containing $v$
        be $S'$;
    **if** $(d \leq N/2)$
        $S =$ the high-aligned region in $S'$
                with size $d'$;
    **else**
        $S =$ the low-aligned region in $S'$
                with size $d'$;
    $path = v$;
    **if** $(\frac{1}{2}|\overline{S}| < |S| \leq \frac{3}{4}|\overline{S}|)$ {
        **if** $(v \in S_d)$ {
            $v = r(v, \overline{S})$;  $path = path ▯ v$;
            /* ▯ is a concatenation operator.*/
            $path = path ▯ inv(v, S_{ar})$;
        } **else** { $/ * \, v \in S * /$
            $v = adj(v, S, dir)$;  $path = path ▯ v$;
            $path = path ▯ inv(v, ADJ(S, dir))$;
        }
    } **else** { $/ * \, \frac{3}{4}|\overline{S}| < |S| \leq |\overline{S}| * /$
        **if** $(v \in S_d)$ {
            $v = r(v, S_{cd})$;  $path = path ▯ v$;
            $v = r(v, \overline{S})$;  $path = path ▯ v$;
            $v = r(v, S_{ab})$;  $path = path ▯ v$;
            $path = path ▯ inv(v, S_{ar})$;
        } **else** { $/ * \, v \in S * /$
            $v = adj(v, S, dir)$;  $path = path ▯ v$;
            $path = path ▯ inv(v, ADJ(S, dir))$;
        }
    }
}
```

```
**procedure** $inv(v, S)$
{
    $path = \textbf{null}$;
    **if** ($S$ is a whole region and $|S| > 1$) {
        $v = r(v, S)$;  $path = path ▯ v$;
    } **else if** ($\frac{1}{2}|\overline{S}| < |S| < \frac{5}{8}|\overline{S}|$) {
        **if** ($v \in S_a$) {
            $v = r(v, S_{ab})$;  $path = path ▯ v$;
            $v = r(v, \overline{S})$;  $path = path ▯ v$;
            $path = path ▯ inv(v, S_c)$;
        } **else if** ($v \in S_b$) {
            $path = path ▯ inv(v, S_b)$;
        } **else if** ($v \in S_c$) {
            $v = r(v, \overline{S})$;  $path = path ▯ v$;
            $v = r(v, S_{bq})$;  $path = path ▯ v$;
            $v = r(v, S_{ab})$;  $path = path ▯ v$;
            $v = r(v, S_{aq})$;  $path = path ▯ v$;
            $path = path ▯ inv(v, S_a)$;
        }
    } **else if** ($\frac{5}{8}|\overline{S}| \le |S| \le \frac{3}{4}|\overline{S}|$) {
        **if** ($v \in S_a$) {
            $v = r(v, S_{ab})$;  $path = path ▯ v$;
            $v = r(v, \overline{S})$;  $path = path ▯ v$;
            $path = path ▯ inv(v, S_c)$;
        } **else if** ($v \in S_b$) {
            $path = path ▯ inv(v, S_b)$;
        } **else if** ($v \in S_c$) {
            $v = r(v, \overline{S})$;  $path = path ▯ v$;
            $v = r(v, S_{ab})$;  $path = path ▯ v$;
            $path = path ▯ inv(v, S_a)$;
        }
    } **else if** ($\frac{3}{4}|\overline{S}| < |S| < |\overline{S}|$) {
        **if** ($v \in S_a$) {
            $v = r(v, \overline{S})$;  $path = path ▯ v$;
            $path = path ▯ move(v, S_{dr})$;
        } **else if** ($v \in S_b$) {
            $path = path ▯ inv(v, S_b)$;
        } **else if** ($v \in S_c$) {
            $v = r(v, \overline{S})$;  $path = path ▯ v$;
            $path = path ▯ move(v, S_{br})$;
        }
    }
    **return**($path$);
}
```

```
procedure $move(v, S)$
{
    $path$ = null;
    if ($\frac{1}{2}|\overline{S}| < |S| < \frac{5}{8}|\overline{S}|$) {
        if ($v \in S_a$) {
            $v = r(v, S_{aq})$;  $path = path \square v$;
            $v = r(v, S_{ab})$;  $path = path \square v$;
            $v = r(v, S_{bq})$;  $path = path \square v$;
            $path = path \square inv(v, S_{br})$;
        } else if ($v \in S_b$) {
            $v = r(v, \overline{S})$;  $path = path \square v$;
            $path = path \square inv(v, S_{cr})$;
        } else if ($v \in S_c$) {
            $v = r(v, S_{cq})$;  $path = path \square v$;
            $v = r(v, S_{cd})$;  $path = path \square v$;
            $v = r(v, S_{dq})$;  $path = path \square v$;
            $path = path \square inv(v, S_{dr})$;
        }
    } else if ($\frac{5}{8}|\overline{S}| \leq |S| \leq \frac{3}{4}|\overline{S}|$) {
        if ($v \in S_a$) {
            $v = r(v, S_{ab})$;  $path = path \square v$;
            $path = path \square inv(v, S_{br})$;
        } else if ($v \in S_b$) {
            $v = r(v, \overline{S})$;  $path = path \square v$;
            $path = path \square inv(v, S_{cr})$;
        } else if ($v \in S_c$) {
            $v = r(v, S_{cd})$;  $path = path \square v$;
            $path = path \square inv(v, S_{dr})$;
        }
    } else if ($\frac{3}{4}|\overline{S}| < |S| < |\overline{S}|$) {
        if ($v \in S_a$) {
            $path = path \square move(v, S_a)$;
        } else if ($v \in S_b$) {
            $v = r(v, \overline{S})$;  $path = path \square v$;
            $path = path \square inv(v, S_{cr})$;
        } else if ($v \in S_c$) {
            $path = path \square move(v, S_c)$;
        }
    }
    return($path$);
}
```

$\Box$

7.4.2 Correctness

In this section, we prove that the Disjoint Link Algorithm routes a message from any node to its cyclic-shifted sink node by using only hypercube links. We first prove the correctness of procedure *inv* and *move*:

Lemma 7.10 *For an aligned region S, we have:*

1. *When $|S| \geq 1$, procedure $inv(v, S)$ routes a message from any node $v \in S$ to node $inv_f(v, S)$ by using a contiguous sequence of hypercube links.*
2. *When $|S| \geq 3$ and S is not a whole region, procedure $move(v, S)$ routes a message from any node $v \in S$ to node $move_f(v, S)$ by using a contiguous sequence of hypercube links.*

Proof. Note that when procedure $inv(v, S)$ or $move(v, S)$ recursively call a procedure, the region parameter (S_a, S_b, S_c, S_{br}, S_{cr}, or S_{dr}) passed to the called procedure is also an aligned region, and the size of the subregion is smaller than $|S|$. From this observation and Lemma 7.9, this lemma can be proved by induction on the size of S. □

The correctness of the Disjoint Link Algorithm can be proved now:

Theorem 7.11 *For any node $G_n(i)$ in an n-dimensional hypercube, the Disjoint Link Algorithm routes a message from node $G_n(i)$ to node $G_n(i + d \text{ (mod } 2^n))$ by using a contiguous sequence of hypercube links.*

Proof. Note that when $d > N/2$, $G_n(i + d \pmod{2^n}) = G_n(i - (N - d) \pmod{2^n})$. The theorem follows immediately from Lemmas 7.9 and 7.10. □

7.4.3 Disjoint Link Property

In this section, we prove that all of the routing paths generated by the Disjoint Link Algorithm are link disjoint. We need three more definitions:

Definition 7.7 Let S_1 be a whole region and S_2 be a subregion of S_1. Then define $links(S_2, S_1) = \{v \rightarrow r(v, S_1) \,|\, \forall v \in S_2\}$. □

Definition 7.8 Given a region S, a link $v_1 \rightarrow v_2$ is called *inside* S if both v_1 and v_2 are in S. Otherwise, the link is called *outside* S. □

Definition 7.9 A link $v_1 \rightarrow v_2$ is *used* by procedure $inv(v, S)$ (or $move(v, S)$) if v_1 and v_2 are two consecutive nodes in $v \, path$, where *path* is the value returned by $inv(v, S)$ (or $move(v, S)$). □

The proof of the disjoint link property (Theorem 7.17) relies on the sequence of Lemmas (Lemma 7.12 to 7.16), sketched below. We will first prove these lemmas:

Lemma 7.12 *For an aligned region S, we have:*

1. No link outside $\overline{S}$ is used by $inv(v, S)$, for any node $v \in S$.

2. When S is not a whole region, no link outside $\overline{S}$ is used by $move(v, S)$ for any node $v \in S$.

Proof. This lemma can be proved by induction on the size of S, the *region parameter* passed to the procedure *inv* or *move*. When $|S| = 1$, the lemma is obviously true. Suppose this lemma (i.e., part 1 and 2) is true for all S with $|S| < k$. We prove that part 1 is true when $|S| = k$ by considering the following cases in the procedure $inv(v, S)$:

Case 1: When S is a whole region, part 1 is obviously true from Lemma 7.9.

Case 2: When $|S| \in (\frac{1}{2}|\overline{S}|, \frac{5}{8}|\overline{S}|)$, consider the following subcases:

- When node $v \in S_a$, statement $v = r(v, S_{ab})$ and $v = r(v, \overline{S})$ route a message from node v in S_a to a node in S_{br}, and then to a node in S_c by Lemma 7.9. It is clear that these links are not outside region $\overline{S}$ by Definition 7.6. In addition, the recursively called procedure $inv(v, S_c)$ does not use any link outside $\overline{S_c}$ by induction hypothesis on part 1 of the lemma. Since any link outside $\overline{S}$ is also outside $\overline{S_c}$, part 1 holds for this subcase.
- When node $v \in S_b$, the recursively called procedure $inv(v, S_b)$ does not use any link outside $\overline{S_b}$ by induction hypothesis on part 1. Since any link outside $\overline{S}$ is also outside $\overline{S_b}$, part 1 holds for this subcase.
- When node $v \in S_c$, statement $v = r(v, \overline{S})$, $v = r(v, S_{bq})$, $v = r(v, S_{ab})$, and $v = r(v, S_{aq})$ route a message from node v in S_c to a node in S_{br}, then to a node in S_{bb}, then to a node in S_{aa}, and finally to a node in S_a by Lemma 7.9. It is clear that these links are not outside region $\overline{S}$ by Definition 7.6. In addition, the recursively called procedure $inv(v, S_a)$ does not use any link outside $\overline{S_a}$ by induction hypothesis on part 1 of the lemma. Since any link outside $\overline{S}$ is also outside $\overline{S_a}$, part 1 holds for

this subcase.

Case 3: When $|S| \in [\frac{5}{8}|\overline{S}|, \frac{3}{4}|\overline{S}|]$, it can be shown that part 1 is true by the discussion similar to Case 2.

Case 4: When $|S| \in (\frac{3}{4}|\overline{S}|, |\overline{S}|)$, consider the following subcases:

- When node $v \in S_a$, statement $v = r(v, \overline{S})$ routes a message from node v in S_a to a node in S_{dr} by Lemma 7.9. It is clear that these links are not outside region $\overline{S}$ by Definition 7.6. In addition, the recursively called procedure $move(v, S_{dr})$ does not use any link outside $\overline{S_{dr}}$ by induction hypothesis on part 2 of the lemma. Since any link outside $\overline{S}$ is also outside $\overline{S_{dr}}$, part 1 holds for this subcase.
- When node $v \in S_b$, the recursively called procedure $inv(v, S_b)$ does not use any link outside $\overline{S_b}$ by induction hypothesis on part 1 of the lemma. Since any link outside $\overline{S}$ is also outside $\overline{S_b}$, part 1 holds for this subcase.
- When node $v \in S_c$, it can be proved that part 1 is true by similar discussion to the above subcase when $v \in S_a$.

From these four cases, it is clear that part 1 is true when $|S| = k$. Part 2 can be proved similarly and hence we omit its proof here. □

The following four lemmas are all based on the previous lemma and can be proved by case discussions as that in the previous lemma.

Lemma 7.13 *Let S be an aligned region but not a whole region, then no link in $links(S_d, \overline{S})$ is used by $move(v, S)$ for any node $v \in S$.*

Proof. This lemma is true based on Lemma 7.9, Lemma 7.12 and Definition 7.6. □

Based on previous two lemmas, we have the following lemma:

Lemma 7.14 *Let S be an aligned region, then we have:*

1. When $|S| \in (\frac{1}{2}|\overline{S}|, \frac{3}{4}|\overline{S}|]$, any link in $links(S_d, \overline{S})$ or $links(S_{ar}, \overline{S})$ is not used by $inv(v, S)$ for any node $v \in S$.

2. When $|S| \in (\frac{3}{4}|\overline{S}|, |\overline{S}|)$, any link in $links(S_{ar}, S_{ab})$, $links(S_b, \overline{S})$, $links(S_{cr}, S_{cd})$, or $links(S_d, \overline{S})$ is not used by $inv(v, S)$ for any node $v \in S$.

Proof. Part 1 of the lemma is true based on Lemma 7.9, part 1 of Lemma 7.12, and Definition 7.6. Part 2 of the lemma is true based on Lemma 7.9, Lemma 7.12, Lemma 7.13, and Definition 7.6. □

Based on previous three lemmas, we have the following lemma:

Lemma 7.15 *Let S be an aligned region but not a whole region, we have:*

1. No link is used by both $inv(v, S)$ and $inv(v_d, S_d)$ for any node $v \in S$ and $v_d \in S_d$.

2. No link is used by both $move(v, S)$ and $inv(v_{ar}, S_{ar})$ for any node $v \in S$ and $v_{ar} \in S_{ar}$.

Proof. By Lemma 7.9, 7.12, 7.14, and Definition 7.6, this lemma can be proved by induction on the size of S. The proof in an induction step involves case by case discussions as that in Lemma 7.12, hence we omit it here. □

Based on previous four lemmas, we have the following lemma:

Lemma 7.16 *Let S be an aligned region, then we have:*

1. No link is used more than once by $inv(v_1, S)$ and $inv(v_2, S)$ for any node v_1 and v_2 in S.

2. When S is not a whole region, no link is used more than once by $move(v_1, S)$ and $move(v_2, S)$ for any node v_1 and v_2 in S.

Proof. By Lemma 7.9, 7.12, 7.14, 7.15, and Definition 7.6, this lemma can be proved by induction on the size of S. □

Now we are ready to prove the disjoint link property:

Theorem 7.17 *The Disjoint Link Algorithm routes messages from all of the nodes in a hypercube to their cyclic-shifted sink nodes by using link-disjoint paths.*

Proof. This theorem is true based on Lemma 7.9, 7.12, 7.14, 7.15, 7.16, and Definition 7.6. □

7.4.4 Communication Complexity

In this section, we show the upper bound on the number of links used to route a message when using the Disjoint Link Algorithm:

Theorem 7.18 *On an n-dimensional hypercube, a message sent from source node $G_n(i)$ to sink node $G_n(i + d \ (mod \ 2^n))$ needs less than $1 + \frac{4}{3} \log \overline{d'}$ communication steps, where $d' = d$ if $0 < d \leq 2^{n-1}$ and $d' = 2^n - d$ if $2^{n-1} < d < 2^n$, provided the Disjoint Link Algorithm is used.*

Proof. Since the Disjoint Link Algorithm yields no link congestion by Theorem 7.17, the number of communication steps needed to route a message is equal to the number of links in the routing path.

In procedure $route()$ of the Disjoint Link Algorithm, routing a message is achieved first by traversing through several links, say k links, and then calling the procedure $inv(v, S'')$, where S'' is the region parameter passed to procedure $inv()$. Table 7.1 summarizes the values of k and the sizes of S'' for various cases. Hence the total number of links used to route a message is k plus the number of links used by procedure $inv(v, S'')$.

We compute the number of links used by the called procedure $inv()$ as follows. Note that routing a message by procedure $inv()$ involves a sequence of recursive procedure calls, with the sizes of the region parameters passed to recursively-called procedures getting smaller and smaller. Let rs_1 be the size of the region parameter of the calling procedure,

rs_2 be the size of the region parameter of the recursively-called procedure. Then define the *size shrinkage ratio* sh to be $\log(\overline{rs_1}/\overline{rs_2})$. Note that sh is always greater than or equal to one in the Disjoint Link Algorithm. In addition, let l be the number of links used by the calling procedure, excluding the number of links used by the recursively-called procedure. Then define av to be the average number of links used by such a recursive call, i.e., $av = l/sh$. Hence the total number of links used by procedure $inv(v, S'')$ cannot exceed $\log |S''|$ times the maximum value of av over all of the procedure calls.

The values of l, sh, av, and the recursively-called procedures for procedures $inv()$ and $move()$ are summarized in Table 7.1. As an example, in procedure $inv(v, S)$, if $v \in S_c$ and $\frac{1}{2}|\overline{S}| < |S| < \frac{5}{8}|\overline{S}|$, four links are used in the calling procedure and then procedure $inv(v, S_a)$ is called recursively. Since $sh = \log(|\overline{S}|/|\overline{S_a}|) > \log 2^3 = 3$, we have $av = l/sh < 4/3$, that is, less than $\frac{4}{3}$ links are used per dimension in this case. The value $\frac{4}{3}$ happens to be the upper bound on all of the possible values of av, as can be observed from Table 7.1.

By referring to procedure $route()$, we have the upper bound on the total number of links in a routing path, i.e., $u = k + av \log |S''|$, as follows:

- When $\frac{1}{2}|\overline{S}| < |S| \leq \frac{3}{4}|\overline{S}|$ and $v \in S$, we have $u \leq 1 + \frac{4}{3}\log \overline{d}$.
- When $\frac{1}{2}|\overline{S}| < |S| \leq \frac{3}{4}|\overline{S}|$ and $v \in S_d$, we have $u \leq 1 + \frac{4}{3}\log \frac{\overline{d}}{2} < 1 + \frac{4}{3}\log \overline{d}$.
- When $\frac{3}{4}|\overline{S}| < |S| \leq |\overline{S}|$ and $v \in S$, we have $u \leq 1 + \frac{4}{3}\log \overline{d}$.
- When $\frac{3}{4}|\overline{S}| < |S| \leq |\overline{S}|$ and $v \in S_d$, we have $u \leq 3 + \frac{4}{3}\log \frac{\overline{d}}{4} < 1 + \frac{4}{3}\log \overline{d}$.

$\square$

It can be verified that the largest value of $\overline{d'}$ is 2^{n-1}. Hence on an n-dimensional hypercube, the number of routing steps needed by any routing path cannot exceed $1 + \frac{4}{3}(n-1) < \frac{4}{3}n$, for any shift distance d.

Table 7.1: Summary of link usage by procedures.

Summary of links usage by procedure $route(v, d, n)$				
	$\frac{1}{2}\lvert\overline{S}\rvert < \lvert S\rvert \leq \frac{3}{4}\lvert\overline{S}\rvert$		$\frac{3}{4}\lvert\overline{S}\rvert < \lvert S\rvert \leq \lvert\overline{S}\rvert$	
	$v \in S$	$v \in S_d$	$v \in S$	$v \in S_d$
k	1	1	1	3
p[†]	$i(v, ADJ(S, dir))$	$i(v, S_{ar})$	$i(v, ADJ(S, dir))$	$i(v, S_{ar})$
remark	$\lvert ADJ(S, dir)\rvert \leq \overline{d'}$	$\lvert S_{ar}\rvert \leq \overline{d'}/2$	$\lvert ADJ(S, dir)\rvert \leq \overline{d'}$	$\lvert S_{ar}\rvert \leq \overline{d'}/4$

Summary of links usage by procedure $inv(v, S)$											
	$\frac{1}{2}\lvert\overline{S}\rvert < \lvert S\rvert < \frac{5}{8}\lvert\overline{S}\rvert$			$\frac{5}{8}\lvert\overline{S}\rvert \leq \lvert S\rvert \leq \frac{3}{4}\lvert\overline{S}\rvert$			$\frac{3}{4}\lvert\overline{S}\rvert < \lvert S\rvert < \lvert\overline{S}\rvert$			$\lvert S\rvert = \lvert\overline{S}\rvert$	$\lvert S\rvert = 1$
	$v \in S_a$	$v \in S_b$	$v \in S_c$	$v \in S_a$	$v \in S_b$	$v \in S_c$	$v \in S_a$	$v \in S_b$	$v \in S_c$	$\lvert\overline{S}\rvert > 1$	
l	2	0	4	2	0	2	1	0	1	1	0
p[†]	$i(v, S_c)$	$i(v, S_b)$	$i(v, S_a)$	$i(v, S_c)$	$i(v, S_b)$	$i(v, S_a)$	$m(v, S_{dr})$	$i(v, S_b)$	$m(v, S_{br})$		
sh	> 3	> 1	> 3	≥ 2	≥ 1	≥ 2	> 1	> 2	> 1		
av	$< \frac{2}{3}$	$= 0$	$< \frac{4}{3}$	≤ 1	$= 0$	≤ 1	< 1	$= 0$	< 1	< 1[‡]	$= 0$

Summary of links usage by procedure $move(v, S)$									
	$\frac{1}{2}\lvert\overline{S}\rvert < \lvert S\rvert < \frac{5}{8}\lvert\overline{S}\rvert$			$\frac{5}{8}\lvert\overline{S}\rvert \leq \lvert S\rvert \leq \frac{3}{4}\lvert\overline{S}\rvert$			$\frac{3}{4}\lvert\overline{S}\rvert < \lvert S\rvert < \lvert\overline{S}\rvert$		
	$v \in S_a$	$v \in S_b$	$v \in S_c$	$v \in S_a$	$v \in S_b$	$v \in S_c$	$v \in S_a$	$v \in S_b$	$v \in S_c$
l	3	1	3	1	1	1	0	1	0
p[†]	$i(v, S_{br})$	$i(v, S_{cr})$	$i(v, S_{dr})$	$i(v, S_{br})$	$i(v, S_{cr})$	$i(v, S_{dr})$	$m(v, S_a)$	$i(v, S_{cr})$	$m(v, S_c)$
sh	> 3	> 1	> 3	≥ 2	> 1	≥ 2	> 1	> 2	> 1
av	< 1	< 1	< 1	$\leq \frac{1}{2}$	< 1	$\leq \frac{1}{2}$	$= 0$	$< \frac{1}{2}$	$= 0$

[†] P denotes the recursively called procedure. $i()$ stands for $inv()$ and $m()$ stands for $move()$.
[‡] Since the size of the whole region is greater than 1, the value of av is no more than 1.

7.5 The Comparison

In this section, we will compare the performance of various cyclic shift algorithms, as well as shows the effects of these algorithms on the performance of a matrix multiplication algorithm which uses cyclic shifts as intrinsic operations. In our following discussion, the multiple-ports model will be used. That is, it is assumed that all links incident on a hypercube node can send/receive messages at the same time step.

The following cyclic shift algorithms will be considered in our comparison:

Algorithm *J1***:** the first cyclic shift algorithm mentioned in Johnsson [68].

Algorithm *SP***:** the Shortest Path Algorithm mentioned in Section 7.3.

Algorithm *DL***:** the Disjoint Link Algorithm mentioned in Section 7.4.

Algorithm *NA***:** the cyclic shift algorithm for natural order embedding mentioned in Ranka and Sahni [108].

For all of these algorithms on synchronous and asynchronous hypercubes, whether or not the link disjoint properties hold are summarized below:

	Algorithm *J1*	Algorithm *SP*	Algorithm *DL*	Algorithm *NA*
synchronous hypercube	yes	yes	yes	yes
asynchronous hypercube	no	no	yes	yes

Note that when the link disjoint property does not hold, routing a message from a source node to a sink node may encounter additional delay due to link congestion. That is,the communication times for Algorithm *J1* and Algorithm *SP* on asynchronous hypercubes will be larger than the corresponding times on synchronous hypercubes.

Mathematically, let $d^{x}_{i,s}$ denote the number of *links* needed to route a message from node $S(i)$ to node $S(i+s)$ by Algorithm x, where $S(y) = y \pmod{2^n}$ for Algorithm *NA*, and $S(y) = G_n(y \pmod{2^n})$ for Algorithm *J1*, *SP*, and *DL*. Furthermore, let $s^{x}_{i,s}$ ($a^{x}_{i,s}$ respectively) denote the number of *communication steps* (i.e., including delays due to link congestion) needed to route a message from node $S(i)$ to node $S(i+s)$ by Algorithm x on a synchronous (an asynchronous, respectively) hypercube. Then we have:

- $s^{J1}_{i,s} = d^{J1}_{i,s}$ and $a^{J1}_{i,s} > d^{J1}_{i,s}$.
- $s^{SP}_{i,s} = d^{SP}_{i,s}$ and $a^{SP}_{i,s} > d^{SP}_{i,s}$.
- $s^{DL}_{i,s} = a^{DL}_{i,s} = d^{DL}_{i,s}$.
- $s^{NA}_{i,s} = a^{NA}_{i,s} = d^{NA}_{i,s}$.

Since Algorithm *J1* has less competitive performance even on synchronous hypercubes, we will not consider its performance on asynchronous hypercubes. For Algorithm *SP* on asynchronous n-cubes, we compute its worst-case behavior $a^{SP}_{i,s}$ as follows. Let $p_0 p_1 \cdots p_k$ be the routing path starting from node $G_n(i) = p_0$ with shift distance s. Define c_j, $0 \le j < k$, to be the total number of paths passing through link $p_j \rightarrow p_{j+1}$ for the cyclic shift operation. (Recall that a cyclic shift operation on an n-cube has 2^n routing paths.) Then we have $a^{SP}_{i,s} = \sum_{j=0}^{k-1} c_j$.

In addition, since a cyclic shift operation involves 2^n routing paths, we are interested in the longest communication time over all of these 2^n routing paths. That is, we are concerned with the value of $p^{x}_{s} = \max\{t_{i,s} \mid 0 \le i \le 2^n - 1\}$, where

- for Algorithm *J1* on synchronous hypercubes, x is *J1S* and $t_{i,s}$ is $s^{J1}_{i,s}$;
- for Algorithm *SP* on synchronous hypercubes, x is *SPS* and $t_{i,s}$ is $s^{SP}_{i,s}$;
- for Algorithm *SP* on asynchronous hypercubes, x is *SPA* and $t_{i,s}$ is $a^{SP}_{i,s}$;

- for Algorithm *DL* on synchronous and asynchronous hypercubes, x is *DL* and $t_{i,s}$ is $d_{i,s}^{DL}$;

- for Algorithm *NA* on synchronous and asynchronous hypercubes, x is *NA* and $t_{i,s}$ is $d_{i,s}^{NA}$.

For convenience, we will say Algorithm *J1S* to stand for Algorithm *J1* on synchronous hypercubes. The same convention will be used for Algorithm *SPS* and *SPA*.

7.5.1 Average and Worst Case Comparison

To gain the average performance of these cyclic shift algorithms, we first compare the average routing steps needed by Algorithm x over all possible shift distances, $l_a^x = \frac{1}{2^n-1}\sum_{s=1}^{2^n-1} p_s^x$. Table 7.2 shows the values of l_a^x for hypercubes from dimension 4 to dimension 14. On synchronous hypercubes, it can be seen that Algorithm *SPS* yields the best performance. In fact, Algorithm *SPS* is optimal as it always uses the shortest paths for routing and guarantees no link congestion in this setting. On asynchronous hypercubes, it can be seen that Algorithm *DL* yields the best performance. This is not surprising, as in our design of the Disjoint Link Algorithm, only a small number of extra links are used to avoid link congestion.

Furthermore, the values of l_a^{J1S} and l_a^{NA} on an n-dimensional hypercubes can be expressed by simple functions of n as follows:

- $l_a^{J1S} = (2n-1)2^{n-1}/(2^n-1)$, because:
  - For a shift distance $s = 2^{s1} + \cdots 2^{sk}$, where $s1 > s2 > \cdots > sk$, we have $d_{i,s}^{J1} = 2k$ if $sk \neq 0$, and $d_{i,s}^{J1} = 2k-1$ if $sk = 0$, for all $0 \leq i \leq 2^n - 1$ [68]. That is, for a given shift distance s whose binary representation contains k 1's, 2 units will be

Table 7.2: Average routing steps of cyclic shift algorithms over all shift distances.

	hypercube dimension										
	4	5	6	7	8	9	10	11	12	13	14
l_a^{SPS}	2.533	3.161	3.778	4.425	5.075	5.734	6.395	7.059	7.724	8.390	9.056
l_a^{DL}	2.533	3.290	4.032	4.835	5.639	6.462	7.286	8.116	8.947	9.779	10.612
l_a^{NA}	3.267	4.161	5.095	6.055	7.031	8.018	9.010	10.005	11.003	12.002	13.001
l_a^{SPA}	3.200	4.194	5.206	6.252	7.310	8.376	9.445	10.516	11.588	12.660	13.732
l_a^{J1S}	3.733	4.645	5.587	6.551	7.529	8.517	9.509	10.505	11.503	12.502	13.501

"charged" to every non-rightmost *1-component* and 1 unit will be "charged" to the rightmost *1-component*.

- For all shift distances $1 \leq i \leq 2^n - 1$, there are $(n-1)2^{n-1}$ non-rightmost 1-components and 2^{n-1} rightmost 1-components.
- Therefore, $l_a^{J1S} = ((n-1)2^{n-1} \cdot 2 + 2^{n-1} \cdot 1)/(2^n - 1) = (2n-1)2^{n-1}/(2^n - 1)$.

- $l_a^{NA} = \frac{(n-1)2^n+1}{2^n-1}$, because:
  - According to [108], $p_s^{NA} = n - f(s)$, where $f(s)$ is a function which returns the position of the rightmost bit of s that is 1.
  - There are 2^{n-1-i} s's between 1 and $2^n - 1$ (inclusively) so that $f(s) = i$.
  - Therefore, $l_a^{NA} = \sum_{i=0}^{n-1}(n-i)2^{n-1-i}/(2^n - 1) = \sum_{j=1}^{n} j2^{j-1}/(2^n - 1) = \frac{(n-1)2^n+1}{2^n-1}$

It is still open how to express other l_a^x's by simple functions of n.

Besides the *average* routing steps l_a^x, let us also consider the *maximal* routing steps needed by Algorithm x over all possible shift distances, $l_m^x = \max_{1 \leq s \leq 2^n-1} p_s^x$. Table 7.3 shows the values of l_m^x for hypercubes from dimension 4 to dimension 14. On synchronous

Table 7.3: Maximum routing steps of cyclic shift algorithms over all shift distances.

	hypercube dimension										
	n	4	5	6	7	8	9	10	11	12	13	14
l_m^{SPS}	$= n$	4	5	6	7	8	9	10	11	12	13	14
l_m^{DL}	$< \frac{4}{3}n$	4	5	7	8	9	11	12	13	15	16	17
l_m^{NA}	$= n$	4	5	6	7	8	9	10	11	12	13	14
l_m^{SPA}	$\leq 2n - 2$	5	8	9	12	13	16	17	20	21	24	25
l_m^{J1S}	$= 2n - 1$	7	9	11	13	15	17	19	21	23	25	27

hypercubes, it can be seen that Algorithm *SPS* and Algorithm *NA* yield the best performance. On asynchronous hypercubes, it can be seen that Algorithm *NA* yields the best performance. In addition, if we are confined to the BRGC order embedding, then Algorithm *DL* yields the best performance on asynchronous hypercubes.

Table 7.3 also shows the upper bound, expressed as a function of n, of l_m^x on an n-cube. The upper bounds for l_m^{SPS}, l_m^{SPA}, and l_m^{DL} are from Section 7.3 and 7.4; the upper bounds for l_m^{J1S} and l_m^{NA} are due to Johnsson [68] and Ranka and Sahni [108] respectively.

Finally, we consider the maximal routing steps needed by Algorithm x for shift distance equal to 1, $l_1^x = p_1^x$. This value is interesting because many parallel algorithms contain *neighboring* communications. The matrix multiplication algorithm that will be discussed in the next section is one of such algorithms. From Section 7.3, Section 7.4, Johnsson[68], and Ranka and Sahni [108], the values of l_1^x for an n-dimensional hypercube are as follows:

- $l_1^{J1S} = l_1^{SPS} = l_1^{SPA} = l_1^{DL} = 1$
- $l_1^{NA} = n$

It can be seen that except Algorithm *NA*, all other algorithms are perfect for neighboring communications. In fact, this is the reason why BRGC order embedding is used by many application algorithms. However, to be fair, Algorithm *NA* has the property that for all shift distances, a cyclic shift can be completed in n steps, and at each step, all of the routing paths can send messages out through hypercube links at the *same* dimension [108]. This property does not exist in other cyclic shift algorithms.

7.5.2 Matrix Multiplication Algorithm

Now let us study a matrix multiplication algorithm [68] to see how different cyclic shift algorithms affect the overall performance. Let A, B, and C be N by N matrices, and $N = 2^n$. The algorithm will compute $C = A \times B$ by using an $N \times N$ processor mesh. Each of these $N \times N$ processors is labeled by $[i, j]$, where $0 \leq i, j \leq N - 1$. Furthermore, processor $[i, j]$ will be allocated to hypercube node $iN + j$ if the natural order embedding is used, or to hypercube node $G_n(i)N + G_n(j)$ if the BRGC order embedding is used. An example of these two allocations for $N = 8$ is shown in Figure 7.4. Note that each row of the mesh is an n-cube, and so is each column.

The algorithm works as follows. On each processor $[i, j]$, registers $R_A^{i,j}$, $R_B^{i,j}$, and $R_C^{i,j}$ are used to store one element of matrices A, B, and C respectively. Initially, $A[i, j]$ and $B[i, j]$ are allocated to $R_A^{i,j}$ and $R_B^{i,j}$ respectively, and $R_C^{i,j}$ is initialized to zero, for every $0 \leq i, j < N$. After this initialization, the following program is executed:

```
1.  forall $(0 < i < N)$
2.     forall $(0 \leq j < N)$ $R_A^{i,j} \leftarrow R_A^{i,j+i \pmod{N}}$;
3.  forall $(0 < j < N)$
4.     forall $(0 \leq i < N)$ $R_B^{i,j} \leftarrow R_B^{i+j \pmod{N},j}$;
5.  for $(k = 0$ to $N - 1)$ {
6.     forall $(0 \leq i, j < N)$ $R_C^{i,j} = R_C^{i,j} + R_A^{i,j} * R_B^{i,j}$;
```

[7,0] 111 000	[7,1] 111 001	[7,2] 111 010	[7,3] 111 011	[7,4] 111 100	[7,5] 111 101	[7,6] 111 110	[7,7] 111 111
[6,0] 110 000	[6,1] 110 001	[6,2] 110 010	[6,3] 110 011	[6,4] 110 100	[6,5] 110 101	[6,6] 110 110	[6,7] 110 111
[5,0] 101 000	[5,1] 101 001	[5,2] 101 010	[5,3] 101 011	[5,4] 101 100	[5,5] 101 101	[5,6] 101 110	[5,7] 101 111
[4,0] 100 000	[4,1] 100 001	[4,2] 100 010	[4,3] 100 011	[4,4] 100 100	[4,5] 100 101	[4,6] 100 110	[4,7] 100 111
[3,0] 011 000	[3,1] 011 001	[3,2] 011 010	[3,3] 011 011	[3,4] 011 100	[3,5] 011 101	[3,6] 011 110	[3,7] 011 111
[2,0] 010 000	[2,1] 010 001	[2,2] 010 010	[2,3] 010 011	[2,4] 010 100	[2,5] 010 101	[2,6] 010 110	[2,7] 010 111
[1,0] 001 000	[1,1] 001 001	[1,2] 001 010	[1,3] 001 011	[1,4] 001 100	[1,5] 001 101	[1,6] 001 110	[1,7] 001 111
[0,0] 000 000	[0,1] 000 001	[0,2] 000 010	[0,3] 000 011	[0,4] 000 100	[0,5] 000 101	[0,6] 000 110	[0,7] 000 111

(a) Natural order embedding.

[7,0] 100 000	[7,1] 100 001	[7,2] 100 011	[7,3] 100 010	[7,4] 100 110	[7,5] 100 111	[7,6] 100 101	[7,7] 100 100
[6,0] 101 000	[6,1] 101 001	[6,2] 101 011	[6,3] 101 010	[6,4] 101 110	[6,5] 101 111	[6,6] 101 101	[6,7] 101 100
[5,0] 111 000	[5,1] 111 001	[5,2] 111 011	[5,3] 111 010	[5,4] 111 110	[5,5] 111 111	[5,6] 111 101	[5,7] 111 100
[4,0] 110 000	[4,1] 110 001	[4,2] 110 011	[4,3] 110 010	[4,4] 110 110	[4,5] 110 111	[4,6] 110 101	[4,7] 110 100
[3,0] 010 000	[3,1] 010 001	[3,2] 010 011	[3,3] 010 010	[3,4] 010 110	[3,5] 010 111	[3,6] 010 101	[3,7] 010 100
[2,0] 011 000	[2,1] 011 001	[2,2] 011 011	[2,3] 011 010	[2,4] 011 110	[2,5] 011 111	[2,6] 011 101	[2,7] 011 100
[1,0] 001 000	[1,1] 001 001	[1,2] 001 011	[1,3] 001 010	[1,4] 001 110	[1,5] 001 111	[1,6] 001 101	[1,7] 001 100
[0,0] 000 000	[0,1] 000 001	[0,2] 000 011	[0,3] 000 010	[0,4] 000 110	[0,5] 000 111	[0,6] 000 101	[0,7] 000 100

(b) BRGC order embedding.

Figure 7.4: Mapping a processor mesh onto hypercube nodes.

```
7.      forall $(0 \le i < N)$
8.         forall $(0 \le j < N)$ $R_A^{i,j} \leftarrow R_A^{i,j+1 \pmod{N}}$;
9.      forall $(0 \le j < N)$
10.        forall $(0 \le i < N)$ $R_B^{i,j} \leftarrow R_B^{i+1 \pmod{N},j}$;
    }
```

Note that Line 2, 4, 8, and 10 are actually cyclic shift operations. Furthermore,

- Line 1 and Line 2 together perform $2^n - 1$ row cyclic shifts simultaneously with shift distances from 1 to $2^n - 1$,
- Line 3 and Line 4 together perform $2^n - 1$ column cyclic shifts simultaneously with shift distances from 1 to $2^n - 1$,
- Line 7 and Line 8 together perform 2^n row cyclic shifts simultaneously with shift distances equal to 1,
- Line 9 and Line 10 together perform 2^n column cyclic shifts simultaneously with shift distances equal to 1.

Let us assume that the cyclic shifts around rows and around columns can be done simultaneously. Therefore, the time spent by the cyclic shifts in Line 1 to 4 is l_m^x, and the time spent by the cyclic shifts in Line 5 to 8 is Nl_1^x (note that the loop is iterated N times). From the formulae given for l_m^x and l_1^x, where x is *J1S*, *SPS*, *SPA*, *DL*, or *NA*, we have the upper bounds on the communication times needed by the matrix multiplication as follows:

$J1S$	SPS	SPA	DL	NA
$2n - 1 + 2^n$	$n + 2^n$	$2n - 2 + 2^n$	$\frac{4}{3}n + 2^n$	$n + n2^n$

From the above table, we conclude that:

- On synchronous hypercubes, the Shortest Path Algorithm (Section 7.3) yields the best performance.
- On asynchronous hypercubes, the Disjoint Link Algorithm (Section 7.4) yields the best performance.

Chapter 8

Conclusion

This thesis studied issues concerning the execution of regular loops on multiprocessor computers. The model of our problem was defined in Chapter 2. Chapter 3 identified a set of important properties of executing regular loops. By using these properties, we analyzed a greedy scheduling scheme for shared-memory multiprocessors in Chapter 4. In Chapter 5, the mapping problem for distributed-memory multiprocessors was formulated and its complexity was investigated. An efficient scheduling scheme for distributed-memory multiprocessors was proposed Chapter 6. Finally, in Chapter 7, two cyclic shift algorithms for hypercubes were explored.

The primary contributions of this work are as follows:

Properties of Regular Loop Execution:

Several important properties of executing regular loops in parallel were identified (Theorem 3.1, 3.2, 3.3, 3.8, 3.9, 3.10, 3.11). These properties reveal the inherent limitations of parallelizing regular loop execution, which were never ascertained before. Based on these properties, the design and analysis of parallelizing loop execution was substantially facilitated (Chapter 4). For example, one of these properties determined the number of processors needed, beyond which no speedup can be obtained

(Theorem 3.2).

Formulation of Mapping Problem:

The problem of mapping regular loops onto distributed-memory MIMD machines was formally formulated (Section 5.1). The formulation of the free mapping problem (Section 5.1.1) can be applied to general task graphs as well (instead of regular loops only). It considered not only the task graph topology, but also the processor graph topology. That is, the paths used for message routings and the avoidance of message congestion can be expressed explicitly. Previous work formulated mapping problems without expressing these explicitly [52,132].

Computational Complexity:

Several NP-hard results, related to the execution of regular loops on multiprocessors, were shown (Theorem 3.5, 3.7, 5.1, 5.2, and 5.6). In particular, we showed that finding an optimal time mapping vector is NP-hard (Theorem 5.6). Note that many previous algorithms used exponential time to find the optimal time mapping vector [84,91], but the problem complexity was never determined. Our result explains why the exponential time algorithms were used, because no polynomial time algorithm for such a problem exists in practice.

Parallelizing Compiler:

An efficient algorithm for mapping regular loops onto (distributed-memory) MIMD machines was proposed (Chapter 6). This algorithm involves finding a time mapping function (Section 6.1) and a processor mapping function (Section 6.2). Although the method for finding a time mapping function (Algorithm 6.3) has the same computational complexity as that in [84, page71], yet it always gives better parallel execution time (Theorem 6.3). In addition, the parallelized loop produced by our processor mapping function uses less processing elements than does that produced by the method in [124] (Lemma 6.6). Furthermore, because of different choices of projection vectors, the parallelized loop can be generated much easier by our algorithm compared to the algorithm in [124] (Section 6.2.5). Another important property of our mapping strategy is that it is interconnection network independent. This is because the parallel execution of a regular loop is reduced to the execution of *cyclic shift operations*. Hence for different interconnection networks, only the cyclic shift algorithm needs to be re-implemented.

Parallel Algorithm:

Two all-node cyclic shifts algorithms for hypercubes were explored (Chapter 7). The Shortest Path Algorithm (Section 7.3) was originally proposed by Johnsson [68] but was not analyzed in detail. In order to avoid link congestion so that better performance can be achieved, Johnsson also proposed another algorithm (we call it *J1*). On the contrary to Johnsson's arguments, we proved that the Shortest Path Algorithm is always better than Algorithm *J1* (Theorem 7.3, 7.5, 7.6, 7.8, Corollary 7.4, 7.7). Although the Shortest Path Algorithm always uses the shortest paths for routing messages, it cannot avoid link congestion totally. To overcome this, we devised the Disjoint Link Algorithm (Section 7.4), with only a small number of extra links being used (Theorem 7.18).

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.

[2] Bill Aiello, Tom Leighton, Bruce Maggs, and Mark Newman. Fast algorithms for bit-serial routing on a hypercube. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 55–64, 1990.

[3] Alexander Aiken and Alexandru Nicolau. Optimal loop parallelization. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, pages 308–317. Atlanta, Georgia, June 1988.

[4] Eugene Albert, Kathleen Knobe, Joan D. Lukas, and Guy L. Steele Jr. Compiling Fortran 8x array features for the Connection Machine computer system. In *ACM unknown yet*, pages 42–56, 1988.

[5] Romas Aleliunas and Arnold L. Rosenberg. On embedding rectangular grids in square grids. *IEEE Transactions on Computers*, C-31(9):907–913, September 1982.

[6] Fran Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the PTRAN analysis system for multiprocessing. In *Proc. First International Conference on Supercomputing*, pages 194–211, Athens, Greece, June 1987.

[7] Frances Allen, Michael Burke, Ron Cytron, Jeanne Ferrante, Wilson Hsieh, and Vivek Sarkar. A framework for determining useful parallelism. In *Proc. Int. Conf. on Supercomputing*, pages 207–215, St. Malo, France, July 4-8 1988.

[8] John R. Allen and Ken Kennedy. PFC: A program to convert Fortran to parallel form. In *Proc. First International Conference on Supercomputing*, pages 186–203, Athens, Greece, June 1987.

[9] Randy Allen and Steve Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *Proc. Conference on Programming Language Design and Implementation*, pages 241–249, Atlanta, Georgia, June 22-24 1988.

[10] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[11] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc., New York, 1989.

[12] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, pages 261–322, September 1989.

[13] Utpal Banerjee. Data dependence in ordinary programs. Master's thesis, Univ. Illinois, Urbana, 1976.

[14] Utpal Banerjee. *Speedup of Ordinary Programs.* PhD thesis, Dep. of Comput. Sci., Univ. Illinois, Urbana, October 1979.

[15] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[16] Utpal Banerjee, Shyh-Ching Chen, David J. Kuck, and Ross A. Towle. Time and parallel processor bounds for Fortran-like loops. *IEEE Transactions on Computers*, pages 660–670, September 1979.

[17] Jarle Berntsen. Communication efficient matrix multiplication on hypercubes. *Parallel Computing*, 12:335–342, 1989.

[18] Sandeep N. Bhatt, Fan R. K. Chung, Jia-Wei Hong, F. Thomason Leighton, and Arnold L. Rosenberg. Optimal simulations by butterfly networks. In *Proc. ACM Symposium on Theory of Computing*, pages 192–204, 1988.

[19] Sandeep N. Bhatt and Ilse C. F. Ipsen. How to embed trees in hypercubes. Technical Report YALEU/DCS/RR-443, Department of Computer Science, Yale University, December 1985.

[20] Thomas Brandes. The importance of direct dependences for automatic parallelization. In *Proc. Int. Conf. on Supercomputing*, pages 407–417, 1988.

[21] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proc. ACM Symp. on Compiler Construction, SIGPLAN Notices,21:7*, pages 162–175, July 1986.

[22] David Callahan, Jack Dongarra, and David Levine. Vectorizing compilers: A test suite and results. Technical Report ANL-88-46, Argonne National Laboratory, November 1988.

[23] R. M. Chamberlain. Gray codes, fast Fourier transforms and hypercubes. *Parallel Computing*, 6:225–233, 1988.

[24] M. Y. Chan. Dilation-2 embeddings of grids into hypercubes. In *Proc. International Conference on Parallel Processing*, pages 295–298, 1988.

[25] M. Y. Chan. Embedding of d-dimensional grids into optimal hypercubes. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 52–57, 1989.

[26] M. Y. Chan and F. Y. L. Chin. On embedding rectangular grids in hypercubes. *IEEE Transactions on Computers*, 37(10):1285–1288, October 1988.

[27] M. Y. Chan and Francis Chin. Parallelized simulation of grids by hypercubes. In *Proceedings of International Computer Symposium 1990, December 17-19, Hsinchu, Taiwan, R.O.C.*, pages 535–544, 1990.

[28] Mee Yee Chan and Shiang-Jen Lee. Distributed fault-tolerant embeddings of rings in hypercubes. *Journal of Parallel and Distributed Computing*, 11:63–71, 1991.

[29] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley Publishing Company, 1988.

[30] Ming-Syan Chen and Kang G. Shin. Processor allocation in an n-cube multiprocessor using gray codes. *IEEE Transactions on Computers*, pages 1396–1407, December 1987.

[31] Woei-Kae Chen and Edward F. Gehringer. A graph-oriented mapping strategy for a hypercube. In *Proc. The Third Conference on Hypercube Concurrent Computers and Applications, vol. 1*, pages 200–209, 1988.

[32] Woei-Kae Chen, Matthias F. M. Stallmann, and Edward F. Gehringer. Hypercube embedding heuristics: An evaluation. *International Journal of Parallel Programming*, 18(6):505–549, 1989.

[33] R. P. Colwell, R. P. Nix, J. J. O'Donnel, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a Trace scheduling compiler. *IEEE Transactions on Computers*, pages 967–979, August 1988.

[34] George Cybenko, David W. Krumme, and K. N. Venkataraman. Fixed hypercube embedding. *Information Processing Letters*, 25:35–39, April 1987.

[35] Robert Cypher and C. Greg Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. In *Proc. ACM Symposium on Theory of Computing*, pages 193–203, 1990.

[36] Ron Cytron. *Compile-Time Scheduling and Optimization for Asynchronous Machines.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1984.

[37] Ron Cytron. Doacross: Beyond vectorization for multiprocessors (extended abstract). In *Proc. International Conference on Parallel Processing*, pages 836–844, 1986.

[38] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[39] Eliezer Dekel, David Nassimi, and Sartaj Sahni. Parallel matrix and graph algorithms. *SIAM Journal on Computing*, 10(4):657–675, November 1981.

[40] Sanjay R. Deshpande and Roy M. Jenevein. Scalability of a binary tree on a hypercube. In *Proc. International Conference on Parallel Processing*, pages 661–668, 1986.

[41] Erik H. D'Hollander. Partitioning and labeling of index sets in Do loops with constant dependence vectors. In *Proc. International Conference on Parallel Processing, Vol. II*, pages 139–144, 1989.

[42] Henry G. Dietz. Finding large-grain parallelism in loops with serial control dependencies. In *Proc. International Conference on Parallel Processing*, pages 114–121, 1988.

[43] Kemal Efe. Embedding mesh of trees in the hypercube. *Journal of Parallel and Distributed Computing*, 11:222–230, 1991.

[44] Zhixi Fang, Pen-Chung Yew, Peiyi Tang, and Chuan-Qi Zhu. Dynamic processor self-scheduling for general parallel nested loops. In *Proc. International Conference on Parallel Processing*, pages 1–10, 1987.

[45] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[46] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[47] Michael J. Flynn. Very high-speed computing systems. *Proceedings of The IEEE*, 54(2):1901–1909, December 1966.

[48] J.A.B. Fortes and F. Parisi-Presicce. Optimal linear schedules for the parallel execution of algorithms. In *Proc. International Conference on Parallel Processing*, pages 322–329. IEEE Computer Society Press, 1984.

[49] Geoffrey C. Fox and Wojtek Furmanski. Communication algorithms for regular convolutions and matrix problems on the hypercube. In *Hypercube Multiprocessors 1987*, pages 223–238. SIAM, 1987.

[50] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, California, 1979.

[51] Milind Girkar and Constantine Polychronopoulos. Compiling issues for supercomputers. In *Proc. Supercomputing'88*, pages 164–173, Orlando, Florida, November 1988.

[52] Milind Girkar and Constantine Polychronopoulos. Partitioning programs for parallel execution. In *Proc. Int. Conf. on Supercomputing*, pages 216–229, St. Malo, France, July 1988.

[53] Allan Gottlieb. An overview of the NYU Ultracomputer project. Technical Report Ultracomputer Note #100, New York University, April 1987.

[54] David S. Greenberg. Minimum expansion embeddings of meshes in hypercubes. Technical Report YALEU/DCS/TR-535, Department of Computer Science, Yale University, August 1987.

[55] David S. Greenberg and Sandeep N. Bhatt. Routing multiple paths in hypercubes. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 45–54, 1990.

[56] Rajiv Gupta. Synchronization and communication costs of loop partitioning on shared-memory multiprocessor systems. In *Proc. International Conference on Parallel Processing, Vol. II*, pages 23–30, 1989.

[57] Malcolm C. Harrison. Synchronous combining of fetch-and-add operations. Technical Report Ultracomputer Note #71, Courant Institute of Mathematical Sciences, 1984.

[58] Malcolm C. Harrison. The add-and-lambda operation: An extension of F&A. Technical Report Ultracomputer Note #104, Courant Institute of Mathematical Sciences, July 1986.

[59] Johan Hastad, Tom Leighton, and Mark Newman. Fast computation using faulty hypercubes. In *Proc. ACM Symposium on Theory of Computing*, pages 251–263, 1989.

[60] John P. Hayes, Trevor N. Mudge, Quentin F. Stout, Stephen Colley, and John Palmer. Architecture of a hypercube supercomputer. In *Proc. International Conference on Parallel Processing*, pages 653–660, 1986.

[61] Richard W. Heuft and Warren D. Little. Improved time and parallel processor bounds for Fortran-like loops. *IEEE Transactions on Computers*, pages 78–81, January 1982.

[62] Ching-Tien Ho. Optimal communication primitives and graph embeddings on hypercubes. Technical Report YALEU/DCS/TR-779, Department of Computer Science, Yale University, March 1990.

[63] Ching-Tien Ho and S. Lennart Johnsson. On the embedding of arbitrary meshes in boolean cubes with expansion two dilation two. In *Proc. International Conference on Parallel Processing*, pages 188–191, 1987.

[64] Ching-Tien Ho and S. Lennart Johnsson. Embedding meshes into small boolean cubes. Technical Report YALEU/DCS/TR-791, Department of Computer Science, Yale University, May 1990.

[65] H. F. Ho, G. H. Chen, S. H. Lin, and J. P. Sheu. Solving linear programming on fixed-size hypercubes. In *Proc. International Conference on Parallel Processing*, pages 112–116, 1988.

[66] Oscar H. Ibarra and Stephen M. Sohn. On mapping systolic algorithms onto the hypercube. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):48–63, January 1990.

[67] Bo Jin and Lan Jin. A new approach to hypercube network analysis. In *Proc. The 9th International Conference on Distributed Computing Systems*, pages 263–268, 1989.

[68] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *Journal of Parallel and Distributed Computing*, 4:133–172, 1987.

[69] Christos Kaklamanis, Danny Krizanc, and Thanasis Tsantilas. Tight bounds for oblivious routing in the hypercube. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 31–36, 1990.

[70] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[71] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3), July 1967.

[72] Howard P. Katseff. Initializing hypercubes. In *Proc. The 9th International Conference on Distributed Computing Systems*, pages 246–253, 1989.

[73] Zvi M. Kedem. Optimal allocation of area for single-chip computations. *SIAM Journal on Computing*, 14(3), August 1985.

[74] Chung-Ta King and Lionel M. Ni. Grouping in nested loops for parallel execution on multicomputers. In *Proc. International Conference on Parallel Processing, Vol. II*, pages 31–38, 1989.

[75] Richard Koch, Tom Leighton, Bruce Maggs, Satish Rao, and Arnold Rosenberg. Work-preserving emulations of fixed-connection networks. In *Proc. ACM Symposium on Theory of Computing*, pages 227–240, 1989.

[76] S. Rao Kosaraju and Mikhail J. Atallah. Optimal simulations between mesh-connected arrays of processors. *Journal of the ACM*, 35(3):635–650, July 1988.

[77] David W. Krumme, K. N. Venkataraman, and George Cybenko. Hypercube embedding is NP-complete. In Michael T. Heath, editor, *Proceedings of the First Conference on Hypercube Multiprocessors*, pages 148–157, Philadephia, 1986. SIAM.

[78] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[79] V. K. Prasanna Kumar and Venkatesh Krishnan. Efficient image template matching on hypercube SIMD arrays. In *Proc. International Conference on Parallel Processing*, pages 765–771, 1987.

[80] H. T. Kung. Why systolic architectures? *IEEE Computer*, pages 37–46, January 1982.

[81] Bradley C. Kuszmaul. Fast, deterministic routing, on hypercubes, using small buffers. *IEEE Transactions on Computers*, 39(11):1390–1393, November 1990.

[82] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, pages 83–93, February 1974.

[83] Duncan H. Lawrie. Access and alignment of data in an array processor. *IEEE Transactions on Computers*, C-24(12):1145–1155, December 1975.

[84] Peizong Lee and Zvi M. Kedem. Mapping nested loop algorithms into multi-dimensional systolic arrays. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):64–76, January 1990.

[85] Tom Leighton, Mark Newman, Abhiram G. Ranade, and Eric Schwabe. Dynamic tree embeddings in butterflies and hypercubes. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 224–234, 1989.

[86] Zhiyuan Li, Pen-Chung Yew, and Chuan-Qi Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, pages 26–34, January 1990.

[87] Björn Lisper. Preconditioning index set transformations for time-optimal affine scheduling. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 360–366, 1990.

[88] Oliver A. McBryan and Eric F. Van de Velde. Matrix and vector operations on hypercube parallel processors. *Parallel Computing*, (5):117–125, 1987.

[89] Rami Melhem and Ghil-Young Hwang. Embedding rectangular grids into square grids with dilation two. *IEEE Transactions on Computers*, 29(12):1446–1455, December 1990.

[90] Samuel P. Midkiff and David A. Padua. Compiler generated synchronization for Do loops. In *Proc. International Conference on Parallel Processing*, pages 544–551, 1986.

[91] D. I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transactions on Computers*, 35(1):1–12, January 1986.

[92] David Nassimi and Sartaj Sahni. Optimal bpc permutations on a cube connected simd computer. *IEEE Transactions on Computers*, C-31(4):338–341, April 1982.

[93] David Nassimi and Yuh-Dong Tsai. Efficient implementations of a class of $\pm 2^b$ parallel computations on a SIMD hypercube. Technical Report CIS-91-10, New Jersey Institute of Technology, 1991.

[94] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, Inc., New York, 1988.

[95] Lionel M. Ni and Chung-Ta King. On partitioning and mapping for hypercube computing. *International Journal of Parallel Programming*, 17(6):475–495, 1988.

[96] Pei Ouyang. Execution of regular DO loops on asynchronous multiprocessors. In *Proc. International Parallel Processing Symposium*, pages 605–610. IEEE Computer Society Press, 1991.

[97] Pei Ouyang and Krishna V. Palem. Very efficient cyclic shifts on hypercubes. In *Proc. Symposium on Parallel and Distributed Processing.* IEEE Computer Society Press, 1991.

[98] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[99] Krishna V. Palem and Barbara B. Simons. Scheduling time-critical instructions on RISC machines. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 270–280, 1990.

[100] Jih-Kwon Peir and Ron Cytron. Minimum distance: A method for partitioning recurrences for multiprocessors. In *Proc. International Conference on Parallel Processing*, pages 217–225, 1987.

[101] C. Greg Plaxton. Load balancing, selection and sorting on the hypercube. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 64–73, 1989.

[102] Constantine D. Polychronopoulos. Compiler optimizations for enhancing parallelism and their impact on architecture design. *IEEE Transactions on Computers*, 27(8):991–1004, August 1988.

[103] Constantine D. Polychronopoulos and Utpal Banerjee. Processor allocation for horizontal and vertical parallelism and related speedup. *IEEE Transactions on Computers*, 36(4):410–420, April 1987.

[104] Constantine D. Polychronopoulos, David J. Kuck, and David A. Padua. Execution of parallel loops on parallel processor systems. In *Proc. International Conference on Parallel Processing*, pages 519–527, 1986.

[105] Patrice Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proc. The 11th Annual International Symposium on Computer Architecture*, pages 208–214. IEEE Computer Society Press, 1984.

[106] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2), April 1989.

[107] Abhiram G. Ranade. How to emulate shared memory. In *Proc. Symposium on Foundations of Computer Science*, pages 185–194, October 1987.

[108] Sanjay Ranka and Sartaj Sahni. Odd even shifts in SIMD hypercubes. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):77–82, January 1990.

[109] Daniel A. Reed and Richard M. Fujimoto. *Multicomputer Networks: Message-Based Parallel Processing.* Scientific Computation Series. The MIT Press, Cambridge, Massachusetts, 1987.

[110] John H. Reif and Robert E. Tarjan. Symbolic program analysis in almost-linear time. *SIAM Journal on Computing*, pages 81–93, February 1981.

[111] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms.* Prentice-Hall, Englewood Cliffs, N.J., 1977.

[112] V. P. Roychowdhury and T. Kailath. Study of parallelism in regular iterative algorithms. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 367–376, 1990.

[113] Youcef Saad and Martin H. Schultz. Topological properties of hypercubes. *IEEE Transactions on Computers*, 37(7):867–872, July 1988.

[114] Joel H. Saltz, Ravi Mirchandaney, and Doug Baxter. Run-time parallelization and scheduling of loops. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 303–312, 1989.

[115] Vijay A. Saraswat. Concurrent constraint programming. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 232–245, 1990.

[116] Vivek Sarkar and John Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Proc. ACM Symp. on Compiler Construction,* in *SIGPLAN Notices, 21:7*, pages 17–26, July 1986.

[117] Eric J. Schwabe. On the computational equivalence of hypercube-derived networks. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 388–397, 1990.

[118] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, pages 484–521, October 1980.

[119] David S. Scott and Joe Brandenburg. Minimal mesh embeddings in binary hypercubes. *IEEE Transactions on Computers*, 37(10):1284–1285, October 1988.

[120] Charles L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[121] Weijia Shang and Jose A. B. Fortes. Independent partitioning of algorithms with uniform dependencies. In *Proc. International Conference on Parallel Processing*, pages 26–33, 1988.

[122] Weijia Shang and Jose A. B. Fortes. Time optimal linear schedules for algorithms with uniform dependencies. *IEEE Transactions on Computers*, 40(6):723–742, June 1991.

[123] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An empirical study on array subscripts and data dependencies. In *Proc. International Conference on Parallel Processing, Vol. II*, pages 145–152, 1989.

[124] J.-P. Sheu and T.-H. Tai. Partitioning and mapping nested loops on multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):430–439, October 1991.

[125] Robert Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, October 1981.

[126] Harold S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, pages 85–93, January 1977.

[127] Harold S. Stone. *High-Performance Computer Architecture.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

[128] Ted Szymanski. On the permutation capability of a circuit-switched hypercube. In *Proc. International Conference on Parallel Processing, Vol. I*, pages 103–110, 1989.

[129] Y. Tanaka, K. Iwasawa, S. Gotoo, and Y. Umetani. Compiling techniques for first-order linear recurrences on a vector computer. In *Proc. Supercomputing'88*, pages 174–181, Orlando, Florida, November 1988.

[130] Don Towsley. Allocating programs containing branches and loops within a multiple processor system. *IEEE Transactions on Software Engineering*, pages 1018–1024, October 1986.

[131] Akao Tsuda and Yoshitoshi Kunieda. V-Pascal: an automatic vectorizing compiler for Pascal with no language extensions. In *Proc. Supercomputing'88*, pages 182–189, Orlando, Florida, November 1988.

[132] Jeffrey D. Ullman. Chapter 4: Complexity of sequencing problems. In E. G. Coffman, Jr., editor, *Computer and Job-Shop Scheduling Theory*, pages 139–164. John Wiley & Sons, 1975.

[133] Jeffrey D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1984.

[134] Eli Upfal. An $O(\log N)$ deterministic packet routing scheme. In *Proc. ACM Symposium on Theory of Computing*, pages 241–250, 1988.

[135] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proc. ACM Symposium on Theory of Computing*, pages 263–277, 1981.

[136] Ravi Varadarajan. Embedding shuffle networks in hypercubes. *Journal of Parallel and Distributed Computing*, 11:252–256, 1991.

[137] Jean Vuilemin. A combinatorial limit to the computing power of VLSI circuits. *IEEE Transactions on Computers*, C-32(3), March 1983.

[138] Alan Shelton Wagner. *Embedding Trees in the Hypercube.* PhD thesis, Department of Computer Science, University of Toronto, October 1987.

[139] David R. Wallace. Dependence of multi-dimensional array references. In *Proc. Int. Conf. on Supercomputing*, pages 418–428, St. Malo, France, July 4-8 1988.

[140] Michael Weiss, C. Robert Morgan, and Zhixi Fang. Dynamic scheduling and memory management for parallel programs. In *Proc. International Conference on Parallel Processing*, pages 161–165, 1988.

[141] Michael Wolfe. Advanced loop interchanging. In *Proc. International Conference on Parallel Processing*, pages 536–543, 1986.

[142] Michael Wolfe. Vector optimization vs. vectorization. In *Proc. First International Conference on Supercomputing*, pages 309–315, Athens, Greece, June 1987.

[143] Amr Zaky and P. Sadayappan. Optimal static scheduling of sequential loops on multiprocessors. In *Proc. International Conference on Parallel Processing, Vol. III*, pages 130–136, 1989.

[144] Stavros A. Zenios and Robert A. Lasken. The Connection Machines CM-1 and CM-2: Solving nonlinear network problems. In *Proc. Int. Conf. on Supercomputing*, pages 648–658, St. Malo, France, July 4-8 1988.