INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Some problems in parallel and distributed computing

Kim, Young Man, Ph.D.
The Ohio State University, 1992
SOME PROBLEMS IN PARALLEL AND DISTRIBUTED COMPUTING

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By

Young Man Kim, B.S., M.S.

* * * * *

The Ohio State University

1992

Dissertation Committee:

Professor Ten Hwang Lai
Professor Ming T. Liu
Professor D. N. Jayasimha

Approved by

Adviser
Department of Computer
and Information Science
ACKNOWLEDGMENTS

I am in deep appreciation of my adviser Professor Ten Hwang Lai for introducing me into the research area of Parallel and Distributed Computing and for guiding me while this work is being done. I am also grateful to Professors Ming T. Liu and D. N. Jayasimha for agreeing to serve as members of my reading committee and for their proper comments.

My many thanks are due to Professor Kyungsook Y. Lee and Professor Dik L. Lee for their guidances and countless enlightening ideas in the research areas of Computer and Communication Network and Information Retrieval System, respectively. Many thanks also go to my colleagues, Dr. Wen-Shyen E. Chen, Dr. Hyunsoo Yoon, Yu-Chee Tseng, and Gaurav Patel for their helpful discussions and comments.

My parents and all the family members deserve the greatest merits and special thanks. They have been always very supportive during my graduate study at The Ohio State University. Abundant credits should go to my wife Seon Ae, whose patience and understanding have made this work possible.

I would also want to acknowledge the good supports I have received from the staff members of the CIS Department. My study in The Ohio State University was really fruitful and pleasant.
VITA

March 1, 1958 ....................................................... Born: Seoul, Korea

March 1976 - Feb. 1980 ......................................... B.S., Department of Mechanical Engineering
Seoul National University, Seoul, Korea

March 1980 - Aug. 1982 ........................................ M.S., Department of Mechanical Engineering
Korea Advanced Institute of Science and Technology, Seoul, Korea

March 1982 - Feb. 1983 .......................................... Consultant
KOPTEC, Seoul, Korea

March 1983 - Feb. 1986 ........................................ Senior Research & Development Engineer, Production Technology Laboratory
Gold Star Co., Changwon, Korea

Sept. 1987 - Dec. 1988 ....................................... M.S., Department of Computer and Information Science
The Ohio State University, Columbus, Ohio

Sept. 1987 - present .......................................... Graduate Teaching/Research Associate, Department of Computer and Information Science
The Ohio State University, Columbus, Ohio
Publications


A Modular High-Speed Switching Network for Integration of Heterogeneous Computing Resources. *Proceedings of the 20th International Conference on Parallel Processing*, 1991. (co-authors: W.-S. E. Chen and M.T. Liu)


Performance Analysis of Buffered Banyan Networks under Nonuniform Traffic. *Proc-


Fields of Study

Major Field: Computer and Information Science

Studies in:

Parallel and Distributed Computing: Prof. Ten Hwang Lai
Computer Architecture: Prof. Kyungsook Y. Lee
Programming Language: Prof. Neelam Soundararajan
Database System: Prof. Dik Lun Lee
# Table of Contents

ACKNOWLEDGMENTS ................................................................. ii

VITA .................................................................................................. iii

LIST OF TABLES ................................................................................ ix

LIST OF FIGURES .............................................................................. x

CHAPTER PAGE

I  Introduction .......................................................................................... 1
  1.1 Background and Motivation ................................................................. 1
  1.2 Problem Descriptions ................................................................. 3
    1.2.1 The Grid Iteration Problem ......................................................... 3
    1.2.2 The Deadlock Detection and Resolution Problem ...................... 4
    1.2.3 The Embedding Problem in a Hypercube ...................................... 5
    1.2.4 The Processor Compaction Problem in a Subcube ...................... 5
  1.3 Organization of the Study ................................................................. 6

II  Pseudo-Optimal Partitioning Structure in a Grid Iteration Problem .... 7
  2.1 Overview ......................................................................................... 7
  2.2 Unit Communication Polygons for Stencils ....................................... 11
    2.2.1 Derivation of a Unit Communication Polygon for a Stencil ......... 15
    2.2.2 Pseudo-Optimal Partitioning Problem ......................................... 19
  2.3 Pseudo-Optimal Partitioning Structures for UCPs ........................... 20
  2.4 Partition Dependency Graph .......................................................... 35
  2.5 Pseudo-Suboptimal Partitioning Structure ......................................... 36
  2.6 Application to a Directed Acyclic Graph ......................................... 40
  2.7 Concluding Remarks ......................................................................... 40
### III A Wait-Free Algorithm for Deadlock Detection and Resolution

- **3.1 Overview** ................................................................. 42
- **3.2 Model of a Resource-Sharing Distributed Database System** ................. 44
  - **3.2.1 Distributed Database System** ......................................... 44
  - **3.2.2 Proposed Model for Distributed Database System** .................. 46
  - **3.2.3 Problem Definition** .................................................. 48
- **3.3 Distributed Deadlock Detection and Resolution Algorithm** .................... 49
  - **3.3.1 Overview** ............................................................. 49
  - **3.3.2 Formal Description** ................................................ 51
- **3.4 Proof Method** .................................................................. 58
  - **3.4.1 A Distributed Computational Model** .................................. 58
  - **3.4.2 Proof System** ........................................................ 59
  - **3.4.3 Justification of Sequential Model** ..................................... 61
- **3.5 Correctness Proof** .......................................................... 70
  - **3.5.1 Basic Properties** ..................................................... 70
  - **3.5.2 Proof of Safety Condition** ......................................... 85
  - **3.5.3 Proof of Progress Condition** ....................................... 90
- **3.6 Performance Analysis** .................................................... 96
- **3.7 Concluding Remarks** .................................................... 97

### IV The Complexity of Congestion-1 Embedding in a Hypercube

- **4.1 Overview** ................................................................. 99
- **4.2 Terminology and Definitions** .......................................... 102
- **4.3 Embedding a General Graph** ......................................... 105
  - **4.3.1 Upper Bounds** ..................................................... 105
  - **4.3.2 NP-Completeness of Congestion-1 Embedding** .................. 113
- **4.4 Embedding a Connected Graph** ....................................... 119
  - **4.4.1 Construction** ..................................................... 119
  - **4.4.2 Constructing an Embedding from a 3-Partition** ................. 126
  - **4.4.3 Extracting a 3-Partition from an Embedding** ................... 131
- **4.5 Proof of Main Theorem** ................................................. 135
- **4.6 Concluding Remarks** .................................................... 150

### V Compacting Free Buddy Subcubes in a Hypercube

- **5.1 Overview** ................................................................. 151
**List of Tables**

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>34</td>
</tr>
</tbody>
</table>

Communication overheads for square, hexagon, and the pseudo-optimal partitioning structure for five common stencils.
# List of Figures

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Typical task domain and 5-point stencil</td>
</tr>
<tr>
<td>2</td>
<td>Communication overhead of a line segment</td>
</tr>
<tr>
<td>3</td>
<td>How to derive a UCP from a stencil.</td>
</tr>
<tr>
<td>4</td>
<td>Five common stencils and their UCPs.</td>
</tr>
<tr>
<td>5</td>
<td>Chain(5) and End-Edge(Chain(5)).</td>
</tr>
<tr>
<td>6</td>
<td>Two cases for the proof of Lemma 1.</td>
</tr>
<tr>
<td>7</td>
<td>Repeated use of Lemma 1 for the proof of Lemma 2.</td>
</tr>
<tr>
<td>8</td>
<td>Diagrams for Definition 4.</td>
</tr>
<tr>
<td>9</td>
<td>Diagram for Case 2 of the proof of Lemma 3.</td>
</tr>
<tr>
<td>10</td>
<td>Diagrams for the proof of Lemma 4.</td>
</tr>
<tr>
<td>11</td>
<td>Some UCP's with their FDP's and DP's.</td>
</tr>
<tr>
<td>12</td>
<td>Construction of a $DP(G, A')$ for Lemma 5.</td>
</tr>
<tr>
<td>13</td>
<td>Diagram for Lemma 6 and Lemma 7</td>
</tr>
<tr>
<td>14</td>
<td>Definition of $vertex-unit(FDP, u_i)$ in a UCP.</td>
</tr>
<tr>
<td>15</td>
<td>$\alpha, \beta, x, \text{ and } y$ for $f(u)$.</td>
</tr>
<tr>
<td>16</td>
<td>Pseudo-optimal partitioning structures ($Popt$) for five common stencils</td>
</tr>
<tr>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>---------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>18</td>
<td>Typical multiprocessor communication networks</td>
</tr>
<tr>
<td>19</td>
<td>Diagrams for Lemma 8 and Lemma 9</td>
</tr>
<tr>
<td>20</td>
<td>Pseudo-optimal partitioning structure for diamond DAG</td>
</tr>
<tr>
<td>21</td>
<td>Time diagram 1.</td>
</tr>
<tr>
<td>22</td>
<td>Time diagram 2.</td>
</tr>
<tr>
<td>23</td>
<td>Time diagram 3.</td>
</tr>
<tr>
<td>24</td>
<td>State transition diagram of edge ((v,w))</td>
</tr>
<tr>
<td>25</td>
<td>Example paths: (f(u,v)) and (f(x,y))</td>
</tr>
<tr>
<td>26</td>
<td>(a) Skeleton graph (G_{m,b}). (b) Its image under a congestion-1 embedding.</td>
</tr>
<tr>
<td>27</td>
<td>The structure of (G)</td>
</tr>
<tr>
<td>28</td>
<td>Embedding of linear chains and glue edges</td>
</tr>
<tr>
<td>29</td>
<td>Square waves.</td>
</tr>
<tr>
<td>30</td>
<td>Part of embedding (f : G_{m,b} \rightarrow H_n)</td>
</tr>
<tr>
<td>31</td>
<td>(a) (P_{ij}) and its vicinity in (G_{m,b}). (b) The corresponding subgraph in (H_n)</td>
</tr>
<tr>
<td>32</td>
<td>A job allocation scenario in an (H_3).</td>
</tr>
<tr>
<td>33</td>
<td>The procedure (Match()).</td>
</tr>
</tbody>
</table>
CHAPTER I

Introduction

1.1 Background and Motivation

Advances in semiconductor technology and interconnection and communication technology have made parallel and distributed computing feasible and attractive. Parallel and distributed systems are known to possess several advantages over conventional ones: reasonable cost, high computational performance, resource sharing, improved utility, fault-tolerance, and adaptability to geometrically dispersed organization.

When a problem is given to be solved in a parallel/distributed system, from designing a program solving the problem to executing the program in the system, there are four phases to pass through: parallel program design, program verification, program implementation, and resource allocation.

When we have a problem to solve in a parallel/distributed system, we first design a parallel program that can solve the problem. This work comprises the first phase. An important objective in this phase is program performance, in addition to the other conventional ones: compactness, readability, portability, modifiability, etc. Because typical inter-process communication primitives are much more expensive than any other computational primitives, one of the important issues in performance improve-
ment is to find a program partitioning method minimizing the amount of interprocess communications for a given problem.

To check if the parallel program designed in the first phase meets its problem specification is important and nontrivial especially when the program is as complex as many real-world practical programs. The second phase, program verification, has thus received much attention from many researchers. To verify a program, we need a sound theoretical tool called *formal proof system*. There is a broad spectrum of formal proof systems proposed in the literature. However, when we verify a real-world complex program using such a well-defined proof system, it is almost impossible to finish the correctness proof within a reasonable amount of time and space.

When we are to execute a program, which is designed to work on a system (source system), on the other system (target system), we need to adapt the program suitable to the target system; this is the task of the third phase, program implementation.

Because the interconnection network of the source system may be different from that of the target system, one of the fundamental issues in this phase is *embedding a source graph* representing the interprocess communications of the program into a *host graph* representing the interconnection network of the target system.

Once a program is implemented into a target system, we allocate system resources, a set of free processors and a portion of the interconnection network that connects the allocated processors, to the program and execute it —the fourth, and final, phase. Because identifying available resources suitable to the request from the arriving job is nontrivial, designing a job allocation scheme is an important issue in this phase.
To maximize the resource utility, sometimes we reconfigure the system by a sequence of job migrations so that the system can allocate some resources to the arriving jobs.

We study four current issues, one in each phase, in this dissertation: *program performance, efficient verification, program embedding, and job migration*. In particular, we select a problem from each issue and solve it. Next section introduces these four problems.

1.2 Problem Descriptions

1.2.1 The Grid Iteration Problem

In the first phase, parallel program design, designing an efficient parallel program is important. Since inter-process primitives are much more time-consuming than any computational primitives, reducing the amount of inter-process messages in the program is frequently the key factor determining its program efficiency. We will study this issue occurring in a *grid iteration problem* that is explained in the below.

Many important scientific problems [58, 77] (e.g., elliptic partial differential equation, finite element method, molecular dynamics, lattice gauge theory, and crystallization simulation) may be transformed to *grid iteration problems* that are solved by an iterative technique on a grid. When we solve a grid iteration problem using a multiprocessor, the communication overhead becomes the major factor determining the program performance.

We will design a method to derive from a given stencil a pseudo-optimal partitioning structure that minimizes the communication overhead under two approximations which simplify the problem characteristics.
1.2.2 The Deadlock Detection and Resolution Problem

In the program verification phase, reducing the amount of steps needed to verify the program correctness is an important issue when the program is complex. We will study this issue in the proof of a distributed deadlock detection and resolution algorithm.

Distributed deadlock detection is a hard problem. Many published distributed deadlock detection algorithms turn out to be incorrect [44, 73]. The main reason for such frequent errors is that they arrived at hasty conclusions without checking the algorithm correctness against every possible situation occurring from program computations.

The complexity of distributed deadlock algorithms is often so high that it is almost impossible to apply conventional formal proof systems. As a result, most researchers either just skipped the correctness proof [74] or resorted to informal schemes such as informal operational proof [53, 57, 65] or simulation [17]. Since these methods easily skip some situations in the proof, they frequently arrived at the false conclusions.

In this study, first an efficient wait-free edge-chasing algorithm is proposed. Then, we propose a simple computational model and its proof system that reduces the effort needed for the verification significantly. Based on our proof system, we present the correctness proof of the proposed algorithm.
1.2.3 The Embedding Problem in a Hypercube

In the next phase, program implementation, we study embedding problem in a hypercube. Embedding a source graph in a host graph has long been used to model the problem of processor allocation in a distributed system, where a source graph represents a distributed algorithm, with nodes representing component processes and edges representing interprocess communications, and a host graph represents a network of processors. In particular, we focus our attention on the computational complexity of the graph embedding schemes in a circuit-switching hypercube.

1.2.4 The Processor Compaction Problem in a Subcube

In the resource allocation phase, job migration is one of the basic schemes used to increase resource utility. Thus, we study this issue in a circuit-switching hypercube.

A hypercube can be partitioned into subcubes of various sizes to run independent jobs. As jobs arrive, grabbing the subcubes, and leave, releasing the subcubes, the system tends to become fragmented. When this happens, one solution that has been proposed is to relocate (or migrate) jobs so as to compact free processors into bigger subcubes. During the process of compaction, it is desirable that each migration step be effective: structure-preserving, adjacency-preserving, path-disjoint, congestion-free, and contention-free.

In this study, an effective algorithm for subcube compaction is proposed for a hypercube system that uses the buddy-system allocation strategy and the circuit-switching communication model.
1.3 Organization of the Study

The organization of this dissertation is as follows.

In Chapter II, a grid partitioning problem is delivered. In particular, a scheme deriving a pseudo-optimal partitioning structure is proposed.

In Chapter III, we address the deadlock detection and resolution problem. We present an efficient deadlock detection and resolution algorithm. A simple computational model and its proof system is proposed. A justification of the proposed computational model is presented. Then, we present the correctness proof of the proposed algorithm.

In Chapter IV, the embedding problem in a hypercube is addressed.

In Chapter V, the processor compaction scheme is studied. A compaction scheme is proposed which generates a migration schedule having several desirable properties. A parallel version of this scheme is also proposed.

Finally, Chapter VI concludes and summarizes the results of this dissertation. Possible future research directions/issues are also suggested.
CHAPTER II

Pseudo-Optimal Partitioning Structure in a Grid Iteration Problem

2.1 Overview

A problem transformation has long been a very successful solution paradigm [64]. In this paradigm, elliptic partial differential equations that govern many scientific problems can easily be solved in the discrete domain. One such method for an elliptic partial differential equation is Gauss-Seidel or Successive Over Relaxation (SOR) method [58]. First, a problem domain for a partial differential equation is discretized into a grid. Then, the elliptic partial differential equation is also discretized to a difference equation relating neighboring grid nodes. Finally, the difference equation is iteratively solved over all grid nodes until it converges.

The data dependency among neighboring grid nodes can be represented graphically by a stencil which shows the data dependence between the center node and its neighboring nodes by edges. At the beginning of each iteration, every node in the grid requires data from its neighboring nodes as specified in the stencil to start the computation.

For example, Fig. 1 shows the 5-point stencil in which one center node is connected to four neighboring nodes: upper, lower, left, and right nodes. The computation at
In each iteration of the value at the node \((i,j)\), the four neighboring node values in \((i,j-1)\), \((i,j+1)\), \((i-1,j)\), and \((i+1,j)\) are required.

Figure 1: Typical task domain and 5-point stencil

each node needs four data from the four neighboring nodes for each iteration.

The problem consisting of a grid and a difference equation is called a grid iteration problem. In addition to elliptic partial differential equations, many other problems in science and engineering can be transformed to or represented as a grid iteration problem. For example, finite element computation, molecular dynamics, lattice gauge theory, and crystallization simulation can be transformed into a grid iteration problem [77].

When a grid iteration problem is solved on a multiprocessor [25, 26], the grid is divided into partitions each of which is allocated to and executed by one processor. Since the partitioning inevitably divides some neighboring nodes into different parti-
tions, the need arises for processors to communicate with each other. Unfortunately, the cost of communication between processors is much more expensive than that of computation in a processor regardless of the machine architecture. Thus the total number of data to be communicated per iteration for a partition is a major factor for the overall system performance [2, 24, 27, 30].

Many researchers have studied the effects of problem size, partitioning size, partitioning structure, and architecture on the performance of a parallel system. Fox and Otto [25, 26] pointed out that the ratio of the computational load to the communication overhead is critical to the efficiency of parallel systems. Vrsalovic et al. [80] discussed the solution of Poisson's equation over a square region using the five-point stencil. Cvetanovic [19] considered the effects of partitioning, allocation, and granularity on the machine performance. Reed et al. [64] showed that nonstandard partitioning structures (e.g. hexagon) are frequently preferable to the standard square in reducing the ratio of communication to computation. Nicol et al. [55] studied the speedup factors in various architectures due to different partitioning structures and problem sizes, and Tang and Li [77] studied the optimal granularity to maximize a particular performance criterion. In this chapter, we present a procedure to determine the pseudo-optimal partitioning structure for a given stencil which minimizes the ratio of communication to computation under two approximations which simplify the problem characteristics.

In general, the data dependency in an algorithm can be graphically represented as a Directed Acyclic Graph (DAG) in which a node represents a computational task
and a directed edge represents the data dependency between two tasks. Since most computationally intensive algorithms have loops, their DAGs tend to have a regular pattern. When a multiprocessor is used to execute a DAG, a partitioning problem similar to that in a grid iteration problem, called a DAG partitioning problem [39, 60, 63, 84], arises. Our procedure for grid iteration problems can also be applied to a DAG partitioning problem to derive a pseudo-optimal partitioning structure which minimizes the communication overhead.

Given a stencil or a DAG and its pseudo-optimal partitioning structure, a Partition Dependency Graph (PDG) describing the data dependency between neighboring partitions can be constructed. Once a PDG is obtained, it needs to be mapped to a host graph which models the processor network. A dilation-$k$ embedding is a mapping of a PDG to a host graph in which an edge in the PDG is mapped into a path in the host graph whose length is less than or equal to $k$. A congestion-$l$ embedding is a mapping of a PDG to a host graph in which an edge in the host graph is shared by at most $l$ edges in the PDG.

For a circuit-switched system, an embedding with congestion-1 and dilation-$k$ is desirable since the communication speed depends on the link contention, but not on the length of a communication path. For a packet-switched system, an embedding with congestion-1 and dilation-1 is desirable since the communication speed depends on both the link contention and the length of a communication path. The mapping of a PDG to a host graph is also studied in this chapter.

The remainder of this chapter is organized as follows. Section 2.2 defines Unit
Communication Polygon (UCP) and presents a procedure which derives a UCP from a given stencil. Section 2.3 studies the relationship between a partitioning structure and the communication overhead, and presents a procedure that constructs the pseudo-optimal partitioning structure given a UCP. The mapping of a PDG to a multiprocessor is considered in Section 2.4, and the pseudo-suboptimal partitioning structures are discussed in Section 2.5. Section 2.6 shows how the procedure for the pseudo-optimal partitioning structure can be applied to the DAG partitioning problem. In Section 2.7, concluding remarks are given.

2.2 Unit Communication Polygons for Stencils

Given a 2-dimensional problem domain of \( N \) grid nodes and \( p \) processors, a balanced partitioning divides the \( N \) grid nodes into \( p \) partitions of \( \frac{N}{p} \) grid nodes distributing the computational load evenly over \( p \) processors. Our objective is to find an efficient balanced partitioning which minimizes the required data communications among partitions. The communication overhead for the partition is proportional to the number of nodes along the boundary of the partition. If every partition can be surrounded by a common geometrical figure, the partitioning is uniform and the geometrical figure is called the governing figure of the partition. As an example, the partitioning of the grid for nine processors in Fig. 1 is uniform and the governing figure is a square.

In this chapter we consider uniform partitioning of grids with polygons as governing figures. The number of nodes in a partition (computational load) is proportional to the area of its governing polygon and the communication overhead for the partition is proportional to the perimeter of its governing polygon.
Figure 2: Communication overhead of a line segment

(a) effect of length

(b) effect of the neighboring line segments

(c) effect of angle

(d) effect of angle
The number of data crossing a boundary line segment in the governing polygon that surrounds a partition depends on the length, location, and the angle of the line segment, as well as the neighboring line segments connected to the line segment.

In Fig. 2(a), the communication overhead of a line segment $\overline{AB}$ is 16 by counting the data passing across $\overline{AB}$ (two data per edge crossing $\overline{AB}$). When the location of $\overline{AB}$ is slightly shifted to the right or left, the communication overhead changes to 14 since one boundary edge is removed from the set of edges crossing $\overline{AB}$. For the effect of neighboring line segments of $\overline{AB}$, see Fig. 2(b). Two data exchanges along the leftmost edge are counted when $\overline{AC}$ is a neighboring line segment of $\overline{AB}$ in the governing polygon; on the other hand, they are not counted if $\overline{AD}$ is the neighboring line segment of the governing polygon. Figs. 2(c) and (d) show the effect of the angle of a line segment on the communication overhead; a horizontal line segment in Fig. 2(c) exchanges 16 data across itself and a line segment of angle 45° whose length is $\sqrt{2}$ times longer than the horizontal line exchanges 17 data across itself. (Note that there are 9 nodes below the line segment and 8 nodes above it so that 17 data are communicated across the line segment.)

However, we observe that all these factors except the length and angle of the line segment in a governing polygon are restricted to the limited region around two vertices of the line segment so that, when the length of the line segment increases, the effect reduces rapidly in comparison to the total communication overhead across the line segment. Thus we use the notion of average. $\text{av-leng-per-comm}(\alpha)$ is the
average length of a line segment of angle \( \alpha \) through which one datum passes. Let

\[
\text{av-leng-per-comm}(\alpha) \overset{\text{def}}{=} \lim_{|u| \to \infty} \frac{|u|}{\text{Comm}(u)}
\]

where \( u \), \( |u| \), and \( \text{Comm}(u) \) are a line segment of angle \( \alpha \), length of \( u \), and the number of data to be communicated over \( u \), respectively. \( \text{av-leng-per-comm}(\alpha) \) can be easily derived from the geometry of a grid and its stencil. For example, with the 5-point stencil shown in Fig. 2(c), \( \text{av-leng-per-comm}(0) = \text{av-leng-per-comm}(\frac{\pi}{2}) = 0.5 \) and \( \text{av-leng-per-comm}(\frac{\pi}{4}) = 0.5\sqrt{2} \).

Suppose that \( \text{av-leng-per-comm}(\alpha) \), \( 0 \leq \alpha \leq 2\pi \), is given. Then, for any line segment \( u \), \( \text{Comm}(u) \) can be approximated as \( \frac{|u|}{\text{av-leng-per-comm}(\text{ang}(u))} \) where \( \text{ang}(u) \) is the angle of \( u \). When \( |u| \) increases, \( \text{av-leng-per-comm}(u) \) rapidly converges to a constant dependent exclusively on \( \text{ang}(u) \). Thus, the longer \( u \) is, the better the estimation for \( \text{Comm}(u) \) will be.

Given a governing polygon \( P \), the number of nodes (computational load) within \( P \) is dependent on the location of \( P \) in grid and the area of \( P \), \( \text{area}(P) \). Average number of nodes per unit area of \( P \), \( \text{av-comp}(P) \), is

\[
\text{av-comp}(P) \overset{\text{def}}{=} \lim_{\text{area}(P) \to \infty} \frac{\text{number of nodes within } P}{\text{area}(P)}.
\]

As \( \text{area}(P) \) increases, the effect of the location of \( P \) on \( \text{av-comp}(P) \) rapidly decreases, and \( \text{av-comp}(P) \) converges to a constant, \( \text{av-comp} \). For a grid with the unit distance 1, the computational load of \( P \), \( \text{Comp}(P) \), is

\[
\text{Comp}(P) \cong \text{area}(P) \cdot \text{av-comp}(P) = \text{area}(P).
\]
In summary, the following two approximations are used which simplify the problem characteristics.

**Approximation 1** Given a governing polygon $P$ which covers a partition $Q$ and consists of $k$ line segments, $u_1, \ldots, u_k$, the communication overhead of $Q$, $Comm(Q)$, is

$$Comm(Q) \approx Comm(P) \approx \sum_{i=1}^{k} Comm(u_i).$$

**Approximation 2** A polygon $P$ contains $Comp(P)$ grid nodes (computational load) that is equal to $area(P)$.

### 2.2.1 Derivation of a Unit Communication Polygon for a Stencil

A *Communication Polygon* ($CP$) $G$ is a symmetric polygon with the property that $Comm(u)$ is a constant, $CP-Comm(G)$, for any line segment $u$ connecting the center point and a boundary point of the polygon. A *Unit Communication Polygon* (UCP) $G$ is a CP such that $CP-Comm(G) = 1.0$. In other words, for any line segment $u$ connecting the center and a boundary point of a UCP, $av-leng-per-comm(ang(u)) = |u|$. Note that, given a stencil, all CPs for that stencil are similar to each other. Moreover, for two CPs for a governing stencil, $P_1$ and $P_2$,$$rac{CP-Comm(P_1)}{CP-Comm(P_2)} = \frac{\sqrt{area(P_1)}}{\sqrt{area(P_2)}}.$$

In the following we present a procedure to derive UCP from a stencil. A stencil consists of one center and its neighbors connected by edges denoting the data dependencies. Let $C$ and $N_i, 1 \leq i \leq n$, denote the center and its $n$ neighbors of a stencil.
such that $N_i$ is the $i$-th neighbor in the counterclockwise direction starting from the positive horizon. Fig. 3(a) shows $C$ and $N_i, 1 \leq i \leq 4$, of the 5-point stencil.

For a neighbor $N$, let $\text{ang}(N)$ and $\text{dist}(N)$ denote $\text{ang}(CN)$ and $|CN|$, respectively. Also let $\text{dist}(N, u)$ denote the distance between $N$ and a line segment $u$ passing through center $C$. Then $\text{dist}(N, u)$ equals $\text{dist}(N) \cdot \sin(\text{ang}(N) - \text{ang}(u))$. For a line segment $u$, let $m$-$\text{neighbor}(u)$ be a neighbor $N$ such that $\text{dist}(N, u)$ is maximum among all neighbors. For example, in Fig. 3(a), $|N_2A|$ is the $\text{dist}(N_2, u)$, and $N_2$ and $N_4$ are $m$-$\text{neighbor}(u)$.

To examine the implication of an $m$-$\text{neighbor}(u)$ for line $u$, draw another line $u'$ which is incident on $m$-$\text{neighbor}(u)$ and $u'\parallel u$ as shown in Fig. 3(a). With $u$ and $u'$, three regions, Regions 1, 2, and 3 in Fig. 3(b), are created. Any nodes located between lines $u$ and $u'$ (Region 2), e.g. $D$, $E$, and $F$, have their communication partners in Regions 1 and 3. In other words, all nodes in Region 2 communicate across line $u$, while no nodes in Region 3 communicate across line $u$. Thus, for an $m$-$\text{neighbor}(u)$ $N$,

$$\text{av-leng-per-comm}(\text{ang}(u)) = \lim_{|u| \to \infty} \frac{|u|}{2 \cdot (\text{number of nodes in Region 2})} = \frac{1}{2 \cdot \text{dist}(N, u)}.$$

Given a stencil, an $m$-$\text{neighbor}(u)$ remains fixed as the $\text{ang}(u)$ is changed within a certain range. Let $m$ denote minimum number of ranges where $m$-$\text{neighbor}(u)$ remains fixed. For example, in Fig. 3(a), $N_2$ is $m$-$\text{neighbor}(u)$ for $-\pi/4 \leq \text{ang}(u) \leq \pi/4$ and $m = 4$. Let $M$ denote $\{(M_i, l_i, h_i) : 1 \leq i \leq m, M_i$ is an $m$-$\text{neighbor}(u)$ from angle $l_i$ to angle $h_i\}$. Then, a UCP consists of a sequence of segments, $(u_1, \ldots, u_m)$, such that $u_i$ is a locus for point $L$ for which
Figure 3: How to derive a UCP from a stencil.
$l_i \leq \text{ang}(CL) \leq h_i$ and $|CL| = \frac{1}{2 \cdot \text{dist}(M,u)}$. In the following, we present a scheme to derive such a locus for a given element in $M$.

Let $(N, \alpha_1, \alpha_2)$ be an element in $M$. First, we draw a point $D$ for which $\text{ang}(CD) = \text{ang}(N) - \pi/2$ and $|CD| = \frac{1}{2 \cdot \text{dist}(N,CD)}$. For example, in Fig. 3(c) with neighbor $N = N_2$, $\text{ang}(CD) = \text{ang}(N_2) - \pi/2 = 0$ and $|CD| = \frac{1}{2 \cdot \text{dist}(N_2,CD)} = \frac{1}{2 \cdot \text{dist}(N_2,\sin(\text{ang}(N_2)-\text{ang}(CD)))} = 0.5$. Next, draw a line $v$ incident on point $D$ such that $v$ is parallel to $CN$ as shown in Fig. 3(d). This line $v$ contains the desired locus since for an arbitrary point $L$ that is incident on the locus,

$$|CL| = \frac{1}{2 \cdot \text{dist}(N) \cdot \sin(\text{ang}(N) - \text{ang}(CL))}$$

$$= \frac{1}{2 \cdot \text{dist}(N) \cdot \cos(\text{ang}(CL) - (\text{ang}(N) - \pi/2))}$$

$$= |CD| \cdot \frac{1}{\cos(\text{ang}(CL) - \text{ang}(CD))}.$$  

For example, with $N = N_2$, $\text{ang}(N_2) = \pi/2$, $\text{ang}(CD) = 0$, and $|CL| \cdot \cos(\text{ang}(CL)) = |CD|$ as shown in Fig. 3(c) and 3(d).

Let $E$ and $F$ be two points incident on $v$ such that $\text{ang}(CE)$ and $\text{ang}(CF)$ are $\alpha_1$ and $\alpha_2$, respectively. Then, the locus under consideration is $EF$. For example, with $N = N_2$, $\alpha_1 = -\pi/4$ and $\alpha_2 = \pi/4$, thus, $|CE| = |CF| = |CD| \cdot \sqrt{2}$ and $EF$ is a part of the UCP for the 5-point stencil for the range $-\pi/4 \leq \text{ang}(CL) \leq \pi/4$ as shown in Fig. 3(d).

The UCPs for the five common stencils have been derived by this procedure and are shown in Fig. 4.
Figure 4: Five common stencils and their UCPs.

2.2.2 Pseudo-Optimal Partitioning Problem

Once a stencil is replaced with a corresponding UCP under Approximations 1 & 2, the Pseudo-optimal partitioning problem can be formulated as follows.

(Pseudo-Optimal Partitioning Problem)

Given a UCP with area $A$ that reflects the communication characteristics of a stencil and the computational load per processor, find a polygon $P$ such that

$$Comm(P) = \min_{G \in Set(A)} \{Comm(G)\},$$

where $Set(A) = \{G : G$ is a polygon with area $A\}$

As can be seen in the next section, for a given stencil, its pseudo-optimal partitioning structures are geometrically similar to each other. Thus, if a pseudo-optimal
partitioning structure $P$ of area $A$ is known for a certain stencil, the pseudo-optimal structure $P'$ of area $A'$ for the stencil can be obtained simply by scaling $P$ so that $area(P') = A'$.

2.3 Pseudo-Optimal Partitioning Structures for UCPs

In this section, some basic properties of a UCP and the necessary conditions which a pseudo-optimal partitioning structure must satisfy are derived. It is then shown that there is a unique pseudo-optimal partitioning structure satisfying such conditions. A UCP is assumed to be a convex symmetric polygon. The key lemmas in this section are Lemma 5 and Lemma 7.

**Definition 1** A partitioning structure of area $A$ is $\text{Po}^{\text{opt}}(A)$ if it has the smallest communication overhead among all polygons of area $A$.

**Definition 2** A figure is a $\text{Chain}(k)$ if it consists of $k$ line segments connected in sequence. For a $\text{Chain}(k)$, $u$, $\text{End-Edge}(u)$ is a line segment connecting two end vertices of $u$.

A $\text{Chain}(k)$ can be represented by $v_1 \cdots v_{k+1}$ where $v_i$ is the $i$-th vertex along the chain, $1 \leq i \leq k+1$. The communication overhead of a $\text{Chain}(k)$, $\text{Comm}(\text{Chain}(k))$, is thus $\sum_{i=1}^{k} \text{Comm}(v_i v_{i+1})$. See Fig. 5 for a $\text{Chain}(5)$ and $\text{End-Edge}(\text{Chain}(5))$.

**Lemma 1** In a UCP, $\text{Comm}(\text{Chain}(2)) \leq \text{Comm}(\text{End-Edge}(\text{Chain}(2)))$.

**Proof.** Consider a $\text{Chain}(2)$, $u = \overline{ABC}$, in a UCP, $G$. A Communication Polygon $P$ similar to $G$ can be constructed such that its center is at $A$ and $C$ is on its boundary as
Figure 5: Chain(5) and End-Edge(Chain(5)).

Figure 6: Two cases for the proof of Lemma 1.

Figure 7: Repeated use of Lemma 1 for the proof of Lemma 2.
in Fig. 6(a). Let $B'$ be the intersection between the boundary of $P$ and the extended line segment of $AB$. There are two possible cases:

**Case 1:** $B$ is located on the boundary of $P$. Then, $\text{Comm}(ABC) \geq \text{Comm}(AB) \geq \text{Comm}(AB') = \text{Comm}(AC) = \text{Comm}(\text{End-Edge}(u))$.

**Case 2:** $B$ is located inside $P$. Then, draw another polygon $P'$ which is similar to $P$ and its center is at $B$ and $B'$ is on its boundary. Then $P'$ is located inside of $P$ by the geometric similarity between $P$ and $P'$, and the fact that $P > P'$. Thus,

\[
\text{Comm}(ABC) = \text{Comm}(AB) + \text{Comm}(BC) = \text{Comm}(AB') + \text{Comm}(BC) - \text{Comm}(BB') = \text{Comm}(AC) + \text{Comm}(BC) - \text{Comm}(BB') \geq \text{Comm}(AC),
\]

since $\text{Comm}(BC) \geq \text{Comm}(BB')$ by the fact that $C$ is on or outside of boundary of $P'$.

\[\square\]

**Lemma 2** In a UCP, $\text{Comm}(\text{Chain}(k)) \leq \text{Comm}(\text{End-Edge}(\text{Chain}(k)))$, $2 \leq k$.

**Proof.** Consider a $\text{Chain}(k)$, $u = \overline{v_1 \cdots v_{k+1}}$. For $k = 2$, Lemma 2 is proved by Lemma 1. For $k > 2$, using Lemma 1 $(k-1)$ times as shown in Fig. 7,

\[
\text{Comm}(\overline{v_1 \cdots v_{k+1}}) \geq \text{Comm}(\overline{v_1 v_3 v_4 \cdots v_{k+1}}) \geq \cdots \geq \text{Comm}(\overline{v_1 v_{k+1}}) = \text{Comm}(\text{End-Edge}(u)).
\]

\[\square\]

**Definition 3** Consider a Communication Polygon $G$ with the center at $C$. The following definitions are depicted graphically in Fig. 8.

(a) A **unit** is a line segment connecting $C$ and a boundary point of $G$. Let the boundary point of a unit $u$ be $\text{Bound}(u)$. If $\text{Bound}(u)$ is a vertex of $G$, $u$ is a **vertex-unit**.
(b) Consider a unit $u$. $\text{Next}(u, C)$ and $\text{Next}(u, CC)$ are neighboring vertex-units of $u$ in clockwise and counterclockwise directions, respectively.

(c) Consider $u$ which is a unit but not a vertex-unit. $\text{Comm}-\text{Edge}(u)$ is a \text{Chain}(2), $\overline{v_1v_2v_3}$, such that $v_1 = C$, $v_3 = \text{Bound}(u)$, $\overline{v_1v_2} \parallel \text{Next}(u, C)$, and $\overline{v_2v_3} \parallel \text{Next}(u, CC)$. If $u$ is a vertex-unit, then $\text{Comm}-\text{Edge}(u) = u$.

(d) Consider a vertex-unit $u$. $\text{Alt-Comm-Edge}(u)$ is a \text{Chain}(2), $\overline{v_1v_2v_3}$, such that $v_1 = C$, $v_3 = \text{Bound}(u)$, $\overline{v_1v_2} \parallel \text{Next}(u, C)$, and $\overline{v_2v_3} \parallel \text{Next}(u, CC)$.

(e) Consider a vertex-unit $u$. $\text{ang}(u, C)$ and $\text{ang}(u, CC)$ are angles between $u$ and two neighboring vertex-units, $\text{Next}(u, C)$ and $\text{Next}(u, CC)$, respectively.

**Lemma 3** For a unit $u$ of a UCP, $\text{Comm}(u) = \text{Comm}(\text{Comm}-\text{Edge}(u))$.

**Proof.** There are two cases.

\textit{Case 1:} $u$ is a vertex-unit. By definition, $\text{Comm}-\text{Edge}(u) = u$. Thus, $\text{Comm}(u) = \text{Comm}(\text{Comm}-\text{Edge}(u))$.

\textit{Case 2:} $u$ is not a vertex-unit. Consider such a $u = \overline{CD}$ in Fig. 9. $\overline{CG} = \text{Next}(u, CC)$, $\overline{CF} = \text{Next}(u, C)$, and $\overline{CED} = \text{Comm}-\text{Edge}(u)$. Then, $\text{Comm}(\text{Comm}-\text{Edge}(u)) = \text{Comm}(\overline{CED}) = \text{Comm}(\overline{CE}) + \text{Comm}(\overline{ED})$. Since the triangles $\triangle EFD$ and $\triangle CFG$ are similar, $\text{Comm}(\overline{ED}) = \frac{\overline{EF}}{\overline{CF}} \cdot \text{Comm}(\overline{CG}) = \frac{\overline{EF}}{\overline{CF}} \cdot \text{Comm}(\overline{CF}) = \text{Comm}(\overline{EF})$. Thus, $\text{Comm}(\text{Comm}-\text{Edge}(u)) = \text{Comm}(\overline{CE}) + \text{Comm}(\overline{EF}) = \text{Comm}(\overline{CF}) = \text{Comm}(\overline{CD}) = \text{Comm}(u)$. □

**Lemma 4** For a vertex-unit $u$ of a UCP, $\text{Comm}(u) < \text{Comm}(\text{Alt-Comm-Edge}(u))$. 
(a) Unit and vertex-unit $u_1$.
(b) Next($u, CC$) and Next($u, C$).

(c) Comm-Edge($u$)
(d) Alt-Comm-Edge of a vertex-unit $u$.
(e) $\text{ang}(u, C)$ and $\text{ang}(u, CC)$ of a vertex-unit $u$.

Figure 8: Diagrams for Definition 4.
Figure 9: Diagram for Case 2 of the proof of Lemma 3.

Figure 10: Diagrams for the proof of Lemma 4.

Figure 11: Some UCP’s with their FDP’s and DP’s.
**Proof.** Consider a vertex-unit \( u = \overline{CD} \) in Fig. 10. \( \overline{CA} \) and \( \overline{CB} \) are \( \text{Next}(u, C) \) and \( \text{Next}(u, CC) \) respectively, and \( \overline{CED} \) is \( \text{Alt-Comm-Edge}(u) \). Depending upon the shape of the UCP, there are two cases.

**Case 1:** \( |\overline{CA}| \leq |\overline{CE}| \). Then, \( \text{Comm}(\text{Alt-Comm-Edge}(u)) = \text{Comm}(\overline{CED}) = \text{Comm}(\overline{CE}) + \text{Comm}(\overline{ED}) \geq \text{Comm}(\overline{CA}) + \text{Comm}(\overline{ED}) = \text{Comm}(\overline{CD}) + \text{Comm}(\overline{ED}) > \text{Comm}(u) \).

**Case 2:** \( |\overline{CA}| > |\overline{CE}| \). Consider another polygon \( P' \) similar to the given UCP \( P \) such that its center is at \( E \) and \( A \) is a boundary point of \( P' \). \( \overline{AFG} \) is a part of the boundary of \( P' \) where \( \overline{EF} \parallel \overline{CD} \) and \( \overline{GF} \parallel \overline{BD} \). Since \( P' < P \) and \( P' \) is included in \( P \) as shown in Fig. 10(b), \( \text{Comm}(\text{Alt-Comm-Edge}(u)) = \text{Comm}(\overline{CED}) = \text{Comm}(\overline{CE}) + \text{Comm}(\overline{ED}) = \text{Comm}(\overline{CE}) + \text{Comm}(\overline{EA}) + \text{Comm}(\overline{GD}) = \text{Comm}(\overline{CD}) + \text{Comm}(\overline{GD}) > \text{Comm}(u) \). \( \square \)

**Definition 4** For a polygon \( P \) of \( k \) vertices, \( \text{Edge}(k) \) denotes the boundary \( \overline{v_1 \cdots v_{k+1}} \) where \( v_1 = v_{k+1} \) and \( v_1, \ldots, v_k \) are vertices of \( P \) in the counterclockwise order.

**Definition 5** Consider a UCP \( G \) of \( \text{Edge}(k) \) with area \( A \).

(a) A polygon \( P \) is **Diagonal Polygon** (DP) of \( G \) of area \( A \), \( \text{DP}(G, A) \), if \( P \) is convex, \( \text{Area}(P) = A \), and for any line segment \( u_1 \) in the boundary of \( P \), there is a vertex-unit \( u_2 \) in \( G \) such that \( u_1 \parallel u_2 \).

(b) A polygon \( P \) is **Fully Diagonal Polygon** (FDP) of \( G \) of area \( A \), \( \text{FDP}(G, A) \), if \( P \) is a \( \text{DP}(G, A) \) with \( \text{Edge}(k) \).

As an example, some UCP's are shown together with their FDP's and DP's in Fig. 11.
Lemma 5 For a UCP G and area A, there exists a polygon P which is a DP(G, A) and $P^{opt}(A)$.

Proof. Consider a UCP G and an arbitrary polygon Q of area A that is not a $DP(G, A)$ as in Fig. 12. For each vertex-unit $u_1$ of G, construct $u_2$ outside of Q such that $u_2 \parallel u_1$ and $u_2$ is long enough to cover Q. With $u_2$ do the plane sweep over Q, finding the points of Q where $u_2$ is tangent to Q. At these points draw the tangent lines as a part of the polygon being constructed. Note that such a plane sweep over a polygon creates at most two tangent lines. When this process is complete for all vertex-units of G, a polygon R is constructed which encloses Q and consists of edges parallel to vertex-units of G. Such an R is a $DP(G, A')$ for some $A' > A$. For example, in Fig. 12(b), R is $A_1A_2\cdots A_6A_1$. 

Figure 12: Construction of a $DP(G, A')$ for Lemma 5.
Construct another polygon $S$ whose vertices are intersections between $Q$ and $R$.

For example, in Fig. 12(b), $S = B_1B_2\cdots B_6B_1$.

Then, by applying Lemma 3 repeatedly, it can be shown that $Comm(Q) \geq Comm(S)$. On the other hand, by Lemma 2, $Comm(S) = Comm(R)$. Thus, $Comm(Q) \geq Comm(R)$. Now, reduce $R$ to $P$ such that $area(P) = A$ and $P$ is similar to $R$. Then, $P$ is a $DP(G, A)$ and $Comm(P) \leq Comm(R) \leq Comm(Q)$. In other words, for any polygon $Q$ of area $A$, a $DP(G, A)$ $P$ can be constructed which has less communication overhead than $Q$. Therefore, a pseudo-optimal polygon $P^{opt}(A)$ must be a $DP(G, A)$.

**Lemma 6** For a UCP $G$ and area $A$, there exists a polygon $P$ which is a $FDP(G, A)$ and $P^{opt}(A)$.

**Proof.** Consider a UCP $G$ in Fig. 13(a). Suppose that a pseudo-optimal partitioning polygon $P$ for $G$ is not an $FDP(G, A)$. Without losing generality, assume that a line segment parallel to vertex-unit $CA_2$ does not appear in $P$ disqualifying $P$ as an $FDP(G, A)$. For example, Fig. 13(b) shows a part of $P$, $abde$, without a line segment parallel to $CA_2$, such that $ab$ and $bd$ are parallel to vertex-units $CA_1$ and $CA_3$ of $G$, respectively. Now, another partitioning polygon $P'$ of area $A$ is defined such that $P'$ is equal to $P$ except that $P'$ has $afhide$ as a part of boundary in place of $abde$ where $fh||CA_2$, $hi||CA_3$, $id||CA_4$, and $g$ is an intersection of $bd$ and $fh$. Since $P$ is pseudo-optimal,

$$Comm(abde) \leq Comm(afhide).$$

(2.1)
Since $bd \parallel hi$ and $jh \parallel di$, $fbg$ and $ghj$ are $Alt-Comm-Edge(fg)$ and $Alt-Comm-Edge(gj)$, respectively.

Thus, $Comm(Alt-Comm-Edge(fg)) + Comm(gj) \leq Comm(fg) + Comm(Alt-Comm-Edge(gj))$.

By Lemma 4, $Comm(Alt-Comm-Edge(fg)) > Comm(fg)$ and $Comm(Alt-Comm-Edge(gj)) > Comm(gj)$. Thus, there exist two positive constants, $k_1$ and $k_2$, such that $Comm(Alt-Comm-Edge(fg)) - Comm(fg) = k_1 \cdot |fg|$ and $Comm(Alt-Comm-Edge(gj)) - Comm(gj) = k_2 \cdot |gj|$. Thus,

$$\frac{|fg|}{|gj|} < \frac{k_2}{k_1}$$ (2.2)
Since \( \text{area}(P) = \text{area}(P') \), \( \text{area}(\overline{fbgf}) = \text{area}(\overline{ghidg}) \), \( \text{area}(\overline{fbgf}) = |\overline{fg}| \cdot (0.5|\overline{bg}| \sin \alpha) = 0.5 \frac{\sin \alpha \cdot \sin \beta}{\sin(\alpha + \beta)} \cdot (|\overline{fg}|)^2 = k_3 \cdot (|\overline{fg}|)^2 \) where \( \alpha, \beta, \) and \( \gamma \) are \( \text{ang}(\overline{CA_2}, CC) \), \( \text{ang}(\overline{CA_4}, CC) \), and \( \text{ang}(\overline{CA_3}, CC) \), respectively. For a small size of \( |\overline{bd}| \), \( |\overline{bd}| < 0.5|\overline{bd}| \), \( \text{area}(\overline{ghidg}) > 0.5|\overline{bd}| \cdot (|\overline{gh}| \cdot \sin \alpha) = 0.5 \frac{\sin \alpha \cdot \sin(\alpha + \beta)}{\sin \beta} \cdot |\overline{gh}| \cdot |\overline{bd}| = k_4 \cdot |\overline{gh}| \cdot |\overline{bd}| \).

Thus,

\[
\frac{|\overline{fg}|}{|\overline{gj}|} > k_4 \cdot \frac{|\overline{bd}|}{|\overline{fg}|} = k_5
\] (2.3)

By combining Eq. 2.2 and Eq. 2.3,

\[
\frac{k_5}{|\overline{fg}|} < \frac{|\overline{fg}|}{|\overline{gj}|} \leq \frac{k_2}{k_1}
\] (2.4)

When \( |\overline{fg}| \to 0 \), this equation turns out to be false since \( \frac{k_5}{|\overline{fg}|} \to \infty \) while at the same time upperbounded by a constant \( \frac{k_2}{k_1} \). Thus, our assumption is contradictory, and the pseudo-optimal partitioning polygon \( P \) must be an \( FDP(G, A) \).

**Definition 6** Suppose that \( G \) is a UCP of \( \text{Edge}(k) \) and a polygon \( P \) is an \( FDP(G, A) \). Then, for \( u_i = \overline{v_iv_{i+1}} \) of \( P \), \( 1 \leq i \leq k \), \( \text{vertex-unit}(FDP, u_i) \) is the vertex-unit of \( G \) which is parallel to \( u_i \).
See Fig. 14 for an example of a vertex-unit \((FDP, u_i)\).

**Definition 7** For a vertex-unit \(u\) in a UCP, let \(\alpha\) and \(\beta\) be \(\text{ang}(u, C)\) and \(\text{ang}(u, CC)\), respectively. Also, let \(x\) be \(\text{Next}(u, C)\) and \(y\) be \(\text{Next}(u, CC)\). Then, \(f(u) = \frac{\sin \alpha}{|x|} - \frac{\sin(\alpha + \beta)}{|y|}\).

See Fig. 15 for the relation among \(u, x, y, \alpha, \text{ and } \beta\).

**Lemma 7** Suppose that \(G\) is a UCP of \(\text{Edge}(k)\) and a polygon \(P\) is an \(\text{FDP}(G,A)\) and \(P_{opt}(A)\) for area \(A\). Let \(u\) and \(v\) represent vertex-unit \((FDP, u_i)\) and vertex-unit \((FDP, u_{i+1})\), respectively. Then, for \(i\), \(1 \leq i \leq k - 1\),

\[
\frac{|u_i|}{|u_{i+1}|} = \frac{\sin(\text{ang}(v, CC)) \cdot f(u)}{\sin(\text{ang}(u,C)) \cdot f(v)}
\]

**Proof.** Consider the UCP \(G\) in Fig. 13(a). Lemma 6 guarantees the existance of \(P_{opt}(A)\) \(P\) which is an \(\text{FDP}(G,A)\). Suppose that Chain(4), \(\overline{fabcg}\) in Fig. 13(c), defines a part of \(P\). Draw a new polygon \(P'\) of area \(A\) that is equal to \(P\) except that the chain \(\overline{fa'b'c'g} (\text{where } a'b'\parallel ab \text{ and } b'c'\parallel bc)\) instead of \(\overline{fabcg}\) defines the boundary of \(P'\). Let \(\overline{db'}\parallel aa'\) and \(\overline{eb'}\parallel c'c\).
Let $\alpha$, $\beta$ and $\gamma$ within $G$ denote $\text{ang}(CA_2, CC)$, $\text{ang}(CA_1, CC)$, and $\text{ang}(CA_3, CC)$, respectively. Then, by a geometric property, $\text{area}(aa'b'\beta''a) = \text{area}(\beta''b'c'c'').$ Since $\text{area}(aa'b'\beta''a) = 0.5(|ab'| + |a'b'|) \cdot b'b' \cdot \sin \alpha = 0.5(2|ab| - 2|b''| - |b'b'| \cdot \cos \alpha \cdot \left(1 + \frac{\tan \alpha}{\tan \beta}\right)) \cdot |b'b'| \cdot \sin \alpha$, and $\text{area}(\beta''b'c'c'') = 0.5(2|bc| + |b''| \cdot \frac{\sin \beta}{\sin (\alpha + \beta)} \cdot |b'b'| \cdot \sin \alpha$,

$$\lim_{|b'b''| \to 0} \frac{|b''|}{|b'b'|} = \lim_{|b'b''| \to 0} \frac{2|ab| - 2|b''| - |b'b'| \cdot \cos \alpha \cdot \left(1 + \frac{\tan \alpha}{\tan \beta}\right)}{2|bc| + |b''| \cdot \frac{\sin \beta}{\sin (\alpha + \beta)}}$$

$$= \frac{|ab|}{|bc|}.$$

Since $P$ is a pseudo-optimal partitioning structure, $\text{Comm}(fabcg) \leq \text{Comm}(fabc'g)$. Using the property that $\text{Comm}(ad) = \text{Comm}(a'b')$ and $\text{Comm}(ec') = \text{Comm}(bc)$, this equation is simplified as $\text{Comm}(db'b''e) \geq \text{Comm}(db''be)$. From geometric properties, $|db'| = |b'b''| \cdot \frac{\sin \alpha}{\sin \beta}$, $|db''| = |b'b''| \cdot (\cos \alpha + \cos \beta \cdot \frac{\sin \alpha}{\sin \beta})$, $|eb| = |eb''| \cdot \frac{\sin \alpha}{\sin \gamma}$, and $|eb''| = |b'b''| \cdot (\cos \alpha + \cos \gamma \cdot \frac{\sin \alpha}{\sin \gamma})$. Thus,

$$\frac{|b'b''|}{|b'b'|} \leq \frac{\sin \gamma \cdot f(u)}{\sin \beta \cdot f(v)}.$$ Combining the above two relations,

$$\frac{|ab|}{|bc|} \leq \frac{\sin \gamma \cdot f(u)}{\sin \beta \cdot f(v)}. \quad (2.5)$$

Draw another new polygon $P''$ of area $A$ that is equal to $P$ except that $fabc'g$ in Fig. 13(d) instead of $fabcg$ defines new boundary of $P''$. Applying the above procedure to $P''$,

$$\frac{|ab|}{|bc|} \geq \frac{\sin \gamma \cdot f(u)}{\sin \beta \cdot f(v)}. \quad (2.6)$$

The lemma can be shown to be true by combining Eq. 2.5 and Eq. 2.6. \qed


**Figure 16:** Pseudo-optimal partitioning structures ($P_{opt}$) for five common stencils

**(Procedure Pseudo-Optimal-Partitioning)**

Given a stencil and a computational load $A$, the following procedure generates a polygon $P$ that is $P_{opt}(A)$.

(a) Derive a polygon $G$ of $Edge(k)$ which is a UCP for the given stencil by the procedure described in Section 2.

(b) Choose any vertex-unit $u$ of $G$. Draw a line segment $u_1$ of unit length such that $u_1 \parallel u$. Initialize $i$ to 1.

(c) Compute $|u_{i+1}|$ from $|u_i|$ using Lemma 7 which defines the ratio of $|u_i|$ to $|u_{i+1}|$.

(d) Draw $u_{i+1}$ at the end point of $u_i$ such that $u_{i+1} \parallel Next(u, CC)$ ensuring that $P$ will be a convex polygon when done.
Table 1: Communication overheads for square, hexagon, and the pseudo-optimal partitioning structure for five common stencils.

<table>
<thead>
<tr>
<th></th>
<th>5-point</th>
<th>9-point</th>
<th>7-point</th>
<th>9-cross</th>
<th>13-cross</th>
</tr>
</thead>
<tbody>
<tr>
<td>square Comm</td>
<td>4√A</td>
<td>4√A</td>
<td>4√A</td>
<td>8√A</td>
<td>8√A</td>
</tr>
<tr>
<td>hexagon Comm</td>
<td>3√A</td>
<td>5√A</td>
<td>4√A</td>
<td>6√A</td>
<td>6√A</td>
</tr>
<tr>
<td>pseudo-optimal partitioning structure Popt diamond</td>
<td>square</td>
<td>skewed hexagon</td>
<td>diamond</td>
<td>diamond</td>
<td></td>
</tr>
<tr>
<td>Comm</td>
<td>2√2√A</td>
<td>4√A</td>
<td>2√3√A</td>
<td>4√2√A</td>
<td>4√2√A</td>
</tr>
<tr>
<td>R1</td>
<td>29.3%</td>
<td>0%</td>
<td>13.4%</td>
<td>29.3%</td>
<td>29.3%</td>
</tr>
<tr>
<td>R2</td>
<td>5.7%</td>
<td>20.0%</td>
<td>13.4%</td>
<td>5.7%</td>
<td>5.7%</td>
</tr>
</tbody>
</table>

Comm: communication overhead, A: computational load, N/p, where N: total number of nodes in grid, p: number of processors, R1: reduction rate of Comm for Popt compared to square, R2: reduction rate of Comm for Popt compared to hexagon.

(e) If i ≤ k, then increase i by one, set u to Next(u, CC), and go back to step (c), else, P has been constructed.

**Theorem 1** The polygon P generated from the Procedure Pseudo-Optimal-Partitioning is a pseudo-optimal partitioning structure.

**Proof.** Lemma 6 guarantees the existence of a polygon that is an FDP(G, A) and Popt(A). A polygon that is an FDP(G, A) and satisfies the condition in Lemma 7 is uniquely determined by the geometric property, and P generated from the above procedure is such a polygon. □

For the five stencils of interest, the pseudo-optimal partitioning structures have been obtained from the procedure and are shown in Fig. 16.
Table 1 shows the communication overheads for three different partitions: square, hexagon, and $P^{opt}$, for the five stencils. The dimensions of the hexagon are the same as those in [64] which recommended hexagon and square as partitioning structures yielding the optimal communication overhead. Note that $P^{opt}$ is square only for the 9-point stencil. $R_1$ and $R_2$ indicate improvement in communication overhead with $P^{opt}$ in place of square and hexagon, respectively.

2.4 Partition Dependency Graph

Once a pseudo-optimal partitioning structure for a stencil, $P^{opt}$, is derived by the procedure in the previous section, the grid can be partitioned according to $P^{opt}$, and the Partition Dependency Graph (PDG) can be obtained. In a PDG, each node represents a partition and an edge represents the data dependency between any pair of partitions. Much work has been done on assigning or embedding a PDG into a multiprocessor system [7, 18, 43, 78].

If the target multiprocessor system is a store-and-forward message passing system, e.g. iPSC hypercube [56], the main objective in mapping is the dilation-1 embedding [43] where each edge in PDG is mapped into one physical link in the multiprocessor system. Such an embedding avoids the delay through intermediate processors along the routing path. If target system implements a circuit switched communication, e.g. iPSC/2 [56], multi-dilation embedding is almost as efficient as dilation-1 embedding since the distance between two processors makes no difference in the data transmission time once the circuit is established. Another important point in processor assignment is that communication sequences for different processors should generate few link
The PDGs derived from the $P_{opt}$ partitioning have the square-mesh structure for the 5-point, 9-point, 9-cross, and 13-point stencils as shown in Fig. 16. Thus, they can be embedded with dilation-1 into hypercube or a square-mesh multiprocessor shown in Fig. 17(a) and (b), respectively.

The PDG for the 7-point stencil has the hexa-mesh structure fitting best the hexa-mesh network shown in Fig. 17(c). On an iPSC/2 hypercube it can be embedded with dilation-2. It also does not create any link contention with a proper communication sequence, e.g. the communication sequence in the order of directions $d_1$, $d_2$, and $d_3$ in Fig. 16 in each iteration.

### 2.5 Pseudo-Suboptimal Partitioning Structure

If the PDG obtained from $P_{opt}$ does not map efficiently on the communication network in a multiprocessor system, a pseudo-suboptimal partitioning structure that yields a different PDG might result in better overall performance. For example, a square as a
partitioning structure for the 7-point stencil requires a smaller communication time than a hexagon if the underlying processor network is square-mesh.

In this subsection, we derive a pseudo-suboptimal partitioning structure ($P^{sopt}$) which is a symmetric quadrangle for a UCP of hexagon.

**Definition 8** A quadrangle $P$ of area $A$ is $P^{sopt}(A)$ if it results in minimal communication overhead among all symmetric quadrangles of area $A$.

**Definition 9** A symmetric quadrangle $P$ is represented as $(P_a, P_b)$ such that $P_a$ and $P_b$ are neighboring boundary lines of $P$ and $Comm(P_a) \leq Comm(P_b)$. For a UCP $G$ with the computational load $A$, a symmetric quadrangle $P$ is a Diagonal Quadrangle, $DQ(G,A)$, if $Area(P) = A$, and $P$ is a $DP(G,A)$.

**Lemma 8** If a quadrangle $P$ is $P^{sopt}(A)$, then $Comm(P_a) = Comm(P_b)$, for a UCP $G$ with the computational load $A$. 

![Diagrams for Lemma 8 and Lemma 9](image-url)
**Proof.** Suppose that $\text{Comm}(P_a) < \text{Comm}(P_b)$. If not, the lemma has been proved. Fig. 18(a) shows such a $P = ABCDA$ with $P_a = CD$ and $P_b = CB$. Upon $P$, embed $Q$ which is a $CP$ with $C$ as the center and $D$ on the boundary of $Q$. Since $\text{Comm}(P_b) > \text{Comm}(P_a)$, $B$ is outside of $Q$. Now, draw a new quadrangle $R = A'B'C'D'A' = (B'C', C'D')$ such that $\text{Comm}(R_a) = \text{Comm}(R_b) = \frac{\text{Comm}(P_a) + \text{Comm}(P_b)}{2}$. Then, $\text{area}(R) = \text{area}(P) \cdot \frac{\text{Comm}(P_a) + \text{Comm}(P_b)}{2\text{Comm}(P_a)} \cdot \frac{\text{Comm}(P_a) + \text{Comm}(P_b)}{2\text{Comm}(P_b)} > \text{area}(P)$ since $(\text{Comm}(P_a) + \text{Comm}(P_b))^2 > 4\text{Comm}(P_a) \cdot \text{Comm}(P_b)$ when $\text{Comm}(P_a) < \text{Comm}(P_b)$. On the other hand, $\text{Comm}(R) = \text{Comm}(P)$. Thus, there exists $S$ that is similar to $R$, $\text{Comm}(S) < \text{Comm}(P)$, and $\text{area}(S) = \text{area}(P)$. That implies $P$ is not the pseudo-suboptimal quadrangle. □

**Lemma 9** For a UCP $G$ and a computational load $A$, there exists a quadrangle $P$ in $DQ(G, A)$ which is $P^{\text{sopt}}(A)$. 
Proof. Suppose that $P$ is $P_{sopt}(A)$. Then, from Lemma 8, $Comm(P_a) = Comm(P_b)$. Suppose that $P$ is $\overline{CDAB}$ where $P_a$ and $P_b$ are $\overline{CD}$ and $\overline{CB}$ respectively as in Fig. 18(b). Draw a polygon $Q$ which is a $CP$ with $C$ as its center and $D$ and $B$ on the boundary of $P$. Assume that $P_a$ is not a vertex-unit as in Fig. 18(b). Then, there exists a quadrangle $R = (R_a, R_b(= P_b))$ such that $area(P) \leq area(R)$ and $R_a = Next(\overline{CD}, C)$ or $Next(\overline{CD}, CC)$. In Fig. 18(b), $\overline{CE}$ and $\overline{CEIBC}$ correspond to $R_a$ and $R$, respectively. Repeating this process one more time on $R$, we obtain another quadrangle $S$, $S = (S_a(= R_a), S_b)$ such that $area(R) \leq area(S)$ and $S_b = Next(\overline{CB}, C)$ or $Next(\overline{CB}, CC)$. In Fig. 18(b), $\overline{CH}$ and $\overline{CEJHC}$ correspond to $S_b$ and $S$, respectively. Thus, $area(S) \geq area(P)$ and $Comm(S) = Comm(P)$. Therefore, there exists a quadrangle in $DQ(G, A)$ which is $P_{sopt}(A)$. □

Definition 10 For a UCP $G$ with the computational load $A$, a quadrangle $P$ in $DQ(G, A)$ is principal if

$$Comm(P)\eta(P) = \min_{Q \in DQ(G, A)} (Comm(Q))$$

.

Theorem 2 For a UCP $G$ with the computational load $A$, a principal quadrangle $P$ of area $A$ is $P_{sopt}(A)$.

Proof. From Lemma 9, there always exists a quadrangle in $DQ(G, A)$ that is $P_{sopt}$. Suppose $P$ is principal. Since $P$ has the least communication overhead among all possible quadrangles in $DQ(G, A)$, $P$ is $P_{sopt}$. □
2.6 Application to a Directed Acyclic Graph

A Directed Acyclic Graph (DAG) is used to represent the data dependency among individual computation units in an algorithm or a program [60, 63, 84]. Our method for finding a pseudo-optimal partitioning structure can be applied to a DAG with a regular pattern.

A diamond DAG [60] shown in Fig. 19(a) has regular data dependency among nodes. A node in a diamond DAG is dependent on two neighboring nodes at the level below. We have derived the UCP and $P^{opt}$ for the diamond DAG and shown in Fig. 19(b) and (c), respectively. A hexagon structure turns out to be the pseudo-optimal partitioning structure for the diamond DAG rather than a diamond itself as was indicated in [60].

2.7 Concluding Remarks

We have defined a new concept of Unit Communication Polygon (UCP) which can be obtained from a discretization stencil expressing the data dependency among the grid nodes. A procedure for the construction of a UCP from a given stencil has been described. We have also presented a procedure which determines a pseudo-optimal partitioning structure of the problem domain for a given UCP. A pseudo-optimal partitioning structure incurs the minimal communication overhead. While rectangles and hexagons have earlier been recommended as partitioning structures for five common stencils, the actual pseudo-optimal structures turn out to be diamonds, square, and skewed hexagon.
Once a pseudo-optimal partitioning structure is derived, the grid is transformed into a Partition Dependency Graph (PDG). The assignment or embedding of PDG into a multiprocessor system has been considered.

The method developed in this chapter can also be used for the DAG assignment problem [60, 63, 84]. A DAG represents data dependency among computational units in a program. If the data dependency in a DAG is regular, the proposed method can be applied to a DAG so that a pseudo-optimal partitioning structure with the minimal data communications among the partitions can be obtained.

When a pseudo-optimal partitioning structure does not map efficiently on a particular processor network, a pseudo-suboptimal partitioning which match the processor network better may be preferred. We have shown how a pseudo-suboptimal partitioning structure can be obtained among quadrangles when the pseudo-optimal structure is a hexagon.
CHAPTER III

A Wait-Free Algorithm for Deadlock Detection and Resolution

3.1 Overview

Distributed deadlock detection is a hard problem. Many published distributed deadlock detection algorithms turned out to be incorrect [44, 73]. The main reason for such frequent errors is that they arrived at the hasty conclusion without checking the algorithm correctness against every possible situation occurring from executing the algorithms. Several researchers [44, 47, 48, 73] thus concluded that a strictly rigorous proof is required for people to believe in the correctness of any proposed deadlock detection algorithm.

Let us study this problem in detail based on previous work in the literature. The complexity of distributed deadlock algorithms is often so high that it is almost impossible to apply conventional formal proof systems. As a result, most researchers either just skipped the correctness proof [74] or resorted to informal schemes such as informal operational proof [53, 57, 65] or simulation [17]. Since these methods easily skip some situations in the proof, they frequently arrived at the false conclusions.

On the other hand, there were a few trials in which formal proofs were given to the proposed deadlock algorithms. For example, some researchers [22] tried to prove
the correctness of their algorithm using CSP axiomatic proof system [75], but they found the whole proofs need a lot of spaces to show themselves, and thus only some parts of the whole proof were selected and presented. The others [47, 71] made some mistakes in their proof since their computational model and its proof system were not clearly defined. For example, when they tried to prove that their algorithm always detects a true deadlock, their arguments were based on a not-yet-proven fact that the algorithm never detects a phantom deadlock.

When we redirect our attention to deadlock algorithm, we become to know that deadlock detection algorithms adopting edge-chasing scheme [44] are known to show the highest performances in terms of the number of messages required to detect a deadlock. For example, [9, 10, 17, 47, 54, 65, 68, 72, 74, 79] use this scheme.

For algorithms in this class, resolving deadlock by aborting a process makes some previous sound probes obsolete and thus such probes may later induce phantom deadlock detection. Thus, a careful consideration is needed to design the resolution phase of the algorithm. Since the treatment of this problem is not trivial, the algorithm in [65] even allows phantom deadlock detection to give a simple algorithm.

In this chapter, an efficient wait-free version of edge-chasing algorithm is presented. As the others do, we provide two program specifications; safety condition demands no phantom deadlock detection and progress condition guarantees eventual detection of any true deadlock. One advantage of our algorithm is that in the process abortion phase the process detecting a deadlock aborts itself immediately without cleaning the out-of-date probes remaining in the deadlock chain. By comparison, previous
algorithms [17, 47, 71] require the process that detects a deadlock to wait until the out-of-data probes, if any, are cleared from the deadlock chain. Algorithm in [65] does adopt immediate abortion as ours, but unfortunately it allows the existence of a phantom deadlock detection while ours does not.

Then, we propose a simple computational model and its proof system. Contrary to conventional proof systems, our proof system turns out to be effective enough to reduce the whole proof into a manageable size. Based on our proof system, we present the complete correctness proof within a reasonable space and time.

This chapter is organized as follows. In Section 3.2, a distributed database system is described and its simplified model is introduced. In Section 3.3, a distributed deadlock detection and resolution algorithm is introduced. In Section 3.4, a serialized computational model and its proof system are introduced. In Section 3.5, two correctness conditions are proved. In Section 3.6, the performance of the proposed algorithm is studied. In Section 3.7, the concluding remarks follow.

3.2 Model of a Resource-Sharing Distributed Database System

3.2.1 Distributed Database System

A database is a structured collection of information. In a distributed database system, the information is spread across a collection of nodes (or sites) interconnected through a communication network. Within a node, there are several processes and data items (or objects). A process is an autonomous active entity that is scheduled for execution. To access one or more data items, which may be distributed over several nodes, a user
creates a transaction process at the local node. The transaction process coordinates actions on all data items participating in the transaction and preserves the consistency of the database.

Data items are passive entities that represent some independently accessible pieces of information. Each data item is maintained by a data manager which has the exclusive right to operate on the data item. If a transaction wants to access a data item, it must send a data-lock request to the data manager that is in charge of the data item. A data manager can maintain several data items simultaneously. However, for ease of presentation, it is assumed that a data manager maintains only one data item.

In addition to data manipulation operations, a data manager provides data scheduling. A data manager honors the lock request of a transaction if the data item is free (not locked by any transaction); otherwise it keeps the lock request pending in a queue. A transaction which has locked the data item is called the holder of the data, whereas a transaction which is waiting in the request queue is called a requester of the data item. When a holder unlocks the data item by sending a data-release message to its data manager, the data manager chooses the next holder from its request queue, and grants the lock to the new holder by sending a data-grant message. However, the scheduling algorithm is not our concern in this chapter.

A transaction can be in one of two states: active or wait. When a transaction sends a lock request to a data manager, it enters the wait state until it receives a grant message from the data manager; otherwise it is in the active state, doing some
meaningful work. It is assumed that a transaction locks data one after another (i.e., at any time it has only one outstanding lock request), and it follows the two-phase locking protocol [23]. Thus, when an active transaction normally finishes its task, it releases all data locks by sending *data-release* messages to the corresponding data managers and terminates itself.

### 3.2.2 Proposed Model for Distributed Database System

A distributed system consists of a set of processes each of which is either *permanent* or *temporary*, representing a data or a transaction manager, respectively. Each process is assigned a unique priority. The priority of every permanent process is higher than the priority of any temporary process, as will be explained later.

There are two types of messages; *connect* message "C" and *disconnect* message "D". The "C" message is used to build a dependence edge between a temporary process and a permanent process. The "D" message is used to break a dependence edge. For a process \(v\), the number of outgoing dependence edges from \(v\) to the other processes is at most one at any instant and the identifier of its destination process, if the edge exists, is recorded in a local variable \(OUT_v\) in process \(v\). On the other hand, \(v\) can have several incoming dependence edges simultaneously and the source processes of those edges are recorded as a set \(IN_v\).

The dynamic behavior of each process in the system is modeled by six actions, denoted as \(A_i\), \(1 \leq i \leq 6\). The first three actions \(A_1-A_3\) can be activated only by a temporary process and the fourth action \(A_4\) is used only in a permanent process. The remaining two actions work in both types of processes. The message channel from
process $u$ to process $v$ is denoted $c[u,v]$. The following explains the function of each action.

1. $A_1$: **Data Request.** If a temporary process $v$ has no outgoing edge (i.e., $OUT_v = null$), it selects a permanent process $w$ and tries to establish a dependence edge to $w$ by sending a “$C$” message to $w$.

2. $A_2$: **Data Release.** If a temporary process $v$ has no outgoing edge, it selects an incoming edge already established, say from process $u$, and disconnects it by sending a “$D$” message to $u$.

3. $A_3$: **Process Termination.** If a temporary process $v$ has neither outgoing nor incoming edge, it is free to terminate itself. A terminated process no longer exists in the system. Notice that any permanent process is not allowed to terminate; thus, it lives permanently in the system.

4. $A_4$: **Data Grant.** If a permanent process $v$ has no outgoing edge, it selects an incoming edge already established, say from $u$, and reverses the direction by sending “$C$” and “$D$” messages to $u$ in this order.

5. $A_5$: **Incoming Edge Connection.** If a process $v$ receives a “$C$” message from a process $u$, then $u$ is added to $IN_v$.

6. $A_6$: **Outgoing Edge Disconnection.** If process $v$ receives a “$D$” message from process $OUT_v$, then $v$ disconnects its outgoing edge by nullifying $OUT_v$. 
When a sequence of processes forms a cycle such that a process in the sequence is connected to the next one along the sequence by an outgoing edge, no process in the cycle can generate any "D" message that can break the cycle since the fact that $OUT \neq null$ for each process in the cycle prevents the generation of "D" message in actions $A_2$ and $A_4$. Thus, all processes in the cycle wait infinitely, i.e., a deadlock occurs.

To detect and remove deadlocks is an important requirement for the progress and performance of distributed database systems. Our algorithm presented in the next section minimizes the processing delay time introduced by deadlock cycle. The six actions modeling database system are described formally in the next section together with the proposed deadlock detection and resolution algorithm.

### 3.2.3 Problem Definition

The following definition defines a deadlock process chain formally.

**Definition 11** Let a process chain $X$ be $(x_1, \cdots, x_k)$ for which any process in $X$ is different from the remaining ones except that $x_1 = x_k$ and $x_1$ has lowest priority among all processes in $X$. $X$ is said to be *deadlocked* iff the following predicate $\text{dead}(X)$ is true.

\[
\text{dead}(X) = \left( OUT_{x_i} = x_{i+1} \right) \land \left( \text{"D(\cdot)" } \notin c[x_{i+1}, x_i] \right) \land (x_i \text{ is not executing } A_6),
\]

for $1 \leq i \leq k - 1$

The deadlock detection and resolution algorithm presented in the next section has to satisfy the following two conditions when it works together with the underlying
distributed system.

(Safety Condition) If a process $v$ claims a deadlock detection for process chain $X$, $X$ is a deadlock process chain.

(Progress Condition) If there exists a deadlock process chain $X$ in the system, $X$ is eventually detected.

3.3 Distributed Deadlock Detection and Resolution Algorithm

3.3.1 Overview

The basic idea in detecting a deadlock chain is to send a probe along the chain and detect the deadlock when the probe returns to its initiator. When an edge from process $v$ to process $w$ is established, a probe $p$ is created and stored at probe queue in process $w$ and process $v$ is said to be the initiator of probe $p$. If process $w$ has an outgoing edge, then $w$ makes a direct copy of probe $p$ and sends it to the next process. If $v$ and $w$ are in a deadlock cycle, then a copy of probe $p$ will eventually return to process $v$ and $v$ will then detect the deadlock. To reduce the required number of messages to detect a deadlock, a unique priority is assigned to every process and a probe can be created only when an edge from a lower to a higher process in the priority is established. Moreover, when a process is to send a direct copy of a probe in its queue to the next process, if the latter is lower in priority than the initiator of the probe, the propagation is cancelled. A process can send a direct copy of the probe in its queue, at most one time, along each outgoing edge. It is possible that
a probe queue in process \(v\) contains several probes whose initiators are identical. In that case, \(v\) assigns one probe among them to be the primary probe and remaining ones to be secondary. To be efficient in message sending, \(v\) sends direct copies of only primary probes in its queue. To resolve the deadlock as soon as possible, whenever a deadlock is detected by process \(v\), \(v\) aborts itself immediately. The probes along the broken cycle that were propagated through \(v\) become meaningless after the abortion and would later develop a wrong deadlock detection. To remove such obsolete probes, \(v\) sends a token message along the deadlock chain to remove such ill-informed probes. In addition, when \(v\) sends a "D" message to its preceding process in the deadlock cycle, say process \(u\), it includes blocking information into "D". When \(u\) receives "D" from \(v\), it blocks any probe propagation from itself until the token message arrives at \(u\) to remove this blocking information. This mechanism prevents obsolete-probe propagation out of a broken deadlock chain.

To implement these ideas, two types of messages are newly devised; probe and token messages. A probe message carries its initiator and it looks like \(uP(x)\), where \(x\) is the initiator of the probe. A token message looks like \(T(x,S)\), where \(x\) and \(S\) represent the aborted process in the broken deadlock chain and a set of initiators of obsolete probes remaining in the chain. In addition, the "D" message is modified to "D(S)" such that \(S\) includes all blocking information. Two new local variables, \(QUE\) and \(BAR\), are added to store arriving probes and blocking information respectively.

Since the propagation of a probe \(p\) stored in a queue depends on the status of probe \(p\), (for example, if \(p\) is a secondary probe, then propagation is not allowed),
$p$ is represented in a queue as an ordered 4-tuple $(	ext{initiator, sender, priority, state})$. 

$\text{initiator}$ and $\text{sender}$ represent the initiator of $p$ and the process who sent $p$, respectively. $\text{priority}$ is primary or secondary depending on the assignment by the process. $\text{state}$ records the propagation status of $p$ and has two values; waiting and sent. When a copy of probe $p$ is sent to the next process, $\text{state}$ is set to sent, otherwise, it remains waiting.

The six actions modeling the distributed system in the previous section are slightly modified to include some statements dealing with probe. When an edge from process $u$ to process $v$ is disconnected by $v$ at action $A_2$ or $A_4$, all probes in $QUE_v$ whose sender is process $u$ are removed from $QUE_v$, and, in addition, if $v$ has some blocking information which indicates that $u$ and $v$ are included in some broken deadlock chain, $v$ sends that information (stored in $S$) via message “$D(S)$”. When $v$ receives a “$C$” message from $u$ such that $u < v$, a probe $(u, u, \cdot, \cdot, \cdot, \cdot, \cdot)$ is created in $QUE_v$. When $v$ receives a “$D(S)$” message from $OUT_v$, all identifiers in $S$ are inserted in $BAR_v$ and all probes in $QUE_v$ are reset to waiting.

### 3.3.2 Formal Description

In this subsection, six actions that model the resource-sharing distributed system are formally described. In addition, three additional actions, $A_7$-$A_9$, that form the nucleus of the proposed deadlock algorithm are also described. The following explain the functions of $A_7$-$A_9$ briefly.

1. **$A_7$: Propagating Probes.** If $OUT_v$ is not null, then process $v$ sends direct copies of all probes in $QUE_v$ which are ready to propagate.
2. \textbf{A8: Receiving Probe Message.} If process $v$ receives a "$P(x)$" message from process $u$, then one of the following three cases occurs. First, if $u$ is not connected to $v$, $v$ ignores that message. Second, if $u$ is connected to $v$ and $x \neq v$, then the arriving probe $(x,u,\cdot,waiting)$ is inserted in $QUE_v$. Third, if $u$ is connected to $v$ and $x = v$ ("$P(v)$" returns to its initiator $v$), a deadlock is detected by $v$ and $v$ aborts itself immediately.

3. \textbf{A9: Receiving Token Message.} If process $v$ receives a "$T(x,S)$" message from process $u$, then it removes all out-of-date probes from $QUE_v$. Furthermore, a set $S'$ is constructed so that $S'$ includes initiators of all removed probes that were propagated to $OUT_v$.

The global system state at the starting time of the system, denoted as $INIT$, is that all channels in the system are empty and all local variables in the system are also empty or null. The system and its deadlock algorithm are formally described as follows.

1. \textbf{[Connecting an edge]} If a temporary process $v$ has no outgoing edge (i.e., $OUT_v = null$), it is free to make a connection to a permanent process by sending to the latter a "$C$" message.

\begin{align*}
A^v_i ::= & \quad \text{if } OUT_v = null \text{ then} \\
& \quad OUT_v := \text{select}(P \cup \{null\}); \\
& \quad \text{if } OUT_v \neq null \text{ then send a "}C\text{" message to } OUT_v;
\end{align*}
2. **[Disconnecting an edge]** If a temporary process \( v \) has no outgoing edge, it is free to disconnect an incoming edge, say \((u, v)\), by sending to \( u \) a "\( D(S) \)" message. In doing so, all probes at \( v \) that came to \( v \) through channel \((u, v)\) are deleted from the probe queue of \( v \).

\[ A_2^v := \text{if } \text{OUT}_v = \text{null and } \text{IN}_v \neq \emptyset \text{ then} \]
\[ \quad u := \text{select}(\text{IN}_v \cup \{\text{null}\}); \]
\[ \quad \text{if } u \neq \text{null then} \]
\[ \quad \text{disconnect edge } (u, v) \text{ by calling procedure DISC}(u, v); \]
\[ \quad \text{delete from } \text{QUE}_v \text{ all probes } p \text{ that have } \text{sender}(p) = u; \]

3. **[Terminating itself]** If a temporary process \( v \) has no outgoing nor incoming edge, it is free to terminate. A terminated process no longer exists in the system.

\[ A_3^v := \text{if } \text{OUT}_v = \text{null and } \text{IN}_v = \emptyset \text{ then} \]
\[ \quad \text{terminate itself;} \]

4. **[Reversing an edge]** If a permanent process \( v \) has no outgoing edge, it is free to reverse the direction of an incoming edge.

\[ A_4^v := \text{if } \text{OUT}_v = \text{null and } \text{IN}_v \neq \emptyset \text{ then} \]
\[ \quad u := \text{select}(\text{IN}_v); \]
\[ \quad \text{send a "C" message to } u; \]
DISC\( (u,v) \);

\[ OUT_v := u; \]

delete all probes of the form \((\cdot, u, \cdot, \cdot)\) from \(QUE_v\);

5. **[Receiving a connection message]** If process \(v\) receives a "C" message from process \(u\), then \(u\) is added to \(IN_v\). Furthermore, if \(u < v\) then a probe is created.

\[
A_5^v := \text{if } v \text{ receives a "C" message from } u \text{ then}
\]

\[
IN_v := IN_v \cup \{u\};
\]

\[
\text{if } (u < v) \text{ then insert a probe } (u, u, \cdot, \cdot, \text{waiting}) \text{ in } QUE_v;
\]

6. **[Receiving a disconnection message]** If process \(v\) receives a "D(Z)" message from process \(OUT_v\), then \(v\) disconnects its outgoing edge. All probes in \(QUE_v\) change their status to waiting. (Note that a "D(\cdot)" message received from a process apart from \(OUT_v\) is simply ignored.)

\[
A_6^v := \text{if } v \text{ receives a "D(Z)" message from } OUT_v \text{ then}
\]

\[
OUT_v := \text{null};
\]

\[
BAR_v := BAR_v \cup \{z \in Z : (z, \cdot, \cdot, \cdot) \in QUE_v\};
\]

\[
\text{for all probes } p \in QUE_v \text{ do } \text{status}(p) := \text{waiting};
\]
7. [Probe propagation] If $OUT_v$ is not null, then send all probes in $QUE_v$ which are ready to process $OUT_v$.

$$A^7_v ::= \text{if } OUT_v \neq \text{null and } BAR_v = \emptyset \text{ then}$$

$$\text{for every } p \in QUE_v \text{ with } \text{ready}(p) = \text{true do}$$

$$\text{send a } "P(\text{initiator}(p))" \text{ message to process } OUT_v;$$

$$\text{status}(p) := \text{sent};$$

$$\text{ready}(p) \equiv (p \text{ is primary and waiting) and } (\text{initiator}(p) \leq OUT_v)$$

8. [Receiving a probe] If process $v$ receives a “$P(x)$” message from process $u$, then three cases occur. First, if $u$ is not connected to $v$, $v$ ignores that message. Second, if $u$ is connected to $v$ and $x \neq v$, then a probe is inserted in $QUE_v$. Third, if $u$ is connected to $v$ and $x = v$ (“$P(v)$” returns to $v$), deadlock is detected by $v$ and $v$ aborts itself.

$$A^8_v ::= \text{if } v \text{ receives a } "P(x)" \text{ message from } u \text{ then}$$

$$\text{case}$$

$$\begin{align*}
(u \not\in IN_v): & \text{ do nothing; } \\
(u \in IN_v \text{ and } x \neq v): & \text{ insert the probe } (x, u, \cdot, \cdot, \cdot, \text{waiting}) \text{ in } QUE_v; \\
(u \in IN_v \text{ and } x = v): & \text{ /* detecting a deadlock */} \\
& \text{ insert } (v, u, \cdot, \text{primary}, \text{sent}) \text{ into } QUE_v;
\end{align*}$$
\(BAR_v := BAR_v \cup \{v\};\)

for every process \(y\) in \(IN_v\) do

\(\text{DISC}(y, v);\)

\(S := \{\text{initiator}(p) : p = (\cdot, \cdot, \cdot, \text{sent}) \in QUE_v\};\)

\((OUT_v \neq \text{null}) \rightarrow \text{send a } "T(v, S)" \text{ message to } OUT_v;\)

terminate itself;

9. [Receiving a token] If process \(v\) receives a "\(T(x, S)\)" from process \(u\), then remove all out-of-date probes from \(QUE_v\). Furthermore, a set \(S'\) is constructed so that \(S'\) includes initiators of all removed probes that were propagated to current next process.

\(A^v_g := \text{ if } v \text{ receives a token } "T(x, S)" \text{ from } u \text{ then}\)

\(S' := \{z \in S : (z, u, \cdot, \text{sent}) \in QUE_v\};\)

\((OUT_v \neq \text{null}) \text{ and } (S' \neq \emptyset) \rightarrow \text{send a } "T(x, S')" \text{ message to } OUT_v;\)

\(BAR_v := BAR_v - \{z \in S : (z, u, \text{primary}, \cdot) \in QUE_v\};\)

delete from \(QUE_v\) all probes \(p\) such that \(\text{initiator}(p) \in S\)

and \(\text{sender}(p) = u\)

\((x = u) \text{ and } (u \in IN_v) \rightarrow\)
\[ \text{IN}_v := \text{IN}_v - \{u\}; \]

delete all probes of the form \((x, u, y)\) from \(QUE_v\);

\((x = u) \text{ and } (\text{OUT}_v = u) \rightarrow \]

\[ \text{OUT}_v := \text{null}; \]

change the status of each probe in \(QUE_v\) to \textit{waiting};

10. [Deleting an edge] To delete an edge coming from \(u\), perform the following.

\[ \text{DISC}(u, v) ::= \]

\[ \text{IN}_v := \text{IN}_v - \{u\}; \]

\[ B := \{b \in \text{BAR}_v : (b, u, \text{primary}, \cdot) \in \text{QUE}_v\}; \]

send a \("D(B)"\) message to \(u\);

\[ \text{BAR}_v := \text{BAR}_v - B; \]

11. [Inserting a probe] When a probe \(p\) is being inserted into \(QUE_v\), it is actually inserted only if no probe in \(QUE_v\) has the same initiator and sender as \(p\). In that case, it is inserted as a primary if there is no other probe in \(QUE_v\) having the same initiator as \(p\); a secondary, otherwise.

12. [Deleting a probe] Whenever a primary probe \(p\) is deleted from \(QUE_v\), a probe in \(QUE_v\) with the same initiator as \(p\), if any, is promoted to a primary probe.
3.4 Proof Method

In this section, a simple computational model is proposed. Since the specification of a distributed deadlock detection and resolution algorithm consists of two conditions, a safety condition (if a deadlock is detected by the algorithm, such deadlock really exists) and a progress condition (if there is a deadlock in the system, it is eventually detected), the proof system has to provide some tools by which these conditions are represented and proved easily. We propose an easy-to-use proof system. Finally, we justify the soundness of our computational model.

3.4.1 A Distributed Computational Model

We assume that all processes and channels are failure-free, the buffer in a channel is unbounded, all messages are delivered in FIFO order with finite delay, and no process is infinitely faster than the others.

A process, when the system initiates, chooses one out of the nine actions representing the underlying database system and the deadlock algorithm, and then executes it. The process repeats this selection and execution until the process terminates or aborts. Selecting the sequence of actions is fair, i.e., the process chooses any action in the program infinitely often (assuming an infinite life for the process). Note that no action includes an indeterminate loop statement so that the execution of any action terminates in a finite time.

A process computation in a process $p$ is a sequence of actions that can be executed in $p$ in the same order. In the real-time computational model, a system computation
(in short, computation) is a sequence of statements that can be executed in the system in the same order. In other words, a computation is the statement-level interleaving of a set of process computations in an executable order.

On the other hand, in our sequential computational model, we restrict the domain of computations such that any computation is a sequence of actions. Note that in the sequential model the interleaving of process computations is done at the action-level. Later, we will show that our model is equivalent to the real-time model in the correctness proof.

3.4.2 Proof System

In this subsection, a simple axiomatic proof system based on the sequential computational model is presented. We drop the idea that a set of axioms representing nine actions would be used to prove program correctness, because the effects of any action on the system state are easy to understand. Instead, we will use informal arguments when we deal with actions.

We provide three properties and their inference rules that are found to be useful. The first property, the invariant property, is a typical tool to represent a safety condition. A predicate \( p \) is invariant if \( p \) is true initially and remains true forever whatever the remaining computations are. In particular, in the sequential model, a predicate \( p \) is invariant if it is true initially and the assertion \( \{ p \} A \{ p \} \) holds regardless of the type of action \( A \) and the process executing \( A \). By induction, these two conditions imply that \( p \) is always true, i.e., invariant. The following inference rule formulates this fact.
The second property, the *ensures* property, that can describe some progress conditions, uses two predicates $p$ and $q$ and its typical representation is of the format "$p$ ensures $q$", meaning that if $p$ is true at some time, then $q$ is also true at the same time, or $q$ becomes true eventually and until then, $p$ remains true. We will now formulate this property. Suppose predicate $p$ is true at time $t$ and action $A$ is to be executed next. A necessary condition for the above property is that after execution of $A$, either $p$ or $q$ is true. However, this does not guarantee that $q$ eventually becomes true. To make sure of this requirement, there must exist an action $A^*$ that will make $q$ true. By the fair selection rule, $A^*$ will eventually be selected so that $q$ becomes true. The following inference rule formulates the *ensures* property.

\[(I1) \quad (INIT \Rightarrow p) \land (\text{for any action } A, \{p\} A \{p\}) \quad \text{invariant } p\]

\[(I2) \quad (\forall A, \{p \land \neg q\} A \{p \lor q\}) \land (\exists A, \{p \land \neg q\} A \{q\}) \quad p \text{ ensures } q\]

The third property, the *leads-to* property, is a generalization of the *ensures* property. "$p \leadsto q$" asserts that predicate $q$ eventually becomes true if predicate $p$ is true. However, $p$ does not have to stay true until $q$ becomes true. In this case, a metric $M$ showing the distance between a current state and a target state, and its domain, a well-defined, finite set $W$, are convenient tools to represent the state transition quantitatively. Any two elements in $W$ are ordered lexicographically (let $<$ denote this order). Suppose that we can prove "$(p \land M = m) \text{ ensures } ((p \land M < m) \lor q)$"
for every value $m$ in $W$. Then, it means that in every case, $M$ decreases unless $q$ becomes true. Since $W$ is finite, $q$ can not remain false infinitely. Thus, $q$ eventually becomes true. The following inference rule formulates the \textit{leads-to} property.

\[
(I3) \quad \forall m \in W, (p \land M = m) \quad \text{ensures} \quad ((p \land M < m) \lor q) \quad p \leftrightarrow q
\]

### 3.4.3 Justification of Sequential Model

In this subsection, we will address the soundness of our sequential model. Initially, we will show that predicate $\text{dead}(X)$, indicating whether process chain $X$ is deadlocked, is stable in the real-time model. Then, we will show that any stable property in the real-time model is \textit{equivalent} to that in the sequential model in the sense that correctness in the former implies that in the latter and vice versa.

**Stability of predicate “Dead(X)” in Real-time Model**

The following lemma proves that predicate $\text{dead}(X)$ defined in the previous section is stable under the real-time model in the underlying database system. Thus, we consider only six actions, $A_1$-$A_6$, without any statement dealing with probe or blocking information.

**Lemma 10** If $\text{dead}(X)$ with a process chain $X = (x_1, \cdots, x_k)$ is true at time $t$, $\text{dead}(X)$ remains true after time $t$ in the real-time model.

**Proof.** Suppose that $\text{dead}(X)$ is true at time $t$. Then no process, $x_i, 1 \leq i \leq k$, is executing action $A_2$ or $A_3$ at time $t$ since $OUT_{x_i}$ must be null throughout all
control points inside of those actions. The definition of $\text{dead}(X)$ also excludes the case that process $x_i$ is executing $A_6$. We will prove that $\text{dead}(X)$ remains true after the execution of any statement $s$ that is ready to be executed. Suppose statement $s$ is to be executed in $x_i$ for some value $i$, $1 \leq i \leq k$. (If not, $\text{dead}(X)$ trivially remains true after the execution of $s$.) If statement $s$ is in $A_1$, then it must be the last statement which sends the "C" message since $\text{OUT}_{x_i}$ is not null. $\text{dead}(X)$ remains true after the execution of such statement $s$. Statement $s$ can not be inside of $A_4$ since $\text{OUT}_{x_i}$ is null over all control points inside of $A_4$, contradicting the definition of $\text{dead}(X)$. Any statement in $A_6$ can not change $\text{dead}(X)$ to false. $x_i$ can not enter into $A_6$ since there is no "D(·)" in channel $c[\text{OUT}_{x_i}, x_i]$ at time $t$, by the definition of $\text{dead}(X)$. Thus, $\text{dead}(X)$ remains true after the execution of statement $s$. □

In the sequential model, we are concerned with system states between two subsequent actions. The last condition in the definition of $\text{dead}(X)$ is thus meaningless under the sequential model. The following definition restates $\text{dead}(X)$ regarding this fact.

**Definition 12** Let a process chain $X$ be $(x_1, \ldots, x_k)$ for which any process in $X$ is different from the remaining ones, except $x_1 = x_k$ and $x_1$ has lowest priority in $X$. $X$ is **deadlocked** iff $\text{dead}(X)$ is true.

$$\text{dead}(X) = (\text{OUT}_{x_i} = x_{i+1}) \land ("D(\cdot)" \notin c[x_{i+1}, x_i]), \text{ for } 1 \leq i \leq k - 1$$
Equivalence between Real-time Model and Sequential Model

Suppose $n$ processes are in the system and they are uniquely ordered. A computation is graphically depicted as a time diagram, which has $n$ horizontal time axes in which $k$-th time axis depicts the process computation of $k$-th process in the system. Refer to Fig. 20. A bar embedded on $k$-th time axis covering the time period $[t_1, t_2]$ denotes an execution of some action $A$ by $k$-th process during the period. We will call this bar event in this subsection. An event $E$ has three parameters, $\text{proc}(E)$, $\text{period}(E)$, and $\text{act}(E)$, which mean the process at which $E$ occurs, the period of $E$, and the type of action executed by $E$, respectively. $t^-(E)$ and $t^+(E)$ represent the starting and the ending time of $\text{period}(E)$, respectively.

A time-vector $T$ is $n$-tuple $(t_1, \cdots, t_n)$ such that $t_k, 1 \leq k \leq n$, denotes an instant for $k$-th process. A global state $S(T, C)$ is a state of computation $C$ at time-vector $T$ where for all $k$, the local variables in $k$-th process are set to the values at time $t_k$ and each channel $c[i, k]$ contains a sequence of messages sent from $i$-th process to $k$-th process up to the time $t_i$ minus a sequence of messages received by $k$-th process up to the time $t_k$.

There are two types of global states: real-time state, $S_R$, and snapshot state, $S_S$. A real-time state is a global state at the time-vector $T = (t_1, \cdots, t_n)$ such that $t_1 = t_2 = \cdots = t_n$. On the other hand, a snapshot state is a global state at the time-vector $T = (t_1, \cdots, t_n)$ such that for all $i$, $1 \leq i \leq n$, $i$-th process does not execute any action at the time $t_i$. For example, in Fig. 20 showing a typical computation, say $C$, the vertical line at time $t_a$ represents time-vector $T_a = (t_1, \cdots, t_n)$.
such that \( t_k = t_a, 1 \leq k \leq n \), and the corresponding real-time state at time-vector \( T_a \) is represented as \( S_R(T_a, C) \) (or \( S_R(t_a, C) \)). The dotted line represents a snapshot time-vector \( T_b \) and its corresponding snapshot state is represented as \( S_S(T_b, C) \) (or \( S_S(e_1, \ldots, e_n, C) \) where \( e_i, 1 \leq i \leq n, \) represents the number of events occurring up to \( T_b \) on \( i \)-th time axis). There is one condition about snapshot state \( S_S(T, C) \); any event appearing before \( T \) in computation \( C \) is independent on all events appearing after \( T \); i.e., no event before time-vector \( T \) reads a message sent from some event occurring after \( T \).

Let \( C \) be a computation, \( T \) be a time-vector \((t_1, \ldots, t_n)\), and \( S \) be a global state \( S(T, C) \). A function \( R(S) \) is an n-tuple \((r_1, \ldots, r_n)\) where \( r_k \) is the number of events initiated by \( k \)-th process up to time \( t_k \) for all \( k \). \( F(S) \) is a set of snapshot states in computation \( C \) such that for any state \( S' \) in \( C \), \( S' \) is in \( F(S) \) if \( R(S) \leq R(S') \). \( F^*(S) \) is a unique snapshot state in \( F(S) \) such that \( R(S) = R(F^*(S)) \). For example, in Fig. 20(a), \( S_S(T_b, C) \) represents \( F^*(S) \) for a real-time state \( S = S_R(t_a, C) \).
Figure 21: Time diagram 2.

Figure 22: Time diagram 3.
For a computation $C$ and its snapshot state $S = S_S(T, C)$ with $T = (t_1, \ldots, t_n)$, $T_l(S)$ and $T_f(S)$ are $\max\{T^+(E) : \text{event } E \text{ occurs before time-vector } T\}$ and $\min\{T^-(E) : \text{event } E \text{ occurs after time-vector } T\}$, respectively. $\text{Trans}(S)$ is another computation derived from computation $C$, such that all events following $T$ in computation $C$ are delayed by a constant time $D\cdot\text{Time}(S) = \max\{T_l(S) - T_f(S), 0\} + \text{gap}$ for which $\text{gap}$ is an arbitrary small positive constant. (Remind that each event occurring before $T$ in computation $C$ is independent on all events occurring after $T$ and thus all events in $\text{Trans}(S)$ can be executable in that order.) $S\cdot\text{Trans}(S) = S_S(T', \text{Trans}(S))$ is a snapshot state in computation $\text{Trans}(S)$ at time-vector $T' = (t'_1, \ldots, t'_n)$ such that $t'_k = t_k + D\cdot\text{Time}(S)$ for all $k$. Let $T\cdot\text{Trans}(S)$ denote $T'$ in this situation. Fig. 20(b) shows computation $Trans(S_S(T_b, C))$ derived from computation $C$ shown in Fig. 20(a).

The following theorem shows three relations between real-time and snapshot states.

**Theorem 3** Let computation $C$ represent a real-time computation, and predicate $p$ be stable in the real-time model. Then, the following statements are true.

(a) In a real-time state $S = S_R(T, C)$, if $p$ is true, then $p$ is also true for any snapshot state in $F(S)$.

(b) In a snapshot state $S = S_S(T, C)$, if $p$ is true, then $p$ is also true in the real-time state $S_R(T_l(S), C)$.

(c) In a snapshot state $S = S_S(T, C)$, if $p$ is true, then $p$ is also true for any snapshot state in $F(S)$. 
Proof. (a) Suppose predicate \( p \) is true at \( S = S_R(T_a, C) \) in computation \( C \). Refer to Fig. 20(a) and (b) which show \( C \) and \( C' = Trans(S) \), respectively. Note that \( S' = S_S(T_b, C) \) is included in \( F(S) \). The time-vector \( T_c \) in computation \( C' \) represents \( T - Trans(S') \) and \( S - Trans(S') \) is equal to \( S_S(T_d, C') \). Since computations \( C \) and \( C' \) are identical to each other before \( T_a \) and \( T_a \), respectively, and during the periods \([T_a, T_b]\) and \([T_a, T_d]\), respectively, \( S_R(T_a, C) = S_R(T_a, C') \) and \( S_S(T_b, C) = S_S(T_d, C') \). Since \( p \) is true at \( S_R(T_a, C) \), it remains true at \( S_R(T_c, C') \). Since there is no execution between time-vectors \( T_c \) and \( T_d \) in computation \( C' \), \( S_R(T_c, C') = S_S(T_d, C') \). Combining these relations, we can easily show \( p \) is true at \( S_S(T_b, C) \).

(b) Suppose predicate \( p \) is true at the state \( S = S_S(T_b, C) \) in computation \( C \). Let \( T_a = T_l(S) \), \( C' = Trans(S) \), and \( T_c = T_a + Trans(S) + gap \), as shown in Fig. 21. Then, by the relation between \( C \) and \( C' \), \( S_S(T_b, C) = S_S(T_b, C') \) and \( S_R(T_a, C) = S_R(T_c, C') \). Since there is no execution between time-vectors \( T_a \) and \( T_b \) in computation \( C' \), predicate \( p \) is also true at \( S_R(T_a, C') \). By the real-time stability, \( p \) remains true forever at any real-time vector after \( T_a \) in computation \( C' \) and thus, \( P \) is true at \( S_R(T_c, C') \) that is equal to \( S_R(T_a, C) \).

(c) Suppose property \( P \) is true at the state \( S = S_S(T_a, C) \) in computation \( C \). Let \( T_b \) be any time-vector such that \( S_S(T_b, C) \in F(S) \). Refer to Fig. 22. We derive two computations \( C' = Trans(S) \) and \( C'' = Trans(S_S(T_b + D\text{-}Time(S), C')) \). To simplify our argument, we assume that there is only one event \( E \) between \( T_a \) and \( T_b \) in computation \( C \). Let \( T_c \) and \( T_d \) be \( T^-(E) + D\text{-}Time(S) \) and \( T^+(E) + D\text{-}Time(S) \), respectively. Let \( T_e = T_b + D\text{-}Time(S) + D\text{-}Time(S_S(T_b, C)) \). Then,
Since there is no execution in computation $C''$ between $T_a$ and $T_c$, and also between $T_d$ and $T_e$, by the similar reasoning to previous cases (a) and (b), we can easily show $p$ is true at $S_S(T_b, C)$. □

Let $p$ be a predicate stable in the real-time model. According to the above theorem, the existence of a snapshot state in computation $C$ with predicate $p$ true implies the existence of a real-time state with $p$ true in $C$ and vice versa.

There remains one more step to show the equivalence between the real-time model and the sequential model.

**Definition 13** Fair computation of a real-time computation $C$, $Fair(C)$, is a sequentialized computation of $C$ generated by the following procedure.

(a) A time pointer $TP$ is set to the initial time of computation $C$.

(b) Move $TP$ along the time axes of $C$ until $TP$ arrives at the first instant an event $E$ starts its execution.

(c) Modify $C$ such that all events in $C$ starting after or on time $TP$ except event $E$ are evenly translated along the time axes until there is no overlap between $E$ and all translating events.

(d) If there remains some events in $C$ that starts after time $TP$, then go to (b).

Otherwise, assign the resulting $C$ to $Fair(C)$.

Each event reads at most one message at its beginning time. Thus, when an event starts at time $t$, it is independent on the following events whose beginning times are no earlier than $t$. (If not, a message should arrive at the destination which is not yet
created up to time \( t \).) \( \text{Fair}(C) \) is thus an executable sequence of actions that is a computation in the sequential model.

Note that the \( k \)-th process computation in \( C \) is identical to that in \( \text{Fair}(C) \) for all \( k \) since the initial state is identical in both \( C \) and \( \text{Fair}(C) \), the sequence of events in \( k \)-th process is also identical, and thus, the arriving sequence of messages in the channels are identical.

It is obvious that the whole domain of computations in the sequential model is a subset of that in the real-time model. Let \( C \) be a real-time computation and \( E \) be an event in \( C \). Then, real-time states immediately before and after \( E \) in \( \text{Fair}(C) \) are identical to the corresponding snapshot states in \( C \). The following theorem proves that if a real-time stable predicate \( p \) is true at some real-time state in \( C \), then there exists a real-time state in \( \text{Fair}(C) \) where \( p \) is true, and vice-versa.

**Theorem 4** Suppose that \( C \) is a real-time computation and \( S \) is a real-time state at time \( t \), \( S_R(t,C) \). Let \( p \) be a stable predicate in the real-time model.

(a) Suppose that \( p \) is true at \( S \). Then, \( p \) is also true at state \( S_R(t',\text{Fair}(C)) \) where \( t' \) is the earliest time in \( \text{Fair}(C) \) when all events in \( C \), which are initiated before time \( t \), are already finished.

(b) If \( p \) is true at some time \( t' \) between two consecutive events in \( \text{Fair}(C) \), then \( p \) is also true at state \( S_R(t,C) \) where time \( t \) is the latest among the finishing times of all events appearing before \( t' \) in \( \text{Fair}(C) \).

**Proof.** (a) Suppose that \( p \) is true at \( S \) and \( S' \) is a real-time state \( S_R(t',\text{Fair}(C)) \) such that \( t' \) is defined as in statement (a). Then, there exists a snapshot state \( S'' \) in
such that all events finished before $S'$ are also finished before $S''$ and vice versa. Thus, $S'$ equals $S''$. Moreover, $S''$ is in $F(S)$. Thus, by Theorem 3(a), $p$ is true at $S'$.

(b) Let $t$ and $t'$ be defined as in statement (b). Suppose that $p$ is true in the real-time state $S_R(t', \text{Fair}(C))$. Then, $S_R(t', \text{Fair}(C))$ equals some snapshot state, $S''$, in $C$ where all events appearing before $t'$ in $\text{Fair}(C)$ are finished. Thus, by Theorem 3(b), predicate $p$ is true at $S_R(t, C)$. □

According to the above theorem, any real-time stable predicate $p$ that is true at some real-time state in $C$ is also true at some other real-time state in $\text{Fair}(C)$ and vice versa. Thus, to detect and resolve a stable property in the sequential model is equivalent to that in the real-time model.

3.5 Correctness Proof

In the first subsection, we will develop several properties that show basic characteristics of the system embedded with algorithm (which together will be simply said the algorithm in this section). We will then prove the safety condition in the following subsection. Finally, we will prove the progress condition in the last subsection. Although two correctness conditions to be proved look simple, a large amount of relevant properties about messages and probes are needed to prove them because the deadlock detection is due to a probe returning to its initiator.

3.5.1 Basic Properties

This subsection provides several basic properties that help prove the safety and the progress conditions in the following.
We define three sets of processes in the following way.

- \( \Sigma_P \equiv \) a set of permanent processes in the system.
- \( \Sigma_T \equiv \) a set of temporary processes in the system.
- \( \Sigma \equiv \Sigma_P \cup \Sigma_T \).

There are two ways for a process \( v \) to decease: \( v \) terminates itself as a normal termination (\( A_3 \)) or \( v \) aborts itself to resolve a deadlock (\( A_8 \)). From the time process \( v \) is either terminated or aborted, all incoming messages to process \( v \) are ignored and no message is sent from process \( v \) to the others. Thus we model all incoming channels and local variables of process \( v \) remain empty after its decease.

**[Deceased process]** For any deceased process \( v \),

(i) \( IN_v = BAR_v = QUE_v = \emptyset \) and \( OUT_v = null \).

(ii) for each process \( u \) in \( \Sigma \), \( c[u,v] = \emptyset \) and no more action is executed by \( v \).

Since a permanent process has higher priority than any temporary process does and the edge construction is possible only between a permanent process and a temporary process, the initiator of a probe is always temporary process and thus no deadlock detection occurs at a permanent process. Furthermore \( A_3 \) is executed only by temporary processes. Therefore, a deceased process, if any, is also a temporary process. The following list enumerates a number of fundamental facts about channel, edge, probe, token message, and probe barrier. All of them can be easily proved by applying inference rule (I1).
Definition 14 The notations \textit{initiator}(p) and \textit{sender}(p) for a probe \( p \) are abbreviated as \( i(p) \) and \( s(p) \), respectively.

Definition 15 For processes \( u \) and \( v \), let \( \text{QUE}_v(u) = \{ p \in \text{QUE}_v : i(p) = u \} \) be the set of all probes in \( \text{QUE}_v \) which were initiated by \( u \).

Definition 16 For each probe \( p \), \textit{holder} of \( p \), \( \hat{p} \), is defined as follows.

\[
\hat{p} = \begin{cases} 
    \text{process } v, & \text{if } p \in \text{QUE}_v \\
    \text{process } w, & \text{if } p \text{ is in a channel toward } w, \text{c}[-,w]
\end{cases}
\]

Also, let \textit{channel}(p) denote the channel from \( s(p) \) to \( \hat{p} \).

Fact 1 Let \( u \) and \( v \) be two processes in \( S \). Let \( p \) be a probe and \( T \) be a token message, \( \text{"T}(v,\cdot)" \).

1. channel, edge

(i) If both \( u \) and \( v \) are either in \( \Sigma_P \) or \( \Sigma_T \), then \( \text{c}[u,v] = \emptyset \).

(ii) If \( v \) is in \( \Sigma_P \) (resp. \( \Sigma_T \)), any process \( u \) in \( \text{IN}_v \) or \( \text{OUT}_v \) is in \( \Sigma_T \) (resp. \( \Sigma_P \)).

(iii) If \( u \in \Sigma_P \), \( u \) is always alive.

(iv) If \( u \in \Sigma_P \) and \( \text{"C"} \in \text{c}[u,v] \), then \( \text{"D}(\cdot)" \) follows \( \text{"C"} \) in \( \text{c}[u,v] \).

2. probe

(v) \( i(p) \in \Sigma_T \)

(vi) Each probe in \( \text{QUE}_v(u) \) is sent from a distinct sender.

(vii) If \( \text{QUE}_v(u) \neq \emptyset \), then it contains one, and only one, primary probe.

(viii) \( \text{QUE}_v \) contains no probe whose initiator is \( v \), and process \( v \) never sends such a probe to the others.
(ix) For each probe \( p \) in \( QUE_v \), \( s(p) \in IN_v \).

(x) If \( OUT_v = \text{null} \), then all probes in \( QUE_v \) are waiting.

(xi) A probe can be propagated only if it is primary.

3. token

(xii) If \( T \) exists in the system, \( v \) is an aborted process and \( T \) is unique.

(xiii) Let \( "T(u, S)" \) be in \( c[v, w] \). If \( u \neq v \), then \( v \notin S \).

Otherwise, it is the last message in \( c[v, w] \) with \( v \in S \).

4. probe barrier

(xiv) For any \( X, Y \in B = \{ BAR \cup \forall v \} \cup \{ S' : \forall u, v, "D(S')" \in c[u, v] \}, \n\)

\( X \cap Y \) is empty and any process in \( X \) is a dead process.

(xv) If \( u \in BAR_v \), then there exists a probe \( p = (u, \cdot, \text{primary}, \text{waiting}) \in QUE_v \)

and no probe in \( QUE_v \) looks like \( (\cdot, s(p), \cdot, \text{sent}) \).

5. priority relation

(xvi) For any probe \( p \), \( i(p), s(p) \leq p \).

(xvii) For any token \( "T(x, S)" \), \( y \leq x \) for each \( y \) in \( S \).

Let \( v \) and \( w \) be two distinct processes. Edge \( (v, w) \) describes a state of the edge from process \( v \) to process \( w \). Initially, edge \( (v, w) \) is empty (i.e., not connected). When process \( v \) sends a connection message to process \( w \), \( (v, w) \) becomes growing. When \( w \) receives the connection message, \( (v, w) \) becomes established. Eventually, process \( w \) will disconnect edge \( (v, w) \) by sending a disconnection message to process \( v \) and \( (v, w) \) becomes shrinking. When \( v \) receives the disconnection message, \( (v, w) \) again becomes empty. We define these edge states formally and show their state transitions.
Definition 17 Let $v$ and $w$ be distinct processes. It is said that edge $(v, w)$ is

- **empty** if $\text{OUT}_v \neq w$, “$C$” $\notin c[v,w]$, $v \notin IN_w$, and “$D(\cdot)$” $\notin c[w,v]$;
- **growing** if $\text{OUT}_v = w$, “$C$” $\in c[v,w]$, $v \notin IN_w$, and “$D(\cdot)$” $\notin c[w,v]$;
- **established** if $\text{OUT}_v = w$, “$C$” $\notin c[v,w]$, $v \in IN_w$, and “$D(\cdot)$” $\notin c[w,v]$;
- **shrinking-1** if $\text{OUT}_v = w$, “$C$” $\notin c[v,w]$, $v \notin IN_w$, and “$D(\cdot)$” $\in c[w,v]$;
- **shrinking-2** if $\text{OUT}_v = w$,
  
  $w$ is aborted, “$T(w, \cdot)$” $\in c[w,v]$, and “$D(\cdot)$” $\notin c[w,v]$;
- **shrinking-3** if $v$ is aborted,
  
  “$T(v, \cdot)$” $\in c[v,w]$, and either “$C$” $\in c[v,w]$ or $v \in IN_w$.

Lemma 11 Let $v$ and $w$ be any two processes. Edge $(v, w)$ is either empty, growing, established, or shrinking, and it changes state according to the transition diagram in Fig. 23.

In order to prove the lemma, we need to use, along with it, that a channel may contain neither two or more “$C$” nor “$D(\cdot)$” messages. The following corollary proves Lemma 11 and itself together.

Corollary 1 If a “$C$” or “$D(\cdot)$” message exists in a channel, then there is no other message of the same type in that channel.

Proof. The lemma and corollary are proven simultaneously. Let $v$ and $w$ be two arbitrary processes, and $A$ be any action. Initially, edge $(v, w)$ is empty and both
Figure 23: State transition diagram of edge \((v, w)\)

\(c[v, w]\) and \(c[w, v]\) are empty—the first part of the lemma, as well as the corollary, is true. Assume that edge \((v, w)\) is either empty, growing, established, or shrinking before action \(A\). It is shown that, after the execution of \(A\), the edge is still in one of those states, and it obeys the transition diagram in Fig. 23. Besides, if there is a "C" or "D(\cdot)" message in \(c[v, w]\) or in \(c[w, v]\), there is no other message of the same type in the channel. (corollary 1)

Observe that an action taken by a process other than \(v\) or \(w\) will not change the state of edge \((v, w)\), for it certainly does not change the contents of \(OUT_v, c[v, w], IN_w,\) or \(c[w, v]\). Consequently, those actions occuring at either \(v\) or \(w\) are only considered.

**Case 1:** \((v, w)\) is empty before action \(A\). First, let \(A\) be an action of \(v\). The only actions that may change the edge's state are \(A_1^v\) and \(A_4^w\), should \(w\) be the selected process. In case of \(A_1^v\), \(w\) is a permanent (and thus not aborted) process, and after the action \((v, w)\) becomes growing and the "C" message in \(c[v, w]\) is unique. In case of \(A_4^w\), it is observed that before the action \(w\) is either alive or aborted, for, as a
consequence of \( w \in IN_v \), edge \((w, v)\) is either established or shrinking-3. Thus, after the action \((v, w)\) becomes either growing or shrinking-2 depending on whether \( w \) is alive or aborted. Now, if \( A \) is an action of \( w \), it is not hard to see that \((v, w)\) remains empty, as no action at \( w \) may possibly change any of these conditions: \( OUT_v \neq w \), "C" \( \notin c[v, w], v \notin IN_w \), and "D(\cdot)" \( \notin c[w, v] \).

Case 2: \((v, w)\) is growing before action \( A \). First, suppose \( A \) is an action of \( v \), which obviously is none of \( A_1-A_4 \) since \( OUT_v \neq null \). \( A_6 \) is also impossible, for at time \( t^{-}(A) \) there is no "D(\cdot)" message in \( c[w, v] \) for \( v \) to receive. In case of \( A_5, A_7, \) or \( A_9 \), the edge remains growing, as such an action is easily seen to not change any condition concerning \((v, w)\)'s state. The only action of \( v \) that may change \((v, w)\)'s state is \( A_8 \) in which \( v \) aborts itself and edge \((v, w)\) becomes shrinking-3. Next, consider the case of \( w \) executing the action. One readily sees that actions \( A_1, A_6 \) and \( A_7 \) have no effect on the state of \((v, w)\), and that actions \( A_2 \) and \( A_4 \) cannot happen at \( w \) since \( v \notin IN_w \). \( A_9 \) will not change the edge's state; that is because \( w \) is live and so even if \( w \) sends a token to \( u \) during the action, the edge's state will not change to shrinking-2. In case of \( A_5 \), \( w \) receives the only "C" message in \( c[v, w] \) and \((v, w)\) becomes established. It remains to consider \( A_3 \) and \( A_8 \), in which \( w \) either terminates or aborts itself. In either case, \( w \) is a temporary process and \( v \) a permanent. By Fact 1(iv), channel \( c[v, w] \) contains (at time \( t^{-}(A) \)) a "D(\cdot)" that follows the "C" message; and thus, at time \( t^{-}(A) \), edge \((w, v)\) is shrinking-1 and \( OUT_w = v \). So \( A_3 \) cannot occur at \( w \), and in case of \( A_8 \) \( w \) aborts itself and adds a "T(w, \cdot)" to \( c[w, v] \), changing edge \((v, w)\)'s state to shrinking-2.
Case 3: \((v, w)\) is established before action \(A\). In this case both \(v\) and \(w\) are alive at \(t^-(A)\). First let \(A\) be an action of \(v\). It cannot be \(A_1, A_2, A_3,\) or \(A_4,\) as \(OUT_v \neq \text{null}\); it cannot be \(A_6\) either since there is no \("D(\cdot)\)" message in \(c[w, v]\) for \(v\) to receive. Actions \(A_5, A_7\) and \(A_9\) each obviously have no effect on the edge’s state. As for \(A_8,\) if \(v\) gets aborted in the action, edge \((v, w)\) will become shrinking-3. Now suppose \(A\) is an action executed by \(w\). If it is an \(A_2, A_4,\) or \(A_8,\) then \(v\) is removed from \(IN_w\) and a \("D(\cdot)\)" message is added to \(c[w, v]\), and edge \((v, w)\) becomes shrinking-1. \(A_3\) certainly cannot happen since \(IN_w \neq \emptyset\). Actions \(A_1, A_5, A_6\) and \(A_7\) each obviously have no effect on \((v, w)\)'s state; and neither does \(A_9,\) since the token received by \(w\) cannot be initiated by the still alive \(v\) (i.e., the condition of the last if statement in \(A_9\) is false).

Case 4: \((v, w)\) is shrinking-1 before action \(A\). First let \(A\) occur at \(v\). It cannot be \(A_1, A_2, A_3,\) or \(A_4\) due to the condition \(OUT_v \neq \text{null}\). Neither \(A_5\) nor \(A_7\) may change the edge’s state. Actions \(A_6\) and \(A_8\) will each result in an empty \((v, w)\) (the edge cannot become shrinking-3 because \("C\) \notin c[v, w]\) and \(v \notin IN_w\). Edge \((v, w)\)'s state may change in \(A_9\) only if the received token is \("T(w, \cdot)\)" (i.e., initiated by \(w\) when \(w\) got aborted). Such a token, if exists, must be the very last message in \(c[w, v]\) (by Fact 1(xiii)) and be preceded by the \("D(\cdot)\)" message. Thus, \(v\) cannot receive such a token and \(A_9\) won’t change \((v, w)\)'s state. Now, if \(A\) is an action of \(w\), it is not hard to see that \((v, w)\) remains shrinking-1, as no action at \(w\) may possibly change any of these conditions: \(OUT_v = w, \ "C\) \notin c[v, w], v \notin IN_w,\) and \("D(\cdot)\) \in c[w, v].\)

Case 5: \((v, w)\) is shrinking-2 before action \(A\). Since \(w\) is already aborted, it is
only needed to consider v’s action. If A is an A_g action in which v receives token “T(w, ·)”, then OUT_v is set to null and edge (v, w) becomes empty. Actions A_1^w-A_4^w cannot happen, and A_5^w-A_8^w will not change the edge’s shrinking-2 state.

Case 6: (v, w) is shrinking-3 before action A. In this case A must be an action of w since v is already aborted. If the “C” message arrives at w in A_5, (v, w) is still shrinking-3. If v ∈ IN_w at t^−(A) and w reverses edge (v, w) in action A_4, then (v, w) becomes empty. (Recall that by the convention channel c[w, v] is always empty once v is aborted.) If w receives “T(v, ·)” in A_6^w, (v, w) becomes empty. It can be easily checked that A_1^w-A_3^w cannot happen and A_5^w-A_8^w will not change the edge’s shrinking-3 state.

□

Corollary 2 Every process has at most one nonempty outgoing edge.

Proof. By Definition 17 and Lemma 11, two edges, say (v, w) and (v, x), may both be non-empty only if both of them are shrinking-3. But even that is impossible, since there is only one “T(v, ·)” and c[v, w] and c[v, x] cannot both contain it.

□

When edge (v, w) for which process w is permanent becomes established, process w creates a probe p whose initiator and sender are set to process v and inserts p in QUE_w. If w has an outgoing edge, w propagates probe p by sending a direct copy of p to the next process. A sequence of probes P is a probe chain if for any two consecutive probes p and q in P, q is a direct copy of p propagated by the holder of p.

Note that the configuration of probe chain P is changed by two end processes which are the holders of the front (first) probe or the tail (last) probe in P except
when an abortion occurs at the process holding an intermediate probe in \( P \). In the former case, when the end process holding the head (last) probe in chain \( P \) decides to propagate the probe or to shrink by sending a disconnection message, chain \( P \) enlarges or shrinks by one, respectively.

In the latter case, \( P \) is partitioned into two separate probe chains and the chain consisting of the later part of \( P \), say \( Q \), is chased by a token message created by the aborting process. If the token message arrives at its destination, the tail (first) probe in \( Q \) is removed so that \( Q \) is reduced by one in size, and another token message is sent to the holder of the probe in \( Q \) that was the second probe in \( Q \) immediately before the abortion and is now the first probe in \( Q \).

**Definition 18** A probe \( p \) is said to be *barren* if it is in a channel \( c[x,y] \) such that \( x \notin \text{IN}_y \) and there is no "C" message in \( c[x,y] \) that precedes \( p \). Let \( \text{barren}(v) \) denote the set of all barren probes initiated by \( v \).

Note that a barren probe will stay barren until it arrives at the destination process, which will then simply discard the probe.

**Definition 19** Let \( p, q \) be two probes in the system; \( q \) is a *child* of \( p \) if \( q \) is a direct copy of \( p \).

Note that probe \( q \), a child of probe \( p \), is created by \( A_7 \) only if \( p \) is ready.

**Definition 20** Let \( q \) be a child of \( p \). The channel \( c[p,q] \) is said to be *clean* if the messages in it preceding \( q \) contains no "C" message, no probe with the same initiator as \( p \)'s, and no token "\( T(\cdot, S) \)" with \( i(p) \in S \).
In the following lemma, an interesting relation between a parent and a child probe is identified.

Corollary 3 If \( p \in \text{QUE}_p \) that is sent has no non-barren child, then \((\hat{p}, \text{OUT}_p)\) is shrinking.

Lemma 12 If \( q \) is a non-barren child of \( p \in \text{QUE}_p \), then \( p \) is primary and sent, \( \text{OUT}_p = \hat{q} \), and channel \( c[\hat{p}, \hat{q}] \) is clean.

Proof. Initially, the lemma is trivially true. Assume the lemma is true before an action \( A \), it is shown to remain true after the action. Let \( q \) be any non-barren child of \( p \in \text{QUE}_p \) as of time \( t^+(A) \). There are two possibilities: 1) \( q \) is a new child which was just created during the action in concern, and 2) \( q \) already existed at time \( t^-(A) \).

Case 1: Let \( q \) be a new child just created and added to \( c[\hat{p}, \hat{q}] \) in action \( A \) (which must be \( A_7 \)). It immediately follows from \( A_7 \) that after the action \( p \) is primary and sent, \( \text{OUT}_p = \hat{q} \), and \( c[\hat{p}, \hat{q}] \) is clean.

Case 2: Let \( q \) be a non-barren child of \( p \) before action \( A \) and remain so even after the action. First the edge \((\hat{p}, \hat{q})\)'s state is shown to be either growing or established, as long as \( q \) is a non-barren child of \( p \). (Note that both \( \hat{p} \) and \( \hat{q} \) are live.) If \( q \) is in a channel, its non-barreness (i.e., \( C \in c[\hat{p}, \hat{q}] \) or \( \hat{p} \in \text{IN}_\hat{q} \)) coupled with \( \hat{p} \)'s and \( \hat{q} \)'s liveness guarantee the edge to be either growing or established (see Definition 17). If \( q \) is in \( \text{QUE}_\hat{q} \), then \( \hat{p} \in \text{IN}_\hat{q} \) (by Fact 1(ix)) and edge \((\hat{p}, \hat{q})\) must be established.

Now, at time \( t^-(A) \), since edge \((\hat{p}, \hat{q})\) is growing/established, \( \text{OUT}_p = \hat{q} \). In action \( A \), \( \hat{p} \) cannot send any \( C \) message (since \( \text{OUT}_p \neq \text{null} \) at time \( t^-(A) \)); or any probe
with the same initiator as \( p \)'s (because of \( p \)'s sent status at time \( t^{-}(A) \)); or any token \( "T(\cdot,S)" \) with \( i(p) \in S \) (as otherwise \( p \) would have been removed from \( QUE_{\hat{p}} \) in \( A \)). So \( c[\hat{p},\hat{q}] \) remains clean after action \( A \). The status of \( p \) may change from sent to waiting only in action \( A_{6} \) in which \( \hat{p} \) receives a \("D(\cdot)" \) message from \( OUT_{p} \) and changes \( OUT_{p} \) to null. But since that is apparently not the case (for \( OUT_{\hat{p}} \neq null \) at time \( t^{+}(A) \)), \( p \)'s sent status does not change in action \( A \). As for \( p \)'s primary status, note that a primary probe remains primary until it is removed. □

**Definition 21** \( P = (p_{1}, \ldots, p_{k}) \) be a sequence of probes currently in the system. \( P \) is said to be a **probe chain** if (i) \( p_{i} \) is a child of \( p_{i-1} \) for each \( i, 1 < i \leq k \), and (ii) \( p_{k} \) is not barren. A probe chain is **maximal** if it is not a proper subsequence of any other chain.

**Definition 22** Suppose that a probe chain \( P = (p_{1}, \ldots, p_{k}) \) is **chased** by a token \("T(\cdot,S)" \) if i) \( "T(\cdot,S)" \) is in the channel where \( p_{1} \) came from (i.e., \( c[s(p_{1}),\hat{p}_{1}] \)), ii) \( i(p_{1}) \in S \), and iii) preceding \( "T(\cdot,S)" \) in the channel there is no \("C" \) message, no probe with the same initiator as \( p_{1} \)'s, and no any other token \("T(\cdot,S')" \) with \( i(p_{1}) \in S' \).

**Definition 23** If \( v \) is an aborted process, then for any process \( u \) let \( CH(u,v) \) denote the maximal probe chain which was initiated by \( u \) and is being chased by a token \("T(v,S)" \) initiated by \( v \) (if there is no such a chain, then \( CH(u,v) = \epsilon \)). If \( v \) is not aborted, let \( CH(v,v) \) denote the maximal chain probe \( (p_{1}, \ldots, p_{k}) \) initiated by \( v \) and with \( sender(p_{1}) = v \) (again, \( CH(v,v) = \epsilon \) if no such chain exists).
\(CH(v, v)\) and \(CH(u, v)\) are well defined in the sense that they each denote a unique probe chain. The following proposition proves this fact.

**Proposition 1** \(CH(u, v)\) and \(CH(v, v)\) are well defined.

**Proof.** Let \(v\) be aborted, and assume that two non-empty chains, \((p_1, \ldots, q_k)\) and \((q_1, \ldots, q_l)\), each satisfy the condition of \(CH(u, v)\). Since only one token initiated by \(v\) may exist, the two chains are chased by the same token, and, therefore, \(p_1\) and \(q_1\) are at the same process and have the same sender (i.e. \(p_1 = q_1\) and \(s(p_1) = s(q_1)\)). By Fact 1(vi), \(p_1\) and \(q_1\) are the very same probe, and it thus follows from the condition of \(CH(u, v)\) and Lemma 12 that \((p_1, \ldots, q_k) = (q_1, \ldots, q_l)\).

Next, consider \(CH(v, v)\) with \(v\) not aborted. Let \((p_1, \ldots, q_k)\) and \((q_1, \ldots, q_l)\) both satisfy the condition of \(CH(v, v)\). The fact that \(p_1\) and \(q_1\) are at the same process must be shown; the rest of the argument will then follow the same line as in the above case. Fact 1(viii) indicates that process \(v\) never sent probe \(p_1\), and so \(p_1\) must be a probe that was created and inserted in \(QUE_{\hat{p}_1}\) when \(\hat{p}_1\) received a "C" message from \(v\). Thus, \(p_1\) is in \(QUE_{\hat{p}_1}\) and therefore, by Fact 1(ix), \(v \in IN_{\hat{p}_1}\). Since \(v\) is not aborted and \(\hat{p}_1\) is live, the only state of edge \((v, \hat{p}_1)\) consistent with all of these conditions is established (see Definition 17). So \(OUT_v = \hat{p}_1\), and, similarly, \(OUT_v = \hat{q}_1\), which proves \(\hat{p}_1 = \hat{q}_1\). \(\square\)

Note that in the above proof, we actually proved the following corollary.

**Corollary 4** If \(v\) is not aborted and \(CH(v, v) \neq \epsilon\), then \(OUT_v \neq null\) and \(CH(v, v)\) starts at process \(OUT_v\).
The following lemma describes a useful relation between a probe and a probe chain $CH$.

Lemma 13 If $p$ is a non-barren probe initiated by a process $v$, then it belongs in some probe chain $CH(v,x)$, where either $x = v$ or $x$ is some aborted process.

Proof. Initially, there is no probe in the system and the lemma is true. Now assume the lemma is true before an action $A$. We show the lemma remains true after the execution of $A$.

At time $t^{+}(A)$, let $p$ be any non-barren probe originally initiated by $v$. There are two possibilities: that $p$ is created during action $A$, and that $p$ already exists at time $t^{-}(A)$. A probe may be created only in action $A_5$ (receiving a “C” message) or $A_7$ (propagating a probe), and $v$ may be aborted, live, or terminated. Thus, $p$ may be created in following cases.

Case 1: $p$ is created in $A = A_7$. Let $w$ be the process that creates $p_k$. At time $t^{-}(A)$, $w$ has a probe $q$ in its $QUE$, with $i(q) = v$ and in action $A$, $w$ sends $p$, a copy of $q$, to $OUT_w$. Since the lemma assumed to be true at time $t^{-}(A)$, $q$ is in some $CH(v,x) = (p_1, \ldots, p_k)$. Since every $p_i, i < k$, is in the sent status (by Lemma 12) and cannot produce a new child, it must be that $q = p_k$. Since $p$ is not barren, $CH(v,x) = (p_1, \ldots, p_k, p)$ after action $A$.

Case 2: $v$ is aborted at time $t^{-}(A)$ and $p$ is created in $A = A_5$. At time $t^{-}(A)$, channel $c[v,\hat{p}]$ contains a “C” message, which is the only such message in the channel according to Corollary 1. The only possible state at time $t^{-}(A)$ for edge $(v, \hat{p})$ is thus shrinking-3, and that means that channel $c[v,\hat{p}]$ contains token “$T(v,S)$” that
was created when \( v \) was aborted. By Fact 1(xiii), \( v \in S \) and "\( T(v, S) \)" is behind "\( C \)" in the channel; and apart from "\( T(v, S) \)", no other token \( T'() , S' \) with \( v \in S' \) is contained in the channel. From these discussions, it is clear that after the action, \( (p) \) is a maximal chain initiated by \( v \) and being chased by "\( T(v, S) \)". So, \( CH(v, v) = (p) \), justifying the assertion that \( p \) belongs in some \( CH(v, x) \).

Case 3: \( v \) is alive at time \( t^{-}(A) \) and \( p \) is created in \( A = A_3 \). It's obvious that after the action \( CH(v, v) = (p) \).

Case 4: \( v \) is terminated at time \( t^{-}(A) \) and \( p \) is created in \( A = A_5 \). This case actually cannot happen, for Lemma 11 and Definition 17 indicate that every outgoing edge of a terminated process is empty.

Now consider the case that \( p \) has been non-barren since \( t^{-}(A) \). At time \( t^{-}(A) \), \( p \) belongs to \( CH(v, x) \) for some \( x \), since the lemma is assumed to be true at that moment. If during the action no probe is deleted from \( CH(v, x) \), then after the action \( p \) certainly is still in \( CH(v, x) \). It is thus enough to consider only those actions in which at least one probe is removed from \( CH(v, x) \): \( A_2, A_4, A_8, \) and \( A_9 \). Let \( CH(v, x) = (p_1, \ldots, p_k) \). Notice that \( A_2 \) or \( A_4 \) is prohibited at \( \hat{p}_l(1 \leq l \leq k - 1) \) since \( OUT_{\hat{p}_l} \) is not null (Lemma 12). Since the channels along \( proj(CH(v, x)) \) is clean, \( A_9 \) at \( \hat{p}_l(2 \leq l \leq k) \) can not remove \( \hat{p}_l \). Moreover, in case that \( x \) is alive\( (v = x) \), there is no token "\( T(, S) \)" , \( v \in S \), in \( c[v, \hat{p}_1] \) (Fact 1(xii) & (xiii)). Thus, \( A_9 \) at \( \hat{p}_1 \) can not remove \( \hat{p}_1 \).

Case 5: \( A = A_2^{\hat{p}_k} \) or \( A_4^{\hat{p}_k} \), in which \( \hat{p}_k \) sends a "\( D(\cdot) \)" message to disconnect edge \((\hat{p}_{k-1}, \hat{p}_k) \). In this case, either \( p_k \) becomes barren (if it is in the channel from \( \hat{p}_{k-1} \)
to \( \hat{p}_k \) or it is removed from the \( QUE \) at \( \hat{p}_k \); and \( CH(v, x) \) reduces to \( (p_1, \ldots, p_{k-1}) \). Since \( p \) is assumed to be not barren, \( p \neq p_k \) and is still in \( CH(v, x) \). (Note that for \( i \neq k \), \( A_2 \) and \( A_4 \) are disabled at \( \hat{p}_1 \), for \( OUT_{\hat{p}_i} \neq null \).)

Case 6: \( A = A_8^{p_l} \ (1 \leq l \leq k) \), in which \( \hat{p}_l \) aborts itself. First note that \( \hat{p}_1, \ldots, \hat{p}_k \) are all different except that \( \hat{p}_k \) may equal one of the others (since, by Lemma 12, each \( \hat{p}_i \), \( i < k \), is primary). If \( \hat{p}_k = \hat{p}_i \) for some \( i \neq k \) and it is \( \hat{p}_k \) that gets aborted, then after the action \( CH(v, x) = (p_1, \ldots, p_{i-1}) \) and \( CH(v, \hat{p}_i) = (p_{i+1}, \ldots, p_{k-1}) \), and \( p \) belongs in one of the two chains. In all other cases, after \( \hat{p}_l \) aborts itself, \( CH(v, x) = (p_1, \ldots, p_{l-1}) \) and \( CH(v, \hat{p}_l) = (p_{l+1}, \ldots, p_k) \); and \( p \), not equal to \( \hat{p}_l \), is in one of them.

Case 7: \( A = A_9^{\hat{p}_1} \), in which the chasing token "\( T(x, S) \)" arrives at \( \hat{p}_1 \). (This case may happen only if \( x \) is aborted. If \( x \) is alive, \( c[v, \hat{p}_1] \) contains no token "\( T(\cdot, S) \)" that satisfies \( v \in S \).) In the action \( p_1 \) is deleted (and discarded) from \( QUE_{\hat{p}_1} \). That implies \( k > 1 \), since \( p \) is in \( CH(v, x) \) before the action and still exists after it. Lemma 12 indicates that, at time \( t^{-}(A) \), channel \( c[\hat{p}_1, \hat{p}_2] \) is clean and thus contains no "\( C \)" message, no probe initiated by \( v \), and no token "\( T(\cdot, S') \)" with \( v \in S' \). During action \( A_9 \), "\( T(x, S) \)" , with \( v \in S \), is added to channel \( c[\hat{p}_1, \hat{p}_2] \), chasing \( (p_2, \ldots, p_k) \), and \( CH(v, x) = (p_2, \ldots, p_k) \). (Note that an \( A_9 \) action at \( \hat{p}_i \), \( i > 1 \), will not change \( CH(v, x) \) since \( c[\hat{p}_{i-1}, \hat{p}_i] \) is clean (by Lemma 12).)

\[ \square \]

### 3.5.2 Proof of Safety Condition

In this subsection, we will prove the safety condition that no phantom deadlock is detected by the algorithm. Suppose a process \( v \) detects a deadlock chain by a
returning probe at the head of a live chain \( CH(v, v) \). The fact \( \text{proj}(CH(v, v)) \) ends with process \( v \) implies \( \text{proj}(CH(v, v)) \) is a real deadlock chain. When process \( v \) aborts itself, a nonempty set of dead chains \( \{CH(x, v) : \forall x\} \) is created. In the set, \( CH(v, v) \) is the longest chain and \( \text{proj}(CH(x, v)) \) is a prefix of \( \text{proj}(CH(v, v)) \) for all \( x \). Moreover, a disconnection message, "\( D(S) \)", is sent to the preceding process of \( v \), say \( u \), in deadlock chain \( \text{proj}(CH(v, v)) \). When the disconnection message arrives at process \( u \), \( u \) inserts \( v \) into \( BAR_u \) to prevent any chain \( CH(x, v) \) from propagating itself out of the broken deadlock chain. On the other hand, a token message is sent by \( v \) to chase probe chain \( CH(v, v) \) as well as \( CH(x, v) \) and remove the out-of-date probes so that eventually \( CH(v, v) \) and \( CH(x, v) \) becomes empty and thus the set \( \{CH(x, v) : \forall x\} \) becomes empty. The following definition formulates this fact.

**Definition 24** Let \( v \) be an aborted process and \( CH(v, v) = (p_1, \ldots, p_k) \). \( CH(v, v) \) is said to be regular if \( CH(x, v) = \epsilon \) for all \( x \), or if \( CH(v, v) \neq \epsilon \) and the following are satisfied.

(a) \( p_k \) is a primary probe in \( QUE_{\hat{p}_k} \).

(b) If \( p_k \) is in the "waiting" status, then \( v \in BAR_{\hat{p}_k} \).

(c) If \( p_k \) is in the "sent" status, then \( \hat{p}_k \)'s outgoing edge is shrinking, being disconnected by a "\( D(S) \)" message (i.e., "\( D(S) \)" \( \in c[\text{OUT}_{\hat{p}_k}, \hat{p}_k] \)) with \( v \in S \).

We first provide some useful properties about a regular probe chain in the following two lemmas. The second lemma proves that if a probe chain \( CH(x, x) \) is regular for every aborted process \( x \), the next deadlock detection is always a true detection. Thus, the only remaining task will be showing that \( CH(x, x) \) is regular for every aborted
Lemma 14 If \( CH(v, v) = (p_1, \ldots, p_k) \neq \epsilon \) is regular, then the following are true.

(a) \( \hat{p}_i \neq \hat{p}_j \) whenever \( i \neq j \).
(b) For any \( j < k \), \( \text{proj}(CH(\hat{p}_j, \hat{p}_j)) \) is a prefix of \( (\hat{p}_{j+1}, \ldots, \hat{p}_k) \).
(c) For every process \( x \in \Sigma \), \( \text{proj}(CH(x, v)) \) is a prefix of \( \text{proj}(CH(v, v)) \).

Proof. Lemma 12 and Definition 24 together indicate that all \( p_i \), \( 1 \leq i \leq k \), are primary, from which statement (a) follows immediately.

Now prove (b). Assume \( CH(v, v) \neq \epsilon \), or it is done. By Corollary 4, \( CH(\hat{p}_j, \hat{p}_j) \) is a probe chain starting at \( p_{j+1} \). Let \( CH(\hat{p}_j, \hat{p}_j) = (q_{j+1}, \ldots, q_l) \), where \( q_{j+1} = \hat{p}_{j+1} \). It follows from Lemma 12 and a simple induction that the two sequences \( (\hat{p}_{j+1}, \ldots, \hat{p}_k) \) and \( (q_{j+1}, \ldots, q_l) \) overlap with each other until the shorter one terminates: \( \hat{p}_i = q_i \) for \( i = j + 1, \ldots, \min(k, l) \). We show that \( l \leq k \). Since \( CH(v, v) \) is regular, \( p_k \) is in \( QUE_{\hat{p}_k} \). If \( p_k \) is in the sent status, then edge \((\hat{p}_k, OUT_{\hat{p}_k})\) is shrinking and no probe chain, even \( CH(\hat{p}_j, \hat{p}_j) \), can go beyond \( \hat{p}_k \). Now suppose \( p_k \) is in the waiting status so that \( i(p_k) \) is in \( BAR_{\hat{p}_k} \). When \( i(p_k) \) was added to \( BAR_{\hat{p}_k} \), all probes in \( QUE_{\hat{p}_k} \) (including \( q_k \), if it was there) changed their status to waiting (see action \( A_6 \)). Since then, all probe chains reaching \( \hat{p}_k \) from \( \hat{p}_{k-1} \) (including \( (q_{j+1}, \ldots, q_l) \), if \( l \geq k \)) have been stopped by \( BAR_{\hat{p}_k} \) from propagation. Hence, \( l \leq k \) and \( CH(\hat{p}_j, \hat{p}_j) \) cannot go beyond \( \hat{p}_k \).

Statement (c) can be proved similarly to (b). \( \square \)
Lemma 15 Assume $CH(x, x)$ to be regular for every aborted process $x$. If in execution of action $A_8$ a process $v$ claims to have detected a deadlock, then at the time immediately before the action, $proj(CH(v, v))$ forms a deadlock.

Proof. Assume that process $v$, in execution of action $A_8$, claims to have detected a deadlock. In the action, $v$ receives a probe $p$ with $i(p) = v$ and $s(p) = u \in IN_v$. Consider the moment immediately before $A_8$ was executed, when $p$ was still in channel $c[u, v]$. Since $u \in IN_v$, $p$ is not barren. By Lemma 13, $p$ is either the last element of $CH(v, v)$ or of $CH(v, w)$ for some aborted process $w \neq v$. In the former case, it is evident that $CH(v, v)$ forms a deadlock. Now, it is shown the later case cannot happen.

Assume the contrary. Then $\hat{p} = v$ is the last element of $proj(CH(v, w))$. By Fact 1(xvii), $v \leq w$. On the other hand, because of $CH(w, w)$'s regular property, $proj(CH(v, w))$ is a prefix of $proj(CH(w, w))$, and $\hat{p}$ is an element in $proj(CH(w, w))$. By Fact 1(xvi), $w \leq \hat{p} = v$. So, $v = w$, contradicting to the assumption that $v \neq w$. \qed

The following lemma proves the premise of Lemma 15; that is, $CH(x, x)$ is regular for every aborted process $x$. Thus, we can show the algorithm satisfies the safety condition, as shown in Theorem 5.

Lemma 16 $CH(v, v)$ is regular for every aborted process $v$.

Proof. Initially, there is no aborted process in the system and the lemma is obviously true. Assume that the lemma is true at time $t^-(A)$, where $A$ is an arbitrary action.
The lemma will be shown to be valid following action $A$, i.e., at time $t^+(A)$.

Let $v$ be any aborted process as of time $t^+(A)$. There are two possibilities: (i) $v$ is alive at time $t^- (A)$ and gets aborted during action $A$, and (ii) $v$ has already been aborted by time $t^- (A)$.

First, suppose that $v$ gets aborted during action $A$. It must be that $A = A^s_8$, and that during the action $v$ sees a probe, originally initiated by $v$ itself, coming back to $v$ and thereby claims to have detected a deadlock. The deadlock is a real one and can be described as $\text{proj}(CH(v,v))$ (by Lemma 15). Let $CH(v,v) = (p_1, \ldots , p_k)$, where $\hat{p}_1 = \text{OUT} \ v$ and $\hat{p}_k = v$. After $v$ gets aborted in action $A$, edge $(\hat{p}_{k-1}, \hat{p}_k)$ starts to shrink, with a "$D(S)$" message going from $\hat{p}_k = v$ to $\hat{p}_{k-1}$. $CH(v,v)$ becomes $(p_1, \ldots , p_{k-1})$, which is now being chased by the newly created token "$T(v,S)$". It is clear that $p_{k-1}$ satisfies the three conditions in Definition 24 and $CH(v,v)$ is regular.

Next, consider the case where $v$ is already aborted by time $t^- (A)$. It is shown that $CH(v,v)$ remains regular after action $A$. Let $u$ be the process that executes $A$. It is enough to consider only those actions which may change $CH(v,v)$’s structure:

1. $A^2_2$ ($\hat{p}_k$ sends a “$D(S)$” message, with $v \in S$, to disconnect edge $(\hat{p}_{k-1}, \hat{p}_k)$):
   In this case, $p_k$ is discarded and $CH(v,v)$ reduces to $(p_1, \ldots , p_{k-1})$, which is evidently regular.

2. $A^6_6$ (a “$D(S)$” message arrives at $\hat{p}_k$): After the action, $CH(v,v)$ is unchanged but $p_k$ becomes waiting and $v \in S$ is added to $BAR_{\hat{p}_k}$. $CH(v,v)$ remains regular.
3. \( A_{\hat{p}_j} \), \( 1 \leq j \leq k \) (\( \hat{p}_j \) aborts itself): At time \( t^-(A) \), \( CH(\hat{p}_j, \hat{p}_j) \) forms a deadlock (by Lemma 15). That cannot happen, however, unless \( j = k \) (as a consequence of Lemma 14(b)). Then, since \( \hat{p}_k \) has a non-shrinking outgoing edge and \( (p_1, \ldots, p_k) \) is regular, \( p_k \) must be in the waiting status and \( v \in BAR_{\hat{p}_k} \). When \( \hat{p}_k \) aborts itself, it sends a "D(S)" message to \( \hat{p}_{k-1} \) (if \( k > 1 \)) with \( v \in S \). Thus, after the action, \( CH(v, v) = (p_1, \ldots, p_{k-1}) \) remains regular.

4. \( A_{\hat{p}_1} \) (the chasing token "T(x,S)" arrives at \( \hat{p}_1 \)): After the action, \( CH(v, v) = (p_2, \ldots, p_k) \), which is regular.

\[ \square \]

**Corollary 5** Let \( v \) be an aborted process. \( CH(v, v) \) never increases its length.

**Theorem 5** If in execution of \( A_{\hat{p}} \) a process \( v \) claims to have detected a deadlock, then \( proj(CH(v, v)) \) formed a deadlock at the time immediately before the action was executed.

**Proof.** It follows directly from Lemmas 15 and 16. \[ \square \]

**3.5.3 Proof of Progress Condition**

In this subsection, we will prove the progress condition that any real deadlock chain is eventually detected. We describe a detailed verification plan in the following.

First, we provide some useful properties of probe chain \( CH(x_1, x_1) \) (say, in short, \( P \)) for a deadlock process chain \( X = (x_1, \ldots, x_k) \). The status of \( P \) is described by a quantitative metric \( M(P) \) consisting of several parameters: (i) the length of \( proj(P) \),
(ii) the location of the last probe $p$ in $P$ (whether it is in a channel or queue), (iii) the location of probe $p$ in the channel, if $p$ is there, and (iv) the blocking status of $p$ in $QUE_{p}$. In the case where probe $p$ is in a queue, there are two feasible reasons by which $p$ is blocked to not propagate to the next process; (i) $BAR_{p}$ is not empty, or (ii) $p$ is not primary. To represent these situations in the blocking status of the last probe $p$ in $P$, we define a set $block(P)$ that includes all dead probe chains incident on $\hat{p}$ that might possibly block the propagation of probe $p$.

Second, we derive some useful properties about a deadlock process chain $X$ and a probe chain $P$. In particular, we show that the last probe $p$ in probe chain $P$ is either in channel or waiting in queue. We then show $block(P)$ is a nonincreasing function and $\hat{p}$ eventually sends a child of $p$ to the next process along $X$ if $block(P)$ is empty.

Finally, we show that metric $M(P)$ always decreases unless deadlock chain $X$ is detected. Since the domain of possible values of $M(P)$ is finite and non-negative, deadlock process chain $X$ is eventually detected.

In the following two lemmas, we will show some useful properties of a deadlock chain.

**Lemma 17** Let $X = (x_1, \ldots, x_n)$ be a deadlock process chain. Let $P = CH(x_1, x_1) = (p_2, \ldots, p_k)$. Then, $P \neq \epsilon$, and $\hat{p}_i = x_i, 2 \leq i \leq k$.

**Proof.** By assumption, $x_1 \in IN_{x_2}$. When the last time $x_1$ was inserted into $IN_{x_2}$, apparently in an $A_5$ action at $x_2$, a new probe $p = (x_1, x_1, \cdot, \cdot)$ was added to $QUE_{x_2}$. Since then, $x_2$ has not received any token $"T(\cdot, S)"$ such that $x_1 \in S$, or $x_1$ would have seen a probe initiated by itself and thereby got aborted. Thus, $p$ is still in
$QU_{E_{22}}$. By Lemma 13, $p$ must be in a chain $CH(x_1,v)$ for some process $v$. Since $p$ is not a copy (or child) of any other probe, it is the first element of $CH(x_1,v)$. The latter does not have a chasing token, and so $CH(x_1,v)$ must be $CH(x_1,x_1)$, or, equivalently, $P$. That proves $P \neq \epsilon$ and $l = 2$. □

Definition 25 Let $X = (x_1, \ldots, x_n)$ be a deadlock process chain. Let $P = CH(x_1,x_1) = (p_2, \ldots, p_k)$ be such that $\hat{p}_i = x_i$, $2 \leq i \leq k$.

- $\text{block}(P) = \{CH(u,u) : u \text{ is an aborted process and } x_k \text{ is in } \text{proj}(CH(u,u))\}$.
- $CH(P)$ is a probe chain in $\text{block}(P)$ whose initiator is lowest among all probe chains in $\text{block}(P)$.
- The state of $P$ is a 6-tuple $M(P) = (m_1, m_2, m_3, m_4, m_5, m_6)$ where
  - $m_1 = n - k$, i.e., the distance along $X$ between $\hat{p}_k$ and $x_n$.
  - $m_2 = 0$ if $p_k \in QUE_{\hat{p}_k}$; 1 otherwise.
  - $m_3$ is the number of messages preceding $p_k$ if $p_k$ is in a channel; 0, otherwise;
  - $m_4 = |\text{block}(P)|$;
  - $m_5$ is the distance between the chasing token of $CH(P)$ and $x_k$.
  - $m_6$ is the number of messages preceding the chasing token of $CH(P)$.
- Let $\Gamma(P)$ denote the set of all possible values of $M(P)$, and let "$<"$ denote the lexicographic order between two elements in $\Gamma(P)$.
Lemma 18 Let \( X = (x_1, \ldots, x_n) \) be a deadlock process chain. Let \( P = CH(x_1, x_1) = (p_2, \ldots, p_k) \) be such that \( \hat{p}_i = x_i, \) \( 2 \leq i \leq k. \) The following are true.

(a) If \( p_k \) is in \( QUE_{p_k} \), then \( p_k \) is in the waiting status.

(b) If \( p_k \) does not change, then \( block(P) \) cannot get any new element.

(c) If \( |block(P)| = 0 \), then \( BAR_{p_k} = \emptyset \) and \( p_k \) is a primary.

Proof. (a) Assume that \( p_k \) is sent. Since \( p_k \) has no non-barren child (if not, \( P \) is not maximal probe chain), \( (x_k, x_{k+1}) \) must be shrinking (Corollary 3). The latter contradicts the assumption that \( X \) is a deadlock process chain.

(b) By Corollary 5, \( P \) never increases its length. Thus, the only possibility for \( block(P) \) to get a new element, say \( CH(v, v) \), is that \( \hat{p}_k \) is in \( CH(v, v) \) before an action and in the action \( v \) aborts itself, making \( CH(v, v) \) eligible to be in \( block(P) \). But if that is the case, then \( CH(v, v) \) forms a deadlock (by Theorem 5) before the action; that is impossible.

(c) Let \( block(P) = \emptyset \). It is first shown \( BAR_{\hat{p}_k} = \emptyset \). Assume to the contrary that \( BAR_{\hat{p}_k} \) is not empty and contains some element \( u \), which is an aborted process. Then, by Fact 1(xv), \( QUE_{\hat{p}_k} \) contains a probe, \( p \), initiated by \( u \). The probe \( p \) is not barren. Therefore, according to Lemma 13, \( p \) is an element of \( CH(u, v) \) for some aborted process \( v \) (\( v \) may or may not equal \( u \)), and \( x_k \) is thus an element of \( proj(CH(u, v)) \).

Since the latter is a subsequence of \( proj(CH(v, v)) \) (by Lemma 16), \( x_k \) is an element of \( proj(CH(v, v)) \) and \( CH(v, v) \in block(P) \), in contradiction to the assumption that \( block(P) = \emptyset \). That proves the first part of (d).

Now, it is shown \( p_k \) to be primary, again assuming \( block(P) = \emptyset \). If \( p_k \) is not
primary, then there is a primary probe \( q \) in \( QU E_{x_k} \) that has the same initiator as \( p_k \) (i.e., \( i(q) = x_1 \)). By Lemma 13, \( q \) belongs to \( CH(x_1, v) \) for some process \( v \), where either \( v = x_1 \) or \( v \) is aborted. Probe \( q \) is not an element of \( P \), since \( p_k \) is already in \( P \), which is known to have no two probes from the same \( QU E \). Therefore, \( q \) is an element of \( CH(x_1, v) \) for some aborted \( v \), and, as \( CH(v, v) \) above, \( CH(v, v) \in block(P) \), which is a contradiction. □

Now we show in the following lemma that the algorithm always works in the way to decrease the metric \( M(P) \) for a probe chain \( P = CH(x_1, x_1) \) where process \( x_1 \) is the first process in a deadlock process chain \( X = (x_1, \cdots, x_n) \).

**Lemma 19** Let \( X = (x_1, \cdots, x_n) \) be a deadlock process chain. For any \( z > 0 \) in \( \Gamma(P) \),

\[
\{\text{dead}(X) \land M(P) = z\} \text{ ensures } \{(\text{dead}(X) \land M(P) < z) \lor \neg\text{dead}(X)\}\]

**Proof.** It suffices to show a) for any action \( A \), \( \{\text{dead}(X) \land M(P) = z\} A\{(\text{dead}(X) \land M(P) \leq z) \lor \neg\text{dead}(X)\} \), and b) there is an action \( A \) such that \( \{\text{dead}(X) \land M(P) = z\} A\{(\text{dead}(X) \land M(P) < z) \lor \neg\text{dead}(X)\} \)

**Proof of (a).** Let \( A \) be any action, \( z = (z_1, z_2, z_3, z_4, z_5, z_6) > 0 \) be any element in \( \Gamma(P) \), and \( \{\text{dead}(X) \land M(P) = z\} \) be true before action \( A \). Recall that \( M(P) = (m_1, m_2, m_3, m_4, m_5, m_6) \).

It is first shown \((m_1, m_2)\) to not increase in action \( A \).

**Case 1:** \( m_2 = 1 \) at time \( t^-(A) \). In this case, \( p_k \) is in channel \( c[x_{k-1}, x_k] \). If \( m_2 \) does not change in the action, \( m_1 \) does not change either. If \( m_1 \) reduces to 0 (that
can happen only when $p_k$ arrives at $x_k$), then either $m_1$ does not change (if $m_1 > 0$) or the deadlock has been detected (if $m_1 = 0$).

**Case 2:** Suppose $m_2 = 0$ at time $t^-(A)$. It's obvious that $m_1$ will decrease if $m_2$ increases to 1. So it is enough to consider only the case where $m_2$ does not change during the action. The value of $m_2$ may possibly change only if $p_k$ is removed, which may possibly happen only in action $A_2$, $A_4$, or $A_9$. However, $A_2$ and $A_4$ cannot happen, since $OUT_{x_k} \neq \text{null}$. Since $x_{k-1}$ and $x_{k+1}$ are alive, there are no token in system like "$T(x_{k-1},.)$" or "$T(x_{k+1},.)$". Moreover, since $P$ is clean, there is no token message "$T(.,S)$", $x_1 \in S$, in $c[x_{k-1},x_k]$ at time $t^-(A)$. Thus, $A_9$ at $x_k$ can not remove $p_k$.

It has been shown $(m_1, m_2)$ to never increase in any action. Now suppose that $(m_1, m_2)$ does not change during action $A$. In this case, $m_3$ cannot increase, and by Lemma 18(c), $m_4$ cannot increase, either. Thus, $(m_1, m_2, m_3, m_4)$ does not increase.

Now suppose $(m_1, m_2, m_3, m_4)$ does not change during action $A$. It's obvious that $(m_5, m_6)$ will not increase in $A$. Therefore, $M(P)$ never increases, which proves (a).

Now prove (b). The case where $(m_2, m_3) > 0$ is trivial, and therefore it is assumed $(m_2, m_3) = 0$, i.e., $p_k$ is in the QUE of process $x_k$. Furthermore, assume $m_1 > 0$, or the deadlock has been detected and it is done.

**Case 1:** $m_4 = 0$. In this case, Lemma 18 indicates $p_k$ to be primary and waiting, and $BAR_{x_k}$ to be empty. Since $OUT_{x_k}$ is known to be not null and $i(p_k) = x_1$ is smallest in $X$, an $A_7$ action by $x_k$ will propagate $p_k$ and reduce $m_1$.

**Case 2:** $m_4 > 0$. If $m_6 > 0$, it's obvious that $m_6$ will eventually become 0. If
$m_6 = 0$, the arrival of the chasing token of $CH(P)$ will reduce $(m_4, m_5)$.

The following theorem proves the progress condition using Lemma 19 and inference rule (13).

**Theorem 6 (Progress condition)** Let $X$ be a deadlock process chain. Then, $\text{dead}(X) \rightarrow \neg \text{dead}(X)$

**Proof.** Use (13) to prove this theorem. When predicates $p$ and $q$ in (13) are matched to $\text{dead}(X)$ and $\neg \text{dead}(X)$, respectively, the premise of (13) is exactly the same as Lemma 19. Thus, the consequence of (13), $\text{dead}(X) \rightarrow \neg \text{dead}(X)$, is true.

3.6 Performance Analysis

We will use *deadlock delay time (DLT)* as a performance measure evaluating any deadlock detection and resolution algorithm. DLT is defined as the period from the time when a deadlock occurs to the time when it is resolved. Intuitively, the less DLT is, the better the system performance is regarding the performance degradation due to deadlock cycle. DLT consists of two periods: *deadlock detection time (DT)* and *deadlock resolution time (RT)*.

$DT$ denotes the period from the time when a deadlock occurs to the time when it is detected by the algorithm. As a basic time unit counting the length of period, we use the number of messages to be sent sequentially in time until a desired event happens. In the case of $DT$, the desired event is the deadlock detection by some process.
$RT$ denotes the period from the time when a deadlock detection occurs to the time when the deadlock cycle is resolved by the algorithm.

Let $L$ be the average number of processes in a deadlock cycle. Then, in our algorithm, $DT = 0.5L$, $RT = 0$, and $DLT = 0.5L$. On the other hand, in the previous algorithm [47], $DT = 0.5L$, $RT = L + 1$, and $DLT = 1.5L + 1$. Therefore, our algorithm is three times faster, in removing the deadlock, than the previous algorithm [47].

3.7 Concluding Remarks

Many distributed algorithms proposed in literature turn out to be incorrect. The main reason why such frequent mistakes occur in this area is that any deadlock detection algorithm is too complex to use an informal argument or operational approach to prove its correctness. Thus, a more strict, formal proof system is naturally required in the proof.

In this chapter, we propose an efficient deadlock detection and resolution algorithm based on probe chasing scheme. Then, a simple computational model and its proof system are proposed. The proof system turns out to be simple to use and reduce the amount of work in the proof significantly. Based on the proof system, we prove two correctness conditions of the algorithm: the safety condition that no phantom deadlock is detected, and the progress condition that any true deadlock is eventually detected.

We believe that our computational model and its proof system are very general and can be applied to the other classes of algorithms. We have proved the soundness
of our simple computational model and its proof system that are applicable to any distributed stability detection algorithms. For example, they can be applicable to the correctness proof of any distributed termination detection algorithm.
CHAPTER IV

The Complexity of Congestion-1 Embedding in a Hypercube

4.1 Overview

An embedding $f$ of a graph $G$ into a graph $H$ is a mapping of the vertices of $G$ into the vertices of $H$ in a one-to-one fashion, together with a mapping of the edges of $G$ into the simple paths of $H$ such that if $e = (u, v) \in E(G)$, then $f(e)$ is a simple path of $H$ with endpoints $f(u)$ and $f(v)$. If $f(e)$ has length greater than one, then it has one or more intermediate nodes, which are all nodes on the path other than the two endpoints. $G$ and $H$ are, respectively, the source and host graphs. For $e \in E(G)$, the dilation of $e$ under embedding $f : G \to H$ is the length of the path $f(e)$ in $H$. For $e' \in E(H)$, the congestion of $e'$ is the number of edges in $G$ with images including $e'$, i.e., the congestion of $e'$ is the cardinality of \{ $e \in E(G) : e'$ is in path $f(e)$ \}. A dilation-$l$ embedding is one for which every edge of the source graph has dilation at most $l$. Similarly, a congestion-$k$ embedding is one for which no edge of the host graph has congestion greater than $k$.

Embedding a source graph in a host graph has long been used to model the problem of processor allocation in a distributed system, where a source graph represents a distributed algorithm, with nodes representing component processes and edges rep-
resenting interprocess communications, and a host graph represents a network of processors. Bokhari [7] defined the mapping problem as the assignment of processes to processors so as to maximize the number of pairs of communicating processes that fall on pairs of directly connected processors. Alternatively, one may define the mapping problem as embedding a source graph in a host graph so as to minimize the dilation of the embedding. These two different mapping problems share a common decision problem: Can a source graph be embedded in a host graph with dilation one? This problem is known to be NP-complete even if the source graph is a tree and the host is a general graph [29], or if the source is a general graph but the host is a hypercube [1, 18]. It remains NP-complete even if the source is a tree and the host is a hypercube [82].

The importance of dilation minimization cannot be over-emphasized in a multiprocessor system in which node-to-node communications are based on store-and-forward message passing. However, this may not be the case for a hypercube that uses circuit switching for node-to-node communication. (In a circuit-switching network, when node \(A\) wants to communicate with node \(B\), the system first builds a communication circuit between \(A\) and \(B\) and then the communication starts. At the end of the communication, the circuit is released for use in other communications. Circuit switching is used in the newly released Intel iPSC/2 concurrent computer [56].) When a parallel algorithm is mapped to such a concurrent machine, an embedding \(f\) with congestion one is practically as good as one with unit dilation. Reasons: a communication circuit can be built in the system between \(f(p)\) and \(f(q)\) for all edges \((p, q)\) in the source
graph; once these circuits are built, interprocess communication between $f(p)$ and $f(q)$ will be as fast as if $f(p)$ and $f(q)$ were adjacent to each other. One significant advantage of congestion-1 embedding is that many graphs that cannot be embedded with dilation one are congestion-1 embeddable. For instance, an $n$-level full binary tree cannot be embedded in an $n$-dimensional hypercube with dilation one, but can be embedded with congestion one. It should be noted that unit dilation implies unit congestion. Thus the class of dilation-1 embeddable graphs is a proper set of the class of congestion-1 embeddable graphs.

Since the success of the Caltech Cosmic Cube project [70], the hypercube has emerged as a popular network architecture for large scale concurrent computers, and the problem of embedding a source graph in a hypercube has attracted the attention of many researchers [5, 6, 32, 52, 81, 85]. In this paper, we study the possible impact of the new direct-connect communication technology on the complexity of the embedding problem on a hypercube. As mentioned above, congestion-1 embedding is as good as dilation-1 embedding in a hypercube employing such a communication mechanism. Thus, we consider the following problem.

**CONGESTION-1 EMBEDDING**

**Instance:** A graph $G$, an $n$-dimensional hypercube $H_n$.

**Question:** Is there a congestion-1 embedding of $G$ into $H_n$?

We show that any graph $G = (V, E)$ can be embedded in $H_n$ with congestion one, provided $n \geq \max\{6\log|V|, G\}$. (In contrast, not every graph can be embedded with dilation one in a hypercube, and for a given graph $G$ it is NP-complete to deter-
mine whether there exists a hypercube $H$ such that $G$ is dilation-1 embeddable in $H$ [1, 18].) Then we show that CONGESTION-1 EMBEDDING as defined above is NP-complete, even if the source graph is connected. The restriction to connected graphs is important because in practice a graph representing a parallel algorithm is usually connected. It should be noted that an $n$-dimensional hypercube may be represented either by a $2^n \times 2^n$ adjacency matrix or simply by the integer $n$. CONGESTION-1 EMBEDDING apparently belongs to $NP$ if the adjacency matrix is used to represent the hypercube. We shall show that CONGESTION-1 EMBEDDING belongs to $NP$ even if the hypercube is represented by the integer $n$. On the other hand, it would be of little interest to show the problem NP-hard with the hypercube represented by the dimension $n$, for, any way, the problem has to deal with $2^n$ nodes. We shall show the embedding problem NP-hard even if the hypercube is represented by an adjacency matrix. Thus, unless $P = NP$, the time complexity of CONGESTION-1 EMBEDDING cannot be bounded by a polynomial in $2^n$.

4.2 Terminology and Definitions

An undirected graph with $2^n$ nodes is called an $n$-dimensional hypercube or $n$-cube and is denoted by $H_n$, if its nodes can be labeled 0 through $2^n - 1$ such that two nodes are joined by an edge iff their labels in binary representation differ by exactly one bit. Saad and Schultz [67] have shown that there are $n!2^n$ different ways to label the $2^n$ nodes of an $n$-cube so as to conform to the above definition; each of them is called a labeling on $H_n$ and is a bijective function from $V(H_n)$ to \{0, 1, ..., $2^n - 1$\}. In this chapter, every hypercube is assumed to have been legitimately labeled, and
unless necessary we do not rigorously distinguish between a node and its label. Thus, for instance, node 1 refers to the node labeled 1.

If the label of a node \( x \in V(H_n) \) is \( b_0b_1\ldots b_{n-1} \) in binary representation (i.e., \( \text{label}(x) = \sum_{i=0}^{n-1} b_i2^{n-i-1} \)), then \( b_i, 0 \leq i \leq n - 1 \) is said to be the bit of \( x \) in dimension \( i \), or simply bit \( i \). Note that the \( n \) dimensions are numbered 0 through \( n - 1 \) from left to right. If \( (x, y) \) is an edge in \( H_n \) and (the labels of) \( x \) and \( y \) differ in dimension \( i \), then \( (x, y) \) is said to be along that dimension. A path \( (x_1, x_2, \ldots, x_k) \) is said to move from \( x_1 \) to \( x_k \) along dimensions \( d_1, \ldots, d_{k-1} \), in that order, if edge \( (x_i, x_{i+1}) \) is along dimension \( d_i \) for all \( i, 1 \leq i \leq k - 1 \). In that case, the path is also said to move from \( x_k \) to \( x_1 \) along dimensions \( d_{k-1}, \ldots, d_1 \).

A \( k \)-subcube of an \( n \)-cube is a subgraph that is itself a \( k \)-dimensional hypercube. A subcube of \( H_n \) can be represented by a string of \( n \) bits with each bit being either 0 or 1 or \(*\), where \(*\) is said to be a “don’t care” bit. Such a string representing a \( k \)-subcube will contain exactly \( k \) \(*\)'s. For instance, “1*0**0” denotes the 3-dimensional subcube in \( H_6 \) which has \( \{1x_10x_3x_40 : x_1, x_3, x_4 = 0 \text{ or } 1\} \) as its vertex set.

The Hamming distance between two nodes \( u \) and \( v \), denoted as \( \text{Dist}(u, v) \), is the number of dimensions (bits) in which the two nodes differ. The Hamming distance, or simply distance, between two subgraphs \( G_1 \) and \( G_2 \) on \( H_n \) is

\[
\text{Dist}(G_1, G_2) \overset{\text{def}}{=} \min\{\text{Dist}(u, v) : u \in V(G_1), v \in V(G_2)\}.
\]

The degree of a node \( v \), denoted \( \deg(v) \), is the number of edges incident upon \( v \). The degree of a graph \( G \), denoted \( \deg(G) \), is the maximum degree of a vertex in \( G \). The length of a path in a graph is the number of edges in the path. A linear chain is a connected
graph in which every node has degree two, except for two nodes each of which has degree one. A \textit{k-chain} is a linear chain of \textit{k} nodes.

If \(G_1\) and \(G_2\) are two graphs, let \(G_1 \cup G_2\) denote the graph \(G\) such that \(V(G) = V(G_1) \cup V(G_2)\) and \(E(G) = E(G_1) \cup E(G_2)\). The union of more than two graphs is defined similarly.

\textsc{Congestion-1 Embedding} will be shown to be \textsc{NP}-hard by reducing from the following problem.

3-PARTITION

\textbf{Instance:} A positive integer \(b\) and a set of \(3m\) integers \(A = \{a_0, a_1, \ldots, a_{3m-1}\}\) such that \(216 < (b + 1)/4 < a_i < b/2\) and \(\sum_{i=0}^{3m-1} a_i = mb\).

\textbf{Question:} Can \(A\) be partitioned into \(m\) disjoint sets \(S_0, \ldots, S_{m-1}\) such that, for \(0 < j < m - 1\), \(\sum_{a \in S_j} a = b\). (As noticed in [29] each \(S_j\) must contain exactly three elements.)

The sets \(S_0, \ldots, S_{m-1}\) are said to be a 3-partition of \(A\) if \(\sum_{a \in S_j} a = b\) for all \(j\). The 3-partition problem is known to be \textsc{NP}-complete in the strong sense [29]. Although the 3-partition problem as stated above is slightly different from the one defined in [29] (where each \(a_i\) is only assumed to satisfy \(b/4 < a_i < b/2\)), the more restricted condition employed in our definition can be easily shown to have no effect on the strong \textsc{NP}-completeness of the problem.
4.3 Embedding a General Graph

This section considers CONGESTION-1 EMBEDDING for general graphs. We first show that any graph $G = (V, E)$ can be embedded in $H_n$ with congestion one, provided $n \geq \max\{6\log|V|, G\}$. Based on this result, we show CONGESTION-1 EMBEDDING to be in $NP$, even if $H_n$ is represented by the integer $n$. Then we show the problem to be NP-hard even if $H_n$ is represented as a $2^n \times 2^n$ adjacency matrix.

4.3.1 Upper Bounds

As mentioned in the introduction, for a given graph $G$, it is NP-complete to determine whether there is a hypercube $H$ such that $G$ is dilation-1 embeddable in $H$. In contrast, when it comes to congestion-1 embedding, every graph is embeddable in some hypercube. To see this, let $n = |V(G)| - 1$ and $V(G) = \{0, 1, \ldots, n - 1\}$, and embed $G$ in $H_n$ as follows. Map node 0 in $G$ to node 0 in $H_n$ and for $i \geq 1$, map node $i$ of $G$ to node $2^{i-1}$ of $H_n$; and for every edge $(i, j)$ in $G$, map $(i, j)$ to the shortest path between the two endpoints' images in $H_n$ that does not include node 0 as an intermediate node. This mapping is readily seen to have congestion one and dilation two. In the following, we obtain a more interesting upper bound on the dimension of the smallest hypercube into which a given graph $G$ is congestion-1 embeddable, i.e., an upper bound on $\min\{n : G$ is congestion-1 embeddable in $H_n\}$. Later on we shall use this result to show CONGESTION-1 EMBEDDING to be in $NP$. 
Theorem 7 Let $G = (V, E)$ be an arbitrary graph. If $n \geq \max\{6[\log |V|], G\}$, then $G$ can be embedded in $H_n$ with congestion one and dilation at most $n + 2$.

Proof. Let $G = (V, E)$ be an arbitrary graph and $n = \max\{6[\log |V|], G\}$. We will embed $G$ in $H_n$ with congestion one. Without loss of generality, let us assume that $\log |V|$ is an integer and $n = 6 \log |V| \geq G$. (If necessary, nodes of degree zero can be added to $G$ so as to validate the assumption. If the modified graph is congestion-1 embeddable then so is the original.)

Think of each edge $(u, v) \in E$ as having two half-edges, one being incident upon $u$ and the other upon $v$. Paint each half-edge in the graph with a color such that no two half-edges incident upon a common node in $V$ are of the same color. (It doesn't matter whether the two halves of an edge are painted differently or with the same color.) Since $G \leq n$, each node has at most $n$ half-edges incident upon it; so, $n$ colors are sufficient to color the half-edges of $G$ as above. (We are aware of Vizing's Theorem [4]: there exists an edge coloring $C : E \to \{1, \ldots, n + 1\}$ such that $C(e_1) \neq C(e_2)$ whenever $e_1$ and $e_2$ have a common endpoint. We do not apply it because we can afford to use only $n$ colors.) For each edge $(u, v) \in E$, let $\text{color}(u, v)$ denote the color of the half-edge that is incident upon $u$, and $\text{color}(v, u)$ the color of the other half-edge (which is incident upon $v$).

Let the $n$ colors be numbered 0 through $n - 1$ so that each color corresponds to a dimension of $H_n$ or to a bit position in the string representation of a node. That will allow us to use phrases such as "dimension color$(u, v)$" or "bit color$(u, v)$."
Let \( k = n/6 = \log |V| \) and let \( V = \{0, 1, \ldots, 2^k - 1\} \). With \( G \)'s half-edges being colored as above, a congestion-1 embedding \( f : G \to H_n \) can be defined as following. For \( u \in V \), let \( f(u) \) be the node in \( H_n \) whose binary representation is the concatenation of six copies of \( u \), with each copy being the \( k \)-bit binary representation of \( u \). We shall write
\[
f(u) = u_0 u_1 u_2 u_3 u_4 u_5
\]
when it is necessary to distinguish between different copies of \( u \), and \( u_i \) will be referred to as component \( i \) of \( f(u) \). In general, for a node \( A \) in \( H_n \) with binary representation \( a_0a_1 \ldots a_{n-1} \), the bit string \( a_ia_ia_{i+1} \ldots a_{i+1}k_1 \ldots k_{i-1}_k-1 \) will be referred to as component \( i \) or the \( i \)th component of node \( A \), where \( 0 \leq i \leq 5 \). For an edge \( (A, B) \) in \( H_n \), if node \( A \) differs from node \( B \) by a bit in component \( i \), then the edge is said to be in component \( i \).

We now define \( f(u, v) \) for \( (u, v) \in E \). Let \( \alpha(u, v), \beta(u, v), \gamma(u, v), \alpha(v, u), \beta(v, u), \gamma(v, u) \) be distinct elements in \( \{0, 1, \ldots, 5\} \), except \( \alpha(u, v) \) and \( \alpha(v, u) \) may possibly be equal, such that \( \alpha(u, v) = \lceil \text{color}(u, v)/k \rceil \) and \( \alpha(v, u) = \lceil \text{color}(v, u)/k \rceil \). (Thus, in the binary representation of \( f(u) \), bit \( \text{color}(u, v) \) is in component \( \alpha(u, v) \).) The path \( f(u, v) \) is the concatenation of three subpaths as described below.

1. Construct a path in \( H_n \) that starts at node \( f(u) \), moves along dimension \( \text{color}(u, v) \), and then moves in increasing order (i.e., lower dimensions first) along those dimensions in which \( u_{\beta(u, v)} \) and \( v_{\beta(u, v)} \) (i.e., \( f(u) \)'s and \( f(v) \)'s \( \beta(u, v) \)th components) are different, and then moves in an increasing order along those dimensions in which \( u_{\gamma(u, v)} \) differs from \( v_{\gamma(u, v)} \). Let \( f'(u) \) denote the
destination node of the path.

2. Similarly, construct a path that starts at \( f(v) \), moving along dimension \( \text{color}(v,u) \), along those dimensions in which \( v_{\beta(v,u)} \) differs from \( u_{\beta(v,u)} \), and then along those dimensions in which \( v_{\gamma(v,u)} \) differs from \( u_{\gamma(v,u)} \). Let \( f'(v) \) denote the destination node of the path.

3. Join \( f'(u) \) to \( f'(v) \) with a path that moves in an increasing order along those dimensions in which the two nodes differ.

4. Let \( f(u,v) \) be the three paths joined together.

Fig. 24 shows two such paths, \( f(u,v) \) and \( f(x,y) \), with \( (\alpha(u,v), \beta(u,v), \gamma(u,v)) = (0,1,2), (\alpha(v,u), \beta(v,u), \gamma(v,u)) = (5,4,3), (\alpha(x,y), \beta(x,y), \gamma(x,y)) = (1,0,2) \), and \( (\alpha(y,x), \beta(y,x), \gamma(y,x)) = (1,4,3) \). The updown arrows indicate where bits change, and \( u'_0, v'_5, x'_1, y'_1 \) denote the components that differ from \( u_0, v_5, x_1, y_1 \), respectively, in dimensions \( \text{color}(u,v), \text{color}(v,u), \text{color}(x,y), \text{color}(y,x) \). Thus, for instance, the subpath \( \text{Path}(f(u), f'(u)) \) moves from \( f(u) \) to \( f'(u) \) by changing bit \( \text{color}(u,v) \) in \( u_0 \), changing \( u_1 \) to \( v_1 \), and then changing \( u_2 \) to \( v_2 \).

We now show that the embedding as defined above has congestion one. Let \( (u,v) \) and \( (x,y) \) be any two different edges in \( G \). Without loss of generality, assume that the four nodes \( u,v,x,y \) are all distinct except that \( u \) and \( x \) may possibly be equal. We show \( f(u,v) \) and \( f(x,y) \) edge-disjoint by showing (1) that \( \text{Path}(f(u), f'(u)) \) and \( \text{Path}(f(x), f'(x)) \) are edge-disjoint and (2) that \( \text{Path}(f'(u), f(v)) \) and \( \text{Path}(f(x), f(y)) \) are edge-disjoint and so are \( \text{Path}(f(u), f(v)) \)
and \( \text{Path}(f'(x), f(y)) \).

First, consider \( \text{Path}(f(u), f'(u)) \) and \( \text{Path}(f(x), f'(x)) \). There are two possibilities: \( u \neq x \) or \( u = x \). If \( u \neq x \), then every node on \( \text{Path}(f(u), f'(u)) \) has three copies of \( u \) (see Fig. 24), which none of the nodes on \( \text{Path}(f(x), f'(x)) \) may possibly have, so the two subpaths are edge disjoint. Now, suppose \( u = x \). Recall that the first edge of \( \text{Path}(f(u), f'(u)) \) and that of \( \text{Path}(f(x), f'(x)) \) are along dimensions \( \text{color}(u,v) \) and \( \text{color}(x,y) \), respectively, with \( \text{color}(u,v) \neq \text{color}(x,y) \). Either \( \alpha(u,v) = \alpha(x,y) \) or \( \alpha(u,v) \neq \alpha(x,y) \).

Case 1: \( \alpha(u,v) = \alpha(x,y) \). In this case, except for \( f(u) \) which equals \( f(x) \), every node on \( \text{Path}(f(u), f'(u)) \) differs from every node on \( \text{Path}(f(x), f'(x)) \), at least in dimensions \( \text{color}(u,v) \) and \( \text{color}(x,y) \). Therefore, the two subpaths are edge disjoint.

Figure 24: Example paths: \( f(u,v) \) and \( f(x,y) \)
Case 2: $\alpha(u, v) \neq \alpha(x, y)$. Fig. 24 illustrates the situation. Assume, for contradiction, that the two subpaths $Path(f(u), f'(u))$ and $Path(f(x), f'(x))$ are not edge disjoint and therefore share a common edge, say, $(A, B)$. Assume $Dist(A, f(u)) < Dist(B, f(u))$. Then $A$ is an intermediate node of both subpaths. Observe the following:

1. On $Path(f(u), f'(u))$, from $f(u)$ to $f'(u)$, components $\alpha(u, v)$, $\beta(u, v)$, $\gamma(u, v)$ change from $u_{\alpha(u,v)}$, $u_{\beta(u,v)}$, $u_{\gamma(u,v)}$ to $u'_{\alpha(u,v)}$, $v_{\beta(u,v)}$, $v_{\gamma(u,v)}$, successfully, in that order. Similarly, components $\alpha(x, y)$, $\beta(x, y)$, $\gamma(x, y)$ change from $x_{\alpha(x,y)}$, $x_{\beta(x,y)}$, $x_{\gamma(x,y)}$ to $x'_{\alpha(x,y)}$, $y_{\beta(x,y)}$, $y_{\gamma(x,y)}$, successfully, on $Path(f(x), f'(x))$. ($u'_{\alpha(u,v)}$ and $x'_{\alpha(x,y)}$ are the components which differ from $u_{\alpha(u,v)}$ and $x_{\alpha(x,y)}$ in bit $color(u,v)$ and bit $color(x,y)$, respectively.)

2. Component $\alpha(u, v)$ and component $\alpha(x, y)$ of $A$ equal $u'_{\alpha(u,v)}$ and $x'_{\alpha(x,y)}$, respectively. (Every intermediate node of $Path(f(u), f'(u))$ has component $\alpha(u, v)$ equal to $u'_{\alpha(u,v)}$ and every intermediate node of $Path(f(x), f'(x))$ has component $\alpha(x, y)$ equal to $x'_{\alpha(x,y)}$.)

3. Edge $(A, B)$ is neither in component $\alpha(u, v)$ nor in component $\alpha(x, y)$. ($Path(f(u), f'(u))$ (resp. $Path(f(x), f'(x))$) has only one edge in component $\alpha(u, v)$ (resp. $\alpha(x, y)$), namely, its first edge, which is apparently not a common edge, since $color(u,v) \neq color(x,y)$.)

Thus, by the time $Path(f(u), f'(u))$ arrives at node $A$, two components ($\alpha(u, v)$ and $\alpha(x, y)$) have each changed one bit; and $(A, B)$ is in neither of them. So, $\beta(u, v) =$
\(\alpha(x,y)\), and \((A, B)\) is in component \(\gamma(u, v)\). Besides, \(v_{\beta(u,v)} = x'_{\alpha(x,y)}\). Similarly, \\
\(\beta(x,y) = \alpha(u,v)\), \((A, B)\) is in component \(\gamma(x,y)\) (so, \(\gamma(u,v) = \gamma(x,y)\)), and \(y_{\beta(x,y)} = u'_{\alpha(u,v)}\). Consequently, \(Path(f(u), f'(u))\) and \(Path(f(x), f'(x))\) each have only three edges, with \((A, B)\) being their common third edge. As \(Path(f(u), f'(u))\) moves along edge \((A, B)\), component \(\gamma(u, v)\) changes from \(u\) to \(v\); and as \(Path(f(x), f'(x))\) moves along edge \((A, B)\), component \(\gamma(x, y) = \gamma(u, v)\) changes from \(x = u\) to \(y\). That is impossible, since \(v \neq y\). Thus the assumption is false that \(Path(f(u), f'(u))\) and \(Path(f(x), f'(x))\) share a common edge.

Next, consider \(Path(f'(u), f(v))\) and \(Path(f(x), f(y))\). Every node on \(Path(f'(u), f(v))\) has two copies of \(v\). A node on \(Path(f(x), f(y))\) may happen to have two copies of \(v\), but not two consecutive nodes. Therefore, \(Path(f'(u), f(v))\) and \(Path(f(x), f(y))\) are edge disjoint, and similarly \(Path(f(u), f(v))\) and \(Path(f'(x), f(y))\) are edge disjoint.

Thus, \(f(u, v)\) and \(f(x, y)\) are edge disjoint and embedding \(f\) has congestion one. It is not hard to see that the embedding has dilation at most \(n + 2\) and that an edge \((u, v)\) has dilation \(n + 2\) iff \(\text{color}(u, v) = \text{color}(v, u)\) and \(u\) and \(v\) are different in every dimension. \(\square\)

We have shown that every graph \(G = (V, E)\) can be embedded with unit congestion in a hypercube of dimension \(n = \max\{6|\log |V||, G\}\). It is not known whether \(G\) can be embedded (with unit congestion) in a hypercube of dimension less than \(n\), especially when \(6|\log |V|| > G\).
CONGESTION-1 EMBEDDING obviously belongs to \( NP \) if \( H_n \) is represented by a \( 2^n \times 2^n \) adjacency matrix. It is however non-trivial to see whether the problem is in \( NP \), if \( H_n \) is represented by integer \( n \). The difficulty is that when a nondeterministic Turing machine guesses a path for an edge, the path may have length up to \( 2^n - 1 \); thus, if the given graph has only a small number of nodes or edges (so that the input size is far less than \( 2^n \)), then the nondeterministic algorithm could be non-polynomial in input size. Fortunately, due to Theorem 7, no guess is necessary when a graph is sufficiently small compared to the given hypercube, and the embedding problem is really in \( NP \) as shown below.

**Theorem 8** CONGESTION-1 EMBEDDING belongs to \( NP \) even if \( H_n \) is represented by the integer \( n \).

**Proof.** Let graph \( G = (V, E) \) and integer \( n \) together be a given instance of CONGESTION-1 EMBEDDING. The size of the instance is at least \( |E| + \log n \). We show that a nondeterministic Turing machine can determine if \( G \) is congestion-1 embeddable in \( H_n \) in a time bounded by a polynomial in \( |E| \) and \( \log n \).

If \( G > n \) or \( |V| > 2^n \), then definitely \( G \) cannot be embedded in \( H_n \) with congestion one.

If \( G \leq n \) and \( |V| \leq 2^n \), it is not hard to see that \( G \) is congestion-1 embeddable in \( H_n \) iff the graph \( G' = (V', E') \) is congestion-1 embeddable in \( H_n \), where \( E' = E \) and \( V' = \{ v \in V : v > 0 \} \). So it suffices to determine if \( G' \) is congestion-1 embeddable in \( H_n \). Consider two possible cases.
Case 1: \( G' \) \( \leq n \) and \( 6 \lceil \log |V'| \rceil \leq n \). In this case, Theorem 7 guarantees that \( G' \) can be embedded in \( H_n \) with congestion one.

Case 2: \( G' \) \( \leq n \) and \( 6 \lceil \log |V'| \rceil > n \). In this case, a nondeterministic Turing machine can determine if \( G' \) is congestion-1 embeddable in \( H_n \) as follows.

1. For every vertex \( x \in V' \), define \( f(x) \in V(H_n) \) by generating an \( n \)-bit integer.
2. For every edge \( (x, y) \in E' \), generate a simple path \( f(x, y) \) of length at most \( 2^n - 1 \).
3. Check if \( f \) is an embedding with congestion one.

The time complexity of this nondeterministic algorithm is readily seen to be bounded above by a polynomial in \( 2^n \), say \( P(2^n) \). Since \( 6 \lceil \log |V'| \rceil > n \), it follows that \( |V'|^6 \geq 2^{n-6} \) and \( P(2^n) = O(P(|V'|^6)) \). Since the size of the input is at least \( |E| = |E'| \geq \frac{1}{2}|V'| \), the time complexity of the above algorithm is bounded by a polynomial in input size. Therefore, CONGESTION-1 EMBEDDING belongs to \( NP \). \( \square \)

4.3.2 NP-Completeness of Congestion-1 Embedding

We have shown that CONGESTION-1 EMBEDDING belongs to \( NP \). In this section, we show the problem NP-complete by constructing a pseudo-polynomial transformation from 3-PARTITION. The basic idea behind the transformation is to construct a skeleton graph that does not change much of its structure when embedded in a hypercube with congestion one. Similar ideas have been used in \([18]\) to establish the NP-completeness of dilation-1 embedding. The skeleton graph constructed in \([18]\),
which remains unchanged under a dilation-1 embedding, is not applicable in our case because its structure may remarkably change in a congestion-1 embedding. We construct in the following a skeleton graph that essentially remains invariant under any congestion-1 embedding into a hypercube.

For given integers \( m, b \), our skeleton graph, denoted \( G_{m,b} \), is constructed from a hypercube by removing a number of subcubes. Specifically, let \( r = \lceil \log \max\{m, b\} \rceil \) and \( n = 11r + 11 \); then \( G_{m,b} \) is the graph obtained from an \( n \)-cube by removing all subcubes \( Q_j, 0 \leq j \leq m - 1 \) as well as all edges that have an endpoint in \( \bigcup_j Q_j \), where \( Q_j \) (\( 0 \leq j \leq m - 1 \)) denotes the following subcube:

\[
Q_j = \overline{\text{binary}(j) \text{binary}(j) \cdots \text{binary}(j) \ast \cdots \ast 000 \cdots 0}^{3r + 1} \quad (4.1)
\]

where binary\((j)\) is the \( r \)-bit binary representation of integer \( j \) (note that \( r \) bits are sufficient to represent \( j \), since \( r \geq \log m \geq \log j \)). The reason for repeating binary\((j)\) that many times is to ensure that every two \( Q \) subcubes are at least a Hamming distance eight apart, which is needed in order to prove Theorem 9. The rightmost ten 0’s are really not necessary; they are introduced here mainly to conform to the case of connected graphs (see Eq. 4.2 in Section 4.4.1. For precisely the same reason, \( Q_j \) is defined to have \( 3r + 1 \) rather than \( r + 2 \ast \)’s, although the latter will be sufficient. (It is crucial for \( Q_j \) to have at least \( 2b + 1 \) nodes; a subcube of dimension \( r + 2 \) is large enough to meet that requirement.)

If \( L \) is a labeling on the \( n \)-cube from which \( G_{m,b} \) was constructed, then, when restricted to \( V(G_{m,b}) \), \( L \) is a labeling on \( G_{m,b} \). In that case, \( G_{m,b} \) is simply said to be labeled by \( L \).
Figure 25: (a) Skeleton graph $G_{m,b}$. (b) Its image under a congestion-1 embedding.

**Definition 26** For any nonnegative integer $k$, let $k$-Exterior($G_{m,b}$) denote the subgraph of $G_{m,b}$ that is induced on the vertex set $\{v \in V(G_{m,b}) : \text{Dist}(v, \bigcup_i Q_i) \geq k\}$.

Our NP-completeness proof hinges on the following theorem, which indicates that, under any congestion-one embedding into $H_n$, $2$-Exterior($G_{m,b}$) remains invariant. The proof of this theorem is deferred to Section 4.5, where it will follow from the line of reasoning developed to prove the corresponding (and more complicated) theorem for the case of connected graphs.

**Theorem 9** Let $G_{m,b}$ be labeled by $L$. For any congestion-one embedding $f : G_{m,b} \to H_n$, there is a labeling $L'$ on $H_n$ such that, for every node $u$ and every edge $(v,w)$ in $2$-Exterior($G_{m,b}$), $L(u) = L'(f(u))$ and $f(v,w) = (f(v), f(w))$.

In other words, if $f : G_{m,b} \to H_n$ is a congestion-1 embedding, then $2$-Exterior($G_{m,b}$) is embedded by $f$ with dilation one. Let $\bar{Q}_j = V(Q_j) \cup \{x \in V(G_{m,b}) : \text{Dist}(x, Q_j) = 1\}$; thus $\bar{Q}_j$ consists precisely of all nodes in and all nodes
"around" (i.e., with distance one from) $Q_j$. Let $\bar{Q}_j'$ denote the corresponding set in $H_n$. Since $\text{Dist}(Q_i, Q_j) \geq 8$, it follows from Theorem 9 that, for each $j$, $\bar{Q}_j' - V(Q_j)$ must be mapped by $f$ into $\bar{Q}_j'$. As a result, each $\bar{Q}_j'$ has exactly $|V(Q_j)|$ nodes not contained in $f(G_{m,b})$, which may be regarded as forming a "hole." So the image of $G_{m,b}$ has $m$ isolated "holes," each of size $|V(Q_j)|$. Fig. 25 pictorially shows a congestion-1 embedding of $G_{m,b}$ into $H_n$, where in (a) the white regions represent the removed $Q_i$'s, the dotted area represents $2\text{-Exterior}(G_{m,b})$, and the black rectangles represent those nodes of $G_{m,b}$ which are not in $2\text{-Exterior}(G_{m,b})$; their images in $H_n$ are correspondingly colored in (b). Note that $2\text{-Exterior}(G_{m,b})$ is invariant under the embedding. The black rectangle around a $Q_i$ (which represents all nodes with distance one to $Q_i$) is in general not invariant under the embedding, but must be embedded in $\bar{Q}_j'$. Also note that the "holes" may have irregular shapes.

**Theorem 10** \textit{CONGESTION-1 EMBEDDING} is NP-hard even if the $H_n$ is given as a $2^n \times 2^n$ adjacency matrix.

**Proof.** The proof is by means of reducing 3-PARTITION to CONGESTION-1 EMBEDDING. Suppose that an instance of 3-PARTITION is given: positive integers $b$ and $m$, and a set of positive integers $A = \{a_0, \ldots, a_{3m-1}\}$ such that $\frac{1}{4}(b+1) < a_i < \frac{1}{2}b$ for all $a_i$, $0 \leq i \leq 3m-1$ and $\sum_{i=0}^{3m-1} a_i = bm$. We construct an instance of CONGESTION-1 EMBEDDING, which consists of a graph $G$ and an $n$-cube $H_n$, such that $A$ has a 3-partition iff $G$ can be embedded in $H_n$ with congestion one.

Let $n = 11r + 11$, where $r = \lceil \log \max\{m, b\} \rceil$. Let $G$ consist precisely of the following components:
• a skeleton graph $G_{m,b}$ as defined above;

• $m$ linear chains with $2^{3r+1} - b$ nodes each (such chains will be used to reduce the size of each "hole" from $2^{3r+1}$ to $b$);

• $3m$ linear chains with $a_0, a_1, \ldots, a_{3m-1}$ nodes, respectively (each chain represents an integer in $A$).

The skeleton graph and the $4m$ linear chains are isolated from each other. That is, the skeleton graph forms a connected component of $G$, and so does each of the linear chains. (A subgraph of $G$ is a connected component if it is connected and is not a proper subgraph of any connected subgraph of $G$.)

Now, if $A$ can be partitioned into disjoint sets $S_0, \ldots, S_{m-1}$ such that $\sum_{a \in S_j} a = b$ for $0 \leq j \leq m - 1$, then a congestion-1 embedding of $G$ into $H_n$ can be constructed as follows.

1. Embed the skeleton graph $G_{m,b}$ into $H_n$ through the identity mapping (which apparently has congestion one). That will leave the $m$ subcubes $Q_0, \ldots, Q_{m-1}$ unoccupied.

2. Embed a $(2^{3r+1} - b)$-chain in each $Q_j$, $0 \leq j \leq m - 1$. That will leave exactly $b$ nodes unoccupied in each $Q_j$.

3. For $0 \leq j \leq m - 1$, embed in $Q_j$ the three $a_i$-chains that represent the three integers in $S_j$.

Note that step 2 and step 3 can be achieved with congestion one, for the four chains embedded in a $Q_j$ totally have only $2^{3r+1}$ nodes and it is well known that any linear
chain with $2^k$ nodes can be embedded in a $k$-cube with dilation one.

Conversely, if $G$ can be embedded by $f$ into $H_n$ with congestion one, then a solution to 3-PARTITION can be extracted as follows. By Theorem 9, the image of $G_{m,b}$ leaves $m$ isolated "holes" in $H_n$; each hole can accommodate $2^{3r+1}$ nodes (see Fig. 25(b), where the white regions are the holes). The $4m$ linear chains in $G$ must be embedded in these holes. Since these holes are isolated from each other, each linear chain must be embedded entirely within a single hole. So, under the embedding, the $4m$ linear chains are partitioned into $m$ groups, with each group embedded in a hole. The total number of nodes in each group must be $2^{3r+1}$. Since $2(2^{3r+1} - b) > 2^{3r+1}$, each group includes exactly one $(2^{3r+1} - b)$-chain; therefore, in each group the $a_i$-chains altogether have a total of $b$ nodes. With all the $(2^{3r+1} - b)$-chains excluded, the $m$ groups of chains immediately define a partition of $A$ into $m$ disjoint sets, with each set of integers summing up to $b$.

We have shown that $A$ has a 3-partition iff the graph $G$ as constructed above can be embedded in $H_n$ with congestion one. Representing $G$ with a $2^n \times 2^n$ adjacency matrix, it is not hard to see that the computing time needed to construct $G$ is bounded by some polynomial in $2^n$. Since $2^n = O((m + b)^{11})$, the above transformation is pseudo-polynomial. Since 3-PARTITION is NP-complete in the strong sense, the NP-hardness of CONGESTION-1 EMBEDDING is thus established. □

The embedding $f$ constructed in the above proof has dilation one. Thus, the same proof can be used to establish the following.

**Corollary 6** For any constant $k$, the problem of determining whether a graph can be
embedded in a given n-cube with congestion-1 and dilation-k is NP-hard.

4.4 Embedding a Connected Graph

We have shown CONGESTION-1 EMBEDDING to be NP-hard. In the NP-hardness proof, the graph constructed from a given instance of 3-PARTITION is disconnected. Since embedding connected graphs is far more important in applications than disconnected graphs, it is important to know whether CONGESTION-1 EMBEDDING is still NP-hard when restricted to connected graphs. In this section, we modify the NP-hardness proof of the previous section and answer the above question in the affirmative. The modification is non-trivial. But the basic idea is to attach the \( a_i \)-chains to the skeleton graph (so that it becomes connected), and to open a path between each \( a_i \)-chain and each subcube \( Q_j \) so that any \( a_i \)-chain can be embedded in any subcube \( Q_j \). What is most important is that the modified skeleton graph must remain invariant under any congestion-1 embedding.

4.4.1 Construction

Suppose we are given an instance of 3-PARTITION: positive integers \( b, m \), and a set of positive integers \( A = \{a_0, a_1, \ldots, a_{3^m-1}\} \) such that \( 216 < (b+1)/4 < a_i < b/2 \) for all \( a_i \) and \( \sum_{i=0}^{3^m-1} a_i = bm \). We want to construct a connected graph \( G \) and an integer \( n \) such that \( A \) has a 3-partition iff \( G \) can be embedded in \( H_n \) with congestion one.

Let \( r = \lceil \log \max\{b, 6m\} \rceil \), and \( n = 11r + 11 \). (Since \( (b+1)/4 > 216 \), \( r \geq 10 \).) We shall construct a graph \( G \) from an \( n \)-cube by removing a number of nodes and edges from the cube and then attaching a number of chains to it.
We first define a number of subcubes: \( R_j, Q_j, A_i \), for \( 0 \leq j \leq m - 1 \) and \( 0 \leq i \leq 3m - 1 \). The subcubes \( Q_j \) will play the same role as in Section 4.3. The subcubes \( A_i \) will be the places at which the \( a_i \)-chains are attached to the skeleton graph. The role of subcubes \( R_j \) will be explained in due course.

For \( 0 \leq k \leq 2^r - 1 \), let \( \text{binary}(k) \) denote the \( r \)-bit binary representation of the integer \( k \). Define subcubes \( R_j, Q_j, A_i \) for \( 0 \leq j \leq m - 1 \) and \( 0 \leq i \leq 3m - 1 \) as follows.

\[
R_j = \underbrace{\text{binary}(j) \ldots \text{binary}(j)}_{8r} \underbrace{** \ldots *}_{3r + 1} \underbrace{0000}_{10}
\]

\[
Q_j = \underbrace{\text{binary}(j) \ldots \text{binary}(j)}_{8r} \underbrace{** \ldots * 00 \ldots 00}_{3r + 1} \underbrace{00 \ldots 00}_{10} \quad (4.2)
\]

\[
A_i = \underbrace{\text{binary}(m + i) \ldots \text{binary}(m + i)}_{8r} \underbrace{** \ldots * 00 \ldots 00}_{3r + 1} \underbrace{00 \ldots 00}_{10}
\]

Note that the Hamming distance between any two of these subcubes is at least 8, except that \( Q_j \) is a subcube of \( R_j \) and so their Hamming distance is zero. Each \( A_i \) or \( Q_j \) has dimension \( 3r + 1 \), while each \( R_j \) has dimension \( 3r + 7 \).

The following lemma indicates that in a \((3r + 1)\)-cube there exists a simple path of length \( b - 1 \) and a set of \( 3m \) nodes such that every two nodes in the set are a Hamming distance of six or more apart.

**Lemma 20** Assume \( r \geq \lceil \log \max\{b, 6m\} \rceil \). On any \((3r + 1)\)-cube \( Q \), there exists a sequence of \( 3m + b + 1 \) distinct nodes, \((Q[0], Q[1], \ldots, Q[3m + b])\), such that (1) the Hamming distance between any two nodes in \( Q[0], \ldots, Q[3m - 1] \) is at least six, and (2) \((Q[3m], \ldots, Q[3m + b - 1])\) is a simple path of length \( b - 1 \).
Proof. It is well known that any hypercube has a Hamiltonian path (e.g., [67]). Let \( g \) be a function from \( \{0, \ldots, 2^r - 1\} \) to \( V(H_r) \) such that \( (g(0), g(1), \ldots, g(2^r - 1)) \) forms a Hamiltonian path in \( H_r \). It is not hard to see that \( k - l \) is even iff the Hamming distance between \( g(k) \) and \( g(l) \) is even. Therefore, if \( k \neq l \) and \( k - l \) is even then the Hamming distance between \( g(k) \) and \( g(l) \) is at least two. Let \( Q \) be a \((3r + 1)\)-cube. Define a sequence of \( 3m + b + 1 \) nodes, \((Q[0], Q[1], \ldots, Q[3m + b])\), as follows.

1. Let \( Q[k] = 0 g(2k) g(2k) g(2k) \) for \( 0 \leq k \leq 3m - 1 \), where \( 0 g(2k) g(2k) g(2k) \) is the concatenation of three copies of \( g(2k) \) preceded by a zero bit.

2. Let \( Q[3m] \) be adjacent to \( Q[3m - 1] \) along the first dimension, i.e.,

\[
Q[3m] = 1 g(6m - 2) g(6m - 2) g(6m - 2).
\]

3. Let \((Q[3m], Q[3m + 1], \ldots, Q[3m + b - 1])\) be any simple path of length \( b - 1 \) in the \( 3r \)-subcube \(* \ast \ldots \ast \).

4. Let \( Q[3m + b] \) be any node other than \( Q[0], \ldots, Q[3m + b - 1] \) in \( Q \).

Since \( r \geq \log 6m \) and \( r \geq \log b \), the sequence above is well defined and evidently has the desired properties. \( \square \)

Definition 27 Apply the above lemma and define for each subcube \( A_i \) and \( Q_j \) a sequence,

\[
A_i[0], A_i[1], \ldots, A_i[3m + b] \quad 0 \leq i \leq 3m - 1
\]

and

\[
Q_j[0], Q_j[1], \ldots, Q_j[3m + b] \quad 0 \leq j \leq m - 1
\]
such that, for each $k$, $A_i[k]$ is identical to $Q_j[k]$ in each of the dimensions $8r$ through $11r$. If $0 \leq p \leq q \leq 3m + b$, then $Q_j[p..q]$ denotes the subsequence $Q_j[p], \ldots, Q_j[q]$.

For any two nodes $x, y$ ($x \neq y$), let $\Psi(x, y)$ denote the path that moves from $x$ to $y$ in increasing order (i.e., lower dimensions first) along those dimensions in which the two nodes differ. For example, if $x = 0010101$ and $y = 0000000$, then $\Psi(x, y)$ is the path that goes from $x$ to $y$ along dimensions 2, 4, 6 in that order. Note that $\Psi(x, y)$ and $\Psi(y, x)$ are two different, edge-disjoint paths unless $\text{Dist}(x, y) = 1$; in the above example, $\Psi(y, x)$ moves from $y$ to $x$ along dimensions 2, 4, 6 in that order (or, equivalently, it moves from $x$ to $y$ in the order of 6, 4, 2). Extend the definition of $\Psi(x, y)$ so that, for any dimension $l$ in which the two nodes are identical, $\Psi_l(x, y)$ denotes the path that moves along $l$, then along those dimensions in which $x$ and $y$ differ (in increasing order), then along $l$ again. For instance, the path $\Psi_l(x, y)$ for the above $x$ and $y$ goes from $x$ to $y$ along dimensions 1, 2, 4, 6, 1 in that order. For $x = y$, define $\Psi(x, y)$ and $\Psi_l(x, y)$ as the zero-length path from $x$ to itself.

Between each $A_i$ and $Q_j$ ($0 \leq i \leq 3m - 1$, $0 \leq j \leq m - 1$) define a path

$$P_{ij} \overset{\text{def}}{=} \Psi(A_i[(i + j) \mod 3m], Q_j[(i + j) \mod 3m]).$$

(4.3)

Thus, for each $A_i$ there are $m$ paths (i.e., $P_{ij}$, $0 \leq j \leq m - 1$) each connecting it to a $Q_j$; these connections are made via $A_i[i], \ldots, A_i[(i + m - 1) \mod 3m]$ on $A_i$ and via $Q_0[i], \ldots, Q_{m-1}[(i + m - 1) \mod 3m]$ on $Q_j$'s (see Fig. 26). The two endpoints of each path $P_{ij}$ may differ only in the first $8r$ dimensions. Consequently, each $P_{ij}$ has length less than or equal to $8r$. Moreover, these paths are pairwise a Hamming distance of six or more apart, as proved below.
Lemma 21 If \((i, j)\) and \((k, l)\) are not equal as ordered pairs, then \(\text{Dist}(P_{ij}, P_{kl}) \geq 6\).

Proof. Assume \((i, j) \neq (k, l)\). If \((i + j) \mod 3m \neq (k + l) \mod 3m\), the lemma follows directly from Definition 27 and Lemma 20. If \((i + j) \mod 3m = (k + l) \mod 3m\), then the four values \(m + i, m + k, j, l\) are pairwise not equal. From Eq. 4.2 and Eq. 4.3 one readily sees that any node on \(P_{ij}\) differs from any node on \(P_{kl}\) in at least six out of the first \(8r\) dimensions. \(\square\)

Definition 28 Let \(G\) be a graph and \(L = (v_1, \ldots, v_k)\) a linear chain. \(L\) is said to be attached to a node \(u \in V(G)\), if a new edge \((u, v_1)\) is introduced. In that case, \(L\) may be simply said to be attached to \(G\), and \((u, v_1)\) is the corresponding glue edge.

We are now ready to construct a connected graph \(G\) from a given instance of 3-PARTITION, \(m, b,\) and \(A = \{a_0, \ldots, a_{3m-1}\}\). Let \(n = 11r + 11, r = \lceil \log \max\{b, 6m\} \rceil\). \(G\) is constructed as following. (See Fig. 26 for illustration.)

Constructing \(G\)

1. Start with an \(n\)-cube \(H\). Let \(R_j, Q_j, A_i, P_{ij}\) be subcubes/paths on \(H\), as defined in Eqs. (4.2) and (4.3).

2. For \(0 \leq j \leq m - 1\), remove from \(H\) all the nodes \(Q_j[0], Q_j[1], \ldots, Q_j[3m + b - 1]\) and all the edges incident upon them. (Note that each node has \(n\) edges incident upon it.)

3. For \(0 \leq i \leq 3m - 1\) and \(0 \leq j \leq m - 1\), remove from \(H\) all edges in \(A_i\) and all edges in \(R_j\) (only those edges with both endpoints in \(A_i\) or both in \(R_j\) are
Figure 26: The structure of $G$
removed; no nodes are removed in this step). (Each $A_i$ contains $\frac{1}{2}(3r + 1)2^{3r+1}$ edges, and each $R_j$ contains $\frac{1}{2}(3r + 7)2^{3r+7}$ edges; some edges in $R_j$ have already been removed in Step 2.)

4. For $0 \leq i \leq 3m - 1$ and $0 \leq j \leq m - 1$, remove from $H$ all edges, but not nodes, on path $P_{ij}$. (Each path contains at most $8r$ edges.)

5. Denote the resulting graph $G_{m,b}$, which is called a skeleton graph.

6. For each $A_i$, $0 \leq i \leq 3m - 1$, attach to node $A_i[i]$ an $a_i$-chain and attach to each node $A_i[(i + j) \mod 3m]$, $1 \leq j \leq m - 1$ a single-node chain. (Thus, whenever a node $A_i[k]$ is an endpoint of a path $P_{ij}$, there is a chain attached to it; the chain has $a_i$ nodes if $j = 0$, and has a single node if $j \neq 0$.) Let $L_{ij}$ denote the linear chain that is attached to node $A_i[(i + j) \mod 3m]$, and let $e_{ij}$ denote the glue edge through which the attachment is made.

7. For $0 \leq j \leq m - 1$, attach three single-node chains to node $Q_j[3m + b]$. Denote by $L^j$ and $e^j$ ($1 \leq l \leq 3$) the attached chains and the corresponding glue edges, respectively.

8. Let $G$ be the graph that results.

In summary, the total number of nodes removed from $H$ is $m(3m + b)$, the total number of nodes attached to $H$ is also $m(3m + b)$, and the total number of edges
removed from $H$ is less than

$$m \cdot (8r + 4) \cdot 2^{3r+1} + 3m \cdot \frac{1}{2} (3r + 1) \cdot 2^{3r+1} + m \cdot \frac{1}{2} (3r + 7) \cdot 2^{3r+7} + 3m^2 \cdot 8r < (38r + 78)2^{3r}$$

(4.4)

The last upper bound will be used later on to establish an important property of the above skeleton graph.

It is clear that the computing time needed to construct graph $G$ is bounded by a polynomial in $2^n$, which is polynomial in $m$ and $b$ (or pseudo-polynomial in input size of 3-PARTITION) since $n = \lceil \log \max\{6m, b\} \rceil$.

4.4.2 Constructing an Embedding from a 3-Partition

Suppose that the given set of integers $A = \{a_0, \ldots, a_{3m-1}\}$ can be partitioned into $m$ disjoint sets $S_0, \ldots, S_{m-1}$ such that, for $0 \leq j \leq m - 1$, $\sum_{a \in S_j} a = b$. In this subsection, it is shown that the graph $G$ as constructed in the preceding subsection can be embedded with unit congestion in an $n$-cube $H_n$.

Denote by $f$ the yet-to-be-specified embedding. We first embed the skeleton graph $G_{m,b}$ in $H_n$ through the identity mapping; i.e., every node $x$ in $G_{m,b}$ is mapped to the node in $H_n$ with the same label as $x$, and every edge $(x, y)$ in $G_{m,b}$ is mapped to edge $(f(x), f(y))$ in $H_n$. With $G_{m,b}$ so embedded, it is convenient to think of $G_{m,b}$ simply as a subgraph of $H_n$, and to think of $V(H_n) - V(G_{m,b})$ and $E(H_n) - E(G_{m,b})$ as consisting of those nodes and edges that have been removed from $H$ in the construction of $G$.

What remain to be embedded by $f$ are the linear chains attached to the skeleton graph and the corresponding glue edges: namely, $L_{ij}$, $e_{ij}$, $L^j$, and $e^j$, where $0 \leq i \leq 3m - 1$, $0 \leq j \leq m - 1$, and $1 \leq l \leq 3$. 
Each $Q_j$, $0 \leq j \leq m - 1$, now regarded as a subgraph of $H_n$, has exactly $3m + b$ nodes not in (the image of) $G_{m,b}$: $Q_j[0], Q_j[1], \ldots, Q_j[3m + b - 1]$. The last $b$ nodes form a simple path (see Lemma 20). The following edges in $H_n$ are also not in (the image of) $G_{m,b}$: all edges in $R_j$, all edges in $A_i$, and all edges in paths $P_{ij}$, $0 \leq i \leq 3m - 1$, $0 \leq j \leq m - 1$. These nodes and edges are all what are available for embedding the above mentioned linear chains and glue edges.

Each $A_i$, $0 \leq i \leq 3m - 1$ will have exactly one chain embedded in each $Q_j$, $0 \leq j \leq m - 1$. If $a_i \in S_j$ in the given 3-partition, then the $a_i$-chain attached to $A_i$ will be embedded in $Q_j$; otherwise, a single-node chain attached to $A_i$ will be embedded in $Q_j$. (Path $P_{ij}$, which connects $A_i$ to $Q_j$, makes it possible to "migrate" a chain from $A_i$ to $Q_j$ and embed it there.) All together, there will be $3m + 3$ chains embedded in $Q_j$: (i) the three chains corresponding to the three integers in $S_j$, (ii) $3m - 3$ single-node chains from $3m - 3$ subcubes $A_i$, and (iii) the three single-node chains attached to node $Q_j[3m + b]$. The chains corresponding to the three integers in $S_j$ (which together have $b$ nodes) will be embedded in the path $Q_j[3m \ldots 3m + b - 1]$. The $3m$ single-node chains will be embedded in $Q_k[0 \ldots 3m - 1]$.

For each $a_i$, $0 \leq i \leq 3m - 1$, let $k_i \in \{0, \ldots, m - 1\}$ be the index such that $a_i \in S_{k_i}$. Let $q_i \in \{1, 2, 3\}$ be such that $a_i$ is the $q_i$th element in $S_{k_i}$, assuming that the elements in $S_{k_i}$ are ordered by their indices. For instance, if $i = 9$ and $S_4 = \{a_5, a_7, a_9\}$, then $k_i = 4$ and $q_i = 3$. Let $\sigma_i$ be the sum of the integers $a_l$ in $S_{k_i}$ such that $l < i$. In the above example, if $(a_5, a_7, a_9) = (14, 20, 17)$, then $\sigma_i = 34$.

Now let us see how the linear chains attached to a subcube $A_i$ are embedded.
Recall that such chains are denoted as $L_{ij}$ and the corresponding glue edges as $e_{ij}$, $0 \leq j \leq m - 1$.

(a) Let $L_{i0}$, the $a_i$-chain representing the integer $a_i$, be embedded in $Q_{k_i}$, with $f(L_{i0}) = Q_{k_i}[x_i \ldots x_i + a_i - 1]$, where $x_i = 3m + a_i$. In order to do so, let the glue edge $e_{i0}$ be mapped to a path that goes first from node $A_i[i]$ to node $A_i[(i + k_i) \mod 3m]$. 

Figure 27: Embedding of linear chains and glue edges
then from $A_i[(i + k_i) \mod 3m]$ to $Q_{k_i}[(i + k_i) \mod 3m]$ using path $P_{i,k_i}$, and then from $Q_{k_i}[(i + k_i) \mod 3m]$ to $Q_{k_i}[x_i]$. (See Fig. 27, where the heavy line indicates the image of $e_{i0}$, and the thin dashed line the image of $L_{i0}$.)

(b) Embed $L_{i,k_i}$ in $Q_0$ (rather than $Q_{k_i}$, since $L_{i0}$ has been embedded in $Q_{k_i}$), with $f(L_{i,k_i}) = Q_0[i]$; and embed the glue edge $e_{i,k_i}$ in a path that goes from $A_i[(i + k_i) \mod 3m]$ to $A_i[i]$ and from there follows path $P_{i,0}$ to $Q_0[i]$. (See Fig. 27; the image of $e_{i,k_i}$ is indicated by the thick dashed line.)

(c) If $0 \leq j \leq m - 1$ but $j \notin \{0, k_i\}$, then $L_{ij}$, which is attached to $A_i[(i + j) \mod 3m]$, is simply sent along $P_{ij}$ and embedded at $Q_j[(i + j) \mod 3m]$. The thin dashed line in Fig. 27 shows the image of $f(e_{ij})$.

It now remains to embed, for each $Q_j$, the three nodes attached to $Q_j[3m + b]$. Let $a_{\rho(j,1)}, a_{\rho(j,2)}, a_{\rho(j,3)}$ be the three elements in $S_j$. There are three nodes in $Q_j[0 \ldots 3m + b - 1]$ which are still unoccupied: namely, $Q_j[(\rho(j,l) + j) \mod 3m], \ l \in \{1, 2, 3\}$. (These nodes are the endpoints of the $P$ paths in $Q_j$ through which the linear chains corresponding to the three integers in $S_j$ are migrated to $Q_j$.) The three nodes attached to $Q_j[3m + b]$ are therefore mapped to $Q_j[(\rho(j,l) + j) \mod 3m], \ l \in \{1, 2, 3\}$.

Denote by $P_1 \circ P_2$ the concatenation of paths $P_1$ and $P_2$, and let $k_i, q_i, \sigma_i$, and $\rho(j,l)$ be defined as above. The embedding described above is formally specified in the following.

**Embedding $G$ in $H_n$**

1. Embed the skeleton graph $G_{m,b}$ in $H_n$ with the identity mapping.

2. For $0 \leq i \leq 3m - 1$ and $0 \leq j \leq m - 1$, embed $e_{ij}$ and $L_{ij}$ as follows.
(a) If \( j = 0 \), then let

\[
e_{ij} \mapsto \Psi_A([i], A_i[(i + k_i) \mod 3m]) \circ P_{i,k_i} \circ \Psi_t(Q_{k_i}[(i + k_i) \mod 3m], Q_{k_i}[x_i])
\]

\[
L_{ij} \mapsto Q_{k_i}[x_i \ldots x_i + a_i - 1]
\]

where \( l = 11r + q_i \) and \( x_i = 3m + \sigma_i \). (Note that the \( \Psi_t \) path uses only edges in \( E(R_j) - E(Q_j) \); the only purpose of \( R_j \) is to provide such paths.)

(b) If \( 0 < j = k_i \), then

\[
e_{ij} \mapsto \Psi([i + k_i] \mod 3m, [i]) \circ P_{i,0}
\]

\[
L_{ij} \mapsto Q_{0}[i].
\]

(c) If \( 0 < j \neq k_i \), then

\[
e_{ij} \mapsto P_{ij}
\]

\[
L_{ij} \mapsto Q_{j}[(i + j) \mod 3m].
\]

3. For each \( S_j \), \( 0 \leq j \leq m - 1 \), let \( a_{\rho(j,1)}, a_{\rho(j,2)}, a_{\rho(j,3)} \) be the three elements in \( S_j \). Embed the three single-node chains attached to \( Q_j[3m + b] \) into nodes \( Q_j[(\rho(j,l) + j) \mod 3m], l = 1, 2, 3 \), respectively. Embed the corresponding glue edges to the following paths:

\[
\Psi_{11r+3+l}(Q_j[3m + b], Q_j[(\rho(j,l) + j) \mod 3m]) \quad l = 1, 2, 3.
\]

We now show that the mapping defined above is an embedding with congestion-1. It is not hard to see that the node mapping is one-to-one, and so the entire mapping defines an embedding. That the embedding has unit congestion is established below.
**Lemma 22** The embedding constructed above has congestion one.

**Proof.** The lemma follows from the following observations: (1) no edge in $G_{m,b}$ is used in the embedding of any chain or glue edge, (2) if $(i,j) \neq (k,l)$ then $P_{ij}$ and $P_{kl}$ are edge-disjoint, (3) each $P_{ij}$ is used only once, (4) for any $j$, the images of the three $a_i$-chains embedded in $Q_j[3m \ldots 3m + b - 1]$ have no overlaps, (5) for any $j$, the six paths $\Psi_l$ and $\Psi_{11+3+k}$ used in step 2(a) and step 3 are pairwise edge-disjoint and make no use of any single edge in $Q_j$, (6) for any $i$, the two $\Psi$ paths ($\Psi$'s without subscripts) used in (a) and (b) of step 2 are edge-disjoint, and (7) the $A_i$, $Q_j$, and $R_j$ subcubes are mutually disjoint, except that $Q_j$ is a subcube of $R_j$. □

### 4.4.3 Extracting a 3-Partition from an Embedding

It was shown in the preceding subsection that if the given set of integers $A$ has a 3-partition then the constructed graph $G$ can be embedded in $H_n$ with congestion one. We now show that if, conversely, $G$ can be embedded in $H_n$ with unit congestion, then $A$ has a 3-partition. Recall that the core of graph $G$ is a skeleton graph, $G_{m,b}$.

**Definition 29** For integers $k, l \geq 0$, let $(k, l)$-Exterior($G_{m,b}$) denote the subgraph of $G_{m,b}$ that is induced on the vertex set \{v $\in V(G_{m,b}) : \text{Dist}(v, \bigcup_{ij} P_{ij}) \geq k$ and $\text{Dist}(v, \bigcup_{ij} \{A_i, R_j\}) \geq l$\}.

Thus, a node belongs to $(k, l)$-Exterior($G_{m,b}$) iff it is at least a Hamming distance $k$ away from every $P_{ij}$ and at least $l$ away from every $A_i$ and every $R_j$. The following theorem is the basis of our arguments in this chapter; it indicates that under any congestion-1 embedding, most of the skeleton graph remains invariant. The proof of
the theorem is very involved and, for clarity, will be presented in a separate section (Section 4.5).

**Theorem 11** Let $G_{m,b}$ be labeled by $L$. For any congestion-1 embedding $f : G_{m,b} \to H_n$, there is a labeling $L'$ on $H_n$ such that, for every node $u$ and edge $(v,w)$ in $(0,2)\text{-Exterior}(G_{m,b})$, $L(u) = L'(f(u))$ and $f(v,w) = (f(v), f(w))$.

Now, suppose that $f : G \to H_n$ is a congestion-1 embedding, where $G$ is as constructed in Section 4.4.1. Since $G_{m,b}$ is a subgraph of an $n$-cube, it is convenient to think of $G_{m,b}$ as a subgraph of $H_n$. By Theorem 11, it loses no generality to assume that $f(u) = u$ and $f(x,y) = (x,y)$ for all nodes $u$ and all edges $(x,y)$ in $(0,2)\text{-Exterior}(G_{m,b})$. The situation is depicted in Fig. 28, where the shaded area indicates the image of $(0,2)\text{-Exterior}(G_{m,b})$ in $H_n$. Observe that the image of $(0,2)\text{-Exterior}(G_{m,b})$ separates $A_i$'s and $R_j$'s from one another. Also observe that there is a path $P_{ij}$ between $A_i$ and $R_j$ for every pair of $A_i$ and $R_j$ (the figure illustrates such paths for only one subcube $A_i$).

Let $A'_i$ be the set of all nodes in the $A_i$ area that are yet to be embedded by $f$, and let $A''_i$ be the set of "positions" still available in that area. Precisely, $A'_i = A_i \cup C_i$, where $A_i = \{x \in V(G_{m,b}) : \text{Dist}(x, A_i) \leq 1\}$ and $C_i$ is the set of all nodes on the linear chains attached to $A_i$; and $A''_i = \{x \in V(H_n) : \text{Dist}(x, A_i) \leq 1\}$. (As sets of vertices, $A_i$ and $A''_i$ are equal.) Similarly, let $R'_j$ and $R''_j$ be the set of nodes yet to be embedded and the set of positions still available, respectively, both in the $R_j$ area. Note that $|R''_j| = |R'_j| + (3m + b - 3)$. 
Figure 28:

For each $i$, $|A'_i| = |A''_i| + (a_i + m - 1)$. Thus, at least $a_i + m - 1$ nodes in $A'_i$ must be embedded in "remote" areas such as $R''_j$ or $A''_k$, $k \neq i$. Let $X_i$ be the set of all nodes in $A'_i$ that are embedded in remote areas; i.e., $X_i = \{x \in A'_i : f(x) \notin A''_i\}$. We show that $X_i = C_i$.

A boundary edge of $X_i$ is one that joins a node in $X_i$ to one outside $X_i$. A boundary edge must be embedded by $f$ along some path $P_{ij}$. Since $A_i$ has only $m$ such paths connected to it, $X_i$ can have at most $m$ boundary edges. Observing that each node in $A_i$ has $8r + 10$ neighbors in $\tilde{A}_i - A_i$ and each node in $\tilde{A}_i - A_i$ has $8r + 9$ neighbors in $(0,2)\text{-Exterior}(G_{m,b})$, one readily sees that $C_i$ is the only subset of $A'_i$ that contains no less than $a_i + m - 1$ elements and has at most $m$ boundary edges. Thus, $X_i = C_i$.

The boundary edges of $C_i$ are $e_{ik}$, $0 \leq k \leq m - 1$. (Recall that $e_{ik}$ joins a linear chain to $A_i$.) These edges are embedded by $f$ along paths $P_{ij}$, $0 \leq j \leq m - 1$, and the attached chains are thereby "exported" to remote areas $R''_j$, $0 \leq j \leq m - 1$, one
chain per $R''_j$. As a result each $R''_j$ receives exactly $3m$ "imported" chains — one from each $A'_i$, $0 \leq i \leq 3m - 1$. The nodes on these chains, as well as the nodes in $R'_j$, must be embedded in $R''_j$, for all $P_{ij}$'s have been used up by edges $e_{ij}$, $0 \leq i \leq 3m - 1$ and $0 \leq j \leq m - 1$. Since each $R''_j$ can accommodate only $b + 3m - 3$ imported nodes, the $3m$ chains that $R''_j$ has received must consist of exactly $3m - 3$ single-node chains and exactly three $a_i$-chains. (If $R''_j$ receives more than three $a_i$-chains, then, since $(b + 1)/4 < a_i$ for each $i$, $R''_j$ will have more than $b + 3m - 3$ imported nodes; and if $R''_j$ receives less than three $a_i$-chains, then some $R''_k$ will have to receive more than three $a_i$-chains.) The three $a_i$-chains imported to $R''_j$ have a total of $b$ nodes. Let $S_j$ contain exactly the three integers corresponding to the three $a_i$-chains embedded in $R''_j$. Then $\{S_0, \ldots, S_{j-1}\}$ is a 3-partition of $A$.

We have shown how to transform 3-PARTITION to CONGESTION-1 EMBEDDING in pseudo-polynomial time. That, together with Theorem 8, proves the following.

**Theorem 12** CONGESTION-1 EMBEDDING is NP-complete even if connected graphs only are concerned.

The embedding constructed in Section 4.4.2 has dilation at most $14r + 3$. So, we have the following.

**Corollary 7** The problem of determining if a connected graph can be embedded in a given $n$-cube with congestion one and dilation $2n$ is NP-complete.

One may have noticed that the source graph $G$ constructed in Section 4.4.1 has the same number of nodes as $H_n$. That is not essential for our NP-completeness
proof, though. If in Eq. 4.2 we define $Q_j$ not only for $0 \leq j \leq m - 1$ but also for $4m \leq j \leq 2^r - 1$ and remove from $G$ all subcubes $Q_j$, $4m \leq j \leq 2^r - 1$, then, still, the resulting graph can be embedded in $H_n$ with congestion one iff the given integer set $A$ has a 3-partition. The new source graph now has no more than $2^n - 2^{4r-1}$ nodes. So, the congestion-1 embedding problem is still NP-complete even if the source graph has fewer nodes than the hypercube.

### 4.5 Proof of Main Theorem

Theorems 9 and 11 are the basis of our NP-hardness proofs in Sections 4.3 and 4.4. They indicate that a skeleton graph $G_{m,b}$ is essentially invariant under any congestion-1 embedding. The two theorems can be proved along the same line. We present in this section a complete proof for Theorem 11, and show what modifications are necessary for the same line of reasoning to yield a proof of Theorem 9.

Throughout this section, unless otherwise designated, $n = 11r + 11$, $r = \lceil \log \max\{b, 6m\} \rceil$, $G_{m,b}$ denotes the skeleton graph constructed in Section 4.4.1, and $f$ denotes a congestion-1 embedding of $G_{m,b}$ into an $n$-cube $H_n$.

**Definition 30** Under an embedding $f$, a node in $G_{m,b}$ is said to be a dilation-1 node if every edge incident upon it has dilation one; it is a non-dilation-1 node if it is not dilation-1.

**Definition 31** Two nodes $x$ and $y$ are neighbors if $\text{Dist}(x,y) = 1$; they are 2-neighbors if $\text{Dist}(x,y) = 2$. 
Theorem 11 states that in a congestion-1 embedding of $G_{m,b}$ into $H_n$, $(0,2)$-Exterior$(G_{m,b})$ is embedded with dilation one. If that is true, then, in particular, all nodes in $(2,3)$-Exterior$(G_{m,b})$ will be dilation-1 nodes. Conversely, if $(2,3)$-Exterior$(G_{m,b})$ is known to contain only dilation-1 nodes, then Theorem 11 will not be hard to prove. Thus first concentrate on establishing the dilation-1 property of the nodes in $(2,3)$-Exterior$(G_{m,b})$. We shall assume, for contradiction, that $(2,3)$-Exterior$(G_{m,b})$ has non-dilation-1 nodes and then show a “chain reaction” phenomenon of such nodes: If, under an embedding $f$, there is a non-dilation-1 node in $(2,3)$-Exterior$(G_{m,b})$, then the node must have at least $\lceil n/2 - 10 \rceil$ non-dilation-1 neighbors in $(2,3)$-Exterior$(G_{m,b})$; each of them in turn must have at least $\lceil n/2 - 10 \rceil$ non-dilation-1 neighbors in $(2,3)$-Exterior$(G_{m,b})$, and so forth. As a result of the chain reaction, it will be shown that there must be at least $\lfloor 2^{n/2 - 10} \rfloor$ non-dilation-1 nodes in $(2,3)$-Exterior$(G_{m,b})$ and, therefore, that at least $\lfloor 2^{n/2 - 11} \rfloor$ edges in $(2,3)$-Exterior$(G_{m,b})$ must be mapped to multi-edge paths. On the other hand, a result in Section 4.4.1, Eq. 4.4, indicates that $|E(H_n)| - |E(G_{m,b})| < (38r + 78)2^{4r}$, meaning that $G_{m,b}$ can have no more than $(38r + 78)2^{4r}$ edges being mapped to multi-edge paths. Therefore, $\lfloor 2^{n/2 - 11} \rfloor \leq (38r + 78)2^{4r}$. But the last inequality is false; therefore, $(2,3)$-Exterior$(G_{m,b})$ cannot have any non-dilation-1 node. Once $(2,3)$-Exterior$(G_{m,b})$ is known to contain only dilation-1 nodes, Theorem 11 can be proved easily.

As one might have seen from the above sketch, our arguments are based on counting nodes and edges of certain types. In particular, we need to know how many
neighbors a node may have in \((2,3)\)-Exterior\((G_{m,h})\), and how many edges a node’s 2-neighbor may connect to. The following two lemmas address such questions.

**Lemma 23** If \(u\) is a node in \((2,3)\)-Exterior\((G_{m,h})\), then \(u\) has at most five neighbors not in \((2,3)\)-Exterior\((G_{m,h})\).

**Proof.** Let \(u\) be any node in \((2,3)\)-Exterior\((G_{m,h})\). We may assume, without loss of generality, that \(\text{Dist}(u, \bigcup P_{ij}) = 2\) and \(\text{Dist}(u, \bigcup \{A_i, R_j\}) = 3\), for that is evidently the case where \(u\) has the largest number of neighbors not in \((2,3)\)-Exterior\((G_{m,h})\).

By Lemma 21, there is a unique path \(P_{ij}\) such that \(\text{Dist}(u, P_{ij}) = 2\). By the fact that every two subcubes defined by Eq. 4.2 are at least a distance of eight apart, there is a unique subcube \(R = A_i\) or \(R = R_j\) such that \(\text{Dist}(u, R) = 3\). Assume \(R = R_j\) (the other case is similar).

Since \(\text{Dist}(u, R_j) = 3\), node \(u\) has exactly \(\binom{3}{1} = 3\) neighbors within distance two of \(R_j\), which are evidently not in \((2,3)\)-Exterior\((G_{m,h})\). In order to prove the lemma, it suffices to show that \(u\) has at most two other neighbors being not in \((2,3)\)-Exterior\((G_{m,h})\); such neighbors, called type-A neighbors, are within distance one of \(P_{ij}\) and have distance at least three from \(R_j\).

Suppose that, as path \(P_{ij}\) goes from node \(A_i[(i+j) \mod 3m]\) to node \(Q_j[(i+j) \mod 3m]\), it moves exactly along dimensions \(d_1, d_2, \ldots, d_k\), in that order. Let \(d_{k+1}, \ldots, d_n\) be the remaining dimensions (along none of which \(P_{ij}\) ever moves). Thus, \((d_1, \ldots, d_n)\) is a permutation of \((0, 1 \ldots, n - 1)\). Let (the label of) node \(Q_j[(i+j) \mod 3m]\) be \(q_1 \ldots q_n\) in binary. Interpreting \(q_i\) as “high” and \(\bar{q}_i\), the complement of \(q_i\), as “low”, one may represent a node by an \(n\)-bit square wave. For instance, the square wave
(a) Node $q_{d_1} \ldots q_{d_{l-1}} \bar{q}_{d_l} \ldots \bar{q}_{d_k} q_{d_{k+1}} \ldots q_{d_n}$.

(b) A node having two neighbors within distance one of $P_{ij}$.

(c) A node having two sets of spikes and two type-A neighbors.

(d) A node having two type-A neighbors.

(e) A node having two type-A neighbors.

(f) A node having two type-A neighbors.

(g) A node with three (five) 2-neighbors on $P_{ij}$ (outside $(1,2)$-Exterior($G_{m,b}$))

Figure 29: Square waves.
in Fig. 29(a) represents the node with bits $q_{d_1}, \ldots, q_{d_{l-1}}, \bar{q}_{d_l}, \ldots, \bar{q}_{d_k}, q_{d_{k+1}}, \ldots, q_{d_n}$ in dimensions $d_1, \ldots, d_n$, respectively. Every node on $P_{ij}$ has a square wave of the form of Fig. 29(a), with $1 \leq l \leq k + 1$ (the wave has no low bit when $l = k + 1$).

Observe how the wave changes as $l$ raises from 1 to $k + 1$ (i.e., as the path goes from $A_i[(i + j) \mod 3m]$ to $Q_j[(i + j) \mod 3m]$). If $x$ is a node with $Dist(x, P_{ij}) = t$, then there is a set of $t$ bits in $x$’s square wave such that reversing all the $t$ bits will result in the wave of a node in $P_{ij}$. Such a set is called a set of spikes. In Fig. 29(c), for instance, the two bits in dimensions $d_{k-3}$ and $b_2$ are a set of spikes, and so are the two in dimensions $d_{k-2}$ and $b_2$. With the help of square waves, we are ready to show that $u$ has at most two type-$A$ neighbors.

Recall that the string representation of $R_j$ has *’s in bits $8r$ through $11r + 6$, and that $\{d_1, \ldots, d_k\} \cap \{8r, 8r + 1, \ldots, 11r + 6\} = \emptyset$. So, without loss of generality, we may assume that $\{d_{n-3r-6}, \ldots, d_n\} = \{8r, 8r + 1, \ldots, 11r + 6\}$.

Recall that $Dist(u, P_{ij}) = 2$ and $Dist(u, R_j) = 3$. Suppose now that $x$ is a type-$A$ neighbor of $u$ whose square wave can be obtained from $u$’s wave by reversing bit $d_i$. If $d_i \in \{d_{n-3r-6}, \ldots, d_n\}$, then bit $d_i$ must be a low one in $u$’s square wave (for reversing a high bit in the range $d_{n-3r-6}$ through $d_n$ will result in a node with $Dist(\cdot, P_{ij}) = 3$). On the other hand, if $d_i \in \{d_1, \ldots, d_{n-3r-5}\}$, then bit $d_i$ must be a high bit in $u$’s square wave (for reversing a low bit in that range will yield a node with $Dist(\cdot, R_j) = 2$); besides, bit $d_i$ must be such that reversing it will lead to a node with $Dist(\cdot, P_{ij}) = 1$. Thus, in order to determine how many type-$A$ neighbors $u$ may have, it suffices to determine the number of such bits $d_i$ in $u$’s square wave.
Now, since $\text{Dist}(u, P_{ij}) = 2$, the square wave of $u$ has a set of two spikes. Let $\{b_1, b_2\}$ be such a set. Let $\Delta = \{b_1, b_2\} \cap \{d_1, \ldots, d_k\}$. Then $|\Delta| = 0, 1, \text{or} 2$.

**Case 1:** $|\Delta| = 0$. In this case, the square wave of $u$ is as shown in Fig. 29(b). Clearly, $u$ has only two neighbors within distance one of $P_{ij}$ (reversing bit $b_1$ or $b_2$ each yields such a neighbor). Therefore it has at most two type-\(A\) neighbors.

**Case 2:** $|\Delta| = 1$. Without loss of generality, assume $b_1 \in \{d_1, \ldots, d_k\}$. By considering all possible positions for $b_1$ and $b_2$, one may readily see that $u$ has two type-\(A\) neighbors if its square wave is as shown in Fig. 29(c), where the position of the high bit between $d_{k-3}$ and $d_k$ could be anywhere between $d_{k-2}$ and $d_k$, inclusively, and $b_2 \in \{d_{n-3r-6}, \ldots, d_n\}$. The two type-\(A\) neighbors can each be obtained from $u$ by reversing either bit $b_2$ or the high bit between $d_{k-2}$ and $d_k$. It is not hard to see that in all other cases where $|\Delta| = 1$, $u$ has at most one type-\(A\) neighbor.

**Case 3:** $|\Delta| = 2$. In this case, both $b_1$ and $b_2$ are in $\{d_1, \ldots, d_k\}$. Again, by considering all possible positions for $b_1$ and $b_2$, one may readily check that $u$ has two type-\(A\) neighbors if its square wave is as shown in Fig. 29(d), (e) or (f) and that $u$ has at most one type-\(A\) neighbor in all other cases. □

**Lemma 24** *If $u$ is a node in $(2,3)$-Exterior($G_{m,b}$), then $u$ has at most five 2-neighbors of degree less than $n$, of which at most three are of degree less than $n - 1$.***

**Proof.** Let $u$ be any node in $(2,3)$-Exterior($G_{m,b}$). A 2-neighbor of $u$ has degree less than $n - 1$ only if it is on some path $P_{ij}$, and it has degree less than $n$ only if it is not in $(1,2)$-Exterior($G_{m,b}$). So we only need to show that $u$ has at most three
2-neighbors on $\bigcup_{ij} P_{ij}$, and at most five 2-neighbors not in $(1,2)$-Exterior$(G_{m,b})$. We show the former first.

Assume $\text{Dist}(u, \bigcup_{ij} P_{ij}) = 2$, or $u$ has no 2-neighbor on $\bigcup_{ij} P_{ij}$ and we are done. Let $P_{ij}$ be the unique path such that $\text{Dist}(u, P_{ij}) = 2$. Let $k$ and $d_1, \ldots, d_k, \ldots, d_n$ be as defined in the proof of Lemma 23. Let $y$ be any node on $P_{ij}$ such that $\text{Dist}(u, y) = 2$, and let $l$ be such that $y$ differs from $Q_j[(i + j) \mod 3m]$ exactly in dimensions $d_l, \ldots, d_k$. The wave of node $y$ is as shown in Fig. 29(a). Consider two possible cases.

**Case 1:** Suppose that nodes $u$ and $y$ differ in dimensions $d_{l+1}$ and $d_{l+3}$, or in $d_{l+1}$ and $d_{l-2}$, or in $d_{l-2}$ and $d_{l-4}$. Node $u$'s square wave will be as shown in Fig. 29(g), where $d_l = c_1, c_3$ or $c_5$ depending on where $u$ and $y$ differ. Reversing the two bits of $u$ in dimensions $c_1 \& c_3$, or $c_1 \& c_4$, or $c_2 \& c_4$ will result in a node on $P_{ij}$, and any other pair of dimensions will have no such effect. Thus, in this case $u$ has exactly three 2-neighbors on $P_{ij}$.

**Case 2:** Suppose $u$ and $y$ differ in any other two dimensions. It is not hard to check that $u$ has less than three 2-neighbors on $P_{ij}$.

So, in all cases, $u$ has at most three 2-neighbors on $P_{ij}$. Meanwhile, $u$ has at most $\binom{3}{2} = 3$ 2-neighbors with Hamming distance one from $\bigcup_{ij} \{A_i, R_j\}$. These two results together indicate that $u$ has at most six 2-neighbors not in $(1,2)$-Exterior$(G_{m,b})$. We shall show that in fact it can have at most five such 2-neighbors. Suppose that $u$ has three 2-neighbors on a path $P_{ij}$ and three 2-neighbors within Hamming distance one of $\bigcup_{ij} \{A_i, R_j\}$. Referring to Fig. 29(f), it must be the case that either $c_1 = d_2$ or $c_5 = d_k$. That is, either $u$ differs from node $A_i[(i + j) \mod 3m]$ in dimensions
$d_1, d_3, d_5$ or $u$ differs from $Q_j[(i + j) \mod 3m]$ in dimensions $d_k, d_{k-2}, d_{k-4}$. In either case, $u$ has a 2-neighbor that is both on $P_{ij}$ and within Hamming distance one of $\bigcup_{ij} \{A_i, R_j\}$. Thus, even if $u$ has three 2-neighbors on a path $P_{ij}$ and three 2-neighbors within Hamming distance one of $\bigcup_{ij} \{A_i, R_j\}$, it has only five 2-neighbors being not in $(1,2)$-Exterior($G_{m,b}$). This completes the proof.  

We need another easy lemma in order to show the chain reaction of non-dilation-1 nodes.

**Lemma 25** Let $x$ be any node in $G_{m,b}$. If $x$ has degree $n - 1$ or more, then for no edge $e$ in $G_{m,b}$ can $f(x)$ be an intermediate node of $f(e)$.

**Proof.** For contradiction, assume that $x$ has degree at least $n - 1$ and $f(x)$ is an intermediate node of $f(e)$. Since $f(x)$ is an intermediate node, two out of the $n$ edges incident upon $f(x)$ are used up for $f(e)$. Since $x$ has degree $n - 1$ or more, at least $n - 1$ edges incident upon $f(x)$ are needed in order to embed the edges incident upon $x$, thereby causing a congestion of at least two. 

We are now ready to show the chain reaction of non-dilation-1 nodes as mentioned in the beginning of this section. Assuming that a non-dilation-1 node exists in $(2,3)$-Exterior($G_{m,b}$), the following lemma shows that such a node must have at least $\lceil n/2 - 5 \rceil$ non-dilation-1 neighbors, of which at least $\lceil n/2 - 10 \rceil$ are in $(2,3)$-Exterior($G_{m,b}$) (by Lemma 23).

**Lemma 26** Under a congestion-1 embedding $f$, if a node $u$ in $(2,3)$-Exterior($G_{m,b}$) is a non-dilation-1 node, then it has at least $\lceil n/2 - 5 \rceil$ non-dilation-1 neighbors.
**Proof.** Let $u$ be any non-dilation-1 node in $(2, 3)$-Exterior$(G_{m,b})$. Suppose that node $u$ has exactly $\lambda$ non-dilation-1 and $n - \lambda$ dilation-1 neighbors. We need to show $\lambda \geq n/2 - 5$.

For every node $w$ in $G_{m,b}$, write $f(w)$ as $w'$; in particular, $u' = f(u)$. Since $u$ is a non-dilation-1 node, $u$ has a neighbor $v$ such that edge $(u,v)$ is mapped to a multi-edge path. Assume $f(u,v) = (u',v_1,\ldots,v_j,v')$ where $j \geq 1$ (see Fig. 30). Let $u$ and $v$ be fixed in the rest of the proof. The basic idea behind the proof is to show the existence of at least $2n - 2\lambda - 10$ nodes in $H_n$ which are adjacent to node $v_1$. Since each node has only $n$ neighbors in $H_n$, it will follow that $2n - 2\lambda - 10 < n$ or, equivalently, $n/2 - 5 < \lambda$.

We first identify $n - \lambda$ nodes adjacent to $v_1$. Let $D$ be the set of all dilation-1 neighbors of $u$. Clearly, $|D| = n - \lambda$ and $v \notin D$. Without loss of generality, assume $|D| > 1$, or we are already done. Consider any node $x$ in $D$. Since $x$ is a dilation-1 node, the image of edge $(x,u)$ must be edge $(x',u')$. By Lemma 25, $x' \neq v_1$. Let $p'_x$ be the unique node in $H_n$ such that $(p'_x,v_1,u',x',p'_x)$ forms a cycle. Since $x$ is dilation-1 and has degree $n$, $p'_x$ must be the image of some node adjacent to $x$. Let $p_x = f^{-1}(p'_x)$. (See Fig. 30 for illustration.) With $u$ and $v$ fixed, $p_x$ is uniquely determined by $x$ and therefore the following defines an injective function from $D$ to $V(G_{m,b})$: \[
\varphi : x \rightarrow p_x \quad x \in D
\]

Note that $f(\varphi(D))$ consists of $n - \lambda$ nodes each adjacent to $v_1$.

Next, we find another set of at least $n - \lambda - 8$ of $v_1$'s neighbors, not necessarily disjoint from $f(\varphi(D))$. Of the $n - \lambda$ nodes in $\varphi(D)$, at least $n - \lambda - 5$ are of degree
Figure 30: Part of embedding $f : G_{m,b} \rightarrow H_n$

$n$ (by Lemma 24). Let $\varphi_n(D) = \{ p_x \in \varphi(D) : p_x = n \}$. $|\varphi_n(D)| \geq n - \lambda - 5$. Let $p_x$ be any node in $\varphi_n(D)$. Since $p_x$ has degree $n$, there is an edge $(p_x, q_x)$ incident upon $p_x$ such that $f(p_x, q_x)$ includes edge $(p'_x, v_1)$ ($q_x$ is uniquely determined by $p_x$). There are two possibilities: (1) edge $(p_x, q_x)$ is mapped to edge $(p'_x, v_1)$ with $f(q_x) = v_1$, or 
(2) $(p_x, q_x)$ is mapped to a path containing $(p'_x, v_1, r_x)$ as a subpath, where $r_x$ is some node adjacent to $v_1$ (see Fig. 30). The mapping $g : p_x \rightarrow r_x$

which is defined for all $p_x$ in $\varphi_n(D)$ such that $f(p_x, q_x) \neq (p'_x, v_1)$ is one-to-one. Let $\varphi_{n,r}(D) = \{ p_x \in \varphi_n(D) : f(p_x, q_x) \neq (p'_x, v_1) \}$. All nodes in $g(\varphi_{n,r}(D))$ are adjacent to $v_1$. Now we show $|g(\varphi_{n,r}(D))| \geq n - \lambda - 8$ by showing $|\varphi_n(D) - \varphi_{n,r}(D)| \leq 3$. Suppose that $p_a \in \varphi_n(D) - \varphi_{n,r}(D)$. Then $f(q_a) = v_1$. All nodes $p_x$ in $\varphi_n(D) - \varphi_{n,r}(D)$ satisfy $q_x = q_a$, for only one node can be mapped to $v_1$ by $f$. Since $q_a$ differs from $u$ by
three bits, there are at most three neighbors \(x\) of \(u\) such that \(q_x = q_a\). Therefore, 
\[|\varphi_n(D) - \varphi_{n,r}(D)| \leq 3 \text{ and } |g(\varphi_{n,r}(D))| \geq n - \lambda - 8.\]
We claim that \(g(\varphi_{n,r}(D))\) and \(f(\varphi(D))\) have at most three nodes in common.
To see that, let \(p_x \in \varphi_{n,r}(D)\) and \(p_y \in \varphi(D)\) be such that \(r_x = g(p_x) = f(p_y) = p'_y\). Observe that \(q_x \neq p_y\), as otherwise \((u, x, p_x, q_x, y, u)\) would form a cycle of odd length, which is impossible in a hypercube. Thus, \(q'_x \neq p'_y\) and hence \(p'_y = r_x\) is an intermediate node of \(f(p_x, q_x)\). By Lemma 25, \(p_y\) must have degree less than \(n - 1\); and by Lemma 24, at most three nodes in \(\varphi(D)\) are of degree less than \(n - 1\). This proves the above claim.

Now, \(f(\varphi(D))\) and \(g(\varphi_{n,r}(D))\) together contain at least \((n - \lambda) + (n - \lambda - 8) - 3\) distinct nodes adjacent to \(v_1\), and \(u'\) provides another such node \((u' \notin f(\varphi(D)) \cup g(\varphi_{n,r}(D)))\). Since \(v_1\) has only \(n\) neighbors, it must be that
\[
((n - \lambda) + (n - \lambda - 8) - 3) + 1 \leq n
\]
or, equivalently, \(\lambda \geq n/2 - 5.\)

The following lemma is interesting in its own right, and it will be needed in the proof of Lemma 28.

**Lemma 27** Let \(G'\) be any subgraph of a hypercube. If every node in \(G'\) has degree \(k\) or more, then \(G'\) has at least \(2^k\) nodes.

**Proof.** By induction on \(k\). If \(k = 1\), \(G'\) evidently has at least 2 nodes. Assume that any subgraph of a hypercube has at least \(2^k\) nodes if every node in it has degree no less than \(k\). Now assume \(G'\) is a subgraph of a hypercube such that every node in
$G'$ has degree $k + 1$ or more. Let $j$ be any bit position in which at least one pair of nodes in $G'$ are not equal. Let $G'_i$ ($i = 0, 1$) be the subgraph of $G'$ induced by $\{v \in V(G') : \text{the } j\text{th bit of } u \text{ is } i\}$. Each node in $G'_i$ has degree at least $k$, and by induction hypothesis $|V(G'_i)| \geq 2^k$. So, $|V(G')| \geq |V(G'_1)| + |V(G'_2)| \geq 2^{k+1}$. \hfill \Box

Lemma 26 above describes the chain reaction of non-dilation-1 nodes, if such nodes exist. In the following we show that in fact non-dilation-1 nodes cannot exist in $(2,3)$-Exterior$(G_{m,b})$, for otherwise their chain reaction would produce so many non-dilation-1 edges that congestion-1 embedding would become impossible.

**Lemma 28** Under a congestion-1 embedding $f$, all nodes in $(2,3)$-Exterior$(G_{m,b})$ are dilation-1 nodes.

**Proof.** Let $S$ be the set of all non-dilation-1 nodes in $(2,3)$-Exterior$(G_{m,b})$. We will show $S = \emptyset$. Assume, for contradiction, that $S \neq \emptyset$ and let $G'$ denote the subgraph of $G_{m,b}$ induced by $S$. Let $u$ be any node in $G'$. By Lemma 26, $u$ has at least $\lceil n/2 - 5 \rceil$ non-dilation-1 neighbors in $G_{m,b}$, of which, by Lemma 23, at most five are not in $(2,3)$-Exterior$(G_{m,b})$. Therefore, $u$ has at least $\lceil n/2 - 10 \rceil$ neighbors in $G'$. Since $u$ is arbitrary, that means every node in $G'$ has degree no less than $\lceil n/2 - 10 \rceil$ and, by Lemma 27, $G'$ has at least $\lceil 2^{n/2 - 10} \rceil$ nodes. In other words, $G_{m,b}$ has at least $\lceil 2^{n/2 - 10} \rceil$ non-dilation-1 nodes. Each non-dilation-1 node is associated with at least one edge whose $f$-image is a multi-edge path. (Such an edge may be associated with two nodes.) So, at least $\lceil 2^{n/2 - 11} \rceil$ edges in $G_{m,b}$ have multi-edge images under $f$. Since $f$ has congestion one, $f(G_{m,b})$ contains at least $|E(G_{m,b})| + \lceil 2^{n/2 - 11} \rceil$ edges. However, $H_n$ has at most $|E(G_{m,b})| + (38r + 78)2^{4r}$ edges, for $G_{m,b}$ was constructed by
removing from an n-cube a number of nodes and less than \((38r + 78)2^{4r}\) edges; and with \(r \geq \log b \geq 10\) and \(n = 11r + 11\), it is not hard to check that \((38r + 78)2^{4r} < 2^{n/2 - 11}\). That means, \(H_n\) will not have sufficient edges for \(f(G_{m,b})\) if \(S \neq \emptyset\). So, \(S\) must be an empty set and every node in \((2,3)\)-Exterior\((G_{m,b})\) is a dilation-1 node. \(\square\)

We are now ready to prove Theorem 11, which for convenience is restated below.

**Theorem 11** Let \(G_{m,b}\) be labeled by \(L\). For any congestion-1 embedding \(f : G_{m,b} \to H_n\), there is a labeling \(L'\) on \(H_n\) such that, for every node \(u\) and edge \((v, w)\) in \((0,2)\)-Exterior\((G_{m,b})\), \(L(u) = L'(f(u))\) and \(f(v, w) = (f(v), f(w))\).

**Proof.** Let \(f\) be a congestion-1 embedding of \(G_{m,b}\) into \(H_n\), where \(G_{m,b}\) is labeled by \(L\). For convenience, use \(z'\) in place of \(f(z)\) for all nodes \(z\) in \(G_{m,b}\).

It follows from Lemma 28 that \((2,3)\)-Exterior\((G_{m,b})\) is embedded by \(f\) with dilation one. Thus, there is a labeling \(L'\) of \(H_n\) such that \(L(u) = L'(u')\) for all nodes \(u\) in \((2,3)\)-Exterior\((G_{m,b})\). We will extend this result to \((1,2)\)-Exterior\((G_{m,b})\).

Let \(x\) be any node in \((1,2)\)-Exterior\((G_{m,b})\), but not in \((2,3)\)-Exterior\((G_{m,b})\). There exist three nodes, say, \(u, v, w\) in \((2,3)\)-Exterior\((G_{m,b})\) such that \((x, u, v, w, x)\) form a cycle of length four. (By Eq. 4.2, at least two of the rightmost four bits of \(L(x)\) are 0's. Changing any of such bits to 1 will yield a node in \((2,3)\)-Exterior\((G_{m,b})\).) Since \(u, v\) and \(w\) are dilation-1 nodes, \((x', u', v', w', x')\), the image of \((x, u, v, w, x)\), must be a cycle of length four. Since \(L(z) = L'(z')\) for \(z \in \{u, v, w\}\), it follows that \(L(x) = L'(x')\). Thus, we have shown \(L(x) = L'(x')\) for all nodes \(x\) in \((1,2)\)-Exterior\((G_{m,b})\).

Now we show \(f(x, y) = (x', y')\) for all edges \((x, y)\) in \((1,2)\)-Exterior\((G_{m,b})\).

Let \((x, y)\) be any edge in \((1,2)\)-Exterior\((G_{m,b})\). If it has an endpoint in
(2,3)-Exterior($G_{m,b}$), then by Lemma 28, $f(x, y) = (x', y')$. If neither endpoint is in (2,3)-Exterior($G_{m,b}$), then since $x$ is of degree $n$, there must be an edge $e_x$ incident upon $x$ such that $f(e_x)$ includes edge $(x', y')$; similarly, there must be an edge $e_y$ incident upon $y$ such that $f(e_y)$ includes $(x', y')$. Since $f$ has congestion one, it must be that $e_x = e_y = (x, y)$ and $f(x, y) = (x', y')$. So, $f(x, y) = (x', y')$ for all edges $(x, y)$ in (1,2)-Exterior($G_{m,b}$).

Now let $x$ be any node in (0,2)-Exterior($G_{m,b}$), but not in (1,2)-Exterior($G_{m,b}$). We show that $L(x) = L'(x')$. Note that $x$ is on some path $P_{ij}$. Let $x_1$ and $x_2$ be the two of $x$'s neighbors on $P_{ij}$. If $\text{Dist}(x, A_i \cup R_j) = 2$, then $x$ has exactly two neighbors within distance one of $A_i$ or $R_j$ (one of them is on path $P_{ij}$). In that case, let $x_3$ be the one not on $P_{ij}$; otherwise, let $x_3 = x_2$. Thus, $x_1$, $x_2$ and $x_3$ are the only neighbors of $x$ not to be in (1,2)-Exterior($G_{m,b}$). (See Fig. 31(a), where the dotted region represents (1,2)-Exterior($G_{m,b}$) and the dashed line represents path $P_{ij}$. Recall that $G_{m,b}$ contains the nodes but not the edges of $P_{ij}$.) Let $x''$, $x''_1$, $x''_2$, $x''_3$ be the nodes in $H_n$ such that $L'(x'') = L(x)$ and $L'(x''_k) = L(x_k)$, $1 \leq k \leq 3$. (See Fig. 31(b).)

Assume $f(x) \neq x''$ (otherwise, $L(x) = L'(x')$ and we are done). Let $N_x$ be the set of all $x$'s neighbors in (1,3)-Exterior($G_{m,b}$) which are not adjacent to any node on $P_{ij}$ but $x$. One may check that $|N_x| > 3$ (e.g., changing any of the rightmost four bits of $x$ yields a node in $N_x$). Let $z$ be any node in $N_x$. For every node $y \neq x$ such that $(z, y)$ is an edge, since both $z$ and $y$ are in (1,2)-Exterior($G_{m,b}$), $f(z, y) = (z', y')$. Therefore, edge $(z, x)$ must be embedded by $f$ along edge $(z', x'')$ and then along one of $(x'', x''_k)$, $1 \leq k \leq 3$ (every neighbor of $x''$, except for $x''_1$, $x''_2$ and $x''_3$, is the image
of some node in \((1,2)\)-Exterior\((G_{m,b})\) and therefore cannot be an intermediate node of \(f(z,x))\). Since there are \(|N_x| > 3\) paths contending for \((x'',x_k')\), \(1 \leq k \leq 3\), \(f\) would have congestion more than one if \(f(x) \neq x''\). So it must be \(f(x) = x''\) and \(L'(x') = L(x)\).

We have shown \(L(x) = L'(x')\) for for all nodes \(x\) in \((0,2)\)-Exterior\((G_{m,b})\). From that and the fact \(f(x,y) = (x',y')\) for all edges \((x,y)\) in \((1,2)\)-Exterior\((G_{m,b})\), it is not hard to see that \(f(x,y) = (x',y')\) for all edges \((x,y)\) in \((0,2)\)-Exterior\((G_{m,b})\). This completes the proof. \(\Box\)

As mentioned in the beginning of this section, Theorem 9 can be proved similarly. Indeed, Lemmas 23–28 will remain true if \((2,3)\)-Exterior\((G_{m,b})\) is substituted with \(3\)-Exterior\((G_{m,b})\); and from Lemma 28, Theorem 9 will follow immediately.
4.6 Concluding Remarks

Deciding whether a source graph can be embedded in a given hypercube with unit congestion is an important problem that arises in assigning the processes of a distributed algorithm to the processors of a hypercube which uses circuit-switching for internode communication. We have shown the problem NP-complete even if the source graph is connected. Thus, it is quite unlikely to have a polynomial-time algorithm for the problem. On the other hand, we have shown that any graph $G$ can be embedded with congestion one in a hypercube of dimension $n \geq \max\{6\log |V(G)|, G\}$.

Many important graphs can be embedded with congestion one in a hypercube of dimension $\lceil \log |V(G)| \rceil$. They include rings, full binary trees, full quad-trees and most meshes [83]. A pyramid can be embedded in $H_{\lceil \log |V| \rceil}$ with congestion two [49], and it is still an open problem whether congestion-1 embedding is possible.
CHAPTER V

Compacting Free Buddy Subcubes in a Hypercube

5.1 Overview

The hypercube has emerged as one of the most popular architectures for parallel processing. A hypercube that supports multiprogramming can be partitioned into subcubes, possibly of different sizes, to execute independent jobs, with each job running on a dedicated subcube. When a job arrives at such a system requesting a subcube of dimension $d$, a free $d$-subcube, if exists, is located and assigned to it. The subcube is released to the system on completion of the job. Allocating free subcubes to incoming jobs is a difficult job, and several methods have been proposed: e.g., buddy system [15, 70], Gray code [15], Free List [42] and MSS [21]. Some distributed subcube management schemes are in [40, 41].

It has been pointed out [15] that as jobs arrive grabbing subcubes and leave releasing them to the system, the hypercube tends to become fragmented. The fragmentation problem occurs when the hypercube has a sufficient number of free processors, but a job is rejected because the underlying subcube-allocation scheme is unable to locate a large enough free subcube to accommodate the job. This problem cannot be solved by simply adopting a perfect subcube-allocation scheme (one that can recognize all
free subcubes, e.g., Free list and MSS). Chen and Shin [16] studied the hypercube fragmentation problem and proposed to migrate (relocate) the jobs currently in the system so as to free up a large free subcube.

A subcube compaction scheme can be partial or full [37]. A partial scheme compacts only enough subcube to free up a subcube of requested size. A full scheme compact all free processors into as few (and thus as large) subcubes as possible. A partial scheme is shown by simulation in [37] to be more efficient.

Job migration posts different problems for different communication models and different allocation schemes. Chen and Shin [16] assumed a store-and-forward communication network and a subcube-allocation scheme based on the Gray codes, and showed how to fully compact all free processors into one or more large subcubes without any deadlock. Huang and Juang [37] assumed the buddy system for subcube allocation and proposed a partial scheme that requires $O(2^d)$ steps of migration to free up a $d$-subcube. Schwederski et al. [69] assumed the communication model of multistage cube networks with either packet- or circuit-switching ability, but no restriction on what type of subcubes is used. The result emphasizes on the migration of a single job; a job can be migrated from a source subcube to a destination subcube in the minimum amount of time. Chen and Lai [14] considered the problem of migrating a single job from one subcube to another on hypercubes with circuit-switching networks; it was shown that no matter how fragmented the hypercube is and no matter what allocation scheme is used, the job’s processes can always be migrated in parallel through disjoint paths between the two subcubes. Circuit switching has been used in
existing hypercubes such as Intel’s iPSC-2 machines [56].

The purpose of this chapter is to present a partial compaction scheme that is able to migrate multiple jobs in parallel and to free up a $d$-subcube in only $d$ steps at most, as compared to $O(2^d)$ steps of [37].

We assume a circuit-switching hypercube that uses the buddy-system allocation strategy. In a migration step, a job is migrated from one subcube, say, $A$ to another subcube, say, $B$. In order to facilitate migrations and avoid interfering with other jobs currently running in the system, it is desirable that each migration step has these properties: (1) structure-preserving: $B$ is also a buddy subcube; (2) adjacency-preserving: two processes in adjacent nodes of $A$ are migrated to adjacent nodes of $B$; (3) path-disjoint: every process has a disjoint migration path from $A$ to $B$; (4) congestion-free: no migration path uses a link belonging in an allocated subcube; and (5) contention-free: the destination subcube, $B$, is free, ready to run. A migration step with these properties is called an effective migration step. We first develop an algorithm that generates a sequence of effective migration steps for freeing up a $d$-subcube; it requires $O(2^d)$ migration steps in the worst case. Then we show how to divide the migration steps into $d$ or fewer sets such that the migration steps in each set can be carried out simultaneously. Thus, defining an effective parallel migration step as a set of effective migration steps which can be carried out all simultaneously, we have an algorithm that frees up a $d$-subcube in at most $d$ parallel migration steps.

One reason for assuming the buddy allocation scheme in this chapter is that it simplifies the subcube compaction problem. In fact, as to be seen in section 5.2,
if a perfect allocation scheme is used, then effective migration steps (in the above sense) do not always exist for subcube compaction. Another reason is that although the buddy system can recognize fewer subcubes than can other existing strategies, a recent result [45] indicated that under most workload conditions, the differences in performance between simple, imperfect allocation schemes (e.g., buddy system and Gray code) and complex, perfect ones (e.g., Free list and MSS) are small; that the job-scheduling discipline has far more impact on performance than allocation strategy, and simple allocation schemes can usually benefit much from scheduling. The buddy system, which is used in many existing hypercubes, is still one of the best subcube allocation schemes.

5.2 Preliminaries

A Boolean n-cube, denoted by $H_n$, is represented by an undirected graph consisting of $2^n$ nodes (processors), each labeled by a unique binary integer $b_n b_{n-1} \ldots b_1$ between 0 and $2^n - 1$. An edge (communication link) is introduced between two nodes iff their addresses differ in exactly one bit. There are totally $n2^{n-1}$ edges. Throughout this chapter, $H_n$ is the hypercube computer under consideration.

A $d$-subcube $H_d$, $d \leq n$, is represented by a ternary string $x_n x_{n-1} \ldots x_1$, where $x_i \in \{0, 1, \ast\}$ and $\ast$ represents the "don’t care" symbol. There are $2^{n-d} \binom{n}{d}$ different $d$-subcubes in $H_n$. The dimension of a subcube, $Q$, is denoted as $|Q|$.

Let $A = x_n x_{n-1} \ldots x_{i+1} 0 x_{i-1} \ldots x_1$ and $B = x_n x_{n-1} \ldots x_{i+1} 1 x_{i-1} \ldots x_1$ be two subcubes differing on exactly one bit, namely, bit $i$. The edge group between $A$ and $B$, denoted $x_n x_{n-1} \ldots x_{i+1} \xi x_{i-1} \ldots x_1$ is the set of all edges that connect a node in
A to a node in $B$. For example, "*01\xi1*" is the set of edges directly connecting subcubes *0101* and *0111*.

If a subcube is assigned to a job, all nodes and edges within the subcube are said to be \textit{blocked}. Nodes and edges which are not blocked are \textit{free}.

Whatever subcube allocation scheme is used, a job can only be assigned to a subcube \textit{recognizable by} the scheme. An allocation scheme which can recognize all possible subcubes (e.g., Free List and MSS) is \textit{perfect}; otherwise, it is \textit{imperfect}. The \textit{buddy system}, which is imperfect, can only recognize the $d$-subcubes, $0 \leq d \leq n$, of the form $x_n \cdots x_{d+1} \cdots \cdots$, where $x_i, i > d$, is 0 or 1; such subcubes are referred to as the \textit{buddy subcubes}. For a $d$-subcube $Q = x_n \cdots x_{d+1} \cdots \cdots$, $\overline{Q} = x_n \cdots x_{d+2} \overline{x}_{d+1} \cdots \cdots$ denotes the subcube different from $Q$ on bit $d + 1$. $\overline{Q}$ is called the \textit{complement} of $Q$.

A \textit{partition set (P-set)} is any set of disjoint subcubes in the hypercube, recognizable or unrecognizable by the underlying allocation scheme. The set of jobs running in the hypercube, called a \textit{job set (J-set)}, is also represented by a set of subcubes, one for each job. Each subcube in the J-set must be recognizable by the underlying allocation scheme. Thus, any J-set is a P-set, but the reverse is not true. For imperfect allocation schemes, a P-set is not necessarily a J-set. However, for perfect allocation schemes, every P-set is a J-set, as proved below.

\textbf{Lemma 29} If a perfect scheme is used for subcube allocation, then every P-set is a J-set.

\textbf{Proof.} Given any P-set $S = \{Q_1, \ldots, Q_m\}$, where each $Q_i$ is a subcube, we construct a computation scenario which will result in a J-set $= S$. 
1. $2^n$ jobs, each requiring one processor, arrive at the system and are each assigned to a processor.

2. For $i = 1$ to $i = m$ (in the increasing order), the following events happen: (i) all jobs resident in $Q_i$ terminate and leave the hypercube, and (ii) a new job, $J_i$, which requests a $|Q_i|$-subcube, arrives and is assigned to $Q_i$.

3. All jobs, except $J_1, \ldots, J_m$, terminate and leave. The J-set = $S$.

The fragmentation problem refers to the situation where a requesting job is rejected because no subcube (recognizable by the underlying allocation scheme) is large enough to satisfy the request, although the total number of free processors exceeds or equals the size of the requested subcube. Note that this problem may happen even if a perfect allocation scheme is used. (For instance, in an $H_2$, let the J-set = $\{01, 10\}$. Then a request for $H_1$ will be rejected no matter what allocation scheme is used.)

The fragmentation problem can be remedied by job migration. Formally, given a J-set $S$, a migration step $S \xrightarrow{(Q_a,Q_b)} S'$ is to migrate the processes in $Q_a \in S$ to $Q_b$ so that a new J-set $S'$ results, where $|Q_a| = |Q_b|$. Furthermore, for performance reason, we require a migration step be effective in the following sense.

**Definition 32** A migration step $S \xrightarrow{(Q_a,Q_b)} S'$ is effective if the following conditions are satisfied:

(a) **structure-preserving:** $S'$ is a J-set based on the same allocation scheme.

(b) **adjacency-preserving:** There is a bijection function $f : Q_a \rightarrow Q_b$ such that node-adjacency in $Q_a$ is preserved in $Q_b$. 
(c) **path-disjoint:** For every node $x \in Q_a$, there is a migration path from $x$ to $f(x)$, which is edge-disjoint from other migration paths.

(d) **congestion-free:** No migration path uses a blocked edge.

(e) **contention-free:** $Q_b$ is a free subcube so that $Q_a$, after arriving at $Q_b$, can resume running immediately. □

**Sequential Cube-Compaction Problem:** Let $S_0$ be the present J-set based on some subcube allocation scheme $\Phi$, and assume a new job arrives requesting a $d$-subcube. Suppose that no free $d$-subcube exists which is recognizable by $\Phi$. The problem is to find a sequence of effective migration steps

$$
S_0 \xrightarrow{(Q_{a_1}, Q_{b_1})} S_1 \xrightarrow{(Q_{a_2}, Q_{b_2})} S_2 \xrightarrow{(Q_{a_3}, Q_{b_3})} \ldots \xrightarrow{(Q_{a_m}, Q_{b_m})} S_m
$$

such that after migrating each $Q_{a_i}$ to $Q_{b_i}$, $\Phi$ can recognize a free $d$-subcube when J-set $= S_m$, as long as the number of free processors is no less than $2^d$. □

Naturally, while solving the above problem, we would like to exploit more parallelism, if possible. Define a parallel migration step as $S \xrightarrow{\text{Pair}} S'$, where $\text{Pair} = \{(Q_{a_1}, Q_{b_1}), \ldots, (Q_{a_k}, Q_{b_k})\}$ is a set of $k$ migration steps. In a parallel migration step, all $Q_{a_i}$, $i = 1..k$, are migrated to $Q_{b_i}$ at the same time. A parallel migration step is effective if each migration step $S_{i-1} \xrightarrow{(Q_{a_i}, Q_{b_i})} S_i$ is effective, and all of them use disjoint migration paths. We thus have a slightly different problem:

**Parallel Cube-Compaction Problem:** This is similar to the sequential one,
except that the goal is to find a sequence of effective parallel migration steps

\[ S_0 \xrightarrow{\text{Pair}_1} S_1 \xrightarrow{\text{Pair}_2} S_2 \xrightarrow{\text{Pair}_3} \ldots \xrightarrow{\text{Pair}_m} S_m \]

Call a cube-compaction algorithm effective if it always succeeds in finding a sequence of effective (sequential or parallel) migration steps, as long as the number of free processors is sufficient. The following theorem indicates that such an algorithm might not exist if a perfect allocation scheme is used.

**Theorem 13** There exists no effective cube-compaction algorithm (sequential or parallel) for hypercubes which use a perfect subcube allocation scheme.

**Proof.** This theorem is proved by a counter example as shown in Fig. 32. It is evident that no sequence of effective migration steps may lead to a free \( H_1 \).

In the next two sections, we develop two effective algorithms for the cube-compaction problem, one sequential and the other parallel, assuming the buddy allocation strategy. From now on, all subcubes are referred to as buddy subcubes.
5.3 Sequential Cube-Compaction Algorithm

We first show how to find parallel migration paths between two subcubes of the same dimension. Let \( S \) be a buddy J-set, and \( Q_a = x_n \ldots x_{d+1} \ldots \) be a \( d \)-subcube \( \in S \). Suppose that \( Q_b = y_n \ldots y_{d+1} \ldots \) is a free \( d \)-subcube to which we will move the job in \( Q_a \). Define a bijection function \( f(x_n \ldots x_{d+1} z_d \ldots z_1) = y_n \ldots y_{d+1} z_d \ldots z_1 \), where \( z_i, i \leq d \), is 0 or 1. Clearly, the node-adjacency in \( Q_a \) is preserved in \( Q_b \) through \( f \). Furthermore, we migrate \( x_n \ldots x_{d+1} z_d \ldots z_1 \) to \( y_n \ldots y_{d+1} z_d \ldots z_1 \) by first changing \( x_n \) to \( y_n \), if \( x_n \neq y_n \), then changing \( x_{n-1} \) to \( y_{n-1} \), if \( x_{n-1} \neq y_{n-1} \), and so on. (Note: this is also one of the shortest paths.) These paths can also be described in terms of edge groups: the whole job will migrate from \( Q_a \) to \( Q_b \) through edge group \( \xi x_{n-1} \ldots x_{d+1} \ldots \), if \( x_n \neq y_n \), then edge group \( y_n \xi x_{n-2} \ldots x_{d+1} \ldots \), if \( x_{n-1} \neq y_{n-1} \), and so on. We describe these paths as a set of edge groups:

\[
Paths(Q_a, Q_b) = \bigcup_{i=d+1}^{n} \{y_n \ldots y_{i+1} \xi x_{i-1} \ldots x_{d+1} \ldots : x_i \neq y_i\}.
\]

Evidently, the above migration step \( S \xrightarrow{(Q_a, Q_b)} S' \) satisfies conditions (a-c) and (e) of Definition 32. Thus, \( Paths(Q_a, Q_b) \) will realize an effective migration step from \( Q_a \) to \( Q_b \), if no edge in it is a blocked edge. The next lemma shows that \( Paths(Q_a, Q_b) \) really contains no blocked edge.

**Lemma 30** Let \( S \) be a buddy J-set, and \( Q_a = x_n \ldots x_{d+1} \ldots \) be a \( d \)-subcube \( \in S \). Suppose that \( Q_b = y_n \ldots y_{d+1} \ldots \) is a free \( d \)-subcube. Then \( Paths(Q_a, Q_b) \) does not contain any blocked edge.
Proof. Suppose, for contradiction, that some edge \( e = y_n \cdots y_{i+1} x_{i-1} \cdots x_{d+1} z_d \cdots z_1 \in Paths(Q_a, Q_b) \) is a blocked edge, where \( z_j, j \leq d, \) is 0 or 1. Then there is a subcube \( X \in S \) containing \( e, \) which implies that \( X \) covers at least the subcube \( y_n \cdots y_{i+1} * x_{i-1} \cdots x_{d+1} z_d \cdots z_1. \) Furthermore, since \( X \) is a buddy subcube, all trailing symbols in \( X \)'s representation must be *'s. Thus, \( X \) covers the subcube \( y_n \cdots y_{i+1} * \cdots *, \) which covers the free subcube \( Q_b, \) a contradiction. \( \square \)

Now we present the cube-compaction algorithm. Let \( S \) be a buddy J-set for which no free buddy \( d \)-subcube exists but the number of free processors is no less than \( 2^d. \) The following algorithm sequentially migrates jobs, one at a time, and frees up a \( d \)-subcube.

**Algorithm: SCC /* Sequential Cube-Compaction */**

1. Let \( F = \{F_1, F_2, \ldots, F_m\} \) be a set of free subcubes of totally \( 2^d \) processors.
   (In a buddy system, it is trivial to find \( F. \)) We assume that there are no two distinct \( F_p, F_q \in F \) such that \( F_p = \overline{F_q}; \) otherwise, combine \( F_p \) and \( F_q \) into a larger subcube.

2. Let \( F_i \) and \( F_j \) be the two smallest subcubes in \( F. \) (They must be of the same dimension.) Delete \( F_i \) and \( F_j \) from \( F. \)

3. Consider the complement cube \( \overline{F_i} \) of \( F_i. \) If \( \overline{F_i} \) contains some job(s), then migrate them in one step through \( Paths(\overline{F_i}, F_j) \) to \( F_j. \) (Note that, in the buddy system, because \( F_i \) is free, no job inside \( \overline{F_i} \) can own nodes \( \notin \overline{F_i}. \))
4. Combine $F_i$ with $F_i$ to form a larger subcube. Add the new subcube to $F$.

Also, repeatedly combine $F_k$ and $F_k \in F$ to a larger subcube until no more mergence is possible.

5. Repeat steps 2–4 until $|F| = 1$. The only subcube in $F$ is a free $d$-subcube.

By Lemma 30, it is evident that every migration step used by Algorithm SCC is effective. Furthermore, the following facts guarantee the algorithm to end with a free $d$-subcube: (1) $|F|$ is a decreasing function, and (2) After each iteration, the total number of processors in $F$ is invariant. The following theorem is thus proved.

**Theorem 14** The algorithm SCC is an effective cube-compaction algorithm.

In the worse case (for example, $2^d$ 0-subcubes evenly distributed over the hypercube), algorithm SCC may need as many as $O(2^d)$ migration steps to generate a $d$-subcube, which obviously is not tolerable. In the next section, we modify the above algorithm to cut the migration time to $d$.

### 5.4 Parallel Cube-Compaction Algorithm

In algorithm SCC, we compact a pair of free subcubes *each time*. In this section, we modify SCC to a parallel one, such that multiple pairs of subcubes can be compacted at the same time. Specifically, we find a sequence of parallel migration steps. In each step, we compact all free $i$-subcubes in parallel to $(i + 1)$-subcubes. By repeating the above process from $i = 0$ to $i = d - 1$, we end up with a free $d$-subcube. So at most $d$ parallel migration steps are needed.
In the following, we first develop the key procedure called \textit{Match()}, given a set of free equal-dimensional subcubes, whose function is to find a parallel migration step for them. Then we develop the parallel cube-compaction algorithm \textit{PCC} based on \textit{Match()}.

5.4.1 Procedure \textit{Match()}

Let $F = \{F_1, \ldots, F_m\}$ be a set of free subcubes, each of dimension $d$, and $m$ is an even number. Assume, for ease of presentation, that their complement subcubes $\overline{F}_i$, $1 \leq i \leq m$, are not free, i.e., each contains some job(s). (Recall that no job inside $\overline{F}_i$ will own nodes not belonging to $\overline{F}_i$.) Given $F$ as the parameter, the purpose of procedure \textit{Match()} is to “pair off” the contents of $F$, or, specifically, to calculate a set $\{(\overline{F}_{i_1}, F_{i_2})|i = 1 \ldots m/2, 1 \leq i_1 \leq m, 1 \leq i_2 \leq m\}$ such that the migration paths $\text{Paths}(\overline{F}_{i_1}, F_{i_2})$, $1 \leq i \leq m/2$, are disjoint with each other. Thus, \textit{Match()} finds a parallel migration step such that $m$ free $d$-subcubes can be compacted at the same time to form $m/2$ free $(d+1)$-subcubes.

Fig. 33 is the procedure \textit{Match()}. Three formal parameters and one global variable are explained below:

1. $F$ is a set of free subcubes as stated above.

2. \textit{job} may refer to any $\overline{F}_i$, or 0. When \textit{job} = $\overline{F}_{i_1}$, it means a set of job(s) contained in the subcube $\overline{F}_{i_1}$ is to be migrated to somewhere.

3. $i$ is an integer to index the $i$-th bit of a subcube. (Specifically, if $G = g_n \cdots g_1$ is a subcube, we use a new notation $G^{(i)}$ to denote $g_i$.)
Procedure Match $(F, job, i)$  /* to match the content of $F$ in pairs */
begin
    case of
    C1: $(|F| = 0)$:
        nothing;
    C2: $(job = \emptyset)$: /* $|F|$ = even */
        $P_0 = \{G \mid G \in F \land G^{(i)} = 0\}$;
        $P_1 = \{G \mid G \in F \land G^{(i)} = 1\}$;
    C2.1: if $|P_0|$ = even then
        call Match($P_0, \emptyset, i - 1$);
        call Match($P_1, \emptyset, i - 1$);
    C2.2: else /* $|P|$ = even */
        Let $G$ be any element of $P_0$;
        call Match($P_0 - \{G\}, \emptyset, i - 1$);
        call Match($P_1, G, i - 1$);
    end if;
    C3: $(job \neq \emptyset)$: /* $|F|$ = odd */
    C3.1: if $|F| > 1$ then
        $P_0 = \{G \mid G \in F \land G^{(i)} = 0\}$;
        $P_1 = \{G \mid G \in F \land G^{(i)} = 1\}$;
    C3.1.1: if $|P_0|$ = even then
        call Match($P_0, \emptyset, i - 1$);
        call Match($P_1, job, i - 1$);
    C3.1.2: else /* $|P_1|$ = even */
        call Match($P_0, job, i - 1$);
        call Match($P_1, \emptyset, i - 1$);
    end if;
    C3.2: else /* $|F| = 1$ */
        Let $G$ be the single element of $F$;
        $Pair := Pair \cup \{(job, G)\}$;
    end if;
    end case;
end.

Figure 33: The procedure Match().
4. The result is returned from the global variable \( P \text{air} \), which contains a set of subcube pairs of the format \( (F_i, F_j) \) as described above. Initially, \( P \text{air} = \emptyset \).

All other variables are local variables which are self-explained by their contexts.

Initially, we call the procedure by saying \( \text{Match}(F, 0, n) \), where \( n \) is the dimension of the hypercube. Procedure \( \text{Match}() \) is a recursive procedure, which follows a divide-and-conquer rule. Throughout the explanation, keep in mind that at any call, the total number of subcubes in \( F \) and in \( \text{job} \) are even, called as an \( E \)-property (If \( \text{job} \neq \emptyset \), it counts for 1, otherwise, 0). It is clear that in the initial call the \( E \)-property holds.

(Divide) The dividing process has two cases: \( \text{job} = \emptyset \) (C2) and \( \text{job} \neq \emptyset \) (C3). Consider C2 first. We first partition \( F \) into two subsets \( P_0 \) and \( P_1 \) according to \( i \): subcubes with their \( i \)-th bit = 0 are in \( P_0 \), and with \( i \)-th bit = 1 in \( P_1 \). Intuitively, we partition the hypercube into two subcubes, say \( S_0 \) and \( S_1 \), with \( S_0 \) containing \( P_0 \) and \( S_1 \) containing \( P_1 \). (\( S_0 \) and \( S_1 \) are in a recursive sense; both of them will be further divided into smaller subcubes in the next, deeper recursion. At the beginning, \( S_0 \) and \( S_1 \) are \((n-1)\)-subcubes. In general, they are \((i-1)\)-subcubes. Also note that the division is from the most significant bit to the least one.) We have two cases:

1. (C2.1) If both \( |P_0| \) and \( |P_1| \) are even (recalling the \( E \)-property), then we recursively call \( \text{Match}() \) with \( P_0 \) and \( P_1 \) as parameters, respectively. The key point is that, during later recursion, we guarantee that \( P_0 \) will be compacted within \( S_0 \), and similarly \( P_1 \) within \( S_1 \).

2. (C2.2) If both \( |P_0| \) and \( |P_1| \) are odd, then we pick any \( G \in P_0 \). Imagine that \( \overline{G} \), which contains some \( \text{job} \)(s), is migrated from \( S_0 \) to \( S_1 \) through the edges
directly connecting \( S_0 \) and \( S_1 \). (\( \overline{G}' \)'s destination, however, is unknown by now.) Then we call \( \text{Match()} \) twice with \( P_0 - \{ G \} \) and with \( P_1 \cup \{ \overline{G} \} \), respectively, still preserving the E-property in each call. The key point is that only \( \overline{G} \) will travel on the edges directly connecting \( S_0 \) and \( S_1 \), and, during later recursion, the compaction of \( P_0 - \{ G \} \) will be solved inside \( S_0 \), and \( P_1 \cup \{ \overline{G} \} \) will be solved inside \( S_1 \). No other subcubes can travel on the edges directly connecting \( S_0 \) and \( S_1 \), so the migration is conflict-free.

Note how index \( i \) changes. By decreasing \( i \), the aforementioned \( S_0 \) and \( S_1 \) are recursively divided into smaller subcubes. The dividing process will eventually terminate. Moreover, jobs which are migrated are changing their addresses from the most significant bit to the least significant one, which is the sufficient condition to apply Lemma 30 to guarantee that no blocked edge is traversed.

Now we consider the case \( \text{job} \neq \emptyset \) (C3). If \( |F| > 1 \) (see C3.1), partition \( F \) into \( P_0 \) and \( P_1 \) as above. We, again, use the concept that the subcube (which is passed from the previous recursion) is partitioned into 2 smaller parts: an \( S_0 \) containing \( P_0 \) and an \( S_1 \) containing \( P_1 \). Recall that before this call \( \text{job} \) has traveled to somewhere in \( S_0 \) or \( S_1 \), depending on the bit \( \text{job}^{(i)} \). Now, consider two cases:

1. \( \text{(C3.1.1)} \) If \( |P_0| \) is even (viz. \( |P_1| \) is odd), then we call \( \text{Match()} \) twice with \( P_0 \) and \( \emptyset \), and with \( P_1 \) and \( \text{job} \), respectively. If \( \text{job}^{(i)} = 0 \), then we imagine that \( \text{job} \) is migrated from \( S_0 \) to \( S_1 \) through the edges directly connecting \( S_0 \) and \( S_1 \) (again, the destination of \( \text{job} \) is unknown yet); otherwise, edges connecting \( S_0 \) and \( S_1 \) will not be used. Still, the E-property holds in each call. Also keep in
mind that in later recursion, the compaction of \( P_0 \) will be done inside \( S_0 \), and \( P_1 \cup \{job\} \) inside \( S_1 \) to guarantee no conflict.

2. (C3.1.2) \(|P_0|\) is odd (viz. \(|P_1|\) is even). The argument is similar to (C3.1.1).

(Conquer) Finally, the recursion is terminated either when \(|F| = 0\) (C1), or when \(job \neq \emptyset\) and \(|F| = 1\) (C3.2), which are discussed in the following:

1. (C1) No compaction is needed here, since both \( F \) and \( job \) are \( \emptyset \).

2. (C3.2) \(job\) is to be migrated to the only free subcube \( G \in F\). Add \((job,G)\) to the set \(Pair\). The migration paths will not conflict with any other migrations.

Lemma 31 Let \( n \) be the dimension of the hypercube, \( F = \{F_1, F_2, \ldots, F_m\} \) be a set of free subcubes, each of dimension \( d < n \), and \( m \) be an even number. Then calling \(Match(F,\emptyset,n)\) will return a set \(Pair = \{((F_i_1,F_i_2)|i = 1...m/2}\), and all migration paths \(Paths(F_i, F_{i+1})\) are edge-disjoint.

5.4.2 Algorithm PC\( C\)

With procedure \(Match()\) at hand, the parallel cube-compaction algorithm \(PC\( C\)\) is easy to develop. We are given a buddy J-set \(S\), in which no free buddy \(d\)-subcube exists, but the number of free processors are no less than \(2^d\). Our goal is to migrate jobs in the hypercube to form a free \(d\)-subcube.

Algorithm \(PC\( C\)\) is shown in the following. The main idea is compacting all 0-subcubes (if any) by calling \(Match()\), then compacting all 1-subcubes (if any) by calling \(Match()\), ..., until a \(d\)-subcube is constructed. The algorithm does not include
the trivial step of finding a set of free subcubes $F = \{F_1, F_2, \ldots, F_m\}$, which contains totally $2^d$ processors. So $F$ is assumed to be known ahead.

**Algorithm: PCC** /* Parallel Cube-Compaction */

for $i := 0$ to $d - 1$ do

Let $F' = \{G | G \in F \land (G \text{ is of dimension } i)\}$

C1: if $F' \neq \emptyset$ then

$F = F - F'$;

$Pair = \emptyset$;

call Match($F', \emptyset, n$);

forall $(\overline{F}_{i_1}, F_{i_2}) \in Pair$ /* parallel migrations */

Migrate jobs in $\overline{F}_{i_1}$ to $F_{i_2}$ through $Paths(\overline{F}_{i_1}, F_{i_2})$;

$F = F \cup \{F_{i_1} + \overline{F}_{i_1}\}$;

end forall;

Adjust $F'$ by combining each $G \in F$ and $\overline{G} \in F$ to a larger subcube;

end if;

end for;

In the algorithm, we first construct a subset $F'$, which contains all $i$-subcubes in $F$, from smaller $i$ to larger $i$. Block C1 will be entered if $F'$ is not an empty set. Note an important fact that $|F'|$ is always an even number. Thus, we can call Match() with $F'$ as the parameter. The result is returned from Pair. For all $(\overline{F}_{i_1}, F_{i_2}) \in Pair$, we migrate $\overline{F}_{i_1}$ in parallel through $Paths(\overline{F}_{i_1}, F_{i_2})$ to $F_{i_2}$. After the migration, $|F'|/2$ free
(i + 1)-subcubes are obtained and are added to F. (We denote by $F_i^++F_i^-$ the new free $(i + 1)$-subcubes.) Because of the above change in $F$, some adjustment on $F$ is necessary: we combine all possible $G$ and $\overline{G} \in F$ to form larger buddy subcubes. Also note that in the above, when we say “migrate jobs in $F_i^+$ to $F_i^-$,” the migration might be just vacuous if $F_i^+$ is a free subcube (but $F_i^- \notin F$). We say so for convenience, or the reader can imagine that the migration finishes instantly.

**Theorem 15** The algorithm PCC is an effective cube-compaction algorithm.

Algorithm PCC uses at most $d$ parallel migration steps to construct a $d$-subcube. If we assume that all migrations take a constant time, then it is easy to prove that, under our definition for effectiveness, this number of migration steps is optimal in the worst case. We can make up a very-fragmented situation where there are only $2^d$ free processors, and each one is a 0-subcube. Furthermore, for every free 0-subcube $A$, we enforce $A$ be occupied by a job, $\overline{A}$ be occupied by a job, and $A + A + A + \overline{A}$ be occupied by a job, ..., and so on. Then no matter how migration is performed, because only a free subcube’s complement can be moved to clean up a larger subcube, the dimension of the free subcubes can only be advanced by 1 in each step. Thus, at most $d$ migration steps are needed in the worst case.

### 5.5 Concluding Remarks

A set of requirements — structure-preserving, adjacency-preserving, path-disjoint, congestion-free, and contention-free — is proposed to ensure the effectiveness of a job migration algorithm. To follow these criteria is very important so as to keep the
job migration cost low. In addition, it maximizes the communication speed in the circuit-switching networks, which use comparably longer time in setting up switches, but have comparably higher transmission rate once the circuits are set up. We have presented two job migration algorithms, both of which satisfy these criteria. The latter is optimal in its migration time in the worst case, assuming all migrations take a constant time.

The first step of selecting $F$ (a set of free subcubes) in both algorithms, although trivial, has large impact in average-case performance. Greedily constructing $F$ with larger free subcubes will, most likely, reduce the compaction time. However, after migration the hypercube may remain fragmented because there are still many small free subcubes; the next compaction might be necessary very soon. We are neutral in this point and leave the flexibility to the designers.

Some heuristics are also possible during compaction. For example, in SCC, let $A$ and $B$ be two free subcubes of the same dimension. Moving $\overline{A}$ to $B$ and moving $\overline{B}$ to $A$ are the same. The design choice may depend on the status of $A$'s and $B$'s neighbors.

Future research may be on extending these results to other allocation strategies.
CHAPTER VI

Conclusion

6.1 Summary

We have classified the whole process from the presentation of a problem to be solved in a parallel/distributed system, to its program execution in the system, into four phases: parallel program design, program verification, program implementation, and resource allocation. Then we selected four problems, one from each phase, that seem to address current issues. Their solutions have been presented.

The summary of each chapter is given in detail as follows.

In Chapter II, we studied a grid iteration problem. Many important scientific problems[58, 77], elliptic partial differential equation, finite element method, molecular dynamics, lattice gauge theory, and crystallization simulation, may be solved by an iterative technique on a grid.

A stencil is a graph representing the data dependency for the grid iteration problem. In this study, given a stencil, we have devised a method to formulate the corresponding pseudo-optimal partitioning structure that minimizes the communication overhead in each partition under two approximations. Our results on pseudo-optimal partitioning structures for five common stencils update the results in Reed et al.[64].
Once a pseudo-optimal partitioning structure is derived, the grid is transformed into a *Partition Dependency Graph* (PDG). The assignment of the PDG into a multiprocessor system has been considered.

The method developed in our approach can also be used for the *Directed Acyclic Graph* (DAG) assignment problem in which the DAG represents the data dependency among computational tasks.

In Chapter III, we studied deadlock detection and resolution. Many distributed algorithms proposed in literature turn out to be incorrect. The main reason why such frequent mistakes occur is that any deadlock detection algorithm is too complex to use an informal argument or operational approach to prove its correctness. Thus more strict, formal proof system is naturally required in the proof.

In this study, we proposed an efficient deadlock detection and resolution algorithm. Then, a simple computational model and its proof system have been proposed. The proof system turns out to be simple to use and reduces total work in the proof significantly. Based on the proof system, we have proved two correctness conditions of the proposed algorithm: a safety condition that no phantom deadlock is detected, and a progress condition that any true deadlock is eventually detected.

Our computational model and its proof system have a general applicability to the other classes of algorithms. We have proved the soundness of our computational model. Particularly, we showed they are applicable to any distributed stability detection algorithms. For example, they can be applied to the correctness proof of the distributed termination detection algorithm.
In Chapter IV, we studied graph embedding in a hypercube. Embedding a source graph in a host graph has long been used to model the problem of processor allocation in a multicomputer system. If the host graph represents a network of processors that uses circuit switching for node-to-node communication, then embedding a graph with congestion one is practically as good as embedding it with adjacency preserved, but has the advantage of allowing far more graphs to be embeddable. In this study, we have shown that any graph $G$ can be embedded with unit congestion in a hypercube of dimension $n \geq \max\{6|\log |V(G)||, G\}$, but it is NP-complete to determine whether $G$ is congestion-1 embeddable in a given hypercube of dimension less than $\max\{6|\log |V(G)||, G\}$, even if the source graph is connected. The restriction to connected graphs is important because, in applications, source graphs are usually connected.

In Chapter V, we studied subcube compaction. A hypercube can be partitioned into subcubes of various sizes to run independent jobs. As jobs arrive, grabbing the subcubes, and leave, releasing the subcubes, the system tends to become fragmented. When this happens, one solution is to relocate (or migrate) jobs so as to compact free processors into bigger subcubes. During the process of compaction, it is desirable that each migration step is effective: structure-preserving, adjacency-preserving, path-disjoint, congestion-free, and contention-free. In this study, an effective parallel algorithm for subcube compaction has been proposed for a hypercube system that uses the buddy-system allocation strategy and the circuit-switching communication model. Assuming that the hypercube has at least $2^d$ free processors, the algorithm
takes at most $d$ parallel migration steps to free up a $d$-subcube. By comparison, the best existing algorithm requires $O(2^d)$ migration steps to achieve that.

6.2 Future Research Issues

The procedure deriving pseudo-optimal partitioning structures in Chapter II needs to be expanded from two-dimensional grid to multi-dimensional grid. In our study about partitioning a grid, we ignored the effect of the boundary of the grid. Since the data exchange across the boundary does not occur, a better formulation of the partitioning structure problem that incorporates this condition is desirable to get a more realistic solution.

The sequential computational model proposed in Chapter III significantly reduces the total amount of steps in the program correctness proof. Thus, searching for the additional classes of algorithms where our model is equivalent to the real-time computation seems to be a valuable research direction.

We have also studied the embedding of a graph $G$ in a hypercube. Contrary to our main results in Chapter IV, many important graphs can be embedded with congestion one in a hypercube of dimension $\left\lceil \log |V(G)| \right\rceil$. They include rings, full binary trees, full quad-trees and most meshes [83]. A pyramid can be embedded in $H_{\left\lceil \log |V| \right\rceil}$ with congestion two [49], and it is still an open problem whether congestion-1 embedding is possible.

Extending our subcube compaction schemes in Chapter V to the other allocation schemes is a possible next step of research in this area. Since subcube compaction time is an important performance measure, it is desirable to reduce the number of
migration steps in the worst case still further. Regarding this, it is an open problem whether a scheme of constant migration steps $O(1)$ exists.
BIBLIOGRAPHY


