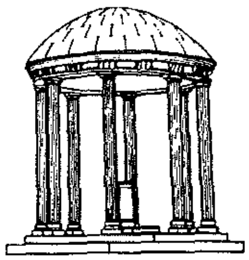
A New Class of Recursive Routing Algorithms on Mesh-connected Computers

TR90-044

December, 1990

Taisook Han



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

A New Class of Recursive Routing Algorithms on Mesh-connected Computers

by

Taisook Han

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill, 1990

Approved by:

[illegible]

Donald F. Stanat, advisor

Gyula Magó

Gyula A. Magó, reader

Jennifer L. Welch

Jennifer Welch, reader

TAISOOK HAN. A New Class of Recursive Routing Algorithms on Mesh-connected Computers. (Under the direction of Donald F. Stanat.)

Abstract

We describe a class of deterministic routing algorithms called "move and smooth" algorithms for one-to-one and one-to-many message routing problems on meshes. Initially, each processor contains at most one message, and each message has one or more designated destinations. No processor is the destination of more than one message.

Move and smooth algorithms are recursive. Initially, the entire mesh is considered a single region. At each recursive stage,

- Each region is partitioned into contiguous subregions;
- A copy of each message is moved to each of the regions that contains one of its destinations (the *move* phase);
- Messages within each region are re-distributed so that each processor contains at most one message (the *smooth* phase).

The recursion continues until each region contains a single row or column of processors, at which time each message has arrived at or can be moved directly to its destination.

We examine two representative move and smooth algorithms in detail. On a square n by n mesh, one of the algorithms requires $5.5n$ parallel communication steps and five buffers per processor; the other requires $9n$ parallel communication steps and three buffers per processor. We show that under appropriate assumptions, these costs are not changed for one-to-many routing problems. The number of buffers is independent of the size of the mesh.

The costs of our move and smooth algorithms are higher than those of some one-to-one routing algorithms, but move and smooth algorithms are suited for a wider variety of problems including one-to-many problems. These algorithms do not rely on sorting. We describe other move and smooth algorithms, and generalizations to higher dimensions.

Acknowledgments

My advisor, Dr. Donald F. Stanat, deserves my sincere gratitude and appreciation. He continuously gave me sparkling ideas and encouraged me. His enthusiastic guidance and support make it possible to accomplish my research in time.

I am also indebted to Dr. Gyula A. Magó for his advice and review of the work. He suggested the dissertation topic and listened patiently to my questions. His comments and careful reading were essential to this dissertation.

I would like to thank the other members of my committee, Dr. David Plaisted and Dr. Akhilesh Tyagi for their suggestions of ideas and improvements, and especially Dr. Jennifer Welch who reviewed my draft and made constructive comments.

Finally, I would like to thank to my family for their support and endurance. My research would not have been possible without love from my wife and daughter.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation describes a class of routing algorithms for a mesh-connected processor system. In Section 1.1, we describe the reasons behind the widespread use of the mesh topology and the role of routing algorithms in parallel processing systems. We state characteristics of an algorithm of special interest to us and give a brief description of a class of routing algorithms introduced in this dissertation. In Section 1.2, we summarize the chapters of this dissertation. In Section 1.3, we define the terms to be used throughout this dissertation and describe the network model to elaborate our algorithms. We also characterize routing problems that our algorithm will solve. In Section 1.4, we summarize the research literature on routing and sorting algorithms on a mesh.

1.1 The Problem

In recent years, much research has explored the potential of parallelism for various problems. The design of many parallel algorithms is based on a computational model with a sufficient number of processors and with conflict-free shared memory. The PRAM (Parallel Random Access Machine) model is one of the typical idealized parallel computers. It has a common global random access memory that is shared by many processors. Each processor can read and write any part of the memory for a uniform cost. Data exchange among processors is done through the shared memory. Analysis of many algorithms is based on such a theoretical computational model.

But with current technology, a multiprocessor with global memory and uniform access cost is not realizable. Memories are either shared by processors or local to processors. In both cases, processors access memories through interconnection networks which connect either processors themselves or processors and memory modules. As a consequence, the communication cost between processors and memory modules cannot be considered uniform. So, data movement among processors is an important factor in the performance of multiprocessor systems.

Because of the limited communication bandwidth among processors, rearranging data is costly; data exchange among processors can be a major time factor in the implementation of practical parallel algorithms. To facilitate data exchange among processors, several interconnection schemes for multiprocessor systems have been proposed and used, including binary tree, hypercube, butterfly, and mesh networks. Implementations differ in channel bandwidth, channels per node, and connection distance between nodes. Some multistage interconnection networks have also been investigated, such as the omega, baseline, indirect binary n-cube, and delta networks. Each of these uses several stages of switching elements to connect processors to processors or processors to memories.

A multiprocessor system with an interconnection network uses a routing algorithm to support fast data exchange among processors. In general, such an algorithm must be able to handle diverse patterns of routing requests. Some problems such as FFT and matrix multiplication require highly regular patterns of data exchanges among processors, but many problems, such as simulating a PRAM model, require infrequent and irregular processor communication. An efficient routing method capable of handling diverse routing requests can improve the performance of a multiprocessor system.

Routing efficiency is profoundly affected by the architecture and interconnection topology of networks. For example, a good routing method for the hypercube connection may not work well for a mesh, because a hypercube connection has more links per node and a smaller network diameter than a mesh connection with the same number of processors; a routing method on a hypercube connection can utilize the many outgoing edges of a node and the small network diameter for fast transit of messages. The nature of applications is also an important factor in choosing a routing method, because the application determines the regularity of data communication patterns as well as the frequency, and size of the data objects.

Routing efficiency is also affected by whether communication is synchronous, and whether it is globally controlled. Synchronous communication is generally faster than asynchronous, because no handshaking is necessary among processors. If a network is controlled by a single global controller, algorithms that use global information can sometimes determine shorter routes, resulting in faster performance than other algorithms that lack such information.

This dissertation is concerned with routing algorithms for a two dimensional rectangular mesh. The two dimensional mesh interconnection scheme has long been of interest because of its simplicity and the regularity of its interconnection pattern [1]. Although the maximum distance between processors is large compared to other interconnection patterns[1], a mesh is wire-efficient when implemented in VLSI, making possible wide buses among processors for increased communication efficiency [5]. The locality of communication is another strength of the mesh for programming; in a two dimensional mesh, a processor can communicate with only four neighbors. Finally, meshes can be scaled up to larger systems.

This dissertation considers a class of routing algorithms for one-to-one communication patterns and one-to-many communication patterns. In one-to-one communication, a processor in a mesh has a single message that will be delivered to another processor, and no processor in the mesh will receive more than one message. In one-to-many communication, a processor has a single message that will be delivered to one or many processor(s), but no processor will receive more than one message. We are especially interested in such one-to-many communication.

The purpose of this dissertation is to construct routing algorithms for moving packets among processors on a mesh. Our routing algorithms assume packet switching. The processors have their own program memories and request data exchange synchronously, based on a global clock. This dissertation describes a class of *move and smooth* algorithms which have the following properties:

- The algorithms are deterministic.
- The algorithms can handle one-to-one and one-to-many message routing problems without any adjustment; the cost of executing each routing algorithm is the same over all problems.
- The algorithms are distributed; that is, each processor operates independently on the basis of messages in its buffers and its state.
- The number of buffers in each processor is the same, and a constant regardless of the size of mesh.

[1]For an N processor system, the maximum distance between processors in a hypercube is $\log N$, whereas in a mesh the maximum distance is $\sqrt{N}$.

- All processors work in lock-step; no handshaking between processors is necessary.

1.2 Dissertation Organization

In Section 1.3, we introduce the terms to be used throughout this dissertation, and describe our network model. In Section 1.4, we briefly summarize the literature on routing and sorting algorithms for meshes, emphasizing work on deterministic routing algorithms.

The core of this dissertation is contained in Chapters 2 and 3. In Chapter 2, we describe the general notion of move and smooth algorithms and analyze the requirements of the message movements in a one dimensional array of processors. The results developed in Chapter 2 are repeatedly cited to analyze algorithms described in this dissertation. In Chapter 3, we describe Algorithm Q, a typical move and smooth algorithm on a square mesh. The procedures described are easily applicable to a rectangular mesh and to other algorithms of this class.

In Chapter 4, we describe Algorithm H, another example of a move and smooth algorithm. We also extend Algorithm H to another partitioning method, multi-strip partitioning, and to multi-dimensional meshes, and we analyze the time complexity and buffer requirements of the extensions. In Chapter 5, we summarize the results of our research and suggest possibilities for further research.

1.3 Definition of Terms

Much research has been performed on mesh connected computer systems, but the meaning of terms sometimes varies from one report to another. In this section, we introduce the terms to be used throughout this dissertation. First, we describe network models in terms of Feng's classification [6]. Next, we describe the mesh network and the characteristics of processors in our network model. Finally, we define the set of routing problems for our algorithms and the evaluation criteria.

1.3.1 Network Classifications

To describe interconnection networks from a practical design viewpoint, Feng [6] suggests four design decisions: operation mode, control mode, switching method, and network topology. The *operation mode* of an interconnection network may be synchronous, asynchronous, or a combination of the two. In *synchronous communication*, communication paths are specified and messages are sent in lockstep over the entire network, with the aid of a global clock. *Asynchronous communication* allows communication paths to be established and messages sent as required; one set of processors may be exchanging messages concurrently with another set establishing a communication path. There may or may not be a global clock. In a *combined mode*, both synchronous and asynchronous processing are supported.

The *control mode* specifies how the switching elements and links are controlled. Control is classified as either centralized (global) or distributed (local). A network with *centralized*, or global, control is a network in which the individual processors execute instructions specified at each time step by a *global controller program* which has global information and control. In a network with global control, the global controller specifies what messages will be sent by each processor at every time step. A network with *distributed*, or local, control is a network in which processors execute individual (but usually identical) programs independently. The central processor may provide a global clock for synchronization, but each processor executes its own program, and the execution path of a program may depend on any part of a processor's state, including the value of time. In a network with distributed control, each processor determines what messages it will send at each time step.

The *switching method* determines what kinds of paths are established between source and destination processors. In *circuit switching*, a physical path is actually established between a source and a destination, and messages are sent, uninterrupted, along the path. In *packet switching*, data is put in a packet and routed through the interconnection network from one processor to another without establishing a physical connection path. In *integrated switching*, both packet switching and circuit switching are used.

The *network topology* describes how the network is depicted by a graph in which nodes represent processors and edges represent communication links. Examples of network topologies include trees, rings, stars, meshes, and hypercubes as shown in Figure 1.1. The cross product of the set of categories in each design decision—{operation mode} $\times$ {control mode} $\times$ {switching method} $\times$ {network topology}—represents the space of interconnection networks.

This research will focus on communication in **mesh-connected** processors which communicate **synchronously** using **packet switching** under

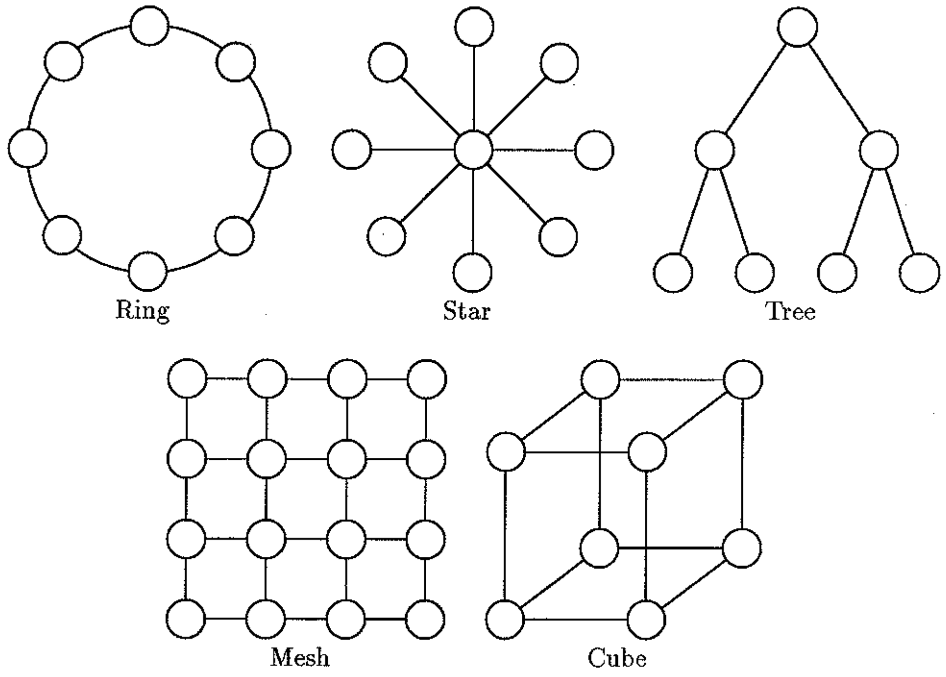


Figure 1.1: Examples of network topologies

distributed control.

1.3.2 Mesh Networks

A two dimensional *mesh-connected* computer system consists of N processors connected as an n by n grid[2], where N is n^2. Processors are on the cross points of the grid, and each processor that is not on the boundary is connected to four neighbors, called its north, south, west, and east neighbors.

The processors in a mesh are always assigned addresses, but addressing schemes vary. Most commonly, a processor $P_{i,j}$ on row i and column j is addressed by a pair of row and column indices (i, j), where $0 \leq i \leq n-1$ and $0 \leq j \leq n-1$. While a pair of row and column indices is sufficient to specify the location of any processor, a linear ordering of the processors is necessary for some problems such as sorting, storing vector elements, and

[2]Meshes, of course, need not be square, but the square configuration is the one most often implemented and investigated. This thesis includes extensions of two algorithms to non-square meshes in Section 3.6.4 and Section 4.6.5.

0	1	2	3	0	1	2	3	0	1	4	5
4	5	6	7	7	6	5	4	2	3	6	7
8	9	10	11	8	9	10	11	8	9	12	13
12	13	14	15	15	14	13	12	10	11	14	15
row-major				snake-like row-major				shuffled row-major			

Figure 1.2: Indexing schemes

computing a prefix sum. An *indexing scheme* makes it possible to address each processor with a single index $A_{I(i,j)}$ where I is a one-to-one mapping from $\{0,1,2,\ldots,n-1\}\times\{0,1,2,\ldots,n-1\}$ to $\{0,1,2,\ldots,n^2-1\}$. Among the many indexing schemes described in the literature, row-major indexing and snake-like row-major indexing are used most commonly. Figure 1.2 shows some common indexing schemes for a four by four array.

In *row-major indexing*, processors are indexed from left to right and from top to bottom. With this indexing scheme, the index $A_{i,j}$ of a processor $P_{i,j}$ is equal to $i \times n + j$. In *snake-like row-major indexing*, processors are indexed from left to right on even-numbered rows, and right to left on odd-numbered rows, from top to bottom. The index $A_{i,j}$ of processor $P_{i,j}$ is given by $i \times n + (-1)^i \times j + \frac{1}{2}\{n - 1 + (-1)^{i+1}(n-1)\}$. In *shuffled row-major indexing* [45], the index $A_{i,j}$ of processor $P_{i,j}$ is given by shuffling the binary representation of the row-major index. For example, shuffling bits of row-major index 5 (0101) gives bit string 0011 which is 3. In *blocked snake-like row-major indexing* [25], the $n \times n$ mesh is divided into $n^{\frac{1}{4}} \times n^{\frac{1}{4}}$ blocks of $n^{\frac{3}{4}} \times n^{\frac{3}{4}}$ processors. Blocks are ordered by the snake-like row-major indexing scheme, and any processor in a lower indexed block has a smaller address than any processor in a higher indexed block. Within a block, processors are ordered by some commonly used indexing scheme.

1.3.3 Properties of Processors

In synchronous packet switching mesh models, processors may execute instructions from a central controller or from their own program memories, but in either case, all processors communicate in lock-step, that is, they all send and receive messages at the same time. All processors rely on the global clock to synchronize communication. Processors also count clock ticks to determine when each phase of an algorithm terminates. A processor can send at most one message to each immediate neighbor at each time step; a receiving processor receives each message in the same time step that it was transmitted.

Two neighboring processors are connected by a *link*, which provides bidirectional communication between the processors; messages can be sent in either direction over a link. Each link is divided into one or two *channels*. The capacity of a channel is one message per unit time. If each link has one channel, it is called *half-duplex*. In a half-duplex model, a processor may send or receive a message (but not both) over each link at each time step. If each link has two channels, it is called *full-duplex*; in this case, adjacent processors can exchange messages in a single time step, because a processor can send and receive messages simultaneously.

A *buffer* is a memory location used to keep messages in a processor; each buffer can store one message. Routing methods vary in the number of buffers required to implement them.

The model used in this dissertation relies on local control, so each processor has its own program memory. Communication between processors is by packet exchange, synchronized by a global clock. Each processor has four links, each with two channels which can be active simultaneously, so a processor may send and receive up to four messages in one time step. Each processor has only a small number of buffers to store transient messages.

1.3.4 Problem Definition

We will consider several classes of communication problems on a mesh. The initial state, or initial configuration, of each problem has at most one message stored in each processor of the mesh. A communication problem requires moving each message to other processors in the mesh. Each final state also has at most one message stored in each processor.

Communication problems can be classified as *sorting* problems or *routing* problems. A sorting problem begins with a collection of messages in an initial configuration, and ends with those messages in some specified subpart of the net, arranged according to a specified linear order. Most commonly, each processor has a message initially, and therefore sorting permutes the messages in the mesh. In a sorting problem, the destination of a message is implicit because the proper destination of each message is determined both by its value and the values of the other messages in the mesh.

In a routing problem, each message is associated with (one or more) *destination* addresses; a copy of each message must be moved to all of its

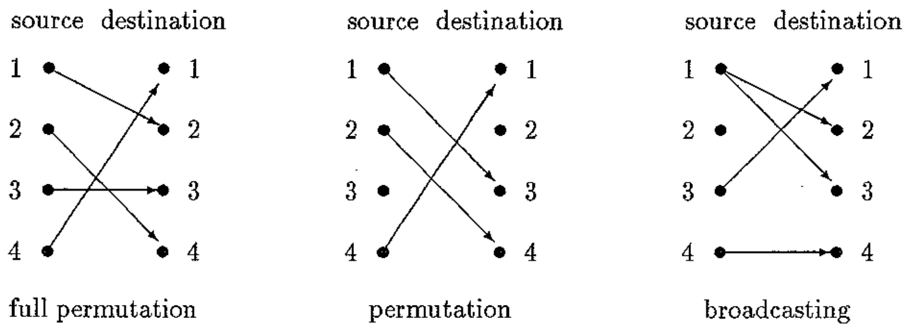


Figure 1.3: Mapping from source to destination

destination addresses. Routing on meshes is important for solving problems such as matrix multiplication and FFT. We will consider several subclasses of routing problems. A *full permutation* problem will be one in which each of N processors has a single message initially and finally, and each message is sent to exactly one processor; thus, a full permutation effects a one to one map of initial messages onto processors. A full permutation problem can be solved by a sorting algorithm, since the messages can be sorted according to their destination addresses. A *(partial) permutation* is a routing problem in which each processor begins and ends with at most one message, and each message is sent to exactly one processor. In a (partial) permutation, a processor may initially have a message but not receive one, or vice versa. A *restricted broadcasting* problem is a routing problem in which each processor initially and finally has at most one message, but some messages are sent to more than one destination processor. We will refer to restricted broadcasting simply as "broadcasting." Figure 1.3 illustrates the various types of routing problems.

An important subclass of full permutation problems are known as *bit-permute-complement* (BPC) permutations because the binary representation of each destination index can be obtained by permuting and complementing the bits of the index of the processor that initially holds the message. BPC permutations include a perfect shuffle and a transpose, which suffice for solving a variety of practical routing problems.

Our model specifies that in the initial configuration, each message resides in a single processor, and no processor contains more than one message.

Movement (that is, communication) begins simultaneously over the net, and continues in a synchronous, lock-step fashion, until all messages have reached (all) their destination processors. In the final configuration, each processor has at most one message.

This dissertation will examine a class of routing methods for full permutations, (partial) permutations, and broadcasting. We will only consider routing methods that can be implemented with distributed control in which each processor runs a copy of a single program and makes routing decisions based on local information. In our discussion of previous work, however, we will also consider sorting methods, since these can be used to solve full permutation problems.

1.3.5 Evaluation Criteria

Routing methods, or algorithms, can be evaluated in a number of ways, including their time complexity, the number of buffers required during routing, and the class of mappings they can handle.

Time Complexity There are two classes of operations performed in a mesh; one is inter-processor and the other is intra-processor. We assume that intra-processor operations are very fast compared to inter-processor operations; hence, all our measures of time complexity are based on the number of *routing steps* required to send all messages (perhaps of various types) to their destinations. The cost of a routing method is taken to be the maximum cost, over all problem instances, of the cost of solving each instance. All processors start communication at the same time and stop when the last message is guaranteed to have been delivered to its destination.

Number of Buffers The number of buffers required by a routing method is defined to be the maximum number of messages that can occupy a processor at any time during execution of the algorithm.

Class of Mappings The class of mappings is the class of routing problems that a routing method can handle without any modification.

1.4 Related Work

This section describes work reported in the literature about routing and sorting on a two dimensional mesh. Principal results are discussed, along with descriptions of the models on which the results are based. Table 1.1 summarizes the results. Although some of the work described here assumes a global clock and global control, all the complexity results are relevant to our model, which is based on a global clock and distributed control.

1.4.1 Principal Network Models

The mesh-connected network models described in the references differ in the control scheme of the processors, the details of the connection topology, and the active links per processor. There are three principal models, with minor variations occurring within the models.

The ILLIAC IV was one of the earliest SIMD parallel computers. It had 64 processors connected in an eight by eight mesh. Much of the literature uses the ILLIAC IV as the model underlying their methods for routing and sorting. We will use the term *ILLIAC model*[3] to denote a subclass of mesh-connected processors with a centralized controller. All instructions are broadcast from the controller; depending on the value of a mask bit in the processor, a processor does or does not execute a given broadcast instruction. The controller can address a group of processors by specifying a bit pattern of processor addresses [42]. Communication is the result of the execution of instructions that specify the direction of packet movement. The SIMD control of the ILLIAC IV requires that all active processors send their messages in the same direction (north, south, east, or west); the direction is specified in any communication instruction. This restriction on message direction is incorporated in our ILLIAC model. A "compare-and-exchange" operation between two processors requires two routing steps, since messages must be sent in the same direction in each step and two messages need to be compared in the same processor. (The ILLIAC model is clearly not a distributed control model, but all the algorithms we will discuss can be implemented with distributed control.) A common variation on this model has wrap-around connections.

[3]In much of the literature, the ILLIAC model is called a *SIMD model*, because processors are controlled by single broadcast instruction from a controller. This is a network with centralized control in Feng's classification.

Table 1.1: Models and problems in literature

routing	model	problem set	complexity	remark
Orcutt[31]	ILLIAC	ps,bitr	$4n-4$	wrap-around
Nassimi & Sahni[30]	ILLIAC	BPC	$4n-4$	
Flanders[8]	ILLIAC	BPC		DAP machine
Raghavendra[36]	ILLIAC	perm	$3n-3$	wrap-around
Nassimi & Sahni[29]	ILLIAC	broad,partial	$O(n)$	
Algorithm Q	dist-ctl	broad,partial	$5.5n$	in Chapter 3
Krizanc et al.[14]	dist-ctl	perm,partial	$2n+o(n)$	random
Kunde[18]	dist-ctl	perm,partial	$2n+o(n)$	UB
Leighton et al.[23]	dist-ctl	perm,partial	$2n-2$	constant buffer
Valiant & Brebner[48]	dist-ctl	perm,partial	$3n+o(n)$	random,UB

sorting	model	indexing	complexity	remark
Orcutt[31]	ILLIAC	row	$O(n\log n)$	wrap-around
Thompson & Kung[45]	ILLIAC	snake	$6n+o(n)$	
Nassimi & Sahni[28]	ILLIAC	row	$14n+o(n)$	
Kumar & Hirschberg[15]	ILLIAC	row	$11n+o(n)$	wrap-around
Lang et al.[20]	dist-ctl	snake	$7n$	
Schnorr & Shamir[41]	dist-ctl	snake	$3n+o(n)$	
Scherson & Sen[40]	dist-ctl	BSLRM	$3n+o(n)$	
Sado & Igarashi[39]	dist-ctl	snake	$5.5n+o(n)$	
Ma et al.[25]	ex-ILLIAC	BSLRM	$8n+o(n)$	$n^{\frac{1}{4}}$ buffer

legend:

ILLIAC: ILLIAC model	ex-ILLIAC: extended-ILLIAC model
dist-ctl: distributed control model	ps: perfect shuffle
bitr: bit reverse permutation	BPC: bit-permute-complement permutation
perm: permutation	partial: partial permutation
broad: broadcasting	row: row-major indexing
snake: snake-like indexing	BSLRM: blocked snake-like row-major indexing
random: randomized algorithm	UB: unlimited buffer size

The ILLIAC model requires all processors to send messages in the same direction at any time. Ma [25] suggests an extended model of ILLIAC IV, which we will hereafter call the *extended-ILLIAC model.* Like the ILLIAC model, his model requires processors in the mesh to send messages either along rows or along columns (and not both at the same time). Processors in the same row (or column) must send messages in the same direction if the mesh is doing row-wise (column-wise) communication. His model differs, however, in that different rows can send messages in different directions, and likewise for columns. Thus, his model also uses single instruction-multiple data mode of operation, but it has a more flexible communication facility.

Lang [20] suggests a model that is still more powerful, widely accepted and which will be the model used for the work in this thesis. He uses a *distributed control model*[4], which is a network of mesh-connected processors, each with its own program memory. Each processor executes instructions from its own memory, but communication is synchronized by a global clock. A processor on the mesh can communicate with any subset of its four neighbors in a single time step. It can receive up to four messages from its four neighbors and send up to four messages, one to each neighbor, in one routing step. This makes it possible for any pair of adjacent processors to perform a compare-and-exchange operation in one step. This model is sometimes extended with wrap-around connections.

1.4.2 Routing Algorithms

This section first reviews routing algorithms for the ILLIAC model. Deterministic routing algorithms on distributed control models are reviewed next; they are the main interest of my research. Some randomized routing algorithms are also reviewed and compared with deterministic algorithms.

Algorithms for the ILLIAC Model

Because in the ILLIAC model the messages can be sent only in the same direction, the lower bound in the worst-case performance occurs when any message must travel from one corner processor to the one in the opposite corner, while another message must travel from the processor in the opposite corner to the one in the first corner. This requires $4n-4$ routing steps, since two messages must travel completely in the opposite directions and these

[4]In many references, this model is called a *MIMD model* in contrast to the (ILLIAC) SIMD model.

messages cannot travel at the same time due to the limited communication control.

The earliest work addressed routing algorithms on an ILLIAC model [31]. Even though the work was based on a centralized control model in which a controller could have complete information about routing requests in the processor elements, research addressed only algorithms for limited kinds of permutations such as perfect shuffle, bit reversal, p-ordered vector [44], and some permutations which can be represented by mathematical formulas. For an arbitrary (irregular or random pattern) permutation on an ILLIAC model, researchers suggested using sorting algorithms.

Orcutt [31] describes routing algorithms that perform perfect shuffle and bit reversal in $O(n)$ steps on the ILLIAC model with wrap-around. His algorithm can perform a perfect shuffle with $4n - 4$ unit routing step using two buffers per processor.

Nassimi and Sahni [30] suggest a more general algorithm for data routing of BPC permutations on the ILLIAC model. Their algorithm takes at most $4n - 4$ unit routing steps for any BPC permutation using three buffers per processor. They also describe an algorithm to prepare the control sequence of the SIMD controller for any BPC permutation in $O(\log^2 N)$ computing time. Note that their algorithm does not use wrap-around connections, which were assumed in Orcutt's algorithm [31]. Moreover, their algorithm performs a perfect shuffle in only $2n - 2$ steps.

The model of Flanders [8] is almost the same as the ILLIAC model except that each processor's memory is one bit wide. BPC permutations, which can be described by "bit mapping vectors", are performed by a method similar to Nassimi and Sahni [30] on the ICL Distributed Array Processor (DAP).

Raghavendra and Kumar [36] describe a three phase algorithm for arbitrary permutations on the ILLIAC model with wrap-around in $3(n-1)$ steps. In the first phase, all data are shuffled along the rows in such a way that no two messages in the same column have the same destination row address. Then, in the second and third phases, one can route data along a column and a row without piling up more than two messages in a processor, and routing of any permutation can be done in $(2n - 2)$ steps for the second and third phases. So the first $(n-1)$ steps are used to shuffle around data in such a way that at most two messages in a processor exist during the next phases. The way to shuffle data in the first phase is shown for BPC permutations. But they do not show clearly how to find a control sequence of the central controller for an arbitrary permutation. Their algorithm is actually simulating a $(\sqrt{N}, \sqrt{N}, \sqrt{N})$ Clos network [24]. (For an arbitrary permutation, it is not clear how to simulate a Clos network with distributed control.)

In Nassimi's study [29], broadcasting on the ILLIAC model is done by sorting destination addresses and distributing them with data. A partial permutation is treated as a special case of broadcasting. The messages are first packed in the processors with smaller row major index and sorted by the destination addresses and then distributed over the mesh. The time complexity of broadcasting is $O(n)$ routing steps, which include a sorting phase and a distributing phase that takes at least $4n$ steps.

In networks with a global control, the controller can use global information to prepare a routing sequence. But access to global information does not in itself make it easy to find an efficient routing sequence.

Buffer requirements of routing algorithms for the ILLIAC model are very small; the number of buffers needed is typically three. Most routing algorithms for the ILLIAC model are restricted to special classes of permutations and take advantage of information about the pattern of permutations to precalculate the control sequence. If a permutation does not belong to a "nice" regular pattern, it may be difficult to find control sequences for routing that are shorter than those for sorting. Thus, sorting (based on destination addresses) seems the best way to handle arbitrary permutations as well as partial permutations and broadcasting.

Algorithms for the Distributed Control Model

The distributed control model allows simultaneous exchanges between any adjacent pair of processors; each processor may send and receive up to four messages at each step. The worst-case performance lower bound clearly occurs when some message must travel from one corner processor to the opposite corner; this requires at least $2n-2$ routing steps. This bound is easily achieved if there is sufficient storage space in the processors.

Consider the following two phase routing method to solve a permutation problem on an n by n mesh. The destination of a message is represented by a row-column address pair. In the first phase, all messages move along the rows to their destination columns, where they are stored in the appropriate processor. In the second phase, messages move along columns to their destination rows (and their destination addresses). This algorithm requires at most $2n-2$ steps for messages to reach their destinations if there is no contention for links among messages. In addition, in the first phase, as messages move along rows, no contention can arise because there is initially only one message per processor and they are pipelined along each row.

When messages reach their destination column, however, they may share the processor with up to $n-1$ other messages and contend with up to $n-2$ messages for a link along the column. Thus, during the second phase, a message could be delayed up to $n-2$ steps; a permutation may therefore require up to $3n-3$ steps if the message that has the farthest to travel is delayed the most. However, by giving higher priority to messages that have farther to go in the second phase, each message is guaranteed to reach its destination row within $n-1$ steps after the second phase begins [14]. In summary, a message will reach its column destination in at most $n-1$ steps in the first phase and its row destination in at most $n-1$ steps in the second phase, giving a total of $2n-2$ steps.

This routing method has a simple control structure and is classified as *oblivious* because the path of each message is determined entirely by the source and destination addresses and is not affected by the paths of other messages. Moreover, the number of steps required is optimal. But the number of buffers required is n; this bound is attained when all messages in a row must travel to the same column. Overcoming this storage requirement without drastically increasing the time required is the goal of most of the work on routing algorithms for meshes.

To reduce storage requirements, Kunde [18] sorts subblocks of the mesh before the actual routing phase. He divides the n by n mesh into ϵ by ϵ submeshes of n/ϵ by n/ϵ processors. For each block of n/ϵ by n/ϵ processors, he sorts messages in column-major order according to each message's destination column address. This spreads out vertically those messages in each block which have the same destination column. Then he applies the same routing method as described in previous paragraphs, first moving messages along rows and then along columns. The number of buffers required by each processor is reduced to 2ϵ. Permutations require $2n + O(n/\epsilon)$ steps if the sorting algorithm used for the blocks is linear in the input size; it takes $O(n/\epsilon)$ steps to sort n/ϵ by n/ϵ meshes and $2n-2$ steps to move messages along rows and columns.

Typical values for ϵ are $\log n$, $n^{1/4}$, or a constant. Smaller values of ϵ decrease the number of buffers required, but increase the number of steps for sorting the submeshes, making the total number of routing steps larger. If a constant is chosen for ϵ, the asymptotic time bound for this routing algorithm is no longer $2n$, because the overhead of sorting is linear in n. For example, if each processor has 8 buffers, then ϵ is 4, and the algorithm begins by sorting $n/4$ by $n/4$ subblocks. If a sorting algorithm is used which takes $3n+O(n^{3/4})$ steps for an n by n mesh, Kunde's algorithm takes $2.75n + O(n^{3/4})$ steps for routing on n by n mesh. If each processor has only four buffers, it will take

A 100 by 100 mesh is divided into 16 subblocks. ($\epsilon = 4$.) The hatched areas represent messages to the same column after sorting. The hatched area of the first, second and the fourth blocks have a single message. The third block has 97 messages in four columns. Therefore, after the first phase, a processor in the destination column must hold at least seven messages destined for its row, plus a transient message during row movement.

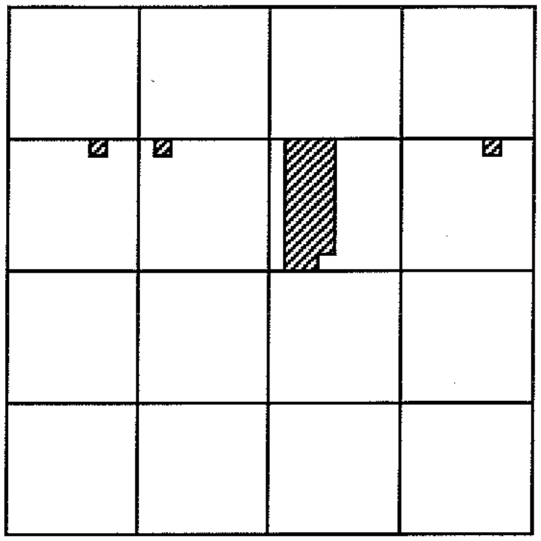


Figure 1.4: Buffer requirements of Kunde's algorithm

$3.5n + O(n^{3/4})$ steps. For a practical value of n (say, 500), a sorting algorithm of $3n + O(n^{3/4})$ steps takes much more than $3n$ steps due to the lower order term [41, 39]. If we apply a sorting algorithm that requires $7n - 7$ steps [20], Kunde's algorithm will take $3.75n$ steps with eight buffers, or $5.5n$ steps with four buffers.

Kunde's algorithm has two properties that are important to our results. First, to keep the time bound $2n + o(n)$, as the size of the mesh increases, his routing algorithm requires the number of buffers per processor to increase without bound. Second, it is not clear that his algorithm can be extended to handle broadcasting (one-to-many) problems.

Leighton et al. [23] improves Kunde's $2n + O(n/\epsilon)$ steps to $2n - 2$ steps by treating specially messages that travel from an m by m corner of the mesh to the corresponding area in the opposite corner. (He calls these messages "critical" messages.) He points out that critical messages must move up to $2n - 2$ steps while other ("ordinary") messages must move no more than $2n - 2 - m$ steps if the size of the corner is m by m. Leighton's algorithm moves the critical messages without sorting. The ordinary messages are sorted, but since the total time required is the maximal distance to be moved plus the time required for sorting, if m is greater than the number of steps required to sort submeshes, then it takes at most $2n - 2$ steps for ordinary messages to reach their destination if they are not held up by the critical messages.

Leighton shows that the movement of the critical and ordinary messages can occur without contention by moving ordinary messages horizontally while critical messages are moving vertically, and vice versa. Moreover, by choosing ϵ to be a small fraction of n, the buffer requirement can be reduced to a constant and the number of steps needed to sort subblocks to less than m, the size of the corner containing the critical messages. Thus, the $O(n/\epsilon)$ sorting time for sorting the submeshes is absorbed into the $2n-2$ distance bound of the critical messages.

The improvement of Leighton's algorithm comes from increased utilization of channel capacity by moving one group of messages vertically while another group moves horizontally. His algorithm is optimal in routing time and needs only a constant number of buffers per processor for a sufficiently large mesh. But the result does not seem to apply for mesh sizes that are now practical, since apparently the number of buffers will be in the hundreds. Moreover, the control structure of the algorithm is fairly complicated, and for large meshes the algorithm must be recursively applied to the m by m submeshes of critical messages.

Kunde [19] described a variant of his previous algorithm [18] which is appropriate when the time to transmit a message between processors is proportional to the length of the message. The variant splits each message into two smaller ones, one belonging to a 'first group,' and the other to a 'second group.' Thus, each processor will have two messages, and the set of messages is divided into two groups. In Kunde's previous routing algorithm, only links along one axis (horizontal or vertical) are active at a given step. The variant algorithm speeds up communication by using the two sets of links concurrently; while routing one group of messages through vertical(horizontal) links, the processors route the other group of messages through horizontal(vertical) links. Since we are assuming that the time to transmit a message is proportional to its length, the time for each routing step is half what it would be if the messages were not split, and overall performance is improved by a factor of two. Storage requirements remain the same because the variant requires that each processor store twice as many messages of half the size.

Randomized Algorithms for the Distributed Control Model

The space-efficient distributed control model deterministic algorithms we have described are based on sorting. Because of the sorting phase, they are not oblivious; that is, the paths of messages are determined not only by source and destination addresses but also by the paths of other messages. Although the channel capacity is often largely wasted, congestion can arise because many packets try to go through a small portion of the mesh. This could be alleviated by an algorithm that would initially scatter messages evenly in the mesh so that no big congestion would occur in any particular area. The following randomized routing algorithms depend upon this idea. Some approaches using randomization are briefly reviewed.

Valiant and Brebner [48] describe a distributed control routing strategy on parallel processors with a small number of connections. To reduce the conflicts on links and the buffer requirement in a network, data in a processor are sent to randomly selected intermediate processors, and then routed through deterministic paths determined by destination and current position. Each data path is determined only by its source and destination addresses, even though its path is selected randomly from several candidates. With this algorithm, an arbitrary permutation takes $3n + O(n^{3/4})$ steps with high probability, and $O(\log n)$ buffers are necessary due to the congestion on links.

Rajasekaran and Tsantilas [37] found a $2n + O(\log n)$ randomized algorithm similar to Valiant's algorithm [48] by changing the queuing discipline in the processors. The maximum buffer requirement is $O(\log n)$. The improvement comes from giving higher priority to the messages which will travel farther, and treating messages which travel from one corner area to the opposite corner in a special way. Krizanc et al. [14] improve the randomized algorithm of Rajasekaran and Tsantilas [37]. Their improved randomized (non-oblivious) routing algorithm realizes any permutation within $2n + O(\log n)$ routing steps and with a large enough constant number of buffers.

Randomized routing algorithms are fully implementable in distributed manner because the decision of a processor depends only on the messages in it and a random variable. The time complexities of randomized routing algorithms are $2n + O(\log n)$. But the number of buffers required is either monotonic with the size of mesh or a large constant (in a sense that it is not specified exactly with a small number).

1.4.3 Sorting Algorithms

Our interest in this thesis is routing algorithms for full permutations, partial permutations and broadcasting, but the topic of sorting on a mesh is related. Recall that we define a sorting problem as a problem that initially has one message in each processor; sorting will permute the set of messages according to keys rather than a given destination address. A permutation problem inherently has more information initially than a sorting problem, since each message of a permutation problem has a specified destination. Orcutt [31] pointed out that a sorting algorithm can use these destination addresses to perform a full permutation. A sorting algorithm, however, must direct each message to a destination that is determined by the other messages in the mesh. Thus, although a sorting algorithm can solve a full permutation problem, a permutation algorithm will not generally suffice to solve a sorting problem. One consequence is that any upper bound on sorting is also an upper bound on permutations.

Sorting can also be used in peripheral ways for routing problems. For one class of broadcasting problems on the ILLIAC model, sorting is used to rearrange messages so that no congestion can arise during the distribution phase in the routing algorithm.

Storage requirements for all the sorting algorithms described in this section (unless we note otherwise) are three buffers per processor on the ILLIAC model and two buffers[5] per processor for the distributed control model.

Sorting on the ILLIAC model

Orcutt [31] describes an $O(n \log n)$ bitonic sorting algorithm in row-major order on the ILLIAC model with wrap-around. Thompson [45] gives an "s^2-way merge sort" without wrap-around, which takes $6n + O(n^{\frac{2}{3}} \log n)$ routing steps to sort n^2 items into snake-like row-major order. He shows that bitonic sorting can be done in $14n$ routing steps with a shuffled row-major indexing scheme.

Nassimi [28] gives a bitonic sorting algorithm on the ILLIAC model without wrap-around. His algorithm takes $14n$ routing steps to merge subblocks of a bitonic sequence into row-major order recursively. By modifying comparison directions, it can sort an n by n mesh array in snake-like row-major order with the same number of steps. Note that $14n$ is the same number of steps as Thompson's bitonic sorting algorithm which sorts into shuffled row-major order.

Kumar [15] uses Batcher's odd-even merge sort algorithm as the basis of his sorting method on the ILLIAC model with wrap-around. His algorithm takes $11n + O(\log^2 n)$ routing steps to sort n^2 elements in row-major order. Its control structure is recursive merging by doubling the size of subblocks. He shows that the wrap-around connections can be eliminated with some extra buffers in the last row.

[5]Each processor needs two buffers to do a compare-and-exchange operation between two processors in a single communication step

Sorting on the Extended-ILLIAC model

Ma [25] shows that an n by n array can be sorted into blocked snake-like row-major order with $4n + O(n^{\frac{3}{4}} \log n)$ routing step on the extended-ILLIAC model with $n^{1/4}$ buffers in each processor to accommodate transient messages. He also describes a recursive sorting algorithm based on dividing a square mesh into four quarters, sorting them in snake-like row major order and merging the results. The algorithm requires $8n$ steps and the same number of buffers, but it has a simpler control structure.

Sorting on the Distributed Control Model

Lang's sorting algorithm [20] merges four sorted subblocks (arranged as a two by two array of submeshes) in snake-like row-major order using $7n$ compare-and-exchange steps[6]. Its control structure is relatively simple, so it can be implemented on systolic arrays.

Schnorr's algorithm [41] takes $3n + O(n^{3/4})$ compare-and-exchange steps to sort an n by n mesh in snake-like row-major order. Subblocks are sorted in sublinear time and merged in $3n + o(n)$ compare-and-exchange steps. He shows his algorithm is asymptotically optimal, but the low order terms of complexity are too big for meshes of practical size, and the control structure is complicated.

Sado [38], on the other hand, describes a "pseudo-merge" sorting algorithm which takes $6.5n + O(\log n)$ compare-and-exchange steps to sort n^2 items in snake-like row-major order. Its complexity has small low order terms and its control structure is relatively simple. He also describes an algorithm with a relatively complex control structure that takes $5.5n + O(\sqrt{n} \log n)$ compare-and-exchange steps.

Scherson [40] describes a "sheer-sort" algorithm which sorts an n by n mesh in snake-like row-major order by alternating row sort and column sort. It has a very simple control structure but has time complexity of $O(n \log n)$. Based on the sheer sort algorithm, he proposes a recursive sorting algorithm which takes $6n$ compare-and-exchange steps to merge four sorted subblocks (arranged as a two by two array) into snake-like row-major order. This is very similar to Sado's algorithm [38]. By sacrificing simplicity of control structure, he constructs a $3n + O(n^{\frac{3}{4}} \log n)$ sorting algorithm in a blocked snake-like row-major ordering scheme.

For the distributed control model, Kunde pointed out that the best way

[6]A single compare-and-exchange step takes two routing steps on the ILLIAC model.

to solve routing problems when only a small constant number of buffers are available (one or two) is the $3n+o(n)$ sorting algorithm for a full permutation [18].

For the distributed control model, lower bounds for sorting have been shown to be $3n-3-\lceil\sqrt{2n}-1/2\rceil$ [16, 41]. Some sorting algorithms approach these bounds asymptotically: Schnorr's algorithm takes $3n+O(n^{3/4})$ steps, and Scherson's algorithm takes $3n+O(n^{3/4}\log n)$ steps. But in all these results, sublinear terms dominate the complexity for meshes of practical size. Moreover, the control structure of the algorithms is complicated, making them impractical for a mesh with processors too small to hold a substantial program.

Chapter 2

Move and Smooth Algorithms

This chapter describes a new class of routing algorithms on a mesh, which we call *move and smooth* algorithms. The chapter consists of two sections.

Section 2.1 describes this class of algorithms. Although we will only treat a few of these algorithms in detail, the form makes the design of other algorithms straightforward.

The formal problem treated in Section 2.2 is finding the time complexity of message movement in a one dimensional array. This result is crucial to the analysis of the time complexity of all the move and smooth algorithms; the result obtained in Section 2.2 will be referenced repeatedly in subsequent chapters.

2.1 Class of Move and Smooth Algorithms

Although move and smooth algorithms can be applied to k-dimensional rectangular meshes, for this initial description of the algorithms, we will assume that the processor array is a square mesh in two dimensions.

Move and smooth algorithms are recursive, with each recursive stage consisting of two steps. Initially, each message is in a processor, with no more than one message per processor. Each message knows the address of its destination processor. The mesh is partitioned into a collection of disjoint regions $R_1, R_2, \cdots R_b$. The *move* step moves each message in the array to the region R_i that contains its destination. The move step may, however, put more than one message into some processors. The *smooth* step redistributes the set of messages in each region R_i so that each processor once again contains at most a single message.

Move and smooth algorithms partition the mesh into b disjoint contiguous regions. Although the regions are not necessarily the same in size and shape, move and smooth algorithms are most elegant when the regions are congruent, and we will direct most of our attention to cases in which our algorithms partition the mesh into b disjoint congruent regions. Thus, for this overview, if a partitioning divides a mesh (or submesh) into b disjoint regions $R_1, R_2, \cdots R_b$, then for all i, j, $1 \leq i, j \leq b$, R_i and R_j have the same size and shape. During the *move* step, each message moves from its current region to its target region (that is, the region that contains its destination) along a path that is assured to be conflict free. Moreover, if the number of regions in the partition is b, our algorithms guarantee that no processor will hold more that $b+1$ messages during the move step, or more than b messages at the end of the move step. However, in order to apply the algorithm recursively, the precondition of 'at most one message per processor' must be re-established.

We will be most concerned with two specific algorithms which we call H (for 'halves') and Q (for 'quarters'). The move steps of the algorithm Q can be described as follows:

> Algorithm Q begins with a partitioning of the n by n square array of processors into four quadrants of equal size. If a message is initially in the correct quadrant (that is, the quadrant that contains its destination address), it does not move during the move step. Otherwise, the move step takes each message from its current location to the processor which has the same relative position in the correct quadrant. This movement is accomplished by moving 0 or $n/2$ steps along a column of the mesh, and then 0 or $n/2$ steps along a row. All messages start moving at the same time and stop as soon as they reach the correct position in the destination quadrant. During the move step, a processor may contain up to five messages, including one that did not move, and four transients. At the end of the move step, a processor contains at most four messages, these being three messages from other regions and one message of its own.

The purpose of the *smooth* step of each recursive stage is to distribute the messages of each region so that each processor contains at most one message without violating the constraint that each message is in the region that contains its destination. Completion of the smooth step re-establishes the precondition of the recursive algorithm that each message is in its target region, and that each processor contains at most one message. Then the algorithm recurs; each region is partitioned into disjoint subregions and each message is sent to the proper subregion (and generally closer to its destination). The algorithm terminates when each region has only one processor.

A requirement of each algorithm is that every processor know its row-column indices for the current region to which the algorithm is being applied. These values can easily be computed by each processor from a knowledge of its row and column index in the mesh and the current level of recursion. Wrap-around connections are not used by any of the algorithms.

2.2 Message Movement in Linear Array

Each of the algorithms we discuss is based on a partitioning into b rectangular regions. Each move step on a two-dimensional mesh transforms the original problem into a set of subproblems that must be smoothed. Each smoothing problem consists of a rectangular region with r rows (indexed $0, \ldots r-1$) and c columns (indexed $0, \ldots c-1$) such that

initially, the region contains no more than rc messages, and
each processor contains no more than b messages.

Smoothing is accomplished in two phases. First, a *counting* (preprocessing) phase informs each processor of the initial configuration of messages. This information enables each processor to determine its roll in achieving the final configuration. The second *distribution* phase entails the movement of messages. Distribution is accomplished by first moving messages along rows, and then along columns. *Row movement* rearranges messages within each row so that no column of the region contains more than r messages. *Column movement* moves messages within each column so that no processor contains more than one message. Both row movement and column movement are accomplished without requiring that any processor hold more than b messages at any time.

It is not necessary to understand the details of one dimensional row and column movement described in this section to understand the behavior of the algorithms described in Chapters 3 and 4. However, the results of this section are used to analyze the complexities of algorithms in later chapters.

2.2.1 Row Movement

Consider a one dimensional array with c processors indexed from 0 to $c-1$ from left to right. Each processor has b message buffers and is connected to adjacent processors only by unidirectional channels with the capacity of a single message. Adjacent processors are connected with two channels, one from left to right and the other from right to left. Although the two channels between adjacent processors can be used concurrently, we will see that they are not used concurrently in row movement.

Each row movement problem is of the following form: Initially, there are N messages in the array where $1 \leq N \leq bc$, and each processor contains no more than b messages. The initial arrangement of messages in the array is called the initial configuration. Messages are to be moved among processors in the array to a specified final configuration, where a final configuration will have the following properties:

- each processor has at least $\lfloor N/c \rfloor$ and at most $\lceil N/c \rceil$ messages, and
- the processors with $\lceil N/c \rceil$ messages will be contiguous. (For the purposes of row movement, the leftmost processor and rightmost processor are considered to be contiguous.)

An initial-final configuration pair specifies a row movement problem. Note that for row (and column) movement, messages are indistinguishable; there is no requirement that specific messages go to specific processors.

Prior to actual movement of messages among processors in row movement, there is a preprocessing phase that enables each processor to determine

1. how many messages it contains initially,
2. the total number of messages contained in processors to its left and the total number of messages contained in processors to its right in the initial configuration,
3. how many messages it will contain finally, and
4. the total number of messages contained in processors to its left and the total number of messages contained in processors to its right in the final configuration.

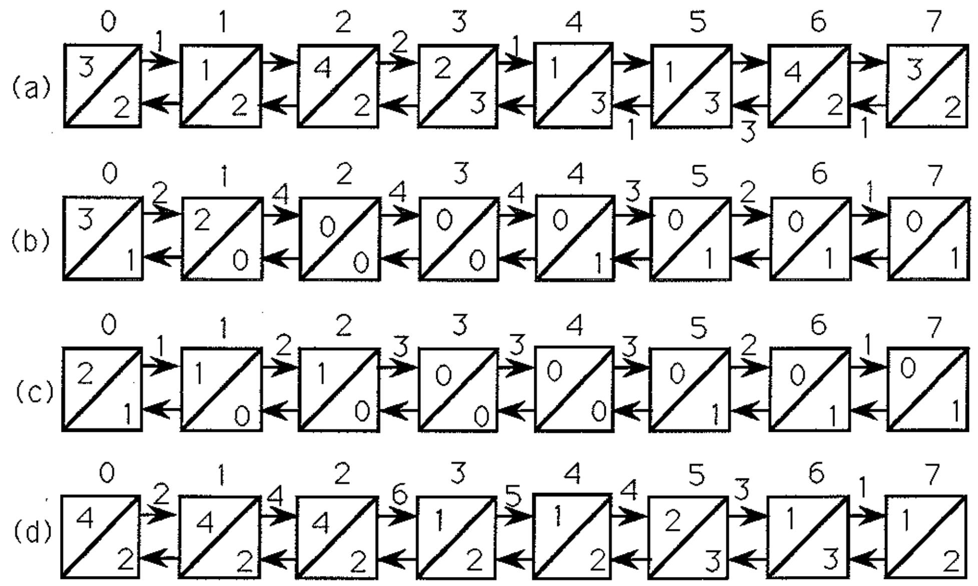
Because messages are indistinguishable, there is never a need for adjacent processors to exchange messages, simultaneously or otherwise, during row movement; an exchange of messages accomplishes nothing toward achieving a final configuration. Thus, at most one channel between two adjacent processors will be used to solve any row movement problem. A link in which only the right channel has nonzero flow will be said to have a *right flow*, and a link in which only the left channel has nonzero flow will be said to have a *left flow.* (Some links may have neither right nor left flow.) Thus the net action of any processor during row movement can be characterized as sending or receiving some number of messages from its left neighbor and sending or receiving some number of messages to its right neighbor. (The net action of an end processor can be characterized the same way, except that the number of messages sent to or received from the non-existent neighbor is zero.)

On the basis of its initial information, each processor can determine how many messages it must send to or receive from its left neighbor and how many messages it must send to or receive from its right neighbor. Let the *flow number* of a channel be the total number of messages to flow through the channel between two adjacent processors during row movement. By comparing the number of messages to its left in the initial and final configurations, each processor can compute flow numbers for the channels of the links to its neighbors before messages begin to move, as illustrated by Figure 2.1 .

For any row movement problem, the set of flow values, taken together, constitutes a solution. This solution is unique, since it represents a net flow of messages, but its implementation is not, since messages could be sent back and forth across the same link between adjacent processors. The simplest implementation is for each processor to send messages as soon as possible to satisfy the flow calculated for each of its outgoing channels. (Note that the flow over a channel may be interrupted if a processor exhausts its messages before the outgoing flow has been satisfied. In this case, the flow is interrupted until the processor receives additional messages from its neighbor.) We will use this mechanism to solve row movement problems, and refer to the resulting movement of messages as *row movement.* We wish to characterize the worst case time complexity of the row movement problem, that is, the maximum number of time steps required to implement a solution by moving messages among processors. We will do so by first showing that row movement (as defined above) is an optimal solution to any row movement problem and then exhibiting and analyzing configurations that maximize the number of steps of row movement.

We first make the following observations about movement of messages along an array.

1. The movement of messages along an array during row movement can be viewed as a collection of right flow segments, where each segment is a series of contiguous processors connected by links with nonzero right flows, left flow segments, where each segment is a series of contiguous processors connected by links with nonzero left flows, and zero flow



Four examples, each showing initial and final configurations of an array with 8 processors. Processors are represented with squares indexed $0 \ldots 7$; indices appear above the squares. Channels between processors are represented with arrows, and the direction of arrows represents the direction of channels. The number of messages initially contained in each processor is shown in the square as an integer above the diagonal; the tuple of all these integers specifies the initial configuration. The number of messages to be contained in each processor after row movement is shown as an integer in the square below the diagonal; this set of integers specifies the final configuration. Integers on arrows are the flow numbers represented by the corresponding channel; no integer indicates a flow of zero. Note that at most one arrow between adjacent processors has a non-zero flow. In the array of Figure 2.1.a, there are 19 messages initially. The final number of messages in each processor is either 2 ($\lfloor 19/8 \rfloor$) or 3 ($\lceil 19/9 \rceil$) and the processors 3, 4 and 5, which contain 3 messages each, are contiguous in the final configuration. In Figure 2.1.a, the processors 0 and 1 form a maximal right flow segment, as do processors 2, 3, and 4. Processors 4, 5, 6, and 7 make up a maximal left flow segment. Note that processor 4 belongs to both a left flow segment and a right flow segment that are separated by processor 4 itself. In Figure 2.1.b, there are only 5 messages in the array, and the four rightmost and one leftmost processors will receive a message in the final configuration. The whole array is a right flow segment. The time required to change from the initial to final configuration is 6 steps; note that the maximum value of flow in the array is 4.

Figure 2.1: Examples of row movement problems

segments. (A processor with flows in different directions on its left and right links belongs to two segments which are separated at that processor.) A segment is called a maximal left (right, zero) flow segment if it is not a proper subset of another left (right, zero) flow segment.

2. Messages moving in different maximal segments do not interfere with or affect one another.

Because maximal segments are independent problems, the time required for row movement in an array is the longest time required for message movement within some segment of the array. Thus the worst case problems will be ones whose solution consists of a single segment.

The time required for row movement is affected by two factors, distance and contention. A distance constraint results from the necessity of some message travelling from one processor to another in the array. A contention constraint results from the inability of a channel to carry more than one message at a time. In Figure 2.1.c, the time complexity is determined by distance; there is no contention for channels, while in Figure 2.1.d, the complexity is determined by contention. Figure 2.1.b exhibits a complexity determined by distance; although there is contention, the complexity of the problem would not be reduced if the channels could carry more than a single message concurrently.

Because the worst case behavior for row movement will occur when the solution consists of a single segment, in the following, we will consider only solutions in which the flow over all links is nonzero and to the right. The following lemma characterizes worst case performance for a class of problems in which the distance to be travelled is the limiting factor.

Definition 1 *We define the function $T(N, b, c)$ as the maximum time required (to implement a solution) for a row movement when an array of c processors, each of which has b buffers, contains N messages.*

Our first row movement lemma places a time bound on row movement for the case when the number of messages is less than the number of processors.

Lemma 2.1 *If* $0 < N < c$, *then* $T(N, b, c) \leq c - 1$.

Proof: Since N, the number of messages in an array, is less than c, the number of processors in the array, any final configuration will have a single message in N processors and no messages in the remaining $c - N$ processors. Recall that the set of final configurations considers processor 0 to be adjacent to processor $c-1$ (although there is no wrap-around connection). Moreover, our restriction to those solutions in which all flow is nonzero and to the right implies that in the final configuration, the rightmost $k \leq N$ contiguous processors of the array will contain messages, as will the leftmost $N-k$ contiguous processors. (The number k should be greater than 0, but $N-k$ may be 0. Thus the very rightmost processor of the array will contain a message; the leftmost processors may or may not contain messages.) Worst case performance will occur when the initial configuration has all N messages packed into the leftmost $\lceil N/b \rceil$ processors. The number of time steps required for row movement in this case is $c - \lfloor N/b \rfloor$, which is equal to the distance that the rightmost message in the initial configuration travels to the rightmost processor of the array. The maximum value of $c - \lfloor N/b \rfloor$ occurs when $N \leq b$, which gives the value $c-1$. □

We now turn to the case when $N \geq c$. The following lemma establishes that in this case, every solution has a *busy channel* in the sense that, if the solution requires t time steps, then t messages will be sent over the busy channel.

Recall that we are considering only right flow segments, that is, only solutions in which only the channels carrying messages from left to right are utilized. We assume that the solution is for an array of c processors indexed from 0 to $c-1$. We begin by indexing the channels (carrying messages from left to right) from 1 to $c-1$; thus the channel between processor $k-1$ and processor k has index k.

Lemma 2.2 *Consider an array which has $N \geq c$ messages. If the maximum flow over a single channel of a solution of a row movement problem is equal to t, then row movement requires exactly t time steps.*

Proof: Without loss of generality, we assume that the solution of a row movement problem is a single right flow segment, and channel k carries t messages. The time required for row movement cannot be less than t, because the flow across channel k is t and only one message can flow across a channel at each step.

Suppose, as an assumption to be proved false, that the row movement takes t' time steps where $t' > t$. Then there is a channel x which carries a message at time t'. Let the flow number of link x be f. Then $f \leq t$, because t is the maximum flow across any channel of the solution. Therefore $f < t'$, from which it follows that channel x is idle at some time. Then we determine two times, $t_1 \leq t_2 < t'$ such that $f_1 < f$ messages flow through channel x during time 0 through t_1, no messages flow through channel x during time

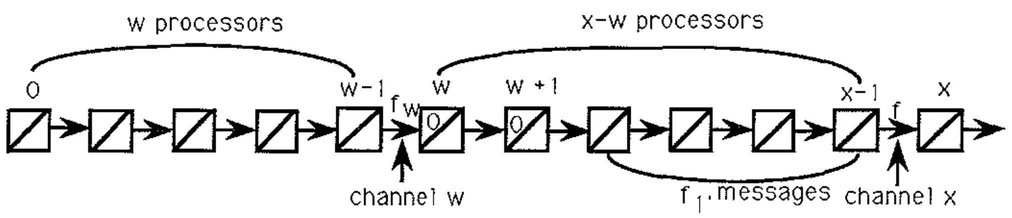


Figure 2.2: The array in the proof of Lemma 2.2

$t_1 + 1$ through t_2, and $f - f_1$ messages flow through channel x during time $t_2 + 1$ through t' and $t' - t_2 = f - f_1$. Note that t_1 may equal 0, or channel x may have idle times during the first t_1 time steps. But $t_2 - t_1 > 0$ (and no messages flow over channel x during that interval), and channel x is never idle during the interval $t' - t_2$.

Let channel $w < x$ be the leftmost channel such that the processors from w through $x - 1$ contain exactly f_1 messages initially. (See Figure 2.2.) Note that such a channel must exist because there is a gap of length $t_2 - t_1$ in the flow of messages after the first f_1 messages and such a gap can occur only if there is a sequence of processors that initially are empty. Thus, the processors indexed from w to $w + (t_2 - t_1)$ must initially be empty.

Now consider the flow over the channel w. The processor $w - 1$ initially contains messages, since w was required to be the leftmost channel such that the processors from w to $x - 1$ contain exactly f_1 messages. Let C_F be the number of messages in processors w through $x - 1$ in the final configuration. Our specification of w requires that the processors from w to $x - 1$ contain exactly f_1 messages. Therefore, the following relations hold:

$$C_F = f_1 + f_w - f$$

That is, the final contents of the processors w through $x - 1$ is equal to the initial contents plus the flow in minus the flow out. Additionally, t_1 and t_2 were chosen so that

$$t' - t_2 = f - f_1$$

From our specification of t_2, f_1 messages flow through channel x during time 0 through t_1, no messages flow through channel x during time $t_1 + 1$ through t_2, and a message flows through channel x at $t_2 + 1$. Hence the processor that holds the message that reaches x at time $t_2 + 1$ must be $w - 1$; that is,

$$x - w = t_2$$

Finally, since $N \geq c$, it follows that the final configuration will have at least one message in each processor, that is,

$$C_F \geq x - w$$

We rearrange the first equation, getting

$$f_w = C_F - f_1 + f$$

and substitute from the others, giving

$$\begin{aligned} f_w &\geq x - w - f_1 + f \\ f_w &\geq t_2 + t' - t_2 = t' \end{aligned}$$

which contradicts our specification that the largest flow number was t, where $t' > t$, and establishes the theorem. □

Corollary 2.1 *The time required for row movement is optimal.*

Proof: By definition, no solution requires that fewer messages be passed between processors. Since the time required is exactly the number of messages passed by the processor passing the most messages, no solution can be implemented in less time. □

We now wish to find a time bound for row movement. We will use the preceding lemma to define a worst-case configuration and then find the time bound for that configuration.

Lemma 2.3 *If* $c \leq N \leq bc$ *and* b *is even, then*

$$T(N, b, c) \leq \frac{b(b+2)c}{4(b+1)}$$

If $c \leq N \leq bc$ *and* $b > 1$ *is odd, then*

$$T(N, b, c) \leq \frac{(b+1)c}{4}$$

Proof: We showed in the proof of Lemma 2.2 that for any row movement problem, in which $N \geq c$, there is a channel which is busy all the time during row movement. Assume that channel k is one of the busiest channels for such a problem; that is, channel k carries t messages in a solution that requires t time steps. As before, we restrict our consideration to solutions (row movements) that consist of a single right flow segment. To maximize the flow over channel k, the initial configuration must have as few messages as possible to the right of channel k, and the final configuration must have as many as possible to the right of channel k. The form of the initial configuration is not constrained, but the final configuration is required to have the messages evenly distributed among the processors, with the number of messages in two processors differing by no more than one. Thus the maximum flow of messages through any channel k can be achieved as follows:

> Initially, all processors to the left of k are filled and all processors to the right of k are empty. (Thus each processor to the left of channel k contains b messages initially, and there is a total of bk messages to the left of channel k).
>
> In the final configuration, each processor to the right of channel k contains $f+1$ messages, and each processor to the left of channel k contains f messages. (Thus each processor to the right of channel k contains one more message than any processor to the left of channel k.)

The flow over channel k is then

$$F(k) = (f+1)(c-k) \tag{2.1}$$

We can now find the time bound for row movement problems by finding the maximum value of $F(k)$.

Because the total number of messages in the final configuration is the same as the number of messages to the left of channel k in the initial configuration, the following holds:

$$bk = fk + (f+1)(c-k) \tag{2.2}$$

Then the maximum value $F(k)$ for some value k is the maximum time bound imposed by the contention on channel k, because one message takes one step and messages will flow continuously.
From Eq. 2.2, we get

$$k = \frac{(f+1)c}{b+1} \tag{2.3}$$

By substituting Eq. 2.3 to Eq. 2.1, we get

$$\begin{aligned} F(k) &= (f+1)(c - \frac{(f+1)c}{b+1}) \\ &= \frac{-c}{b+1} f^2 + \frac{c(b-1)}{b+1} f + \frac{cb}{b+1} \end{aligned}$$

Recall that f is an integer, where $1 \le f < b$. If b is even, $F(k)$ has the maximum value

$$\frac{b(b+2)c}{4(b+1)} \text{ when } f = b/2 \text{ or } b/2 - 1$$

The values of k that maximize $F(k)$ are

$$k = \begin{cases} \frac{(b+2)c}{2(b+1)} & \text{when } f = b/2 \\ \frac{bc}{2(b+1)} & \text{when } f = b/2 - 1 \end{cases}$$

(Note that these values are approximately equal to $c/2$, which means that the time bound is largest when channel k is located approximately in the middle of the array. This is intuitively reasonable, since when k/c is small, there cannot be enough messages to the left of the channel k to cause a large flow over the channel, and when k/c is near 1, there is not enough space to the right of channel k to accommodate a large flow.)

The above values of k used to maximize $F(k)$ may not be integers, although k must be a processor index. But for any integer values of b and c, the maximum value of $F(k)$ occurs either at $k = \lceil \frac{(f+1)c}{b+1} \rceil$ or $k = \lfloor \frac{(f+1)c}{b+1} \rfloor$ whether $f = b/2$ or $f = b/2 - 1$. Thus, $\frac{b(b+2)c}{4(b+1)}$ is the upper bound of the number of steps required for row movement when b is even, although the value may not be an integer. Therefore, whenever this value is referenced as the number of steps for routing, we will use it as an upper bound.

If b is odd, $F(k)$ has the maximum value

$$\frac{(b+1)c}{4} \text{ when } f = (b-1)/2 \text{ and } k = c/2$$

□

Theorem 2.1 *When b is even,*

$$T(N, b, c) \le \max(c - 1, \frac{b(b+2)c}{4(b+1)})$$

When $b > 1$ is odd,

$$T(N, b, c) \leq \frac{(b+1)c}{4}$$

Proof: Lemma 1 and Lemma 3 give the theorem immediately. □
Row movement requires at most $c - 1$ time steps for an array of c processors with two buffers, and at most $1.2c$ time steps when each processor has four buffers.

We now consider the number of buffers required during row movements. The following theorem assures us that no extra buffer is necessary during row movement.

Theorem 2.2 *During row movement, the maximum number of messages stored in any processor is no greater than the maximum number stored in some processor for the initial and final configurations.*

Proof: During row movement, the array is divided into maximal segments, and movement of messages in different maximal segments is independent. In each segment, the flow of messages is unidirectional. The number of messages in a processor will only increase when it receives a message over one channel and does not send out a message over the other. Because flows are satisfied as soon as messages are available, it follows that the number of messages in a processor will only increase when either the processor is empty or the flow out of the processor is complete. Thus, whenever the number of messages in a processor increases because there is no outflow, that number will never exceed the number of messages the processor must hold in the final configuration. □

If processors have b buffers, then the initial and final configurations are constrained so that no processor holds more than b messages. The preceding theorem establishes that the implementation of row movement does not increase the buffer requirements of the processors.

2.2.2 Column Movement

We will show in Section 3.3.2 (on page 52) how the first step in smoothing, row movement, rearranges messages within each row so that each column of r processors contains no more than r messages, with at most b messages in any single processor. After row movement is complete, the second step, column movement, rearranges the messages in each column of r processors so that each processor contains at most a single message. These steps together establish the precondition for the move and smooth algorithm, that each message be in the appropriate region of a partition, and that no processor contain more than one message.

Consider a one dimensional array with r processors indexed from 0 to $r-1$ from top to bottom. We assume that r is even and refer to the top and bottom halves of the column. Each processor has b message buffers and is connected to each adjacent processor by two unidirectional channels with the capacity of a single message. (As with row movement, only one channel of each link is used at any time during column movement.)

We consider the following problem:

Initially, there are N messages in a column array where $1 \leq N \leq r$, and each processor contains no more than b messages. We want to rearrange messages among processors in the array so that each processor has at most one message.

As in a row movement, messages are indistinguishable. Based on the preprocessing phase completed prior to row movement, each processor can determine, prior to column movement,

- the number of messages contained in processors above it, if it is a processor in the top half of the array, and
- the number of messages contained in processors below it, if it is a processor in the bottom half of the array.

Processors rearrange messages by first assigning an index to each message and then moving the messages among the processors. Indexing is done as follows:

1. If P is a processor in the top half of the array, and there are u messages in processors above P, and P contains k messages, then the messages in P are assigned indices u, $u+1$, $\ldots$ $u+k-1$.
2. If P is a processor in the bottom half of the array, and there are v messages in processors below P, and P contains k messages, then the messages in P are assigned indices $r-v-1$, $r-v-2$, $\ldots$, $r-v-k$.

Message movement is done using the assigned indices as target addresses within the column. When two messages compete for the same channel, priority is given to the message with the longest distance to travel. Note that there can be no overlap of indices assigned by processors in the top and bottom halves of the array since the total number of messages in the array does not exceed r. Thus, each assigned index will be between 0 and $r-1$, and will be unique.

We now characterize the maximum number of time steps required to rearrange messages among the r processors for any initial configuration. The individual processors follow the same strategy as with row movement, that is, each processor sends out messages as soon as possible until the proper number of messages are above (or below) it.

We observe the following facts:

1. The index (or target address) of any message in processor i is less than the index of any message in processor j, if $i < j$.

2. As with row movement, the processors of the array can be divided naturally into segments, where, within a segment, all messages move in the same direction. As before, some processors may belong to two segments, and some processors may not belong to any segment, because there is no traffic from, through, or to those processors.

We will compute the worst case time steps for column movement in an array by exhibiting an initial configuration which takes the maximum number of time steps. Without loss of generality, we consider only initial configurations of the top half of an array; the same time complexity results apply to the bottom half.

When the top half of an array contains no more than $r/2$ messages, the targets of all the messages will be processors in the top half of the array. Because a message with the farthest target will go first, the maximum number of time steps required to move messages in the top half is $r/2-1$. This worst case occurs when the processor with index $r/2-1$ contains a single message and all other processors in the top half are empty. (Thus, the message must travel from processor $r/2-1$ to processor 0.)

Suppose the top half of an array contains N messages where $r/2 < N \leq r$. The N-th message in the top half is assigned the target address $N-1$. For the purpose of exhibiting the worst case, we can assume that the messages in the entire column all move in the same direction and that all processors are part of a single flow segment.

Recall that messages are indexed and given target addresses prior to moving. Denote by $M(i \rightarrow j)$ a message originating from processor i with target address j. Suppose it takes t steps for message $M(i \rightarrow j)$ to travel from processor i to processor j. Two cases are possible; either $M(i \rightarrow j)$ travels without any delay, in which case $t = j - i$, or $M(i \rightarrow j)$ is delayed by other messages. Suppose $M(i \rightarrow j)$ is delayed by other messages. Then, because of the way the messages are indexed for column movement, $M(i \rightarrow j)$ will be delayed by $M(i' \rightarrow j+1)$, where $i' \geq i$, and $M(i' \rightarrow j+1)$ will travel one step ahead of $M(i \rightarrow j)$ from the time they meet in a processor to the time they reach their target processors. As a result, $M(i \rightarrow j)$ and $M(i' \rightarrow j+1)$ will reach their targets at the same time, and since they started at the same time, we can conclude they will take the same number of steps. By induction, it follows that for any message $M(i \rightarrow j)$ that is delayed and takes t time steps to reach its destination, there is a message $M(i' \rightarrow j')$ which travels without any delay and also takes t time steps to reach its target.

From the above it follows that the worst case time will be required of some message that travels from its initial position to its final position without delay. A message $M(i \rightarrow j)$ that travels downward without delay requires $j-i$ steps to reach its target. The value $j-i$ is maximized when i is smallest and j is largest. Since indexing of messages begins with 0, the origin of $M(i \rightarrow j)$ (that is, processor i) cannot be above processor $\lfloor j/b \rfloor$ because there must be j messages above $M(i \rightarrow j)$, and no processor initially contains more than b messages. It follows that $M(i \rightarrow j)$ takes at most $j - \lfloor j/b \rfloor$ steps to reach processor j. The value of $j - \lfloor j/b \rfloor$ is maximized by substituting $N-1$ for j, giving $N - 1 - \lfloor (N-1)/b \rfloor$. If we consider a downward flow segment of length N in a column array with r processors, the worst case number of steps to distribute messages is $r - 1 - \lfloor (r-1)/b \rfloor$ because the maximum value of N is r. If r is a multiple of b, this simplifies to $(b-1)r/b$. This establishes the following:

Theorem 2.3 *Consider a one-dimensional array of r processors, where each processor has b buffers. Initially, each processor has at most b messages, and there are no more than r messages in the array. If one message can be moved between any two adjacent processors in one time step, then the messages can be rearranged among the processors so that each processor has at most one message in at most $r - 1 - \lfloor (r-1)/b \rfloor$ time steps. If r is a multiple of b, the number of time steps is $(b-1)r/b$.*

Chapter 3

Algorithm Q

The algorithm to be described in this chapter is called Q because it divides square meshes into quarters, which we refer to as *quadrants.* We provide details of this algorithm in Sections 3.2 and 3.3 and calculate the time complexity and buffer requirement in Section 3.4. We describe how Algorithm Q can handle broadcasting problems without additional routing costs in Section 3.5. In Section 3.6, we investigate the extension of Algorithm Q to k^2*-partitioning* of a two dimensional mesh and discuss the difficulties in application to multi-dimensional meshes.

3.1 Overview

The differences among move and smooth algorithms depend on the number and shape of the regions into which a mesh is partitioned. A natural partition of a square mesh would be to divide the mesh into four quadrants, each one a smaller square mesh. Even though Algorithm Q can be generalized to a rectangular mesh, we describe the algorithm for the special case of a mesh of size n by n, where $n = 2^p$.

Algorithm Q is based on successive partitioning of the mesh into four disjoint regions. It involves a move step followed by a smooth step and then recurs on the four regions simultaneously. The move step begins with a partitioning of the square array of processors into four quadrants of equal size. Initially, each processor has at most one message. If a message is initially in the correct quadrant (that is, the quadrant that contains the message's destination address), the message does not travel during the move step. Otherwise, a message travels from its current location to the processor which has the same relative position in the correct quadrant. This movement is accomplished by moving 0 or $n/2$ steps along a column of the mesh, and then 0 or $n/2$ steps along a row. Thus, a message may remain in place, or it may travel $n/2$ steps along a row or a column, or it may travel $n/2$ steps along a column followed by $n/2$ steps along a row. In any case, when the first move step is finished, each message will be in the quadrant that contains its destination processor. All messages start moving at the same time, and they stop as soon as they reach the target position in the correct quadrant. During the move step, a processor may contain up to five messages; one that did not move, and four transients. All messages move concurrently, and since there is no contention for links, the time for the first move step is n.

At the end of a move step, a processor may contain up to four messages: one from before the move step, and three that have been moved there from corresponding processors in the other three quadrants. However, the number of messages in a quadrant does not exceed the number of processors in the quadrant, because each processor is the destination of at most one message. In order to recur, however, we must re-establish the precondition that each processor contains at most one message. (Otherwise, the number of messages in a processor could grow exponentially with the number of recursive steps.) So, the smooth step follows.

The smooth step distributes the messages among the processors. It has two phases, counting and distribution. The details of the smoothing algorithm will be described in Section 3.3. For now, we simply note that smoothing requires less than $1.75n$ steps for an $n/2$ by $n/2$ mesh. This second step ends the first stage of the recursive algorithm, which leaves each message in the quadrant that contains its destination, with at most one message per processor. (Figure 3.1 shows an example of Algorithm Q on an eight by eight mesh.)

The algorithm terminates when each quadrant contains a single processor. Because the move step takes n steps and the smooth step takes $1.75n$ steps, the complexity of the algorithm is given by the following system:

$$\begin{aligned} T(1) &= 0 \\ T(n) &\leq T(n/2) + 2.75n \qquad (\text{for } n = 2^p) \end{aligned}$$

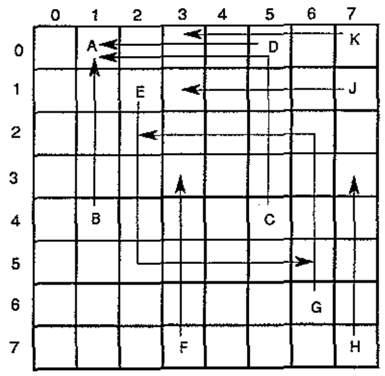
which has the solution $T(n) \leq 5.5n$. In the next sections, we will derive this value and refine the time complexity.

Processors in the 8 by 8 array are addressed by row-column index pairs. Then messages, labelled A through K, have destinations as specified in the Destination Table at the right.

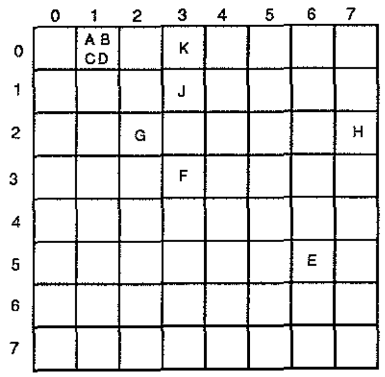
Destination Table

Message	A	B	C	D	E
Destination	2,0	2,1	2,2	2,3	6,7
Message	F	G	H	J	K
Destination	0,1	0,2	0,6	3,0	3,1

In the first phase of Algorithm Q, each message travels to the correct quadrant of the mesh. The figure at the right shows the original position of the messages and the paths followed by them during the first move step. Note that since message A is already in the correct quadrant, it does not change position. Because all messages begin travelling simultaneously, there is no contention for links. At the end of the first move step, processor (0,1) contains four messages.



The result of the first move step, ready for the smooth step, which begins with counting. Each processor receives sufficient information to determine the message flow of the row and column movements during the distribution phase. The row movement spreads messages almost evenly in each row of a quadrant so that the number of messages is almost even in each column. The messages A,B,C,D and K are spread in row 0 of the top left quadrant.



The array after the distribution phase of the first smooth step. The positions of messages A,B,C and D could be permuted, since they occupied the same processor prior to smoothing and messages are indistinguishable during smoothing. The positions of other messages are determined by the smoothing algorithm. Messages in the upper half are packed in low-indexed processors and messages in the lower half are packed in high-indexed processors. The algorithm will now recur, breaking each 4 by 4 quadrant into four 2 by 2 quadrants and doing another move and smooth.

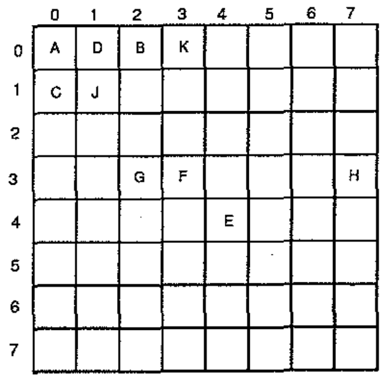


Figure 3.1: The first recursive stage of Algorithm Q

3.2 Move Step of Algorithm Q

Algorithm Q divides the n by n mesh into four disjoint submeshes at every recursive step. This section will describe the move step of the first recursive stage on an s by s submesh. Each processor in the mesh knows its relative row and column addresses in the s by s submesh.

The move step begins with the partitioning of the s by s mesh into four disjoint $s/2$ by $s/2$ submeshes, or quadrants. We index the quarters as I, II, III, and IV, beginning in the top right quadrant and moving counter clockwise. Each processor in the mesh can determine which quadrant it belongs to. Initially, each processor has at most one message. A processor having a message can determine whether the destination of its message is in the same quadrant where the processor belongs. By examining the most significant bits of column and row addresses of the destination, a processor can determine the quadrant that contains the destination processor of the message. The objective of the move step is to rearrange the messages in the mesh so that each message is located in the quadrant which contains its destination processor. We call this the *target* quadrant of the message.

If a message initially is not in its target quadrant, it is sufficient to move it to any processor in that quadrant to satisfy the objective of the move step. Because there are many messages that must travel from one quadrant to another, and processors don't have global information about the messages moving among quadrants, a processor cannot select an arbitrary processor in the target quadrant. If a message were sent to an arbitrary processor in the target quadrant, two problems could arise. First, some processors might receive many messages, exceeding the buffer capacity of the processor. Second, the time required to send all messages to their target processors would be unpredictable since there may be contention for links. Algorithm Q avoids contention and ensures that the maximum number of messages per processor during and after the move is predictable.

When a processor has a message which is not in its target quadrant, the processor sends it to the target processor which is at the same relative position in the correct quadrant. In an s by s mesh, the target processor is $s/2$ steps away along the row, or $s/2$ steps away along the column, or $s/2$ steps along the row and $s/2$ steps along the column away (on the diagonal). After the target address of each message has been determined, the movement of messages begins simultaneously in all processors. If the target processor of a message is on the same row as the current processor, the message is sent along the row. If the target processor of a message is on the same column,

Msg	I	D	T
A	(0,1)	(0,5)	(0,5)
B	(0,5)	(1,3)	(0,1)
C	(0,6)	(0,1)	(0,2)
D	(2,1)	(5,6)	(6,5)
E	(1,2)	(7,2)	(5,2)
F	(6,1)	(4,4)	(6,5)
G	(2,5)	(7,5)	(6,5)
H	(6,5)	(4,7)	(6,5)
J	(5,3)	(1,4)	(1,7)

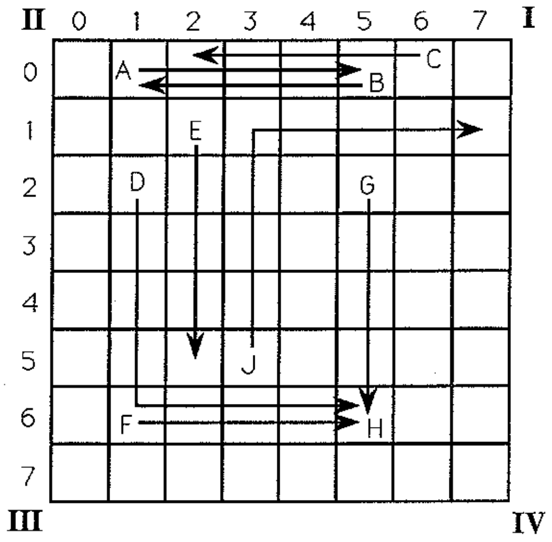


Figure 3.2: The move step in Algorithm Q

the message is sent along the column. Otherwise, the processor sends the message first along the column until the message reaches the row of the target processor; then the message is sent along that row until it reaches the target processor.

Since links are bidirectional and a processor can receive and send messages at the same time, contention for links arise only when messages change direction. There is no contention if each message moves along a row or a column. Hence, there is no contention among messages in a processor during the move step. At the beginning of movement, there is at most one message in a processor which must travel to another quadrant. During the first $s/2$ steps, each message either remains in place, moves $s/2$ steps along a column, or moves $s/2$ steps along a row. No message changes direction, so this part of the movement is contention-free. After $s/2$ time steps, the messages which were $s/2$ steps away from their destination have reached their target processor and stop moving. Only those messages whose destination was on a diagonal need to travel $s/2$ steps further along a row. These messages all change direction at the same time, and no processor contains more than one such message. Hence there is no contention when the messages change direction, and there is no contention during the next $s/2$ time steps because all messages move only along rows in the mesh.

Figure 3.2 shows an example of the move step for an 8 by 8 mesh. On the left of the figure, initial position (I), destination address (D), and computed target address (T) of each message are listed. For example, message A in quadrant II has its destination in quadrant I. Its target address is $(0, 5)$, which is at the same relative position in quadrant I as processor (0,1) in quadrant II. Note that the destination and the target address of message A coincidentally happen to be same. For another example, message D in quadrant II has its destination address in quadrant IV, so the target address is $(6, 5)$, which is the same relative position in quadrant IV as $(2, 1)$ in quadrant II. Message D is four steps along the row and four steps along the column away from its target processor. Since message H has its destination address in the same quadrant as its current position, the target address of message H is the same as its current position, even though its destination address is not same as its current position. Compare the target addresses and destination addresses of other messages. During the move step, messages A, B, C and F are sent along their rows to their targets that are four steps away. Messages E and G travel along their columns, because their targets are on the same column. Messages D and J will move along the column first even though their target processors are on the diagonal. (Recall that if a message has to travel along a row and a column, it travels along the column first.) Note that there is no contention among messages B and C because they are pipelined along the row. Message H does not move. After four time steps, messages A, B, C, E, F, G, and H have arrived at their target processors. But messages D and J are still four steps away from their target processors and will move along their rows, completing the move step for our example.

The time required to rearrange messages in the move step in an s by s mesh is s steps, because there is no contention among messages, and the farthest target is s steps away for messages which must travel to the diagonally-opposite quadrant. (For example, messages moving from quadrant I to III, or from quadrant II to IV, will take s time steps.)

During the move step, no processor will contain more than five messages at any time. During the first $s/2$ time steps of the move step, some messages are moving along rows, others are moving along columns, and others remain at their initial processors. A processor which contains a message initially in the correct quadrant may receive four transient messages, one across each link, that are travelling through the processor. Note, however, that there is no contention, so all four transient messages can be sent out at the next step. This is the only case in which a processor may contain five messages during the first $s/2$ time steps. In Figure 3.3, for example, processor $(2, 2)$ contains message E which is in the correct quadrant, quadrant II. Message A is travelling from processor $(0, 2)$ of quadrant II to processor $(4, 6)$ of

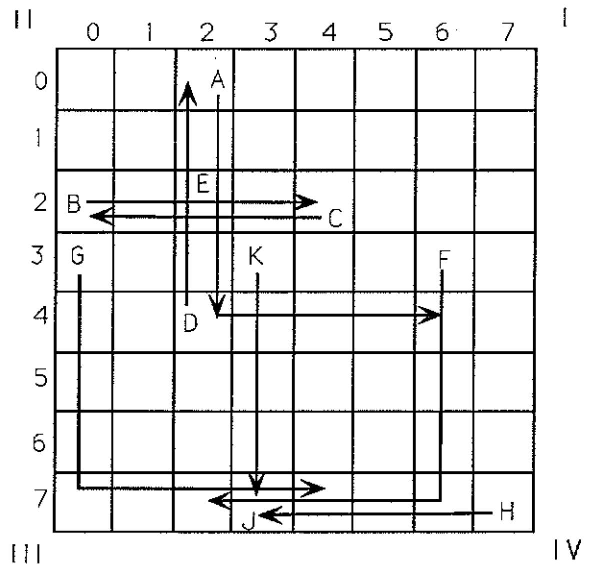


Figure 3.3: Buffer requirements during the move step

quadrant IV. Message D is travelling from processor $(4,2)$ to processor $(0,2)$. Message B is travelling from processor $(2,0)$ to processor $(2,4)$. Message C is travelling from processor $(2,4)$ to processor $(2,0)$. So, after two time steps, processor $(2,2)$ happens to have five messages, one that did not move (message E), and four transient messages (messages A, B, C, and D).

During the first s/2 time steps of the move step, each message travelled along a row or along a column. During the next s/2 time steps, each message will travel only along a row, since the messages whose targets were on the diagonal have moved to the target row and only those messages will move to the target column. So, there will be at most five messages in a processor during next $s/2$ times steps at any time, one message that did not move during the entire move step, at most two messages that were delivered during the first $s/2$ time steps and that will stay till the end of the move step, and at most two transient messages that are moving along the row. In figure 3.3, for example, processor $(7,3)$ contains three messages after four time steps; message J that did not move, message K that was sent along the column and message H that was sent along the row. After another three time steps, messages G and H pass through processor $(7,3)$ (that is, messages G and H are transient messages), and messages J, H and K stay until the end of the move step. So processor $(7,3)$ happens to contain five messages after seven time steps, but it will contain only three messages after the move step. Note that all message movement during the second $s/2$ steps stops at the same time, because all messages that move travel exactly the same distance.

After the move step, however, no processor will contain more than four messages. Initially, there is at most one message per processor. During the move step, a processor may keep the message that it contains initially. Since there is only one processor at the same relative position in each quadrant, and only those processors can send a message to each other, a processor may receive messages from at most three other processors in other quadrants. So after the move step a processor can contain up to four messages, one that was there initially and three more that are sent from processors in other quadrants. In Figure 3.2, for example, processor $(6,5)$ will contain four messages: D, F, and G from other quadrants and H from itself.

This section can be summarized in the following theorem.

Theorem 3.1 *The move step of the first recursive stage on an s by s mesh will take s time steps, and no processor will contain more than five messages at any time. After the move step, no processor contains more than four messages.*

3.3 Smooth Step of Algorithm Q

At the end of a move step, a processor may contain up to four messages. However, the number of messages in each quadrant does not exceed the number of processors in a quadrant. In order to recur, the precondition of Algorithm Q (at most one message per processor) must be restored to avoid exponential growth of the number of messages in a processor. The objective of the smooth step of each recursive stage is to rearrange the messages in each quadrant so that each processor contains only one message without moving any message out of the correct quadrant.

Smoothing of a quadrant is done in two phases. The *counting* phase informs each processor about the initial and final configurations of messages in each quadrant that contains the processor. After the counting phase, each processor knows the number of messages it will receive from and send to its neighbors to accomplish smoothing in a quadrant. The *distribution* phase moves each message in the mesh to achieve the final configuration of the smooth step. During the distribution phase, processors first transmit and receive messages along rows and then transmit and receive messages along columns. (Messages are moving either along rows or along columns at any time during smoothing, but not along both.) After the distribution phase, each processor will have at most one message.

After the move step is applied to an s by s mesh, smoothing is done in the four $s/2$ by $s/2$ quadrants simultaneously and independently. Because no message or information is exchanged between processors in different quadrants, we describe smoothing on a single q by q square mesh. The counting and distribution phases are described in Sections 3.3.1 and 3.3.2 respectively.

3.3.1 Counting

Algorithm Q relies on *smoothing*, which takes a square submesh having several messages in a processor and distributes the messages so that no processor contains more than one message. Each quadrant that undergoes smoothing is

> a square mesh with q^2 processors arranged in q rows and q columns; initially, no processor contains more than four messages; and the total number of messages in the quadrant is no greater than q^2.

We'll assume that q is an even integer. As well, we will use the row-major index of each processor in the mesh as its address. Thus, for a q by q mesh, the address of the processor at (r, c) is given by $rq + c$ where $0 \leq r, c < q$.

To speed up the process of smoothing, we divide the square mesh into halves (top and bottom), and we perform counting and distribution on the two halves simultaneously and independently. If there are u messages in the top half before smoothing, and v messages in the bottom half, then $u+v \leq q^2$. After the smooth step, each processor with index 0 through $u-1$ and each processor with index q^2-v through q^2-1 will contain a single message, while remaining processors will not contain any message. Note that u or v may be greater than the number of processors in the half. If the number of messages in the upper half is greater than the number of processors in the upper half, the excess messages spread into the lower half of the mesh. The same is true when there are excess messages in the lower half. In the rest of this section, we describe only the part of the algorithm for counting the upper half of the square mesh. The part of the algorithm for counting the lower half of the square mesh can be constructed by a simple transformation on the processor indices.

We denote by $C(r, c)$ the number of messages in the processor (r, c) before the counting phase; $0 \leq C(r, c) \leq 4$ for all $0 \leq r, c < q$. The algorithm we will describe provides each processor in the mesh with the following information, which will be used to determine the final configuration of each row and the flow of messages through the links during distribution:

For a processor (r, c) in the upper half of the mesh, where $0 \leq r < q/2$ and $0 \leq c < q$,

1. the total number of messages in row r, that is, $\sum_{i=0}^{q-1} C(r, i)$, and

2. the number of messages to the left of processor (r, c) in row r (that is, $\sum_{i=0}^{c-1} C(r, i)$), and

3. the number of messages above the current row r, that is, the total number of messages in all the processors of rows 0 through $r - 1$ (that is, $\sum_{i=0}^{r-1} \sum_{j=0}^{q-1} C(i, j)$).

For the processor (r, c) in the lower half of the mesh, where $q/2 \leq r < q$ and $0 \leq c < q$,

1. the total number of messages in row r, and

2. the number of messages to the right of processor (r, c) in row r, and

3. the number of messages below row r, that is, the total number of messages in the processors of row $r + 1$ through $q - 1$.

We will describe the algorithm that counts the messages in the upper half of a mesh. If a mesh contains a single processor, there is no need to count, because the processor already has all information about the mesh. If a mesh contains more than one column ($q > 1$), the following procedure is used. (Recall that we treat only the upper half of the mesh.)

In the following, we represent count information from eastern, western and northern neighbors by E, W, and N respectively. The number of messages contained in a processor is denoted by Z. Input from a nonexistent neighbor (at the edge of the network) or a neighbor that is not part of the mesh (of the current region) is 0.

1. In parallel, for each row, a cumulative count of messages flows from the left end of each row to the right of the row (from west to east), with each processor sending to its eastern neighbor the value $Z + W$. The flow begins with the leftmost processor of each row sending to its eastern neighbor the value Z (since the leftmost processor has no western neighbor in the submesh and hence W is 0 by convention). This message wave continues for $q - 1$ steps. The value of the input

value W to each processor is equal to the number of messages to the left of the processor in the current row. Similarly and concurrently, the processors in the rows each send the value $E + Z$ to their western neighbor. The flow begins with the rightmost processor of each row sending its western neighbor the value Z (since the rightmost processor has no eastern neighbor in the submesh and hence E is 0 by convention). This message wave continues for $q - 1$ steps. The value of the input value E to each processor is equal to the number of messages to the right in the current row. As soon as a processor receives the value E from its eastern neighbor and the value W from its western neighbor, it can compute the total number of messages in the current row, which is the value $E + Z + W$. Since this is done in parallel for each row, this part of the counting phase requires $q - 1$ steps. Note that the information passed in counting consists simply of an integer with no more than $\log_2 4q$ bits. Because this may be substantially smaller than the size of a message being routed through the mesh, we refer to these as *integer messages*, and say that the first part of counting requires $q - 1$ integer message steps.

2. As soon as a processor receives the values E and W, it can compute the value $E + W + Z$, the number of messages in the entire row and pass this information to the processor in the row below it. Information thus passes along each column from the top row to the bottom row (of the upper half), informing each processor how many messages are contained in processors with smaller row indices. By convention, each processor in the top row receives the input $N = 0$ from its northern neighbor, and thus the value passed south by each processor is $N + E + W + Z$. Since counting is done in the upper and lower halves of a region independently, this requires $q/2 - 1$ integer message steps, each with $\log_2 q^2$ bits. Note that the value N provided to each processor by its northern neighbor is the total number of messages in the processors above the current row. Because processors in the middle of a row are the first to receive both E and W, the columnar information flow begins in the middle columns first, and gradually spreads to outer columns. The counting phase continues until the last processors on the outermost columns receive the information from their northern neighbors.

Figure 3.4 shows an example of the counting phase on the upper half of an eight by eight mesh. Processors are represented by the squares; the row and column indices appear above each column and to the left of each row. The number of messages in a processor is shown in the square. The arrows represent the information flow between processors. The direction of arrows is the direction of information flow, and the integer above or below the arrows is the value of the information. For example, as soon as processor $(0,3)$ receives the value 3 from its western neighbor and the value 7 from its eastern neighbor, it knows that there are 12 messages in row 0, since it has 2 messages. As soon as processor $(0,3)$ computes the value 12, the total number of messages in row 0, it passes the value to the southern neighbor. For another example, processor $(1,2)$ receives the value 7, 11, and 12 from its western, eastern, and northern neighbors respectively. Now the processor $(1,2)$ knows that there are 21 messages in row 1, since Z is 3, W is 7, and E is 11. It knows also that there are 12 messages in row 0, since N is 12. Now it passes the value 33 to its southern neighbor.

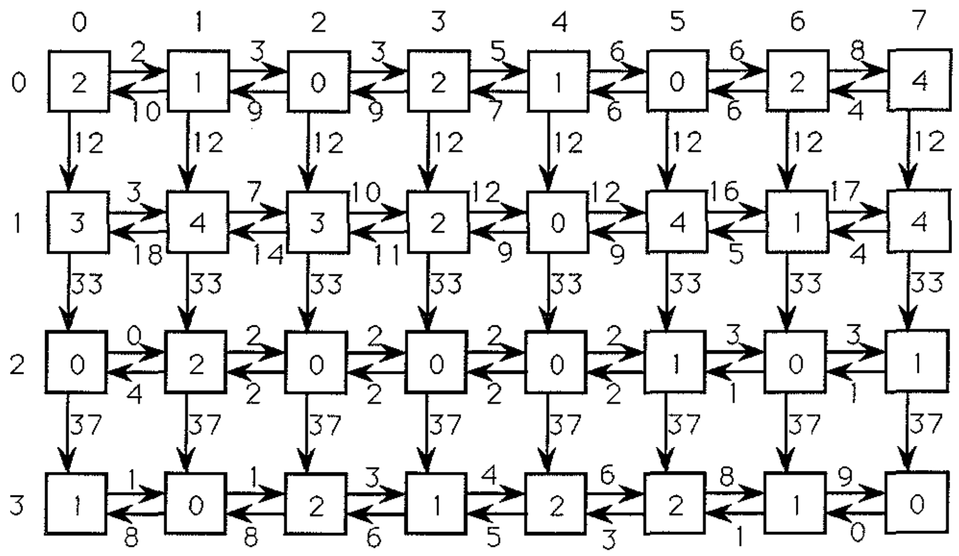
Lemma 3.1 *Counting a q by q square mesh requires $3q/2-2$ integer message steps.*

Proof: Recall that the counting is performed on the upper and lower halves simultaneously. Step 1 of the procedure above is executed in parallel on each row. Since the information flow along columns begins last along the outermost columns, Step 2 begins on the outermost columns after $q-1$ integer message steps. Then, Step 2 will take $q/2-1$ integer message steps because the information flows involve only the upper or the lower half of the mesh. □

3.3.2 Distribution

After the counting phase is complete, distribution of the messages occurs in two phases, *row movement* and *column movement.* Row movement redistributes the messages in each row so that no column of the mesh contains more than q messages. Roughly speaking, row movement spreads messages as evenly as possible within each row. Column movement redistributes messages in each column so that no processor has more than one message. If these phases do not overlap in time (that is, movement along rows is completed before movement along columns begins), no more than four buffers are needed. Contention for communication links can arise during both phases, but only of a restricted type since no message changes direction during either phase.

Since processors initially have up to four messages, and each link can handle only one message at a time, messages compete for the links. During



Except for incoming values of 0 for boundary elements, all E, W and N values are shown as labels on arrows.

Figure 3.4: Counting in the upper half of an 8 by 8 mesh.

row movement, messages are treated as indistinguishable, so priorities are not used. During column movement, each messages is assigned a target and contention is resolved according to priorities based on the distance to be travelled: the message that must travel the farthest has the highest priority. The distance to be travelled is based on the target address which is assigned prior to the column movement.

We first describe the row movement in a q by q submesh. The purpose of row movement is to distribute the messages in the submesh as evenly as possible across the columns of the submesh. This can be approximated by distributing the messages of each row as evenly as possible across the processors of each row. The only difficulty is making sure that, when the number of messages in a row is not evenly divisible by the number of processors, the 'extra' messages of different rows are not placed in the same columns. Row movement handles this problem by making sure that the 'extra' messages of successive rows are placed in successive columns. Thus, if row 0 has k_0 extra messages (that is, the number of messages mod $q = k_0$), then processors in columns 0 through $k_0 - 1$ each have one more message after row movement than the other processors of that row. And if row 1 has k_1 extra messages, then these messages are placed in columns k_0 through $(k_0 + k_1 - 1) \bmod q$.

For each row, only extra messages are shown. The set of extra messages comprises those that cannot be evenly distributed among the processors of a row. If a row has c processors, the number of extra messages is between 0 and $c-1$. Extra messages in each row are distributed evenly through the columns. Note that wrap-around of the extra messages occurs in row 2. Columns 0 through 4 each has three extra messages, while columns 5 through 7 have two extra messages.

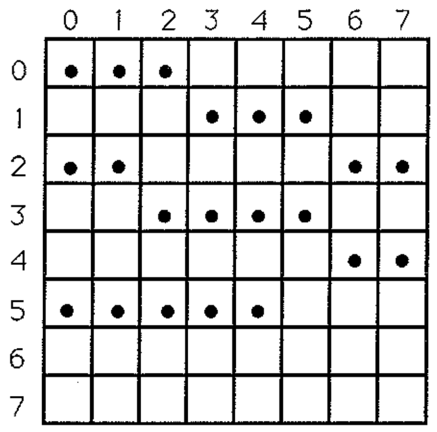


Figure 3.5: Layout of extra messages after row movement

Thus, the submesh row movement distributes the extra messages of each row evenly through the columns. Figure 3.5 illustrates how extra messages are arranged by row movement.

We now describe how row movement rearranges each row in a q by q mesh. Since there is no interaction between rows and counting is done on two halves of the square separately, we will describe how the processors in the upper half perform row movement.

Initially, each processor in the upper half of the square mesh has at most four messages. Messages are to be moved among processors in each row of the upper half to reach a specified final configuration, where a final configuration will have the following property:

Final Configuration Property: *For any submesh consisting of the first r rows, $0 \leq r < q/2$, either the total number of messages in each column is the same, or there exists some k, such that the total number of messages in any column $c < k$ is greater by one than the total number of messages in any column $c' \geq k$.*

Note that the Final Configuration Property is an assertion about all submeshes consisting of the first i rows, $1 \leq i \leq q/2$. This implies, for example, that for $i = 1$, the messages must be distributed approximately evenly in the processors. More generally, it is easy to show by induction that the Final Configuration Property implies the following property:

Post Row Movement Property: *For each row, the numbers of messages in any two processors either are equal, or differ only by one. Furthermore, if we consider the leftmost processor to be contiguous to the rightmost processor in a row, then the processors with an extra message are contiguous.*

Figure 3.6 shows an example of initial and final configurations for the upper half of an eight by eight square mesh. Processors are represented by squares; the row and column indices appear above each column and to the left of each row. The messages are shown with shaded circles.

In the final configuration of Figure 3.6.b, the numbers of messages in two processors of any row differ by at most one. The Final Configuration Property is illustrated by considering the submesh consisting of the first three rows, rows 0, 1, and 2. Each of columns 0 through 4 contains five messages in this submesh, while each of columns 5 through 7 contains four messages. Moreover, in each row the processors with an extra message are positioned as described by the Post Row Movement Property; for example, processor $(0,3)$ contains two messages and processor $(0,4)$ contains one message in row 0, while processor $(1,4)$ contains one more message than processor $(1,3)$ in row 1.

We will now describe how each processor first computes the number of messages that it will contain in the final configuration and then determines the number of messages to send out and receive from its neighbors during row movement. From the counting phase, processor (r,c) in the upper half knows the following:

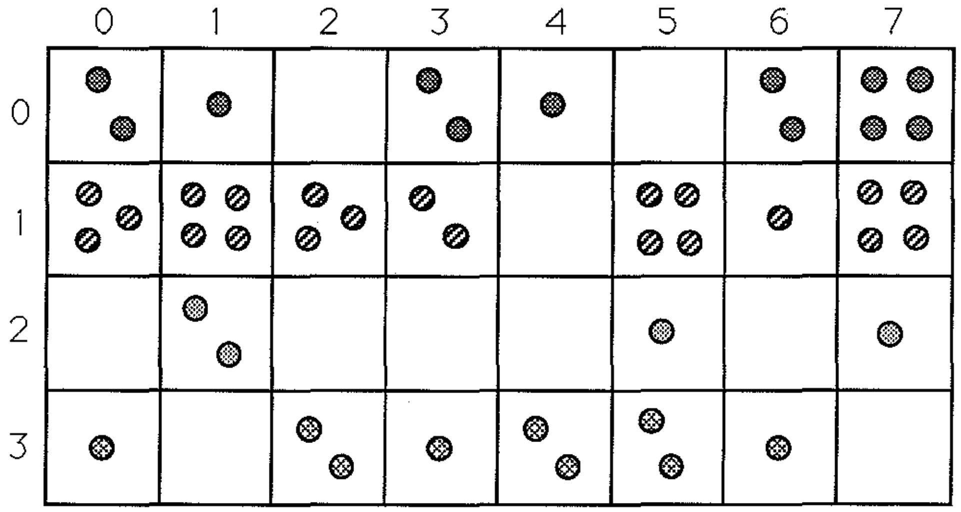
- the total number of messages in row r, which we denote by N_r, and
- the number of messages to the left of processor (r,c) in row r, which we denote by $L_{r,c}$, and
- the total number of messages in rows 0 through $r-1$, which we denote by S_{r-1}.

We define $S_{-1}=0$, and note that $S_r = S_{r-1} + N_r$ for $r \geq 0$.

We denote by $I_{r,c}$ the number of messages contained in processor (r,c) initially, and we denote by $F_{r,c}$ the number of messages contained in processor (r,c) in the final configuration. We define $U_{r,c}$ by the following equations:

$$\begin{aligned} U_{r,c} &= \lceil S_r/q \rceil \quad \text{if } 0 \leq c < (S_r \bmod q) \\ U_{r,c} &= \lfloor S_r/q \rfloor \quad \text{if } (S_r \bmod q) \leq c < q \end{aligned}$$

(a) Before row movement



(b) After row movement

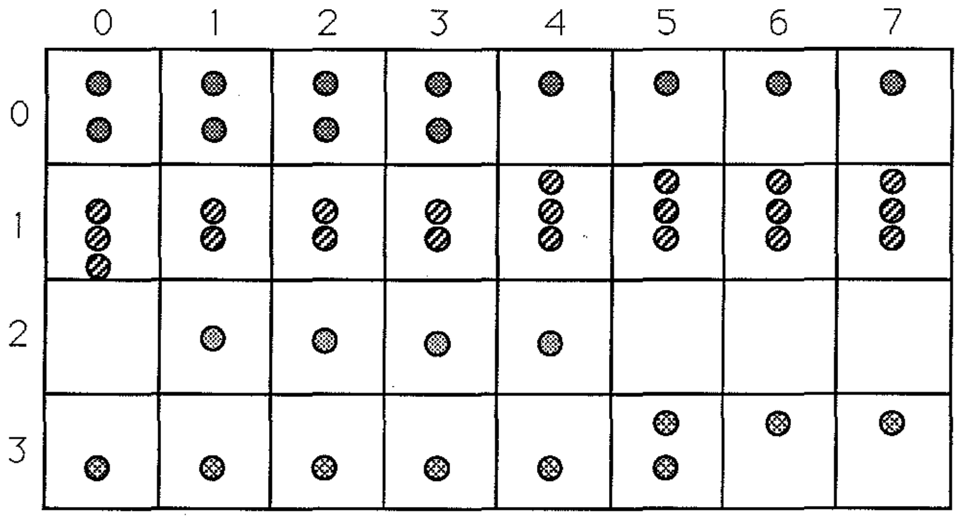


Figure 3.6: Row movement in the upper half of an 8 by 8 mesh.

Then $U_{r,c}$ is the total number of messages in column c of the submesh consisting of the first r rows in the final configuration.

Each processor (r, c) determines $F_{r,c}$ by the following equation:

$$F_{r,c} \quad = \quad U_{r,c} - U_{r-1,c} \tag{3.1}$$

Lemma 3.2 *The collection of $F_{r,c}$ for $0 \le r < q/2$ and $0 \le c < q$, which are defined by Equation 3.1, satisfies the Final Configuration Property. That is, for each processor (r, c), $F_{r,c}$ is the number of messages in the final configuration.*

Proof: By Equation 3.1, $\sum_{i=0}^{r} F_{i,c} = U_{r,c}$ for $0 \le c < q$. For $r \ge 0$, we do a case analysis:

(Case 1) When $S_r \bmod q = 0$:
Since $U_{r,c} = \lfloor S_r/q \rfloor$ for $0 \le c < q$, the number of messages in each column is the same.

(Case 2) When $S_r \bmod q = k$, where $k \ne 0$:
By definition of $U_{r,c}$,

$$\begin{array}{ll} \text{for } 0 \le c < k, & U_{r,c} = \lceil S_r/q \rceil = \lfloor S_r/q \rfloor + 1, \text{ and} \\ \text{for } k \le c < q, & U_{r,c} = \lfloor S_r/q \rfloor. \end{array}$$

This satisfies the Final Configuration Property. □

For any row r, ($N_r \bmod q$) processors contain $\lceil N_r/q \rceil$ messages and are located in contiguous columns starting at $S_{r-1} \bmod q$ through $(S_r - 1) \bmod q$, while the rest of processors in that row contain $\lfloor N_r/q \rfloor$ messages. Figure 3.6 shows an example. In row 1, there are 21 messages, that is, $N_1 = 21$. Since $S_0 \bmod 8 = 4$, and $S_1 \bmod 8 = 1$, each processor at columns 4, 5, 6, 7 and 0 contains 3 ($\lceil N_1/8 \rceil$) messages, and the rest contains 2 ($\lfloor N_1/8 \rfloor$) messages. In row 1 the column index of the last processor with an extra message is 0, and in row 2 the first position of the processor with an extra message is 1, since $S_1 \bmod 8$ is 1, where S_1 is 33 (the number of messages in rows 0 and 1).

The information required to determine the shape of a row in the final configuration is common to all processors within a row; this information consists of the number of messages in the current row and the total number of messages in all previous rows. This means that, for the final configuration, each processor can determine the number of messages that will be in its row to its left in the final configuration.

After counting, each processor (r, c) in the upper half can compute the following:

1. the number of messages initially contained in the processor, that is, $I_{r,c}$,

2. the number of messages to be contained in the processor after row movement, that is, $F_{r,c}$,

3. the number of messages in row r to the left of the processor initially, that is, $L_{r,c}$, and

4. the number of messages in row r to the left of the processor after row movement (that is, in final configuration), which we denote by $L'_{r,c}$.

Using these numbers, each processor can determine the number of messages it will send to or receive from its neighbors. For processor (r, c), the number of messages which will travel through the left link is given by the value $(L_{r,c} - L'_{r,c})$. If this value is positive, processor (r, c) will receive $(L_{r,c} - L'_{r,c})$ messages from its left neighbor during row movement to achieve the final configuration. If the value is negative, processor (r, c) will send $(L'_{r,c} - L_{r,c})$ messages to its left neighbor during row movement. The number of messages which will travel through the right link is given by the value $(L_{r,c} + I_{r,c} - L'_{r,c} - F_{r,c})$. If this value is positive, processor (r, c) will send $(L_{r,c} + I_{r,c} - L'_{r,c} - F_{r,c})$ messages to its right neighbor during row movement. If the value is negative, processor (r, c) will receive $(L'_{r,c} + F_{r,c} - L_{r,c} - I_{r,c})$ messages from its right neighbor during row movement.

Figure 3.7 shows an example of flow calculation for the upper half of an eight by eight square mesh. Processors are represented by squares; the row and column indices appear above each column and to the left of each row. The number of messages initially contained in a processor is shown in the square as an integer above the diagonal, while the number of messages contained in the processor after row movement is shown as an integer below the diagonal. Arrows represent the channels between processors, and the arrows represent their direction. Integers on the arrows are the number of messages which flow through the channel. An arrow with no integer indicates that no message will flow through the channel. For processor $(1, 3)$, there are 10 messages in the processors to its left initially and 7 messages in the final configuration, which means that 3 messages will be received from the left neighbor.

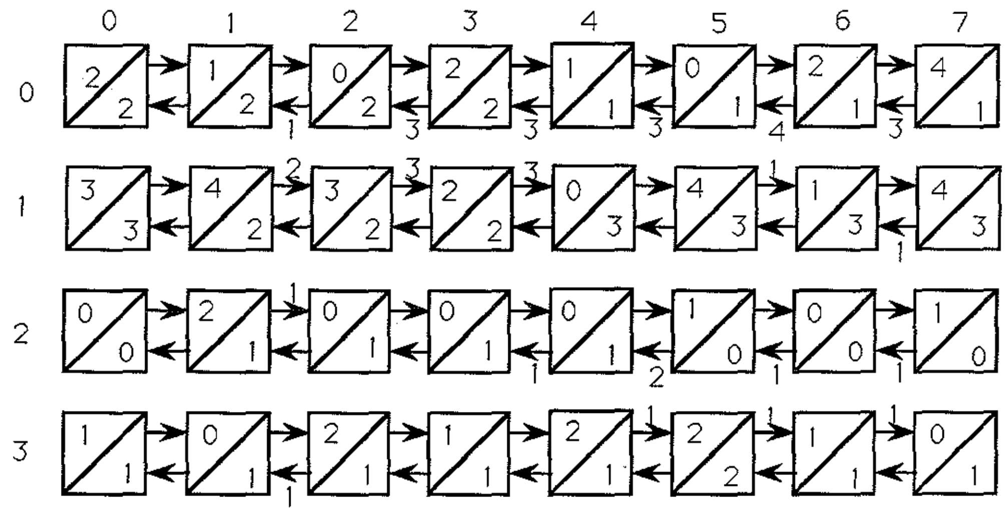


Figure 3.7: Message traffic across channels during row movement

We have described the algorithm for the upper half of the square mesh. For the lower half of the mesh, the same algorithm is applied, but the index needs to be transformed; that is, processor (i, j) in the lower half corresponds to processor $(q-1-i, q-1-j)$ in the upper half, where $0 \leq i, j < q$.

Once a processor in the mesh determines the number of messages that will travel through its channels, it sends messages to its neighbors as soon as those messages are available. Completion by all processors establishes the final configuration for row movement.

The following lemma summarizes the effect of row movement.

Lemma 3.3 *After row movement in the mesh, there are at most q messages in any column of the mesh.*

Proof: Assume that there are $xq + y$ messages in a mesh, where x and y are integers such that $0 \leq x \leq q$ and $0 \leq y < q$. If there are $uq + v$ messages in the upper half and $lq + m$ messages in the lower half, where u, v, l and m are integers such that $0 \leq u, l \leq q$ and $0 \leq v, m < q$, then $xq + y = (u + l)q + v + m$, and $0 \leq v + m < 2q$.

(Case 1) When $y = v = m = 0$:

Since $v = m = 0$, there are $u + l$ messages in each column, that is, there are $x \leq q$ messages in each column.

(Case 2) When $y = 0$ and $v + m = q$:

Note that $v, m > 0$. By the Final Configuration Property, in the upper

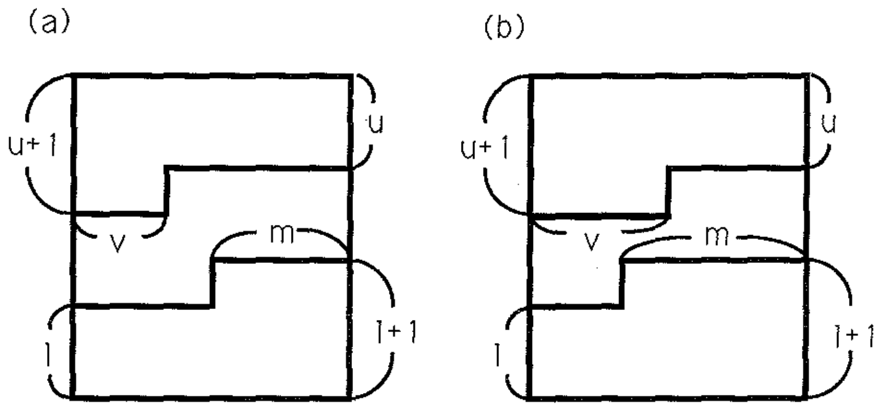


Figure 3.8: Message distribution after row movement

half, there are $u + 1$ messages in each of columns 0 through $v - 1$, and u messages in the rest. Likewise, in the lower half, there are l messages in each of columns 0 through $q - m - 1$, and $l + 1$ messages in the rest. Since $v - 1 = q - m - 1$, there are $u + l + 1$ messages in each column. Hence, there are $x \leq q$ messages in each column.

(Case 3) When $0 < y = v + m < q$:
By the Final Configuration Property, there are $u + l + 1$ messages in each of columns 0 through $v - 1$ and $q - m$ through $q - 1$. There are $u + l$ messages in the rest. (See Figure 3.8.a.) So, $u + l + 1 = x + 1 \leq q$, since $y > 0$.

(Case 4) When $0 < y = v + m - 1 < q$:
By the Final Configuration Property, there are $u + l + 2$ messages in each of columns $q - m$ through $v - 1$, and $u + l + 1$ messages in the rest. (See Figure 3.8.b.) Since $x = u + l + 1$ and $x < q$, $u + l + 2 \leq q$.

□

Lemma 3.4 *Consider a q by q square mesh with at most four messages in each processor, and no more than $4q$ messages in each row. Row movement of the smooth step on this mesh takes at most $1.2q$ data message steps.*

Proof: From the counting phase, in the current row each processor knows the number of messages to its left and the number of messages to its right. By the Final Configuration Property, for the final configuration each processor can compute the number of messages it will contain and the number of messages to its left. Therefore, Theorem 2.1 of Chapter 2 applies, since the message movement in each row is independent and messages travel only along the current row. The lemma follows immediately by substituting $b = 4$ and $c = q$. □

After row movement, there are at most q messages in any column of the mesh. Moreover, each processor in the upper half of the mesh can determine the number of messages in the column currently above it, since the Final Configuration Property holds for all processors in the upper half. For processor (r, c), $U_{r-1,c}$ is the number of messages in rows 0 through $r - 1$ of column c. Likewise, each processor in the lower half of the mesh can determine the number of messages in the column currently below that processor.

Since the movement of messages in each column is independent and the number of messages in the rows above (below) the processor is known to each processor in the upper (lower) half, the column movement of the square mesh is reduced to the one dimensional column movement problem discussed in Section 2.2.2. As described in Chapter 2, after column movement each processor in the mesh contains at most one message.

Note that target addresses must be assigned for column movement, since messages may travel from the top half of an array into the bottom half or vice versa. It is not possible for all the processors of a column to calculate the flow that will occur during column movement as was done for row movement. Figure 3.9 shows an example of column movement in an eight by eight square mesh. Each processor knows the target address of its messages. The arrows in Figure 3.9.a show the target of messages computed prior to column movement.

The following lemma describes the time steps required for column movement.

Lemma 3.5 *Assuming that q is a multiple of four, the column movement takes at most $3q/4$ data message steps on a q by q square mesh in which each processor contains at most four messages.*

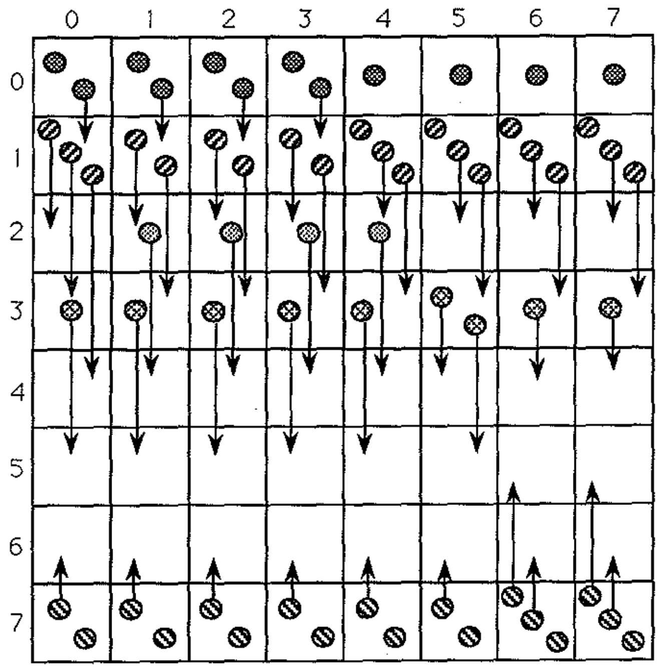
Proof: Immediate from Theorem 2.3 in Section 2.2.2. □

The following theorem summarizes the cost of the distribution phase.

Theorem 3.2 *The distribution phase in a q by q square mesh takes at most $1.95q$ data message steps. During the distribution phase, no processor contains more than four messages.*

Proof: The number of routing steps required follows immediately from Lemma 3.4 and Lemma 3.5. The buffer requirement follows immediately from Theorem 2.2 of Section 2.2. □

(b) Before column movement



(b) After column movement

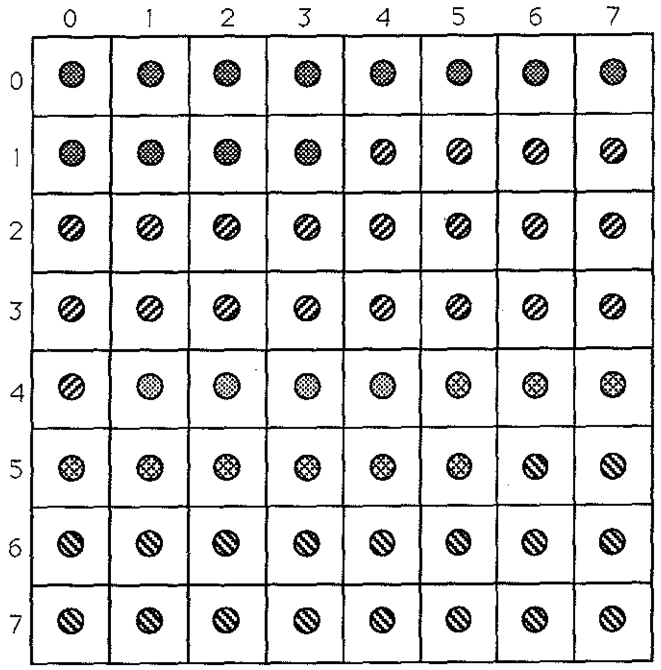


Figure 3.9: Column movement in an 8 by 8 mesh.

3.4 Time Complexity and Buffer Requirement

Algorithm Q is a recursive algorithm, in which recursion stages are applied to each subproblem simultaneously. In this section, we describe the time complexity of Algorithm Q. When the algorithm is applied to an n by n square mesh, it takes at most $5.5n$ message steps. At any time, Algorithm Q requires no more than five buffers per processor, regardless of the mesh size. Even though Algorithm Q is applicable to any rectangular mesh, for simplicity's sake, we will discuss the time complexity in the context of an n by n mesh, where n is a power of two.

Evaluation of the time complexity of Algorithm Q is based on the following assumptions:

1. The computation in a processor is considerably faster than the packet transmission from one processor to another. Therefore, we equate the time needed to execute a routing algorithm with the time required for passing messages between processors.

2. There are two kinds of messages: data messages and integer messages. A data message is a message with a destination address, flags, and data. An integer message is a message with only one integer field, which is used during the counting phase of the smooth step.

3. Throughout this dissertation, t_M denotes the cost of sending a data message packet from one processor to another, and t_m denotes the cost of sending an integer message packet from one processor to another. Typically, $t_M > t_m$.

Algorithm Q is based on successive partitionings of the mesh into four submeshes with the same size and shape and recursive applications of each recursive stage on submeshes simultaneously. Each recursive stage of Algorithm Q consists of two steps, the move step and the smooth step. As mentioned before, a smooth step consists of two consecutive phases, counting and distribution.

To establish a recurrence equation for Algorithm Q, we measure the size of the problem by the length of a side of the square mesh, that is, by the number of processors in one side of the mesh. We let $T_Q(n)$ represent the cost of solving a routing problem with Algorithm Q on an n by n square mesh. The first recursive stage on an n by n mesh begins with partitioning the mesh into four quadrants. Next, the move step is performed on the n by n mesh. Then the smooth step is applied to four $n/2$ by $n/2$ quadrants simultaneously.

Since the destination address of each message is contained in the quadrant containing the current processor of the message after the first recursive stage, the second recursive stage can handle four quadrants as independent half-size subproblems simultaneously. (That is, there is no message passing among quadrants.) This argument yields the following recurrence equation:

$$\begin{aligned}T_Q(n) \;\leq\; & \text{(cost of the move step on an } n \text{ by } n \text{ mesh)} \\ & + \text{(cost of the smooth step on four } n/2 \text{ by } n/2 \text{ quadrants)} \\ & + T_Q(\frac{n}{2})\end{aligned}$$

The cost of the move step on an n by n mesh is n data message steps by Theorem 3.1; that is, nt_M. The smooth step consists of two phases, counting and distribution. The cost of counting on an $n/2$ by $n/2$ quadrant is $(3n/4 - 2)$ integer message steps by Lemma 3.1; that is, $(3n/4 - 2)t_m$. The cost of distribution on a $n/2$ by $n/2$ quadrant is $0.975n$ data message steps by Theorem 3.2; that is, $0.975n\, t_M$.

This gives the following recurrence equations:

$$\begin{aligned}T_Q(1) &= 0 \\ T_Q(n) &\leq T_Q(\frac{n}{2}) + n\, t_M + (\frac{3n}{4} - 2)t_m + 0.975n\, t_M \\ &\leq T_Q(\frac{n}{2}) + 1.975n\, t_M + \frac{3n}{4} t_m\end{aligned}$$

Solving the equations, we get

$$\begin{aligned}T_Q(n) &\leq 3.95n\, t_M + 1.5n\, t_m \\ &\leq 4n\, t_M + 1.5n\, t_m\end{aligned}$$

By Theorem 3.1, no processor contains more than five messages at any time during the move steps of Algorithm Q. By Theorem 3.2, no processor contains more than four messages at any time during the smooth steps. Both these bounds are independent of the mesh size. Therefore, Algorithm Q requires only five buffers in a processor.

Theorem 3.3 *Algorithm Q on an n by n array of processors requires five buffers, $4n$ data message steps and $1.5n$ integer message steps.*

Proof: Immediate from the argument above. □

The size of the mesh is $n/2$ by $n/2$. Each processor in the hatched triangle contains four messages, and the total number of messages is $n^2/4$. After smoothing, each processor will receive a single message. So will the processor in the upper right corner. The nearest message is $(1 - \frac{1}{2\sqrt{2}})n$ steps away from the corner, which gives the lower bound of smoothing.

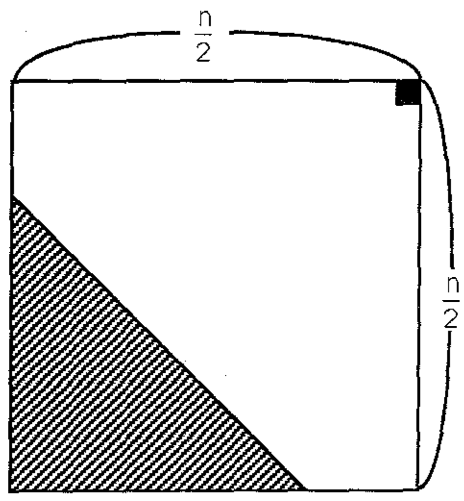


Figure 3.10: Message layout for the lower bound of smoothing

If each processor in a mesh has enough buffers (n buffers for an n by n mesh), we can solve routing problems using an algorithm similar to Algorithm Q, but without smoothing. If we apply Algorithm Q without smooth steps to an n by n mesh, messages can be delivered to their destination processors in $2n\, t_M$. From this point, Algorithm Q pays $1.5n\, t_m + 2n\, t_M$ as the cost of having only a small constant number of buffers in a processor.

The cost of move and smooth algorithms consists of the costs of moving and smoothing. In an n by n mesh, the lower bound for the move steps of Algorithm Q is $2n - 2$ routing steps, which is determined by the distance between two processors on the opposite corners. The lower bound of smoothing on an $n/2$ by $n/2$ submesh is $(1 - \frac{1}{2\sqrt{2}})n$; the bound is determined by the distance between the processor on the upper right corner and the nearest processor that has messages when $n^2/4$ messages are packed in the triangle on the lower left corner of the mesh. (See Figure 3.10.) Since smoothing has to be performed every recursive stage, the total cost of smoothing is at least $(2 - \sqrt{2}/2)$ routing steps. Therefore, the total cost of Algorithm Q is at least $3.3n$ routing steps for an n by n mesh if the move and smooth steps are not overlapped. Our algorithm described in this chapter has $4n$ data messages steps, which is close to the lower bound, $3.3n$ routing steps. But there is additional cost for counting, which requires $1.5n$ integer message steps.

If a data message is much bigger than an integer message (therefore, $t_M \gg t_m$ holds), Algorithm Q costs $4n\, t_M$ to solve a routing problem. But, in practice, the setup cost of communication is not negligible compared to actual transmission cost, so it may be more realistic not to ignore the cost of integer message, in which case Algorithm Q costs $5.5n$ routing steps. These costs are based on the assumption that the computation cost is negligible compared to the communication cost. In real machines, the computation cost in a processor is not free. Nevertheless, Algorithm Q can perform address calculations by comparing a single bit in each of the row and column addresses of the destination processor with the corresponding bits of the current processor. Because the address computation can be done by two simple bit-wise logical operations, the computation cost will indeed be small compared to the communication cost.

3.5 Restricted Broadcasting with Algorithm Q

Algorithm Q can solve (restricted) broadcasting problems[1] without additional buffers and routing cost because messages are duplicated appropriately during the move steps and the new copies introduce no contention. This section describes the method by which a processor duplicates messages with multiple destinations during the move steps and the effect that message duplication has on the smooth steps of Algorithm Q.

In broadcasting problems, initially each processor has at most one message, but a message may have multiple destination addresses. The total number of destination addresses for all messages is no greater than the number of processors. After communication is complete, no processor will contain more than one message. To represent multiple destination addresses, we need to refine the destination address field of a message, since a simple binary representation of a single address will not work anymore. Multiple destination addresses can be represented either as a list of processor addresses or by a mapping function that produces multiple destination addresses.

The actual representation of multiple addresses for a message is not important for Algorithm Q, so long as a processor can examine the collection of destination addresses at each move step. We assume the representation chosen meets this requirement. However, representation of multiple addresses may result in performance degradation of a routing algorithm since multiple addresses may require a longer message format. A longer message format could require a larger buffer for each message and more time (a slower clock) to send messages between processors. In this section, we do not consider the additional costs incurred by longer messages, and assume that these effects

[1]Defined in Section 1.3.4.

have been accommodated by adjusting the unit data message travelling time (that is, t_M) and the buffer size.

For each recursive stage Algorithm Q is modified as follows. At the beginning of each move step, each processor examines the destination addresses of the message it contains and determines which quadrants contain destinations of the message. A single copy of the message is sent to the processor at the same relative position in each quadrant that contains a destination address. Although a message may have multiple destination addresses in a single quadrant, only one copy of the message is sent to that quadrant. If the destination addresses of a message are contained in more than one quadrant, the processor makes an appropriate number of copies of the message and sends one copy to each of the quadrants. If one of the destination addresses is contained in the current quadrant, the processor keeps a copy of the message.

Recall that a message with a single destination address is sent along the column first and along the row next when its target processor is located in the quadrant on the diagonal. For broadcasting, only a single copy of the message is sent along the column, even when destination addresses of the message are contained by the quadrants both above (or below) and on the diagonal. Thus, if a copy sent along the column from a bottom quadrant has destination addresses in both upper quadrants, a copy is made by the target processor as soon as the message finishes its movement along the column; the new copy is then sent along the row to the processor on the diagonal.

In summary, a processor examines the message with multiple destination addresses and makes the proper number of copies, which may be two for the originating processor, but is no more than one for any other. Each copy of the message is sent to the corresponding processor in the correct quadrant as if there were no other copies. Specifically, in an s by s mesh, during the first $s/2$ steps, messages travel along the column and along the row. During the next $s/2$ steps, messages which need to travel another $s/2$ steps will travel along the row. Before the second $s/2$ step movement along the row, a message may be duplicated.

Despite the need to duplicate messages, there is no contention for links during the move step. Even though two messages begin to travel from the same processor, they are moving in different directions, one along a row and the other along a column. By delaying copying messages until additional copies are needed, contention for a link along a column is avoided.

At any time during the move steps, Algorithm Q requires no more than five buffers to handle broadcast problems. At the beginning of a move step, the message in a processor may be made into three copies; one copy stays, and the others travel along the row and along the column. During the first half of the move step, a processor contains up to five messages, one that did not move and four transients. After the first half of the move step, a processor contains up to three messages: one that did not move, another from the row, and the third from the column. One of the three copies, the one sent along the column, may be copied again, and a copy is sent along the row in the second half of the move step. During the second half of the move step, a processor contains up to five messages, three that stay and two transients moving along the row. After the move step, no processor contains more than four messages; four messages would include one that did not move and three from processors in other quadrants.

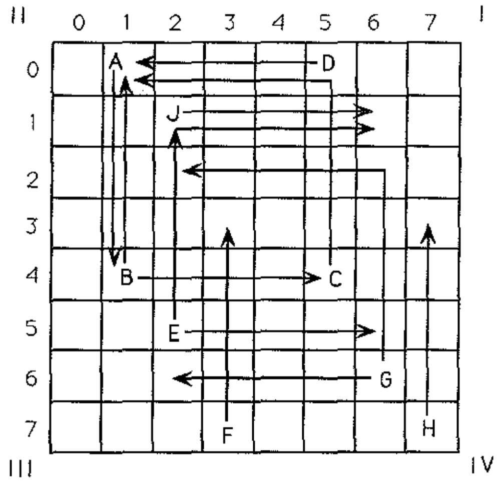
Figure 3.11 shows an example of the first recursive stage of Algorithm Q which solves a broadcasting problem on an eight by eight mesh. The destination addresses are shown in the table; each message is associated with a list of destination addresses; some have a single address, and others have several addresses across the mesh. Figure 3.11.b shows the original position of the messages and the paths they follow during the move step. Message A has two destination addresses, one in the current quadrant and one in the quadrant below. One copy of message A remains at processor $(0,1)$ and the other copy travels along the column. Message E has six destination addresses which cover four quadrants. Three copies exist during the first part of the move step; one remains at the current position, another travels along the row, and the other travels along the column. Another copy of message E is created by processor $(1,2)$ as soon as message E arrives at that processor, and a copy of message E travels along the row to processor $(1,6)$. Both message J and a copy of message E are delivered to processor $(1,6)$ along the same path, but there is no contention because they travel during different time intervals. Figure 3.11.c shows the result of the move step. No processor contains more than four messages. Although several copies of a message may co-exist in the mesh, each copy is in a different quadrant.

The smooth step is not affected by the multiple destination addresses, because it is applied to each quadrant independently and is unaffected by destination addresses. After the move step, the number of messages in a quadrant does not exceed the number of processors in the quadrant, and no processor contains more than four messages. Since messages are treated as indistinguishable by the smooth step, the smooth step can be used for broadcasting problems without modification by treating each copy of a message in the same way.

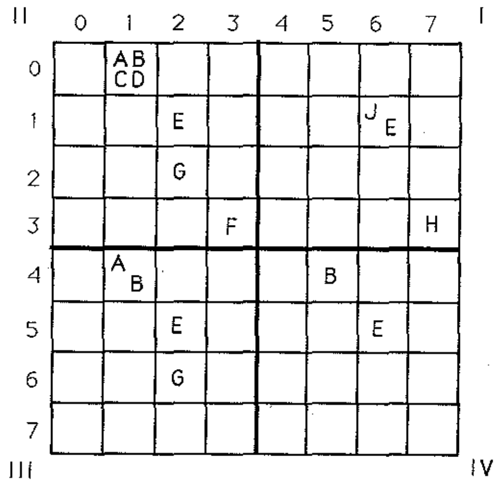
(a) **Destination Table**

Message	Destinations
A	(2,0) (5,3)
B	(2,1) (6,6) (7,3)
C	(2,2)
D	(2,3)
E	(0,1) (6,7) (3,4) (3,5) (5,0) (5,1)
F	(1,0)
G	(0,2) (6,3)
H	(0,6) (1,6) (2,6)
J	(1,7)

(b)



(c)



(d)

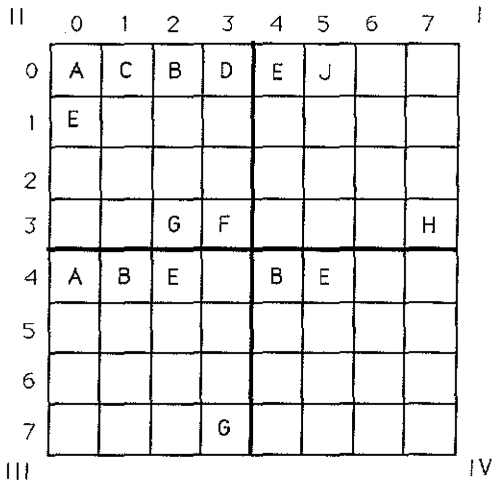


Figure 3.11: Move and smooth steps for broadcasting

If we assume the time required to examine destination addresses, determine what copies are necessary, and make copies of messages is negligible compared to routing costs, the time complexity and buffer requirement of broadcasting are the same as those for routing.

This section is summarized by the following theorem.

Theorem 3.4 *On an n by n mesh, Algorithm Q can solve a broadcast problem with five buffers per processor in* $5.5n$ *message steps if routing costs are the dominant expense.*

3.6 Extensions of Algorithm Q

Algorithm Q partitions a mesh into four square submeshes at each recursive stage. It can be extended in two ways; one extension is to partition a two-dimensional mesh into k^2 square submeshes of the same shape and size, and the other is to partition a K-dimensional hypercube into 2^K hypercubes with the same shape and size. We will describe both extensions of Algorithm Q and examine the effects and difficulties of these extensions.

In Section 3.6.1, we explain the extension of Algorithm Q by partitioning a two dimensional mesh into k^2 square submeshes. In this case, the move step of each recursive phase is no longer free from contention. That is, messages are delayed by contentions for links during move steps, because more than one message needs to travel over the same channel. The smooth step of each recursive step can be performed at the cost of additional time steps, because there are initially up to k^2 messages in a processor.

In Section 3.6.2, we will describe the extension to a K-dimensional mesh. Algorithm Q partitions a K-dimensional hypercube into 2^K sub-hypercubes by dividing the cube into two sections along every axis of the cube. Move steps are not contention-free anymore, because the messages which will travel along the same axis may arrive at their target processors through channels along several axes at the same time. Smooth steps can be done with $K-1$ row movements followed by a column movement. The counting phase of a smooth step must handle the K-dimensionality of the mesh.

In Section 3.6.3, we will describe a method to apply Algorithm Q to an arbitrary square mesh. Since the size of mesh is not a power of two, the submesh size is not an even number at some recursive stage. We will describe methods to partition an odd size mesh and to determine target addresses. We will then examine modifications of move step and the costs of unbalanced quadrants.

3.6.1 Extended Algorithm Q for k^2 Partitioning

As a move and smooth algorithm begins routing operations by partitioning an n by n mesh into b congruent regions, Algorithm Q can be extended to partition a mesh into k^2 square submeshes, each of which contains n/k by n/k processors, instead of four square submeshes. (Throughout this section, we assume that the size of mesh n is a power of k.) The extended Algorithm Q begins by partitioning the mesh into k^2 square submeshes. It performs a move step followed by a smooth step and then recurs on the k^2 submeshes, each of which is one k-th the size of the original. Like Algorithm Q, each processor in a mesh determines the submesh which contains the destination address of the messages that the processor contains. If a message is in the correct submesh (that is, the submesh that contains the destination address of the message), it does not travel during the move step. Otherwise, a message travels from its current position to the processor which has the same relative position in the correct submesh. Unlike the move step of Algorithm Q, the move step of the extended algorithm is not contention-free, in the sense that contention is possible because all messages cannot arrive at their target processors within the time bound imposed by the distance between the originating and target processors. Since up to k^2 messages may move to any given processor and since the processor in a corner submesh may be unable to receive more than two messages per n/k time steps, it can take up to $kn/2$ time steps for the processor to receive k^2 messages from the processors at the same relative position of other submeshes. See Figure 3.12 for a worst case example. Even though it is very unlikely that the worst case pattern occurs in more than one recursive stage, we cannot exclude the possibility when examining the time complexity. Note that for $k \geq 4$, $kn/2$ is greater than $2(k-1)n/k$, the distance between the farthest pair of processors in the same relative position of submeshes.

Unfortunately, it is difficult to find a strategy which accomplishes the move step in $kn/2$ time steps during the first recursive phase for an n by n mesh. To accomplish such a move step, a strategy has to make the messages heading to a processor come into the processor evenly through each link of the processor. That is, the number of messages over each channel to the processor has to be even for all channels. Since we are considering the extension of Algorithm Q, we will adapt the similar moving strategy to that of Algorithm Q during the move step. The rule is as follows:

1. Each processor determines the correct quadrant for the message that the processor contains. Each processor contains at most one message

Targets of the messages in the hatched rows and columns belong to the lower left corner. Since the messages in the same subblock move in the same direction during each n/k step interval, only two out of the k^2 messages on the black spots will arrive at the target processor in the lower left corner subblock every n/k step interval. In this example, k is 4.

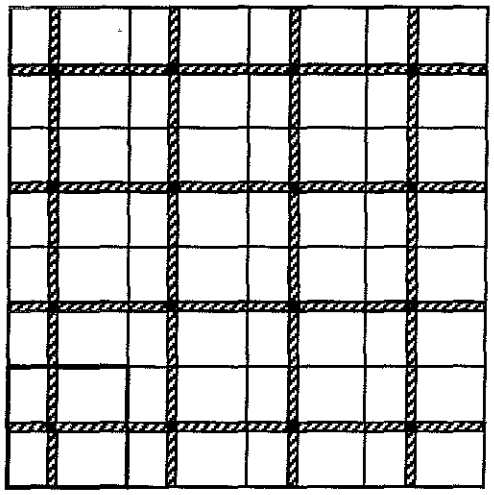


Figure 3.12: Worst case move step of extended Algorithm Q

initially. The target processor of each message is the processor at the same relative position in the correct square submesh. The messages begin to move simultaneously. Once a message starts moving, it has a higher priority than any message which is waiting for a channel or any message which is changing its direction of movement. If there is contention among moving messages, the message which is moving straight has priority; there can be only one message which is moving straight over a channel in the processor. The others are stored in the buffers and wait for the channel. If there is contention among messages which are in the buffers or which change direction, the message that will travel farthest has priority.

2. If the target processor is in the same row but in another column, the message travels along the row until it arrives at the target column. This movement is accomplished by moving 0, n/k, $2n/k$,....., or $(k-1)n/k$ steps along the row. These messages will never suffer from any delay, because they begin to move immediately and march straight to their target processors. These messages will arrive at their target processors after at most $(k-1)n/k$ time steps.

3. If the target is in the same column but in another row, the message travels along the column until it arrives at the target row. This movement is accomplished by moving 0, n/k, $2n/k$,....., or $(k-1)n/k$ steps along the column. These messages will never suffer from any delay,

because they begin to move immediately and march straight to their target processors. These messages will arrive at their target processors after at most $(k-1)n/k$ time steps.

4. If the target is in another column and in another row, the message travels along the current column until it reaches the target row. Then it changes direction and travels to the target column along the row until it arrives at the target processor. The first movement along the column is accomplished by moving 0, n/k, $2n/k$,....., or $(k-1)n/k$ steps along the current column, and will be never delayed, because these messages begin to move immediately and march straight to the target row. But when the message is changing its direction of movement, it will suffer from delays because of contentions with messages moving straight ahead, messages waiting for a channel in the buffers, or other messages that arrived simultaneously on other channels which are also changing their direction of movement.

Our strategy for a move step makes the channel utilization of a processor in the corner submesh very poor; up to $k-1$ messages may come through a column channel, and up to $k(k-1)$ messages come through a row channel. With the movement strategy described above, the move step of the first phase for an n by n mesh requires $(k-1)n$ time steps to move all messages to the correct submeshes.

During a move step, a processor may contain up to k^2+2 messages. For the first $(k-1)n/k$ steps, there are at most (k^2-4) messages that did not move and four transients. After $(k-1)n/k$ steps, no message travels along the column; that is, only two transient messages travel along the row. During the rest of the move step, a processor may contain up to k^2 messages that did not move and two transient messages.

At the end of the move step, a processor may contain up to k^2 messages; one is from before the move step, and k^2-1 have moved there from the corresponding processors in the other submeshes. However, the number of messages in the submesh does not exceed the number of processors in a submesh, because each message has a distinct destination address in the submesh. The smooth step follows the move step to re-establish the precondition that each processor contains at most one message. The smooth step has two phases: counting and distribution. The details of the smooth step are the same as those of Algorithm Q except that there are up to k^2 messages in a processor; during the first recursive stage, the smooth step is applied to k^2 submeshes of n/k by n/k size simultaneously. In keeping with the theorems in Chapter 2, the smooth step of the first recursive stage for an n by n mesh requires $1.5n/k$ integer message steps and $\frac{k^6+6k^4-4}{4k^3(k^2+1)}n$ data message steps, when $k \geq 4$ is even; the counting phase requires at most $1.5n/k$ integer message steps, the row movement of the distribution phase requires $\frac{k(k^2+2)}{4(k^2+1)}n$ data message steps, and the column movement of the distribution phase requires $\frac{k^2-1}{k^3}n$ data message steps. When $n = k^p$ for some integer p and even integer $k \geq 4$, the complexity of the extended algorithm is given by the following system:

$$\begin{aligned} T(1) &= 0 \\ T(n) &\leq T(n/k) + (k-1)nt_M + \frac{1.5n}{k}t_m + \frac{k(k^2+2)}{4(k^2+1)}nt_M + \frac{k^2-1}{k^3}nt_M \end{aligned}$$

which has the solution

$$T(n) \leq \frac{5k+3}{4}nt_M + \frac{1.5}{k-1}nt_m \qquad \text{(when } k \geq 4 \text{ is even)}$$

The number of time steps required for the extended Algorithm Q is linearly proportional to k, the number of submeshes along one side of the square. The extended Algorithm Q has the best performance when $k = 4$; that is, $T(n) = 5.75n\,t_M + 0.5n\,t_m$. It takes $6.25n$ routing steps when dividing a mesh into 16 submeshes, compared to $5.5n$ routing steps in case of partition into four submeshes. Moreover, the buffer requirement increases as k increases. The cost of the counting phase decreases as k increases, but the costs of move step and distribution phase increase linearly with the value k. The performance of the extended Algorithm Q becomes worse with an increasing number of submeshes, because the algorithm utilizes only channels either along a row or along a column at certain times of each recursive stage, but the number of messages in a processor is increased quadratically.

3.6.2 Extended Algorithm Q to a K Dimensional Mesh

A K-dimensional mesh is a K dimensional hypercube of processors, each of which is connected to two neighbors along each axis. So, each processor has $2K$ neighbors with which it can communicate in a single step. Algorithm Q can be extended by dividing a K dimensional mesh in the same way as that Algorithm Q divides a two dimensional mesh. Algorithm Q divides a K dimensional mesh into 2^K identical hypercubes of half size along every axis; that is, Algorithm Q divides the mesh into two halves for every axis by the plane which is perpendicular to that axis. For example, for a three dimensional n by n by n mesh, Algorithm Q will divide the mesh into eight cubes of size $n/2$ by $n/2$ by $n/2$, using three planes which are perpendicular to the x axis, the y axis and the z axis. In this section, we will briefly show that the move steps are no longer free from contention and that the number of message steps required in smoothing increases exponentially with the dimension, K.

Consider a K dimensional hypercube which consists of n^K processors. The size of the mesh is n for each side of the hypercube. Algorithm Q begins by dividing the K dimensional mesh into 2^K identical sub-hypercubes, each with $(n/2)^K$ processors. During the move step of the first recursive stage, a processor may receive $2^K - 1$ messages from the corresponding processors of each sub-hypercube. The corresponding processor in the farthest sub-hypercube is $Kn/2$ steps away, so the move step requires at least $Kn/2$ data message steps. But, unlike Algorithm Q in a two dimensional mesh, there is contention in the case of the K dimensional hypercube.

A processor can receive up to K messages every $n/2$ steps from its K corresponding neighbors which are $n/2$ steps away. Since the corresponding processors in each of 2^K sub-hypercubes may send a message to the same processor, the processor may receive up to 2^K messages, one from each sub-hypercube. In this case, the move step will take at least $2^K n/2K$ steps. But this bound is too optimistic, since we assume that a processor receives K messages every $n/2$ steps. This assumption is unfair because there are K messages $n/2$ steps away, $K(K-1)/2$ messages n steps away, $K(K-1)(K-2)/3!$ messages $3n/2$ steps away and so on.

After the move step of the first recursive stage, processors may contain up to 2^K messages. In order to recur, the smooth step rearranges messages so that each processor has at most one message. The smooth step operates on each sub-hypercube of size $(n/2)^K$. The details of the smooth step on a K dimensional mesh will be explained in Chapter 4. But now we briefly examine the time complexity of the smooth step. The smooth step consists of the counting phase and the distribution phase. The counting phase on a sub-hypercube of size $(n/2)^K$ requires $(2K-1)n/4$ integer message steps. The distribution phase consists first of $(K-1)$ row movements on the arrays of $n/2$ processors containing up to 2^K messages, and consists of a single column movement on the array of $n/2$ processors containing up to 2^K messages, two procedures which were explained in Chapter 2. If we assume that n is 2^p, each row movement requires $\frac{2^K(2^K+2)}{4(2^K+1)}\frac{n}{2}$ data message steps, while each column movement requires $\frac{2^K-1}{2^K}\frac{n}{2}$ data message steps.

$T_{Q_K}(n)$ represents the optimistic cost required to solve a routing problem on a K dimensional hypercube mesh consisting of n^K processors. By optimistic, we mean that the cost of the move steps includes only the minimum cost imposed by the channel capacity without considering unbalanced traffic. We can construct the following recursive equations, where $n = 2^p$:

$$\begin{aligned} T_{Q_K}(1) &= 0 \\ T_{Q_K}(n) &\leq T_{Q_K}(n/2) + \frac{2^K}{2K} n t_M + \frac{2K-1}{4} n t_m \\ &\quad + \{(K-1)\frac{2^K(2^K+2)}{4(2^K+1)} + \frac{2^K-1}{2^K}\}\frac{n}{2} t_M \end{aligned}$$

By solving the above equations, we get

$$\begin{aligned} T_{Q_K}(n) &\leq \frac{2^K}{K} n\, t_M + (K - \frac{1}{2}) n\, t_m + \{\frac{(K-1)2^K(2^K+2)}{4(2^K+1)} + \frac{2^K-1}{2^K}\} n\, t_M \\ &\cong O(K 2^K n\, t_M) \end{aligned}$$

Using the above equation, we find that the cost increases linearly with the size of mesh. However, the cost increases exponentially in relation to the dimension of mesh. The dimension of the mesh, K, affects the cost in two ways; one is a K factor, which is linear, and the other is a 2^K factor, which is exponential. The exponential increment of the cost comes mainly from the fact that the maximum number of a processor's messages during the smooth step increases exponentially in relation to the dimension of mesh, while channels along only a single axis are utilized in turn at any time during smoothing. We can conclude that Algorithm Q is inefficient for the higher dimensional mesh, since the cost increases exponentially.

3.6.3 Algorithm Q on an Arbitrary Square Mesh

When the size of a two-dimensional square mesh is an odd integer, the first recursive stage of Algorithm Q cannot partition the mesh into submeshes of the same size and shape. Instead, it is necessary to divide the mesh into submeshes of nearly the same size. In this section, we describe four methods to overcome difficulties caused by unbalanced quadrants and assess the costs incurred by uneven quadrants. We will show that Algorithm Q can be applied to an odd-sized mesh at the cost of additional routing steps, an additional buffer, or more complicated control structure.

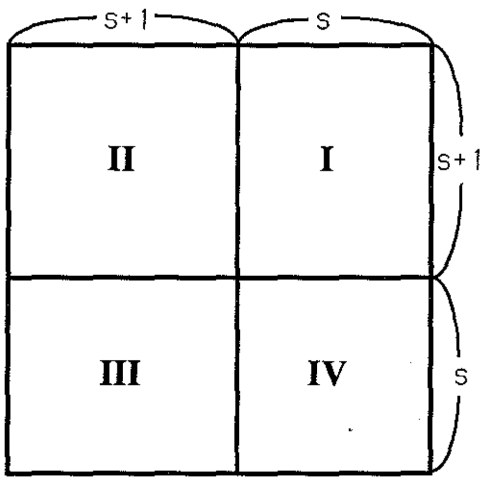


Figure 3.13: Partition of a square mesh with odd size

When the size of a square mesh is even, Algorithm Q divides the mesh into four identical quadrants; messages are sent to the same relative position of the correct quadrant during the move step. When the size of mesh is odd, Algorithm Q partitions the mesh into four quadrants, which may differ in size by one column or row or both. Because the quadrants are not the same size and shape, some processors in one quadrant do not correspond to any processor in a quadrant with a smaller size or different shape. Thus the move step must be modified in order to make it possible for messages to move to the target quadrant without conflict or with a few predictable conflicts. A modification of the move step will affect Algorithm Q in three ways. First, a processor may be required to perform additional computation to determine the target address of its message prior to the move step. Second, a processor may be required to execute a longer procedure to route messages to their target processors during the move step. Third, additional costs may be incurred from an increase in buffer requirements and the time complexity of the move and smooth steps. We will describe the target mappings and paths of messages from a bigger quadrant to a smaller quadrant and then discuss the buffer requirements and the time complexity.

When the size of the mesh is odd, that is, $2s+1$ by $2s+1$, Algorithm Q partitions the mesh into quadrants of four different sizes: $s+1$ by $s+1$, $s+1$ by s, s by $s+1$ and s by s. We index the submeshes as I, II, III, and IV, beginning from the top right and moving counter clockwise, as illustrated in Figure 3.13. We denote by processor (i,j) the processor at the i-th row and

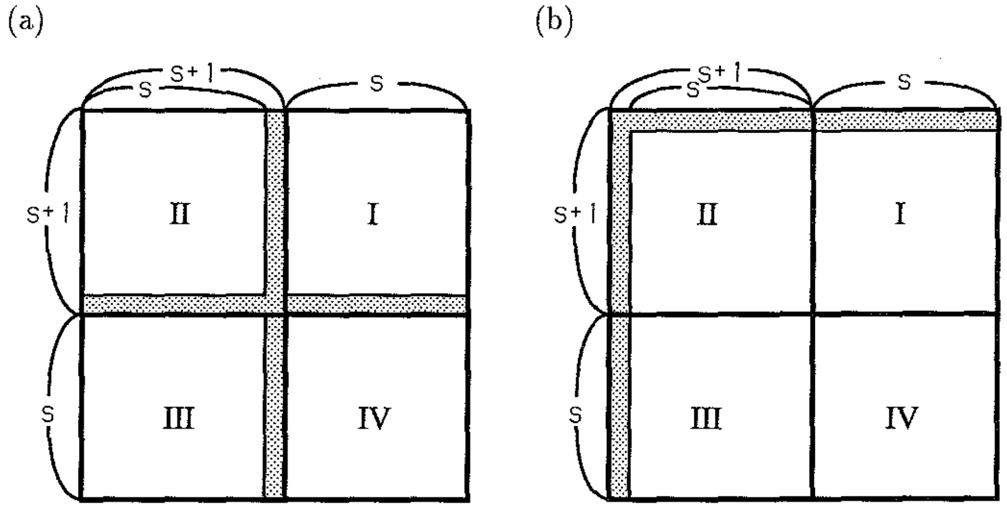


Figure 3.14: Alternatives for special regions

the j-th column of a submesh.

Since the submeshes do not have the same size, we can partition each of the bigger submeshes into two regions; the first is the region which has the same size and shape as the smallest submesh, and the second is the rest of the submesh, which is to be treated specially.

Figure 3.14 shows two ways of designating which parts of a mesh are to be special regions. The first way, illustrated in Figure 3.14.a, specifies that each processor in the middle row and column of the mesh will be part of a special region of submesh I, II or III. The second way, illustrated in Figure 3.14.b, specifies that each processor in the first row and column of the mesh will be part of a special region. The three methods of message movement we discuss all treat the subregions that are of the same size and shape as the smallest submesh in the usual way during the move step; that is, each message in these subregions is sent to the corresponding processor in the correct region. The methods differ only in which way they designate special regions, and in how the messages in the special regions are treated.

The first mapping method treats the top row and the leftmost column of a mesh as special regions, as illustrated in Figure 3.14.b. The target of a message originating from an ordinary region is s steps away along a column, a row or both. A message in the upper left-hand corner travels to the upper left hand corner of its target quadrant. Each other message originating in the leftmost column must travel $s + 1$ steps along a row if its target is in

The target of special region rows is the top row of the target quadrant. The target of special region columns is the leftmost column of the target quadrant. The gray area of submesh IV is also the target of messages from the ordinary region of each submesh. Therefore, the top (gray) row of submesh IV is the target of up to six messages.

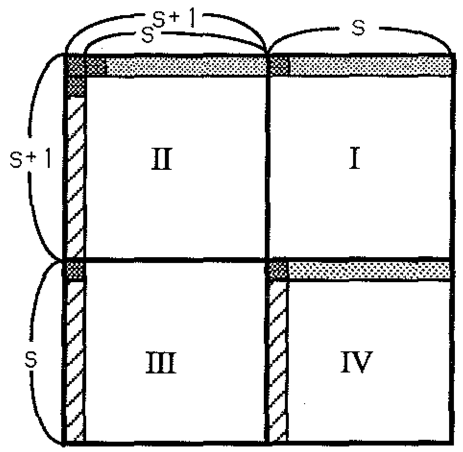


Figure 3.15: Targets of messages to Submesh IV of the first mapping

a rightmost quadrant, and s steps along a column if it must travel along a column. Each message originating from the top row must travel $s+1$ steps along a column if its target is in a lower quadrant, and s steps along a row if it must travel along a row. Figure 3.15 shows targets of messages destined to submesh IV.

During the move steps of the first mapping scheme, messages except those in the top row of the mesh travel first to their target row and then travel to their target processors. In the top row of the mesh, messages travel to target column first if necessary, and then travel along a column. Therefore, the control procedure during the move step is simple. After the move step, the top leftmost processor of submesh IV may contain up to nine messages; four from the ordinary region of each submesh, one from the top leftmost processor of submesh I, one from the top leftmost processor of submesh III, and three from processors on the upper left corner of the special region in submesh II. The longest path length is $2s+2$ steps, that is, $n+1$ steps.

After the move step, processors in the top row of submesh IV may contain six messages except the leftmost processor, which may contain nine messages. Analysis shows that the presence of up to nine messages in the leftmost processor increases the (worst case) cost of smoothing by two steps. The total additional cost of smoothing is about $0.52n$ routing steps, because processors may contain up to six messages instead of four messages, and by Theorem 2.1, these additional messages requires about $0.26n$ more routing steps for row movement in the first recursive stage.

In summary, this mapping method allows simple computation of target addresses. The additional costs are four more buffers per processor (that is, nine buffers are required in a processor,) and $0.52n + \log n$ routing steps, which include about $0.52n$ additional smoothing steps and $\log n$ additional move steps.

The second mapping method requires a more complicated computation of target addresses than the previous method, but it allows a simple control procedure during the move step. The additional costs of the method are one more buffer per processor and $O(\log n)$ routing steps.

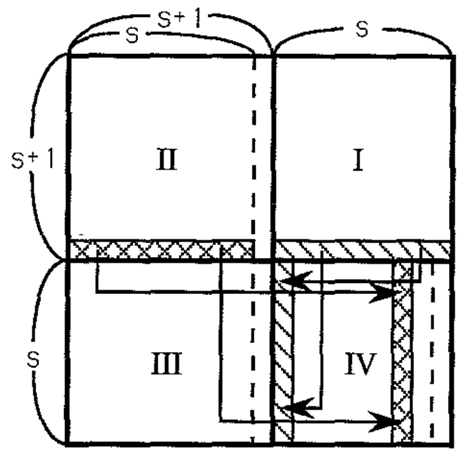
In this mapping, we divide each submesh into two regions. The first is an ordinary region which has the same shape and size as the smallest submesh. The second is a special region which includes the middle row and column of the mesh, as illustrated in Figure 3.14.a. The target of messages originating from an ordinary region is the same relative position of the ordinary region of another submesh. Now, we describe how to determine the target of messages from the special region of a bigger submesh to a smaller submesh.

We start with the target of messages to the smallest submesh IV. For $0 \le j < s$, the target of a message from processor (s, j) in submesh I is processor $(s-1-j, 0)$ in submesh IV. For $0 \le j < s$, the target of a message from processor (s, j) in submesh II is processor $(j, s-3)$ in submesh IV, as shown in Figure 3.16.a. For $0 \le i < s$, the target of a message from processor (i, s) in submesh II is processor $(i, s-2)$ in submesh IV. The target of a message from processor (s, s) in submesh II is processor $(s-1, s-1)$ in submesh IV. For $0 \le i < s$, the target of message from processor (i, s) in submesh III is processor $(i, s-1)$ in submesh IV. These mappings from special regions to submesh IV are illustrated in Figure 3.16.b.

The targets of messages heading to submesh I are determined as follows: For $0 \le i \le s$, the target of a message from processor (i, s) in submesh II is processor $(i, s-1)$ in submesh I. For $0 \le j < s$, the target of a message from processor (s, j) in submesh II is processor (s, j) in submesh I. For $0 \le i < s$, the target of a message from processor (i, s) in submesh III is processor $(i, s-2)$ in submesh I. These are illustrated in Figure 3.17.a.

The targets of messages heading to submesh III are determined as follows: For $0 \le i < s$, the target of a message from processor (i, s) in submesh II is processor (i, s) in submesh III. For $0 \le j < s$, the target of a message from processor (s, j) in submesh II is processor $(j, s-1)$ in submesh III. The target of a message from processor (s, s) in submesh II is processor $(s-1, s)$ in submesh III. For $0 \le j < s$, the target of messages from processor (s, j) in submesh I is processor $(s-1-j, 0)$ in submesh III. These are illustrated in Figure 3.17.b.

(a)



(b)

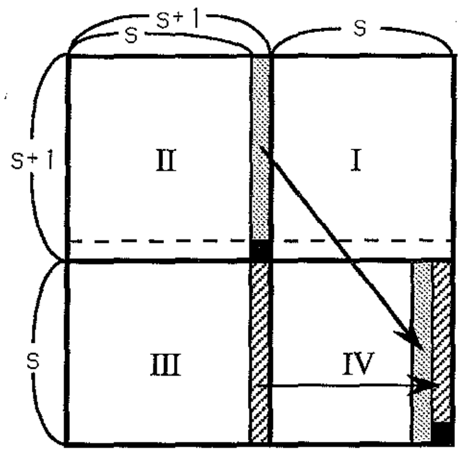
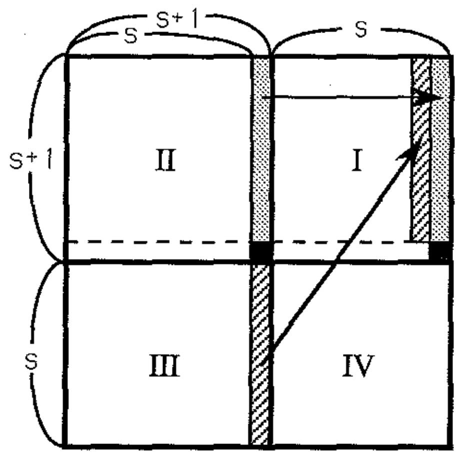


Figure 3.16: Targets of messages to Submesh IV of the second mapping

(a)



(b)

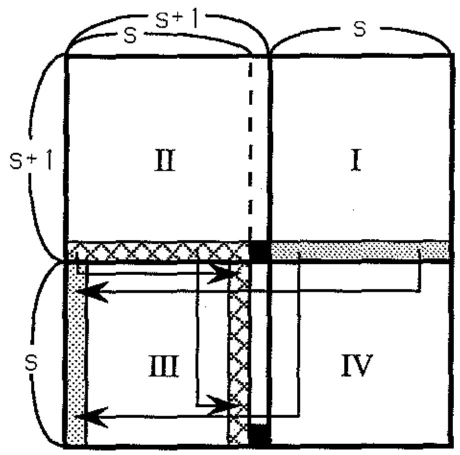


Figure 3.17: Targets of messages to submeshes I and III of the second mapping

During the move step, a message travels to its target processor along a column (if necessary) and then to the target processor along a row. Therefore, each processor must check the target address of its messages and send them through appropriate channels. Most messages travel to their targets without conflict, but there may be static conflicts in the rightmost column of subregion III during the second half of the move step. Specifically, each processor (i, s), $0 \leq i < s$, may have two messages that compete for the channel to the eastern neighbor. If there is contention, a processor will resolve it by giving priority to the message with the greater distance to travel. This contention is of little consequence, since there are at most two messages competing for any channel at any time during the move step.

A message which travels the longest path from processor $(0, 0)$ in submesh II to processor $(0, 0)$ in submesh IV will be delayed one step at processor $(0, s)$ in submesh III. This message requires $2s + 3$ routing steps to reach its target. Therefore, since $n = 2s + 1$, the move step requires two more routing steps than that of Algorithm Q on an even size mesh. Analysis shows that the cost of smoothing will increase by at most two routing steps, since after the move step most processors will contain up to four messages except processors which are the targets of messages from special regions. The target processor of messages from special regions may contain up to five messages, and in any row of submeshes, there are at most four such processors.

Since the additional costs of move and smooth steps are four routing steps, the total additional cost of move and smooth steps is $4 \log n$ routing steps. Moreover, this mapping method requires six buffers per processor, since in processors $(i, s-1)$ of submesh III and processors $(i, 0)$ of submesh IV, $0 \leq i < s$, there may be four messages that stay and two transients during the second half of the move step. In summary, this mapping method allows almost conflict-free move steps and requires only one additional buffer and $4 \log n$ additional routing steps, but the computation of target addresses is more complicated.

The third mapping scheme is more complicated in determining target addresses than the previous mapping schemes. It requires $0.3n$ additional routing steps and no additional buffers. The control procedure during the move step is almost same as the previous mapping schemes, but needs a small modification. We describe how to determine the target of a message moving from one submesh to another. There is no difficulty in mapping messages from a smaller submesh to a bigger submesh, and we will only discuss the troublesome cases.

We first treat messages moving to submesh I, shown in Figure 3.18. When

For a message from the gray area of submesh II, its target in submesh I is a processor on the same row of the gray area of submesh I. For a message from the hatched area of submesh III, its target in submesh I is a processor in the hatched area of submesh I. Submesh I and the white area of submesh II are congruent. The white area of submesh III is mapped to a proper subset of submesh I.

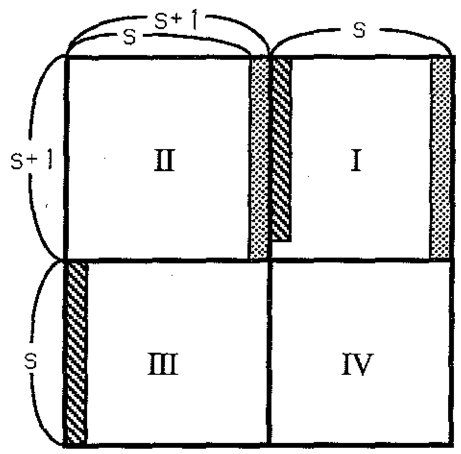


Figure 3.18: Targets of messages to Submesh I of the third mapping

a processor (i, j) in submesh II has a message to send to submesh I, the target processor is processor (i, j) in submesh I for $0 \leq i < s + 1$ and $0 \leq j < s$. The target for any message from a processor (i, s), $0 \leq i < s+1$, is processor $(i, s-1)$ in submesh I. Note that processor $(i, s-1)$ in submesh I, $0 \leq i < s+1$, is the target of two processors, $(i, s-1)$ and (i, s), in submesh II. Thus the processors in the rightmost column of submesh I will contain up to five messages after the move step.

When $0 \leq i < s$ and $1 \leq j < s + 1$ and a processor (i, j) in submesh III has a message to send to submesh I, the target processor is processor $(i, j-1)$ in submesh I. For $0 \leq i < s$, the target processor of a message in processor $(i, 0)$ in submesh III is processor $(i, 0)$ in submesh I. (Figure 3.18 illustrates targets of messages destined for submesh I.) Therefore, each processor $(i, 0)$ in submesh I, $0 \leq i < s$, will contain up to five messages after the move step.

The smallest submesh among the partitions is submesh IV. Figure 3.19 illustrates the targets of messages destined for submesh IV. When a processor (i, j) in submesh III, $0 \leq i, j < s$, has a message to send to submesh IV, the target is processor (i, j). For each processor (i, s) in submesh III, $0 \leq i < s$, the target is processor $(i, s-1)$ in submesh IV. That is, each processor $(i, s-1)$ in submesh IV is the target of both processors $(i, s-1)$ and (i, s) in submesh III. When a processor (i, j) in submesh I, $0 \leq i, j < s$, has a message to send to submesh IV, the target is almost always processor (i, j) in submesh IV. There is one exception; the target of the message contained in processor $(s-1, s-1)$ in submesh I is processor $(s-2, s-2)$ in submesh IV, rather than processor $(s-1, s-1)$. For each processor (s, j) in submesh I,

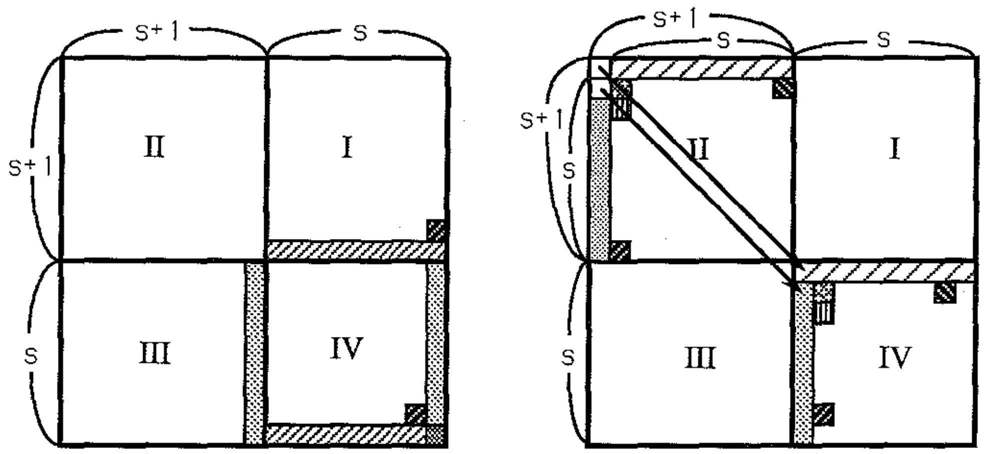


Figure 3.19: Targets of messages to Submesh IV of the third mapping

$0 \le j < s$, the target is processor $(s-1, j)$ in submesh IV. Finally, messages from the biggest submesh II can be mapped to the processors of the smallest submesh IV as follows. When a processor (i, j) in submesh II contains a message destined for submesh IV, its target processor is processor $(i-1, j-1)$ for $1 \le i, j < s+1$. There are four exceptions; the target of processor $(1,1)$ in submesh II is processor $(1,1)$ in submesh IV instead of processor $(0,0)$, the target of processor $(2,1)$ in submesh II is processor $(2,1)$ in submesh IV instead of processor $(1,0)$, the target of processor $(s,1)$ in submesh II is processor $(s-2,1)$ in submesh IV instead of processor $(s-1,0)$, and the target of processor $(1,s)$ is processor $(1,s-2)$ in submesh IV instead of processor $(0,s-1)$. Each processor $(0,j)$ in submesh II, $1 \le j < s+1$, sends messages to processor $(0,j-1)$ in submesh IV. Each processor $(i,0)$, $2 \le i < s+1$, sends messages to processor $(i-1,0)$ in submesh IV. For processor (0,0) in submesh II, the target is processor $(0,0)$ in submesh IV. For processor (1,0) in submesh II, the target is processor (1,0) in submesh IV. The above mapping provides each processor in submesh II with a target for any message that has a destination in submesh IV. Moreover, no processor in submesh IV will contain more than five messages after the move step.

The mapping from originating processor to target processor for each message allows almost conflict-free movement of messages during the move step. Each processor sends messages along columns first then along rows. The only restriction is that the messages are not allowed to travel south during the time interval from $s+1$ through $2s$. Thus, each message that has to travel only south stays at the current processor (one short of its target) until time

$2s$ and each message that has to move both south and in another direction travels in the direction other than south first, and then takes its final step south. This movement avoids the congestion that would result from having messages from two rows in the bottom row of the mesh.

Since the message from processor (0,0) in submesh II destined for submesh IV must travel $2s + 2$ steps and will be delayed one step at processor $(s, 0)$ in submesh I during the movement, the move step on the $2s + 1$ by $2s + 1$ mesh requires $2s + 3$ routing steps; that is, $n + 2$ data message steps. During the move step, no processor requires more than five buffers at any time. Since the additional cost of each move step is two routing steps regardless of the size of mesh, the effect on the overall routing cost is at most $2 \log n$ data message steps. Moreover, no processor needs an additional buffer even though some processors may contain up to five messages after the move step.

Although the number of messages in some processors in a small submesh is five instead of four, the total number of messages in the submesh does not exceed the number of processors in the submesh. Therefore, the procedure of smoothing is not affected except that some additional routing steps are required to handle rows in which processors may contain five messages. That is, in the top and bottom rows of submeshes III and IV, processors may contain five messages instead of four after the move step. On an n by n mesh, row movement of the distribution phase of smoothing requires a total of $1.5n$ data message steps when n is not a power of two, compared to $1.2n$ data message steps when n is a power of two. Column movement does not incur additional costs since at most two processors per column may contain five messages; other processors will contain up to four messages. Therefore, the cost of column movement during the distribution phase does not increase.

In summary, we can handle the move and smooth steps on an n by n mesh, where n is not a power of two, for a cost of $2 \log n + 0.3n$ additional data message steps. But the control structure of each recursive stage becomes more complicated, and the computation of target addresses during move steps becomes more complex.

We have described three mapping methods to implement Algorithm Q on a square mesh with odd size, and discussed the additional costs of each method. Each method requires different additional costs incurred by the modification of move step. Other methods are easy to conceive, and in practice, the user of Algorithm Q can choose an appropriate method according to the constraints of his network configuration.

3.6.4 Algorithm Q on a Rectangular Mesh

In this section, we describe modifications that will enable Algorithm Q to be applied to a rectangular mesh. We first sketch the move and smooth steps and examine the time complexity and the buffer requirement when each dimension is a power of two. Next, we describe how to apply Algorithm Q to a rectangular mesh with an arbitrary number of rows and columns.

We describe Algorithm Q on a rectangular mesh with r rows and c columns, where both r and c are powers of 2 and $r > c$. Algorithm Q begins by partitioning a mesh into four congruent quadrants with size $r/2$ by $c/2$. A message in the correct quadrant does not travel during the move step. Otherwise, a message travels from its current location to the processor which has the same relative position in the correct quadrant. This movement is accomplished by moving 0 or $r/2$ steps along a column of the mesh, and then 0 or $c/2$ steps along a row. Thus, a message may remain in place, or it may travel $c/2$ steps along a row, or it may travel $r/2$ steps along a column, or it may travel $r/2$ steps along a column followed by $c/2$ steps along a row. When messages that must travel along both columns and rows arrive at the target row, all messages that travelled only along a column or row have already arrived at their target. Consequently, there is no conflict during the move step. During the move step, a processor may contain up to five messages. The move step requires $(r + c)/2$ data message steps.

At the end of the move step, a processor may contain up to four messages; one from the current quadrant and three from the other three quadrants. The smooth step is applied to each of the four quadrants simultaneously and independently to restore the condition that each processor contains at most one message. The smooth step on a $r/2$ by $c/2$ rectangular mesh consists of the same counting and distribution phases described in Section 3.3.1 and Section 3.3.2 respectively. Counting on a $r/2$ by $c/2$ rectangle requires $r/4 + c/2 - 2$ integer message steps. Distribution consists of row movement followed by column movement. Since $b = 4$, row movement in an array of $c/2$ processors requires $0.6c$ data message steps (by Theorem 2.1), and column movement in an array of $r/2$ processors requires $3r/8$ data message steps (by Theorem 2.3). This ends the first recursive stage.

The recursion terminates when each quadrant contains a single column of processors. When a quadrant has a single column of r/c processors, the smoothing step can be skipped and each message can travel directly to its destination without conflict. The last move step requires $r/c-1$ data message steps, since a quadrant consists of r/c processors.

$T(r,c)$ represents the cost of solving a routing problem with Algorithm Q on an r by c rectangular mesh. The complexity of Algorithm Q on an r by c rectangular mesh is bounded by the following recurrence equations, when $r = 2^s$, $c = 2^t$, and $s > t$:

$$\begin{aligned} T(r/c, 1) &= r/c - 1 \\ T(r,c) &\leq T(r/2, c/2) + \frac{r+c}{2} t_M + (\frac{r}{4} + \frac{c}{2} - 2)\, t_m + (\frac{3r}{8} + 0.6c) t_M \end{aligned}$$

Solving the above recurrence equations, we get

$$\begin{aligned} T(r,c) \leq\ & (r - \frac{r}{c} + c - 1) t_M + (\frac{r}{2} - \frac{r}{2c} + c - 1 - 2\log c) t_m \\ & + (\frac{3r}{4} - \frac{3r}{4c} + 1.2c - 1.2) t_M \end{aligned}$$

Dropping negative terms strengthens the inequality to the following:

$$\begin{aligned} T(r,c) &\leq (r + c) t_M + (\frac{r}{2} + c) t_m + (\frac{3r}{4} + 1.2c) t_M \\ &= (1.75r + 2.2c) t_M + (\frac{r}{2} + c) t_m \end{aligned}$$

When either r or c is not a power of two, the number of columns or rows of a submesh is not an even number at some recursive stage. In Section 3.6.3, we showed how Algorithm Q can partition a mesh into submeshes of nearly the same size. Specifically, the algorithm can divide a mesh into regions whose numbers of row and columns differ by at most one. The same technique can be used when the number of rows or columns of a rectangular mesh is an odd number. The target address of each message can be determined by a mapping similar to that developed for a square mesh and described in Section 3.6.3.

The move step can be modified according to the mapping. Using Algorithm Q on a rectangular mesh of odd size requires more message routing steps during both the move step and row movement of the smooth step. The additional cost incurred during the smooth step is a total of $0.3c$ data message steps; the cost of row movement of the smooth step at the first recursive stage increases from $0.6c$ to $0.75c$, since processors in some rows may contain five messages after the move step, which in the final solution gives a total of $1.5c$ instead of $1.2c$. The additional cost of the move step is a total of $2 \log c$ data message steps, since two more steps are necessary at each move step. We can conclude that Algorithm Q is applicable to a two dimensional mesh of any size, with an additional cost of no more than $0.3c$ data message steps.

Chapter 4

Algorithm H

The second algorithm we treat, which we call H (for 'Halves'), has a somewhat worse time complexity than Algorithm Q, but it requires only three buffers in each processor. Sections 4.2 and 4.3 describe the move and smooth steps of Algorithm H, and Section 4.4 describes the time complexity and buffer requirements in detail. Section 4.5 shows that broadcasting problems do not change the time complexity of Algorithm H. Section 4.6 discusses extensions of Algorithm H.

4.1 Overview

Algorithm H can be applied to a rectangular mesh of arbitrary size. But for simplicity's sake, we will describe Algorithm H for an n by n square mesh where $n = 2^p$. In the special case of a square mesh, Algorithm H first partitions the mesh into two rectangular regions, each of which is half the mesh. The move and smooth steps are then performed on these two non-square regions, after which every message is in the correct half of the array, and every processor has at most one message. The algorithm then partitions each of these two non-square regions into two square regions and performs move and smooth steps on those regions. Then the algorithm recurs. Like Algorithm Q, each stage of this algorithm moves each message into the correct subquadrant of the current partition. This algorithm, however, requires two separate move and smooth steps for each recursive stage. The first move step will move a message either 0 or $n/2$ positions along its current row; the second move step will move each message either 0 or $n/2$ steps along its current column. Thus, the maximum distance moved by any message in stage one of this algorithm is n, the same as for Algorithm Q.

Algorithm H differs from Algorithm Q in the complexity of its smoothing step. In Algorithm H, at the end of each move step, a processor may contain up to two messages, one from before the move step and the other a message that was moved from the corresponding processor in another region either along a column (during the first move step of each recursive stage) or along a row (during the second move step of each recursive stage). We will show that the first smoothing step of the first stage divides an n by n mesh into halves and requires $2n$ steps, while the second smoothing step of the first stage divides each of the halves into quarters of the original mesh and requires $1.5n$ steps. Thus, adding in the n steps required for the move, the complexity of the algorithm is given by

$$\begin{aligned} T(1) &= 0 \\ T(n) &\leq T(n/2) + 4.5n \quad (\text{for } n = 2^p) \end{aligned}$$

giving a solution of $T(n) \leq 9n$.

4.2 Move Steps of Algorithm H

Algorithm H is based on the successive partitioning of a mesh into two disjoint regions; at each recursive step, it divides a square mesh into two rectangular halves of the mesh, and then divides each of the two halves into two square submeshes of half size. By two successive partitionings at each (recursive) stage, Algorithm H recurs on four half-size square meshes simultaneously. This section describes the two move steps of the first recursive stage on an s by s square mesh.

Consider a square mesh with s rows and s columns, where $s = 2^p$. Each processor in the mesh has at most one message which has a distinct destination address. (We are now considering permutation problems only. We will consider restricted broadcasting problems later in Section 4.5.) The first move step begins by partitioning the s by s mesh into left and right halves that are s by $s/2$ rectangular meshes. Each processor in the mesh can determine which half it belongs to and which half contains the destination address of its message. The objective of the first move step is to rearrange the messages so that all messages are contained in the correct half, which contains the destination address of each message.

As in the move step of Algorithm Q, the messages that must travel should be moved in a way which prevents contention and in which the number of messages in a processor is predictable after the move step. When a processor has a message which is not in the correct half, the processor sends the message to the corresponding processor which is at the same relative position in the correct half (that is, the other half of the mesh). The corresponding processor in the correct half is $s/2$ steps away along the current row. When, however, a processor has a message which is in the correct half, the processor keeps the message during the first move step. Once each processor determines the target processor of its message by examining the destination address, messages which are not in the correct half travel simultaneously along their current row by $s/2$ steps. During this movement, there is no contention among messages, because there was only at most one message in a processor initially, and messages begin to travel simultaneously along the current row and stop at the same time after $s/2$ steps. During the first move step, only links along rows are active. Links are bidirectional, so no contention occurs since messages travel in lock step in both directions. The time required for the first move step is $s/2$ data message steps.

After the first move step, each message is contained in the half which contains the destination address of the message. No processor contains more than two messages. The first smoothing step restores the condition that each processor contains at most one message, so that no processor will contain more than two messages after the second move step. (Without smoothing, two messages may compete for a channel during the second move step, and up to four messages may be contained in a processor after the second move step. Therefore, except that the time bound of move step increases due to delays, Algorithm H would be nearly equivalent to Q.) The first smoothing step is performed on both of the rectangular halves simultaneously and will be explained in Section 4.3.

The second move step begins by partitioning each half of the mesh into two square quarters, each of whose size is $s/2$ by $s/2$, a division that results in a total of four half-size ($s/2$ by $s/2$) square submeshes. The second move step is applied to the two halves simultaneously, and the movements in the two halves are independent. Because the second move step occurs in the same way in both halves of the mesh, we describe the second move step only for the left half of the mesh.

During the second move step, each processor determines the quarter which contains the destination address of the message that the processor contains. If a processor contains the message which is in the correct quarter, that processor keeps the message during the second move step. If a processor contains a message which is not in the correct quarter, it sends the message to the processor at the same relative position in the other quarter. The corresponding processor is $s/2$ steps away along the current column. Note that the destination of a message is either in the lower quarter of the left half or in the upper quarter of the left half. It never belongs to the quarter on the lower right or on the upper right.

The movement of messages begins in all processors of the mesh simultaneously. The target address of each message is either the originating processor itself or the processor which is $s/2$ steps away along the current column. No contention occurs during the second move step, because each processor has at most one message initially and all messages travel in lock step along a column, either up or down, and stop moving at the same time. Thus, the second move step takes $s/2$ data message steps to rearrange messages so that each message is contained in the correct quarter.

After the second move step, each message is contained in the quarter which contains its destination address. No processor contains more than two messages; the two possible messages consist of one that has stayed during the second move step and another that came from the other quarter in the same half. Now the second smoothing step follows; it will be explained in Section 4.3.

During both move steps, no processor contains more than three messages at any time; one which stays there and two transients. During the first move step, a processor contains up to two transient messages, since messages are moving both left and right along rows. During the second move step, a processor contains up to two transient messages, since messages are moving both up and down along columns.

The following theorem summarizes this section.

Theorem 4.1 *The move steps of the first recursive phase on an s by s mesh will take a total of s (data message) steps, and no processor will contain more than three messages at any time. Specifically, the first and second move steps take $s/2$ (data message) steps each. After each move step, a processor may contain up to two messages.*

4.3 Smooth Steps of Algorithm H

Recall that each recursive stage of Algorithm H has two separate move and smooth steps, which are applied to the submeshes of different sizes. At the end of each move step, a processor may contain up to two messages, but the number of messages in each subregion does not exceed the number of processors in the subregion. We will describe the two smooth steps of the first recursive stage on an s by s square mesh. After the first move step, the first smooth step is performed on two disjoint s by $s/2$ rectangular submeshes simultaneously. After the second move step, the second smooth step is performed on four disjoint $s/2$ by $s/2$ square meshes simultaneously. Both smooth steps use basically the same procedure as Algorithm Q, except that the maximum number of messages in a processor is at most four in Algorithm Q.

4.3.1 First Smooth Step

The first smooth step is performed on each half of an s by s mesh simultaneously to establish the precondition that each processor contains at most one message and that each message is in the correct region. Since the smoothing operations on two halves are independent and simultaneous, we will only describe the first smooth step for the left half of the mesh.

The first smooth step takes two rectangular halves which have s rows and $s/2$ columns and rearranges the messages in the two halves at the same time. Assuming that s is $2q$, we rephrase the first smooth step as follows:

> In a rectangular mesh with $2q$ rows and q columns, a processor has at most two messages initially, and the total number of messages does not exceed $2q^2$. The smooth step rearranges the messages in the mesh so that no processor contains more than one message after the movement.

Like the smooth step of algorithm Q, smoothing on the rectangular mesh is done in two phases, the counting phase and the distribution phase. The counting phase informs each processor in the mesh about the number of messages that each processor will receive from and send to its neighbors during the distribution phase. The distribution phase rearranges the messages as specified by the counting phase. During the distribution, the messages travel along rows first, a movement which is called row movement, and then along columns, a movement which is called column movement.

Counting is performed by the same procedure described in Section 3.3.1 of Chapter 3. The only difference is that the number of rows in Algorithm H is twice as large as the number in Algorithm Q. We briefly summarize the procedure and evaluate the number of time steps required.

To speed up the counting phase, we divide the rectangular mesh into the upper and lower square submeshes, each of which consists of q rows and q columns. Counting is performed on two square submeshes simultaneously, and we only describe the procedure for the upper square submesh.

1. Simultaneously, for each row, information flows from the left end of each row to the right of the row. At the same time, corresponding information flows from the right end of each row to the left end of the row. These two information waves provide the basis for computing the total number of messages in each row and the number of messages to the left of each processor in its current row. These integer message waves require $q-1$ steps, so the first part of the counting phase takes $q-1$ integer message steps.

2. A second wave of messages now occurs in each column, with each processor sending its southern neighbor the total number of messages in all rows above the current row. This wave originates in the top row. Since counting occurs concurrently in the upper and lower square submeshes, this wave requires $q-1$ integer message steps.

Lemma 4.1 *The counting phase on a $2q$ by q rectangular mesh requires $2q-2$ integer message steps.*

Proof: The last wave of Step 2 starts in the two outermost columns $q-1$ steps after Step 1 begins. Because Step 2 requires $q-1$ integer messages steps, the counting of the upper or lower half requires $2q-2$ integer message steps. Since counting is performed on the lower and upper square submeshes simultaneously, it requires the same number of time steps as the counting of the upper square mesh. □

Once counting is completed on the $2q$ by q rectangular mesh, distribution occurs in two phases. In the first phase, which we call row movement, the messages in the mesh are rearranged along their current rows so that no column of the mesh contains more than $2q$ messages. In the second phase, which we call column movement, the messages travel along their current columns so that no processor has more than one message. Row movement and column movement are done as in Algorithm Q, described in Section 3.3.2. The only difference is that there are no more than two messages per processor in Algorithm H, while there are up to four messages per processor in Algorithm Q.

We briefly summarize row movement and column movement on the $2q$ by q rectangular mesh in which each processor has no more than two messages initially. Before row movement, each processor has to compute the number of messages which will flow through its channels. The counting phase provides sufficient information for each processor to calculate the number of messages that must move over each channel. Each processor follows the procedure described in Section 3.3.2 to determine the final configuration for row movement. Once the computation is complete in each processor, message movement begins. For each row, there are q processors, each of which has up to two messages initially. By Theorem 2.1 of Chapter 2, row movement in each row requires at most $q-1$ data message steps. Since row movement is performed simultaneously in each row, row movement of a $2q$ by q rectangular mesh takes $q-1$ data message steps.

After row movement, each column of the rectangular mesh has no more than $2q$ messages, and no processor has more than two messages. When each processor determines its role in row movement, it also determines where its messages will be sent after row movement. The details are exactly same to those of Algorithm Q in Section 3.3.2 except that in Algorithm Q, a processor may contain up to four messages. For each column, there are $2q$ processors, the total number of messages in a column is no more than $2q$, and the target of each message is already known. By Theorem 2.3 of Chapter 2, column movement takes at most q data message steps. Since column movement in each column is independent and performed simultaneously, column movement on the $2q$ by q rectangular mesh requires at most q data message steps.

During row movement, no processor contains more than two messages by Theorem 2.2 of Chapter 2, and there are no more than two messages in a processor after row movement. During column movement no processor contains more than two messages, since the number of messages does not increase except when an empty processor receives a single message from its northern or southern neighbor. During the distribution phase, therefore, no processor contains more than two messages at any time.

Lemma 4.2 *The distribution phase takes $2q-1$ data messages steps on the $2q$ by q rectangular mesh when each processor in the mesh contains at most two messages initially. Specifically, row movement takes $q-1$ data message steps, and column movement takes q data message steps. During distribution, no processor holds more than two messages at any time.*

Proof: By Theorem 2.1, row movement takes $q-1$ data message steps since $b=2$ and $c=q$. By Theorem 2.3, column movement takes q data message steps since $b=2$ and $r=2q$. The number of messages in a processor during the movements does not exceed the maximum number of messages in the processor initially. By Theorem 2.2, no processor contains more than two messages, since $b=2$. □

After column movement, a processor in the left rectangular submesh contains at most one message. With the same smoothing operation, a processor in the right rectangular submesh contains at most one message. The smooth step is performed on the two rectangular submeshes simultaneously. After the first smooth step, each processor in the mesh contains at most one message, and each message is contained in the correct half of the mesh. The following theorem summarizes the time complexity of the first smooth step.

Theorem 4.2 *The smooth step on a $2q$ by q rectangular mesh takes $2q-1$ integer message steps and $2q-1$ data message steps when each processor in the mesh contains at most two messages initially. Specifically, the counting phase takes $2q-1$ integer message steps and the distribution phase $2q-1$ data message steps. At any time during the smooth step, no processor requires more than two buffers.*

4.3.2 Second Smooth Step

The second move step began with partitioning each of two rectangular halves into two $s/2$ by $s/2$ square submeshes. After the second move step, each processor may contain up to two messages. But each message is contained in the quarter which contains its destination address. In order to recur, the second smooth step rearranges the messages in each of those four quarters so that no processor contains more than one message. The second smoothing operation is performed on each $s/2$ by $s/2$ quarter independently and simultaneously.

Each configuration of the second smooth step of Algorithm H can occur in the smooth step of Algorithm Q. It follows that the same smoothing process is adequate, and it is, in fact, used without change. The only difference is that the maximum number of messages in a processor in Algorithm H is two rather than four. We will now briefly analyze the time complexity of the second smooth step of Algorithm H.

The counting phase is exactly the same as that of Algorithm Q. Since the number of messages in a processor does not affect the counting procedure, the time complexity is the same; that is, by Lemma 3.1, the counting phase of the second smooth step in Algorithm H takes $3s/4-2$ integer message steps.

The distribution phase is the same except that the number of messages in a processor is at most two rather than four as in the case of Algorithm Q. Since the number of messages initially in a processor affects the time complexity, we need to examine the number of time steps required for the row and column movements.

In the case of Algorithm H, for each row, there are q processors, and each processor contains no more than two messages, so row movement takes at most $q-1$ data message steps by Theorem 2.1 of Chapter 2. After row movement, there are at most two messages in each processor. For each column, there are q processors, and each processor contains at most two messages before column movement begins. Column movement takes at most $q/2$ data message steps. (We assume that q is even for simplicity's sake.) During row and column movement, by Theorem 2.2 no processor contains more than two messages at any time, since the maximum number of messages in a processor prior to each movement is two.

The following theorem summarizes the requirements during the second smooth step.

Theorem 4.3 *The second smooth step in Algorithm H for a q by q quarter takes $3q/2-2$ integer message steps and $3q/2-1$ data message steps, where q is even. At any time during the second move step, no processor contains more than two messages.*

Proof: Immediate from Theorems 2.1, 2.2, and 2.3.

The following theorem concludes this section.

Theorem 4.4 *The two smooth steps on an s by s mesh take a total of $7s/4-4$ integer message steps and $7s/4-2$ data message steps, where s is a multiple of four. No more than two buffers are required at any time during the smooth steps.*

Proof: From Theorem 4.2, the first smooth step takes $s-2$ integer message steps and $s-1$ data message steps. From Theorem 4.3, the second smooth step takes $3s/4-2$ integer message steps and $3s/4-1$ data message steps. The theorem follows immediately.

4.4 Time Complexity and Buffer Requirement

In this section, we show that Algorithm H applied to an n by n square mesh requires $9n$ message steps, consisting of $5.5n$ data message steps and $3.5n$ integer message steps. At any time during routing, Algorithm H requires at most three buffers per processor regardless of the size of the mesh. In this section, we derive the requirements of Algorithm H on an n by n square mesh where n is a power of two. However, the results of this section can be generalized to obtain the complexity of Algorithm H for an arbitrary rectangular mesh.

To compare the complexities of Algorithms Q and H, we will use the same assumptions as in Section 3.4 (stated in page 61). We denote by $T_H(n)$ the cost of solving a routing problem with Algorithm H on an n by n square mesh. We will now establish a recurrence relation for $T_H(n)$. The first recursive stage of Algorithm H on an n by n square mesh begins by partitioning the mesh into two rectangular halves, and the first move step follows. Then the first smooth step is performed on two n by $n/2$ halves simultaneously, which leaves two rectangular halves that constitute the independent routing problems. The algorithm then partitions each of those two rectangular regions into two square meshes and performs the second move step on two n by $n/2$ meshes simultaneously. Then, the second smooth step ends the first recursive stage by rearranging messages in the four square meshes simultaneously. Algorithm H then recurs on four submeshes of half size simultaneously. Thus, the following recurrence equation holds if n is a power of two.

$$\begin{aligned} T_H(n) \ \leq\ & (\text{cost of the first move step on the } n \text{ by } n \text{ mesh}) \\ & +(\text{cost of the first smooth step on the } n \text{ by } n/2 \text{ mesh}) \\ & +(\text{cost of the second move step on the } n \text{ by } n/2 \text{ mesh}) \\ & +(\text{cost of the second smooth step on the } n/2 \text{ by } n/2 \text{ mesh}) \\ & +T_H(n/2) \end{aligned}$$

By Theorem 4.1, the cost of the first move step and the second move step is a total of n data message steps, that is, $n\,t_M$. By Theorem 4.2, the first smooth step on the n by $n/2$ mesh takes $n-2$ integer message steps and $n-1$ data message steps, that is, $(n-2)t_m+(n-1)t_M$. By Theorem 4.3, the second smooth step on the $n/2$ by $n/2$ mesh takes $3n/4-2$ integer message steps and $3n/4-1$ data message steps, that is, $(3n/4-2)t_m+(3n/4-1)t_M$. The following recurrence equations are obtained.

$$\begin{aligned} T_H(1) &= 0 \\ T_H(n) &\leq T_H(n/2) + n\,t_M + (n-2)t_m + (n-1)t_M \\ &\quad +(3n/4-2)t_m + (3n/4-1)t_M \\ &\leq T_H(n/2) + 2.75n\,t_M + 1.75n\,t_m \end{aligned}$$

Solving the equations, we get

$$T_H(n) \ \leq\ 5.5n\,t_M + 3.5n\,t_m \quad (\text{for } n = 2^p)$$

During the move steps of Algorithm H, no processor contains more than three messages at any time, regardless of mesh size, by Theorem 4.1. During the smooth steps, no processor contains more than two messages at any time, regardless of mesh size, by Theorems 4.2 and 4.3. So Algorithm H requires only three buffers in a processor, regardless of mesh size.

Theorem 4.5 *If n is a power of two, then Algorithm H on an n by n mesh requires three buffers, $5.5n$ data message steps and $3.5n$ integer message steps to route any permutation problem.*

Proof: Immediate from the argument above.

4.5 Restricted Broadcasting with Algorithm H

Algorithm H can solve broadcasting problems without additional buffers or routing costs because messages are duplicated appropriately during the move steps, and because new copies introduce no contention. Since each processor will receive at most one message in the final configuration, message copying does not introduce any change into the smooth steps. Even though the messages are duplicated during the move steps, the new copies do not violate the condition that the number of messages in a subregion is no greater than the number of processor in the subregion. In this section, we describe the method by which during the move steps, a processor makes copies of a message with multiple destination addresses and then show that the new copies of messages do not alter the complexity of the smooth steps.

Recall that for a broadcasting problem, each processor initially has at most one message with a single or several destination address(es), and that each destination address of a message is distinct. Therefore, no processor will receive more than one message in the final configuration. As in Algorithm Q, we assume that the destination address of a message is contained in the message packet and that each processor can examine all the destination addresses of a message without any additional costs.

Prior to each move step, Algorithm H partitions a mesh into two regions with the same size and shape. Then, each processor examines the destination address of its message and determines whether the message will move to the other region of the mesh or will stay in the current processor. In a broadcasting problem, a message has a list of one or more destination addresses. For each message in a processor, all destinations of the message are contained in the current region, or all entries are contained in the other region, or some entries are contained in the current region and the rest of the entries are contained in the other region. If all destination addresses are in the current region, the message stays in the current processor as if it had a single destination address contained in the current region. If all destination addresses are contained in the other region, the message will move to the processor at the same relative position of the other region along the current row or column during the move step, as if it had a single destination address contained in the other region. If some destination addresses are contained in the current region and the rest are in the other region, the processor duplicates the message, keeps one copy, and sends the other copy to the corresponding processor in the other region. In short, a processor examines destination addresses of each message and either treats the message as one with only a single destination address or duplicates the message, keeps one copy and sends the other copy as if each of them were a message with a single destination address.

We now describe in detail how a processor copies and sends its message during the first recursive stage on an n by n square mesh. Recall that each processor has at most one message and that this condition is restored after each move step. The first move step begins with partitioning the mesh into left and right halves of rectangular shape. Each processor then examines the destination addresses of its message. If the destinations of the message are contained either only in the left half or only in the right half, the processor keeps the message or sends it to the other half. If the destinations of the message occur in both halves of the mesh, the processor duplicates the message. Then, the current processor keeps one copy of the message during the first move step and sends another copy to the corresponding processor in the other half along a row, which takes $n/2$ data message steps. During the first move step, no contention occurs since all messages that travel begin to move and stop moving at the same time. Note that in each processor, there is at most one copy of a message which has to travel during the first move step.

The second move step is similar to the first move step. Prior to the second move step, Algorithm H divides each of two rectangular halves into two square quarters. Each processor examines the destination addresses of each message and determines which quarters contain destinations of the message. If the destination addresses of a message span only a single quarter, the message is sent to the correct quarter. If the destination addresses span both quarters of the same half, the message is duplicated. One copy is sent to the other quarter in $s/2$ steps along a column while the other copy is kept in the current processor. It is clear that no contention occurs during the second move step for the same reason as in the first move step.

An example is given in Figure 4.1. In Figure 4.1.b, messages B and E are duplicated, and one copy of each message is sent to the other half. Messages A, G and H are not duplicated since all the destination addresses are contained in the same half of the mesh. Messages A, F and H do not

(a) **Destination Table**

Message	A	B	C	D	E	F	G	H	J
Desti-	(2,0)	(2,1)	(2,2)	(2,3)	(0,1)(6,7)	(1,0)	(0,2)	(0,6)	(1,7)
nations	(5,3)	(6,6)			(3,4)(5,0)		(6,3)	(1,6)	
		(7,3)			(3,5)(5,1)			(2,6)	

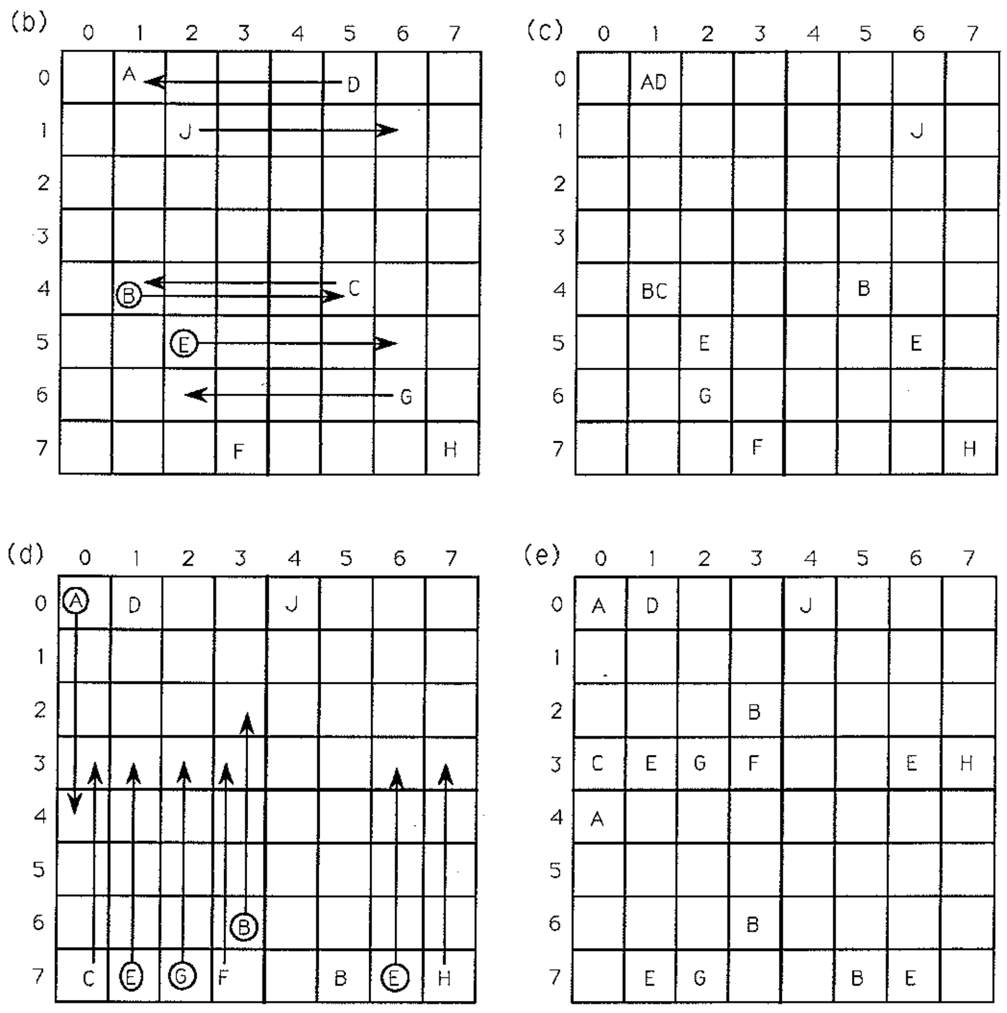


Figure 4.1: Broadcasting with Algorithm H

move since they are in the correct half. Messages C and J are not duplicated but travel to the processors on the right half. Message G travels to the left half as a single message even though it has two destination addresses. In Figure 4.1.d, messages A and G are duplicated, and one copy of each message is sent to the other quarter in the same half. Message H is not duplicated again, since all destination addresses are contained in the upper right quarter. Messages E in both halves are duplicated and sent to the proper quarters. But message B in the lower right quarter is not duplicated, while message B in the lower left quarter is duplicated and sent to the proper quarters. Messages D and J stay, and messages C and F are sent to the processors in the correct quarter.

After the move steps, no processor will contain more than two messages, these being one which did not move and one which has come from the other region. Since all destination addresses of messages in the same region are distinct, the number of destination addresses of messages in a region does not exceed the number of processors, even though some messages have multiple destination addresses.

Although messages must be copied during the move steps, no additional communication cost is incurred over that of partial permutation problems. There are no additional costs for the smooth steps. Hence, the time complexity of Algorithm H is not changed for any broadcasting problem if we assume the costs of examining address lists and copying are negligible compared to the communication costs. The new copies of duplicated messages also do not require additional buffers, since only one copy travels between any two regions. In practice, however, the multiple destination addresses may require a longer message format. The longer message may require more communication time and bigger buffers. We did not consider these costs, since they can be accommodated by adjusting the data message travelling time and the buffer size.

Theorem 4.6 *Algorithm H can solve any restricted broadcasting problem. No additional buffers are required, although buffers may be larger since message packets must contain a list of destination addresses. No additional communication steps are required, although steps may be larger to transmit the larger message packets. In addition, prior to each move step, a processor must examine the list of addresses of its message, and possibly make a copy. Thus, on an n by n mesh, Algorithm H can solve any restricted broadcasting problem with three buffers per processor in* $5.5n$ *data message steps and* $3.5n$ *integer message steps if routing costs are the dominant expense.*

Theorem 4.6 can be generalized to a mesh with an arbitrary size. We will describe Algorithm H for an arbitrary size mesh later in Section 4.6.4. On any size or shape mesh, Algorithm H can solve restricted broadcasting problems with the same costs as solving (partial) permutation problems.

4.6 Extensions of Algorithm H

Algorithm H partitions a two-dimensional mesh into two submeshes successively at each recursive stage. It can be extended in three independent ways. First, the algorithm can be extended to dimensions higher than two. A second extension is to partition a mesh into k strip submeshes of the same shape and size. Third, the algorithm can also be generalized to apply to dimensions that are not powers of two. In this section, we describe several illustrative extensions and analyze the complexity of each, although presentation of a completely general result is not included.

In Section 4.6.1, we extend Algorithm H to a three dimensional mesh, and in Section 4.6.2, we generalize it to a K-dimensional mesh. We evaluate the costs of these extensions. These sections assume a hypercube mesh of size n, where n is a power of two.

In Section 4.6.3, we explain an extension of Algorithm H which partitions a two dimensional mesh into k strip submeshes vertically and then horizontally. This extended version requires fewer routing steps as k increases, but the buffer requirement increases. We assume that the mesh size n is a power of k.

In Section 4.6.4, we describe a method to apply Algorithm H to an arbitrary two dimensional square mesh. Since the size of the mesh is not a power of two, the submesh size is not an even number at some recursive stage. We will describe how to partition an odd size mesh and to determine target addresses and show that only minor modifications are necessary for the move and smooth steps and no additional cost is incurred by unbalanced partitions.

4.6.1 Extension to a Three Dimensional Mesh

In this section, we extend Algorithm H to a three dimensional mesh and examine time complexity and buffer requirements. We show that for an n by n by n three dimensional mesh, where $n = 2^p$, the extended Algorithm H requires three buffers per processor and $22n$ message steps; specifically, it requires $12.5n$ data message steps and $9.5n$ integer message steps.

A three dimensional mesh is a cube of processors in which each interior processor is connected to six neighbors: two neighbors along the x axis, two neighbors along the y axis, and two neighbors along the z axis. The processors on the boundaries have no neighbors on one side of some axes. A three dimensional mesh of size $n \times n \times n$ consists of N processors, where $N = n^3$. The address of each processor is the triple of three indices (x, y, z), which indicate the location of the processor along each axis. We will mean by an "x-row" a one dimensional array of processors whose y and z indices are specified by some constants. Any x-row is parallel to the x-axis. We will assign "y-row" and "z-row" similar meanings. The "x-row index", "y-row index" and "z-row index" are given by the values $yn + z$, $zn + x$, and $xn + y$ respectively. We will use the term "yz-plane" to represent a two dimensional mesh of processors whose x index is specified by some constant. The "yz-plane index" is given by the value x. We will also assign "xz-plane" and "xy-plane" similar meanings. Note that an x-row is perpendicular to a yz-plane.

In a two dimensional mesh, each recursive stage of Algorithm H performs two separate move and smooth steps. The extension of Algorithm H to a three dimensional mesh performs three separate move and smooth steps at each recursive stage. The first stage of recursion begins by dividing a three dimensional mesh into two disjoint congruent regions along the z axis; that is, a cube of processors is divided into two identical halves by a plane perpendicular to the z axis. In the first move step, each message whose destination address is not contained in the current half is moved to the correct half along the z axis. Messages which are not in the correct half travel the same distance, and all movements are performed within z-rows; therefore, no contention occurs among messages.

After the first move step, a processor may have up to two messages, but the total number of messages in each half of the cube does not exceed the total number of processors in that half. Then the smooth step follows to restore the condition that each processor have at most one message. During the smooth step (see detailed descriptions on page 104 in this section), the messages are rearranged three times, along each axis. That is, messages are first distributed within each z-row so that the number of messages contained in any xy-plane differs from that of any other xy-plane by at most one. Next, the messages are moved within each y-row so that no x-row has more messages than the number of processors in that x-row. Finally, the messages are moved within each x-row, so that each processor in the cube has at most one message.

The second move step begins by dividing each of two halves into two congruent quarters of the original cube. That is, both halves are divided by a plane perpendicular to the y axis. Like the second move and smooth steps in a two dimensional mesh, the extended Algorithm H moves messages between quarters and then smoothes each quarter of the three dimensional mesh independently. After the second move and smooth steps, a processor in each quarter has at most one message whose destination address is contained in that quarter.

The third move and smooth steps divide each of the four quarters into two congruent cubes by a plane perpendicular to the x axis. The extended Algorithm H moves messages between the cubes and then smoothes the eight cubes simultaneously so that each processor in the three dimensional mesh has at most one message. Then the algorithm recurs on the half-size three dimensional subcubes of the original n by n by n cube.

Move steps in each recursion stage of three dimensional Algorithm H are very similar to those in the two dimensional case. Let us assume that the size of the current cube is $n \times n \times n$. During the first move step, the three dimensional mesh is divided into two halves by a plane perpendicular to the z axis, and the messages move parallel to the z axis. (See Figure 4.2.) Each message may remain in the current processor or travel along the z axis to the other half's corresponding processor, which is $n/2$ steps away. There is no contention among messages during movement along the z axis, because every message is moving parallel to the z axis. Like the first move step, the second move step divides the mesh into four quarters with a plane perpendicular to the y axis, and the messages travel parallel to the y axis. Likewise, the third move step divides the mesh by a plane perpendicular to the x axis into eight identical cubes, and the messages travel parallel to the x axis. We can see that each of the second and third move steps also takes $n/2$ data message steps.

Each smooth step on a three dimensional mesh consists of two phases, counting and distribution. We denote by $C(x, y, z)$ the number of messages in processor (x, y, z) before the counting phase. Then, the counting phase informs each processor (x, y, z) of three numbers:

1. the total number of messages contained in the current z-row's processors with a smaller processor index than processor (x, y, z), that is, $\sum_{k=0}^{z-1} C(x, y, k)$,

2. the total number of messages in the current yz-plane's z-rows with a smaller z-row index than processor (x, y, z), that is, $\sum_{j=0}^{y-1} \sum_{k=0}^{n-1} C(x, j, k)$,

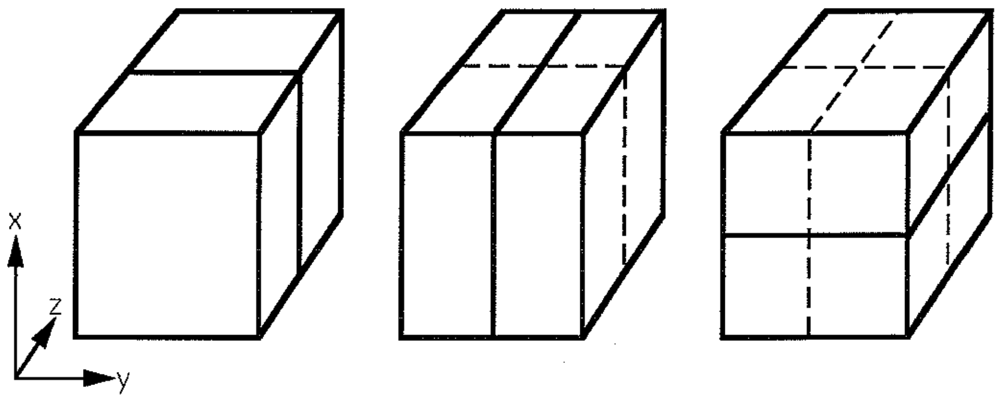


Figure 4.2: Partitions of a cube in Algorithm H

and

3. the total number of messages in yz-planes with a smaller yz-plane index than the current yz-plane, that is, $\sum_{i=0}^{x-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} C(i, j, k)$.

The distribution phase consists of three movements: the movement along z-rows, the movement along y-rows, and the movement along x-rows. Before and after each of the first two movements of the distribution phase, each processor contains at most two messages, and the first two movements are actually one-dimensional row movements, as described in Chapter 2. Before the third movement, each processor contains at most two messages. After the third movement, each processor contains at most one message. Therefore, the third movement of the distribution phase is actually a one-dimensional column movement, as described in Chapter 2.

For each movement within rows during the distribution phase, the processors have to compute the target configurations prior to the actual movements. The target configurations can be determined with the numbers which are received during the counting phase. For the purpose of computing the target configuration of the first z-row movement, the $n \times n \times n$ three dimensional cube is mapped onto an $n^2 \times n$ two dimensional mesh in such a manner that processor (x, y, z) in the three dimensional mesh is mapped to processor $(nx + y, z)$ in the two dimensional mesh. Then each processor can determine the proper target configuration of the current z-row as if that processor were in the two dimensional mesh and would determine the target configuration of two dimensional row movement as described in Chapter 3. (Recall that the target configuration of row movement in the two dimensional mesh satisfies the Final Configuration Property, which implies the numbers of messages in any two columns either are the same or differ by one. Since each column in the $n^2 \times n$ two dimensional mesh corresponds to an xy-plane of the $n \times n \times n$ three dimensional mesh, the Final Configuration Property implies that after z-row movement the numbers of messages in any pair of xy-planes either are same or differ by one.) For the second movement, which occurs within y-rows, each xy-plane of the cube is a two-dimensional mesh, and the y-rows in the cube are the rows of the two-dimensional mesh, just as the x-rows in the cube are the columns of the two-dimensional mesh.

For an n by n by n cube, the time complexity of the extended Algorithm H consists of the costs of the three move steps and the three smooth steps. The following two lemmas evaluate the time complexity of the extended Algorithm H.

Lemma 4.3 *For an n by n by n three dimensional mesh, there are three move steps in each recursive stage of extended Algorithm H. Each move step of the first recursive stage requires $n/2$ data message steps; the total cost of the move steps in the first recursive stage is $3n/2$ data message steps. During the move steps, no processor holds more than three messages at any time.*

Proof: For the first stage of recursion, there are three move steps within each z-row, each y-row and each x-row. Prior to each move step, the cube is divided into two regions by a plane perpendicular to an axis. Each move step requires $n/2$ data message steps, since all messages that move travel the same distance of half the cube along an axis. During each move step, all message movement is parallel to a single axis. Therefore, a processor may contain up to two transients and a single message which stays in place during the move step. □

The counting phase for a three dimensional mesh of size $S_x \times S_y \times S_z$ can be done by a procedure similar to that of the one applied to the two dimensional mesh in Chapter 3. To explain the counting phase, we consider the three dimensional mesh as a stack of yz-planes, each of which are indexed by the x index of the processors in that plane. To speed up the counting phase, we divide the stack of yz-planes into two halves, the upper half, which contains yz-planes with indices 0 through $S_x/2-1$, and the lower half, which contains yz-planes with indices $S_x/2$ through $S_x - 1$. We will describe the procedure for the upper half by the following:

1. To count the number of messages in a z-row, information travels along

each z-row from the processors at each end to the processors at the other end. These two information waves enable each processor to determine the total number of messages in its z-row and the number of messages in its row in processors with a smaller z index.

2. As soon as each processor knows the total number of messages in its z-row, information waves flow along y-rows. The information waves begin from the processors at both ends of each y-row and flow to the opposite ends. The number of messages in each z-row accumulates in each processor so that each processor can determine two numbers: the total number of messages in processors located in the z-rows with smaller z-row indices, and the total number of messages in its yz-plane.

3. As soon as each processor knows the number of messages in processors contained in its yz-plane, information waves begin along x-rows. Information waves begin from the processor of the yz-plane with index 0 to the processor of the yz-plane with index $S_x/2-1$. When these waves end, each processor knows the total number of messages in processors located in the yz-planes with smaller yz-plane index than its yz-plane index.

Lemma 4.4 *A smooth step of the extended Algorithm H applied to an S_x by S_y by S_z three dimensional mesh requires $(S_x/2+S_y+S_z-3)$ integer message steps for the counting phase. The distribution phase requires $(S_x/2+S_y+S_z-2)$ data message steps and two buffers per processor.*

Proof: The message steps for counting cease when the processors on the ends of x-rows with index $S_x/2-1$ receive the total number of messages in half the cube. Step 1 requires S_z-1 integer message steps, step 2 requires S_y-1 integer message steps, and step 3 requires $S_x/2-1$ integer message steps. Therefore, for a three dimensional mesh with size S_x by S_y by S_z, the counting phase requires $S_z+S_y+S_x/2-3$ integer message steps.

After the counting phase, each processor can determine its role in bringing about the smoothed configurations. The distribution phase of the smooth step for a three dimensional mesh of size S_x by S_y by S_z consists of three movements: row movement along z-rows, row movement along y-rows, and column movement along x-rows. The distribution phase requires, therefore, $S_x/2+S_y+S_z-2$ data message steps according to Theorems 2.1 and 2.3, since $b=2$. Before any row or column movement, each processor contains at most two messages. According to Theorem 2.2, no processor holds more than two messages at any time. □

There are three smooth steps at the first recursive stage of the extended Algorithm H for the n by n by n cube. The first smooth step operates on n by n by $n/2$ submeshes, and requires $2n-3$ integer message steps and $2n-2$ data message steps. The second smooth step operates on n by $n/2$ by $n/2$ submeshes, and requires $3n/2-3$ integer message steps and $3n/2-2$ data message steps. Finally, the third smooth step operates on $n/2$ by $n/2$ by $n/2$ submeshes, and requires $5n/4-3$ integer message steps and $5n/4-2$ data message steps.

If we denote by $T_{H_3}(n)$ the cost of the extended Algorithm H to solve a routing problem on an n by n by n three dimensional mesh, we get the following recurrence relations:

$$\begin{aligned}
T_{H_3}(1) &= 0 \\
T_{H_3}(n) &\le T_{H_3}(n/2) + 1.5n\,t_M + (4.75n-9)t_m + (4.75n-6)t_M \\
&\le T_{H_3}(n/2) + 6.25n\,t_M + 4.75n\,t_m \qquad (\text{where } n = 2^p)
\end{aligned}$$

We solve the equations and get

$$T_{H_3}(n) \le 12.5n\,t_M + 9.5n\,t_m \qquad (\text{where } n = 2^p)$$

The following theorem summarizes the cost of the extended Algorithm H on the cube.

Theorem 4.7 *For a three dimensional mesh of size n by n by n, where $n = 2^p$, the extended Algorithm H can solve any routing problem within $12.5n$ data message steps and $9.5n$ integer message steps. The extended Algorithm H requires only three buffers per processor.*

4.6.2 Extension to a K Dimensional Mesh

In this section, we further extend Algorithm H to a K dimensional mesh and examine the time complexity and buffer requirement. We will only sketch the procedures needed to handle a K dimensional mesh by generalizing from a three dimensional mesh rather than describing details of the algorithms.

A K dimensional mesh is a K dimensional hypercube of n^K processors, each of which is connected to two neighbors along each axis. We will call the axes the first axis, the second axis, ..., and the K-th axis. Extended Algorithm H on a K dimensional mesh consists of K consecutive move and smooth steps for each recursive stage. We assume $n = 2^p$.

On a K dimensional mesh of n^K processors, the first move step of the first recursive stage begins dividing the K dimensional hypercube into two disjoint congruent regions along the first axis; that is, a hypercube of processors is divided into two identical halves by a hyperplane perpendicular to the first axis. Then each processor determines which half contains the destination of its message. If the destination is contained in the current half, the message stays. If the destination is contained in the other half, the message is sent to the corresponding processor, which is $n/2$ steps away along the first axis. All message movement occurs along the first axis; therefore, there is no contention among messages during the first move step. As a result of the first move step, each processor contains up to two messages, but the total number of messages in each half of the hypercube does not exceed the total number of processors in that half. The first smooth step of the first recursive stage performs smoothing simultaneously and independently on each half of the hypercube.

The second move step begins by dividing each of two halves into two congruent quarters by a hyperplane perpendicular to the second axis. Like the first move step, the messages that must move travel $n/2$ steps along the second axis. As a result of the second move step, a processor contains up to two messages, so the second smooth step rearranges the messages in each of the four quarters simultaneously and independently so that each processor has at most one message.

The procedure continues for progressively smaller congruent regions. Finally, using a hyperplane perpendicular to the K-th axis, the K-th move step divides each of 2^{K-1} (K dimensional) submeshes into two sub-hypercubes with the half size in all axes of the original hypercube with n^K processors. During the K-th move step, according to the location of its destination, each message either stays in its current processor or travels $n/2$ steps along the K-th axis. The K-th smooth step takes 2^K sub-hypercubes of $(n/2)^K$ processors simultaneously and independently and rearranges the messages so that the precondition of each recursive stage is restored; that is, each processor has at most one message, and each message is contained in the same region with its destination. Then the algorithm recurs on half-sized hypercubes which are K dimensional meshes with $(n/2)^K$ processors.

Each move step of the first recursive stage requires $n/2$ data message steps on a K dimensional mesh with n^K processors, since the messages will either stay or travel to a corresponding processor along the designated axis during each move step. During each move step, there is no conflict, and up to three messages may be contained in a processor, since at any time, messages are moving parallel to a single axis. After each move step, a processor contains up to two messages.

Each smooth step operates on submeshes of various size, but consists of counting and distribution phases. The distribution phase of each smooth step consists of $K-1$ consecutive row movements and a column movement along each axis. We briefly examine the time requirement of a smooth step on a K dimensional mesh of size S_1 by S_2 by $\ldots$ by S_K.

The counting phase for each smooth step is performed on two halves simultaneously and independently by dividing the mesh into two halves by the hyperplane perpendicular to K-th axis. (The K-th axis is chosen because it is the last one to be divided by a partition.) Counting information first flows along the first axis and then flows along the second axis, and so on. Finally, the counting information waves flow along the K-th axis, and the counting phase is complete when the information waves from each end of the rows along the K-th axis meet in the middle of the row. The time requirement of counting phase is $(S_1 + S_2 + \cdots + S_{K-1} + S_K/2 - K)$ integer message steps.

The distribution phase consists of $K-1$ row movements along the first axis, then the second axis, $\ldots$, and finally along the $(K-1)$-th axis, followed by a column movement along the K-th axis. Row movements along each axis are performed within each row along that axis simultaneously and independently. In the beginning and end of row movement, each processor contains up to two messages. The row along the i-th axis consists of S_i processors; therefore, row movement along i-th row requires $S_i - 1$ data messages steps. Column movement along the K-th axis will require $S_K/2$ data message steps. Hence, the total time requirement of the distribution phase is $(S_1 + S_2 + \cdots + S_{K-1} + S_K/2 - K + 1)$ data message steps.

At the first recursive stage of extended Algorithm H on a K dimensional mesh of n^K processors, the first smooth step is performed on each of two halves of size $n/2 \times n \times n \times \cdots \times n$. Hence the first smooth step requires $n/2 + (K-2)n + n/2 - K$ integer message steps for counting and $n/2 + (K-2)n + n/2 - K + 1$ data message steps for distribution. The second smooth step is performed on each of quarters of size $n/2 \times n/2 \times n \times \cdots \times n$. Hence the second smooth step requires $2 \times n/2 + (K-3)n + n/2 - K$ integer message steps for counting and $2 \times n/2 + (K-3)n + n/2 - K + 1$ data message steps for distribution. In the same way, the i-th smooth step requires $i \times n/2 + (K-1-i)n + n/2 - K$ integer message steps for counting and $i \times n/2 + (K-1-i)n + n/2 - K + 1$ data message steps for distribution. Finally, the K-th smooth step, which takes sub-hypercubes of $(n/2)^K$ processors, requires $(K-1)n/2 + n/4 - K$ integer message steps for counting and $(K -$

$1)n/2+n/4-K+1$ data message steps for distribution. Therefore, the total cost of smooth steps at the first recursive stage on K dimensional mesh of size n^K processors is

$$\sum_{i=1}^{K-1}\{(2K-1-i)\frac{n}{2}-K\}+(K-1)\frac{n}{2}+\frac{n}{4}-K$$

integer message steps for counting and

$$\sum_{i=1}^{K-1}\{(2K-1-i)\frac{n}{2}-K+1\}+(K-1)\frac{n}{2}+\frac{n}{4}-K+1$$

data message steps for distribution. By simplifying above equations, we find that the total cost of counting phases is $(3K^2-3K+1)n/4-K^2$ integer message steps, and that the total cost of distribution phases is $(3K^2-3K+1)n/4-K^2+K$ data message steps.

Using the fact that the total cost of K move steps at the first recursive stage on a K dimensional mesh of size n^K is $Kn/2$ data message steps, we obtain the following recurrence equations for the time complexity of the extended Algorithm H:

$$\begin{aligned} T(1) &= 0 \\ T(n) &\leq T(\frac{n}{2})+\frac{K\,n}{2}t_M+\{\frac{(3K^2-3K+1)n}{4}-K^2\}t_m \\ &\quad +\{\frac{(3K^2-3K+1)n}{4}-K^2+K\}t_M \end{aligned}$$

By solving the equation and dropping small negative terms, we get

$$\begin{aligned} T(n) &\leq \{K\,n+\frac{(3K^2-3K+1)n}{2}\}\,t_M+\frac{(3K^2-3K+1)n}{2}\,t_m \\ &= O(K^2n)t_M+O(K^2n)t_m \end{aligned}$$

For the extended Algorithm H to a K dimensional mesh, the total cost of move steps is $Kn\,t_M$, which is equal to the bound imposed by the distance between the farthest processors. The cost of the algorithm is dominated by the cost of smoothing, which increases quadratically as the dimension of the mesh increases, while the number of processors increases exponentially with the dimension. Therefore, the quadratic increase of the cost may be tolerable. The buffer requirements for the extended Algorithm H are still three regardless of the number of dimensions.

4.6.3 The k-strip Algorithm H

Algorithm H divides a two dimensional mesh into two identical halves. Instead of dividing a mesh into two halves, Algorithm H can be extended to divide a mesh into k identical strips prior to each move step. In this section, we describe the k-strip Algorithm H on a two dimensional square mesh and examine its time complexity.

We will consider a mesh with n rows and n columns where n is equal to k^p and $k > 2$ is even. Like Algorithm H, the k-strip Algorithm H consists of two separate move and smooth steps. (Partitionings are illustrated in Figure 4.3.) The first move step of the first recursive stage begins by dividing the mesh into k identical strips having n rows and n/k columns. Each processor determines which strip contains the destination address of its message. A message whose destination address is in the current strip stays at the current processor. Each message whose destination address is contained in another strip is moved to the processor at the same relative position of the strip which contains its destination address. During the first move step, messages travel along rows and stop as soon as they reach the target processors. There is no contention among messages since the links are bidirectional. As in Algorithm H, after the first move step, the number of messages contained in each strip is no greater than the number of processors in that strip. Each processor, however, may contain up to k messages. The first smooth step rearranges the messages in each strip so that each processor contains at most one message.

The first smooth step is almost the same as the smooth step of Algorithm H, but a processor may now contain up to k messages compared to two messages in Algorithm H. That is, the first smooth step consists of the counting phase and the distribution phase which are performed on an n by n/k rectangular mesh.

After the first smooth step, each message resides in the strip that contains its destination. The second move step begins by dividing each strip into k identical square meshes with n/k rows and n/k columns. Each processor determines which square mesh contains the destination address of its message and, if necessary, sends the message to the correct square mesh in the same way as in the first move step. During the second move step, messages travel along columns, and there is no contention. Again, after the second move step, each processor may contain up to k messages. The number of messages in

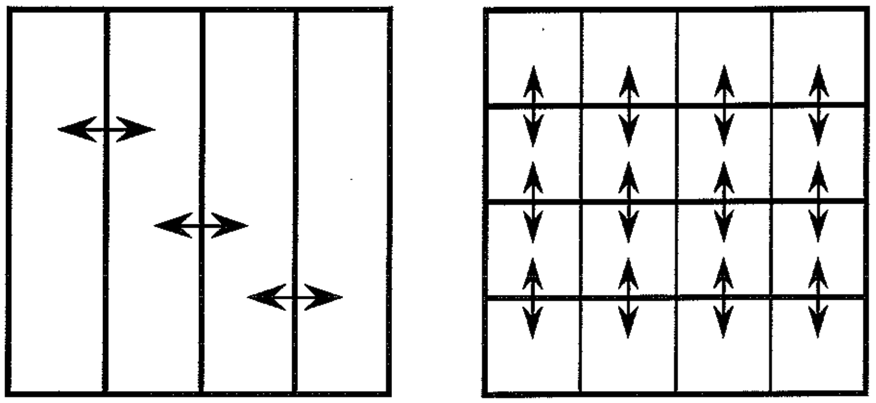


Figure 4.3: Partitions of a square mesh for the k-strip Algorithm H

each square mesh, however, does not exceed the number of processors in that square, and each message resides in the square that contains its destination. As usual the second smooth step follows to restore the precondition that each processor contain at most one message. Then the algorithm recurs on each of the square meshes simultaneously.

We denote by $T_{H_k}(n)$ the cost of solving a routing problem with the k-strip Algorithm H on an n by n two dimensional mesh. We will construct the recurrence equations for $T_{H_k}(n)$. $T_{H_k}(n)$ consists of the cost of the first recurrence stage and the cost of solving the problem of $1/k$ size meshes, $T_{H_k}(n/k)$.

The first move step requires $(k-1)n/k$ data message steps along a row, since the longest distance between the corresponding processors is from the processor in the leftmost strip to the one in the rightmost strip. Like the first move step, the second move step requires $(k-1)n/k$ data message steps along a column. Note that no processor contains more than $k+1$ messages, since prior to the last step of the move, there are at most $k-1$ messages which have reached the target processor and at most two transient messages.

The first smooth step operates on k strips of an n by n/k mesh simultaneously. The counting phase on an n by n/k mesh requires $(1/k+1/2)n-2$ integer message steps. The distribution phase consists first of row movement on rows of n/k processors having up to k messages and then of column movement on columns of n processors having up to k messages. Assuming that $k \geq 4$ is even, row movement requires $\frac{(k+2)n}{4(k+1)}$ data message steps by applying Theorem 2.1, and column movement requires $(k-1)n/k$ data message steps by applying Theorem 2.3.

Since the k vertical strips of size n by n/k are divided into k horizontal strips again, the second smooth step operates on k^2 square meshes of size n/k by n/k simultaneously. The counting phase requires $3n/2k - 2$ integer message steps. The distribution phase consists of row movement and column movement on the arrays of n/k processors having up to k messages. Row movement requires $\frac{(k+2)n}{4(k+1)}$ data message steps by applying Theorem 2.1, and column movement requires $(k-1)n/k^2$ data message steps by applying Theorem 2.3.

The cost function, $T_{H_k}(n)$, has the following recurrence equations where $k \geq 4$ is even.

$$\begin{aligned}
T_{H_k}(1) &= 0 \\
T_{H_k}(n) &\leq T_{H_k}(n/k) + \frac{2(k-1)n}{k}t_M + \frac{(k+5)n}{2k}t_m \\
&\quad + \{\frac{k+2}{2(k+1)} + \frac{k-1}{k} + \frac{k-1}{k^2}\}n\,t_M
\end{aligned}$$

Solving the equation where $n = k^p$, we get

$$\begin{aligned}
T_{H_k}(n) &\leq 2nt_M + \frac{(k+5)n}{2(k-1)}t_m + \{\frac{k(k+2)}{2(k-1)(k+1)} + \frac{k+1}{k}\}n\,t_M \\
&\leq 2nt_M + (\frac{1}{2} + \frac{3}{k-1})n\,t_m + \{\frac{3}{2} + \frac{2k+1}{2(k^2-1)} + \frac{1}{k}\}n\,t_M \\
&\leq 4.05n\,t_M + 1.5nt_m \qquad \text{(when } k = 4\text{)}
\end{aligned}$$

From the equation, we can see that for an n by n square mesh, the move steps take a total of $2n$ data message steps, and the cost of the move steps does not increase even though k increases. The cost of the smooth steps, however, decreases as we increase k, the number of strips. The smooth step is an additional step whose purpose is to keep the number of buffers as a small constant. Note that $k+1$ buffers are required to implement the k-strip Algorithm H.

Algorithm H requires $9n$ routing steps and three buffers. A four-strip Algorithm H requires $5.55n$ routing steps and five buffers. Algorithm Q requires $5.5n$ routing steps and five buffers. By comparing Algorithm H with a four-strip Algorithm H, we can see that two more buffers reduce the time requirement to about 60 percent. This improvement comes from the reduction of the smoothing cost, since the number of recursive stages is reduced by half and the cost of smoothing in each recursive stage is not doubled. Doubling the number of strips of Algorithm H requires double the amount of buffers and more time for row movements, but cuts the number of recursive stages in half. Therefore, total routing costs decrease as the number of strips is increased (all other things being equal). Note that the total cost of move steps in any algorithm is always $2n$ data message steps, no matter what number of strips is selected. If each processor has five buffers available, Algorithm Q and four-strip Algorithm H have almost the same time requirements. Furthermore, the time requirement of k-strip Algorithm H decreases as the number of buffers per processor increases, which is not the case in Algorithm Q.

Another form of the k-strip algorithm would divide a mesh into k vertical strips at each recursive stage instead of alternating horizontal and vertical divisions. This algorithm would require a total of $O(n \log n)$ routing steps on an n by n mesh, because each counting phase and column movement would require at least $n/2$ routing steps at every recursive stage, and $\log n$ recursive stages would be necessary by the time of termination.

4.6.4 Algorithm H on an Arbitrary Square Mesh

When the size of a two dimensional square mesh is not a power of two, at the beginning of some recursive stage, the size of submeshes is not even. Therefore, the submesh cannot be partitioned into two identical congruent regions. In this section, we sketch how the move and smooth steps of Algorithm H can be performed on a square mesh of odd size and examine how the time complexity and the buffer requirements are affected.

When the size of a square mesh is odd, the mesh is divided into $2s+1$ by $s+1$ and $2s+1$ by s rectangular submeshes when the size of the square mesh is $2s+1$ by $2s+1$, as illustrated in Figure 4.4. Then, during the move step, the messages that must move from the smaller half to the larger half have no problem. On the other hand, some of the messages that must move from the larger half to the smaller half do have a problem because processors in the same relative position in the smaller half do not exist. That is, for $0 \leq i < 2s+1$, processor (i, s) has no corresponding processor in the smaller half, since the smaller one has one less column. If for $0 \leq i < 2s+1$, processor (i, s) in the larger half sends its message that must move to processor $(i, s-1)$ in the smaller half, then messages will travel to their target processors with s steps during the move step. Other messages that must move will travel to their target processor with $s+1$ steps during the move step.

The white region of the left half is identical to the right half. Therefore, the target processor of each message that must move to the other half is the processor on the same relative position of that half. Messages in the hatched area are sent to the gray area in the right half, if necessary. As a result, processors in the gray area contains up to three messages after the move step.

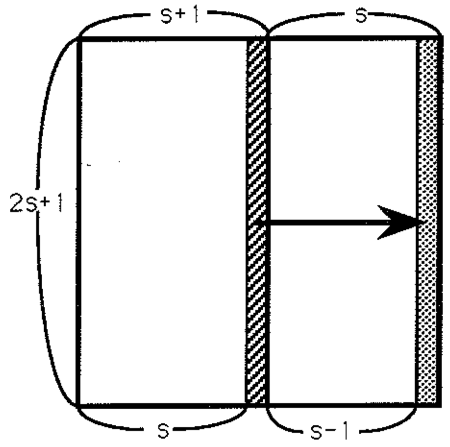


Figure 4.4: Partitions of an odd-sized mesh by Algorithm H

When each processor determines the target processor of its message that must move by the mapping described above, there is no contention among messages during the move step, since all messages travel along their rows by $s+1$ steps except the messages in column s of the larger half. The messages from column s of the larger half moving to the column $s-1$ of the smaller half cause no contention, either. Even though processors in column $s-1$ of the smaller half will contain up to three messages after the move step, no processor needs more than three buffers at any moment in the move step. During the move step, processors of the inner columns may contain three messages, one that stays and two transients. Processors in column $s-1$ of the smaller half may contain two messages during the move step and three messages after the move step. (Note that during the move step there are no transients at the outermost columns of the mesh.)

After the move step, a processor contains up to two messages except those in column $s-1$ of the smaller half, which contain up to three messages. But the number of messages in each half does not exceed the number of processors in that half. The smooth step rearranges the messages in each half simultaneously and independently so that each processor contains at most one message.

The smooth step is performed in the same way as when the size of the mesh is even. Even though processors in column $s-1$ of the smaller half may contain three messages, there is no difference in the counting phase. The row movement within each row of the smaller half is not affected by an extra message in the rightmost processor; that is, the time complexity of row movement is not changed even though there are at most $2s+1$ messages in an array of s processors. Column movements in the smaller half of the mesh are also not affected by an extra message in each row. The buffer requirement of the smoothing step is three, since there are at most three messages in a processor and the number of messages does not increase during the smooth step according to Theorem 2.2.

In summary, when the size of a square mesh is not even, Algorithm H divides the n by n mesh into a larger half and a smaller half which differ by one in the column size. The messages that must move from the smaller half to the larger half will travel to the target processor which is $\lceil n/2 \rceil$ steps away along the current row. The messages that must move from the larger half to the smaller half will travel to the target processor in the same way, except for the messages in the last column of the larger half. The messages in the last column of the larger half will travel to the target processor which is in the last column of the smaller half by $\lfloor n/2 \rfloor$ steps along the current row. The buffer requirement and the time complexity are not changed even though the control structure of the algorithm becomes somewhat more complicated.

4.6.5 Algorithm H on a Rectangular Mesh

We now show that Algorithm H can be applied to a rectangular mesh of any size. In this section, we describe Algorithm H with an r by c rectangular mesh where $r = 2^s$, $c = 2^t$, and $r > c$. For a rectangular mesh, Algorithm H first partitions the mesh vertically into two rectangular regions of size r by $c/2$, each of which is half the mesh. The move and smooth steps are then performed on these two regions, after which every message is in the correct half of the array, and every processor has at most one message. The algorithm then partitions each of these two regions horizontally into two rectangular regions of size $r/2$ by $c/2$ and performs move and smooth steps on those regions. Then the algorithm recurs. The recursion terminates when each region has a column of r/c processors, at which point each message can travel to its destination without conflict, since each region is a one dimensional array of processors.

This algorithm requires two separate move and smooth steps for each recursive stage. The first move step will move a message either 0 or $c/2$ steps along its current row; the second move step will move each message either 0 or $r/2$ steps along its current column. Thus, the maximum distance moved by any message in stage one of this algorithm is $(r + c)/2$, the same as for Algorithm Q on a rectangular mesh.

The first smoothing step of the first stage is performed on two r by $c/2$ submeshes simultaneously and independently and requires $r/2 + c/2 - 2$ integer message steps for counting and $r/2 + c/2 - 1$ data message steps for distribution. (By Theorem 2.1, row movement on an array of $c/2$ processors with $b = 2$ requires $c/2 - 1$ data message steps.) The second smoothing step of the first stage is performed on each of $r/2$ by $c/2$ submeshes and requires $r/4 + c/2 - 2$ integer message steps for counting and $r/4 + c/2 - 1$ data message steps for distribution. Adding in the $(r+c)/2$ steps required for the move, when $r = 2^s$, $c = 2^t$, and $r > c$, the complexity of the algorithm is given by

$$\begin{aligned} T(r/c, 1) &= r/c - 1 \\ T(r, c) &\leq T(r/2, c/2) + \frac{r+c}{2} t_M + (\frac{3r}{4} + c - 4)t_m + (\frac{3r}{4} + c - 2)t_M \end{aligned}$$

Solving above recurrence equation, we get

$$\begin{aligned} T(r, c) \leq\ & (r - \frac{r}{c} + c - 1)t_M + (\frac{3r}{2} - \frac{3r}{2c} + 2c - 2 - 4\log c)t_m \\ & + (\frac{3r}{2} - \frac{3r}{2c} + 2c - 2 - 2\log c)t_M \end{aligned}$$

Dropping negative terms strengthens the inequality to the following:

$$T(r, c) \leq (2.5r + 3c)t_M + (1.5r + 2c)t_m$$

Note that $T(r, c)$ gives the same time complexity as that on a square mesh when $r = c$. The buffer requirement of Algorithm H on a rectangular mesh does not change; it requires three buffers per processor.

When the size of a rectangular mesh is not a power of two on either side, we can apply the same method of mapping described in Section 4.6.4 in order to determine target addresses. As shown in that section, the time complexity and the buffer requirements do not change for a mesh in which the number of rows or columns or both is odd. We conclude that Algorithm H is applicable to any size mesh with the complexity results stated above.

Chapter 5

Conclusion

5.1 Summary

We have described several versions of "move and smooth" routing algorithms, a class of routing algorithms on mesh-connected computers. They can solve several classes of message routing problems: full permutation, (partial) permutation, and restricted broadcasting. Each processor in a mesh initially contains either no message or a single message with one or more destinations, and each processor is the destination of at most one message.

Move and smooth algorithms are recursive. Initially, the entire mesh is considered a single region. At each recursive stage,

- Each region is partitioned into subregions;
- A copy of each message is moved to each of the regions that contains one of its destinations (the move step);
- Messages within each region are re-distributed so that each processor contains at most one message (the smooth step).

The recursion continues until each region contains a single row or column of processors, at which time each message has arrived at or can be moved directly to its destination.

We first analyzed message movement within a one dimensional array of processors. We then described two representatives of move and smooth algorithms; one called Q because it divides a mesh into four regions (Quarters) at each recursive stage, and the other called H because it divides a mesh into two regions (Halves).

Algorithms	Time Complexity	Number of Buffers	Note
Algorithm Q	$4n\,t_M + 1.5n\,t_m$	5	$n \times n$ mesh where $n = 2^p$
Q with k^2-partitioning	$\frac{5k+3}{4}n\,t_M + \frac{1.5}{k-1}n\,t_m$	$k^2 + 2$	$n \times n$ mesh, $n = 2^p$, $4 \leq k \leq \sqrt{n}$
Q on K dimensional mesh	$O(K2^K n\,t_M)$	$2^K + 1$	n^K mesh, $K \geq 3$, $n = 2^p$
Q on arbitrary square mesh	$\cong (4.3n\,t_M + 1.5n\,t_m)$	5	$n \times n$ mesh where $n \neq 2^p$
Q on rectangular mesh	$(1.75r + 2.2c)t_M$ $+ (0.5r + c)t_m$	5	$r \times c$ mesh, $r = 2^s$, $c = 2^t$, $r > c$

Table 5.1: Time complexities and buffer requirements of Algorithm Q

In Chapter 3, we investigated the details of Algorithm Q on a two dimensional square mesh. We showed that Algorithm Q can solve full permutation and (partial) permutation problems in $5.5n$ routing steps with five buffers per processor on an n by n square mesh. We also showed that Algorithm Q can handle restricted broadcasting problems without additional routing costs if the cost of message copying is negligible compared to the cost of communication between processors. We considered the extension of Algorithm Q to a K dimensional mesh and found that it is difficult to perform move steps without conflicts, and that the cost of smooth steps increases exponentially with K. Another extension of Algorithm Q partitions a mesh into k^2 regions instead of four. We showed that the cost of Algorithm Q with k^2-partitioning increases linearly as the value k increases, leading us to conclude that the original Algorithm Q uses the most efficient partitioning of a square mesh in this class. In Section 3.6.3, we sketched a method to overcome the problem of unbalanced partitions when the size of a mesh is odd. We exhibited a mapping from bigger regions to smaller regions that allows conflict-free movement during move steps. In the last section of Chapter 3, we described Algorithm Q on an r by c rectangular mesh. Table 5.1 summarizes the time complexities and the buffer requirements of Algorithm Q and its extensions.

Algorithms	Time Complexity	Number of Buffers	Note
Algorithm H	$5.5n\,t_M + 3.5n\,t_m$	3	$n \times n$ mesh where $n = 2^p$
H with k strip	$\leq 4.05n\,t_M + 1.5n\,t_m$ $\geq 3.5n\,t_M + 0.5n\,t_m$	$k+1$	$n \times n$ mesh, $n = 2^p$, k is even, $4 \leq k < n$
H on K dimensional mesh	$(\frac{3K^2}{2} - \frac{K}{2} + 1)n\,t_M$ $+ (\frac{3K^2}{2} - \frac{3K}{2} + 1)n\,t_m$	3	n^K mesh, $n = 2^p$, $K \geq 3$
H on arbitrary square mesh	$\cong (5.5n\,t_M + 3.5n\,t_m)$	3	$n \times n$ mesh where $n \neq 2^p$
H on rectangular mesh	$(2.5r + 3c)t_M$ $+ (1.5r + 2c)t_m$	3	$r \times c$ mesh, $r = 2^s$, $c = 2^t$, $r > c$

Table 5.2: Time complexities and buffer requirements of Algorithm H

In Chapter 4, we described the details of Algorithm H. On an n by n square mesh where $n = 2^p$, Algorithm H requires $9n$ routing steps and three buffers per processor. Algorithm H can handle restricted broadcasting problem with no additional cost. In Section 4.6.1, we described an extension of H to a three dimensional mesh and then generalized it to a K dimensional mesh. For a K dimensional mesh, Algorithm H requires $(3K^2 - 2K + 1)n$ routing steps and three buffers per processor. Another extension of H, which divides a two dimensional mesh into k strips on successive move and smooth steps, was discussed in Section 4.6.3. We showed that the time complexity of the k-strip extension decreases as the value k increases, but the buffer requirement increases. In Section 4.6.4, we described a method to implement the move step when the size of a two dimensional mesh is not a power of two and the regions are not congruent. Even though the control structure of the algorithm becomes more complicated, no additional routing cost is incurred by unbalanced partitions. In the last section of Chapter 4, we described Algorithm H on a rectangular mesh. Table 5.2 summarizes the time complexities and the buffer requirements of Algorithm H and its extensions.

Table 5.3 compares our algorithms with other deterministic routing algorithms on a square mesh. Kunde's routing algorithm [18] makes substantial

Algorithms	Time Complexity	Number of Buffers	Note
Algorithm Q	$5.5n$	5	full and partial permutations, restricted broadcast
Algorithm H	$9n$	3	full and partial permutations, restricted broadcast
Kunde's	$3.5n \sim 5.5n$	4	full and partial permutations, sorting-based
Leighton et al.	$2n-2$	constant	full and partial permutations, # of buffers > 500, optimal time, sorting-based
Sorting	$3n + O(n^{\frac{3}{4}}) \sim 7n$	2	full permutations

Table 5.3: Comparison of deterministic routing algorithms

use of sorting as a method to reduce the buffer requirement. If Kunde's algorithm uses four buffers and a sorting algorithm with the time requirement of $3n + O(n^{\frac{3}{4}})$ [41], it can solve a (partial) permutation problem with $3.5n + O(n^{\frac{3}{4}})$ data message steps on an n by n mesh. However, for practical values of n, asymptotically fast sorting algorithms are inferior to simpler parallel sorting algorithms such as the shear sort and the algorithm by Lang et al. [39]. If Kunde's routing algorithm uses four buffers and a sorting algorithm with time complexity of $7n$ [20], it takes $5.5n$ data message steps to solve a (partial) permutation problem. It is not clear how to apply Kunde's algorithm to restricted broadcasting problems.

The algorithm by Leighton et al. [23] is based on Kunde's algorithm. It requires a constant number of buffers and $2n-2$ data message steps, which is optimal for an n by n mesh. However, the buffer requirement is at least a few hundred; therefore, this algorithm is not practical for a small size mesh. It is not clear whether this algorithm can be applied to restricted broadcasting problems.

Algorithm Q requires five buffers and $5.5n$ routing steps, of which $4n$ are data message steps and 1.5n steps are integer passing steps. Moreover, our move and smooth algorithms can solve restricted broadcasting problems with the same time complexity as for permutations.

We conclude the dissertation with the properties of move and smooth algorithms:

- The algorithms are deterministic.
- The algorithms can handle (full and partial) permutation and restricted broadcasting problems without modification; the cost of executing each routing algorithm is the same over all problems.
- The algorithms are distributed.
- The number of buffers required in a processor is constant regardless of the size of mesh.
- The algorithms may be as fast as other known deterministic distributed routing algorithms that use a small number of buffers for meshes of practical size.

5.2 Suggestions for Further Work

Because our network model is one in which each processor communicates with its neighbors synchronously, and all processors can be viewed as executing the same program, our move and smooth algorithms can be implemented on a SIMD machine such as the ILLIAC model in Section 1.4. In our network model, a processor can send and receive up to four messages in a single routing step, while it takes four routing steps to do so in an ILLIAC model. Therefore, a straightforward application of our algorithms to an ILLIAC model requires four times more routing steps than our network model. But in fact our algorithms will require substantially fewer routing steps than four times that of the distributed model, because messages travel in a single direction along columns or rows during most phases of both move and smooth steps. A more precise analysis is necessary to determine the usefulness of move and smooth algorithms for SIMD machines.

We believe we can improve the time bound of move and smooth algorithms by overlapping the counting and distribution phases at the cost of a more complicated control structure and possibly an additional buffer. On a two dimensional mesh, for each recursive stage, the processors on the border of each partitioned region can determine the number of messages in each row of the current submeshes as soon as the move step ends. These values can then be broadcast along each row and accumulated along the border column at the same time. As soon as a processor receives information about the initial configuration of the current row, it can begin a preliminary distribution phase assuming a *standard* final configuration which positions "extra" messages in the middle of the row. The actual final configuration can be achieved after the information of the actual final configuration is received by all processors in the mesh. With this overlapping, we expect the cost of Algorithm H to be reduced to $7.5n$ routing steps. (Specifically, $5.5n\, t_M + 2n\, t_m$.) An analogous approach can reduce the cost of Algorithm Q to $5n$ routing steps ($4n\, t_M + n\, t_m$). Further investigation is necessary to determine the precise time complexity and the buffer requirements.

It will be also interesting to examine whether our move and smooth algorithms can handle many-to-many broadcasting problems. Under the assumption that messages to the same destination are merged by an associative operator, many-to-many broadcasting problems may be solved under some restrictions. If a packet can carry k messages to different destinations and each processor will receive no more than k messages, the messages to the same processor will be merged at the target processors and finally delivered to their destinations.

We can also examine meshes in which the connection topology is not a grid. For example, if neighboring processors are connected with diagonal connections as well as row and column connections on a two dimensional mesh, our Algorithm Q is clearly applicable, but will exhibit a different behavior since the move step in the first recursive stage on an $n \times n$ mesh will require $n/2$ steps rather than n.

Bibliography

[1] G. H. Barnes, R. M. Brown, M. Kato, D. L. Slotnick, and R. A. Stokes. The ILLIAC IV computer. *IEEE Transactions on Computers*, C-17(8):746–757, August 1968.

[2] D. Bitton, D. J. DeWitt, D. K. Hsiao, and J. Menon. A taxonomy of parallel sorting. *ACM Computing Surveys*, 16(3):287–318, September 1984.

[3] A. Borodin and J. E. Hopcroft. Routing, merging and sorting on parallel models of computation. In *Proceedings of the 14th ACM Symposium on Theory of Computing*, pages 338–344, San Francisco, CA, May 5-7, 1982.

[4] A. Borodin and J. E. Hopcroft. Routing, merging, and sorting on parallel models of computation. *Journal of Computer and System Sciences*, 30(1):130–145, February 1985.

[5] William J. Dally. Wire-efficient VLSI multiprocessor communication networks. In Paul Losleben, editor, *Advanced Research in VLSI: Proceedings of the 1987 Stanford Conference*, pages 391–415, Cambridge, Massachusetts, 1987. The MIT Press.

[6] Tse-yun Feng. A survey of interconnection networks. *Computer*, 14(12):12–27, December 1981.

[7] Charles M. Flaig. VLSI mesh routing system. Technical Report 5241:TR:87, Computer Science Department, California Institute of Technology, 1987.

[8] Peter M. Flanders. A unified approach to a class of data movements on an array processor. *IEEE Transactions on Computers*, C-31(9):809–819, September 1982.

[9] M. J. Flynn. Very high-speed computing systems. *Proceedings of IEEE*, 54:1901–1909, 1966.

[10] D. Gannon and J. Van Rosendale. On the impact of communication complexity in the design of parallel numerical algorithms. *IEEE Transactions on Computers*, C-33(12):1180–1194, December 1984.

[11] Yijie Han and Yoshihide Igarashi. Time lower bounds for parallel sorting on a mesh-connected processor array. In J. H. Reif, editor, *VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing, AWOC 88*, pages 434–443, Corfu, Greece, June 28–July 1, 1988. Springer-Verlag.

[12] C. R. Jesshope. Some results concerning data routing in array processors. *IEEE Transactions on Computers*, C-29(7):659–662, July 1980.

[13] Danny Krizanc. Integer sorting on a mesh-connected array of processors. Technical Report 332, Dept. of Computer Science, University of Rochester, March 1990.

[14] Danny Krizanc, Sanguthevar Rajasekaran, and Thanasis Tsantilas. Optimal routing algorithms for mesh-connected processor arrays. In J. H. Reif, editor, *VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing, AWOC 88*, pages 411–422, Corfu, Greece, June 28–July 1, 1988. Springer-Verlag.

[15] M. Kumar and D. S. Hirschberg. An efficient implementation of Batcher's odd-even merge algorithm and its application in parallel sorting schemes. *IEEE Transactions on Computers*, C-32(3):254–264, March 1983.

[16] M. Kunde. Lower bounds for sorting on mesh-connected architectures. *Acta Informatica*, 24(2):121–130, April 1987.

[17] M. Kunde. Parallel routing on multi-dimensional grids of processors. In *CONPAR88: Conference on Algorithms and Hardware for Parallel Processing*, pages 687–694, UMIST, Manchester, UK, September 12–16, 1988. Cambridge University Press.

[18] Manfred Kunde. Routing and sorting on mesh-connected arrays. In J. H. Reif, editor, *VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing, AWOC 88*, pages 423–433, Corfu, Greece, June 28–July 1, 1988. Springer-Verlag.

[19] Manfred Kunde and Thomas Tensi. Multi-packet-routing on mesh connected arrays. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 336–343, Santa Fe, New Mexico, June 18–21, 1989.

[20] Hans-Werner Lang, Manfred Schimmler, Hartmut Schmeck, and Heiko Schröder. Systolic sorting on a mesh-connected network. *IEEE Transactions on Computers*, C-34(7):652–658, July 1985.

[21] T. Lang. Interconnections between processors and memory modules using the shuffle-exchange network. *IEEE Transactions on Computers*, C-25(5):496–503, May 1976.

[22] Tom Leighton, Bruce Maggs, and Satish Rao. Universal packet routing algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 256–269. IEEE, Oct. 24-26, 1988.

[23] Tom Leighton, Fillia Makedon, and Ioannis Tollis. A 2n-2 step algorithm for routing in an $n \times n$ array with constant size queues. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 328–335, Santa Fe, New Mexico, June 18–21, 1989.

[24] G. Lev, N. Pippenger, and L. G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Transactions on Computers*, C-30(2):93–100, February 1981.

[25] Yiming Ma, Sandeep Sen, and Isaac D. Scherson. The distance bound for sorting on mesh-connected processor arrays is tight. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 255–263, October 27–29, 1986.

[26] Gyula A. Magó and Donald F. Stanat. The FFP machine. In Veljko M. Milutinović, editor, *High Level Language Computer Architecture*, chapter 12. Computer Science Press, 1989.

[27] Fillia Makedon and Adonis Simvonis. Fast parallel communication on mesh connected machines with low buffer requirements. In *Proceedings of 1990 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pages 78–81, Cambridge, Massachusetts, September 17-19, 1990.

[28] D. Nassimi and S. Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Transactions on Computers*, C-28(1):2–7, January 1979.

[29] D. Nassimi and S. Sahni. Data broadcasting in SIMD computers. *IEEE Transactions on Computers*, C-30(2):101–107, February 1981.

[30] David Nassimi and Sartaj Sahni. An optimal routing algorithm for mesh-connected parallel computers. *Journal of the ACM*, 27(1):6–29, January 1980.

[31] S. E. Orcutt. Implementation of permutation functions in Illiac IV-type computers. *IEEE Transactions on Computers*, C-25(9):929–936, September 1976.

[32] M. C. Pease. The indirect binary n-cube microprocessor array. *IEEE Transactions on Computers*, C-26(5):458–473, May 1977.

[33] N. Pippenger. Parallel communication with limited buffers. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, pages 127–136, October 24–26, 1984.

[34] David A. Plaisted. An architecture for fast data movement in the FFP machine. In *Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 147–163, Nancy, France, September 16–19, 1985.

[35] Prabhakar Raghavan. Robust algorithms for packet routing in a mesh. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 344–350, Santa Fe, New Mexico, June 18–21, 1989.

[36] C. S. Raghavendra and V. K. P. Kumar. Permutations on Illiac IV-type networks. *IEEE Transactions on Computers*, C-35(7):662–669, July 1986.

[37] S. Rajasekaran and Th. Tsantilas. An optimal randomized routing algorithm for the mesh and a class of efficient mesh-like routing networks. In *7th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 226–241, Pune, India, December 1987. Springer-Verlag. Lecture Notes in Computer Science 287.

[38] Kazuhiro Sado and Yoshihide Igarashi. A function for evaluating the computing time of a bubbling system. *Theoretical Computer Science*, 54:315–324, 1987.

[39] Kazuhiro Sado and Yoshihide Igarashi. Fast parallel sorting on a mesh-connected processor array. *The Transactions of the Institute for Electronics and Information and Communication Engineering*, E 71(4):422–430, April 1988.

[40] Isaac D. Scherson and Sandeep Sen. Parallel sorting in two-dimensional VLSI models of computation. *IEEE Transactions on Computers*, 38(2):238–249, February 1989.

[41] C. P. Schnorr and A. Shamir. An optimal sorting algorithm for mesh connected computers. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 255–263, Berkeley, CA, May 28–30, 1986. The ACM Special Interest Group for Automata and Computability Theory.

[42] Howard Jay Siegel. *Interconnection Networks and Masking Schemes for Single Instruction Stream - Multiple Data Stream Machines.* PhD thesis, Dept. of Electrical Engineering and Computer Science, Princeton University, May 1977.

[43] Howard Jay Siegel. *Interconnection Networks for Large-Scale Parallel Processing.* Lexington Books, D.C. Heath and Company, Lexington, Massachusetts, 1985.

[44] R. C. Swanson. Interconnections for parallel memories to unscramble p-ordered vectors. *IEEE Transactions on Computers*, C-23(11):1105–1115, November 1974.

[45] C.D. Thompson and H.T. Kung. Sorting on a mesh-connected parallel computer. *Communications of the ACM*, 20(4):263–271, April 1977.

[46] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, May 1982.

[47] L. G. Valiant. Optimality of a two-phase strategy for routing in interconnection networks. *IEEE Transactions on Computers*, C-32(9):861–863, September 1983.

[48] L.G. Valiant and G.J. Brebner. Universal schemes for parallel communication. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 263–277, Milwaukee, Wisconsin, May 11–13, 1981.