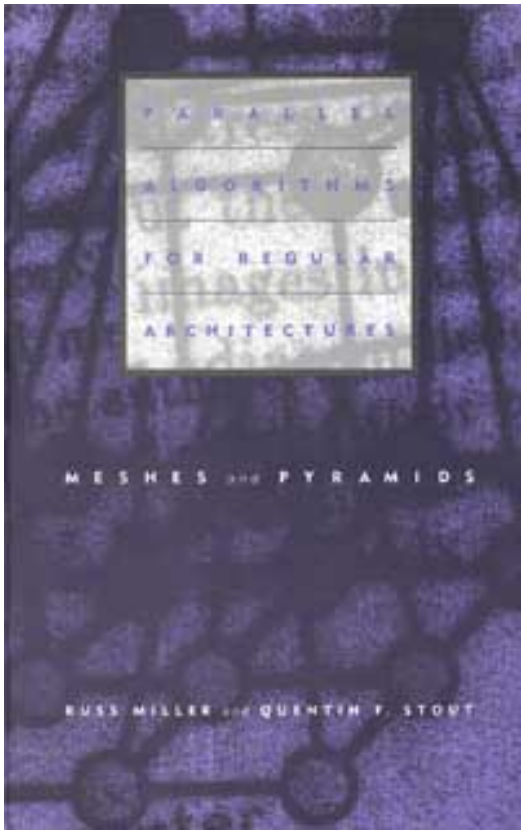





# Parallel Algorithms for Regular Architectures: Meshes and Pyramids

Russ Miller
Quentin F. Stout

The MIT Press
Cambridge, Massachusetts
London, England





**Library of Congress Cataloging-in-Publication Data**

Miller, Russ.
Parallel algorithms for regular architectures: meshes and pyramids.

Bibliography: p.
1. Parallel programming (Computer science) 2. Algorithms.
3. Computer architecture. I. Stout, Quentin F. II. Title. III. Series
QA76.6.M5226 1996 005.1 87-35360
ISBN 0-262-13233-8



## Contents

## List of Figures

## Preface

This book is designed for a variety of purposes. As a research monograph, it should be of interest to researchers and practitioners working in the field of parallel computing. It may be used as a text in a graduate course on Parallel Algorithms, as a supplementary text in an undergraduate course on Parallel Algorithms, or as a supplementary text in a course on Analysis of Algorithms, Parallel Computing, Parallel Architectures, Computer Architectures, or VLSI arrays. It is also appropriate to use this book as a supplementary text in an advanced graduate course on Vision, Image Analysis, or Computational Geometry. Excerpts from preliminary versions of this book have been used successfully in senior undergraduate and first year graduate courses on Analysis of Algorithms, advanced graduate courses on Parallel Algorithms, graduate level seminars on Computational Geometry and Parallel Computing, and a first year graduate course on Computer Architecture.

The focus of this book is on developing optimal algorithms to solve problems on sets of processors configured as a mesh or pyramid. Basic algorithms, such as sorting, matrix multiplication, and parallel prefix, are developed, as are algorithms to solve fundamental problems in image processing, computational geometry, and graph theory. The book integrates and synthesizes material from the literature with new concepts, algorithms, and paradigms. The reader has the opportunity to gain insight into developing efficient parallel algorithms by following the design process presented by the authors, who originally developed the vast majority of the algorithms that are presented.

This book uses a consistent approach to derive efficient parallel solutions to problems based on

1. algorithmic techniques, showing how to apply paradigms such as divide-and-conquer, and

2. the development and application of fundamental data movement operations.

Such data movement operations play a role that is analogous to data structures in the sequential setting, in that they provide a framework for describing higher level operations in terms of lower level ones. The basic structure of the higher level algorithms is often unchanged, even though efficient implementations of these data movement operations will



vary among architectures, as will the times they require. The presentation of the material in this book is such that a reader should be able to adapt a given algorithm to a variety of machine models beyond those discussed here. In fact, many of the algorithms presented in this book have already been adapted in a straightforward fashion to related architectures, including the hypercube and a variety of bus-based mesh architectures.

In addition to researchers working in the area of parallel algorithms, this book can aid practitioners who need to implement efficient parallel programs. The fundamental algorithms and operations developed in this text can be incorporated into a wide range of applications, and the design and analysis techniques utilized can be exploited in an even greater range. The algorithms and paradigms that are presented can be adapted to multiprocessor machines with varying degrees of granularity and possessing a variety of processor configurations. For example, they can be utilized inside a single VLSI chip, on special-purpose parallel machines, on large parallel computers, or on intermediate systems.

**Overview of Chapters**

Chapter 1 is an introductory chapter that defines the computer models, problems to be solved, forms of input, and notation that will be used throughout the book. It also serves to introduce the concept of designing *machine independent parallel algorithms* in terms of abstract data movement operations. This concept can be viewed as the parallel analogue of designing sequential algorithms in terms of abstract data types, without regard to detailed implementation issues. Many of these data movement operations are defined in Chapter 1, while others are introduced in later chapters as they are needed.

Chapters 2, 3, and 4 focus on the *mesh* computer, presenting data movement operations, algorithms, lower bounds, and paradigms. Chapter 2 gives optimal algorithms for fundamental problems such as matrix multiplication, transitive closure, sorting, computing semigroup properties, and fundamental data movement operations. These results serve as the foundation for mesh algorithms presented in subsequent chapters. Chapter 3 gives optimal algorithms to solve graph and image processing problems. These algorithms solve problems such as labeling connected components, determining bridge edges, finding nearest neighbors in an image, and deciding whether or not figures are convex. Chapter 4 gives optimal algorithms to solve a variety of geometric problems. These algorithms solve problems such as locating nearest neighbors, determining



intersections among objects, and finding the area covered by a set of overlapping rectangles.

Chapters 5 and 6 focus on the *pyramid* computer, presenting data movement operations, algorithms, lower bounds, and paradigms. Chapter 5 introduces asymptotically optimal algorithms that exploit the (quad) tree connections that exist between layers of the pyramid. This chapter also presents optimal solutions to problems such as computing commutative semigroup operations, answering point queries, determining convexity properties of single figures, and deciding whether or not a given figure could have arisen as the digitization of a straight line segment. In Chapter 6, efficient algorithms are given that show that the pyramid is useful for more than simple tree-like operations. Fundamental data movement operations are derived for a variety of input formats and situations. Algorithms are given in terms of these operations to solve complex problems for graphs and images, problems such as determining connected components, determining the nearest neighbor of each figure, and determining convexity of every figure. These algorithms are significantly faster than those possible for the mesh.

Throughout the book, *image data* is assumed to be given in the form of a black/white digitized picture; *graph data* is given either as matrix input (an adjacency or weight matrix, as appropriate) or as unordered lists of edges; and *geometric data* is given as unordered sets of points, line segments, rectangles, circles, etc. For some geometric problems, and for many of the data movement operations, the data has a label attached to each item and the problem being solved involves both the label and the associated data. For example, one might want to determine, for each label, the smallest value associated with the label.

**Recommended Use**

In an algorithms-based course, it is recommended that the presentation of the material commence with an introduction to some basic parallel models of computation, including the mesh, pyramid, mesh-of-trees, hypercube, and PRAM. At the discretion of the instructor, the tree and *x*-tree machine models might also be mentioned for the purpose of motivating the design of the pyramid in terms of its mix of mesh and tree interconnections. As each model is introduced, the *communication diameter* of the model should be discussed, since this serves as a lower bound on the running time for many fundamental problems. In addition, a 'wire-counting' (*bisection width*) argument is useful in terms of

discussing lower bounds on running times for more complex problems, such as sorting, that require extensive data movement. Finally, for each model, an algorithm to efficiently compute a semigroup operation (i.e., an associative binary operation, such as minimum, summation, or parity) can be described as a means of introducing some basic algorithmic techniques for the model. After introducing the models, either of the following approaches are recommended.

1. In an *architecture-oriented approach*, one would discuss a variety of problems and solutions for each model in sequence. First, one would look at a set of problems for the mesh, then a set of problems for the pyramid, and so forth.

2. In a *problem-oriented approach*, one would discuss algorithms and techniques to solve problem $P_1$ on a variety of architectures, then discuss algorithms and techniques to solve problem $P_2$ on a variety of architectures, and so forth. This would allow one to directly compare algorithms, paradigms, lower bounds, and running times within the same framework of problem and input definition. For this approach, one may want to first develop several data movement operations on all of the architectures, before discussing more advanced problems.

The first approach allows for a systematic traversal of the book, chapter by chapter. The second approach requires a comparison of related sections from different chapters.

**Correspondence**

We are interested in receiving any constructive criticism or suggestions that you might have. Please send all correspondence concerning this book to

*Parallel Algorithms for Regular Architectures*
Department of Computer Science
State University of New York
Buffalo, NY 14260 USA
**para-comments@cs.buffalo.edu**

The MIT Press maintains a home page on the World Wide Web at the following location:

**http://www-mitpress.mit.edu/**



This web site contains information about their books and journals, including a home page for *Parallel Algorithms for Regular Architectures: Meshes and Pyramids*. For those who wish to access the web site for this book directly, it can be found at the following location:

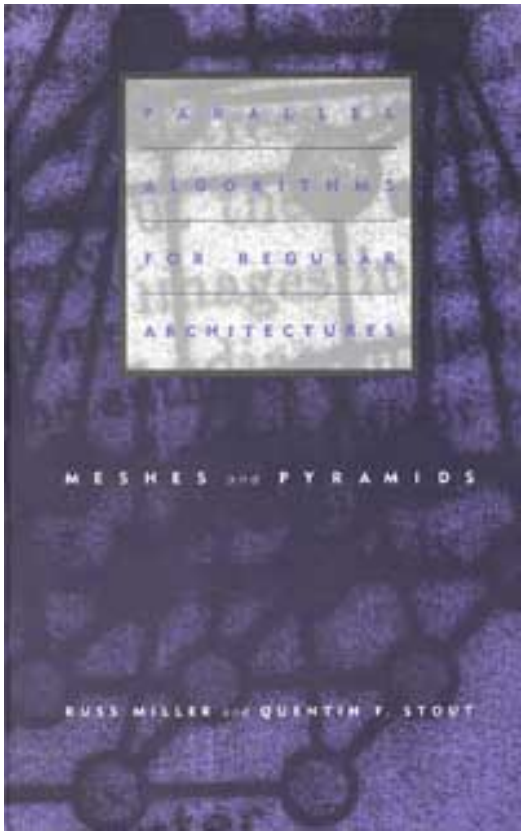
**http://www-mitpress.mit.edu/mitp/recent-books/comp/mileh.html**

The home page for this book contains up-to-date information about this project, including corrections, suggestions, and hot links to interesting parallel computing web sites.

**Acknowledgments**

The authors would like to express their appreciation to Peggy Newport, Susan Miller, and Leo Thenor for drawing many of the figures. In particular, special thanks to Peggy and Mike Newport for recently updating most of the figures. In addition to the students in a variety of graduate level parallel algorithms classes at the State University of New York at Buffalo and the University of Michigan, the authors would like to thank Mike Atallah, Gregory Bachelis, Johnnie Baker, Larry Boxer, Ed Cohen, Richard Fenrich, Susanne Hambrusch, Renaud Laurette, Dar-Shyang Lee, Marilynn Livingston, Susan Miller, Franco Preparata, Andrew Rau-Chaplin, Todd Sabin, Leonard Uhr, and Bette Warren, for reading early drafts of the book and making useful suggestions. Special thanks go out to Andrew Rau-Chaplin for his assistance in evaluating the final version of this manuscript. The authors would also like to thank Devon Bowen, Tony Brancato, Amy Hendrickson, Ken Smith, Davin Milun, and the DCO staff of EECS at the University of Michigan, for supporting a variety of hardware and software systems that were used in the preparation of this manuscript. The authors would like to extend a very special thanks to our original editor, Terry Ehling, for her extreme patience and support. Thanks also go out to our editor, Bob Prior, and The MIT Press for showing confidence in this project. Finally, the encouragement of Janis Hardwick during the completion of this book is greatly appreciated.

RUSS MILLER & QUENTIN F. STOUT, 1996






# 1 Overview

## 1.1 Introduction

Advances in VLSI technology have provided a cost-effective means of obtaining increased computational power by way of multiprocessor machines that consist of anywhere from a few processors to many thousands and potentially millions of processors. These processors cooperate in various ways to solve computationally intensive problems. While multiprocessor machines are targeted at increased performance, such architectures are vastly different from the single processor computing machines that are so prevalent. It is not surprising, therefore, to find that designing algorithms to exploit the (massive) parallelism available from these multiprocessor machines is a subject of intense research, since designing efficient algorithms for parallel machines is vastly different from designing efficient algorithms for single processor machines.

From a programmer's point of view, it would be ideal to develop parallel algorithms for a *parallel random access machine* (*PRAM*). A PRAM is a machine consisting of numerous identical processors and a global memory, where all processors have the ability to access any memory location in the same fixed unit of time, regardless of how large the memory is or how many processors are available. Unfortunately, due to current technological limitations, PRAMs cannot be built without significant delays in the access time, unless very few processors are used and the memory is limited. Some bus-based machines with a small number of processors are conceptually similar in design to a PRAM. However, with current technology, such machines cannot scale to thousands of processors while retaining the same time unit of access to global memory.

Machines that consist of numerous processors typically take the approach of having local memory attached to every processor, and using some interconnection network to relay messages and data between processors. Examples of such machines include the Massively Parallel Processor (MPP) with 16,384 processors interconnected as a square grid [Batc81, Pott85]; the Thinking Machines Corporation's CM1 and CM2, with 65,536 processors interconnected as a square grid and as a hypercube [Hill85]; the Thinking Machines Corporation's CM5, with thousands of processors interconnected as a fat-tree; the Intel iPSC



and NCube hypercubes with hundreds of processors [Inte86, HMSC86]; the Intel Paragon with hundreds of processors interconnected as a two-dimensional torus; and the Cray T3D with hundreds of processors interconnected as a three-dimensional torus.

Unfortunately, the interconnection networks usually have the property that not all pairs of processors can communicate with the same delay, and so performance concerns dictate that programs should minimize communication between processors that have large delays between them. To obtain highly efficient programs, this apparently requires writing different programs for each different interconnection network. However, this book attempts to show that the situation is not as bad as it seems, in that often the same algorithmic approach can be used on a wide range of networks and still yield efficient implementations. To help achieve machine independence, many of the algorithms are expressed in terms of fundamental data movement operations. That is, for a set of parallel models that exhibit certain common underlying traits, such as the mesh, pyramid, mesh-of-trees, and hypercube, parallel algorithms for certain classes of problems can be written in terms of fundamental data movement operations. These data movement operations can be viewed as taking the place of abstract data types that are used for designing machine and language independent serial algorithms. It is important to realize that an efficient implementation of a data movement operation is typically strongly dependent upon the interconnection network. However, such an effort allows for higher level algorithms to be written with a great deal of network independence.

The algorithmic problems considered in this book are chosen predominantly from the fields of image processing, graph theory, and computational geometry. Many of the algorithms rely on efficient sorting and matrix algorithms, which are also presented. The paradigms exhibited by these algorithms should give the reader a good grasp on techniques for designing parallel algorithms.

Each chapter is reasonably self-contained, so the book need not be read in a linear fashion. However, later chapters in the book do assume a knowledge of the material that is presented in the remainder of this introductory chapter.

Section 1.2 discusses notation and parallel models of computation. Section 1.3 describes a variety of input formats for the problems considered throughout the book, while Section 1.4 focuses on defining the specific problems. Generic descriptions of fundamental data movement operations are given in Section 1.5. Finally, Section 1.6 serves to synthesize the material presented in these earlier sections and introduce



fundamental paradigms for designing efficient parallel algorithms. This is accomplished by giving generic parallel algorithms in terms of abstract data movement operations to solve two fundamental problems with various input formats.

## 1.2 Models of Computation

In this section, notation and general parallel models of computation are discussed. In addition, specific models are defined for which algorithms will be presented in later chapters (or the next volume) of the book.

### *1.2.1 Preliminaries*

Throughout the book, $\Theta$, O, $\Omega$, $o$, and $\omega$ notation are used, where $\Theta$ means 'order exactly', O means 'order at most', $\Omega$ means 'order at least', $o$ means 'order less than', and $\omega$ means 'order greater than'. For formal definitions and some examples of this notation, the reader is referred to Appendix A.

Many of the algorithms developed in the book are recursive, often involving parallel divide-and-conquer solution strategies. As a result, the running times for these algorithms are often expressed in terms of recurrence equations. General solutions to most of the recurrences that are used throughout the book are given in Appendix B.

### *1.2.2 Classification Schemes*

In a *distributed memory*, or *local memory*, machine, each memory cell is attached to a specific processor, and a processor can directly access only the memory attached to it. For processor 1 to access the contents of a memory cell attached to processor 2, a message containing a copy of the memory cell in processor 2 must be sent to processor 1. Distributed memory systems are also known as *message-passing* systems. A distributed memory system is often interpreted as not having a global addressing scheme for memory, just local addressing within each processor, though logically the (processor ID, local address) pair forms a global address. All of the machine models considered in later chapters have distributed memory.

In a *shared memory* machine, memory is equally accessible to all processors, using a global addressing mechanism. Typically, in a shared memory machine processors do not directly communicate with

each other, but rather through the shared memory. Originally, shared memory was interpreted as meaning all access took the same time, but this is hard to achieve in practice. For this reason, the *nonuniform memory access* (*NUMA*) model is more realistic. For example, machines from Kendall Square Research and Silicon Graphics Incorporated implement the shared memory model while maintaining physically distributed memory over the processors. While each processor can access all memory, accesses to local memory is typically an order of magnitude faster than accesses to memory in other processors.

A *single instruction multiple data* (*SIMD*) machine typically consists of *n processors*, a control unit, and an interconnection network or interconnection function. The control unit stores the program and broadcasts the instructions to all processors simultaneously. Active processors execute the instruction on the contents of their own local memory. Through the use of a *mask*, processors may be in either an active or inactive state at any time during the execution of the program. Each processor is connected via a unit-time bidirectional communication link to each of its *neighbors*. A *unit of time* is generally defined to be the time necessary for each processor to execute some fixed number of arithmetic and Boolean operations on the contents of its local memory, as well as to send and receive a piece of data from each of its neighbors.

A *multiple instruction multiple data* (*MIMD*) machine typically consists of *n* processors, *n* memory modules, and an interconnection network. In contrast to the single instruction stream model, the multiple instruction stream model allows each of the *n* processors to store and execute its own program. Processors are coupled with memory modules, and are connected to each other through a fixed interconnection scheme by bidirectional unit-time communication links.

Variants to the SIMD and MIMD descriptions just given are possible. For instance, one popular variant is to uncouple memory from the processors, and to allow the interconnection network to link processors to each other and to the memory modules.

For distributed memory parallel computers, such as those discussed in this book, information is exchanged as messages between processors, and hence the distance information travels becomes a dominant consideration. While the logical arrangement of information in data structures plays a major role in serial algorithms, the physical arrangement of information plays a major role in algorithms for these distributed memory parallel computers. One uses data movement operations in parallel computers to perform the physical movement needed, much as one uses operations on data structures in serial computers.



To determine the communication time required to solve a problem on a given parallel machine, two methods often aid in determining simple lower bounds. The *distance between two processors* in a network is defined to be the minimum number of communication links information needs to traverse to get from one to the other. The *communication diameter* of a network is defined to be the maximum distance between any two processors in the network. Therefore, the communication diameter of a machine gives a lower bound on the running time for problems where data needs to be exchanged between processors at maximum distance. As will be shown later in the book, the communication diameter is sometimes an overly optimistic lower bound for certain problems and machine models.

Another method of determining lower bounds for problems that require extensive data movements is by a *wire-counting* (*wire-cutting*, *cut-set*, *bandwidth*, *bisection width*) argument. For instance, suppose one is concerned with the minimum time necessary to sort or route data on a particular machine, and it can be shown that in the worst-case, all of the data from one 'half' of the machine must be exchanged with all of the data from the other 'half' of the machine. If there are $w$ wires that connect the two halves of the machine, then in 1 unit of time only $2w$ elements can cross these $w$ bidirectional communication wires. Therefore, if each half of the machine has $n/2$ pieces of data, then $n/2w$ time is required simply to move data between the two halves of the machine.

The term *granularity* is often used to refer to the number and complexity of processors in a parallel system. A *fine-grained system* has large numbers of relatively simple computational units, each having relatively little memory. Since the individual processors cannot hold much data, they must communicate frequently in order to do anything productive. For example, the neurons in the brain are a fine-grained system. In a *course-grained system* there are few, powerful processors, each with a large amount of memory. This enables each processor to do a significant amount of calculation using only the data in its own memory. For example, a network of workstations is a coarse-grained system. Since the communication to calculation ratio is relatively high in fine-grained systems, they tend to be implemented so that the time of communication is close to that of the time of a calculation, while in some (but not all) coarse-grained systems, the time for communication is very high compared to the time for calculation. With current technology, fine-grained parallel computers have on the order of 10,000 simple processors, while coarse-grained parallel computers have on the order of 10 powerful processors. A particularly interesting area of research is designing



algorithms to exploit *medium-grained machines*, which consist of, say, 100s of microprocessors processors that are a compromise in performance and size between processors of fine-grained and coarse-grained machines. The majority of general-purpose parallel supercomputers today can be classified as medium-grained machines.

In general, SIMD machines are thought of (and constructed) as fine-grained machines, where all processors operate in lockstep fashion on the contents of their own small local memory. MIMD machines are more often thought of as coarse-grained machines that either share a global memory or have the memory distributed among the processors.

Many of the algorithms presented in this book for a given machine will require that a region of the machine simulate a larger region of the same type. For instance, if the region consists of $n$ processors, the algorithm might require that the region simulate a $cn$ processor region, for some constant $c$. This can usually be accomplished in a straightforward manner by having each processor of the region simulate $c$ processors. Notice that this will adversely affect the running time of an algorithm by a multiplicative constant.

In all of the models described, it is assumed that every processor has a fixed number of registers (words), each of size $\Omega(\log n)$, and can perform standard arithmetic and Boolean operations on the contents of these registers in unit time. Each processor can also send or receive a word of data from each of its neighbors in unit time. Each processor will contain a unique identification register, which provides an ordering to the processors (though occasionally different orderings may also be used). The contents of this register will be specified for each model.

### *1.2.3 Mesh Computer*

The *mesh computer* (*mesh*) *of size n* is a machine with $n$ simple processors arranged in a square lattice. To simplify exposition, it is assumed that $n = 4^c$, for some integer $c$. For all $i, j \in [0, \ldots, n^{1/2} - 1]$, processor $P_{i,j}$, representing the processor in row $i$ and column $j$, is connected via bidirectional unit-time communication links to its four *neighbors*, processors $P_{i\pm1,j}$ and $P_{i,j\pm1}$, assuming they exist. (See Figure 1.1.)

Each processor contains its row and column indices, and the identification register is initialized to the processor's row-major index, shuffled row-major index, snake-like index, or proximity order index, as shown in Figure 1.2, depending on the application. (If necessary, these values can be generated in $\Theta(n^{1/2})$ time.)

The communication diameter of a mesh of size $n$ is $\Theta(n^{1/2})$, as can be



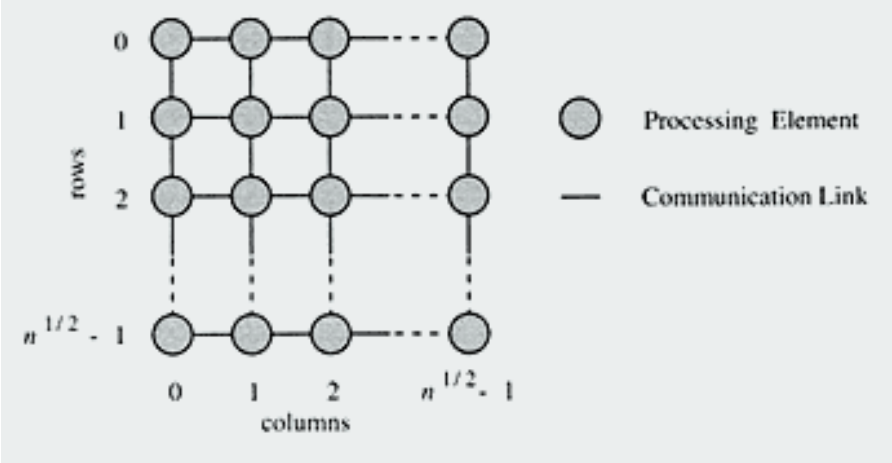



Figure 1.1:
A mesh computer of size $n$.

seen by examining the distance between processors in opposite corners of the mesh. This means that if a processor in one corner of the mesh needs data from a processor in another corner of the mesh sometime during an algorithm, then a lower bound on the running time of the algorithm is $\Theta(n^{1/2})$.

There are some variations of the mesh that deserve mention. Moore's pattern of connecting each processor to its 8 nearest neighbors [Moor62] has been implemented in the MasPar MP1 and MP2, and Golay's use of a hexagonal decomposition of 2-dimensional space where each processor communicates with its 6 nearest neighbors [Gola69] has been implemented in the HARTS machine [CSK90]. Other interesting variations are derived from connecting the boundaries of the mesh to form a cylinder (north-south or east-west), torus (doughnut), spiral, and so on. In fact, the mesh topology of the Loral's Massively Parallel Processor (MPP) is software configurable to select the interconnection of the border elements [Pott85], and the Intel Paragon machine uses a 2-dimensional torus pattern. While toroidal connections reduce the communication diameter by a factor of 2, as do the 8-nearest neighbor connections, in an O-notational sense such differences are masked. Therefore, only the simple mesh of Figure 1.1 will be considered.

A more significant change is to require that the word size be $\Theta(1)$ instead of $\Theta(\log n)$. This model is known as a *cellular automata*, *iterative array*, *parallel processing array*, or *mesh automata*. It is equivalent to requiring that all processors be copies of some fixed finite state automaton. Cellular automata are quite popular for modeling physical and biological



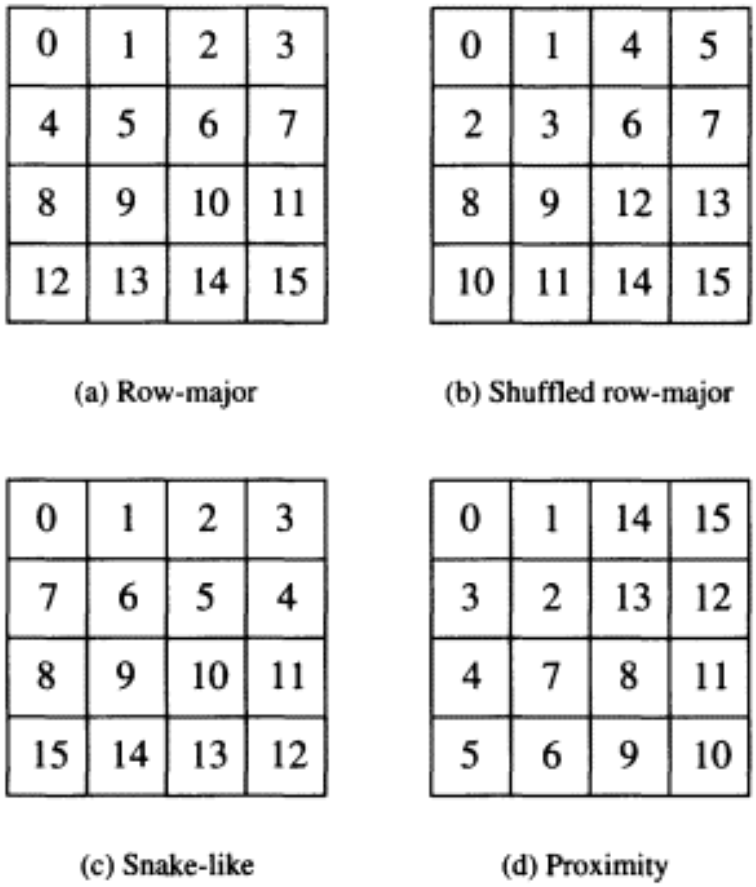



Figure 1.2:
Indexing schemes for the processors of a mesh.



phenomena such as crystal growth, phase transitions, and plant growth. However, as a computational model, for any fixed automaton, once $n$ is sufficiently large, a processor does not have enough memory to store its ID or coordinates, which seriously complicates matters. While cellular automata were widely studied as a computational model (e.g., [Beye69, Gola69, Gray71, Levi72, Moor62, Stou82b, Stou83a, Unge59, Unge62, VanS80]), the more powerful mesh model is used for general purpose computing. To the best of the authors' knowledge, all real mesh computers have processors capable of storing their coordinates. There are also more powerful variations, such as the mesh computer augmented with broadcasting [Stou86a], but their study is outside the bounds of this book.

### *1.2.4 Pyramid Computer*

A *pyramid computer* (*pyramid*) *of size n* is a machine that can be viewed as a full, rooted, 4-ary tree of height $\log_4 n$, with additional horizontal links so that each horizontal *level* is a mesh. It is often convenient to view the pyramid as a tapering array of meshes. A pyramid of size $n$ has at its base a mesh of size $n$, and a total of $^4/_3 n - {}^1/_3$ processors. The levels are numbered so that the base is level 0 and the apex is level $\log_4 n$. A processor at level $i$ is connected via bidirectional unit-time communication links to its 9 neighbors (assuming they exist): 4 siblings at level $i$, 4 children at level $i - 1$, and a parent at level $i + 1$. (A sample pyramid is given in Figure 1.3.) Each processor contains registers with its level, row, and column coordinates, the concatenation of which are in the processor identification register. These registers can be initialized in $\Theta(\log n)$ time if necessary.

One advantage of the pyramid over the mesh is that the communication diameter of a pyramid computer of size $n$ is only $\Theta(\log n)$. This is true since any two processors in the pyramid can exchange information through the apex. In Chapter 5, algorithms with a running time of $\Theta(\log n)$ are presented to solve a variety of problems on a pyramid of size $n$. Of course, if too much data is trying to be passed through the apex, then the apex becomes a bottleneck. In Chapter 6, it is shown that for a variety of problems on a pyramid computer of size $n$, the $\Omega(\log n)$ lower bound is overly optimistic and must be replaced by a bound closer to $\Omega(n^{1/4})$. Even to attain this larger bound, algorithms must avoid operations that require extensive data movement, since a simple wire-counting argument shows that $\Omega(n^{1/2})$ time is required for communication-intensive problems such as sorting or routing all of the



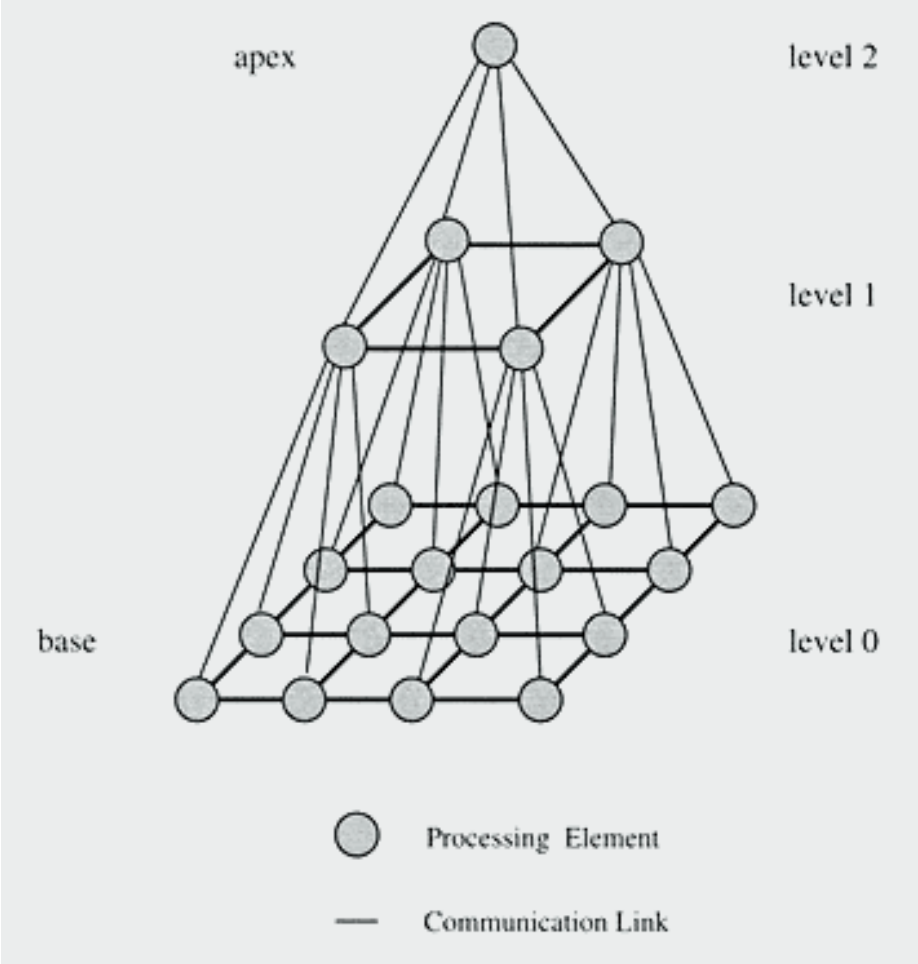



Figure 1.3:
A pyramid computer of size 16.

data in the base. To see this, consider the number of wires crossing the middle of the pyramid versus the number of items that potentially must move from one half to the other. In the base of the pyramid there are $n^{1/2}$ wires crossing the middle of the pyramid, in the next level there are $\frac{n^{1/2}}{2}$ such wires, and so on, giving the total number of wires that crosses the middle of a pyramid of size $n$ to be $\sum_{i=0}^{\log_4(n)-1} \frac{n^{1/2}}{2^i}$ which is $2n^{1/2} - 2$. Since all $n$ pieces of data that initially reside in the base of the pyramid may need to cross from one side of the base mesh to the other, then $\left\lceil \frac{n}{2n^{1/2}-2} \right\rceil$ time units, or $\Omega(n^{1/2})$ time is required just to get data across the middle of the pyramid.

The reader is referred to [Buva87, CFLS85, ClMe87, FKLV83, Scha85, SHBV87, Tani82a, Uhr84] for a description of constructed and proposed pyramid computers.

### *1.2.5 Mesh-of-Trees Architecture*

A *mesh-of-trees of base size n*, where $n$ is an integral power of 4, has a total of $3n - 2n^{1/2}$ processors. $n$ of these are base processors arranged as a mesh of size $n$. Above each row and above each column of the mesh is a perfect binary tree of processors. Each row (column) tree has as its leaves an entire row (column) of base processors. All row trees are disjoint, as are all column trees. Every row tree has exactly one leaf processor in common with every column tree. Figure 1.4 shows a sample mesh-of-trees. Each base processor is connected to 6 neighbors (assuming they exist): 4 in the base mesh, a parent in its row tree, and a parent in its column tree. Each processor in a row or column tree that is neither a leaf nor a root is connected to exactly 3 neighbors in its tree: a parent and 2 children. Each root in a row or column tree has its 2 children as neighbors. Each processor contains identity registers with its level, row, and column coordinates (the base being level 0), the concatenation of which are the contents of the processor identification register.

Like the pyramid, the mesh-of-trees also has a communication diameter proportional to the logarithm of the number of base processors. Also, like the pyramid, a simple wire-counting argument shows that for operations that require extensive data movement, such as sorting or routing, $\Omega(n^{1/2})$ time is required since only $2n^{1/2}$ wires cross the middle of the mesh-of-trees. However, the mesh-of-trees can sort a restricted amount of data given in certain configurations in $\Theta(\log n)$ time.

While no significant mesh-of-trees has been built, it is a very useful



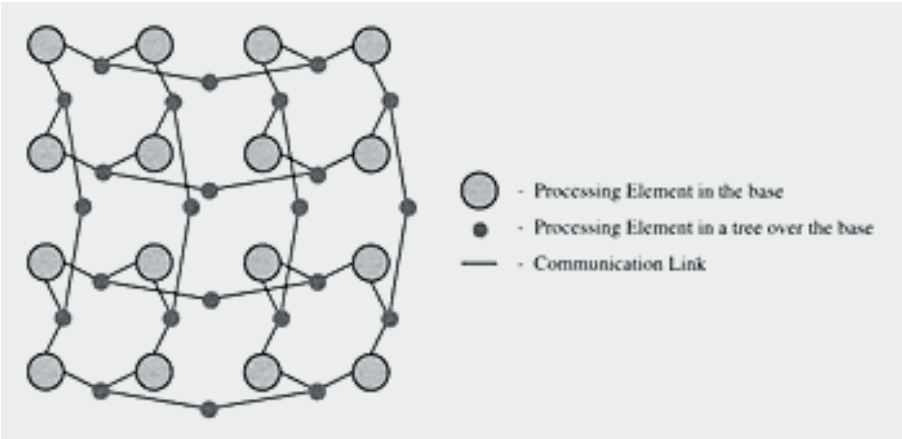


Figure 1.4:
A mesh-of-trees of base size $n = 16$.
Note: The mesh connections have been omitted for clarity.

architecture in VLSI because it embeds nicely into the plane [Ullm84]. Usually the mesh-of-trees does not include the connections between the base processors, but these connections seem particularly natural when one is processing images in which some of the lowest level operations involve comparing adjacent pixels. It is easy to show that these additional connections do not change the planar embedding properties of the mesh-of-trees.

### *1.2.6 Hypercube*

A *hypercube of size n*, where $n$ is an integral power of 2, has $n$ processors indexed by the integers $\{0, \ldots, n - 1\}$. Viewing each integer in the index range as a $\log_2 n$-bit string, two processors are connected via a bidirectional communication link if and only if their indices differ by exactly one bit. A hypercube of size $n$ is created recursively from two hypercubes of size $n/2$ by labeling each hypercube of size $n/2$ identically and independently with the indices $\{0, \ldots, n/2 - 1\}$, and then appending a 1 in front of the bit-strings of one of the cubes and a 0 in front of the other, which 'creates' a new link from each processor in one cube to the corresponding processor in the other cube. See Figure 1.5. The contents of the processor identification register corresponds to this label.

It is easy to see that like the mesh-of-trees and pyramid, the communication diameter of a hypercube of size $n$ is $\Theta(\log n)$. However, unlike the mesh-of-trees or pyramid, a wire-counting argument only shows that



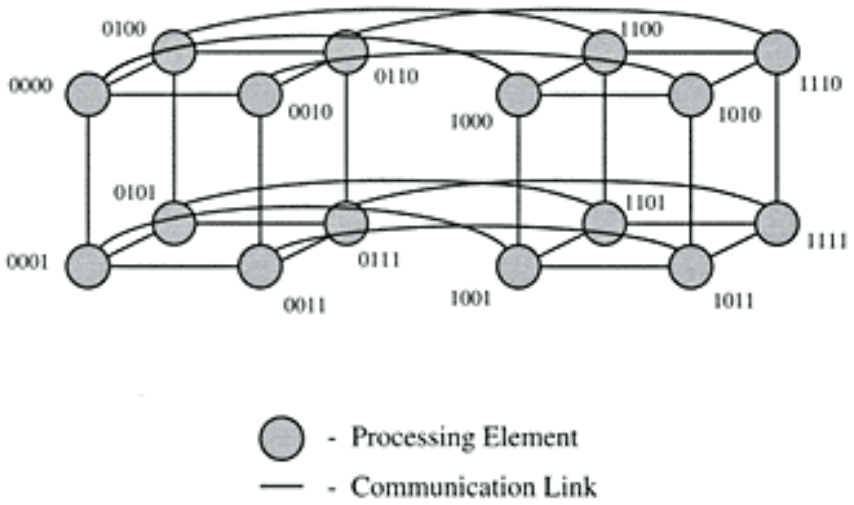



Figure 1.5:
A hypercube of size $n = 16$ with the nodes labeled using a binary representation.

$\Omega(1)$ time is required for operations that require extensive data movement, since there are $n/2$ wires that connect two hypercubes of size $n/2$ in a hypercube of size $n$. This allows for many problems to be solved more efficiently on the hypercube than on the mesh, pyramid, or mesh-of-trees.

A variety of hypercubes have been marketed commercially, including fine-grained machines such as the Connection Machine [Hill85], and medium-grained machines by companies such as Intel [Inte86], Ncube [HMSC86], FPS [GHS86], and Ametek [Amet86]. Some machines, such as the MasPar MP1 and MP2 and the Thinking Machines Corporation CM5, have communication properties that make them very similar to a hypercube for most purposes.

### *1.2.7 Pram*

A *parallel random access machine* (*PRAM*) is an idealized parallel model of computation, with a unit-time communication diameter. A PRAM is often described as a machine that consists of a set of identical processors and a global memory, where all processors have unit-time access to any memory location. Alternately, a PRAM can be described as a machine consisting of a set of fully connected processors, where memory is distributed among the processors so that each processor maintains some



(fixed) number of memory locations.

A PRAM is not a regular architecture, and the unit-time memory access requirement is not scalable, so they are not the focus of this book. However, in describing algorithms for regular architectures, it is often useful to describe a PRAM algorithm and then either perform a stepwise simulation of the PRAM operation on the target machine, or perform a higher-level transformation by using data movement operations. This is particularly true for algorithms involving the mesh-of-trees and hypercube. For this purpose, the alternate description of the PRAM will be used, which is now stated more formally.

A *PRAM of size n* consists of *n* processors connected as a complete graph. The processor identification register is set to a number from 0 to *n* - 1, so that no two processors have the same number. A *concurrent read*, *exclusive write* (*CREW*) *PRAM* permits multiple processors to read data from the same memory location simultaneously, but permits only one processor at a time to attempt to write to a given memory location. A *concurrent read*, *concurrent write* (*CRCW*) *PRAM* permits concurrent reads as above, but allows several processors to attempt writing to the same memory location simultaneously, with some tie-breaking scheme used so that only one of the competing processors succeeds in the write. An *exclusive read*, *exclusive write* (*EREW*) *PRAM* is the most restrictive version of a PRAM in that only one processor can read and write from a given memory location at a given time. An *exclusive read*, *concurrent write* (*ERCW*) *PRAM* is a machine that only allows one processor to read from a given memory location at a time, while allowing multiple processors to write to any given memory location at the same time, with some tie-breaking scheme used so that only one of the competing processors succeeds in the write.

The reader interested in pursuing algorithms for the PRAM may wish to refer to [KaRa90, JaJa92, Reif93].

## 1.3 Forms of Input

In this book, efficient algorithms are presented to solve problems on a variety of regular parallel architectures, defined in the previous section. The majority of the problems will be chosen from fields such as image processing, graph theory, and computational geometry. In this section, input formats for the problems considered in this book are given. For the mesh and hypercube, the input data is assumed to be distributed throughout all processors of the machine, while for the pyramid and

mesh-of-trees, the input data is assumed to be distributed among the base processors.

1. *Unordered Edge Input*. The edges of a graph are initially distributed in a random fashion throughout the processors of the machine. It is assumed that each edge of the graph is represented by a pair of vertices, that edges may be represented more than once, and that no processor contains more than some fixed number of edges.

2. *Matrix Input*. The processors of the machine are labeled in a systematic and consistent fashion so that processor $P_{i,j}$ initially contains the $(i, j)$ entry of the adjacency or weight matrix that represents a graph.

3. *Digitized Picture Input*. A digitized black/white picture is initially stored one *pixel* (*picture element*) per processor in a systematic and consistent fashion so that neighboring pixels in the picture are mapped onto neighboring processors in the machine. It is assumed that the interpretation is a black picture on a white background.

4. *Geometric Data Input*. The geometric problems considered in this book are all *planar*. That is, they occur in standard Euclidean 2-space. Geometric objects, and collections of such objects, are represented in a number of ways. For problems involving points or sets of points, it is assumed that the input is planar points represented as Cartesian coordinates, stored no more than some fixed number per processor. If the input is sets of points, then each point will also have an attached label indicating the set it belongs to. Circles are represented by their radius and the Cartesian coordinates of their center, and are stored no more than some fixed number per processor. Simple polygons (i.e., polygons that do not intersect themselves) are given as labeled line segments represented by the Cartesian coordinates of their endpoints, stored no more than some fixed number of line segments per processor.

For problems involving geometric objects, it is assumed that no two distinct points have the same $x$-coordinate or y-coordinate. It is also assumed that no two endpoints from line segments have the same $x$-coordinate or y-coordinate, unless the line segments share a common endpoint. These are common assumptions in computational geometry as they simplify exposition by eliminating special



cases. Further, by rotating the points slightly these assumptions can always be met.

## 1.4 Problems

This section highlights some specific problems for which efficient solutions are presented throughout the book. For convenience, these problems have been divided into two, not necessarily disjoint, broad areas: (a) graph and image problems, and (b) problems from computational geometry. Many of the algorithms to solve problems in graph theory, image processing, and computational geometry will rely on efficient sorting and matrix algorithms. Therefore, the reader should note that while sorting and matrix algorithms are not explicitly mentioned in the remainder of this section, such algorithms will be presented in later chapters of the book.

### *1.4.1 Graph and Image Problems*

In this section, several graph and image problems are defined for which solutions are presented throughout the book for a variety of machine models and input formats.

1. *Component Labeling*. The input to the problem is an undirected graph $G = (V, E)$, given as an adjacency matrix, a set of unordered edges, or as a digitized picture. It is assumed that the elements of $V$ have a linear order. The component labeling problem is to assign a component label to each vertex, such that two vertices receive the same component label if and only if there is a connected path between them. The component label will be chosen to be the minimum label of any vertex in the component. For digitized picture input, considered as black objects on a white background, components are created by considering black pixels to be vertices and pairs of neighboring black pixels to be undirected edges. For digitized picture input, the term *figure* will be used to refer to a (black) maximally connected component.

2. *Minimal Spanning Forest*. Given a weighted undirected graph, mark the edges of a minimal-weight spanning tree for each component of the graph. For a connected graph $G = (V, E)$, and a weight function $w$ that assigns a weight $w(e)$ to every edge $e \in E$, a minimal-weight spanning tree $T = (V, E')$ of $G$, $E' \subseteq E$, is



a connected graph with $|V|$ - 1 edges having the property that $\sum_{e \in E'} w(e)$ is a minimum. The input can be given as an adjacency matrix, a set of unordered edges, or a digitized picture.

3. *Nearest/Farthest Neighboring Component*. Given a digitized picture with its figures already labeled, determine for each figure the label of the nearest (farthest) figure to it and its corresponding distance. The $l_p$ metrics will be used to measure distance, where for $1 \leq p < \infty$, the $l_p$ distance from $(a, b)$ to $(c, d)$ is $[|a - c|^p + |b - d|^p]^{1/p}$, and the $l_\infty$ distance from $(a, b)$ to $(c, d)$ is $max\{|a - c|, |b - d|\}$.

The reader might note that the connection scheme of the mesh is based on the $l_1$ ("taxi-cab" or "city block") metric. This means that efficient solutions to image problems for mesh-based models are often easiest when expressed in terms of the $l_1$ metric. Further, simple techniques can also be applied to solve problems in terms of the $l\infty$ metric for mesh-based models. However, for other metrics, such as the important $l_2$ ("Euclidean") metric, more sophisticated solution strategies will be needed for mesh-based machines. For nonmesh-based machines, sophisticated solution strategies will be developed to solve distance problems for all $l_p$ metrics.

4. *Transitive Closure*. Compute the transitive closure, denoted $A^*$, of a symmetric Boolean matrix, $A$. If $A$ is interpreted as the adjacency matrix representing an undirected graph (i.e, $A(i, j) = A(j, i) = 1$ means there is an undirected edge between vertices $i$ and $j$, while $A(i, j) = A(j, i) = 0$ means no such edge exists), then $A^*(i, j) = 1$ if and only if vertices $i$ and $j$ are in the same connected component, and 0 otherwise.

5. *Bipartite Graphs*. Given an undirected graph $G = (V, E)$, decide if $G$ is bipartite. That is, decide whether or not $V$ can be partitioned into sets $V_1$ and $V_2$ so that each edge of $G$ joins a member of $V_1$ to a member of $V_2$.

6. *Cyclic Index*. Compute the cyclic index of an undirected graph $G = (V, E)$, where the cyclic index of $G$ is the largest number $s$ so that $V$ can be partitioned into sets $V_0, \ldots, V_{s-1}$, such that for any edge $(x, y)$, if $x \in V_i$ then $y \in V_{(i \pm 1) \bmod s}$.

7. *Bridge Edges*. Given an undirected graph, decide which edges are bridge edges, where an edge is called a bridge edge if its removal increases the number of components.



8. *Articulation Points*. Given an undirected graph, decide which vertices are articulation points, where a vertex is called an articulation point if its removal (along with the edges incident on it) increases the number of components.

9. *Biconnectivity*. Given an undirected graph, decide for each component whether or not it is biconnected, where a component is said to be biconnected if for any two points in the component there are two disjoint paths between them.

10. *Internal Distance*. Given a digitized picture, determine for each black pixel the distance of a minimal internal path (traversing only black pixels) to a marked pixel. Notice that the distance to a marked pixel is $\infty$ for all pixels not in the same figure as a marked pixel, and is otherwise defined to be the minimum number of pixels in an internal path to a marked pixel.

11. *Minimal Paths*. Given two sets of pixels, $A$ and $B$, mark and count the minimal distance internal paths from $A$ to $B$.

### *1.4.2 Computational Geometry Problems*

In this section, several problems from computational geometry are defined for which solutions are presented throughout the book. Many algorithms from computational geometry are based on the ability to efficiently determine the convex hull of an object. The convex hull is a geometric structure of primary importance that has been well studied for the serial model of computation [PrSh85, Sham78, Tous80, Avis79, Yao81]. It has applications to normalizing patterns in image processing, obtaining triangulations of sets of points, topological feature extraction, shape decomposition in pattern recognition, and testing for linear separability, to name a few.

The *convex hull* of a set of points $S$, denoted *hull*($S$) or $CH(S)$, is the smallest convex polygon $P$ for which each point of $S$ is in the interior or on the boundary of $P$, as shown in Figure 1.6. A point $p \in S$ is defined to be an *extreme point* of $S$ if and only if $p \notin \text{hull}(S - \{p\})$. That is, $p$ is an extreme point of hull ($S$) if and only if $p$ is on the boundary of hull($S$) at a point where a trace of the boundary results in a change of slope (i.e., $p$ is situated at a corner of the boundary). The edges between adjacent extreme points, with respect to a trace of the boundary of this polygon, will be referred to as the *edges of the hull* ($S$). Note that a convex polygon is completely determined by its extreme points.

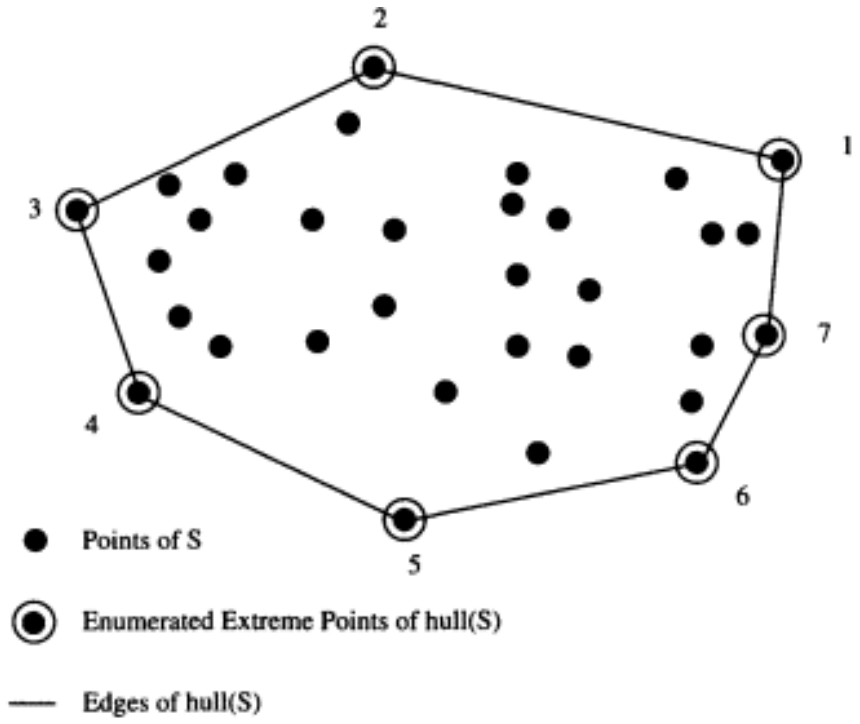



Figure 1.6:
Convex hull of $S$.

For algorithms involving the convex hull, it is often useful to be able to represent the orientation of an extreme point of $S$, or an edge of the convex hull of $S$, with respect to $S$. The orientation typically used makes use of the definition of *the angle of a half-plane*, which is in the range $[0, 2\pi)$. To define the angle of a half-plane $H$, translate the origin so that it lies on the edge of $H$. The angle of $H$ is the angle $\alpha$ such that $H$ contains all rays from the origin at angles $\alpha + \beta$ for $\beta$ in $(0, \pi)$. For example, when considering a half-plane determined by the $x$-axis, the angle of the upper half-plane is 0, while the angle for the lower half-plane is $\pi$.

For an extreme point $p$ of a set $S$, the *angles of support of p* are represented by an interval corresponding to the angles of half-planes with edges through $p$ which contain $S$. For example, if hull($S$) is an iso-oriented rectangle, then the angles of support of the northwest extreme point are $[\pi, 3\pi/2]$, the angles of support of the southwest extreme point are $[3\pi/2, 2\pi] \cup 0$, the angles of support of the southeast extreme point are $[0, \pi/2]$, and the angles of support of the northeast extreme point are $[\pi/2, \pi]$. For an edge $e$ of the hull of a set $S$, the *angle of incidence*

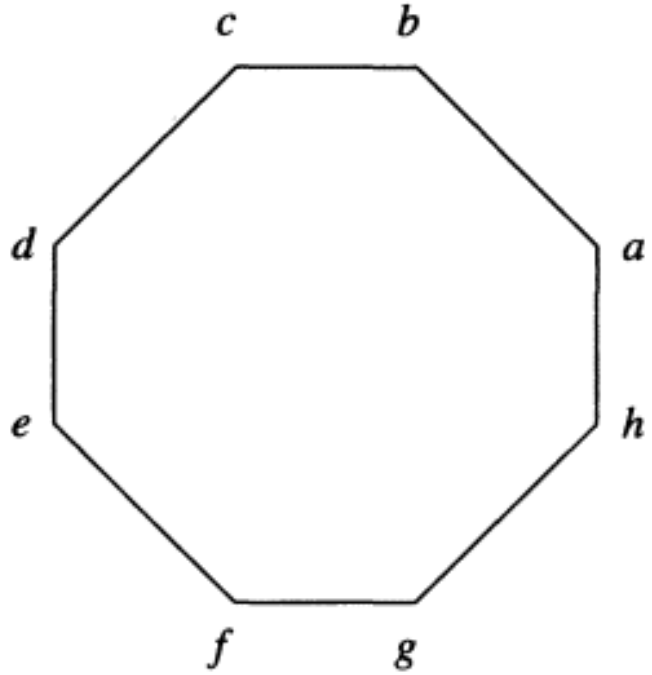



Figure 1.7:
The angle of incidence of hull edge $\overline{ha}$ is *0*, of edge $\overline{ab}$ is π/4,of edge $\overline{bc}$ is π/2, and so forth. The angles of support of extreme point a are [0, π/4], of point b are [π/4,π/2], of point c are [π/2,3π/4], and so forth.

*of e* is the angle of the half-plane containing *S* with edge containing *e*. See Figure 1.7.

For problems involving a digitized picture $D = \{d_{i,j}\}$ as input, the pixels are mapped in a natural fashion to the processors so that processor $P_{i,j}$ assumes responsibility for pixel $d_{i,j}$. For convexity and proximity problems, it often makes sense to identify processor $P_{i,j}$ with the integer lattice point $(i, j)$. In this setting, a set of processors (possibly corresponding to a set of pixels from a given digitized picture) is said to be *convex* if and only if the corresponding set of integer lattice points is convex, i.e., the smallest convex polygon containing them contains no other integer lattice points. This is the proper notion of convexity for integer lattice points, but it does have the annoying property that some disconnected sets of points, such as {(1,1), (3,4)}, are convex.

For several of the algorithms presented, it will be useful to impose an ordering on the extreme points of *S*. The ordering will be in a coun-



terclockwise fashion, starting with the easternmost point. Notice that for a given machine, the size of the data is finite. Therefore, in the case of picture data, if there are multiple easternmost points, then the southernmost one of these is chosen as the starting point in the counterclockwise ordering. For point data input, as discussed in Section 1.3, it is assumed that no two points have the same *x*-coordinate. Therefore, for point data input there exists a unique easternmost point.

For many of the convexity problems presented in the book, it is said that *the extreme points of S have been identified*, and hence hull(*S*) has been identified, if for each processor *Q* containing a point of *S*,

1. $Q$ has a Boolean variable 'extreme', and 'extreme' is true if and only if the point contained in $Q$ is an extreme point of $S$, and

2. for every processor $Q$ containing an extreme point of $S$
(a) $Q$ contains the position of its point in the counterclockwise ordering,
(b) $Q$ contains the total number of extreme points, and
(c) $Q$ contains the positions of its adjacent extreme points in the counterclockwise ordering.

Several problems from the field of computational geometry are now defined. Solutions to these and related problems are presented throughout the book for a variety of machine models and input formats.

1. *Convex Hull Problems*. This book considers a variety of problems involving convex hulls. One of the fundamental convex hull problems for a labeled digitized picture, or for sets of planar points, is to identify the extreme points of the convex hull for each (labeled) set of points. Given digitized picture input, another common query is to determine whether or not each figure is convex. A solution to the convex hull identification problem will often be used as a first step to solving many of the problems described below.

2. *Linear Separability*. Given a digitized picture or planar point data, determine if two sets of points, say $A$ and $B$, are linearly separable. That is, determine whether or not there exists a straight line in the plane such that all of $A$ lies to one side of the line and all of $B$ lies to the other side.

3. *Smallest Enclosing Figures*. Given a digitized picture or planar point data, determine for each set of points a smallest enclosing box, the smallest enclosing circle, and a smallest enclosing triangle.



4. *External Diameter*. Given a metric $d$ and either a digitized picture or planar point data, determine the external diameter for each set of points, where the external diameter of a set $S$ is

$$max\{d(s_1, s_2) | s_1, s_2 \in S\}$$

5. *Nearest Problems*. Given geometric data as input, find for each point, set of points, line segment, rectangle, or simple polygon, the nearest point, set of points, line segment, rectangle, or simple polygon, respectively.

6. *Minimal Distance Spanning Tree*. Given a set of planar points, determine a minimal distance spanning tree, where the distances are measured using the Euclidean metric.

7. *Intersection Problems*. For each line segment, set of line segments, rectangle, convex bull, half-plane, circle, or simple polygon determine intersection information with other line segments, sets of line segments, rectangles, half-planes, circles, or simple polygons, respectively.

8. *Area Problems*. Given a set of rectangles or, more generally, simple polygons, determine the total area covered.

**1.5 Data Movement Operations**

In designing serial algorithms, a major concern is the proper choice and implementation of data structures and their associated algorithms. In designing algorithms for parallel machines with regular interconnection topologies, the data structure is typically determined by the machine model. That is, the physical interconnection topology of the processors will determine the data structure. Therefore, in order to design efficient parallel algorithms for regular architectures, efficient operations are required to manipulate the data by exploiting the interconnection network.

Recently, there has been a trend towards developing cost-effective serial systems in terms of *abstract data types* (*ADTs*), where an ADT consists of an abstract data structure (e.g, list, tree, stack) together with a set of basic operations to be performed on the data in the structure (e.g, find, insert, push). The advantage of designing systems in terms of ADTs is that it allows the system to be designed with the essential properties of the data type in mind, but without worrying about implementation



constraints and details of the specific machine. In this book, *abstract data movement operations* are viewed as the parallel analogue of ADTs. That is, parallel algorithms can be expressed in terms of fundamental data movement operations without worrying about their implementation or the specific interconnection of the processors.

Most of the algorithms given in this book are expressed in terms of fundamental data movement operations, such as those defined in this section. Sorting is a central data movement operation, since several other operations assume that the data is already in sorted order. In addition, sorting is often used for the purpose of routing data. This will be discussed in some detail during the presentation of the concurrent read and concurrent write operations (pages 24-26). Several of the data movement operations that are presented in this section are performed in parallel on disjoint consecutive sequences of items in the sorted order. These sequences will be referred to as (*ordered*) *intervals*.

1. *Sorting:* Suppose there is a linear ordering of a collection of processors $P$, and there is data $D$ chosen from a linearly ordered set, with $D$ distributed one item per processor in arbitrary order. Then a sort operation will move the elements of $D$ so that they are stored in order with respect to $P$, one element per processor. Since general-purpose sorting algorithms are of interest, only comparison-based algorithms will be considered in this book.

2. *Merging:* Suppose there is a linear ordering of a set of processors $P$, and that a set of data $D$ is chosen from a linearly ordered set. Suppose $D$ is partitioned into subsets $D_1$ and $D_2$, and $D_1$ is stored in order, one item per processor, in one subset $P_1$ of the processors, and $D_2$ is stored in order, one item per processor, in $P$ - $P_1$ . Then a merge operation will combine $D_1$ and $D_2$ to store $D$ in sorted order in $P$, one item per processor.

Merging can be used not only to develop efficient sorting algorithms, but to develop efficient algorithms that avoid sorting in favor of merging data that is given as ordered subsets. Since merging can be performed at least as fast as sorting, it is desirable to design algorithms that favor merging over sorting when the situation allows. This scenario will arise in intermediate stages of many of the algorithms given in later chapters of the book.

3. *Semigroup Computation:* Suppose each processor has a record with data and a label, and that these records are in sorted order by their

label. Determine the result of applying a unit-time associative binary operation to all data items with the same label, with each processor receiving the answer for its data label. Such an operation is often referred to as a *semigroup operation*. Examples of semigroup operations include minimum, maximum, summation, logical-OR, logical-AND, and parity, to name a few.

4. *Broadcast/Report:* Broadcasting and reporting are often viewed and implemented as inverse operations. Both operations involve moving data within disjoint ordered intervals. They also both require a distinct processor, called the *leader*, of each interval. In broadcasting, the leader of each ordered interval has a data item which is to be delivered to all other processors in its interval. In reporting, within each interval all processors have data. A semigroup operation (i.e., an associative binary operation such as minimum, summation, or parity) is to be applied to the data, with the result being sent to the leader of the interval. For example, suppose each processor contains a labeled record and that processors with the same label form an ordered interval. Then broadcast and report may be used to inform all processors of the result of applying some semigroup operation over its ordered interval as follows. For ease of explanation, let the semigroup operation be minimum. First, the minimum data value of each interval is reported to the leader of the interval, and then the leader of each interval broadcasts this minimum value to all processors in its interval.

5. *Scatter/Gather:* These operations are closely related to the broadcast and report operations. Suppose each processor has a record containing data, a label, and a Boolean flag called 'marked'. Further, assume that all processors containing records with the same label form an ordered interval. A *gather* operation has all records within each interval with marked=*true* sent to the leader of its interval. The *scatter* operation is a natural complement to the gather operation, where the leader of each interval sends a (potentially different) piece of data to some set of processors in its interval.

6. *Concurrent Read/Write:* A *concurrent read* may be used in a situation where a set of processors wishes to obtain data associated with a set of keys, but where there is no a priori knowledge as to which processor maintains the data associate with any particular key. For example, processor $P_i$ might need to know the data asso-



ciated with the key 'blue,' but might not know which processor in the system is responsible for maintaining the information associated with the key 'blue.' In fact, all processors in the system might be requesting one or more pieces of data associated with, not necessarily distinct, keys. Similarly, a *concurrent write* may be used in a situation where a set of processors wishes to update the data associated with a set of keys, but again do not necessarily know for any key, which processor is responsible for maintaining the data associated with that key. These concurrent read/write operations generalize the read/write operations of a PRAM by making them associative, i.e., locating data by key rather than by address.

In order to maintain consistency during concurrent read and concurrent write operations, it will be assumed that there is at most one *master record*, stored in some processor, associated with each unique key. In a concurrent read, every processor generates one *request record* corresponding to each key that it wishes to receive information about (a bounded number). A concurrent read allows multiple processors to request information about the same key. A processor requesting information about a nonexistent key will receive a null message at the end of the operation.

One implementation of a concurrent read on a parallel machine with $n$ processors, based on previously defined data movement operations, follows.

(a) Every processor creates $C_1$ *master records* of the form (Key, Return Address, data, 'MASTER' ), where $C_1$ is the maximum number of keyed master records maintained by any processor, and Return Address is the index of the processor that is creating the record. (Processors maintaining less than $C_1$ master records will create dummy records so that all processors create the same number of master records.)

(b) Every processor creates $C_2$ *request records* of the form (Key, Return Address, data, 'REQUEST'), where $C_2$ is the maximum number of request records generated by any processor, and Return Address is the index of the processor that is creating the record. (Processors requesting information associated with less than $C_2$ master records will create dummy records so that all processors create the same number of request records.) Notice that the data fields of the request records are presently undefined.



(c) Sort all $(C_1 + C_2)n$ records together by the Key field. In case of ties, place records with the flag 'MASTER' before records with the flag 'REQUEST.'

(d) Use a broadcast within ordered intervals to propagate the data associated with each master record to the request records with the same Key field. This allows all request records to find and store their required data.

(e) Return all records to their original processors by sorting all records on the Return Address field.

Therefore, the time to perform a concurrent read, as described, is bounded by the time to perform a fixed number of sort and interval operations.

As with the concurrent read, in the concurrent write it will be assumed that there is at most one *master record*, stored in some processor, associated with each unique key. In a concurrent write, processors generate *update records* which specify the key and piece of information about that key that they wish to update. If two or more update records contain the same key, then a master record will be updated with the minimum data value of these records. (In other circumstances, one could replace minimum with any other commutative, associative, binary operation.) The concurrent write may be accomplished by a method similar to that of the concurrent read, in which case the time to complete the operation is bounded by the time required for a fixed number of sort and interval operations.

7. *Compression:* It is often desirable to compress data into a region of the machine where optimal interprocessor communication is possible. The operation that places the data in such a region is called *compression*.

8. *Searching and Grouping:* Suppose every processor $P_i$ contains a *searching item* $s_i \in S$ and a *target item* $t_i \in T$, $1 \le i \le n$, where there is an ordering on the elements of $T$. Further, suppose there exists a Boolean relation $R(s, t)$,$s \in S$, $t \in T$. A solution to the *searching problem* requires each processor $P_i$ to find the largest $j$ such that $R(s_i, t_j)$ is true. For example, consider a collection of boxes of different shapes, where the $j^{th}$ box costs less than the $j$ - $1^{st}$, and a collection of objects to put into the boxes. Viewing the boxes as targets, the objects as searching elements, and the



relation *fits inside* as $R$, then the searching problem finds, for each object, the cheapest box that it fits inside of. Depending on what is known about $S$, $T$, and $R$, different algorithms may be the most efficient. Cases of interest include the following.

(a) Suppose there are decreasing (increasing) functions $f$ and $g$ mapping $s \in S$ and $t \in T$ into the same linearly ordered set, and a relation $U$ such that $U(f(s), g(t)) = R(s, t)$. Further, suppose that $U(x, y)$ = true implies $U(x', y)$ = true whenever $x' > x$ ($x' < x$) and $U(x, y')$ = true whenever $y' < y$ ($y' > y$). Then the searching problem can be solved by a *grouping technique* requiring only one pass, as follows. Mapping $s$'s and $t$'s via $f$ and $g$, respectively, sort all elements of $S$ together with all elements of $T$ into linear order and then perform a broadcast operation within every interval delimited by members of $T$ so as to inform members of $S$ as to their interval, which gives the required answer. A final sort based on the index of the searching and target items is used to return the items to their original processors.

As a simple example, the reader may construct the appropriate functions $f$ and $g$, the binary relation $R$, and the relation $U$ to solve the *interval location problem* for a set of $n$ numbers. That is, assume every processor has an element of a set $S$ of real numbers, and an element of set $T = \{t_1, t_2, \ldots, t_n\}$, where $-\infty = t_0 < t_1 < \ldots < t_n < t_{n+1} = +\infty$, and for each $s \in S$ it is required that the interval $t_i < s \le t_{i+1}$ be determined.

(b) Suppose functions $f$ and $g$ that map elements in $S$ and $T$ into the same linearly ordered set cannot be found. However, suppose that the elements of $S$ and $T$ have an ordering such that $R(s_i,t_j)$ = true implies $R(s_{i+1},t_j)$ = true and $R(s_i, t_{j-1})$ = true. In this situation, multiple parallel binary searches can be used to solve the search problem.

(c) Finally, suppose functions $f$ and g can be found, but the Boolean relation $R$ only has the property that $R(s_i, t_j)$ = true implies $R(s_i, t_{j-1})$ = true. For example, this would occur if the boxes mentioned above had the property that the $j^{th}$ box fits inside the $j$ - $1^{st}$ box. Another example follows.

Suppose that $S$ is a set of $n$ planar points represented by

their Cartesian coordinates $(x, y)$, such that $l \leq x \leq r$. Suppose $T$ is a set of $n$ line segments, where each line segment is represented by the Cartesian coordinates of its two endpoints, as $[(x_1, y_1), (x_2, y_2)]$. Further, assume that all line segments have $x_1 = l$ and $x_2 = r$. That is, the left endpoint of every line segment is $l$ and the right endpoint of every line segment is $r$. Finally, assume that no pair of line segments intersect, so that the line segments can be ordered by saying that segment $s_1$ is less than segment $s_2$ if $s_1$ lies below $s_1$. Note that the segments partition the vertical column $l \leq x \leq r$ into regions bounded above and below by consecutive line segments. Then the search problem in question is that of determining, for each point in $S$, the region that it belongs to. See Figure 1.8.

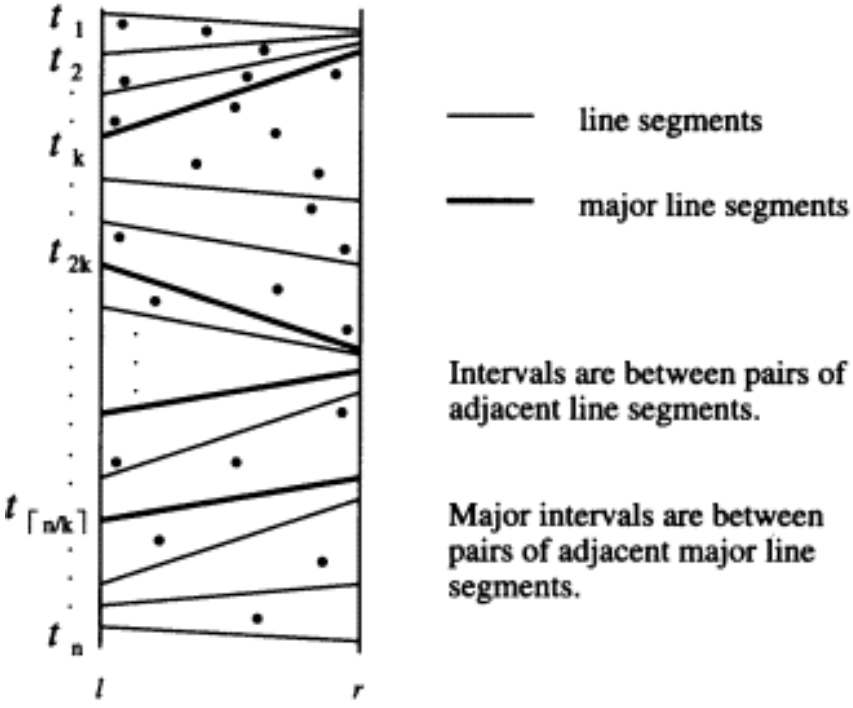



Figure 1.8:
Searching to find interval for points.

A solution to this search problem using a two-pass grouping technique follows. First, sort the line segments by $y$-coordinate with respect to the left endpoint $l$. Let this oper-



ation result in the ordered set $t_1, t_2, \ldots, t_n$, where $t_1 < t_2 < \ldots < t_n$. Next, all processors view $t_k$, $t_{2_k}, \ldots t_{k\lfloor n/k \rfloor}$, for some machine dependent constant $k$. While viewing these values, every processor decides which of the $[n/k]$ *major intervals* its search point is in. Finally, $S$ and $T$ are sorted so that all points of $S$ are grouped together (forming ordered intervals) with all line segments in their major interval. Next, all line segments in each major interval are circulated within that interval so that each point can detect which interval it is in. A final sort returns the points to their original processors. The reader should note that if the line segments of $T$ are restricted to being horizontal line segments, then the problem is equivalent to, and can be solved, using the method presented in (a).

9. *Parallel Prefix:* Given values $a_1, a_2, \ldots, a_n$, and a binary associative operator $\otimes$, the *product computation* problem is to compute the product $a_1 \otimes a_2 \otimes \ldots \otimes a_n$. The *initial prefix* problem is to compute all $n$ initial prefixes $a_1, a_1 \otimes a_2, \ldots, a_1 \otimes a_2 \otimes \ldots \otimes a_n$. The initial prefix problem when solved on a parallel model of computation is known as *parallel prefix*. This operation is quite powerful. For example, it can be used to sum elements, find the minimum or maximum of a set of elements, broadcast values, compress data items, and so forth.

10. *Reducing a Function:* Given sets $Q$, $R$, and $S$, let $g$ be a function mapping $Q \times R$ into $S$, and let * be a commutative, associative, binary operation over $S$. Define a map $f$ from $Q$ into $S$ by $f(q) = * \{g(q, r) \mid r \in R\}$, where $f$ is said to be the *reduction of* $g$. For example, if $Q$ and $R$ are sets of points in some metric space, if $S$ is the real numbers, if $g(q, r)$ is the distance from $q$ to $r$, and if * is the minimum, then $f(q)$ is the distance from $q$ to the nearest point in $R$.

## 1.6 Sample Algorithms

In this section, generic solutions are presented to two fundamental problems that are considered in later chapters of the book. This section serves the purpose of familiarizing the reader with

- the *component labeling problem*,



- the *convex hull problem*,
- problems involving image data,
- problems involving geometric data,
- designing generic, machine independent, parallel algorithms in terms of fundamental abstract data movement operations, as presented in Section 1.5, and
- general techniques, such as *divide-and-conquer*, *data reduction*, and *generating cross-products* that are frequently used to design efficient parallel algorithms for regular architectures.

### *1.6.1 Component Labeling for Digitized Picture Input*

The component labeling problem was defined in Section 1.4.1. For the algorithms presented in this section, it is assumed that an $n \times n$ digitized picture $A = \{a_{i,j}\}$is stored one pixel per processor on a machine with $n^2$ processors. The pixels are assumed to be in one of two states: black or white. The interpretation is that of a black picture on a white background. The picture is stored in a *natural fashion*, meaning that pixels that are adjacent in the picture are mapped to processors that are directly connected in the machine. For simplicity, a 4-connected definition of connectedness for the *figures* (i.e., connected black components) is assumed. That is, a black pixel $a_{i,j}$ is a neighbor of black pixels $a_{i+1,j}$, $a_{i-1,j}$, $a_{i,j+1}$, *and* $a_{i,j-1}$. (Notice that black pixels $a_{i,j}$ and $a_{i+1,j+1}$, for example, are not neighbors, though they may still be in the same figure if there is a 4-connected path of black pixels between them.)

Each processor that contains a black pixel uses its unique index as the label of the pixel that it contains. When a labeling algorithm terminates, every processor that contains a black pixel will store the label of its pixel and the label of the smallest labeled pixel that its pixel is connected to. That is, each such processor will know the label of the figure that its pixel is a member of.

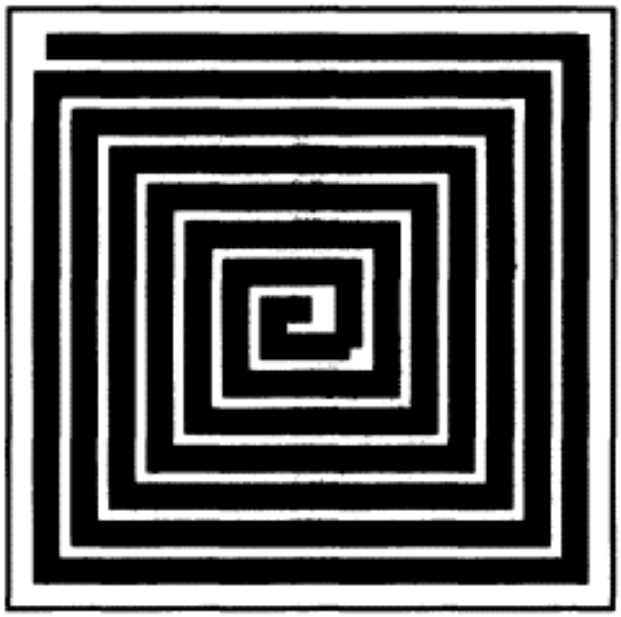
A simple parallel *propagation* algorithm can be used to label the figures, as follows. Every black processor (i.e., a processor containing a black pixel) initially assumes that the label of its pixel is the component label of the figure that its pixel is a member of. During each iteration of the algorithm, every black processor sends its current component label to its (at most) four black neighbors. Every black processor then compares



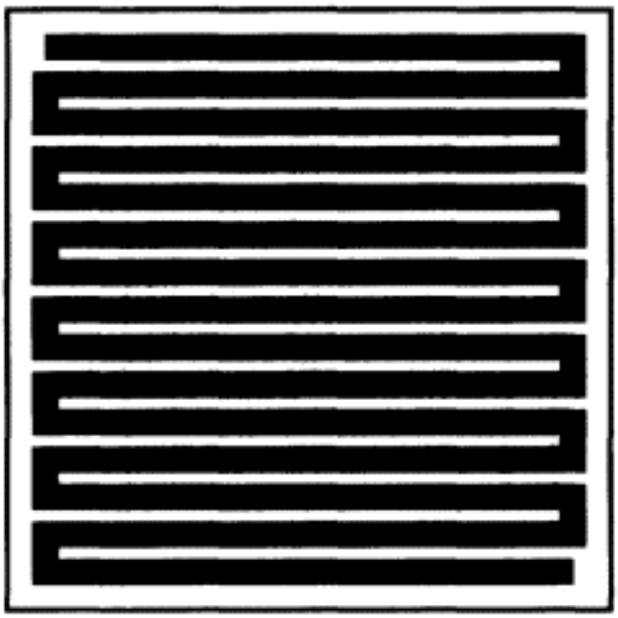
Figure 1.9:
A picture containing 'blob-like' figures.

its current label with the (at most) four labels just received, and keeps as its new label the minimum of these labels. It is easy to see that for each figure, the minimum label $L$ is propagated from processor $P_L$ (i.e., the processor with index $L$) to each black processor $P_i$ in its figure in the minimum number of steps required to pass a message from $P_L$ to $P_i$, under the restriction that data is only passed between neighboring black processors. Therefore, this labeling algorithm terminates in $\Theta(D)$ time, where $D$ is the maximum number of communication links any label must traverse. So, given 'blob-like' figures, as in Figure 1.9, all processors can know the label of their figure in $O(n)$ time. However, it is easy to construct non-'blob-like' figures, such as spirals or snakes, as shown in Figure 1.10, for which this propagation algorithm will require $\Theta(n^2)$ time.

In contrast to the $O(n^2)$ parallel propagation algorithm, two algorithms are given in this section that will label all figures regardless of the number, shape, or size of the figures, much more efficiently, in the worst case, when implemented on the machines considered in this book. Both algorithms follow a recursive divide-and-conquer solution strategy, which will be used throughout the book to produce efficient parallel so-



(a) A spiral is not a 'blob-like' figure.



(b) A snake is not a 'blob-like' figure.

Figure 1.10:
Pictures consisting of non-'blob-like' figures.

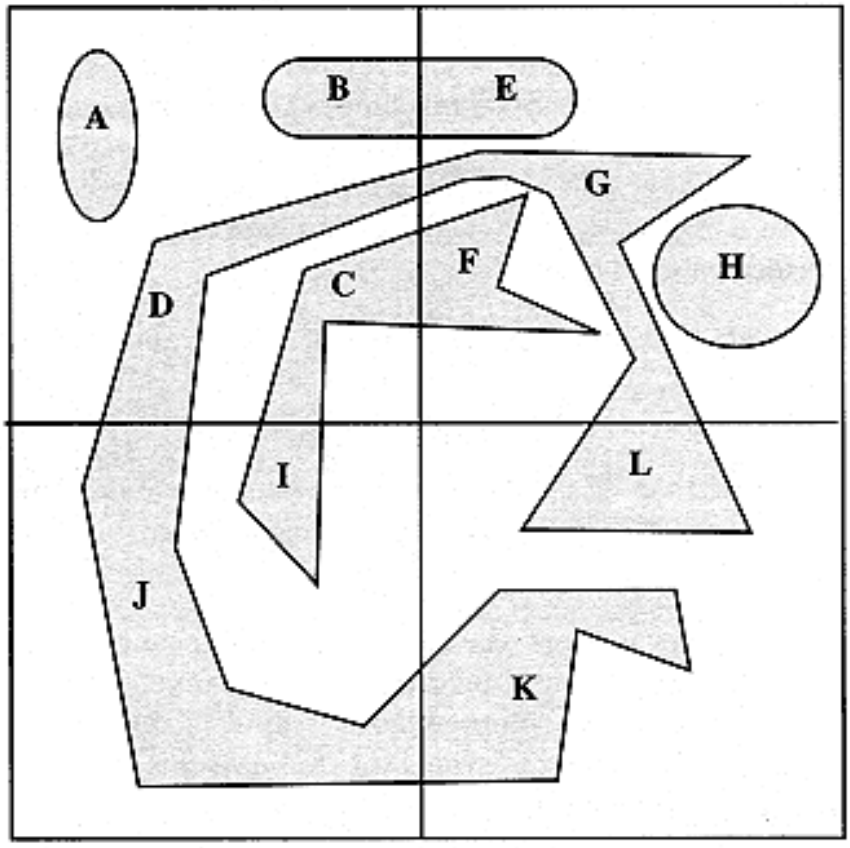



Figure 1.11:
Sample labeling after recursively labeling each quadrant.

lutions to a variety of problems. Both algorithms also serve as good examples of *data reduction algorithms*.

The first step of these algorithms is to recursively label the four quadrants of the picture independently, treating each quadrant as a complete picture. After this step, the only figures that could have pixels with differing labels are those figures that span a border between the quadrants. For instance, assuming Figure 1.11 represents the labels of figures after the independent and parallel recursive labeling of the quadrants, then figures *A* and *H* are labeled correctly, while the other figures contain pixels with an incorrect final label. Two methods for resolving the labeling conflicts to obtain correct global labels from the local (quadrant) labels are given. Both algorithms exploit the fact that the pertinent data has been reduced from an amount proportional to the area of the image to an amount proportional to the perimeter of the image.



**Compression Algorithm**

This method of resolving label conflicts introduces the concept of compressing data to a region of the machine where interprocessor communication is minimized. Specifically, the O($n$) pertinent pieces of data remaining in the machine after the recursive labeling is complete, will be moved to a region of the machine where subsequent computations can be performed efficiently.

1. Each black processor on the border of a quadrant, creates an *edge record* corresponding to an edge between its black pixel and any of the (at most) 4 neighboring black pixels.

2. Compress these O($n$) edge records to a subset of O($n$) processors of the machine that will allow for efficient interprocessor communication.

3. In this set of processors, the problem has now changed from a problem involving a digitized picture to that of solving the component labeling problem for unordered edge input. Using an unordered edge input algorithm over the data stored in this subset of processors, resolve the labels.

4. Use a concurrent read so that all processors in the machine obtain their (possibly new) labels from this final set of labels just computed.

Therefore, the running time for the entire component labeling algorithm is given by the recurrence

$$T(n^2) = T(n^2/_4) + Comp(n, n^2) + Edge(n) + CR(n^2),$$

where $T(n^2/4)$ is the time to perform the recursive labeling on the input $n \times n$ image, $Comp(n)$ is the time to compress O($n$) items on a machine of size $n^2$, $Edge(n)$ is the time to perform the unordered edge labeling algorithm for the O($n$) edges contained in the compressed subset of O($n$) processors, and $CR(n^2)$ is the time to perform a concurrent read on a machine of size $n^2$.

Once the compression operation has been performed, interprocessor communication is reduced so that the intermediate processing can be performed in an efficient manner. Further, reducing the communication diameter of the remaining information typically means that many processors remain idle during the intermediate processing. For instance, in



the compression version of the component labeling algorithm just given on a mesh of size $n^2$, O($n^2$) processors will remain idle during the unordered edge algorithm of Step 3 that resolves the border labels. A number of efficient algorithms will be given in later chapters of the book that exploit data reduction techniques. Some of these algorithms will exploit an iterative data reduction technique where at various stages of the algorithm, the remaining pertinent data is reduced, as are the number of active processors.

**Cross-Product Algorithm**

The method used in the Compression Algorithm for resolving labels relies on compressing O($n$) items on a machine of size $n^2$ to a place where interprocessor communication is minimized. Therefore, during the core of the algorithm, it is possible for O($n^2$) processors to remain idle. In contrast to the technique used in the Compression Algorithm, this method makes use of the available processors by creating an adjacency matrix to represent the cross-product of the O($n$) items, thereby utilizing the full complement of the O($n^2$) available processors.

1. Each black processor on the border of a quadrant, creates an *edge record* corresponding to an edge between its black pixel and any of the (at most) 4 neighboring black pixels.

2. Initialize all entries, except those along the diagonal, of the adjacency matrix to 0. The diagonal entries are initialized to 1. This corresponds to initially assuming that there are no edges in the graph, although it is assumed that a vertex is in the same figure as itself.

3. Using a mapping by border indices to rows and columns of an adjacency matrix, use a concurrent write for each border processor to place a 1 into an entry of the matrix corresponding to each of its edge records.

4. The problem has now been reduced to that of solving an adjacency matrix version of the component labeling algorithm. Using an adjacency matrix component labeling algorithm, resolve the labels.

5. Use a concurrent read so that all black processors obtain their (possibly new) label, where the diagonal elements of the matrix store the final label of the entry for its row and column.



The running time of the entire component labeling algorithm is given by

$$T(n^2) = T(n^2/4) + Create(n^2) + Adj(n^2) + CR(n^2),$$

where $T(n^2/4)$ is the time to perform the recursive labeling, $Create(n^2)$ is the time to create the adjacency matrix, including initialization and the concurrent write, $Adj(n^2)$ is the time to perform the adjacency matrix component labeling algorithm on the machine of size $n^2$, and $CR(n^2)$ is the time to perform a concurrent read on a machine of size $n^2$.

A number of efficient algorithms will be given in later chapters of the book that exploit the concept of creating cross-products. As a simple example, consider the problem of sorting $n$ distinct items on a machine of size $n^2$. A *counting sort* may be used, where a cross-product is first created in which the $n^2$ ordered pairs of items are stored one per processor. Each item $a_i$ need only count (sum) the number of pairs $(i, j)$ in which $a_j < a_i$ to determine its *rank* (i.e., final position of item $a_i$ in the sorted list). The items are then routed to their final destinations.

### *1.6.2 Convex Hull for Planar Point Data Input*

In Section 1.4.2, the problem of identifying the extreme points representing the convex hull of a set of planar points was discussed. In this section, generic machine independent parallel solutions to the convex hull problem are given. These algorithms reinforce advantages of paradigms such as divide-and-conquer and data reduction, and introduce new paradigms. They are also given in terms of fundamental data movement operations, some of which were not used in the component labeling algorithms of the previous section.

In this section, two distinct strategies are given for marking the extreme points representing the convex hull of a set $S$ of $n$ planar points distributed arbitrarily one point per processor on a machine with $n$ processors. Both algorithms follow a general divide-and-conquer solution strategy. The first algorithm divides the points into a fixed number of subsets, while the second algorithm divides the points into multiple subsets.

**Fixed Subset Division Algorithm**

1. *Preprocessing:* Sort the $n$ planar points so as to order them by $x$-coordinate. That is, after sorting the points, they will be ordered

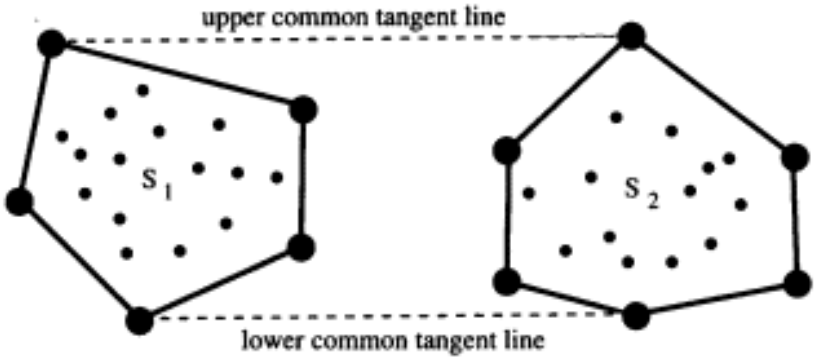



Figure 1.12:
Upper and lower tangent lines between linearly separable sets $S_1$ and $S_2$.

so that the $x$-coordinate of the point in processor $P_i$ is less than the $x$-coordinate of the point in processor $P_j$, for $i < j$.

2. If $n \leq 2$, then all points are extreme points. Otherwise, let $S_1$ denote the points in processors $P_1, P_2, \ldots,$ $P_{n/2}$, and $S_2$ denote the points in processors $P_{n/2+1}, P_{n/2+2}, \ldots, P_n$ . Notice that $S = S_1 \cup S_2$, that $S_1$ and $S_2$ each contain $n/2$ points, and that the points in $S_1$ have $x$-coordinates less than those in $S_2$. (Dividing $S$ into 2 subsets makes the presentation of this sample algorithm easier. The particular constant number of subsets can be appropriately modified for implementation on a given machine. For example, on mesh-based machines, $S$ would typically be divided into 4 subsets, one corresponding to each quadrant of the (base) mesh.)

3. Recursively identify the extreme points of $S \ldots 1$ and the extreme points of $S_2$. (Note: this is a recursive call to Step 2, not Step 1.)

4. Identify the upper and lower common tangent lines between hull($S_1$) and hull($S_2$) by performing a grouping operation. See Figure 1.12. Two different grouping operations may be used to determine these tangent lines.

(a) The first grouping operation is based on the fact that an extreme point $p_k$ of $S_1$, with $p_{k-1}$ and $p_{k+1}$ as its preceding and



succeeding extreme points, respectively, with respect to the counterclockwise ordering of the extreme points of $S_1$, is the left endpoint of the upper common tangent line between $S_1$ and $S_2$ if and only if no points of $S_2$ lie above the line $\overline{p_{k+1}p_k}$, while at least one point of $S_2$ lies above the line $\overline{p_kp_{k-1}}$. (Recall that the extreme points are labeled in counterclockwise fashion.) Similar remarks can be made about the other three endpoints.

Searching for an endpoint is accomplished by a grouping operation that uses multiple binary searches. For instance, the extreme point that corresponds to the left endpoint of the upper tangent line between $S_1$ and $S_2$ may be determined as follows. (The other endpoints are determined similarly.)

i. Suppose there are $n_1$ extreme points of $S_1$ which are labeled $1, 2, \ldots, n_1$. Let $l = 1$ and $r = n_1$.

ii. Let $k$ = Broadcast extreme points $p_k$, $p_{k+1}$, and $p_{k-1}$ of $S_1$ to the processors that are maintaining extreme points for $S_2$.

iii. Using a concurrent write, all processors maintaining extreme points of $S_2$ that are above the line $\overline{p_{k+1}p_k}$, send a message to the processor maintaining $p_k$ of $S_1$, which in turn broadcasts the response to all processors of $S_1$.

iv. If there is at least one such point of $S_2$ that is above $\overline{p_{k+1}p_k}$, then the binary search continues on the points labeled $k + 1, k + 2, \ldots, r$ (i.e., set $l = k + 1$ and return to Step 4(a)ii).

v. If there are no points of $S_2$ above $\overline{p_{k+1}p_k}$, then the processor of $S_1$ that maintains $p_k$ broadcasts this fact to the processors of $S_2$. This is followed by performing a concurrent write, where all processors maintaining extreme points of $S_2$ above the line $\overline{p_kp_{k-1}}$, send a message to the processor maintaining $p_k$, which in turn broadcasts the response to all processors of $S_1$.

A. If there are no points of $S_2$ above $\overline{p_kp_{k-1}}$, then the binary search continues on the set of points labeled $1, 2, \ldots, k - 1$ (i.e., set $r = k - 1$ and return to Step 4(a)ii).

B. If there is at least one point of $S_2$ above $\overline{p_kp_{k-1}}$, then $p_k$ is the left endpoint of the upper tangent line between $S_1$ and $S_2$. Notice that $p_k$ has the property



that all points of $S_2$ lie below the line $\overline{p_{k+1}p_k}$, while at least some points of $S_2$ lie above the line $\overline{p_kp_{k-1}}$

After no more than $\lceil \log_2 n_1 \rceil$ iterations, an extreme point of $S_1$ will be found that is the left endpoint of the upper tangent line between $S_1$ and $S_2$. For some of the machines considered in this book, it will be advantageous to interleave steps of the binary search to find the left common tangent point with steps of the binary search to find the right common tangent point, compressing all of the remaining candidates from $S_1$ and $S_2$ jointly after each pair of searches.

(b) A different grouping operation may be used in order to determine the endpoints of the upper and lower common tangent lines. This operation is a one pass operation based on the *angles of incidence* (*AOI*), as defined on page 20, of the hull edges. Specifically, suppose $\overline{p_iq_j}$, $p_i \in S_1$, $q_j \in S_2$, is the upper tangent line between convex sets $S_1$ and $S_2$, as in Figure 1.12. Then it can be shown [PrHo77] that

i. $\text{AOI}(\overline{p_{i-1}p_i}) \leq \text{AOI}(\overline{q_jp_i}) < \text{AOI}(\overline{p_ip_{i+1}})$, and
ii. $\text{AOI}(\overline{q_{j-1}q_j}) < \text{AOI}(\overline{q_jp_i}) \leq \text{AOI}(\overline{q_jq_{j+1}})$.

Therefore, each extreme point $P_i$ simply needs to locate the edges of the other set with angles of incidence just above and just below the angles of incidence of $\overline{p_{i-1}p_i}$ and $\overline{p_ip_{i+1}}$. This is accomplished by sorting records representing both endpoints of the angles of support of every extreme point of $S_1$ ($S_2$) together with records representing the angles of incidence of the hull edges of $S_2$ ($S_1$) and then performing broadcasts within the intervals delimited by the hull edges.

5. Eliminate all extreme points between the common tangent lines (i.e., all extreme points of $S_1$ and $S_2$ that are inside the quadrilateral formed by the four endpoints representing the common tangent lines) and renumber the remaining extreme points. This is accomplished by broadcasting the information pertaining to the four endpoints to all processors maintaining a point of $S$, and then having each processor make a constant time decision as to whether or not it remains an extreme point, and if so, what its new number is.

The running time of the algorithm is given by

$$T(n) = T'(n) + Sort(n),$$



where $Sort(n)$ is the time to sort $n$ items distributed one per processor on a machine of size $n$, and $T'(n)$ is the time to perform all but the first (preprocessing) step. $T'(n)$ satisfies the recurrence

$$T'(n) = T'(n/2) + Group(n) + Broad(n) + Elim(1),$$

where $T'(n/2)$ is the time for the recursive call, $Group(n)$ is the time to perform an appropriate grouping operation to determine the upper and lower common tangent lines, $Broad(n)$ is the time to perform a broadcast operation on a machine of size $n$, and $Elim(1)$ is the time required for each processor to make the final extreme point decision.

**Multiple Subset Division Algorithm**

1. *Preprocessing:* Using sorting, partition the $n$ planar points of $S$ into $n^{1/k}$ subsets, where $k$ is a machine dependent constant that minimizes the running time of the algorithm. The partitioning is done so as to produce the following.

(a) $S = \bigcup_{i=1}^{n^{1/k}} S_i$, where each of the $n^{1/k}$ sets $S_1, S_2, \ldots, S_{n^{1/k}}$, is of size $n^{(k-1)/k}$

(b) The $x$-coordinates of all points in $S_i$ are less than the $x$-coordinates of all points in $S_j$, for $i < j$.

(c) Define *region* $R_i$ to be a subset of processors of the machine responsible for set $S_i$. It is assumed that the set $S_i$ is stored in the $i^{th}$ subset of processors of size $n^{(k-1)/k}$ and that this ordering holds recursively within each such region of size $n^{(k-1)/k}$

2. If $n \leq 2$, then all points are extreme points. Otherwise, for each region $R_i, 1 \leq i \leq n^{/k}$, recursively identify the extreme points of $S_i$. (Note: this is a recursive call to Step 2, not Step 1.)

3. Using a two pass grouping operation, each region $R_i$ determines the endpoints of the upper and lower tangent lines between $S_i$ and every other set of points $S_j$, for $i \neq j$, as follows.

(a) Each region $R_i, 1 \leq i \leq n^{1/k}$, sends $n^{1/k}$ query points to every other region $R_j$, $i \neq j$. The query points are equally spaced with respect to the counterclockwise numbering of the extreme points of $S_i$. Further, each query point is represented by two records, one corresponding to each endpoint of the range of its angles of support.

(b) Each region $R_i$,$1 \le i \le n^{1/k}$, receives $n^{1/k}$ query points from every other region $R_j$, $i \ne j$. These records are merged so that they are received completely ordered with respect to the key field (an angle).

(c) Each region $R_i$,$1 \le i \le n^{1/k}$, merges the $O(n^{2/k})$ ordered query point records it just received with its $O(n^{(k-1)/k})$ ordered angle of incidence records that correspond to its hull edges.

(d) Within each region $R_i$,$1 \le i \le n^{1/k}$, perform a broadcast within ordered intervals, as determined by consecutive pairs of angle of incidence records. Each set of ordered query points that arrived from region $R_j$, $i \ne j$, now contains information that can be used to decide which consecutive pair of its query points represents the interval of $R_j$'s extreme points that needs to be further explored in search of the endpoints of the tangent lines between $R_j$ and $R_i$. (Properties of angles of incidence, as introduced on page 39 in Step 4b of the Fixed Subset Division Algorithm, are used.)

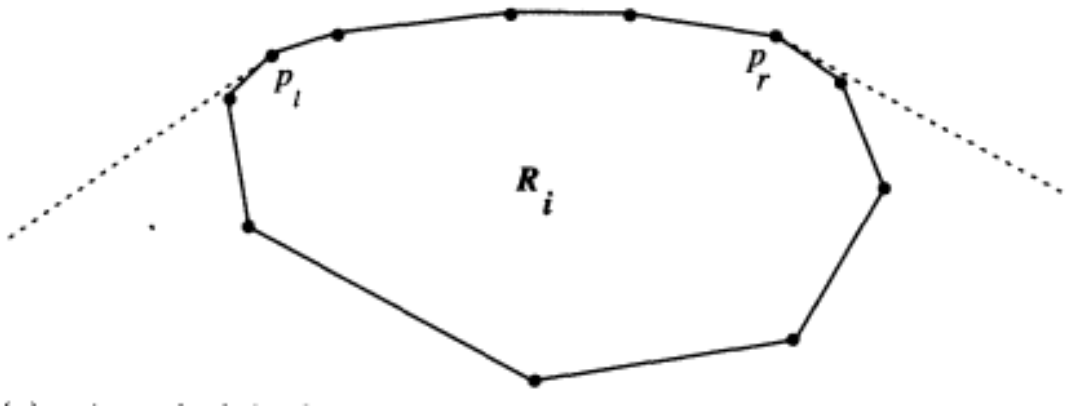
(e) The query points are returned to their original regions ordered by angles of support, consecutive items are compared to determine the appropriate interval, and the second pass of the grouping operation is performed much as the first, but with the $n^{1/k}$ query points sent to each region determined by the appropriate consecutive pair of points determined at the end of the first pass.

4. Within each region $R_i$,$1 \le i \le n^{1/k}$, determine the minimum slope of a tangent line between $R_i$ and $R_j$, for $j < i$ (i.e., those regions to the left of $R_i$). Let $p_i$ be the extreme point of $R_i$ that is an endpoint of the common tangent line. Determine the maximum slope of a tangent line between $R_i$ and $R_j$, for $j > i$ (i.e., those regions to the right of $R_i$). Let $p_r$ be the extreme point of $R_i$ that is an endpoint of the common tangent line. If $p_r$ is to the left of $p_i$, or $p_r = p_i$ and the angle open to the top, formed by these two line segments, is less than 180°, then no points of $R_i$ are extreme points of $S$. Otherwise, those extreme points of $R_i$ between $p_i$ and $p_r$ are extreme points of $S$. See Figure 1.13.

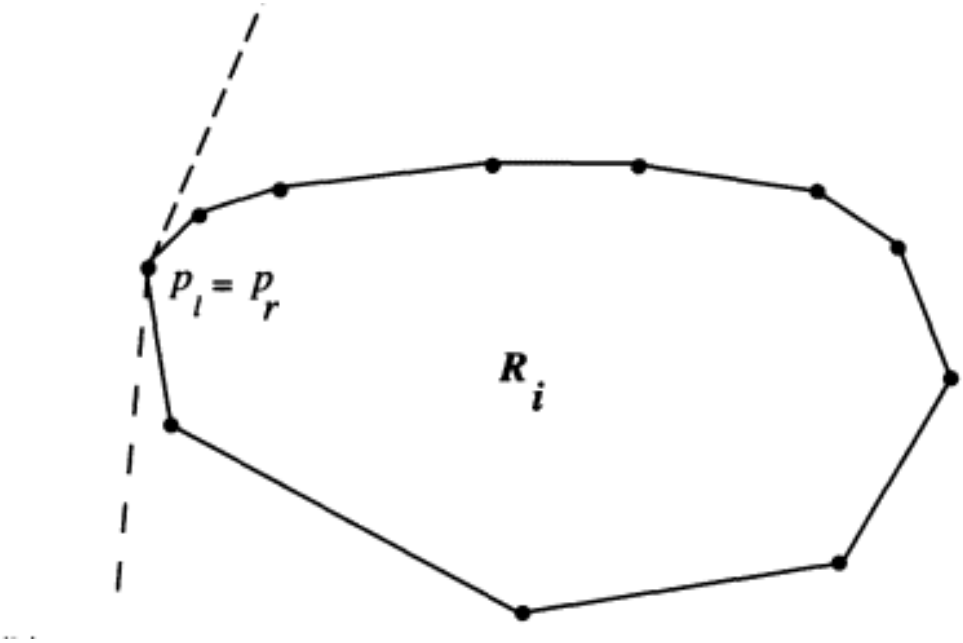
The running time of the algorithm is given by

$$T(n) = T'(n) + Sort(n),$$



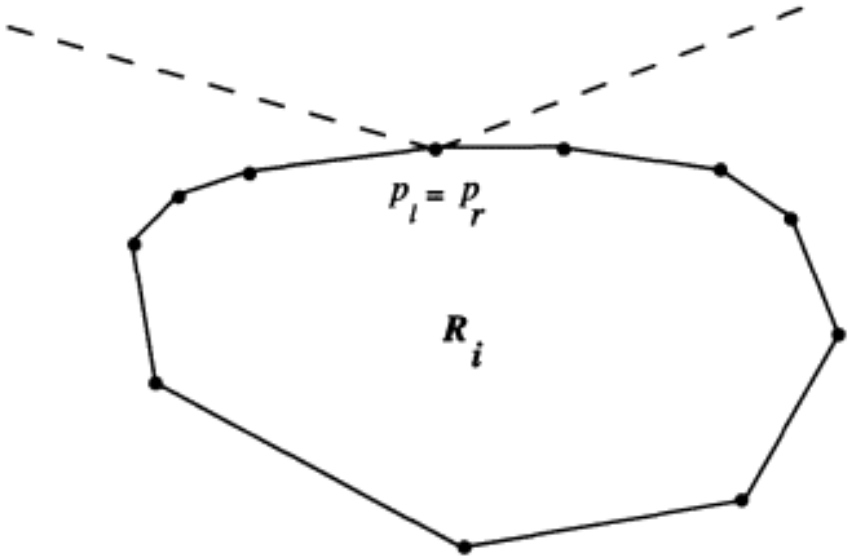



(a) $p_l$ is to the left of $P_r$ and the angle open to the top is > 180°. Those extreme points of $R_i$ that are between $p_l$ and $P_r$ are extreme points of $S$.

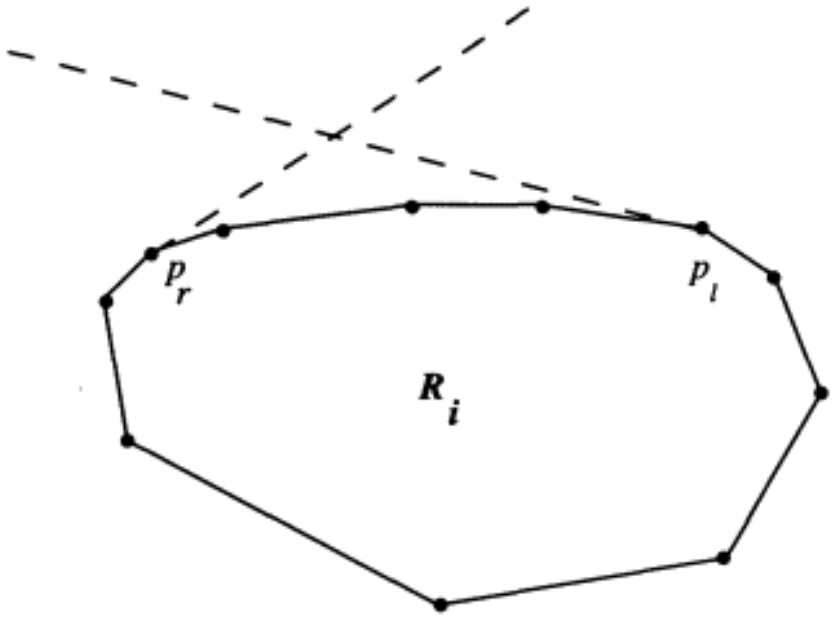



(b) $p_l$ is equal to $p_r$ and the angle open to the top is > 180°. $p_l$ (= $P_r$ ) is an extreme point of $S$.






(c) $p_l$ is equal to $P_r$ and the angle open to the top is ≤ 180°. No extreme points of $R_i$ are extreme points of $S$.




(d) $p_r$ is to the left of $p_l$. No extreme points of $R_i$ are extreme points of $S$.

Figure 1.13:
Using $p_l$ and $p_r$ to determine extreme points.



where *Sort*($n$) *is* the time to sort $n$ items distributed one per processor on a machine of size $n$ in the partition step, and $T'(n)$ is the time to perform all but this first step. $T'(n)$ satisfies the recurrence

$$T'(n) = T'(n^{1/k}) + Group(n) + Semi(n) + Broad(n) + Elim(1),$$

where $T(n^{1/k})$ is the time for the recursive call, *Group*($n$) is the time to perform an appropriate grouping operation to determine the endpoints of the upper and lower common tangent lines, $Semi(n^{(k-1)/k})$ is the time to perform a semigroup (associative binary) operation to determine minimum and maximum slopes in a region of size $n^{(k-1)/k}$, $Broad(n^{(k-1)/k})$ is the time to perform a broadcast operation within each such region, and *Elim*(1) is the time required for each processor to make the final extreme point decision.

## 1.7 Further Remarks

In this chapter, a variety of models of computation have been defined. The problems that will be solved in the later chapters of the book have also been introduced, along with the types of inputs that the problems may have. The concept of describing machine independent parallel algorithms for regular architectures in terms of abstract data movement operations was introduced, and a variety of these fundamental data movement operations were discussed. Finally, sample parallel algorithms were given in terms of abstract data movement operations that also introduced fundamental paradigms for solving problems on regular architectures. The reader may now proceed comfortably to Chapter 2 or Chapter 5.

# 2
# Fundamental Mesh Algorithms

## 2.1 Introduction

In this chapter, standard data movement operations and fundamental algorithms are presented for the mesh computer. All of these algorithms have optimal $\Theta(n)$ worst-case running times on a mesh of size $n^2$. In Section 2.2, basic mesh definitions are reviewed. Section 2.3 concentrates on showing that for problems requiring global communication, a mesh of size $n^2$ must take $\Omega(n)$ time to solve the problem. Fundamental mesh algorithms, such as passing information in rows or columns, computing a semigroup (i.e., associative binary) operation, and determining a parallel prefix, are given in Section 2.4. Section 2.5 presents optimal mesh algorithms to solve problems involving matrices. These problems include transposition, multiplication, transitive closure, and inverse. Algorithms involving ordered data are presented in Section 2.6, including algorithms for sorting, performing basic operations within (ordered) intervals, concurrent reads and writes, and compressing data.

## 2.2 Definitions

The *mesh computer* (*mesh*) *of size* $n^2$ is a machine with $n^2$ simple processors arranged in a square lattice. To simplify exposition, it is assumed that $n = 2^c$, for some integer $c$. For all $i, j \in [0,\ldots,n-1]$, processor $P_{i,j}$, representing the processor in row $i$ and column $j$, is connected via bidirectional unit-time communication links to its four *neighbors*, processors $P_{i\pm1,j}$ and $P_{i,j\pm1}$, assuming they exist. (See Figure 2.1.) Each processor has a fixed number of registers (words), each of size $\Omega(\log n)$, and can perform standard arithmetic and Boolean operations on the contents of these registers in unit time. Each processor can also send or receive a word of data from each of its neighbors in unit time. Each processor contains its row and column indices, as well as a unique identification register, the contents of which is initialized to the processor's row-major index, shuffled row-major index, snake-like index, or proximity order index, as shown in Figure 2.2. (If necessary, these values can be generated in $\Theta(n^{1/2})$ time.)

For some of the problems in this and subsequent mesh chapters, it



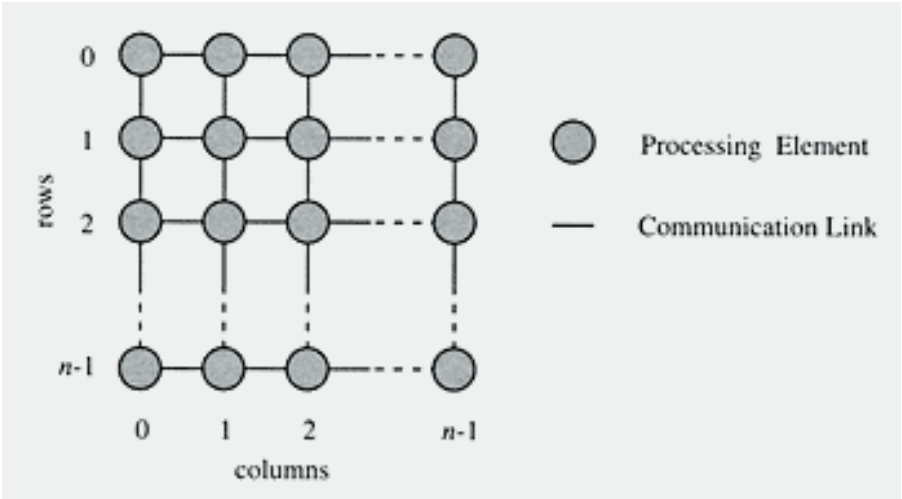



Figure 2.1:
A mesh computer of size $n^2$.

will be convenient to conceptually partition the mesh into *submeshes* or *squares* of size $S$. What is meant by this is that the mesh will be completely partitioned into disjoint submeshes (or squares) of size $S$, where $S$ is a power of 4. Using this partitioning, the concept of *the square of size S containing processor P* is well-defined.

### 2.3 Lower Bounds

The communication diameter of a mesh of size $n^2$ is $\Theta(n)$ since any arbitrary pair of processors can communicate in $O(n)$ time, and some processors require $\Omega(n)$ time. For instance, information starting at diagonally opposed corners of the mesh cannot meet in less than $n - 1$ steps, data from one of these diagonally opposed processors cannot reach the other processor in less than $2n - 2$ steps, information starting in processors at opposite edges of the mesh cannot meet in less than $n/2$ steps, and data from a processor on one edge of the mesh cannot reach a processor on the opposite edge of the mesh in less than $n - 1$ steps. A problem is said to require *global communication* if at least one processor must receive information that might originate in any processor. A lower bound on the worst-case running time of an algorithm to solve a problem that requires global communication is $\Omega(n)$. In fact, for many problems that involve global communication of data, it is easy to devise inputs for which any algorithm to solve the problem on a mesh of size $n^2$ must take $\Omega(n)$ time. Similarly, a lower bound on the running time of



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | | 0 | 1 | 4 | 5 |
| 4 | 5 | 6 | 7 | | 2 | 3 | 6 | 7 |
| 8 | 9 | 10 | 11 | | 8 | 9 | 12 | 13 |
| 12 | 13 | 14 | 15 | | 10 | 11 | 14 | 15 |
| (a) Row-major | | | | | (b)Shuffled row-major | | | |
| 0 | 1 | 2 | 3 | | 0 | 1 | 14 | 15 |
| 7 | 6 | 5 | 4 | | 3 | 2 | 13 | 12 |
| 8 | 9 | 10 | 11 | | 4 | 7 | 8 | 11 |
| 15 | 14 | 13 | 12 | | 5 | 6 | 9 | 10 |
| (c) Snake-like | | | | | (d) Proximity | | | |

Figure 2.2:
Indexing schemes for the processors of a mesh.



an algorithm in which 'distant' processors (i.e., processors that require $\Omega(n)$ steps to communicate with each other) must exchange or combine information is $\Omega(n)$.

The wire-counting (bisection width) approach can also be used to show that many problems require $\Omega(n)$ time on a mesh of size $n^2$. For example, sorting or merging require $\Omega(n)$ time since it is possible that all of the data initially residing in processors in columns $0 \ldots {}^{n}/_{2} - 1$ must be moved to processors in columns ${}^{n}/_{2} \ldots n - 1$, and vice versa. Since there are $n$ wires connecting these two sets of processors, then in order to move $n^2$ data items between these two sets requires $\Omega(n)$ time. Similar arguments apply to operations such as matrix transposition and component labeling of graphs.

## 2.4 Primitive Mesh Algorithms

Fundamental mesh algorithms, such as those presented in this section, will form the foundation of advanced mesh algorithms that appear later in this chapter and throughout the book.

### *2.4.1 Row and Column Rotations*

A frequent situation is that every processor needs to transmit a fixed amount of data to be viewed by all other processors in the same row (column). On a mesh of size $n^2$ this can be done in $n - 1$ steps, simultaneously for all processors, as follows. Initially, all processors in a row (column) send copies of their data in both directions. As a processor receives data from neighboring processors, it views the data and then (in the next step) continues to send it in the direction that it was traveling. Data that reaches edge processors are viewed and then discarded.

Occasionally, more control is needed over the timing in terms of coordinating when each processor receives such information. This occurs when row information is being passed around that must be combined with column information that is also being passed around, and matching pairs of data must arrive at the appropriate processor at the same time. (Recall that each processor only has a bounded amount of memory, and hence cannot store all data that passes through it.) One useful variant, called *row* (*column*) *rotation*, is as follows. Copies of the data from each processor move towards the easternmost (northernmost) processor of their row (column) in lock-step fashion. Once the data reaches the extreme processor, the copies of information reverse themselves until they



reach the westernmost (southernmost) processor of their row (column) where they reverse themselves again. Notice that at any step, a processor has copies of (and views) at most two sets of data. The algorithm terminates the step before processors would simultaneously receive copies of their original data from their western (southern) neighbor. This rotation takes exactly $2n - 3$ steps.

### *2.4.2 Passing a Row (Column) Through the Mesh*

Suppose that every processor of a mesh of size $n^2$ needs to view a fixed amount of data from every processor of a given row (column). This can be done in $\Theta(n)$ time as follows. Rotate all columns (rows) simultaneously so that a copy of (the required data from) the given row (column) exists in every row (column) of the mesh. Now, simply rotate all rows (columns) of the mesh simultaneously.

### *2.4.3 Semigroup Operations, Reporting, and Broadcasting*

Suppose that every processor of a mesh of size $n^2$ needs to know the result of applying some semigroup operation (i.e., an associative binary operation such as minimum, summation, or parity) to $n^2$ pieces of data distributed one item per processor. The result of applying the function to the data can be computed and distributed to all processors in $\Theta(n)$ time by reporting the result to processor $P_{0,0}$ and then broadcasting this value to all processors. To *report* this value to processor $P_{0,0}$, first perform a row rotation for all rows simultaneously, so that in $\Theta(n)$ time every processor in the first column knows the result of applying the semigroup operation over all values in its row. Using a column rotation in the first column of the mesh, processor $P_{0,0}$ can know the result of applying the semigroup operation over the $n$ row results, which gives processor $P_{0,0}$ the result of applying the semigroup operation over all $n^2$ pieces of data. To *broadcast* this value to all processors, simply reverse the process.

Alternately, the result of a semigroup operation could be known to all processors in $\Theta(n)$ time without performing a report and broadcast. Simply perform a row rotation simultaneously in all rows so that every processor knows the result of applying the semigroup operation over the values stored in its row. Then perform a column rotation simultaneously for all columns so that every processor knows the result of applying the semigroup operation over the previously computed row values, which is



the result of applying the semigroup operation over all $n^2$ values. Notice that asymptotically, both methods finish in optimal $\Theta(n)$ time.

### *2.4.4 Parallel Prefix*

Using the row-major indexing scheme of a mesh of size $n^2$, suppose processor $P_i$, $0 \le i \le n^2 - 1$, initially contains $a_i$, and that every processor $P_i$ is to determine the $i^{th}$ initial prefix $a_0 \otimes a_1 \otimes \cdots \otimes a_i$, where $\otimes$ is an associative binary operator. Notice that according to the row-major indexing scheme, processor $P_i$, $0 \le i \le n^2 - 1$, resides in row $j = \lfloor i/n \rfloor$. The solution to this problem can be obtained in $\Theta(n)$ time by a series of row and column rotations, as follows. (See Figure 2.3 for an example.)

1. Perform a row rotation so that every processor $P_i$ in row $j$ knows $a_{j*n} \otimes a_{j*n+1} \otimes \cdots a_i$.

2. Perform a row rotation so that processor $P_{j*n}, 0 \le j \le n - 1$, knows the value stored in processor $P_{[(j+1)*n]-1}$, namely, $a_{j*n} \otimes a_{j*n+1} \otimes \cdots a_{[(j+1)*n]-1}$.

3. Perform a column rotation in column 0 so that processor $P_{j*n}, 1 \le j \le n - 1$, knows $V_j = a_0 \otimes a_1 \otimes \cdots \otimes a_{j*n-1}$.

4. Perform a row rotation in every row $j$, $0 \le j \le n - 1$, to broadcast $V_j$ to all processors in row $j$ so they may update their value to obtain $a_0 \otimes a_1 \otimes \cdots \otimes a_i$, for every processor $P_i$, $0 \le i \le n^2 - 1$.

It should be noted that if the data is originally ordered according to some indexing scheme other than row-major, then Section 2.6.1 will show how to use sorting to put the data into row-major order, so that parallel prefix can be computed as described, and then how to use sorting to return the values to the required processors, all without affecting the asymptotic running time of the algorithm.

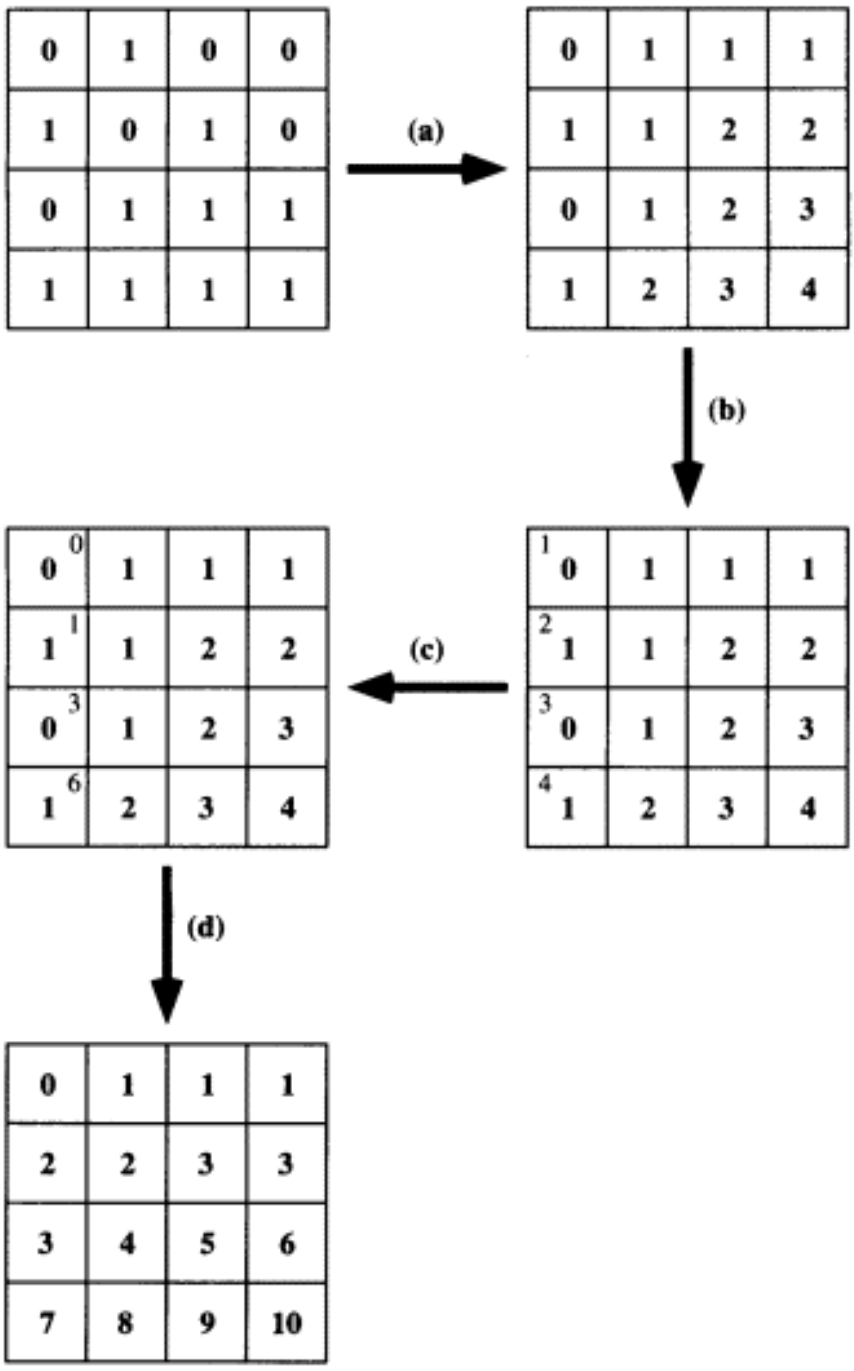
## 2.5 Matrix Algorithms

In this section, algorithms are presented that involve $n \times n$ matrices mapped onto meshes of size $\Theta(n^2)$.

### *2.5.1 Matrix Transposition*

In this section, an optimal $\Theta(n)$ time algorithm is presented to compute the transpose of an $n \times n$ matrix $A = \{A_{i,j}\}$, initially stored in a mesh





| 7 | 8 | 9 | 10 |
|---|---|---|---|

Figure 2.3:
An example of computing the parallel prefix on a mesh of size $n^2$.
The operation $\otimes$ is taken to be addition (+) in this example.



of size $n^2$ so that processor $P_{i,j}$ contains element $A_{i,j}$. The transpose of a matrix $A$ is given by $A^T_{i,j} = A_{j,i}$. The algorithm consists of two complimentary phases that are each completed in $\Theta(n)$ time, as follows. Denote diagonal processors $P_{i,i}$, $1 \leq i \leq n$, as *routers*. For all above-diagonal processors $P_{i,j}$, $i < j$, send the value of $A_{i,j}$ down to diagonal processor $P_{j,j}$ in lock-step fashion. Each value $A_{i,j}$, $i < j$, reaches diagonal processor $P_{j,j}$ in $k = j - i$ steps. As each router $P_{j,j}$ receives an $A_{i,j}$, it sends the data to the left where it will move for $k = j - i$ steps, until it reaches below-diagonal processor $P_{j,i}$. Next, in a similar fashion all below-diagonal processors $P_{i,j}$, $i > j$, send their data to the right, where diagonal processor $P_{i,i}$ routes the data upwards. Therefore, in $\Theta(n)$ time every processor $P_{i,j}$ contains $A_{j,i}$.

**Theorem 2.1** *Given an $n \times n$ matrix $A$ stored in a natural fashion on a mesh of size $n^2$, the transpose of $A$ can be computed in $\Theta(n)$ time.*

### *2.5.2 Matrix Multiplication*

Given two $n \times n$ matrices, $A$ and $B$, the matrix product $C = AB$ is given by $C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$. The first algorithm of this section shows how to compute $C = AB$ in $\Theta(n)$ time on a mesh of size $4n^2$. This algorithm is then modified to compute $C = AB$ in $\Theta(n)$ time on a mesh of size $n^2$.

Assume that matrix $A$ is stored in the lower-left quadrant, matrix $B$ is stored in the upper-right quadrant, and that the resultant matrix $C$ is to be constructed in the lower-right quadrant of a mesh of size $4n^2$, as shown in Figure 2.4. At time 1, in lock-step fashion all processors containing an element of the first row of $A$ send their values to the right and all processors containing an element of the first column of $B$ send their values down. The processor responsible for $C_{1,1}$ can now begin to compute its running sum. At time 2, row 1 of $A$ and column 1 of $B$ continue to move in the same direction, and row 2 of $A$ and column 2 of $B$ start to move right and down, respectively. In general, at time $i$, the $i^{th}$ row of $A$ and the $i^{th}$ column of $B$ start to move right and down, respectively. Each processor that simultaneously receives a piece of data from a processor to its left and from a processor above computes the product of these two values and adds it to its running sum. At time $i + 1$, every processor sends the values received during time $i$ to neighboring processors in the direction that they were moving. So, at time $k$, rows $1 \ldots k$ of $A$ and columns $1 \ldots k$ of $B$ move right and down, respectively, where this is the first such movement for row $k$ of $A$ and column $k$ of $B$. Therefore, row $n$ of $A$ and column $n$ of $B$ start moving

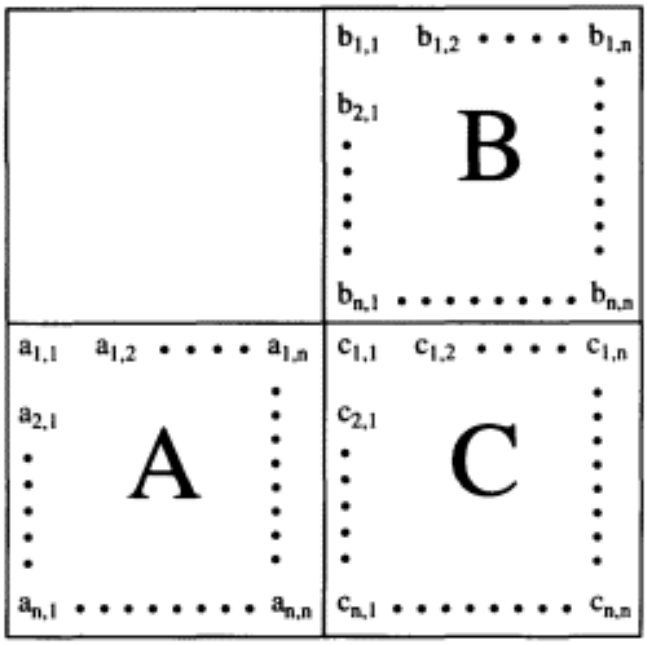



Figure 2.4:
Multiplying matrices on a mesh of size $4n^2$.

at time $n$, $C_{n,n}$ is the last value determined, and the running sum for $C_{n,n}$ is completed at time $3n - 2$. Hence, the algorithm runs in $\Theta(n)$ time on a mesh of size $4n^2$.

The algorithm can be modified to run on a mesh of size $n^2$ in $\Theta(n)$ time, as follows. Assume that processor $P_{i,j}$ initially contains $A_{i,j}$ and $B_{i,j}$, and that processor $P_{i,j}$ must contain $C_{i,j}$ at the conclusion of the algorithm. For all columns, simultaneously perform a column rotation so that processor $P_{i,j}$ contains $B_{(n-i+1)j}$. For all rows, simultaneously perform a row rotation so that processor $P_{i,j}$ contains $A_{i,(n-j+1)}$. Now, follow the previous algorithm (adjusting for the fact that the elements are initially situated so that $C_{1,1}$ is ready to begin computing its running sum) while substituting a single step of a row and column rotation for each lock-step movement of a row or column. The elements of $A$ are involved in the row rotation (viewed as rotating to the left, though, as before, the values are used as they go to the right) and the elements of $B$ are involved in the column rotation (viewed as rotating up, though, as before, the elements are used as they go down). Elements of $A$ and $B$ meet at the appropriate processors with the same timing as in the previous algorithm. The initial row and column rotations take $\Theta(n)$ time,



and the previous algorithm requires $3n - 2$ steps, so the entire algorithm is complete in $\Theta(n)$ time. It should be noted that a similar $\Theta(n)$ time mesh algorithm for matrix multiplication is possible that avoids the preprocessing row and column rotations. Such an algorithm is left to the reader.

**Theorem 2.2** *Given two $n \times n$ matrices $A$ and $B$ stored in a mesh of size $n^2$ so that processor $P_{i,j}$ initially contains $A_{i,j}$ and $B_{i,j}$, the product $C = AB$ can be computed in $\Theta(n)$ time so that processor $P_{i,j}$ stores $C_{i,j}$.*

### *2.5.3 Transitive Closure*

Let $G = (V, E)$ be a *directed graph*, where $V = \{1, \ldots, n\}$ is the set of vertices and $E$ is the set of edges. Assume $G$ is represented by its adjacency matrix $A$, where $A(i, j)$ is 1 if there is an edge from vertex $i$ to vertex $j$, and is 0 otherwise. The *transitive closure of A*, denoted $A^*$, is the $n \times n$ matrix such that $A^*(i, j)$ is 1 if there is a *path* in $G$ from vertex $i$ to vertex $j$, and is 0 otherwise. $A^*$ is sometimes called the *connectivity matrix of A*. One can obtain $A^*$ by multiplying $A$ by itself $n$ times, which would lead to an inefficient $\Theta(n^2)$ algorithm, or by repeatedly squaring $A$ a logarithmic number of times, which would lead to a suboptimal $\Theta(n \log n)$ time algorithm. However, it is possible to compute $A^*$ in optimal $\Theta(n)$ time by exploiting a modification of Warshall's algorithm. A serial version of Warshall's algorithm [RND77, Wars62] to compute the transitive closure of $A$ is given in Figure 2.5.

```
for k:= 1 to n do
    for i:=1 to n do
        for j:= 1 to n do
            A_k(i,j) := A_{k-1}(i,j) ∨ [A_{k-1}(i,k) ∧ A_{k-1}(k,j)]
```

Figure 2.5:
Warshall's algorithm for computing the transitive closure of matrix $A$.

In this algorithm, $A_0 = A$ and $A_n = A^*$. The interpretation of $A_k(i, j)$ is quite simple: $A_k(i, j)$ is 1 if there is a path from vertex $i$ to



vertex $j$ using no intermediate vertex greater than $k$, and is 0 otherwise. Given this interpretation, the assignment statement in Warshall's algorithm merely states that there is a path from vertex $i$ to vertex $j$ using no intermediate vertex greater than $k$ if and only if

1. there is a path from $i$ to $j$ using no intermediate vertex greater than $k$ - 1, or

2. there is a path from $i$ to $k$ using no intermediate vertex greater than $k$ - 1 and there is a path from $k$ to $j$ using no intermediate vertex greater than $k$ - 1.

In [VanS80], it was shown that if $A(i, j)$ is initially stored in processor $P_{i,j}$, then in $\Theta(n)$ time $A_n$ can be computed, where processor $P_{i,j}$ contains $A_n(i, j)$ when the algorithm terminates. Somewhat remarkably, this algorithm was presented for the weak cellular automata model. What follows is essentially the algorithm as presented in [VanS80], modified only slightly to avoid the extra complications introduced by the cellular automata.

The movement of data in this algorithm is very interesting. For all $k$ and $i$, the value $A_k(i, k)$ moves away from processor $P_{i,k}$ horizontally in row $i$, while for all $k$ and $j$, the value $A_k(k, j)$ moves away from processor $P_{k,j}$ vertically in column $j$. This creates a pattern of data movement that looks like nonoverlapping diamond-shaped waves. The algorithm proceeds so that $A_k(i, j)$ is computed in processor $P_{i,j}$ at time $3k + |k - i| + |k - j| - 2$. The movement of the data can be observed by assuming that the computations have been performed correctly for $k$ - 1 and all values of $i$ and $j$, and then considering the case for $k$ and all values of $i$ and $j$, a discussion of which follows.

To calculate $A_k(*, *)$, note that $A_{k-1}(k, k)$ was "computed" in processor $P_{k,k}$ at time $3k - 3$, and that $A_k(k, k) = A_{k-1}(k, k)$. Hence, $A_k(k, k)$ is "computed" at time $t_k = 3k - 2$. This value will be passed north, south, east, and west at time $t_k + 1$. (See Figure 2.6.) At time $t_k + 1$, processors $P_{k-1,k}$, $P_{k+1,k}$, $P_{k,k+1}$, and $P_{k,k-1}$ receive $A_k(k, k)$ and use it to compute $A_k(k - 1,k)$, $A_k(k + 1,k)$, $A_k(k, k + 1)$, and $A_k(k, k - 1)$, respectively. At time $t_k + 2$, processors $P_{k-1,k}$, $P_{k+1,k}$, $P_{k,k+1}$, and $P_{k,k-1}$ continue to pass copies of $A_k(k, k)$ to the north, south, east, and west, respectively. In addition, at time $t_k + 2$, processor $P_{k-1,k}$ initiates $A_k(k -1,k)$ on its journey horizontally in row $k - 1$, processor $P_{k+1,k}$ initiates $A_k(k + 1, k)$ on its journey horizontally in row $k + 1$, processor $P_{k,k+1}$ initiates $A_k(k, k + 1)$ on its journey vertically in column $k + 1$, and processor $P_{k,k-1}$ initiates $A_k(k, k - 1)$ on its journey



vertically in column $k - 1$. Notice that at time $t_k + 2 = 3k$, processors $P_{k\pm2,k}$, $P_{k,k\pm2}$, and $P_{k\pm1,k\pm1}$, receive the values necessary to compute their $A_k$ values. Further, at time $t_k + d$, $d \geq 2$, all processors at internal distance $d$ from $P_{k,k}$ receive the values required to compute their $A_k$ values. (Some basic algebra can be used to show that the processors had computed their own $A_{k-1}$ entries before time $t_k + d$.) In general, at time $t_k + d$, $d \geq 2$, processors continue to send data received during time $t_k + d - 1$ in the same direction they were going. In addition, during this time step, processors of the form $P_{k\pm d,k}$ ($P_{k,k\pm d}$) initiate the journey of $A_k(k\pm d, k)$ ($A_k(k, k\pm d)$) east and west (north and south). Thus, each entry of $A_k$ is computed at the time claimed.

In this algorithm, no processor ever needs to hold more than a fixed number of entries. Most of these entries are values that are received by a processor, used to compute a new value, and then propagated along the direction they were headed to a neighboring processor. Notice that when a processor $P_{i,j}$ computes a new value $A_k(i, j)$, that $A_k(i, j)$ will supersede $A_{k-1}(i, j)$, which may therefore be written over.

**Theorem 2.3** *Given an $n \times n$ matrix $A$ stored in a mesh of size $n^2$ so that processor $P_{i,j}$ contains $A_{i,j}$, the transitive closure of $A$ may be computed in $\Theta(n)$ time so that processor $P_{i,j}$ knows $A^*_{i,j}$.*

It is easy to see that this pattern of data movement can be extended to show that any recurrence of the form

$$f_k(i, j) = g(f_{k-1}(i, j), f_{k-1}(i, k), f_{k-1}(k, j)) \qquad (2.1)$$

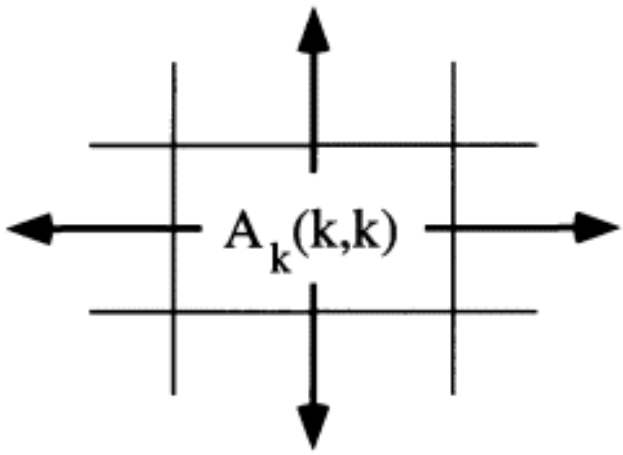
or

$$f_k(i, j) = g(f_{k-1}(i, j), f_k(i, k), f_k(k, j)) \qquad (2.2)$$

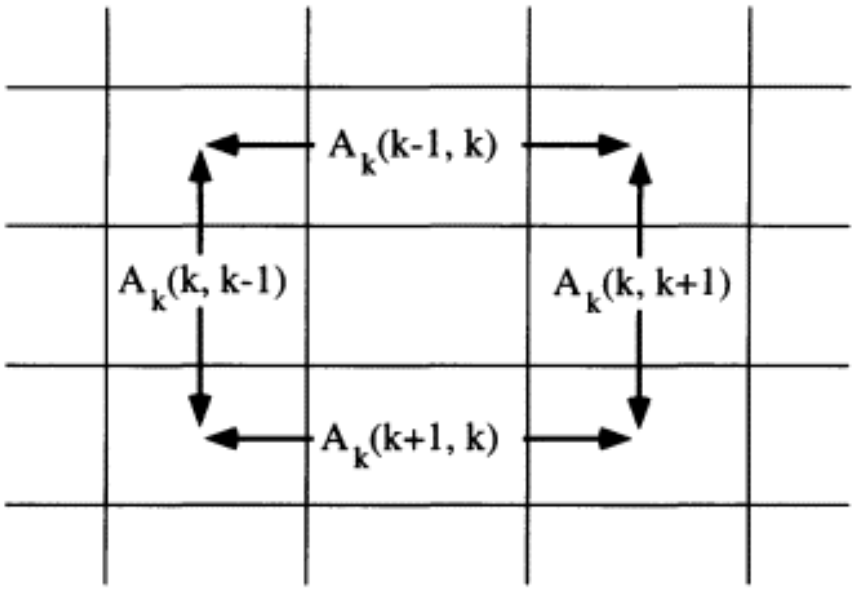
can be solved for all $f_n(i, j)$ in $\Theta(n)$ time if the function $g$ can be computed in O(1) time by a single processor, and if $f_0(i, j)$ is initially stored in processor $P_{i,j}$. Upon termination of the algorithm, $f_n(i, j)$ will be stored in processor $P_{i,j}$.

Finally, some natural uses of van Scoy's transitive closure algorithm should be mentioned. The algorithm can be used to solve the component labeling problem. Suppose that the adjacency matrix $A$ represents an undirected graph $G$. Then the connected components of $G$ can be determined in $\Theta(n)$ time on a mesh of size $n^2$ by computing $A^*$, as just described, followed by a row rotation so that every processor $P_{i,j}$ determines the column index of the first non-zero entry in its row, which will be used as the component label for vertex $i$.

Page 57




At time $t = 3k - 2$, $A_k(k, k)$ is computed in processor $P_{k,k}$
At time $t + 1$, copies of $A_k(k, k)$ begin traveling north, south, east, and west, as indicated.




Values computed at time $t + 1$ and directions of travel at time $t + 2$.

Figure 2.6:
Data movement of the transitive closure algorithm.

Page 58

Another application of the generalized transitive closure algorithm is to solve the *all shortest path problem*. Suppose that a weight matrix $W$ is initially stored one entry per processor on a mesh of size $n^2$ such that processor $P_{i,j}$ initially contains $W(i,j) \geq 0$, which represents the weight of directed edge $\vec{ij}$ (from vertex $i$ to vertex $j$). Then, the all shortest path matrix $W^* = W_n$ can be computed in optimal $\Theta(n)$ time on a mesh of size $n^2$ by following van Scoy's transitive closure algorithm. This results in processor $P_{i,j}$ storing $W_n(i,j)$, which represents the minimum weight of a directed path from vertex $i$ to vertex $j$. The reason that van Scoy's algorithm may be applied is that

$$W_k(i,j) = \min[W_{k-1}(i,j), W_{k-1}(i,k) + W_k\text{-}1(k,j)], \qquad (2.3)$$

which follows the general recurrence given in Equation 2.1. However, notice that a slight modification to the algorithm is required in order to allow processor $P_{i,j}$ to store the label of the vertex incident on vertex $i$ that yields the first edge in a minimal weight path from vertex $i$ to vertex $j$. Details of this modification are left to the reader. In addition, the problem of modifying this algorithm to allow negative weights, in which case it is possible for all vertices on a given cycle to have a minimum weight path of $-\infty$ between each other, is also left to the reader.

The optimal $\Theta(n)$ time transitive closure algorithm can also be used in a straightforward fashion to decide whether or not a graph $G$, given as an adjacency matrix $A$, is a tree. A *tree* with $n$ vertices may be defined as an undirected connected graph with $n$ - 1 edges. The algorithm follows.

1. *Determine if G is undirected:* In $\Theta(n)$ time it can be determined if the graph $G$ is undirected by deciding whether or not $A^T = A$, as described in Section 2.5.1.

2. *Count the number of edges in G:* If the graph is undirected, then a semigroup operation (i.e., an associative binary operation) can be used to count the number of undirected edges in $G$ in $\Theta(n)$ time.

3. *Decide whether or not G is connected:* The transitive closure algorithm can be used to label the vertices of $G$ in $\Theta(n)$ time, as described on page 56. Once labeled, a semigroup operation can be used to decide if all vertices received the same label (i.e., to decide whether or not $G$ is connected).

Since each step of this algorithm takes $\Theta(n)$ time, the entire algorithm is complete in $\Theta(n)$ time.



### *2.5.4 Matrix Inverse*

The intent of this section is to continue with the presentation of mesh algorithms that demonstrate interesting movements of data. In this section, algorithms are presented that overlap pipelined data movements in order to determine the inverse of a matrix. The beginning of this section serves as a review of some fundamental concepts in linear algebra that are used to determine the inverse of a matrix. The reader who is unfamiliar with the linear algebra presented at the beginning of this section may wish to skip this section or concentrate on the data movements that are presented.

An $n \times n$ matrix $A$ is called *nonsingular*, or *invertible*, if and only if there exists an $n \times n$ matrix $B$ such that $AB = BA = I_n$, where $I_n = \{e_{i,j}\}$ is the $n \times n$ *identity matrix* defined by $e_{i,i} = 1$, for $1 \leq i \leq n$, and $e_{i,j} = 0$, for $i \neq j$. The matrix $B$ is called the *inverse of A*, it is unique, and it is typically denoted as $A^{-1}$. If no such inverse matrix exists, then $A$ is called *singular*, or *noninvertible*.

Define an *elementary row operation* on a matrix $A$ to be any one of the following.

1. Interchange row $i$ and row $j$ of $A$.

2. Multiply row $i$ of $A$ by a constant $c \neq 0$. That is, every element $a_{i,j}$ is replaced by $ca_{i,j}, 1 < j < n$.

3. Add a constant $c$ times row $i$ of $A$ to row $j$ of $A$. That is, each element $a_{j,k}$ is replaced by $a_{j,k} + ca_{i,k}, 1 \leq k \leq n$.

*Gaussian elimination*, followed by *back-substitution*, can often be used to determine the inverse of a given $n \times n$ matrix $A = \{a_{i,j}\}$. This method, which requires $\Theta(n^3)$ time on a serial machine, is described in Figure 2.7. A sample of this algorithm is given in Figure 2.10. Notice that if $a_{i,i} = 0$ at the beginning of phase $i$ in Step 2(a) of the algorithm presented in Figure 2.7, then the algorithm will terminate in order to avoid division by zero. Unfortunately, this termination only means that the algorithm fails. It does not necessarily mean that $A^{-1}$ does not exist (i.e., that $A$ is singular). Later in this section, modifications to the algorithm of Figure 2.7 will be discussed that guarantee to find the inverse of a matrix, if one exists.

To implement the algorithm given in Figure 2.7 on a mesh of size $n^2$, a decision needs to be made as to how and where to store the $n \times 2n$ augmented matrix $[A_{n \times n} | I_n]$ which will, through the use of elementary row operations, be transformed into $[I_n | A^{-1}_{n \times n}]$. The input to the matrix



1. Form the augmented matrix $[A_{n\,n} | I_n]$.

2. Perform elementary row operations to transform $[A_{n \times n} | I_n] \rightarrow [I_n | A^{-1}_{n \times n}]$ by straightforward Gaussian elimination, as follows.

   (a) At phase $i, 1 \leq i \leq n-1$, do the following.

      i. If $a_{i,i} = 0$, then terminate the algorithm to avoid division by 0.

      ii. Otherwise, all rows $j, j > i$, use row $i$ to eliminate (i.e., set to zero) their entry in column $i$. Specifically, for all $j, i < j \leq n$, set $a_{j,k} \leftarrow a_{j,k} - a_{i,k} \frac{a_{j,i}}{a_{i,i}}$

   (b) In every row $i$, set $a_{i,j} \leftarrow a_{i,j}/a_r$, for all $j, i \leq j \leq n$. This creates the augmented matrix, as shown on the right side of the arrow in Figure 2.8.

   (c) The final step of this algorithm continues to rely on elementary row operations. Back-substitution uses elementary row operations to transform the left half of the augmented matrix (which is currently in the form of an upper-triangular matrix) into the identity matrix, as follows. (See Figure 2.9.) At phase $i, 2 \leq i \leq n$, all rows $j, 1 \leq j < i$, use row $i$ to eliminate their entry in column $i$. Specifically, for all $j, 1 \leq j < i$, set $a_{j,k} \leftarrow a_{j,k} - a_{i,k} \frac{a_{j,i}}{a_{i,i}}$

Figure 2.7:
Using Gaussian elimination, followed by back-substitution,
to determine the inverse of an $n \times n$ matrix $A = \{a_{i,j}\}$.

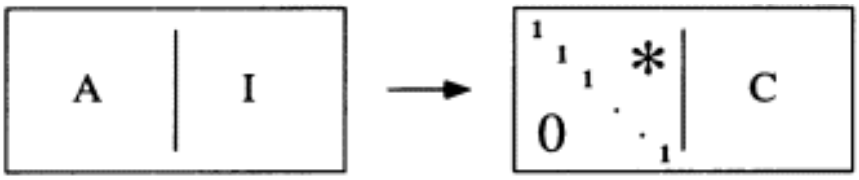



Figure 2.8:
Transform $A$ to an upper-triangular matrix.



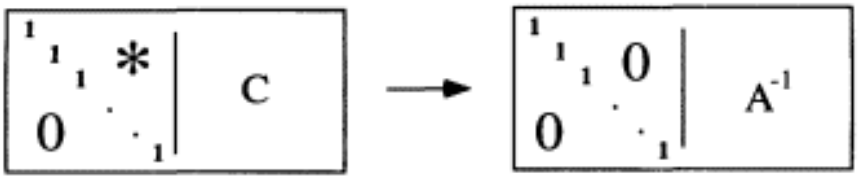



Figure 2.9:
Transform upper-triangular matrix to identity matrix.

inverse problem on a mesh of size $n^2$ is an $n \times n$ matrix $A = \{a_{i,j}\}$, stored so that processor $P_{i,j}$ contains $a_{i,j}$. The problem requires the resultant matrix $A^{-1} = \{a_{i,j}^{-1}\}$, if it exists, be stored so that processor $P_{i,j}$ contains $a^{-1}$. Let $I_n = \{e_{i,j}\}$. Two representations for embedding $[A_{n \times n} | I_n]$ in a mesh of size $n^2$ follow.

1. Let the $n \times n$ mesh simulate an $n \times 2n$ mesh, with each processor being responsible for two consecutive column entries of the augmented matrix. That is, given $G_{n \times 2n} = [A_{n \times n} | I_n] = \{g_{i,j}\}$, let processor $P_{i,j}$ be responsible for $g_{i,2j-1}$ and $g_{i,2j}$. A disadvantage of this representation is that the original matrix must be packed into the appropriate region, and the resultant inverse matrix must be unpacked. Although packing and unpacking can be done so as not to affect the asymptotic running times of the algorithms, these extra steps may be avoided by using the following representation.

2. Let processor $P_{i,j}$ be responsible for $a_{i,j}$ and $e_{i,j}$. When the algorithm terminates, processor $P_{i,j}$ will contain $a_{i,j}^{-1}$. Notice that this representation initially superimposes matrix $A$ with identity matrix $I$.

The second representation will be used in the mesh algorithms described in this section, as it avoids packing and unpacking, and also allows for the algorithms to be presented in terms of elementary row operations performed on $A$, with the understanding that the equivalent operations are performed on $I$.

Details of a straightforward implementation of the Gaussian elimination stage of the algorithm presented in Figure 2.7 for a mesh of size $n^2$ are given in Figure 2.11. The assumption is made in the algorithm presented in Figure 2.11 that the elements of $I$ are moved in lock-step with the elements of $A$, and equivalent operations are performed on these entries. Further, the details of the back-substitution stage of the



| A | | | $I_3$ | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 1 | 0 | 0 | |
| 2 | 5 | 3 | 0 | 1 | 0 | Subtract 2 times the first row from the second row to obtain: |
| 1 | 0 | 8 | 0 | 0 | 1 | |
| 1 | 2 | 3 | 1 | 0 | 0 | |
| 0 | 1 | -3 | -2 | 1 | 0 | Subtract 1 times the first row from the third row to obtain: |
| 1 | 0 | 8 | 0 | 0 | 1 | |
| 1 | 2 | 3 | 1 | 0 | 0 | |
| 0 | 1 | -3 | -2 | 1 | 0 | Add 2 times the second row to the third to obtain: |
| 0 | -2 | 5 | -1 | 0 | 1 | |
| 1 | 2 | 3 | 1 | 0 | 0 | |
| 0 | 1 | -3 | -2 | 1 | 0 | Multiply the third row by -1 to obtain: |
| 0 | 0 | -1 | -5 | 2 | 1 | |
| 1 | 2 | 3 | 1 | 0 | 0 | |
| 0 | 1 | -3 | -2 | 1 | 0 | |
| 0 | 0 | 1 | 5 | -2 | -1 | |

$A$ has been transformed to an upper-triangular matrix.
(The example is continued on the next page.)



(continued from previous page)
Transform the upper-triangular matrix A to the identity matrix.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | | 1 | 0 | 0 |
| 0 | 1 | -3 | | -2 | 1 | 0 |
| 0 | 0 | 1 | | 5 | -2 | 1 |
| 1 | 0 | 9 | | 5 | -2 | 0 |
| 0 | 1 | -3 | | -2 | 1 | 0 |
| 0 | 0 | 1 | | 5 | -2 | -1 |
| 1 | 0 | 9 | | 5 | -2 | 0 |
| 0 | 1 | 0 | | 13 | -5 | -3 |
| 0 | 0 | 5 | | 5 | -2 | -1 |
| 1 | 0 | 0 | | -40 | 16 | 9 |
| 0 | 1 | 0 | | 13 | -5 | -3 |
| 0 | 0 | 1 | | 5 | -2 | -1 |
| $I_3$ | | | | $A^{-1}$ | | |

Add -2 times the second row to the first row to obtain:

Add 3 times the third row to the second row to obtain:

Add -9 times the third row to the first row to obtain:

Figure 2.10:
Sample of Gaussian elimination followed by
back-substitution to determine the inverse of matrix $A_{3\times3}$.



algorithm have been omitted since they are similar to the Gaussian elimination stage. At the conclusion of the entire algorithm, including the back-substitution, $A$ will be reduced to the identity matrix and the initial identity matrix will have been transformed to $A^{-1}$. Notice that the mesh algorithm given in Figure 2.11 finishes in $\Theta(n^3)$ time, which is the same as the time it takes to complete the algorithm of Figure 2.7 when implemented on a serial machine. Improvements to the mesh algorithm are now discussed.

By incorporating pipelining into the mesh algorithm of Figure 2.11, the running time can be reduced to $\Theta(n^2)$. This is done by noticing that there is no need to wait for the $i^{th}$ row to be completely processed in order to initiate computations involving the $(i + 1)^{st}$ row. Pipelining may be used to move the rows of matrix $A$ through the mesh, while making sure that the values arrive in the required sequence. Specifically, computations involving row $i + 1$ may begin as soon as row $i + 2$ has completed its computations that involve row $i$.

In order to further reduce the running time of the mesh algorithm given in Figure 2.11, pipelining must occur not only row-wise (vertically), as just described, but within each row (horizontally). Horizontal pipelining will be used to avoid the delay previously incurred in waiting for a row to complete all of its computations before sending data to the next row. This creates a wave-like movement of data, not unlike the data movement associated with the transitive closure algorithm presented in Section 2.5.3. The algorithm is presented in Figure 2.12. (Again, the algorithm is described with respect to the processors operating on matrix $A$, where it is understood that the equivalent operations are performed by the processors on matrix $I$.) These final modifications to the mesh algorithm of Figure 2.11 produce an optimal $\Theta(n)$ time mesh implementation of the algorithm given in Figure 2.7. (It should be noted that this algorithm is conceptually similar to the systolic array algorithms developed in [GeKu81] for triangularizing a matrix.)

***Theorem 2.4*** *Given an $n \times n$ matrix $A = \{a_{i,j}\}$ stored on a mesh of size $n^2$ so that processor $P_{i,j}$ contains $a_{i,j}$, the inverse of $A$ can be determined by the method of Gaussian elimination, followed by back-substitution, in $\Theta(n)$ time so that processor $P_{i,j}$ stores $a_{i,j}^{-1}$, if the method produces the inverse.*

Certain numerical stability issues have not been considered in this section. Throughout, it has been assumed that either an *exact* inverse of $A$ exists, or that the inverse of $A$ does not exist. That is, the elements



{Perform Gaussian elimination to transform $A$ into an upper-triangular matrix.}

for $i$:= 1 to $n$ -1 do

{Initiate the elimination process based on row $i$.} processor $P_{i,i}$ terminates the algorithm if $a_{i,i} = 0$.

for $j$:=$i$ + 1 to $n$ do

{Use row $i$ to perform elimination on row $j$. Note that a copy of row $i$ currently exists in row $j$ - 1 of the mesh.}

In unit time, send a copy of the $i^{th}$ row of $A$ from row $j$ - 1 of the mesh to row $j$ of the mesh.

processor $P_{j,i}$ computes row $j$'s *multiplier*, $M = a_{j,i}/a_{i,i}$.

for $k$:=$i$ to $n$ do

processor $P_{j,k}$ sets $a_{j,k} \leftarrow a_{j,k} - Ma_{i,k}$.

Send $M$ from processor $P_{j,k}$ to processor $P_{j,k+1}$.

endfor

endfor

endfor

{Perform back-substitution to finish transformation of $A \rightarrow I$.} forall $i$ do

for $j := i$ to $n$ do

processor $P_{i,j}$ sends a copy of $a_{i,i}$ to processor $P_{i,j+1}$

processor $P_{i,j}$ sets $a_{i,j} \leftarrow a_{i,j}/a_{i,i}$

end forall

Figure 2.11:
Straightforward mesh implementation of a Gaussian elimination algorithm for finding the inverse of a matrix.



1. At time $3i - 2$, processor $P_{i,i}$ initiates computations involving element $a_{i,i}$.

(a) If $a_{i,i} = 0$, then processor $P_{i,i}$ broadcasts a 'halt' message to all processors, and the algorithm terminates. Otherwise, the algorithm continues.

(b) Processor $P_{i,i}$. sends a copy of $a_{i,i}$ to processors $P_{i+1,i}$ and $P_{i,i+1}$.

(c) Processor $P_{i,i}$ sets $a_{i,i} \leftarrow 1$.

2. When any processor $P_{i,j}$ receives a data value, call it $W$, from processor $P_{i,j-1}$, and does not receive a data value from processor $P_{i-1,j}$, processor $P_{i,j}$ performs the following.

(a) Update $a_{i,j}$ by setting $a_{i,j} \leftarrow a_{i,j}/W$.

(b) Send an updated copy of $a_{i,j}$ to processor $P_{i+1,j}$, and a copy of $W$ to processor $P_{i,j+1}$.

3. When any processor $P_{i,j}$ receives a data value, call it $N$, from processor $P_{i-1,j}$, the following is performed.

(a) If processor $P_{i,j}$ simultaneously received a piece of data, call it $W$, from processor $P_{i,j-1}$, then processor $P_{i,j}$ will perform the following.

i. Update $a_{i,j}$ by setting $a_{i,j} \leftarrow a_{i,j} - NW$.

ii. Send a copy of $N$ to processor $P_{i+1,j}$, and a copy of $W$ to processor $P_{i,j+1}$.

(b) If processor $P_{i,j}$ does not simultaneously receive a piece of data from processor $P_{i,j-1}$, then processor $P_{i,j}$ will perform the following.

i. Determine $M = a_{i,j}/N$, the multiplicative constant to be used by row $i$ in the Gaussian elimination.

ii. Set $a_{i,j} \leftarrow 0$.

iii. Send a copy of $M$ to processor $P_{i,j+1}$, and a copy of $N$ to processor $P_{i+1,j}$.

4. At time $3n-1$, back-substitution begins. This is similar to the Gaussian elimination given in Steps 1-3.

Figure 2.12:
An optimal mesh algorithm for using Gaussian elimination followed by backsubstitution to find the inverse of an $n \times n$ matrix $A = \{a_{i,j}\}$.



of $A$ have been assumed to be taken over a finite field. However, even with these restrictions, it is easy to create matrices for which an inverse exists, and for which the method of finding the inverse that is given in Figure 2.7 will fail.

To avoid a situation in which the inverse of a matrix $A$ exists, and the method described in Figure 2.7 fails to find $A^{-1}$, *pivoting* may be used to guarantee that for an invertible matrix over a finite field, the inverse will always be determined. Quite simply, pivoting deals with the situation in which at the beginning of a phase that will perform computations with row $i$, entry $a_{i,i} = 0$. The algorithm can be modified to incorporate pivoting as follows. If $a_{i,i} = 0$ at the beginning of the phase concerned with row $i$, then row $j$ and row $i$ are interchanged, where $i < j \leq n$ is chosen to be the smallest value such that $a_{j,i} \neq 0$. If no such j exists, then the algorithm halts with the assurance that no inverse exists.

This modification can be incorporated into the mesh implementation given in Figure 2.12 and still finish in optimal $\Theta(n)$ time. A sketch of this modification is now given, with the details left to the reader. Suppose that at time $3i - 2$, processor $P_{i,i}$ detects that $a_{i,i} = 0$. At this point, processor $P_{i,i}$ initiates a message that will be passed down column $i$ to search for the first nonzero entry $a_{j,i}$. Notice that if such a $j$ is found, then all entries in column $i$ between row $i$ and row $j$ are necessarily zero, while if such a $j$ does not exist, then processor $P_{n,i}$ may broadcast a message to all processors indicating that the original matrix is singular. Assuming that such a $j$ is found, row $j$ is 'bubbled-up' to row $i$ in a wave-like fashion that incorporates vertical and horizontal pipelining (starting with row $j - 1$ and ending with row $i$, each row $k$ is moved to row $k + 1$ as they detect row $j$ moving through them). While row $j$ is being bubbled-up, it also initiates the Gaussian elimination of the column entries under it, following the previous algorithm. Finally, it should be noted that when processor $P_{i,i}$ detects that $a_{i,i} = 0$, it simply sends out a flag alerting

the other processors of the situation, instead of sending out data. Notice that this does not impede the start of subsequent phases of the algorithm.

**Corollary 2.5** *Given an $n \times n$ matrix $A = \{a_{i,j}\}$ stored on a mesh of size $n^2$ so that processor $P_{i,j}$ contains $a_{i,j}$, the inverse of A can be determined by the method of Gaussian elimination with pivoting, followed by back-substitution, in $\Theta(n)$ time so that processor $P_{i,j}$ stores $a_{i,j}^{-1}$, if A exists. If $A^{-1}$ does not exist, then all processors will be so informed.* ·



## 2.6 Algorithms Involving Ordered Data

Besides organizing data in rows or columns, it is often advantageous to have the data ordered with respect to a linear ordering of the processors. (For sample orderings of the mesh processors, the reader is referred to Figure 2.2.) The *snake-like ordering* has been popular for mesh algorithms since it has the useful property that processors with consecutive numbers in the ordering are adjacent in the mesh. The *shuffled row-major ordering* has also been popular for mesh algorithms since it has the property that the first quarter of the processors form one quadrant, the next quarter form another quadrant, etc., with this property holding recursively within each quadrant. This property of shuffled row-major ordering is quite useful for implementing recursive divide-and-conquer algorithms on a mesh. *Proximity ordering* combines the advantages of the snake-like and shuffled row-major orderings, and will be used in Chapter 4 in order to facilitate a more cohesive explanation of some sophisticated algorithms. Since the proximity ordering requires additional overhead to compute the indices, it will not be used in the algorithms presented in this chapter or in Chapter 3.

In Section 2.6.1, optimal mesh algorithms are given for sorting data into any predefined linear ordering of the processors. Sorting is often used to place data into *ordered intervals*, i.e., disjoint consecutive sequences of items in the sorted order. In Section 2.6.2, the notion of row and column rotations is generalized to the notion of rotating data within ordered intervals. In Section 2.6.3, an optimal mesh algorithm is given to perform an associative binary operation within every ordered interval. Section 2.6.4 presents mesh implementations of concurrent read and concurrent write. The implementations of these operations are based on being able to efficiently sort data on the mesh. Section 2.6.5 also uses sorting as a fundamental operation. This section presents an optimal algorithm to compress distinct elements into a section of the mesh where optimal interprocessor communication will be possible. Finally, it should be noted that many of the fundamental *grouping operations*, which will be introduced in Chapter 4, are based on sorting data into ordered intervals.

### *2.6.1 Sorting*

It is well known that comparison-based sorting of $n^2$ elements on a serial machine requires $\Omega(n^2 \log n)$ time [CLR92]. Thompson and Kung [ThKu77] have shown that $n^2$ elements, distributed one element per



Figure 2.13:
A linear array of size $n$ with input from the left and output to the right.

processor on a mesh of size $n^2$, can be sorted in $\Theta(n)$ time by using a recursive merging procedure that adapts the *odd-even transposition sort*. They also noted that any algorithm that sorts $n^2$ elements in $\Theta(n)$ time on a mesh of size $n$, for some processor ordering $R$, can be used to sort $n^2$ elements in $\Theta(n)$ time for any other processor ordering $R'$, given the following constraints. Every processor $i \in R$ (i.e., every processor at position $i$ with respect to processor ordering $R$) must be able to determine, in $\Theta(n)$ time, the processor index $\pi(i) \in R$, where processor $\pi(i) \in R$ corresponds to processor $i \in R'$. For example, referring back to Figure 2.2, if $R$ is row-major ordering and $R'$ is shuffled row-major ordering, then processor $i$ must determine $\pi(i)$ as follows.

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\pi(i)$ | 0 | 1 | 4 | 5 | 2 | 3 | 6 | 7 | 8 | 9 | 12 | 13 | 10 | 11 | 14 | 15 |

Sorting with respect to $R'$ may be accomplished as follows. First, sort the data with respect to $R$. Next, determine for each processor $i \in R$, the aforementioned index $\pi(i) \in R$. Finally, re-sort the data with respect to $R$, using the $\pi(i)$ values as keys. Notice that the reordering can be accomplished in $\Theta(n)$ time if each processor can determine its position in each ordering in $\Theta(n)$ time (details left to the reader). In practice, this reordering can usually be carried out by simpler operations such as row and column rotations.

Nassimi and Sahni [NaSa79] showed that *bitonic sort* [Batc68] can be implemented to sort $n^2$ elements in $\Theta(n)$ time on a mesh. Much work has been done to reduce the high order coefficients of mesh sorting algorithms, including [KuHi83, MSS86, ScSe89, SSM89, ScSh86b], to name a few. In fact, an algorithm is presented in [ScSh86b] that is optimal including the high order coefficient.

Consider a *linear array* of $n$ processors, in which data items $d_1, d_2, \ldots, d_n$ are input from the left and results are output to the right, as shown in Figure 2.13. The input data may be efficiently sorted, such that the $i^{th}$ processor contains the $i^{th}$ smallest value, by allowing the



processors to *i*) view the data as it passes from left to right and *ii*) retain the minimum data item encountered. The data can then be output in nonincreasing order by performing a series of $n$ lockstep shifts to the right. Each processor initializes both its *min-reg* and *transfer-reg* registers to $+\infty$. During time step $t$, $1 \le t \le 2n-1$, the following operations are performed. (Refer to Figure 2.14.)

1. Processor $P_0$ receives input data item $d_t$, $1 \le t \le n$.

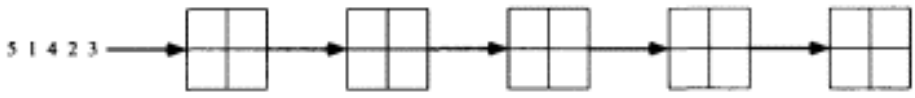
2. Processor $P_{i-1}$ sends a copy of its *transfer-reg* register to processor $P_i$, $\lfloor t/(n+1) \rfloor (t-n-1)+1 \le i \le \lfloor (t-1)/2 \rfloor$

3. Every processor $P_i$, $\lfloor t/n \rfloor (t-n) \le i \le \lfloor (t-1)/2 \rfloor$, compares the data item just received to the minimum value of the data it has seen thusfar, which is stored in its *min-reg* register, and places the minimum of these two values in *min-reg* and the maximum of these two values in *transfer-reg*.

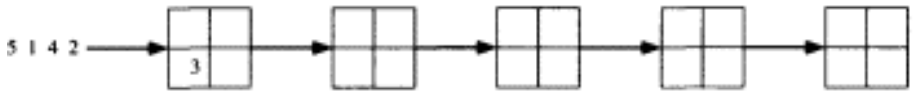
Notice that at time $n$, the final piece of input data is input to and viewed by processor $P_0$ . Processor $P_0$ sends its final message to processor $P_1$ at time $n + 1$. In general, a final message is sent from processor $P_{i-1}$ to processor $P_i$ at time $n + i$, $1 \leq i \leq n - 1$. Therefore, after step $2n - 1$ is complete, the array contains the sorted set of data with respect to the indexing of the processors, as shown in Figure 2.14(j), which may now be output (in nonincreasing order) by a series of $n$ lockstep right shifts of this final ordered set of data.

Consider the problem of sorting $n$ data items, arbitrarily distributed one per processor on a 1-dimensional mesh of size $n$. Upon termination of the algorithm, assume that the data is to be distributed one per processor such that the item contained in processor $P_{i-1}$ is less than or equal to the item contained in processor $P_i$, $1 \leq i \leq n- 1$. That is, the data is to be sorted into nondecreasing order with respect to the indexing of the processors. Notice that a simulation of the first $2n - 1$ steps of the linear array algorithm, as just described, will suffice. This simulation is straightforward. In addition to the *min-reg* and *transfer-reg* registers, each processor also maintains an *input-reg* register, which is used to track the movement of the input data items as discussed in the linear array algorithm. Initially, input data item $d_i$ resides in the *input-reg* register of processor $P_{i-1}$, $1 \leq i \leq n$. The algorithm is identical to the linear array algorithm, with the exception that processor $P_0$ gets the input data item from its *input-reg* register instead of from the external environment, and that at the end of every step, every processor $P_i$ sends

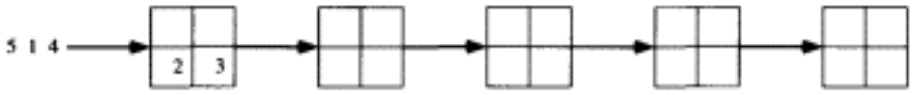




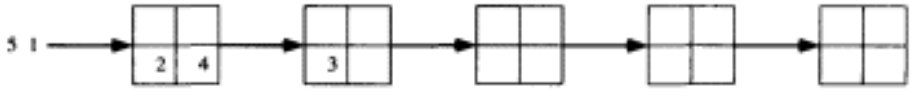

(a) Initial configuration.



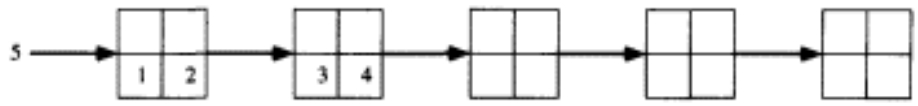

(b) Configuration at the completion of time $t = 1$.



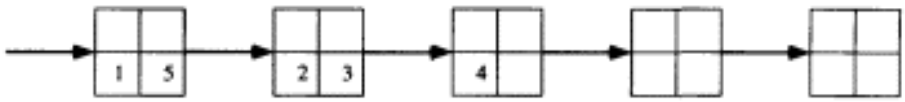

(c) Configuration at the completion of time $t = 2$.




(d) Configuration at the completion of time $t = 3$.

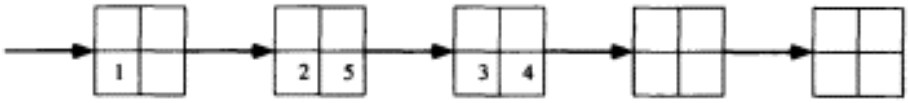



(e) Configuration at the completion of time $t = 4$.

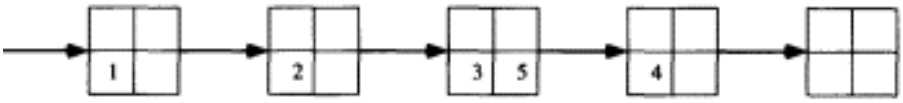



(f) Configuration at the completion of time $t = 5$.

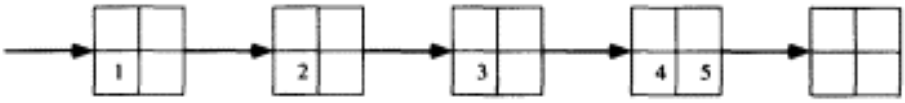




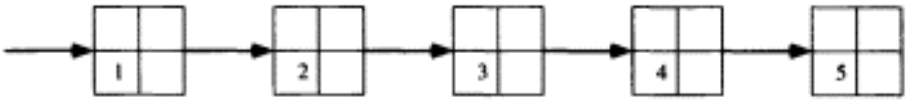

(g) Configuration at the completion of time $t = 6$.




(h) Configuration at the completion of time $t = 7$.




(i) Configuration at the completion of time $t = 8$.




(j) Configuration at the completion of time $t = 9$.

Figure 2.14:
Sorting data on a linear array of size 5 with input from the left.



a copy of it *input-reg* register to processor $P_{i-1}$, $1 \leq i \leq n - 1$, which sets its *input-reg* register to this new value. See Figure 2.15. That is, at the end of every step, the remaining, unprocessed, input data items, are shifted to the left in lockstep fashion so as to simulate the linear array input to processor $P_0$ from the external environment. Therefore, the running time is again $\Theta(n)$, which is optimal in the worst case.

Notice that this rotation-based sorting algorithm is the parallel analogue of *selection sort*. That is, processor $P_0$ considers all $n$ pieces of data and retains the minimum. Processor $P_1$ considers the remaining $n - 1$ pieces of data and retains the minimum of these. In general, processor $P_i$ considers the $n - i$ largest elements and retains the minimum.

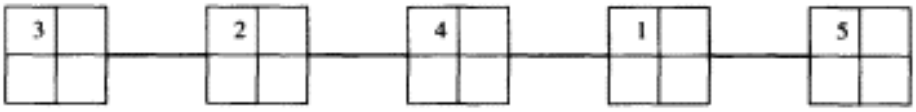
**Lemma 2.6** *Given n pieces of data arbitrarily distributed one per processor on a* 1*-dimensional mesh of size n, in optimal* $\Theta(n)$ *time the data can be sorted by a rotation-based algorithm.*

An alternative method, known as *odd-even transposition sort*, can also be used to give a worst-case asymptotically optimal algorithm to sort data on a 1-dimensional mesh. This algorithm is, in fact, the parallel analogue of *bubblesort*, and is used as the base case in 2-dimensional mesh sorting algorithms presented later in this section. Given $n$ pieces of data stored one piece per processor on a 1-dimensional mesh of size $n$, in odd-even transposition sort, every odd numbered processor $P_{2_i-1}$, $1 \le i \le {}^n/_2$, alternately compares its data with the data in processor $P_{2_i-2}$, $1 \le i \le {}^n/_2$ and data in processor $P_2 i$, $1 \le i \le {}^n/_2$ 1. In the first step, every odd numbered processor $P_{2_i-1}$ receives the data element stored in processor $P_{2_i-2}$. It compares the keys of the data elements, keeping the larger and returning the smaller. In the second step, every odd numbered processor $P_{2_i-1}$ receives the data element stored in processor $P_{2_i}$, compares the keys of the two data elements, keeps the smaller and returns the larger. This alternates for a total of $n$ iterations.

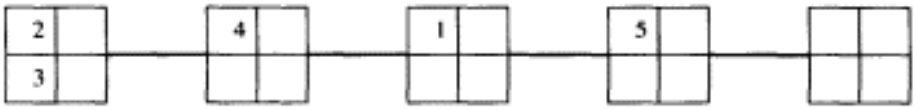
The correctness of the algorithm [Habe72] is based on the fact that after $i$ pairs of comparisons no element is more than $n - 2i$ positions from its final destination. Therefore, ${}^n/_2$ pairs of comparisons are sufficient to sort the $n$ data elements.

**Lemma 2.7** *Given n pieces of data arbitrarily distributed one per processor on a* 1*-dimensional mesh of size n, in* $\Theta(n)$ *time the data can be sorted by odd-even transposition sort.*

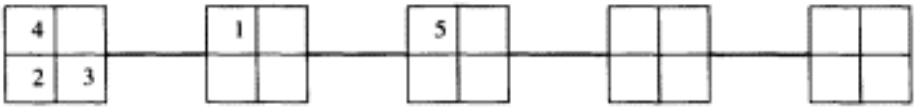





(a) Initial configuration.

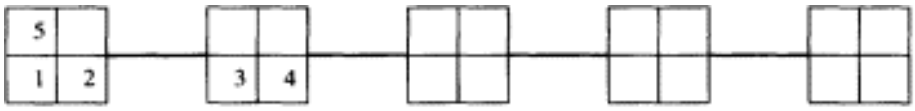



(b) Configuration at the completion of time $t = 1$.

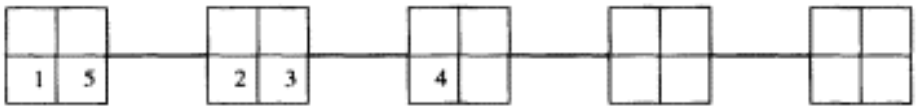



(c) Configuration at the completion of time $t = 2$.


(d) Configuration at the completion of time $t = 3$.

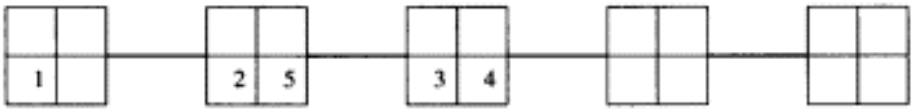



(e) Configuration at the completion of time $t = 4$.

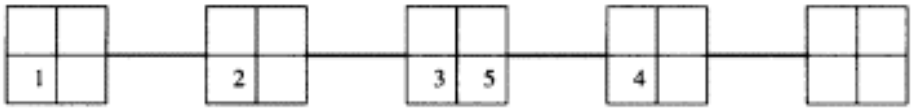



(f) Configuration at the completion of time $t = 5$.

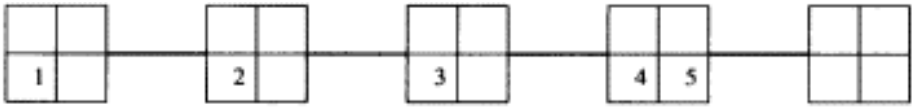




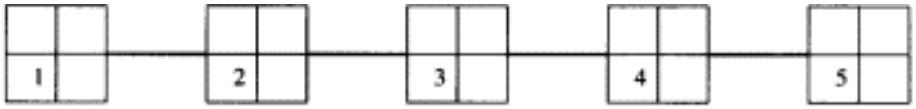

(g) Configuration at the completion of time $t = 6$.




(h) Configuration at the completion of time $t = 7$.




(i) Configuration at the completion of time $t = 8$.




(j) Configuration at the completion of time $t = 9$.

Figure 2.15:
Sorting data on a 1-dimensional mesh of size 5.



Notice that by using the snake-like ordering of processors, a 2 dimensional mesh of size $n^2$ can be viewed as a 1-dimensional mesh of size $n^2$. Therefore, given 1 piece of data per processor on a 2-dimensional mesh of size $n^2$, the odd-even transposition sort can be used to sort these $n^2$ items in $\Theta(n^2)$ time. Although this is an improvement over the $\Omega(n^2 \log n)$ serial comparison-based sorting bound, it is quite far from the $\Omega(n)$ lower bound discussed in Section 2.4 for a 2-dimensional mesh of size $n^2$.

Thompson and Kung [ThKu77] present an algorithm to sort $n^2$ elements, distributed one per processor on a 2-dimensional mesh of size $n^2$, in asymptotically optimal $\Theta(n)$ time. This algorithm, given in Theorem 2.9, is essentially a mergesort algorithm that exploits Batcher's odd-even merge technique. Lemma 2.8 shows how Batcher's odd-even merge algorithm can be used to merge the concatenation of two ordered sequences, distributed in a natural linear fashion one element per processor on a 1-dimensional mesh of size $n$, in asymptotically optimal $\Theta(n)$ time. The techniques and results of Lemma 2.7 and Lemma 2.8 will be used in the asymptotically optimal 2-dimensional mesh sorting algorithm that is presented in Theorem 2.9.

**Lemma 2.8** *Given the concatenation of two ordered sequences, distributed in a natural linear fashion one element per processor on a 1-dimensional mesh of size n, the elements may be merged by Batcher's odd-even merge algorithm in* $\Theta(n)$ *time.*

*Proof.* Given the concatenation of two sorted sequences, say $\{u_i\}$ and $\{v_i\}$, each of length $j = 2^{k-1}$, distributed one element per processor on a 1-dimensional mesh of size $n = 2^k$ so that processor $P_i$ contains $u_i$ and processor $P_{i+n/2}$ contains $v_i$, $1 \le i \le n/2$, the data may be merged into sorted order by Batcher's odd-even merge, as follows.

(1) *Unshuffle* the even and odd terms to form

(a) *odd sequences* $\{u_1, u_3, \ldots, u_{j-1}\}$ and $\{v_1, v_3, \ldots, v_{j-1}\}$, concatenated and stored one element per processor in the first $n/2$ processors, and

(b) *even sequences* $\{u_2, u_4, \ldots, u_j\}$ and $\{v_2, v_4, \ldots, v_j\}$, concatenated and stored one element per processor in the last $n/2$ processors.

(2) Recursively merge the odd and even sequences in parallel, to yield $\{o_i\}$ and $\{e_i\}$, respectively, the concatenation of which is stored one element per processor.



(3) *Shuffle* the odd and even sequences back together, so that the sequence $\{f_i\} = \{o_1, e_1, o_2, e_2, \ldots, o_j, e_j\}$ is stored in a linear fashion, one element per processor.

(4) Perform a *comparison-interchange* between $f_i$ and $f_{i+1}$, $i = 2, 4, \ldots, 2j - 2$, so that if $f_i \le f_{i+1}$, the elements remain in their current position, while if $f_i > f_{i+1}$, the elements are swapped.

Figure 2.16 shows an example of odd-even merge for a 1-dimensional mesh. The proof of correctness of the algorithm can be found in the appendix of [Batc81] and relies on a straightforward counting argument. Of primary importance, is showing that after Step (3) of the algorithm, no element is more than one position from its final destination, and that Step (4) will properly correct any such problems.

Step (1) of the algorithm can be complete in $\Theta(n)$ time since elements just march towards their destinations. No processor is ever required to store more than a small fixed number of additional records (one passing through from left to right and one from right to left). Step (3) is just the inverse of Step (1) and is also complete in $\Theta(n)$ time. Step (4) requires $\Theta(1)$ time, since all comparison-interchange operations are disjoint and can be done in parallel. Therefore, the running time of the algorithm obeys the recurrence $T(n) = T(n/2) + \Theta(n)$, which is $\Theta(n)$.

An asymptotically optimal $\Theta(n)$ time algorithm from [ThKu77] is now presented that will sort $n^2$ elements on a (2-dimensional) mesh of size $n^2$. The algorithm is based on the techniques and results presented in the previous two lemmas.

**Theorem 2.9** *Given $n^2$ pieces of data distributed one piece per processor on a mesh of size $n^2$, in $\Theta(n)$ time the data can be sorted.*

*Proof.* The algorithm follows a standard bottom-up recursive mergesort strategy. Therefore, as with any mergesort, the crucial step to describe is the merge step. The mergesort itself simply consists of merging runs of length 1 to create ordered runs of length 2, then merging ordered runs of length 2 to create ordered runs of length 4, and so on until two ordered runs, each containing half of the elements, are merged to form the sorted list.

Given two sorted lists, the major steps of the *merge* operation are similar to the steps given in the 1-dimensional odd-even merge algorithm of Lemma 2.8, and are as follows.

1. *Unshuffle*



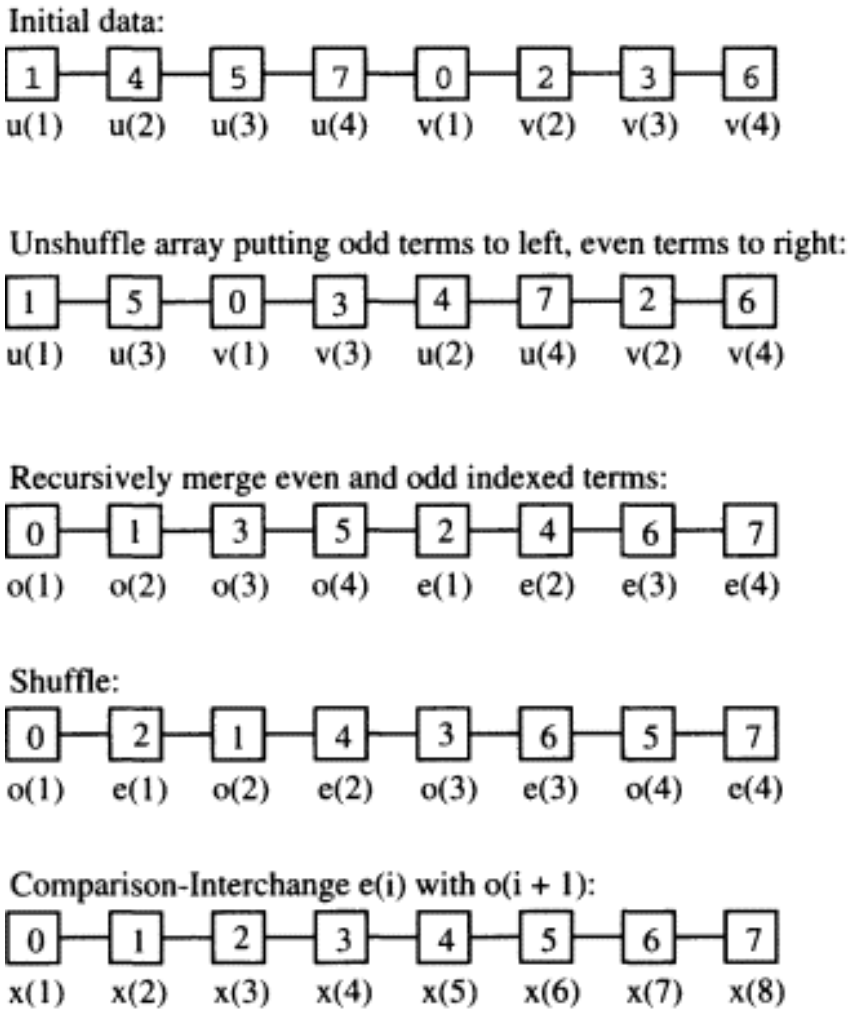



Figure 2.16:
Merging the concatenation of $u$ and $v$ into $x$ on a 1-dimensional mesh by odd-even merge.

2. *Recursively merge*

3. *Shuffle*

4. *Comparison-Interchange*

Define $M(j, k, s)$ to be the algorithm for merging $2s$ disjoint arrays, each of size $j/s \times k/2$, in a $j \times k$ submesh, where $j$, $k$, and $s$ are all powers of 2, and the elements of the arrays are ordered by the snake-like indexing scheme. Notice that $M(j,2, s)$ can be performed by using a 1-dimensional sorting algorithm with respect to the snake-like indexing of the $2j$ elements in the $j \times 2$ array. The general algorithm for $M(j, k, s)$ can be expressed as follows (see Figure 2.17).

1. *Unshuffle* the arrays:

   (a) Perform a single interchange on even rows if $j > s$, so that the columns contain either all even or all odd indexed elements. If $j = s$, then do nothing since the columns are already segregated as such.

   (b) Unshuffle all rows, as described in Lemma 2.8, so that each column has either all even or all odd indexed terms from the original arrays.

2. *Recursively merge* by calling $M(j, k/2, s)$ on each $j \times k/2$ half of the mesh.

3. *Shuffle* the arrays by performing the inverse operations of Step 1 in reverse order:

   (a) Shuffle all rows.

   (b) Perform a single interchange on even rows if $j > s$.

4. Viewing the $j \times k$ mesh as a 1-dimensional mesh of size $jk$, defined by the snake-like indexing scheme, perform the first $2s - 1$ parallel *comparison-interchange* steps of the odd-even transposition sort, as described in Lemma 2.7.

The proof of correctness is demonstrated by use of the *0-1 principle* [Knut73], which states that if a network sorts all sequences of 0's and 1's, then it will sort any arbitrary sequence of elements chosen from a linearly ordered set. Therefore, assume that all inputs to the merge algorithm are 0's and 1's. After unshuffling, there may be as many as $2s$ more 0's on the left half as on the right half of the $j \times k$ mesh. After

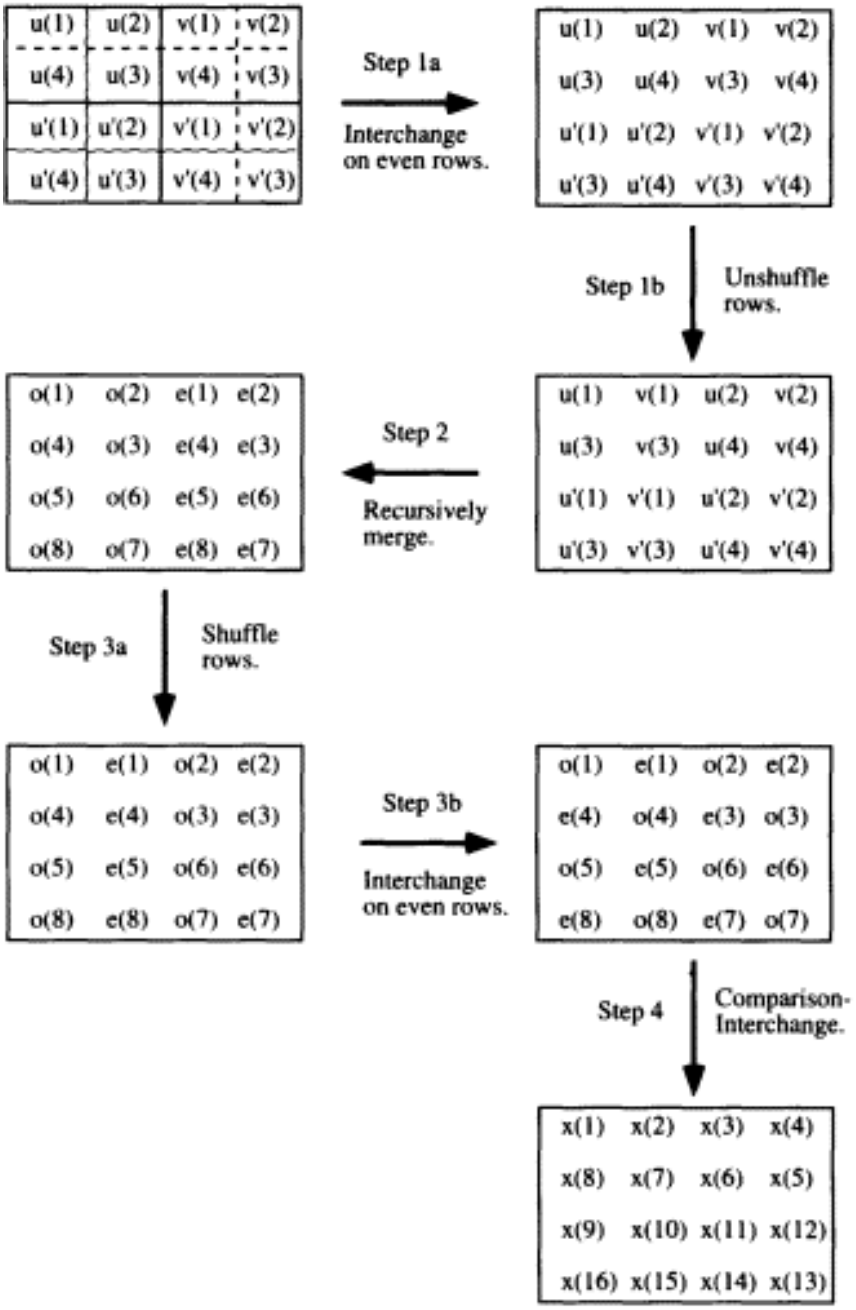



Figure 2.17:
Merging 4 arrays with odd-even merge on a mesh of size 16.



recursively merging each half of the mesh and shuffling the data back together, no element is more than $2s$ - 1 positions (in snake-like order) from its destination. Further, the first $2s$ - 1 steps of an odd-even transposition sort (in snake-like order) will suffice to order these misplaced elements in the resulting array.

Let $T(j, k, s)$ be the time required by the algorithm $M(j, k, s)$. The base case of the recursion is the column-based (i.e., 1-dimensional) sorting routine $M(j, 2, s)$, which has running time $T(j, 2, s) = O(j)$. For $k > 2$, $T(j, k, s) = O(k + s) + T(j, k/2, )$, where $O(k + s)$ is the time required for the shuffle, unshuffle, and comparison-interchange steps, and $T(j, k/2, )$ is the time required for the recursive call. Therefore, $T(j, k, s) = O(j + k + s\log k + s)$.

To use the merge algorithm to sort, let $s = 2$ and define the time to sort a mesh of size $n$ to be $S(n, n) = S(n/2, n/2) + T(n, n, 2)$, which is $O(n)$. Since data may be initially distributed so that the $\Omega(n)$ lower bound of Section 2.4 holds, the $\Theta(n)$ time algorithm just presented to sort $n^2$ items, distributed one item per processor on a mesh of size $n$, is optimal. ·

**Corollary 2.10** *Given $n^2$ pieces of data, distributed one piece per processor on a mesh of size $n^2$, and some $O(n)$ time computable index function $g$ that assigns to each processor a unique index, where $g$ is known to all processors, the data can be sorted according to this index function in $\Theta(n)$ time.*

*Proof*. Sort the data into snake-like order by the algorithm of Theorem 2.9, so that each processor $P_i$ contains the $i^{th}$ element of the data set in a variable called y. Each processor $P_i$ computes in $O(n)$ time the value $x = g(i)$, and creates a record $(x, y)$. Now, sort these records with $x$ as the key by the algorithm of Theorem 2.9. ·

### *2.6.2 Rotating Data within Intervals*

Suppose each processor on a mesh of size $n^2$ contains a record consisting of a key and data, and suppose that all processors with the same key reside in a contiguous sequence of processors (i.e., form an *ordered interval*) with respect to the snake-like ordering. If it is known that there are no more than $D$ processors in any one interval, then in $\Theta(D)$ time all processors can view a piece of data from all other processors in its interval. The data is rotated within each interval as follows. First, using snake-like indexing, each processor $P_i$ checks the keys of its neighbors



$P_{i-1}$ and $P_{i+1}$, assuming they exist, to determine if it is the first or last processor in its interval. Then the data is rotated similar to a row rotation, with each processor passing data to adjacent processors, where the first processor with a given key acts just as the westernmost processor of a row, and the last processor with a given key acts just as the easternmost processor of a row. Notice that the data may traverse more than a single row. This operation is most useful in situations where it is known a priori that $D = O(n)$. For situations where $D$ is large and the data rotation is used to compute an associative binary operation within intervals, it will be more efficient to use one of the algorithms defined later in this section.

### *2.6.3 Semigroup Computation within Intervals*

Suppose each processor on a mesh of size $n^2$ contains a record consisting of a key and data, and suppose that all processors with the same key form an ordered interval in the snake-like ordering. Furthermore, suppose that every processor needs to know the result of applying a semigroup operation (i.e., an associative binary operation such as minimum, summation, or parity) over the pieces of data in its ordered interval. If data is rotated in snake-like order within each interval, as described in the previous section, the algorithm has a worst-case running time of $\Theta(n^2)$. In order to compute a semigroup operation within ordered intervals in $\Theta(n)$ time, the following algorithm can be performed (snake-like indexing of the processors is assumed).

1. For every processor $P_i$, examine the key field of the data stored in processors $P_{i-1}$ and $P_{i+1}$. If processor $P_i$ is a westernmost or easternmost processor in its row, and processor $P_{i-1}$ stores the same key, then processor $P_i$ is marked as a *pre-connecting processor*. If processor $P_i$ is a westernmost or easternmost processor in its row, and processor $P_{i+1}$ stores the same key, then processor $P_i$ is marked as a *post-connecting processor*.

2. Perform a row rotation so that every processor knows whether or not

   (a) its ordered interval is completely contained in its row,

   (b) its ordered interval continues onto the next row (i.e., there is a post-connecting processor in its row with the same label), or



   (c) its ordered interval was continued from the previous row (i.e., there is a pre-connecting processor in its row with the same label).

3. Perform a row rotation so that every processor knows the result of computing the semigroup operation over all data with the same key in its row (i.e., in its row restricted ordered interval).

4. Every pre-connecting processor (post-connecting processor) $P_i$ sends its restricted row result to processor $P_{i-1}$ ($P_{i+1}$) if processor $P_i$ is in a row for which there is not a post-connecting processor (pre-connecting processor) with its key.

5. Perform a row rotation so that the values just transmitted are absorbed into the semigroup computation of every processor in the row-restricted ordered interval of the processor(s) receiving data.

6. Perform a column rotation, where every processor in a row with both pre- and post-connecting processors for its key, obtains and absorbs the running values from processors of a similar nature with its key. (This step serves to combine results among neighboring complete rows storing the same key.)

7. Every pre-connecting processor (post-connecting processor) *Pi*, sends the final result for its key to processor $P_{i-1}$ ($P_{i+1}$) if processor $P_{i-1}$ ($P_{i+1}$) is in a row for which there are not both pre- and post-connecting processors for its key.

8. Perform a final row rotation to distribute the result in row restricted ordered intervals for which either a pre- or post-connecting processor exists, but not both.

Since the semigroup computation within intervals is completed after a fixed number of row and column rotations, the time of the algorithm is $\Theta(n)$, which is optimal. It should be noted that several of the row operations could be combined, but this would only affect the running time by a multiplicative constant.

### *2.6.4 Concurrent Read and Concurrent Write*

Two other common data movement operations for the mesh are *concurrent read* and *concurrent write*, also known as *random access read* and *random access write*, respectively. These operations were described in Section 1.5 on page 22.

Concurrent read and concurrent write are used to allow the mesh to simulate the concurrent read and concurrent write capabilities of a *Concurrent Read, Concurrent Write Parallel Random Access Machine* (*CRCW PRAM*), where multiple processors are permitted to simultaneously read a value associated with a given key (concurrent read), and multiple processors are permitted to simultaneously attempt to update the value associated with a given key (concurrent write). In the case of multiple writes, only one processor succeeds, according to some tie-breaking scheme such as minimum data value.

Algorithms for restricted versions of concurrent read and concurrent write were presented in [NaSa81]. In this section, significantly different algorithms are given for more general versions of these operations. In order to maintain consistency during concurrent read and concurrent write operations, it is assumed that there is at most one *master record* per key, where every processor maintains no more than some fixed number of master records. In a concurrent read, every processor generates no more than some fixed number of request records, where each *request record* specifies a key that is used to identify the master record that the processor wishes to receive information about. In a concurrent write, every processor generates no more than some fixed number of update records, where each *update record* includes the key, field specification, and data corresponding to the master record it wishes to update. It should be noted that for many applications, a processor will maintain master records and also generate request or update records. A detailed description of a concurrent read is given for the mesh followed by a detained mesh description of a concurrent write.

**Concurrent Read**

In a concurrent read, every processor creates no more than some fixed number of master records, where each master record consists of a key, the data associated with the key, and some bookkeeping information. In a concurrent read, the purpose of the master records is to make the data associated with every unique key available to any processor that might want to read it. Every processor also creates no more than some fixed number of request records, each of which specifies the key that is associated with the data it wishes to receive. Unlike the master records, multiple request records can specify the same key. If a processor generates a request record for which there is no master record, then at the end of the operation it will receive a null message corresponding to that request. An implementation of a concurrent read in terms of fundamental data movement operations on a mesh of size $n^2$ follows.



1. All processors create the same fixed number, call it $M$, of master records corresponding to those keys for which the processor is responsible. Each record contains the key and associated data, as well as the ID (i.e., snake-like index) of the processor creating the record. Some or all of the master records created by a processor may be 'dummy' records, so as to balance the number of items per processor in subsequent sort steps. Notice that since there is only one master record maintained for each key, a key value may be represented by at most one master record somewhere in the mesh.

2. All processors create the same fixed number, call it $R$, of request records, which contain the desired key, as well as the ID of the processor creating the record. Some or all request records created by each processor may be 'dummy' records, so as to balance the number of items per processor in subsequent sort steps.

3. Sort all $(M + R) * n^2$ master and request records together into snake-like order by key, where ties are broken in favor of master records, and ties between request records are broken arbitrarily.

4. By performing a broadcast operation within ordered intervals with respect to keys, every request record will receive a copy of the data it desires.

5. Sort the $(M + R) * n^2$ master and request records by the snake-like index of the processor that originally created them (i.e., by the ID field) so that they are returned to the initiating processors. Notice that the master records are not conceptually needed for the sort step, but are used so as to balance the number of items in each processor during this step. The master records are discarded after the sort is complete.

The concurrent read is accomplished through a fixed number of sort and interval operations, and for fixed constants $R$ and $M$ is completed in $\Theta(n)$ time on a mesh of size $n^2$. (Notice that throughout most of the algorithm, the mesh of size $n^2$ simulates a desired mesh of size $(M + R) * n^2$.)

**Concurrent Write**

In a concurrent write, every processor creates no more than some fixed number of master records, consisting of a key and some bookkeeping information, for each of the master entries that it maintains and is willing



to receive an updated value for. At the end of the concurrent write, a processor will receive a record corresponding to each of the master records it created, indicating the new value of a data item(s) to be associated with that key. Each processor creates no more than some fixed number of update records, each consisting of a key, a data value, and some bookkeeping information. If two or more update records contain the same key, then the master record associated with that key, if one exists, will receive the minimum such update data value. (In other circumstances, one could replace minimum with some other tie-breaking mechanism, as discussed below.) An implementation of a concurrent write in terms of fundamental data movement operations on a mesh of size $n^2$ follows.

1. All processors create the same fixed number, call it M, of master records, corresponding to those master entries for which the processor is willing to receive data. Each master record contains a key and data, as well as the ID (i.e., snake-like index) of the processor creating the record. Some or all of the master records created by a processor may be 'dummy' records, so as to balance the number of items per processor in subsequent sort steps.

2. All processors create the same fixed number, call it $U$, of update records, which contain a key, field specifier, and data, as well as the ID of the processor creating the record. Some or all update records created by a processor may be 'dummy' records, so as to balance the number of items per processor in subsequent sort steps.

3. Sort all $U * n^2$ update records by key into snake-like order, breaking ties of the same key arbitrarily.

4. Apply the concurrent write tie-breaking mechanism within the ordered intervals. This should be a mechanism computable in $O(n)$ time, such as one that can be computed by performing a semigroup operation within ordered intervals. While the tie-breaker most often needed in this book is the minimum, other possibilities are average, product, median, mode, or choosing any value. In each ordered interval, replace the first data item with this new value. This becomes the *representative* for the key, and the record maintaining this information will be called the *representative update record*.

5. Sort all $(M + U) * n^2$ master and update records together by key, where ties are broken in favor of master records, and ties between



update records are broken in favor of the representative update record.

6. All master records obtain their updated value from their representative update record, which is stored, if it exists, in the succeeding processor (in snake-like order).

7. Sort all $(M + U) * n^2$ records by the snake-like index of the processor that originally created them (i.e., by the ID field) so that they are returned to the initiating processors. Notice that the update records are not conceptually needed for the sort step, but are used so as to balance the number of items in each processor during this step. The update records are discarded after the sort is complete.

Like the concurrent read, the concurrent write is accomplished through a fixed number of sort and interval operations, and is completed in $\Theta(n)$ time on a mesh of size $n^2$.

### *2.6.5 Compression*

The worst-case communication time for $k$ elements distributed arbitrarily over a mesh of size $n^2$ is $\Theta(n)$. However, if these $k$ elements can be placed in a submesh (square) of size $4^{\lceil \log_4 k \rceil}$, then this bound can be reduced to $\Theta(k^{1/2})$. This can be accomplished as follows. Sort the $k$ data elements into snake-like order, where processors that do not contain one of these elements create data with a key of $\infty$. After performing a sort, report, and broadcast, each processor that contains one of the $k$ pieces of data knows its position in the order (i.e., its rank). Further, every processor knows the total number $k$ and the value $4^{\lceil \log_4 k \rceil}$. Using a concurrent write based on the snake-like index of the processor, each processor now determines which processor in the submesh to send its data to, and all processors in the submesh indicate their willingness to receive. Therefore, the time required to place $k$ pieces of data arbitrarily distributed over a mesh of size $n^2$ into a submesh where their worst-case communication will be minimized is $\Theta(n)$.

## 2.7 Further Remarks

In this chapter, several fundamental mesh algorithms have been presented. These algorithms include fundamental data movement operations, such as row and column rotations, sorting, concurrent read, concurrent write, and data compression. Algorithms were also presented for

solving fundamental problems such as computing semigroup (i.e., associative binary) operations, matrix transposition, matrix multiplication, and computing the transitive closure of a matrix. All of the algorithms run in optimal $\Theta(n)$ time on a mesh of size $n^2$. It should be noted that if an input of size $m^2$ is initially stored in a submesh of size $m^2$ on a mesh of size $n^2$, $m \leq n$, then the algorithms can be modified to run in $\Theta(m)$ time. Finally, it should be noted that a mesh automaton of size $n^2$ can also perform some of these algorithms, such as transitive closure, matrix transpose, and modular matrix multiplication, in $\Theta(n)$ time.



# 3
# Mesh Algorithms for Images and Graphs

## 3.1
## Introduction

The mesh computer is a natural architecture for solving problems that involve matrices and digitized pictures. This is due to the fact that, in either case, adjacent input elements can be mapped in a natural fashion to the same or neighboring processors of the mesh. Given an $n \times n$ adjacency or weight matrix representing a graph $G$, with $n$ vertices mapped in a natural fashion onto a mesh of size $n^2$, Section 3.2 presents asymptotically optimal $\Theta(n)$ time mesh solutions to fundamental problems from graph theory. These problems include marking a breadth-first spanning forest of $G$, determining whether or not $G$ is bipartite, marking the articulation points and bridge edges of $G$, and marking a minimum-weight spanning forest of $G$. These algorithms are described predominantly in terms of fundamental mesh algorithms and data movement operations, which were presented in Chapter 2. Several of these graph theoretic results are used to solve image problems that are presented later in the chapter. However, since the majority of the image algorithms presented in this chapter do not rely on graph algorithms, the reader interested primarily in image algorithms may wish to move directly to Section 3.3, referring back to the results of Section 3.2 when necessary.

Due to the natural mapping of images to the mesh, local operations on images, such as edge detection or median filtering, can be performed by local operations on the mesh, enabling such algorithms to exploit efficiently the massive parallelism available. Many discussions of actual image-processing meshes, such as SOLOMON, ILLIAC III, CLIP4, or MPP, emphasize their speed on local operations [DaLe81, HwFu82, Reev84, Rose83, Pott85], and most of the early papers on meshes, such as those of Unger [Unge58, Unge59, Unge62] and Golay [Gola69], similarly emphasized local operations.

Starting with Section 3.3, the remainder of the chapter is devoted to higher level image processing and pattern recognition tasks that require combining information globally, including geometric problems involving connectivity, convexity, internal distance, and external distance. For

some problems, such as computing the Euler number [Gray71, MiPa69], counting figures [Beye69, Levi72], and skeletization [StRo71], it is possible to iterate local operations to achieve an optimal solution to a global problem. In Section 3.3.1, the Beyer-Levialdi [Beye69, Levi72] "shrinking" approach, which is based solely on local operations, is used to give an asymptotically optimal algorithm for counting the number of *figures* (i.e., connected black components) present in a digitized black/white picture. This approach of using local operations to obtain efficient mesh solutions seems to work only for isolated problems. Instead of local operations, most of the remaining image algorithms in this chapter emphasize the use of data movement operations and graph-theoretic algorithms.

In Section 3.3.2, a worst-case optimal $\Theta(n)$ time component labeling algorithm is presented in terms of fundamental data movement operations and graph-theoretic algorithms. Section 3.3.2 also shows that an algorithm to solve the component labeling problem can be constructed by local operations, but requires $O(n^2)$ time.

In Section 3.4, optimal $\Theta(n)$ time algorithms are given for computing internal distances, marking minimal internal paths, and counting the number of these paths for every figure in the image. In Section 3.5, an optimal $\Theta(n)$ time algorithm is given for marking the extreme points of the convex hull for every labeled set of processors. Optimal $\Theta(n)$ time algorithms are also given for deciding if the convex hull of each figure contains pixels that are not members of the figure, for deciding if two sets of processors are linearly separable, for solving the smallest box problem, and for deciding if each black figure is convex.

In Section 3.6, an optimal $\Theta(n)$ time algorithm is given to compute the distance between figures and the external diameter of each figure, where the distance can be measured by almost any metric. Section 3.6 also contains optimal solutions to nearest neighbor, radius query, and farthest point problems.

### 3.2 Fundamental Graph Algorithms

In this section, optimal mesh algorithms from [AtKo84] are presented to solve some fundamental graph theoretic problems. The reader is referred to [AtKo84] for proofs of correctness and any omitted details of the algorithms. Given a graph $G$ represented as an adjacency or weight matrix, algorithms are presented to determine all bridge edges and articulation points of $G$, to determine whether or not $G$ is bipartite, and to find a minimum-weight spanning forest of $G$. Additional optimal



algorithms, solving problems such as finding the length of a shortest cycle and determining the cyclic index of a graph, appear in [AtKo84].

Some graph-theoretic definitions are necessary. For simplicity, a *graph* $G = (V, E)$ consists of a finite nonempty set of *vertices* $V = \{1, \ldots, |V|\}$, and a set of *edges* $E \subseteq V \times V$. If the edges are ordered pairs $(i, j)$ of vertices, then the graph is said to be *directed*, while if the edges are unordered pairs $\{i, j\}$ of distinct vertices, then the graph is said to be *undirected*. The graph is said to be *weighted* if there is a realvalued weight $w(i, j)$ for each edge $(i, j)$, where $w(i, j) = \infty$ if there is no edge from $i$ to $j$. (If the graph is unordered, then $w(i, j) = w(j, i)$.) An *i - j walk* in $G$ is a finite sequence of vertices $i = v_0 v_1 \ldots v_l = j$ such that $(V_m, V_{m+1}) \in E$ for every $m \in \{0, 1, \ldots, l - 1\}$. Vertices $i$ and $j$ are the *endpoints* (or *end vertices*) of that walk, and the walk is said to be *from i to j*. The *length* of that walk is 1, and $v_1, v_2, \ldots, v_{l-1}$ are its *intermediate vertices*. If all intermediate vertices of a walk are distinct, then the walk is a *path*. A path of positive length from a vertex to itself is called a *cycle*. A graph that contains no cycles is termed *acyclic*. An *i - j/k path* (*walk*) is an *i- j* path (walk) in which all intermediate vertices belong to the set $\{1, 2, \ldots, k\}$. An *i - j/0* path is defined to be an *i - j* path with no intermediate vertices.

A spanning tree $T$ of an undirected graph $G$ is *breadth first* with respect to a vertex $r$ if every $r - j$ path in $T$ is also a shortest, or *minimum distance*, $r - j$ path in $G$. Such a spanning tree can be assigned directions away from the root to result in a directed tree $T'$ rooted at $r$, where $T'$ is referred to as a *directed breadth-first* (*spanning*) *tree* of the undirected graph $G$.

$G_*$ is used to denote the *transitive closure graph* of a graph $G$, where an edge $(i, j)$ is in $G_*$ if and only if there is a (possibly degenerate) $i - j$ path in $G$. ($G_*$ is often referred to as the *reflexive transitive closure*, since it includes all edges of the form $(i, i)$, whether or not such a nondegenerate path exists in $G$.) Given a directed or undirected graph $G$, define $R_G(i)$, the vertices in $G$ *reachable* from $i$, to be $R_G(i) = \{j |$ there is an $i - j$ path in $G\}$.

If $(i, j)$ is an edge of $G$, then $G - \{(i, j)\}$ denotes the graph obtained by removing edge $(i, j)$ from $G$, and if $v$ is a vertex of $G$, then $G - \{v\}$ denotes the graph obtained by removing vertex $v$ and all edges incident on it from $G$.

Many of the problems given in this section consider simple graphs. A *simple* graph is one in which there are no self-loops (i.e., edges from a vertex to itself). Note that the definition of graph presented on page 91 precludes parallel edges (i.e., multiple edges between the same pair of



vertices).

Given a graph $G = (V, E)$, $|V| = n$, let $S_k(i, j)$ denote the length of a shortest $i- j/k$ path, for $i, j \in \{1, 2, \ldots, n\}$ and $k$ e $\{0,1, \ldots, n\}$. If there is no $i - j/k$ path, then define $S_k(i, j)$ to be $\infty$. The initial values of $S$ are given by

$$S_0(i, j) = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{if } i = j, \\ \infty & \text{otherwise.} \end{cases}$$

The first problem of this section is concerned with determining the length of a shortest path (i.e., the minimum distance) between every pair of vertices. For the problems considered in this section, processor $P_{i,j}$ initially contains the $(i, j)$ entry of the adjacency or weight matrix. That is, it is assumed that the matrix is *stored in a natural fashion*. If the end result of a given problem is to determine a relationship between vertices $i$ and $j$, then when the algorithm terminates, processor $P_{i,j}$, will know this relationship.

**Theorem 3.1** *Given the adjacency matrix of an undirected graph G mapped in a natural fashion onto a mesh of size* $n^2$, *in* $\Theta(n)$ *time the minimum distance between every pair of vertices can be determined.*

*Proof.* It is easy to show that

$$S_{k+1}(i, j) = min\{S_k(i, j), S_k(i, k + 1) + S_k(k + 1, j)\},$$

which is the form required in order to apply the generalized transitive closure algorithm of Section 2.5.3. Therefore, in $\Theta(n)$ time all $S_n(i, j)$ can be computed on a mesh of size $n^2$ so that processor $P_{i,j}$ contains the value $S_n(i, j)$.

The minimum distance between all pairs of vertices of an undirected simple graph $G$ can be used to mark a breadth-first spanning forest (i.e., a breadth-first spanning tree within every connected component) of $G$ in $\Theta(n)$ time on a mesh of size $n^2$. This can be done by arbitrarily choosing a root vertex in each connected component and then using the generalized transitive closure algorithm to determine for every vertex i) the minimum distance to its root and ii) its parent in the breadth-first spanning tree of its component. The details, including how to create the adjacency matrix corresponding to the breadth-first spanning forest, follow.



**Theorem 3.2** *Given the adjacency matrix of an undirected simple graph* $G = (V, E)$ *mapped in a natural fashion onto a mesh of size* $n^2$, *in* $\Theta(n)$ *time a directed breadth-first spanning forest* $T = (V, A)$ *can be created. As a byproduct, the undirected breadth-first spanning forest edge set* $E_A$ *can also be created, where* $E_A$ *consists of the edges of A and the edges of A directed in the opposite direction.*

*Proof.* Compute $S_n(i, j)$, for all $i, j \in V$, by the algorithm of Theorem 3.1. Simultaneously in every row $i$, perform a row rotation so that each processor in row $i$ determines the index $r(i) = j$ of the first non-$\infty$ entry $S_n(i, j)$. Vertex $r(i)$ will be the root of the spanning tree containing vertex $i$. (Note that $r(i)$ can also be used as a unique component label for the connected component containing vertex $i$.) Since $S_n(i, r(i))$ is the distance from vertex $i$ to vertex $r(i)$, and since vertex $r(i)$ is the root of the spanning tree containing vertex $i$, then $S_n(i, r(i))$ is the level of vertex $i$ in the spanning tree rooted at vertex $r(i)$. Let $level(i) = S_n(i, r(i))$. Simultaneously in every row $i$, perform a row rotation to broadcast $level(i)$. The result of this rotation is that every processor in row $i$ knows the level of vertex $i$ in its spanning tree. Next, perform a column rotation in every column $j$ to broadcast $level(j)$ from processor $P_{i,j}$ to all processors in column $j$. Now every processor $P_{i,j}$ knows $level(i)$ and $level(j)$.

For every $i \neq r(i)$, let the *parent* of vertex $i$, denoted $\boldsymbol{P}(i)$, be defined as

$$\boldsymbol{P}(i) = min\{j \mid (i, j) \in E \text{ and } level(i) = level(j) + 1\}.$$

The directed graph $T = (V, \mathrm{A})$, where $A$ consists of all directed edges $(\boldsymbol{P}(i), i)$, is a directed breadth-first forest. This forest is formed as follows. Simultaneously for all rows $i$, perform a row rotation so that all processors in row $i$ know $\boldsymbol{P}(i)$, the parent of vertex $i$. (This is accomplished by a row rotation where every processor $P_{i,k}$ sends $(k, level(k))$ to be viewed by all other processors in row $i$.) Next, simultaneously for all columns $j$, perform a column rotation to broadcast $\boldsymbol{P}(j)$, the parent of vertex $j$, from processor $Pj\ ,j$ to all processors in column $j$. Every processor $P_{i,j}$ now knows the value of $\boldsymbol{P}(i)$ and $\boldsymbol{P}(j)$. Finally, every processor $P_{\boldsymbol{P}(j),j}$ determines that $(\boldsymbol{P}(j),j) \in \mathrm{A}$ and that $(\boldsymbol{P}(j), j) \in E_A$, and every processor $P_{i,\boldsymbol{P}(i)}$ determines that $(i, \boldsymbol{P}(i)) \in E_A$. (Recall that $A$ is the set of edges in a directed breadth-first spanning forest and $E_A$ is the set of edges in the corresponding undirected breadth-first spanning forest.)

The transitive closure and rotations used to determine the level information each take $\Theta(n)$ time. The rotations used to determine parent



information each take $\Theta(n)$ time. The final step in creating $T$ requires $\Theta(1)$ time. Therefore, the running time of the algorithm is as claimed.

It is now shown that a breadth-first spanning forest may be used to determine whether or not an undirected simple graph is bipartite, where a *bipartite* graph $G = (V, E)$ is one in which all vertices of $V$ can be partitioned into two disjoint sets of vertices, say $V_1$ and $V_2$, where $V_1 \cup V_2 = V$, such that every edge in $E$ connects a vertex of $V_1$ with a vertex of $V_2$. The algorithm consists of creating a breadth-first spanning forest $T$ of $G$, and then using the property that $G$ is bipartite if and only if for every vertex $i$ in $G$, the level of vertex $i$ in spanning forest $T$ differs by 1 from the level of vertex $j$ in $T$, for every vertex $j$ such that $(i, j) \in E$. Notice that those vertices on odd levels of $T$ can be defined as $V_1$, while those vertices on even levels of $T$ can be defined as $V_2$.

**Corollary 3.3** *Given the adjacency matrix of an undirected simple graph $G = (V, E)$ mapped in a natural fashion onto a mesh of size $n^2$, in $\mathrm{O}(n)$ time it can be decided whether or not $G$ is bipartite.*

*Proof.* Using the algorithm associated with Theorem 3.2, mark a directed breadth-first spanning forest $T$ of $G$, and determine $level(i)$ for every vertex $i$ in $G$. Using the observation stated above, every processor $P_{i,j}$, where $(i, j) \in E$, sets its Boolean flag *bipartite* to *true* if $level(i)$ differs by 1 from $level(j)$, and to *false* otherwise. Every processor $P_{i,j}$, where $(i, j) \notin E$, sets *bipartite* to *true*. A simple semigroup (i.e., associative binary) operation can be used to compute the logical AND of these values (*bipartite*) in order to obtain the answer to the query. Since the algorithm from Theorem 3.2 and the semigroup operation both require $\Theta(n)$ time, the running time of the algorithm is as claimed.

### *3.2.1 Bridge Edges*

In this section, an optimal $\Theta(n)$ time mesh algorithm is presented to determine all bridge edges of an undirected simple graph $G = (V, E)$. A *bridge edge* is an edge $e \in E$ whose removal will increase the number of connected components in $G$.

The first step of the algorithm is to mark a directed spanning forest $T = (V, A)$, by using the algorithm associated with Theorem 3.2. Again, let $E_A$ consist of the edges of $A$ and the edges of $A$ directed in the opposite direction. Notice that the edges in $E - E_A$ cannot be bridge edges since each such edge is not needed for at least one spanning forest



of $G$, namely $T$. Therefore, only the edges $E_A$ of $T$ need to be tested. (The reader should notice that $T$ need not be a breadth-first spanning forest. Any spanning forest will suffice for this algorithm, including the minimum spanning forest obtained in Section 3.2.3.)

A result from [AtKo84] will be used to detect the bridge edges. Recall that $R_G(i)$ denotes the vertices in $G$ *reachable* from $i$, where $R_G(i) = \{j \mid$ there is an $i - j$ path in $G\}$, and that $P(i)$ denotes the parent of $i$ in the spanning forest, as defined in Theorem 3.2. The result states that if $i = P(j)$, then $(i, j) \in E_A$ is a bridge edge if and only if $R_T(j) = R_{\hat{G}}(j)$, where $\hat{G} = (V, \hat{E})$ is the directed graph whose set of directed edges $E$ is the union of $A$ and the set of directed edges obtained by replacing every edge of $E - E_A$ by its *two* oppositely directed edges. (I.e., $(i, j)$ is a bridge edge if and only if the set of vertices that may be reached from $j$ using the directed edges of the spanning forest is the same as the set of vertices that may be reached from $j$ using all edges of $G$ with the exception of $E_A - A$, the upward directed edges associated with the spanning forest. So, $(i, i)$ being a bridge edge means that the only way $j$ may reach a vertex of $G$ that is not one of its descendants in $T$ is by traversing the edge from $j$ to $i$, its parent in the spanning tree.) Notice that the adjacency matrix of $\hat{G}$ is created by logically OR-ing the adjacency matrix of $T$ with that of $G' = (V, E - E_A)$, which can be done in 0(1) time. Also notice that the transitive closure of $T$ and $G$ can be computed in $\Theta(n)$ time by the algorithm of Section 2.5.3, giving $R_T$ and $R_{\hat{G}}$, respectively.

In order to determine the bridge edges according to the reachability criteria, simultaneously for all $i : r(i)$ (i.e., for all vertices $i$ that are not the root vertex of their spanning tree), test whether or not the $i^{th}$ row of the adjacency matrix of $R_T$ is the same as the $i^{th}$ row of the adjacency matrix of $R_{\hat{G}}$. This can be accomplished by performing a row rotation, simultaneously for all rows, so that every processor knows the answer for its row. Finally, in $\Theta(n)$ time all diagonal processors $P_{i,i}$ with the answer 'yes' for their row inform processors $P_{\boldsymbol{P}(i),i}$ and $P_{i,\boldsymbol{P}(i)}$ that the edge $(\boldsymbol{P}(i), i)$ is a bridge edge of $G$ and should be marked as such. This can be accomplished by either performing a row rotation followed by a column rotation, or by performing a concurrent write.

Therefore, the following is obtained.

**Theorem 3.4** *Given the adjacency matrix of an undirected simple graph G mapped in the natural fashion onto a mesh of size* $n^2$, *in* $\Theta(n)$ *time all bridge edges of G can be marked.*



### *3.2.2 Articulation Points*

Given an undirected simple graph $G = (V, E)$, the problem of detecting the articulation points of $G$ is now examined, where an *articulation point* is a vertex $v \in V$, whose removal, along with incident edges, increases the number of connected components of $G$. The algorithm presented in this section will 'mark' all processors $P_{i, r(i)}$ for which vertex $i$ is an articulation point, where vertex $r(i)$ is the root of the spanning tree containing vertex $i$. In addition to the previously defined graphs $T$ and $\hat{G}$, the algorithm requires an undirected graph $H = (V, E')$, which is defined as $(i, j) \in E'$ if and only if $\boldsymbol{P}(i) = \boldsymbol{P}(j)$ and there is an edge of $G$ between $R_T(i)$ and $R_T(j)$. Intuitively, the edges in $H$ represent pairs of vertices of $G$ that have the same parent in $T$ and have at least one pair of descendants in $T$ that are connected by an edge of $G$.

The algorithm will be based on examining three sets of vertices of $T$, namely, the roots of trees in $T$, the leaves of $T$, and the remaining interior vertices of $T$. It is easy to see that the removal of a root vertex $r$, along with all incident edges, will not disconnect its component if and only if there is a path between every pair of its children that does not include $r$. It is also easy to see that a leaf of $T$ cannot be an articulation point. A nonleaf vertex $v \neq r(v)$ of $T$ is not an articulation point if and only if for each of its children there is a path in $G - \{v\}$ from that child to outside the subtree rooted at $v$. Given $s$ a child of $v$ in $T$, such a path surely exists if there is an edge in $E - E_A$ between some vertex in $R_T(S)$ and a vertex not in $R_T(V)$. Define a vertex $w$ as *special* if there is an edge between a vertex in $R_T(W)$ and a vertex not in $R_T(\boldsymbol{P}(w))$. For

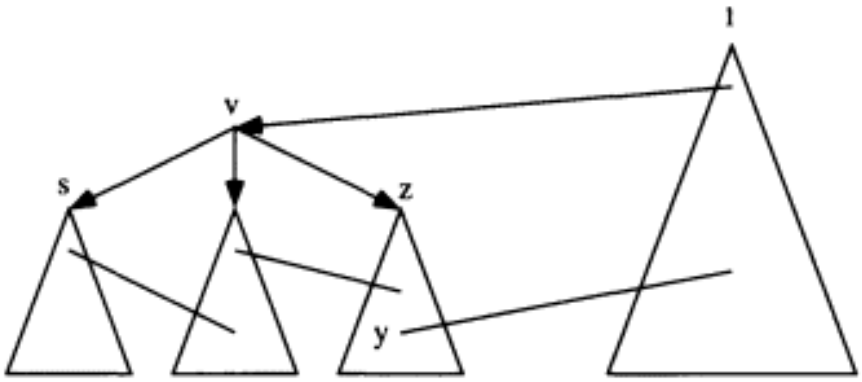



Figure 3.1:
$z$ is a *special* vertex, while $s$ is not.



example, in Figure 3.1 vertex $z$ is special while vertex $s$ is not.

Lemma 3.5 summarizes the three cases for deciding whether or not a given vertex is an articulation point of $G$.

**Lemma 3.5** *Given a spanning forest $T$ of an undirected simple graph $G = (V, E)$, the following hold.*

1. A root of a tree in $T$ is not an articulation point of $G$ if and only if for every pair of its children in $T$, say $i$ and $j$, there is an $i$ - $j$ path using edges of $G$ with the exception of $(i, \boldsymbol{P}(i))$ and $(j, \boldsymbol{P}(j))$.

2. *No leaf of $T$ is an articulation point of $G$.*

3. A nonleaf, nonroot, vertex $v$ is not an articulation point if and only if for each of its children there is a path in $G - \{v\}$ from that child to a vertex outside the subtree rooted at $v$.

Recall that the edges in $H$ represent pairs of vertices of $G$ that have the same parent in $T$ and have at least one pair of descendants in $T$ that are connected by an edge of $G$. Then it can be shown that an interior vertex of $T$ is not an articulation point if and only if for every one of its children there is a path in $H$ from that child to at least one special vertex. Lemmas 3.6 and 3.7 formally state these sufficient conditions for deciding whether or not an interior vertex of $T$ is an articulation point of $G$.

**Lemma 3.6** *Suppose* $s \in V$, $s \neq r(s)$. *Then there is a path in* $G - \{\boldsymbol{P}(s)\}$ *from* $s$ *to outside the subtree of* $\boldsymbol{P}(s)$ *if and only if there is a path in* $H$ *from* $s$ *to at least one special vertex (possibly* $s$ *itself).*

**Lemma 3.7** *A vertex* $v \neq r(v)$ *that is not a leaf is not an articulation point if and only if for every one of its children there is a path in* $H$ *from that child to at least one special vertex.*

Therefore, following the previous discussion, efficient algorithms are needed to construct $H$ and detect special vertices. Fortunately, both of these can be determined in $\Theta(n)$ time on a mesh a size $n^2$. The algorithms for creating $H$ and detecting special vertices rely on using the previously defined matrices $T$ and $G$, being able to efficiently compute the transitive closure of a matrix, being able to efficiently multiply



matrices, and being able to use several efficient fundamental data movement operations that have been previously defined. The details of an optimal $\Theta(n)$ time algorithm for constructing $H$ and marking special vertices follow.

**Lemma 3.8** *In* $\Theta(n)$ *time,* $H = (V, E')$ *can be created and all processors in row* $i$ *and column* $i$ *can know whether or not vertex* $i$ *is special.*

*Algorithm:* In $\Theta(n)$ time create $T$, $T^*$, and $\hat{G}$. The adjacency matrix for a new graph $Z = (V, X)$, where $(i, j) \in X$ if and only if there is an $i$ - $k$ path in $T$ and an edge $(k, j) \in E - E^A$ for some vertex $k$, can be created in $\Theta(n)$ time as the logical multiplication of matrices $T^*$ and $G'$ $(V, E - E_A)$.

Using a column rotation, every processor $P_{x,y}$ sends the values of $T^*(x, y)$ and $\boldsymbol{P}(x)$ to all of the processors in column y. When the contents of $P_{x,y}$ reaches processor $P_{i,y}$, processor $P_{i,y}$ checks to see whether

1. $\boldsymbol{P}(i) = \boldsymbol{P}(x)$,

2. $(x, y)$ is a directed edge of $T^*$, and

3. $(i, y)$ is a directed edge of $Z$.

If all three conditions are satisfied, then $i$ and $x$ are siblings in $T$, y is a descendant of $x$ in $T$ (i.e., $y \in R_T(X)$), and $(a, y) \in E - E_A$ for some $a \in R_T(i)$. Hence, $P_{i,y}$ can decide that $H(i, x) = 1$. It should be noted that for a given $P_{i,y}$, these three conditions are simultaneously satisfied at most once during the column rotation, and that if $H(i, x) = 1$, then there must exist a y such that the three conditions hold simultaneously. After the rotation is complete, a concurrent write or row rotation is used to create $H$.

In order to let every processor $P_{i,j}$ know whether or not $i$ and $j$ are special, the above column rotation is modified so that processor $P_{i,y}$ checks to see if

1. $x = P(i)$,

2. $(x, y)$ is not a directed edge of $T^*$, and

3. $(i, y)$ is a directed edge of $Z$.

If these conditions are satisfied, then $P_{i,y}$ notes that vertex $i$ is special. A final row and column rotation send the required information to all processors. Since the row and column rotations take $\Theta(n)$ time, as does the optional concurrent write, the algorithm finishes in $\Theta(n)$ time.



At this point, it is possible to describe a straightforward algorithm to determine all articulation points of an undirected simple graph $G$ in optimal time on a mesh. The algorithm follows the discussion in this section of examining three general cases of vertices in $G$ with respect to $T$, corresponding to the roots of $T$, the leaves of $T$, and the interior vertices of $T$.

**Theorem 3.9** *Given the adjacency matrix of an undirected graph G mapped in the natural fashion onto a mesh of size* $n^2$, *in* $\Theta(n)$ *time all articulation points of G can be identified.*

*Algorithm:* Create $T$, $T^*$, $H$, and $H^*$. As a byproduct of this, every processor in row $i$ and column $i$ will know whether or not vertex $i$ is special. Each of these processors will also know $\boldsymbol{P}(i)$, the parent of vertex $i$ in $T$, and $r(i)$, the root vertex of the spanning tree in $T$ containing vertex $i$. Every processor $P_{i,j}$ can check to see if it can decide that vertex $r(i)$ is an articulation point by testing to see if $\boldsymbol{P}(i) = \boldsymbol{P}(j) = r(i)$ and $H^*(i, j) = 0$. Either a concurrent write or a semigroup (i.e., associative binary) operation can be used to inform processor $P_{r(i),r(i)}$ as to whether or not vertex $r(i)$ is an articulation point.

Next, using row rotations, every processor $P_{i,r(i)}$ checks to see whether or not vertex $i$ is a leaf in $T$. If the answer is affirmative, then $P_{i,r(i)}$ decides that vertex $i$ is not an articulation point. Finally, using row rotations, every processor $P_{i,r(i)}$ for which $P(i) \neq r(i)$ checks to see whether or not there exists a vertex $k$ that is special, such that $H^*(i, k) = 1$. If the answer is negative, then $P_{i,r(i)}$ creates a message to inform processor $P_{\boldsymbol{P}(i),r(i)}$ that vertex $P(i)$ is an articulation point. These messages are routed using column rotations.

Creating $T$, $T^*$, $H$, and $H^*$ each take $\Theta(n)$ time, as described previously. The rotations each take $\Theta(n)$ time. Therefore, the running time of the algorithm is as claimed. ·

### *3.2.3 Minimum Spanning Tree*

Given a weighted undirected graph $G = (V, E)$, with weight $w(i, j)$ associated with edge $(i, i) \in E$, this section considers the problem of determining a minimum-weight spanning forest $T = (V, E_T)$ *of* $G$. The *weight of a spanning forest* is the sum of the weights of the edges in the forest, and a *minimum-weight spanning forest* (*minimum spanning forest*) *of* $G$ is a spanning forest of $G$ with minimal weight. (While the weight of a minimum-weight spanning forest is unique, a minimum spanning forest

need not be unique.) It is well known for a variety of parallel models of computation that efficient algorithms to determine minimum-weight spanning forests are similar to efficient component labeling algorithms for the same parallel model and form of input [CLC82, HaSi81, SaJa81]. The minimum spanning forest algorithm presented in this section follows the general component labeling approach given in [HCW79].

$T$ is constructed through a number of stages, where at each stage *clubs* that represent subtrees of $T$ are combined by adding minimum-weight edges between them. Each club has a label, which is the minimum label of any vertex in the club. Initially, each vertex of $G$ is its own club, and the set of edges of $T$, denoted $E_T$, is empty. During each stage of the algorithm, for each club of $T$, a minimum-weight edge of $G$ joining that club with a different club of $T$ is chosen, with ties broken in favor of the club of smallest label. The set of edges just chosen is added to $E_T$, and clubs are combined that are connected by these edges. The process is repeated until only one club remains for each connected component. It will be clear that the terminal condition is reached because no club will have any edges to any other club.

At most $\lfloor \log_2 n \rfloor$ stages are required to reduce the initial clubs, representing the $n$ vertices, to the final clubs, since the number of unfinished clubs is reduced by at least a factor of 2 during each stage of the algorithm. After each stage of the algorithm, all edges of $G$ consisting of endpoints that are in the same club of $T$ may be discarded. Further, if there is more than one edge between two clubs, all but one minimum-weight edge between the clubs may be discarded. These observations form the heart of the algorithm associated with the following theorem.

**Theorem 3.10** *Given the weight matrix of an undirected simple graph $G = (V, E)$ mapped in a natural fashion onto a mesh of size $n^2$, in $\Theta(n)$ time a minimum-weight spanning forest $T = (V, E_T)$, can be determined.*

*Algorithm:* The algorithm is based on being able to efficiently collapse the vertices of $G$ that belong to the same club of $T$ into a single vertex, remove all loops from the resulting graph, and keep only a minimum-weight edge between any pair of new vertices. Let $G_t$ denote the "collapsed" version of $G$ right after the $t^{th}$ stage of the algorithm. Stage 0 of the algorithm is defined by setting $G_0 = G = (V, E)$ and $T = (V, \emptyset)$. The $t^{th}$ stage of the algorithm is defined by the following steps.

1. For every vertex $v$ of $G_{t-1}$, choose a minimum-weight edge $(v, x)$, with ties broken in favor of the smallest $x$. Let $H_t$ be the set of chosen edges.



2. Add to $E_T$ the edges of $G$ that are represented by the edges in $H_t$.

3. $G_t$ is obtained by "collapsing" the vertices of $G_{t-1}$ that are in the same club (i.e., connected component) with respect to $H_t$. A club is represented as a vertex that inherits as its label the minimum label of any vertex in the club. This "collapsed" version of $G_{t-1}$ might have loops and parallel edges. All loops are discarded and only one minimum-weight edge between any pair of clubs (vertices) is kept, with ties broken arbitrarily. Any club without edges to any other club is removed. The resulting graph is $G_t$.

Notice that just prior to the $t^{th}$ stage of the algorithm, $G_{t-1}$ has as many vertices as $T$ has unfinished clubs. Since the number of unfinished clubs is at least halved after each iteration of the algorithm, $G_{t-1}$ has no more than $\frac{n}{2^{t-1}}$ vertices. Assume that the weight matrix of $G_{t-1}$ is compressed to the upper-left $m \times m$ corner of the mesh, where $m \leq \frac{n}{2^{t-1}}$, and that the $t^{th}$ stage of the algorithm requires $\Theta(m)$ time. Then the running time, $T(n)$, of the entire algorithm can be expressed by the recurrence $T(n) = \Theta(n) + T(n/2)$, which is $\Theta(n)$.

It only remains to show that the $t^{th}$ stage of the algorithm can be completed in $\Theta(m) = \Theta(\frac{n}{2^{t-1}})$ time. At the beginning of stage $t$, the weight matrix of $G_{t-1}$ is stored in the upper-left $m \times m$ region of the mesh, as shown in Figure 3.2. If $G_{t-1}(i, j) = 1$, then processor $P_{i,j}$, where $i, j \leq m$, contains the edge $(i', j') \in E$ of $G$ that edge $(i, j)$ of $G_{t-1}$ represents, as well as the weight $w(i, j) = w(i', j')$ of the edge. Further, the edges of $T$ that were chosen before stage $t$ are stored outside this region of the mesh as a collection of *special edges* $(x, y)$, with no more than one such edge per processor. When stage $t$ terminates, the weight matrix of $G_t$ must be stored in the upper-left $m' \times m'$, $m' \leq {}^{m}/_{2}$, region of the mesh, and the edges of $T$ chosen during stage $t$ must be stored as special edges, no more than one per processor outside the upper-left $m' \times m'$ region, but inside the upper-left $m \times m$ region.

In the following description of stage $t$, references to all processors and operations are with respect to the $m \times m$ region. Stage $t$ begins by performing a row rotation so that all processors in row $x$ know the minimum weight edge $(x, y)$ in $G_{t-1}$ (with ties broken in favor of minimum $y$) and the edge $(x', y') \in E$ that $(x, y)$ represents. Using a column rotation, these special edges are moved from diagonal processors $P_{i,i}$ to processors $P_{m,i}$. Notice that these edges have been moved to processors outside the upper-left ${}^{m}/_{2} \times {}^{m}/_{2}$ region but inside the $m \times m$ region.

Further, no processor contains more than one special edge.



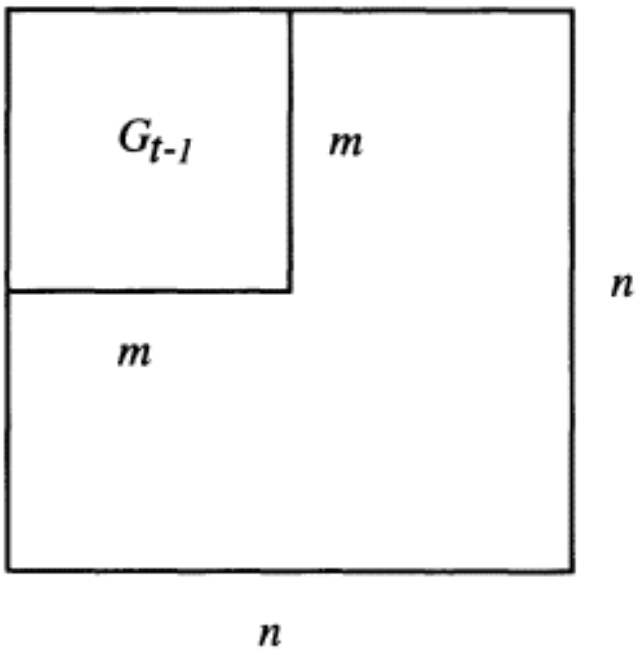


Figure 3.2:
$G_{t-1}$ is stored in the $m \times m$ region, where $m \leq 2t$-

In order to create the adjacency matrix for $H_t$, every processor $P_{x,y}$ for which row $x$ chose edge $(x, y)$ as the minimum weight edge, sets its entry of the adjacency matrix to 1. All other processors in row $x$ set their entry to 0. Finally, every processor $P_{x,y}$ for which row $x$ chose edge $(x, y)$ as the minimum weight edge, creates a message to inform processor $P_{y,x}$ that its entry in the adjacency matrix of $H_t$ should be 1, and a concurrent write is performed.

The last step of stage t is to create $G_t$ in the $m \times m$ region, and then compress it to a mesh of size no more than $\frac{m^2}{4}$ in the upper-left corner of the region. $G_t$ is created in the $m \times m$ region as follows. Compute $H_t^*$, the transitive closure of $H_t$, by the algorithm given in Section 2.5.3. Perform a row rotation so that every diagonal processor $P_{i,i}$ determines $c(i)$, the minimum index of a vertex in the component of $H_t$ that contains vertex $i$. Using a column and row rotation, every processor $P_{i,j}$ will know $c(i)$ and $c(j)$. Now, every processor $P_{i,j}$ for which $G_{t-1}(i, j) = 1$, $c(i) \neq c(j)$, and $(i, j)$ represents $(i', j') \in E$ creates a message to inform processor $P_{c(i),c(j)}$ that $G_t(c(i), c(j)) = 1$, that $(c(i), c(j))$ represents $(i', j')$, and that this edge has weight $w(i', j')$. A concurrent write, with ties broken



appropriately, will route the data properly.

$G_t$ can be compressed as follows. Perform a row rotation so that every processor $P_{i,1}$ knows whether or not there is a $j$ such that $G_t(i, j) = 1$. Perform a column rotation in column 1 so that all processors $P_{i,1}$ know the total number of vertices $m'$ in $G_t$, and the *rank* of vertex $i$ (i.e., the position in which vertex $i$ will appear, if at all) in $G_t$. Perform a row rotation, followed by a column rotation, so that all processors $P_{i,j}$ for which $G_t(i, j) = 1$ know $rank(i)$ and $rank(j)$. Every processor $P_{i,j}$, where $i, j \leq m'$, sets its entry for the adjacency matrix of $G_t$ to 0. Every processor $P_{i,j}$, for which $G_t(i, j) = 1$, creates a message that contains the original edge $(i', j') \in E$, and its associated weight $w(i', j')$, for which the processor is responsible. A concurrent write is used to route these messages to processors $P_{rank(i),rank(j)}$, which initialize the $(rank(i), rank(j))$ entry of $G_t$ to $w(i', j')$. Finally, all information regarding $G_{t-1}$ may be purged from the processors in the upper-left $m' \times m'$ region, concluding stage $t$.

The $t^{th}$ stage of the algorithm consists of a fixed number of operations, all of which are restricted to the $m \times m$ region. Therefore, the $t^{th}$ stage of the algorithm requires $\Theta(m) = \Theta(\frac{n}{2^{t-1}})$ time. Hence, the running time of the entire algorithm is $\Theta(n)$.

## 3.3 Connected Components

For the algorithms presented in this section, it is assumed that an $n \times n$ digitized picture $A = \{a_{i,j}\}$ is stored in a mesh of size $n^2$ so that processor $P_{i,j}$ contains pixel $a_{i,j}$. The pixels are in one of two states: black or white. It is useful to think of this digitization as being a black picture on a white background.

### *3.3.1 Counting Connected Components*

This section considers the problem of counting the number of *figures* (i.e., connected black components) in $A$ by a method known as "shrinking". The general idea of shrinking a digitized picture is that during every iteration of the algorithm each figure of the picture is trimmed until it becomes a single black pixel and then vanishes, all the while preserving the connectedness properties of the figures. To count figures by a shrinking algorithm, the processor responsible for a vanishing figure will add one to its local running sum of figures. A final report and



broadcast over the local sums will compute the global sum and inform all processors in the mesh as to the total number of figures in $A$.

Some definitions of connectedness are in order. A set $S$ of lattice points (e.g., a set of pixels or processors) is called *8-connected* if for all $P, Q \in S$ there exists a finite sequence $P = P_0, P_1, \ldots, P_l = Q$ of points of $S$ such that $P_i$ is a horizontal, vertical, or diagonal neighbor (i.e., an *8-neighbor*) of $P_{i-1}$, where $1 \leq i \leq l$. If only horizontal and vertical neighbors are considered (i.e., 4-*neighbors*), then $S$ is called 4-*connected*. [RoKi82]

Beyer [Beye69] and Levialdi [Levi72] independently arrived at an interesting method of "shrinking" the figures of a picture that guarantees that connected components (figures) always remains connected and components that are not connected always remain disconnected. Both algorithms assume an 8-connected definition of connectedness for the black pixels. As an aside, it should be noted that, in general, if an 8-connected definition is used for the black pixels, then a 4-connected definition is needed for the white pixels, and vice versa, in order to maintain the *Jordan Curve Theorem* (c.f., [Rose79]). The algorithm that follows shrinks figures toward the top right of the mesh. The reader is referred to [Levi72, Beye69] for details.

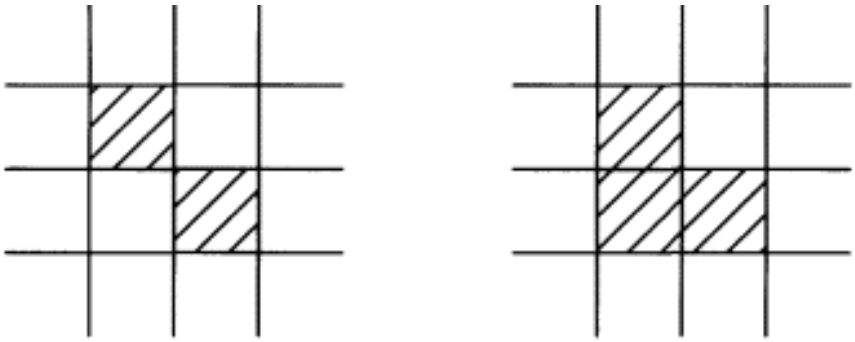
Assume that at the end of iteration $k$ of the algorithm, the picture $A$ has been transformed $k$ times according to the "shrinking" algorithm. Denote this picture as $A^k = \{a_{i,j}^k\}$, where $A^0 = A$ is the original picture. The transformation from $A^k$ to $A^{k+1}$ is given by describing how processor $P_{i,j}$ transforms $a_{i,j}^k$ to $a_{i,j}^{k+1}$. Consider $a_{i,j}^k$ to be the top right pixel of a $2 \times 2$ window. The transformation is given as follows.

1. If the configuration in the vicinity of white pixel $a_{i,j}^k$ matches either diagram presented in Figure 3.3 (a), then $a_{i,j}^{k+1}$ will become black.

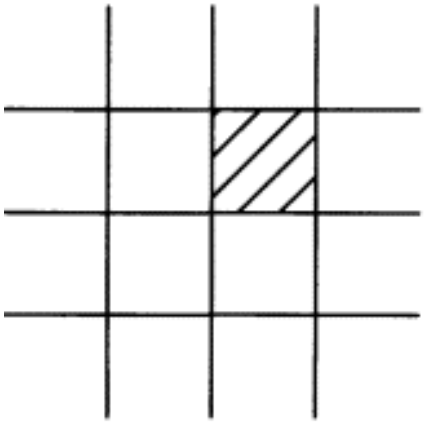
2. If the configuration in the vicinity of black pixel $a_{i,j}^k$ matches the diagram presented in Figure 3.3(b), then $a_{i,j}^{k+1}$ will become white. (In this situation, if all of the 8-neighbors of $a_{i,j}^k$ are white, then processor $P_{i,j}$ also increments its component counter.)

3. If neither situation illustrated in Figure 3.3 applies to $a_{i,j}^k$, then $a_{i,j}^{k+1} = a_{i,j}^k$ .

Using this shrinking scheme, it is shown in [Levi72] that during each iteration of the algorithm only black pixels that do not disconnect a figure are erased, and that white pixels do not become black when this implies the merging of two or more distinct figures. Further, it is shown



(a) Two situations for which $a_{i,j}^k$ is white and $a_{i,j}^{k+1}$ becomes black.



(b) The sole situation for which $a_{i,j}^k$ is black and $a_{i,j}^{k+1}$ becomes white.

Figure 3.3:
Assume that $a_{i,j}^k$, is the top right pixel of a 2 × 2 window.
Then there are exactly three situations in which $a_{i,j}^{k+1}$ will be black.



in [Levi72] that each figure will shrink to a single black pixel, at which point it is counted, in time proportional to the diameter of the minimum area iso-oriented rectangle that encloses the figure.

Therefore, the running time of the algorithm is determined by the time to perform the shrinking, which is O($n$), and the time to sum the counters. Since a global sum operation, as described in Section 2.4.3, finishes in Θ($n$) time, the running time for the entire algorithm is Θ($n$). A statement of the result follows.

**Theorem 3.11** *Given an $n \times n$ digitized black/white image, distributed one pixel per processor on a mesh of size $n^2$, the number of figures (connected components) can be determined in Θ($n$) time.* ·

### *3.3.2 Labeling Connected Components*

In this section, several algorithms are presented to label the *figures* (i.e., connected black components) of a digitized black picture on a white background. The algorithms presented in this section are mesh implementations of the generic component labeling algorithms given in Section 1.6.1.

As in Section 3.3.1, it is assumed that the $n \times n$ picture $A = \{a_{i,j}\}$ is stored in a mesh of size $n^2$ so that processor $P_{i,j}$ contains pixel $a_{i,j}$. However, in contrast to the assumptions of the previous section, this section defines two black pixels to be neighbors if and only if they are mapped to neighboring processors. That is, in this section a 4-connected definition of connectedness is assumed for the black pixels. This assumption is only for convenience of presentation, as an 8-connected definition will only change the running times of the algorithms by a constant factor.

Every processor that contains a black pixel uses its snake-like index as the label of the pixel that it contains. When a labeling algorithm terminates, every processor that contains a black pixel will also store the minimum label of any pixel that its pixel is connected to. Therefore, the label of a connected component will be the minimum label of any pixel in the component, and at the termination of the algorithm, every processor will know the label of the connected component that its pixel is a member of.

Following the order of presentation in Section 1.6.1, the first algorithm considered in this section can be classified as a simple *propagation* algorithm. Initially, every black processor (i.e., a processor containing a black pixel) defines its component label to be its snake-like index. During each iteration of the algorithm, every black processor sends its current



component label to its (at most) four black neighbors. Every black processor then compares its component label with the (at most) four labels just received, and keeps as its new component label the minimum of these labels. It is easy to see that for each figure, the minimum label $L$ is propagated from $P_L$ (using snake-like indexing of processors) to every processor $P_i$ in its figure in the minimum number of steps required to pass a message from $P_L$ to $P_{i,}$ under the restriction that data is only passed between neighboring black processors. Therefore, this labeling algorithm terminates in $\Theta(D)$ time, where $D$ is the maximum internal distance between any black pixel and the pixel of minimum label in its figure. (The internal distance between two black pixels is defined to be the length of a shortest connected black path between the pixels.) So, given 'blob-like' figures, all processors can know the label of their figure in $\Theta(n)$ time. However, it is easy to construct non-'blob-like' figures, such as the spirals or snakes depicted in Figure 1.10 of Section 1.6.1, for which this propagation algorithm would require $\Theta(n^2)$ time.

As an alternative to the propagation algorithm, one might consider exploiting the generalized transitive closure algorithm associated with Theorem 3.1 to solve the component labeling problem. Unfortunately, this algorithm cannot be used directly since there may be $\Theta(n^2)$ black pixels (vertices), which would require a matrix containing $\Theta(n^4)$ entries.

In contrast to the $O(n^2)$ propagation algorithm, the next two algorithms will label all figures in $\Theta(n)$ time, regardless of the number, shape, or size of the figures. Both algorithms use an efficient mesh implementation of a recursive divide-and-conquer solution strategy, following the generic divide-and-conquer component labeling algorithms presented in Section 1.6.1. (Descriptions of such algorithms begin on page 31.)

The first step of these algorithms is to recursively label the four quadrants of the mesh independently. After this step, the only figures that could have an incorrect global label are those figures that have a pixel on the border between the quadrants. See Figure 3.4. Each border processor $P$, (using snake-like indexing of processors) that contains a black pixel examines its (at most 2) neighboring processors in distinct quadrants. Border processor $P_x$ creates an unordered *edge record* (*label*1, *label*2, *clabel*1, *clabel*2) for each such border processor $P_y$ that also contains a black pixel, where *label*1 represents the label corresponding to the figure of $P_x$ after the recursive labeling, *label*2 represents the label corresponding to the figure of $P_y$ after the recursive labeling, and *clabel*1 and *clabel*2 will be used to determine the correct global labels of $P_x$ and $P_y$ and are initially defined to be *label*1 and *label*2, respectively. There are at most $4n$ - 4 processors along the border, and they



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | 4 | 4 | | |
| | 1 | | | | 4 | 4 | 4 |
| | | 18 | 18 | 20 | | | 4 |
| | 18 | 18 | | 20 | 20 | | 4 |
| | 33 | | | 36 | 36 | | 39 |
| 33 | 33 | | 44 | 36 | | | 39 |
| 33 | | | | | | 39 | 39 |
| 33 | 33 | 33 | 33 | 39 | 39 | 39 | |

Figure 3.4:
Sample labeling after recursively labeling each quadrant.

generate at most $4n$ such records. (Referring to Figure 3.4, edge records would be generated for component label pairs (1, 4), (18, 20), (18, 33), (20, 36), (4, 39), (36, 44), and (33, 39). Specifically, using snake-like indexing, processor $P_3$ would generate edge record (1, 4, 1, 4), processor $P_4$ would generate edge record (4, 1, 4, 1), processor $P_{19}$ would generate edge record (18, 20, 18, 20), processor $P_{20}$ would generate edge record (20, 18, 20, 18), processor $P_{24}$ would generate edge record (4, 39, 4, 39), and so forth.)

An important observation is that the amount of data that needs to be processed has been reduced from an amount proportional to the area of the mesh (image) to an amount proportional to the perimeter of the mesh (image). Also, the form of the data has changed from image data

representing a picture $A$, to geometric data in the form of an undirected graph $G = (V, E)$, where $V$ is the set of component (figure) labels for the border processors, and $(i, j) \in E$ if and only if $i, j \in V$, and $i \neq j$ are connected (neighbors). For instance, referring to Figure 3.4, the unordered edge set $E$ is {(1, 4), (18, 20), (18, 33), (20, 36), (4, 39), (36, 44), (33, 39)}. With the exception of edge (20, 36), each of these edges would be generated twice, and there would be a distinct edge record representing each instance. Notice that the edge (20, 36) would be generated four times (creating four edge records, only two of which are distinct), once each by processors with snake-like indices 26, 27, 36, and 37.

At this point, the two algorithms are distinguished. The first algorithm follows the compression algorithm presented on page 34 of Section 1.6.1 to resolve the global labels. The first step is to compress the O($n$) edge records to a submesh of size $4n$. The problem can now be viewed as solving the connected component labeling problem for unordered edge input given O($n$) edges on a mesh of size $4n$, where the vertices represent component labels and the edges represent adjacent components that need to be combined. Sort the edge records on the first field, *label*1, and let the first processor of each ordered interval create a *label record* (*label*1, *newlabel*), where *newlabel* is initialized to *label*1. These label records will be used to keep track of the component label for each of the O($n$) vertices. Notice that initially each vertex corresponds to a unique component label.

For a logarithmic number of iterations, update the *newlabel* field associated with each vertex so that after iteration $i$, *newlabel* represents the minimum label of vertices that are within a distance of $2^{i-1}$. This can be done as follows. Every processor maintaining label record (*label*1, *newlabel*) creates pseudo master record (*newlabel*, *temp-newlabel*), and every processor responsible for edge record (*label*1, *label*2, *clabel*1, *clabel*2) creates update records (*clabel*1, *clabel*2) and (*clabel*2, *clabel*1). A modified concurrent write that accommodates multiple pseudo master records with identical keys is used to update the *tempnewlabel* field of a pseudo master record, but only if the minimum update value is less than *newlabel*. The modified concurrent write sorts by key field, breaking ties in favor of pseudo master records (arbitrarily), and breaking ties of update records in favor of minimum data field, before using an interval operation to propagate the minimum update data value to all master records with that key. The next step is to update the label records based on the new information in the pseudo master records. This can be accomplished by performing a concurrent read based on the index of the processor with the label record that created the pseudo



master record. (I.e., this information could be stored in a field of these records.) The final step is to update the current label (*clabel*) fields of the edge records. To do this, perform a concurrent read where all label records (*label*, *newlabel*) act as master records, and each edge record (*label*1, *label*2, *clabel*1, *clabel*2) creates request records (*label*1, *clabel*1) and (*label*2, *clabel*2) for the purpose of updating the values of *clabel*1 and *clabel*2. This completes an iteration of the unordered edge component labeling algorithm.

Since the concurrent reads and modified concurrent write each take $\Theta(n^{1/2})$ time on a mesh of size $4n$, this unordered edge component labeling algorithm finishes in $\Theta(n^{1/2} \log n)$ time. Finally, all processors in the entire mesh of size $n^2$ that contain a black pixel perform a concurrent read to obtain the (possibly) updated label of their component from the label records that are stored in the submesh of size $4n$.

Compression and the concurrent read each take $\Theta(n)$ time. Since the unordered edge component labeling algorithm only takes $\Theta(n^{1/2} \log n)$ time (because the data was compressed to a submesh of size $4n$), the running time of the algorithm obeys the recurrence $T(n^2)=T(n^2/4) + \Theta(n)$, which is $\Theta(n)$. It should be noted that the $\Theta(\log n)$ time PRAM component labeling algorithm for unordered edge input from [ShVi82] can be simulated in the compressed mesh by having each step of the PRAM algorithm simulated with the aid of a mesh concurrent read and concurrent write that is restricted to the compressed region. Therefore, the algorithm given in [ShVi82] would also finish in $\Theta(n^{1/2} \log n)$ time when simulated in the compressed mesh. Further, [ReSt] gives an unordered edge component labeling algorithm for the mesh that runs in edgelength time (i.e., $\Theta(n^{1/2})$ time on a mesh of size $n$), thus eliminating the additional logarithmic factor. However, even if the algorithm from [ReSt] is used in the compressed mesh, there will be no affect on the asymptotic running time of the algorithm just described.

The second algorithm differs from the first in that compression is not used in order to resolve the global labels corresponding to the (at most) $4n - 4$ border pixels. Instead, following the cross-product algorithm presented on page 35 of Section 1.6.1, a symmetric adjacency matrix $M$ is created to assist in coalescing adjacent border elements, where $M$ represents the unordered edge records generated after the recursive call, and is created as follows. (Creating $M$ is similar to creating $G_t$ as the last step of stage $t$ in the algorithm of Theorem 3.10.) Sort the edge records by the first field. The first record within each interval is marked as the "leader" of the interval. Resort the records with the major key being those records marked as leaders and the minor key being the



first label field of the record. This collects the leaders together and allows each leader to determine the rank of its label with respect to the distinct labels generated at the end of the recursive labeling. All border processors that contain a record (*label*1, *label*2, *clabel*1, *clabel*2) perform a concurrent read to determine *rank*(*label*1) and *rank*(*label*2).

Notice that $M$ can represent at most $2n$ labels (vertices). In order to maintain $M$, the mesh of size $n^2$ will simulate a mesh of size $4n^2$. This can be done by having each processor responsible for a $2 \times 2$ submatrix of $M$. Specifically, entry $M(i, j)$ will be stored in processor $P_{\lceil \frac{i}{2} \rceil, \lceil \frac{j}{2} \rceil}$ of the mesh of size $n^2$. All processors initialize their 4 entries of the adjacency matrix $M$ to 0. Next, a concurrent write is performed to finish the initialization of $M$, where every processor containing edge record (*label*1, *label*2, *clabel*1, *clabel*2) creates a message to processor $P_{\lceil \frac{rank(label1)}{2} \rceil, \lceil \frac{rank(label2)}{2} \rceil}$ to inform that processor that the (*rank*(*label*1), *rank*(*label*2)) entry of $M$ should be 1.

Once the adjacency matrix $M$ is generated, the transitive closure, $M^*$, is computed by the algorithm of Section 2.5.3, and a row rotation with respect to $M$ is used to determine the label for each of the border elements. A final concurrent read corrects any possibly incorrect labels that existed after the recursive solution was known. The transitive closure algorithm and data movement operations each take $\Theta(n)$ time. Therefore, the running time of the algorithm obeys the recurrence $T(n^2) = T(n^2/4)+ \Theta(n)$, which is $\Theta(n)$.

**Theorem 3.12** *Given an $n \times n$ digitized black/white image, distributed one pixel per processor on a mesh of size $n^2$, the figures (connected components) can be uniquely labeled in $\Theta(n)$ time.* ·

Nassimi and Sahni [NaSa80] were the first to prove that the component labeling problem for digitized pictures on a mesh of size $n^2$ could be solved in $\Theta(n)$ time. It should be noted that the algorithm presented in [NaSa80] is different from the algorithm given in this section, and is quite interesting in its own right.

### 3.4 Internal Distances

In this section, solutions are presented to several problems involving distances within figures of digitized pictures. The term *black* (*white*) *processor* is again used to refer to a processor that maintains a black (white) pixel. For black processors $A$ and $B$, an *A-B path* is a sequence



of 4-connected black processors originating at $A$ and terminating at $B$. A *minimal A-B path* is an $A$ - $B$ path containing the minimum number of processors over all possible $A$ - $B$ paths. The *internal distance* from $A$ to $B$, denoted $d(A, B)$, is defined to be one less than the number of processors in a minimal $A$ - $B$ path. (Note: while a minimal $A$ - $B$ path may not be unique, the internal distance between $A$ and $B$ is.)

The problems in this section assume that an $n \times n$ digitized black/white picture $A = \{a_{i,j}\}$ is stored in a mesh of size $n^2$ so that processor $P_{i,j}$ contains pixel $a_{i,j}$. Given a special marked black pixel $X$, the main problem of this section is to determine $d(S, X)$ for every pixel $S$ in the same figure as $X$. This problem will be referred to as the *all-points minimum distance problem*. This problem occurs in image processing [HKW82], and from its solution one can find an internal spanning tree in $\Theta(1)$ additional time.

The all-points minimum distance problem can be solved by a simple propagation algorithm, similar to the propagation algorithm for labeling figures presented in Section 3.3.2. Every black processor, except $X$, initializes its minimum distance from $X$ to $\infty$. The processor containing $X$ initializes its distance to 0. At each iteration of the algorithm, every black processor sends its current minimum distance to its (at most) four neighboring black processors. Every processor then takes the minimum of a) its current distance and b) one more than the minimum of the distances that it just received, and uses this value as its new minimum distance. Again, for 'blob-like' figures, all processors will know their minimum distance to $X$ in $\Theta(n)$ time. However, it is again easy to construct figures, such as spirals and snakes, that will require $\Theta(n^2)$ time to propagate this information. It should be noted that the algorithm will work as described even if one marked pixel per figure is allowed. In this case, all pixels will determine the minimum distance to the marked pixel in its figure, if one exists.

In contrast to the $O(n^2)$ propagation algorithm, the next algorithm has a worst-case running time of $\Theta(n)$. This algorithm is based on using the generalized transitive closure operation described in Section 2.5.3. It will be assumed that the figures of the digitized picture have been labeled in $\Theta(n)$ time using the algorithm from Section 3.3.2. It will also be assumed that all processors of the mesh have been informed as to the label of $X$'s figure in $\Theta(n)$ time. This may be accomplished by performing a row operation that will inform all processors in $X$'s row as to the label of $X$'s figure, followed by a column operation, where every processor in $X$'s row informs all processors in their column as to $X$'s label. (Alternately, a standard report and broadcast may be used.)

Given a directed graph $G$ with $n$ vertices, define $S_k(i, j)$ to be the minimal length of a path from $i$ to $j$ using no intermediate vertex greater than $k$, as in Section 3.2. Then $S_k$ satisfies the recurrence

$$S_{k+1}(i, j) = min\{S_k(i, j), S_k(i, k) + S_k(k, j)\},$$

where

$$S_0(i, j) = \begin{cases} 0 & \text{if } i = j, \\ 1 & \text{if there is an edge from } i \text{ to } j, \text{ and} \\ \infty & \text{otherwise.} \end{cases}$$

Notice that $S \ldots n(i, j)$ is $d(i, j)$.

Unfortunately, the solution to the all-pairs minimum distance problem for matrix input, given in Theorem 3.1, cannot be used directly since there may be $\Theta(n^2)$ black pixels (vertices), which would require a matrix with $\Theta(n^4)$ entries. To reduce the matrix to $O(n^2)$ entries, the underlying geometry of the digitized picture is used. An optimal $\Theta(n)$ time solution to the all-points minimum distance problem will be described as a two-phase algorithm, with both phases being implemented via a recursive divide-and-conquer strategy.

At a given stage $i$ of a divide-and-conquer, let $k = 2^i$. The *outer border elements* of a $k \times k$ square are defined to be those processors in rows and/or columns 0 and $k$ - 1, with respect to the $k \times k$ square, that contain the same label as that of the marked processor. The *inner border elements* of a $k \times k$ square are defined to be those processors in rows and/or columns $\frac{k+1}{2}$ and $\frac{k+1}{2} - 1$, with respect to the square, that contain the same label as the marked processor. The term *border elements* is used to refer to the collection of inner and outer border elements of a $k \times k$ square. (See Figure 3.5.) Notice that the $k \times k$ squares are assumed to be aligned so that processors $P_{c*k-1, d*k-1}$, $1 \leq c, d \leq [n/k]$, mark the southeast processor of each $k \times k$ square.

The objective of the first phase of the algorithm is to obtain the distance to the marked processor for all border elements of the $n \times n$ mesh. This phase is implemented using a bottom-up divide-and-conquer solution strategy. The objective of the second phase of the algorithm is to obtain the distances to the marked processor for the remaining processors that are in the same figure as the marked processor. The second phase will be implemented via a top-down divide-and-conquer solution strategy, where each iteration requires an application of phase 1. The details of the algorithm are given in the proof of the following theorem.

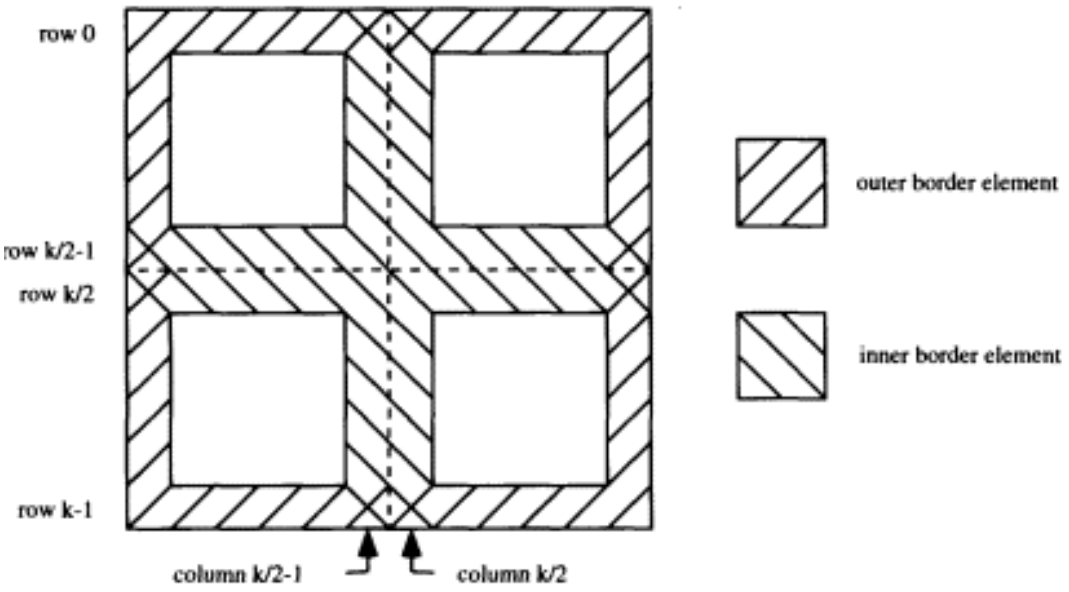



Figure 3.5:
Possible border elements of a submesh of size $k^2$.

**Theorem 3.13** *Given an $n \times n$ digitized black/white picture stored one pixel per processor in a mesh of size $n^2$, and given a marked processor $X$, in $\Theta(n)$ time every processor can compute its (possibly infinite) internal distance to $X$.*

*Proof.* The algorithm consists of two phases, as previously mentioned, which are given below.

*Phase 1:* The first phase follows a bottom-up divide-and-conquer strategy. It is presented for an arbitrary stage $i$, where $k = 2^i$. Before performing computations at stage $i$ on a $k \times k$ square $A$, the following must hold for each of the four $k/2 \times k/2$ subsquares of $A$ at the completion of stage $i$ - 1:

- A $(4k - 15) \times (4k - 15)$ matrix exists, with a row and column associated with each inner and outer border element of the subsquare, and a row and column associated with the marked processor $X$. (By convention, let the last row and column correspond to $X$.) The $(i, j)$ entry of this matrix is the restricted internal distance from the $i^{th}$ border element (or marked processor) to the $j^{th}$ border element (or marked processor), where *restricted internal distance*



refers to the minimum distance using only paths within the subsquare.

- Every entry in the matrix contains the unique IDs of the processors that the distance represents, where the ID is the row-major index of the processor.

- Every border element has a register containing its restricted internal distance to $X$.

The algorithm for stage $i$ follows. For all $k \times k$ squares $A$, the distance matrix $D$ for the border elements and marked processor must be set up. This matrix has a maximum size of $(8k - 15) \times (8k - 15)$. Notice that each of the 4 subsquares contributes a maximum of $2k - 4$ rows and columns (representing the outer border elements of the subsquare) to $D$. For simplicity, an $8k \times 8k$ simulated mesh is used to represent $D$. (Each processor of the $k \times k$ mesh simulates a mesh of size 64.)

Within each of the four $k/2 \times k/2$ subsquares, compress the $(4k - 15) \times (4k - 15)$ matrix from step $i - 1$ to the northwest by logically deleting the rows and columns that are not needed for the computations in square $A$ at step $i$. That is, by deleting from the step $i - 1$ matrices, those rows and columns not associated with $X$ or the outer border elements of the subsquare. Once each of the matrices has been compressed in its submesh, move the matrices to the regions as illustrated in Figure 3.6. This can be accomplished via a concurrent write in $\Theta(k)$ time since the only information necessary is the size of each of the four submatrices, which can be computed in $\Theta(k)$ time. Initialize the remaining entries of $D$ to $\infty$.

Perform a row rotation and a column rotation to propagate the coordinates (IDs) of the processor represented by each row and column of the matrix to the new entries (those just initialized to $\infty$) of $D$ so that they know which processors they represent. If an entry detects that it represents the distance between two adjacent inner border elements that were in different squares at stage $i - 1$, then the $\infty$ is replaced with a distance of 1.

Now, in $\Theta(k)$ time compute $D^*$ by using the generalized transitive closure operation of Section 2.5.3. Notice that $D^*$ represents the minimal internal path lengths between border elements and the marked processor, restricted to paths within the $k \times k$ region $A$. Next, pass the row representing $X$ through the $k \times k$ subsquare so that every border element can obtain and record its (perhaps infinite) restricted distance to $X$. This concludes stage $i$.



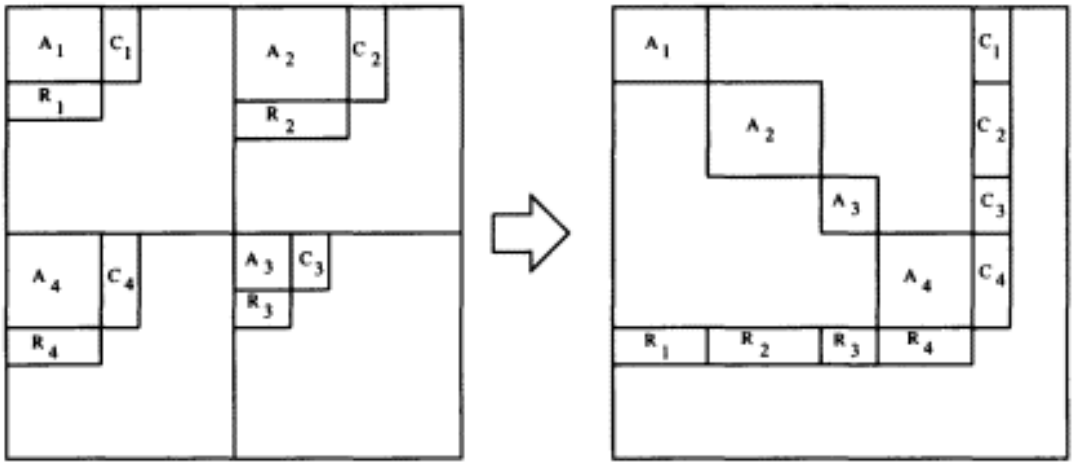



Figure 3.6:
Rearranging distance matrices to form $D$.

After $O(\log n)$ iterations, phase 1 will be complete and each of the border elements of the $n \times n$ mesh will have its correct internal distance to $X$.

*Phase 2:* To obtain the correct internal distances to $X$ for all border elements of each $n/2 \times n/2$ subsquare, simply apply phase 1 to each of the $n/2 \times n/2$ subsquares of the mesh. The only difference in the reapplication of the algorithm to each of the subsquares is that the distances obtained from the outer border elements of the $n/2 \times n/2$ squares to $X$ are correct and are used to obtain the correct internal distance for the (possibly incorrect) inner border elements of the subsquares. To obtain the correct distance for every processor in the same figure as the marked processor, simply continue this process recursively for $\Theta(\log n)$ iterations.

*Analysis:* The time to label the picture initially and pass the label of the marked processor to all processors is $\Theta(n)$. The time to complete phase 1 is $\Theta(n)$ since the time to complete each stage $i$ of the divide-and-conquer is $\Theta(2^i)$. The time to complete phase 2 obeys the recurrence $T(n^2) = T(n^2/4) + \Theta(n)$, which is $\Theta(n)$, since the time to compute the distances for the border elements of a $k \times k$ square is the time to complete phase 1 on that $k \times k$ square, which is $\Theta(k)$. Therefore, the running time of the algorithm is $\Theta(n)$. ·

The algorithm given above can be extended with minor modifications to the situation where there is one marked processor per connected component. At each stage of the recursion, the last row and column of the



border matrix will represent the set $X$ of processors, rather than a single processor.

**Corollary 3.14** *Given an $n \times n$ digitized black/white picture stored one pixel per processor in a mesh of size $n^2$, and given a set of marked processors $X$, with at most one marked processor per figure (i.e., connected component), in $\Theta(n)$ time each processor can compute its (possibly infinite) internal distance to $X$.* ·

In addition to knowing the internal distance between processors, it is sometimes desirable to *mark* minimal internal paths and to *count* the number of such paths. The all-points minimum distance algorithm presented in Theorem 3.13 can be used to mark all minimal paths between a pair of marked processors. This may be accomplished by applying the all-points minimum distance algorithm once with respect to each marked processor, broadcasting the minimum distance between the pair of marked processors to all processors, and then performing local computations within every processor based on the broadcast value and both minimum distance values. Once all of the minimal paths have been marked between the pair, every processor on multiple paths can locally eliminate all but one of the paths. This has the effect of marking a single minimal path between the pair. Finally, after marking all minimal paths between the pair, a modification of the all-points minimum distance algorithm may be used to determine (count) the number of minimal paths between the pair of marked processors.

**Theorem 3.15** *Given an $n \times n$ black/white picture stored one pixel per processor in a mesh of size $n^2$, and given marked processors $A$ and $B$, if the distance from $A$ to $B$ is finite, then in $\Theta(n)$ time*

*a) all minimal $A$ - $B$ paths can be marked,*

*b) a single minimal $A$ - $B$ path can be marked, and*

*c) the number of minimal $A$ - $B$ paths can be determined.*

*Proof.* Algorithms for these related problems follow.

a) The all-points minimum distance algorithm from Theorem 3.13 is performed twice; first with $A$ as the marked processor, and then with $B$ as the marked processor. Next, processor $A$ broadcasts $d(A, B)$ to all processors. Now, every processor $C$ such

that $d(A, C)+d(C, B)=d(A, B)$ determines that it is on some minimal $A$ - $B$ path. To mark all minimal $A$ - $B$ paths, every such $C$ "creates" an edge $(C, D)$ to each neighbor $D$ such that $d(D, B)=d(C, B)- 1$.

b) After marking all minimal $A$ - $B$ paths, every processor contains between zero and four edges. In order to mark a single minimal path, each processor discards all but one of its edges. Now there exists exactly one directed, minimal, $A$ - $B$ path. In order to mark this path, perform part a) again using only the directed edges that were just created.

c) After marking all minimal $A$ - $B$ paths, the remainder of the algorithm to count the number of minimal $A$ - $B$ paths is similar to phase 1 of Theorem 3.13, and is described for an arbitrary stage $i$ of the bottom-up divide-and-conquer strategy, where $k = 2^i$. Assume that each $k \times k$ square contains a $(4k - 2) \times (4k - 2)$ matrix $M$ that represents the number of distinct minimal paths between $A$,$B$, and the outer border elements of the $k \times k$ square, where only paths within the square are considered.

Merge four $k \times k$ squares to create a $(16k - 14) \times (16k - 14)$ matrix $M$. Place a 1 in $M(i, j)$ if the $(C, D)$ edge exits, where $i = rank(C)$ and $j = rank(D)$, where rank is with respect to $M$, and where $C$ and $D$ are neighbors from distinct $k \times k$ squares that are merged. Next, compute the number of minimal paths between entries in $M$. The $(i, j)$ entry of $M$ will get the value $fi(i, j)$, where $I = 16k - 14$, and $f$ is defined as

$$f_k(i,j) = \begin{cases} f_{k-1}(i,j) + f_{k-1}(i,k) * f_{k-1}(k,j) & \text{if } d(i,k)+d(k,j) = d(i,j). \\ f_{k-1}(i,j) & \text{otherwise.} \end{cases}$$

This requires a slight modification of the transitive closure algorithm that was presented in Section 2.5.3. When one processor passes an arbitrary $fk\text{-}l(i, j)$ to another processor, it must also pass $d(i, j)$, since this information is necessary in order to insure the proper evaluation of the function.

Finally, compress the matrix by deleting the rows and columns that do not represent $A$,$B$, or outer border elements of the $2k \times 2k$ square. The result is a matrix of size $(4(2k)\text{-}2) \times (4(2k)\text{-}2)$. Continue this merging, computing, and compressing process until the computations have been performed on the entire $n \times n$ mesh.



The entry $M(a, b)$, where $A$ is the $a^{th}$ row of the matrix and $B$ is the $b^{th}$ column, contains the number of minimal paths from $A$ to $B$.

In the theorem just presented, it is not necessary to restrict $A$ and $B$ to single processors. With only minor modifications to the previous algorithms, $A$ and $B$ can be arbitrary sets of processors. Given sets $A$ and $B$ of processors, define the internal distance from A to $B$ to be $min\{d(x, y) \mid \times \in A, y \in B\}$. (If $A$ and $B$ overlap then define $d(A,B)$ to be 0, and define marking minimal paths to mean indicating which processors are in both sets.) With a minor modification to the matrix M, the following result is obtained.

**Theorem 3.16** *Given an $n \times n$ digitized black/white picture stored one pixel per processor in a mesh of size $n^2$, and given (not necessarily disjoint) sets A and B of marked processors, in $\Theta(n)$ time each processor can compute its (perhaps infinite) distance to the set B, and the distance from A to B can be determined. Further, if the distance from A to B is finite, then in $\Theta(n)$ time*

*a) all minimal A - B paths can be marked,*

*b) a single minimal A - B path can be marked, and*

*c) the number of minimal A - B paths can be determined.*

One application of these internal distance algorithms is to the situation in which each figure has exactly one marked pixel, for example, the pixel with ID identical to the component label of the figure. Then by applying an algorithm closely related to those that have been presented, in $\Theta(n)$ time a breadth-first spanning tree can be constructed within each figure, where the *breadth-first spanning tree* of a graph is a spanning tree such that each vertex is at the minimal possible distance from the root.

It should be noted that the previous algorithms work equally well if the edges between pixels are directed and have arbitrary positive weights. If negative edge weights are allowed, then if there is a cycle with a negative total weight, the cycle can be repeated arbitrarily often to make distances as negative as desired. Therefore, any path touching such a cycle with a negative total weight should be given a total distance of $-\infty$. With proper modifications, negative weights can be accommodated.



**Theorem 3.17** *Given a mesh of size $n^2$ such that each processor contains a directed weighted edge to each of its neighbors, where the weights can be $+\infty$ or any real number, and given (not necessarily disjoint) sets A and B of processors, in $\Theta(n)$ time each processor can compute its (perhaps infinite) distance to B, $d(A, B)$ can be determined, and it can be decided whether or not all cycles have a positive total distance. Further, if all cycles have a positive total distance and $d(A,B)$ is finite, then in $\Theta(n)$ time*

*a) all minimal A - B paths can be marked,*

*b) a single minimal A - B path can be marked, and*

*c) the number of minimal A - B paths can be determined.*

*Proof.* If there is a cycle with negative total weight then the recurrence used to calculate internal distances is no longer correct. To remedy this, when working in any square, first run the algorithm as before, except that all diagonal entries are initialized to be $+\infty$. The $(i, i)$ entry of the resulting matrix is negative if and only if vertex $i$ is on a cycle of total negative distance. (The entry is 0 if and only if the vertex is on a cycle of zero total distance, and is not on any negative cycles.) If the $(i, i)$ entry is negative, it is replaced with $-\infty$, as is any entry other than $+\infty$ in the *ith* row and *ith* column. (This is because any path leading into or out of a negative cycle should have a path length of $-\infty$.) Now the generalized transitive closure algorithm is run again, where the definition $(-\infty) + (+\infty) = +\infty$ is used. It can be shown that the resulting matrix has the correct distances, and is therefore ready for the next stage. ·

The *internal diameter* of a set $A$ of processors is defined to be $max\{d(x, y) \mid x, y \in A\}$. (External diameters are discussed in Section 3.6.) Fischler [Fisc80] shows how the internal diameter can be used to classify cracks in an industrial inspection application. For an arbitrary set $A$ of processors, an efficient algorithm for determining the internal diameter is an open problem. However, for an important case where $A$ is a connected component without holes, then its internal diameter can be determined efficiently. (This includes the case of interest to Fischler.)

The outline of the solution is given, which is based on the algorithm associated with Theorem 3.13. When working on finding distances in some square, for each black processor (vertex) $P_x$ on the border of the square there is another black processor (vertex), denoted $F(P_x)$, that is one of the furthest processors from $P_x$ in the square, subject to the condition that $F(P_x)$ is connected to $P_x$ in the square. (It may be



that $F(P_x)$ is another border processor, in which case it was already a vertex, and it may be that $F(P_x)$ and $F(P_y)$ are the same even though $P_x$ and $P_y$ are not. In these cases, the redundant vertices are eliminated.) The important fact is that, in a figure with no holes, for each border processor $P_x$, $F(P_x)$ can be selected from among $\{F(P_y) \mid P_y$ is a border element of a subsquare$\}$. Further, it can be shown that the largest finite internal distance ever calculated during any stage is the internal diameter. Incorporating these facts, the following is obtained.

**Theorem 3.18** *Given an $n \times n$ digitized black/white picture stored one pixel per processor in a mesh of size $n^2$, then in $\Theta(n)$ time every processor in a figure without a hole can determine the internal diameter of its figure.* ·

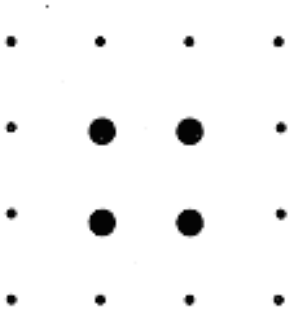
It should be noted that using techniques from [Beye69, Levi72, MiPa69], in $\Theta(n)$ time each figure can decide whether or not it has any holes.

### 3.5 Convexity

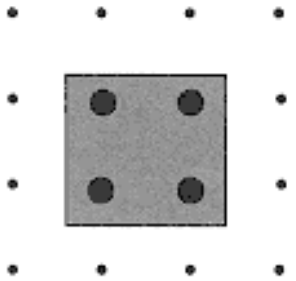
In this section, solutions are presented to several problems involving convexity. Central to this work is the identification of the processor at position $(i, j)$ with the integer lattice point $(i, j)$. A set of processors is defined to be *convex* if and only if the corresponding set of integer lattice points is convex, i.e., the smallest convex polygon containing them contains no other integer lattice points. While this is the proper notion of convexity for integer lattice points, it does have the annoying property that some disconnected sets of points, such as $\{(0, 0), (2, 3)\}$, are convex.

The relationship between the convexity of processors and the convexity of the original figures is complicated by the digitization process. If the integer lattice corresponding to the processors is placed atop a black/white picture (*not* a *digitized* picture) and each lattice point is given the color of the point it covers, then a convex black figure will yield a convex set of black lattice points. For each convex set of black lattice points, there is a convex black figure whose digitization is the given set, but this figure is never unique. Further, there are nonconvex black figures which also yield the given set. See Figure 3.7.

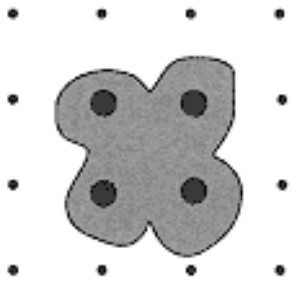
Some digitization schemes associate each lattice point with the center of a closed unit square (with adjacent squares overlapping on their



A convex set $S$ of PEs.



A convex black figure
whose digitization is $S$.



A nonconvex black figure
whose digitization is $S$.

Figure 3.7:
Convex and nonconvex figures that yield the same convex set of lattice points.



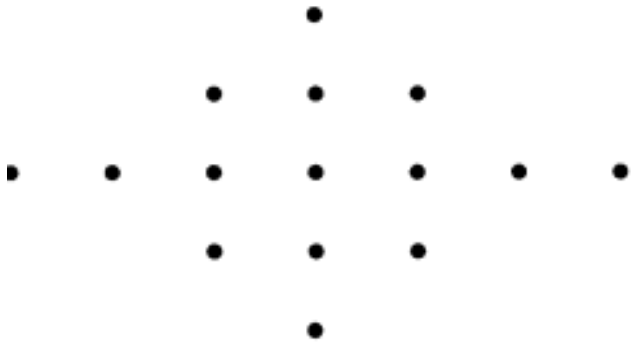


Figure 3.8:
A convex set of black lattice points which, in some digitization schemes, cannot arise as the digitization of a convex black figure.

edges). If a lattice point is colored black when all of its square is black, then once again a convex black figure will yield a convex set of black lattice points, and for any convex set of black lattice points there are both convex and nonconvex black figures which yield the given set.

For some digitization schemes, however, the correspondence is not quite as nice. For example, suppose each lattice point is again viewed as the center of a closed unit square, but now it is colored black if any part of its square is black. Convex black figures will yield convex connected sets of black lattice points, but not all convex connected sets of black lattice points can arise as the digitization of a black convex figure, as shown in Figure 3.8. Readers interested in pursuing the relationship between convexity and digitization are referred to [KiRo82b], and the references contained therein.

Given a set $S$ of processors, the *convex hull of* $S$, denoted *hull*($S$), is the smallest convex set of processors that includes $S$. A processor $P \in S$ is defined to be an *extreme point* of $S$ if and only if $P \notin \text{hull}(S - \{P\})$. That is, the extreme points of $S$ are the corners of the smallest convex polygon containing $S$. It is said that *the extreme points of S have been identified* if each processor in $S$ has decided whether or not it is an extreme point of $S$. It is said that the *extreme points of S have been enumerated* if for every processor $P_i$ containing a point $p \in S$, the following hold. (See Figure 3.9.)

1. $P_i$ has a Boolean variable 'extreme', and extreme is true if and only if $p$ is an extreme point of $S$.



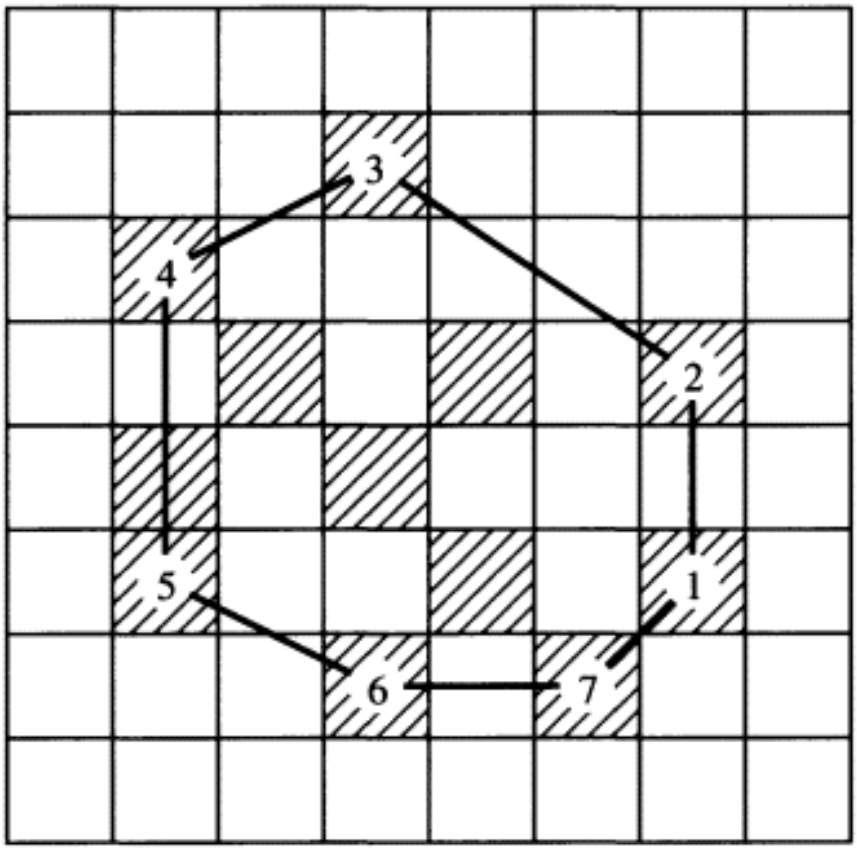



Figure 3.9:
Enumerated extreme points of $S$.

2. $P_i$ stores the total number of extreme points of *hull*($S$).

3. If $p$ is an extreme point of $S$, then $P_i$ stores the position of $p$ in the counterclockwise ordering of extreme points. (The rightmost extreme point is assigned the number 1. If there are two rightmost extreme points, then the lower one is assigned the number 1.)

4. If $p$ is an extreme point of $S$, then $P_i$ stores the Cartesian coordinates of the extreme points that precede and succeed $p$, as well as the ID of the processors that contain them.

Many queries concerning $S$ can be reduced to questions concerning the extreme points of $S$. On an $n \times n$ mesh, $S$ may have $\Theta(n^2)$ points, but since $S$ has at most two extreme points in any row, it has $O(n)$ extreme points. In fact, by using a little number theory, it has been shown that $S$ has only $O(n^{2/3})$ extreme points [VoK182]. Since it takes



$\Omega(n)$ time to move data across an $n \times n$ mesh, the $O(n^{2/3})$ bound on the number of extreme points does not help in most algorithms, though it is crucial to the algorithm of Theorem 3.33. The reader interested in serial algorithms and general descriptions of the convexity problems considered in this section are referred to [Sham78, Tous80].

The first problem considered in this section is that of identifying the extreme points for every labeled set of processors. Initially, every processor contains a label, and when the algorithm terminates each processor must know whether or not it is an extreme point with respect to all processors containing its label. These labels may arise as the labels of figures (i.e., connected components), but there is no requirement that they do so. An algorithm to solve this problem first identifies the leftmost and rightmost processors for each label present in every row. That is, every row uses a row rotation to solve its restricted extreme point identification problem for all labels in its row. Sorting is then used to gather these row-restricted extreme points together for every label, where it may be noted that no label will contain more than $2n$ such points. Finally, all row-restricted extreme points are viewed by all other row-restricted extreme points with the same label, during which it is possible for each such point to decide whether or not it is an extreme point with respect to its label.

The algorithms presented in this section assume that there are constant time serial algorithms enabling a processor to decide if one integer lattice point is on the line segment between two others, if one integer lattice point is on the line determined by two others, if one integer lattice point is in the angle determined by three others (one of which is designated as the vertex), and if one integer lattice point is in the closed triangle determined by three others.

**Theorem 3.19** *In a mesh of size $n^2$, simultaneously for all labels $A$, in $\Theta(n)$ time the extreme points of the processors labeled $A$ can be identified.*

*Proof.* First, each processor determines whether or not it is either the leftmost or rightmost processor containing its label in its row. This is done in $\Theta(n)$ time by rotating within every row the label and position of all processors in the row. When finished, each processor that is either a leftmost or rightmost processor for its label within its row places its label and position into its sort field, while all other processors put $\infty$ and their position into their sort field. These values are sorted into snake-like order using the label as the key.

For all finite labels, rotate in snake-like fashion, as described in Section 2.6.2, the position information in the sort field among all processors

with the same label in their sort field. Since for each finite label there are at most $2n$ such positions, this can be done in $O(n)$ time. As the information rotates, each processor determines whether or not the position in its sort field is an extreme point with respect to the positions that the processor has viewed. This is done as follows. Suppose $X$ is the position in the sort field of a given processor. The positions of at most two more processors with the same label will be stored. As each new position $Y$ arrives, if no other position has been stored then $Y$ is copied. If only one other position $U$ has been stored, then the processor determines whether or not $X$ is on the line segment between $Y$ and $U$. If it is, then $X$ is not an extreme point; otherwise $Y$ and $U$ are stored, unless $Y$ is on the line determined by $X$ and $U$, in which case only $U$ is kept.

Finally, if two positions $U$ and $V$ are being stored when $Y$ arrives, then the processor determines whether or not $X$ is in the (perhaps degenerate) closed triangle formed by $Y$, $U$, and $V$. If it is, then $X$ is not an extreme point. Otherwise the processor needs to determine which two of $Y$, $U$, and $V$ are to be stored. One of these must be in the angle formed by the other two with $X$ at the vertex, and it is this one that is eliminated. (If $Y$ lies on the line determined by $X$ and one of the others, then $Y$ is eliminated.)

There is some constant $C$, such that after $Cn$ time units the information is finished rotating. It is easy to show that at any time in the computation, if a processor has not yet determined that the position in its sort field is not an extreme point, then the position is an extreme point of the set of points that have passed through the processor thusfar. Therefore, when the information is finished rotating, if a processor has not yet determined that the position in its sort field is not an extreme point, then the position must be an extreme point. (Notice that if a processor responsible for $X$ determines that $X$ is an extreme point, then the hull edges incident on $X$ are represented by $\overline{XU}$ and $\overline{VX}$, where $U$ and $V$ are the two positions stored in the processor responsible for $X$ at the end of the rotation.) A final sort step based on the snake-like index of the processor that originated the records, sends the information back so that each processor knows whether or not it is an extreme point for its label. Sorting, rotating data within intervals of size at most $2n$, and a row rotation each take $\Theta(n)$ time. Therefore, the running time of the algorithm is as claimed.

The problem of enumerating the extreme points for every labeled set of processors can also be solved in mesh optimal $\Theta(n)$ time. Once



the extreme points have been identified for every labeled set, sort the extreme points by label so that all extreme points of each set are in a contiguous region of the mesh. Simultaneously within each such ordered interval, perform a rotation so that all processors know the leftmost and rightmost extreme points (breaking ties in favor of lower) of their label, perform a semigroup (i.e., associative binary) operation to determine the number of extreme points representing the upper hull (i.e., the number of extreme points on or above the line between the leftmost and rightmost extreme points), and finally perform a rotation where each upper (lower) extreme point counts the number of upper (lower) extreme points in higher (lower) numbered columns. With this information, every processor can determine in $\Theta(1)$ time the number of the extreme point it contains with respect to the enumerated sequence of extreme points of its label. Once the extreme points are numbered, an ordered interval rotation will allow every point to determine the necessary information regarding its preceding and succeeding extreme points. A final sort sends the points back to the original processors.

Given digitized picture input, a fundamental image operation is that of determining for each figure (i.e., connected component) whether or not it is convex. The algorithm that follows solves the problem by first determining in each row whether or not the restriction of each figure to that row is convex. Sorting is then used to group together the at most $2n$ row-restricted extreme points for each label. For each label that is comprised solely of row-restricted convex segments, a rotation within ordered intervals of these row-restricted extreme points is used to determine for each figure whether or not it is convex.

**Corollary 3.20** *Given an $n \times n$ digitized black/white picture stored one pixel per processor in a mesh of size $n^2$, in $\Theta(n)$ time every figure (i.e., connected black component) can decide whether or not it is convex.*

*Proof.* A connected set $S$ of processors is convex if and only if for each processor $P$, such that $P \notin S$ and $P$ is the right or left neighbor of a processor in $S$, $P \notin hull(S)$. For each such $P$, the algorithm will check if $P \notin hull(S)$ by checking if $P$ is an extreme point of $S \cup \{P\}$.

First, use a $\Theta(n)$ time labeling algorithm from Section 3.3.2 to label the processors. Then, by rotating information within rows, each figure determines if its restriction to every row is convex, e.g., an "O" shaped figure would find rows in which it is not convex, but a "Z" shaped one would not. By sorting with respect to figure labels, performing a semigroup (i.e., associative binary) operation within sorted intervals,



and then resorting by the snake-like index of the processor creating the record, in $\Theta(n)$ time every processor can know whether or not all rows of its figure are convex. The algorithm associated with Theorem 3.19 is then used to identify the extreme points of each figure for which all rows are convex. This is done for all figures simultaneously in $O(n)$ time, since each figure has at most $2n$ points involved in the algorithm. As the information is rotating and the processors are determining if the position in their sort field is an extreme point, they also determine if the processor to the right of that position (if the position is the rightmost processor with its label in its row), or to the left of that position (if the position is the leftmost processor), is an extreme point. The coordinates of these extra processors are not rotated, but the algorithm of Theorem 3.19 is performed for them. When done, if any one of these extra processors is not an extreme point, then the figure is not convex, while otherwise it is. Finally, a $\Theta(n)$ time concurrent read can be used to insure that each black processor knows whether or not its figure is convex. ·

In the early 1960s, Unger [Unge62] gave a $\Theta(n)$ time algorithm for detecting horizontal and vertical concavities, i.e., concavities detectable by traveling along a horizontal or vertical line, respectively. He also noted that although any figure with such concavities is not convex, there are nonconvex figures, such as the digitization of a pear or banana, without such concavities. So, while an algorithm to detect features such as horizontal and vertical concavities is useful, it cannot be used to decide convexity. The algorithm to detect such concavities is given below and is based on exploiting straightforward row and column rotations.

**Corollary 3.21** *Given an $n \times n$ digitized black/white picture stored one pixel per processor in a mesh of size $n^2$, in $\Theta(n)$ time the number of vertical and horizontal concavities can be determined for every figure (i.e., connected black component).*

*Proof*. First, use a $\Theta(n)$ time component labeling algorithm from Section 3.3.2 to label the processors. An algorithm to detect horizontal concavities for each figure is outlined (the vertical concavity detection algorithm is similar). Simultaneously for all rows, perform a row rotation so that each processor determines whether or not there are any black pixels in its figure to its left. Each black processor with a white processor to its left now knows if it is the right end of a horizontal concavity within its row. A sort step to bring together information from each figure label, semigroup (i.e., associative binary) operation within intervals to determine for each figure the number of horizontal concavities, and final sort step to redistribute the results, can be used to inform



every processor as to the number of horizontal concavities in its figure. Therefore, the algorithm is complete in $\Theta(n)$ time. ·

Solutions to two problems at the end of this section show applications of convexity that are not obvious. First, a useful algorithm is given to decide for every figure whether or not its convex hull contains any black pixels which are not in the figure. The algorithm follows from techniques used in the algorithms of Theorem 3.19 and Corollary 3.20.

In $\Theta(n)$ time, label the processors via the component labeling algorithm presented in Section 3.3.2. Using the $\Theta(n)$ time algorithm of Theorem 3.19, identify the extreme points of every figure. Next, use a row rotation to mark the leftmost and rightmost point for every label in each row. Another row rotation determines for each figure whether or not there is a point of a different label in the convex hull of the restriction of the figure to the row. For those figures that have thusfar not found a point from another figure in their hull, perform a row rotation so that in every row the nearest point of a different label to the left of the leftmost point and to the right of the rightmost point is determined (if they exist). Now, with minor modifications, the core of the algorithm from Theorem 3.19 can be used to determine for every figure whether or not any of these additional points are in the convex hull of the figure.

**Corollary 3.22** *Given an $n \times n$ digitized black/white picture stored one pixel per processor in a mesh of size $n^2$, in $\Theta(n)$ time every figure (i.e., connected black component) can decide whether or not its convex hull contains any black pixels not in the figure. Further, in $\Theta(n)$ time every figure can decide whether or not any processors in it are in the convex hull of another figure.*

The problem of determining whether or not two sets of processors are linearly separable [Tous80] is related to determining convexity. Suppose $A$ and $B$ are, not necessarily disjoint, sets of processors, and that each member of $A$ contains the label $A$, and that each member of $B$ contains the label $B$. Then the set of processors labeled $A$ is *linearly separable* from the set of processors labeled $B$ if and only if there exists a straight line in the plane such that all lattice points corresponding to processors labeled $A$ lie on one side of the line, and all lattice points corresponding to processors labeled $B$ lie on the other side. An observation that allows for the application of the algorithm associated with Corollary 3.22 to this problem is that two sets are linearly separable if and only if their convex hulls are disjoint. Therefore, the following is obtained.



**Corollary 3.23** *In a mesh of size $n^2$, in $\Theta(n)$ time it can be decided whether or not the processors labeled $A$ are linearly separable from the processors labeled $B$.*

Given a set $S$ of points in the plane, a *smallest (enclosing) box* of $S$ [FrSh75, Tous80] is a rectangle of smallest area containing $S$. Notice that a smallest enclosing box of $S$ is not necessarily unique, but that the area of a smallest enclosing box of $S$ is unique. It can be shown that each of the sides of a smallest enclosing box of $S$ must contain an extreme point of $S$, and that at least one side of such a box must contain two (consecutive) extreme points of $S$ [FrSh75]. The following algorithm finds for every hull edge (i.e, pair of consecutive extreme points) a smallest enclosing box of the figure that has one side collinear with the edge, and then takes the minimum over all such boxes.

**Corollary 3.24** *In a mesh of size $n^2$, in $\Theta(n)$ time every labeled processor can determine a smallest enclosing box containing all processors with the same label.*

*Proof.* First, perform the algorithm associated with Theorem 3.19, except that there is no need to perform the final sort and send the extreme point information back to the originating processors. If a processor $P$ has position $X$ in its sort field, and if $X$ is an extreme point, then at the conclusion of the algorithm the other two points being stored (call them $U$ and $V$) are also extreme points. (If no other points are being stored then only one processor has its label, while if only one other point is being stored, then the processors with that label form a straight line segment.) By using the angle $UXV$, $P$ can determine whether traveling from $X$ to $U$ or from $X$ to $V$ will produce a counterclockwise traversal around the convex hull. For convenience, assume that it is from $X$ to $U$.

Processor $P$ now tries to determine the corners of the rectangle, as illustrated in Figure 3.10. It does this by finding $R$, $S$, and $T$, where $R$ is a point furthest from the line $XU$, $S$ is a point whose projection onto the line $XU$ is the most negative (where $X$ is the origin and $U$ is at a positive location), and $T$ is a point whose projection onto $XU$ is the most positive. To enable each processor to find the coordinates of $R$, $S$, and $T$, corresponding to the point $X$ that it is storing, rotate the position information again. When finished, each processor $P$ having an extreme point $X$ in its sort field will know the points $R$, $S$, and $T$, that correspond to $X$, and hence can compute the corners and area



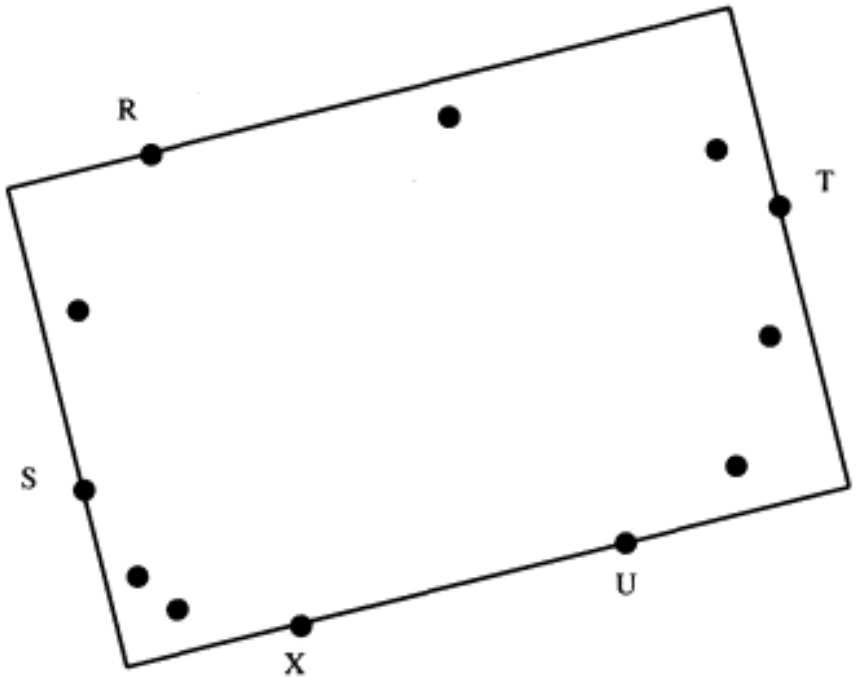


Figure 3.10:
A smallest rectangle.

of its box. These boxes are rotated in snake-like fashion among the processors with the same label in their sort field, which enables each processor to determine a smallest box for the label of the point in its sort field. A concurrent read is used so that every labeled processor is notified regarding the identity of a smallest enclosing box. ·

## 3.6 External Distances

In this section, problems are considered that involve external distances between processors, where again the processor at position $(i, j)$ is identified with the integer lattice point $(i, j)$. The most common distance measures used are the $l_p$ metrics, where for $1 \leq p < \infty$, the $l_p$ distance from $(a, b)$ to $(c, d)$ is $[|\ a - c\ |^p + |\ b - d\ |^p]^{1/p}$ . (The $l_\infty$ distance from $(a, b)$ to $(c, d)$ is $max\{|\ a - c\ |, |\ b - d\ |\}$.) The connection scheme of the mesh is based on the $l_1$ ("taxi-cab" or "city block") metric, so problems are usually easiest when expressed in terms of this metric (c.f., Theorem 3.25 and Theorem 3.28). Further, simple techniques can also be applied to solve problems in terms of the $l_\infty$ metric. However, for



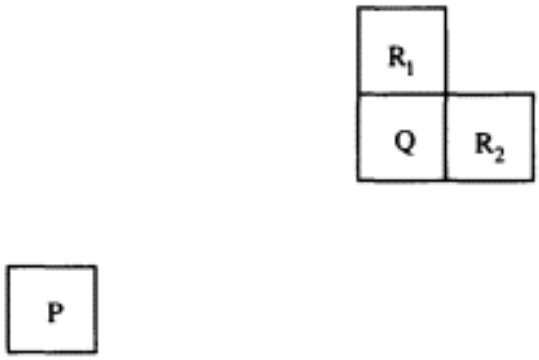



Figure 3.11:
For a monotone metric $d$, $d(P, Q) \leq d(P, R_1)$ and $d(P,Q) \leq d(P, R_2)$.

other metrics, such as the important $l_2$ (Euclidean) metric, slightly more sophisticated methods are needed to solve external distance problems on the mesh.

In this section, it is assumed that there is a function $d(x, y)$ which computes, in unit time, the distance between points $x$ and $y$. The function $d$ cannot be completely arbitrary, for then there would be no connection between the metric and the underlying geometry of the mesh. To avoid this, only monotone metrics will be considered, where a metric $d$ is said to be *monotone* if for all processors $P$, $Q$, and $R$, if $Q$ and $R$ are neighbors and the $l_1$ distance from $P$ to $R$ exceeds the $l_1$ distance from $P$ to $Q$, then $d(P, R) \geq d(P, Q)$. (See Figure 3.11.) All $l_p$ metrics are monotone, and it seems that monotone metrics are the only ones ever encountered in practice.

Let $S$ be a set of processors. The *external diameter* of $S$ is defined to be $max\{d(x, y) \mid x, y \in S\}$. For the $l_1$ and $l_\infty$ metric, Dyer and Rosenfeld [DyRo81] presented an algorithm to compute the external diameter of a given set $S$ of processors in $O(n)$ time on a mesh of size $n^2$. Given a mesh of size $n^2$ with each processor labeled black or white, Theorem 3.25 presents an algorithm to compute a nearest and farthest black processor for each processor, as well as to compute the external diameter of a set of black processors, all in optimal $\Theta(n)$ time for the $l_1$ and $l_\infty$ metrics. The algorithm exploits the connection scheme of the mesh to solve the problem by a straightforward combination of row and column rotations, during which each processor retains the identity of a nearest or farthest neighboring black processor.

Page 133

**Theorem 3.25** *Given a set $S$ of black processors on a mesh of size $n^2$, in $\Theta(n)$ time every processor can know a nearest and farthest black processor, and in $\Theta(n)$ time every processor can know the external diameter of $S$. These computations are with respect to either the $l_1$ or $l_\infty$ metric.*

*Proof.* The following algorithm can be used with respect to either the $l_1$ or $l_\infty$ metric to determine the nearest black neighbor for every processor in the mesh. Simultaneously for all rows, perform a row rotation so that every processor, including those that are not black, determines a nearest black processor in its row (ties broken arbitrarily). Each processor now creates a *neighbor record* that contains its row coordinate and the column coordinate of the nearest black processor in its row that was just determined. Simultaneously for all columns, perform a column rotation, where every processor views the neighbor records of all processors in its column and retains the data record corresponding to a nearest black neighbor (ties broken arbitrarily), with respect to the appropriate metric.

To determine a farthest neighbor for every processor in the mesh, simply modify the algorithm so that maximum distances instead of minimum distances are retained during the rotations. To determine the external diameter of the set $S$ of black processors, simply perform a semigroup operation taking the maximum over all of the farthest neighbor distances, where only the distances contained in the black processors are used. Each of the row and column rotations takes $\Theta(n)$ time, as does the semigroup (i.e., associative binary) operation. ·

Suppose that instead of one set of black processors, the input consists of multiple sets of labeled processors. Further, suppose that it is necessary to compute the external diameter of each set with respect to an arbitrary monotone metric, as is the case in some image algorithms that require the use of the $l_2$ (Euclidean) metric (c.f., [Fisc80]). In the following theorem, an optimal mesh algorithm is presented to determine the external diameter for every labeled set of processors, with respect to an arbitrary monotone metric. The algorithm exploits the fact that for arbitrary monotone metrics, the external diameter of a set $S$ of processors is $max\{d(x, y) \mid x, y$ are the rightmost or leftmost elements of $S$ in their rows$\}$.

**Theorem 3.26** *In a mesh of size $n^2$, for any monotone metric, in $\Theta(n)$ time every labeled processor can determine the external diameter of the processors with its label.*

Page 134

*Proof*. Let $S$ denote a set of processors. If the metric were an $l_p$ metric, then the fact that the external diameter of $S$ is equal to $max\{d(x, y) \mid x, y$ are extreme points of $S\}$ could be exploited. For arbitrary monotone metrics this is no longer true, but it is true that the external diameter of $S$ is equal to $max\{d(x, y) \mid x, y$ are the rightmost or leftmost elements of $S$ in their rows$\}$. As in Theorem 3.19, every processor first determines if it is either a leftmost or rightmost processor with its label in its row. Each such processor puts its label and coordinates into its sort field and all other processors put infinity and their coordinates into their sort field. These elements are then sorted with the label as primary key. For each finite label, the coordinates are now rotated (in snakelike fashion), and each processor keeps track of the maximum distance from the coordinates in its sort field to any of the received coordinates. When the coordinates are done rotating these maxima are rotated. The solution for each ordered interval (i.e., each labeled set) is the largest of the maxima. A concurrent read then insures that each labeled processor knows the external diameter for its label.

A closely related problem to that of the previous theorem is the *all-points farthest point problem* [Sham78], in which for every labeled processor, the greatest distance to a processor with the same label is to be determined. With a slight change to the preceding algorithm, the following is obtained.

**Corollary 3.27** *In a mesh of size $n^2$, for any monotone metric, in $\Theta(n)$ time the all-points farthest point problem can be solved.*

Theorems 3.25 and 3.26 were concerned with finding distances among processors with the same label, while the following theorems are concerned with finding distances between processors with different labels. The first problem considered is that of determining for every processor the distance and label to a nearest (farthest) distinctly labeled processor, with respect to the $l_1$ and $l_\infty$ metrics. The algorithm presented to solve the problem is similar to that of Theorem 3.25 in that it is based on a combination of row and column rotations.

**Theorem 3.28** *In a mesh of size $n^2$, for the $l_1$ and $l_\infty$ metrics, in $\Theta(n)$ time every labeled processor can determine the distance and label of a nearest and farthest processor with a different label, if such a processor exists. Further, for every set of processors, a nearest distinctly labeled set of processors can be determined in $\Theta(n)$ time.*



*Proof*. Simultaneously for all rows, perform a row rotation so that every processor finds a nearest distinctly labeled processor in its row. Perform a second row rotation so that every processor finds a nearest distinctly labeled processor that is of a different label than the one just found. Each processor now creates a record that contains distance and label information of the (at most) two distinctly labeled processors just determined. Simultaneously for all columns, perform a column rotation so that every labeled processor can detect a nearest processor of a different label, if such a processor exists.

This information can now be used to determine a nearest set of processors for each labeled set of processors as follows. For every labeled processor $X$ containing the label and coordinates of a nearest processor $Y$ with a different label, create a sort record containing the label of $X$, the distance to $Y$, the label of processor $Y$, and the snake-like index of $X$. For processors that are not labeled, create dummy sort records. Sort these records by processor labels (the first field of each record), with ties broken (arbitrarily) in favor of minimum distance to a distinctly labeled processor (the second field of each record). The leader (processor containing the first record) of each sorted interval now contains the label of a nearest distinctly labeled set of processors. Within each sorted region, broadcast to all other processors the label contained in the leader. A final sort by the original snake-like index (fourth field of each record) sends the label of a nearest distinctly labeled set of processors back to all labeled processors. Since sorting, broadcasting and reporting within ordered intervals, and row rotations take $\Theta(n)$ time, the running time is as claimed.

Notice that the algorithm can be modified so that in $\Theta(n)$ time each labeled processor finds a farthest distinctly labeled processor. ·

The previous theorem was restricted to finding neighboring information between processors with different labels with respect to the $l_1$ and $l_\infty$ metrics. The following theorem shows that every processor can find a nearest processor of a different label, if such a processor exists, with respect to an arbitrary monotone metric. The algorithm is similar to the previous one in that it simply uses a combination of row and column rotations.

**Theorem 3.29** *In a mesh of size $n^2$, for any monotone metric, in $\Theta(n)$ time every labeled processor can determine the distance and label of a nearest processor with a different label, if such a processor exists.*

*Proof.* Let $P$ be an arbitrary processor and let $Q$ be a nearest labeled processor to $P$ with a different label. Let $R$ be the processor



in $P$'s column and $Q$'s row. Since $d$ is monotone, it must be that $R$ and all processors between $R$ and $Q$ are either unlabeled or have the same label as $P$. This forms the basis of a simple algorithm that is similar to the algorithm of Theorem 3.25. Rotate the labels in every row so that each processor determines the closest (in the $l_1$ sense) labeled processor to its right. (If the processor is itself labeled, then it is the closest labeled processor.) Another rotation is used so that every processor finds the closest labeled processor to its right with a label different from the first found one. This procedure is also performed for the left side. When finished, every processor has determined at most 4 label/coordinate records, which are now rotated within the columns. Each processor determines the minimal distance to a record with a different label, completing the algorithm.

One application of this theorem is to the situation where a digitized black/white image is given in which each white pixel is unlabeled and each black pixel is labeled by its coordinates. In this case, the result gives a solution to the *all-points closest point problem* (*all-points nearest neighbor problem*) [Sham78, Tous80]. Given a digitized picture in which the figures (i.e., connected black components) have been labeled, an application of the previous theorem and a concurrent write can be used to determine the distance between figures, where the *distance between figures* $A$ and $B$ is defined to be $min\{d(P, Q) \mid P \in A, Q \in B\}$.

**Corollary 3.30** *In a mesh of size $n^2$, for any monotone metric, in $\Theta(n)$ time every processor can determine the minimum distance from its figure to a nearest figure.*

Theorem 3.29 can also be applied to solve the *largest empty circle problem* [Sham78] for any monotone metric, in which each processor is marked or unmarked and a processor $P$ must be found which maximizes $min\{d(P, Q) \mid Q$ is marked$\}$, subject to the additional constraint that $P$ must lie in the convex hull of the marked processors. The algorithm that follows introduces a paradigm that exploits the fact that one of the two input sets of data can be drastically reduced. This allows multiple copies of the reduced set to be made available to the other (nonreduced) set of data for processing. Specifically, once the $O(n)$ extreme points of the black processors are identified, $n$ copies of these points can be placed in distinct subsquares of the mesh so that every processor can view all $O(n)$ such points in $O(n)$ time. (Alternately a copy of the $O(n)$ extreme points could be place in every row (column) and a simple row (column) rotation would allow every processor to view all $O(n)$ extreme points.)



**Corollary 3.31** *In a mesh of size $n^2$, for any monotone metric, the largest empty circle problem can be solved in $\Theta(n)$ time.*

*Proof.* Identify the extreme points of the black processors by using the algorithm associated with Theorem 3.19. Compress the coordinates of these points to the upper-left submesh of size $n$. Using a concurrent read, make copies of this data in every disjoint submesh of size $n$. Within each disjoint submesh of size $n$, rotate the extreme points in snake-like order so that all white processors know whether or not they are in the convex hull of the black processors (following the method described in the algorithm associated with Theorem 3.19). Now, use the algorithm of Theorem 3.29 so that each white processor in the convex hull finds its nearest black pixel. Finally, to solve the largest empty circle problem, perform a semigroup (i.e., associative binary) operation over the nearest neighbor information in the white processors that are in the convex hull of the black processors.

Given a nonempty set $S$ of processors, there are several natural definitions for the center of $S$. If $S$ is connected then an *internal center* of $S$ is a processor $P \in S$ which minimizes $max\{d(P, Q) \mid Q \in S\}$, where $d$ is the internal distance. For any metric $d$, a *planar center* of $S$ is a point $x$ in the real plane which minimizes $max\{d(x, Q) \mid Q \in S\}$, and a restricted planar center of $S$ is a processor $P \in S$ which minimizes $max\{d(P, Q) \mid Q \in S\}$. For each definition of center there is also a corresponding definition of radius. For any $l_p$ metric ($1 < p < \infty$), the planar center is unique, but the restricted planar center may not be. For example, the four indicated points in Figure 3.12 are restricted planar centers, and all of the points of the figure are internal centers.

The proof of the following theorem is similar to that of Theorem 3.26 and will be omitted.

**Theorem 3.32** *In a mesh of size $n^2$, for any monotone metric, in $\Theta(n)$ time every labeled processor can determine whether or not it is a restricted planar center among the processors with its label. Further, in $\Theta(n)$ time every processor can determine the restricted planar radius of the processors with its label.*

The following theorem is concerned with determining the Euclidean planar center and radius for all figures in $\Theta(n)$ time on a mesh of size $n^2$. It should be noted that the algorithm that is presented uses facts which are specific to the Euclidean metric.

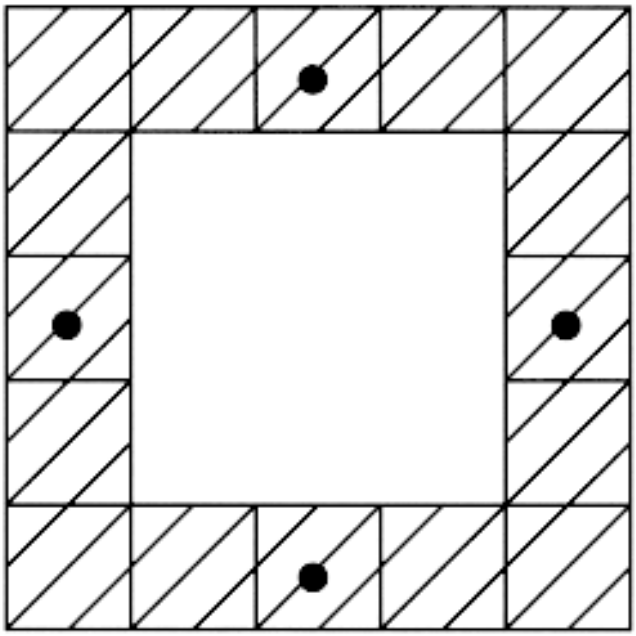


Figure 3.12:
All black (hashed) pixels are internal centers.
The four restricted centers, for any *lp* metric, are also marked.

**Theorem 3.33** *In a mesh of size $n^2$, in $\Theta(n)$ time every labeled processor can determine the Euclidean planar center and Euclidean planar radius of its figure.*

*Proof.* For the Euclidean metric, this problem is known as the *smallest enclosing circle problem* [Sham78]. The following facts and assumptions will be used.

1. If a set has only one or two points, then the smallest enclosing circle can be found in $\Theta(1)$ time if the coordinates of the points are in a single processor.

2. For a set of 3 points, either all 3 points are on the boundary of the smallest enclosing circle, or else 2 of the points form a diameter of the circle. In either case, the center and radius of the circle can be found in $\Theta(1)$ time if the coordinates of the points are in a single processor.

3. For a set $S$ of 3 or more points, there is a 3-element subset $T$ of $S$ such that the smallest enclosing circle of $T$ is the smallest enclosing



circle of $S$. The radius of the smallest enclosing circle of $T$ is the maximum radius of any smallest enclosing circle of a 3-element subset of $S$. Further, $T$ can be taken to be a subset of the extreme points of $S$, except when all of $S$ lies on a straight line, in which case $T$ contains the two endpoints and any third point.

The algorithm is straightforward. In $\Theta(n)$ time, by using the algorithm associated with Theorem 3.19, find the extreme points of every labeled set. Next, for every labeled set $S$, find the smallest enclosing circle for each 3-element subset of the extreme points. Notice that if there are $e$ extreme points of $S$, then this will require $\binom{e}{3} = \Theta(e^3)$ calculations. However, as mentioned on page 124, on a mesh of size $n^2$ the worst-case value of $e$ is $\Theta(n^{2/3})$, which requires only $\Theta(n^2)$ calculations. If these calculations must be done in the e processors, they will require at least $\Omega(n^{4/3})$ time. This is prevented by using the processors of $S$, and not just those that are extreme points of $S$, to perform the calculations. If a figure has $p$ processors, then $e = O(min(r)^{2/3})$, so at most $O(min(p, n)^2)$ calculations are required. By suitably dividing these calculations among the $p$ processors, they can be completed in $O(n)$ time.

Finding the $l_1$ and $l_\infty$ planar radii and planar centers are particularly easy. For these two metrics, the planar radius is half of the diameter, which can be computed in $\Theta(n)$ time by Theorem 3.28. The $l_1$ (and $l_\infty$) planar centers form a straight line segment (see Figure 3.13) which may degenerate to a single point. The details of finding the endpoints of these segments is left to the reader.

The final (distance) problem considered in this section is the *all-points radius query*, also known as the *all-points fixed radius near neighbor problem* [Bent80]. Given a radius $r$, determine for each pixel the number of black pixels at distance $r$ or less. The set of processors at distance $r$ or less from a processor $P$ is called an $r$-ball centered at $P$.

To perform the all-points radius query efficiently, an additional restriction on the metric is imposed. A metric is a *vector metric* if it is monotone and if $d(P, Q)$ is dependent only on the vector from $P$'s position to $Q$'s position. Vector metrics have the property that for any radius $r$ and any processors $P$ and $Q$, the $r$-ball centered at $P$ is just a rigid translation (with no rotation) of the $r$-ball centered at $Q$, i.e., the metric looks the same everywhere. All $l_p$ metrics are vector metrics, and it seems that all metrics encountered in practice are vector metrics.



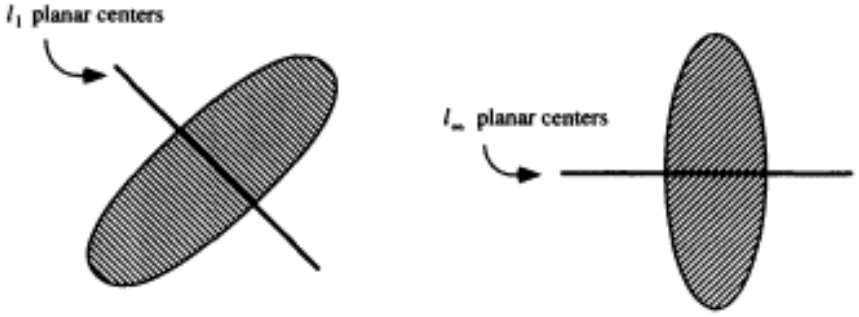



Figure 3.13:
Figures with nonunique planar centers.

**Theorem 3.34** *In a mesh of size $n^2$, for any vector metric and for any radius, the all-points radius query can be solved in $\Theta(n)$ time.*

*Proof*. Suppose the radius $r$ is sufficiently small so that the $r$-ball centered at processor $P_{n/2, n/2}$ lies entirely within the $n \times n$ mesh. The monotonicity guarantees that to traverse the perimeter of the $r$-ball, at most $4n$ processors will be visited. (Figure 3.14 shows a typical $r$-ball.) Suppose each processor has a value, denoted $B$, which is the number of black pixels in its row to its left. Consider a traversal of the perimeter of an $r$-ball during which a running total will be kept. Initially, the total is 0, and as the traversal reaches a processor which is rightmost in its row (among those in the $r$-ball), the $B$ value is added, plus 1 if the pixel there is black. At each processor which is leftmost in its row (among those in the $r$-ball), the $B$ value is subtracted. The total at the end of the traversal is the number of black pixels in the $r$-ball.

Using the above procedure is quite simple. To insure that the traversal does not try to move off of the $n \times n$ mesh, consider the $n \times n$ mesh as being in the center of a $3n \times 3n$ mesh, where all the added pixels are white and each real processor must simulate 9 processors. Further, redefine the $r$-ball centered at a processor $P$ to be $\{Q \mid d(P, Q) \leq r$ and the $l_\infty$ distance from $P$ to $Q$ is $\leq n\}$. Notice that the new $r$-ball centered at a processor in the original mesh lies entirely in the $3n \times 3n$ mesh and contains the same processors of the original mesh as does the original $r$-ball. In particular, it contains exactly the same number of black pixels.

To start, use a row rotation so that every processor determines its $B$ value. Then, all processors in the original mesh create a record which



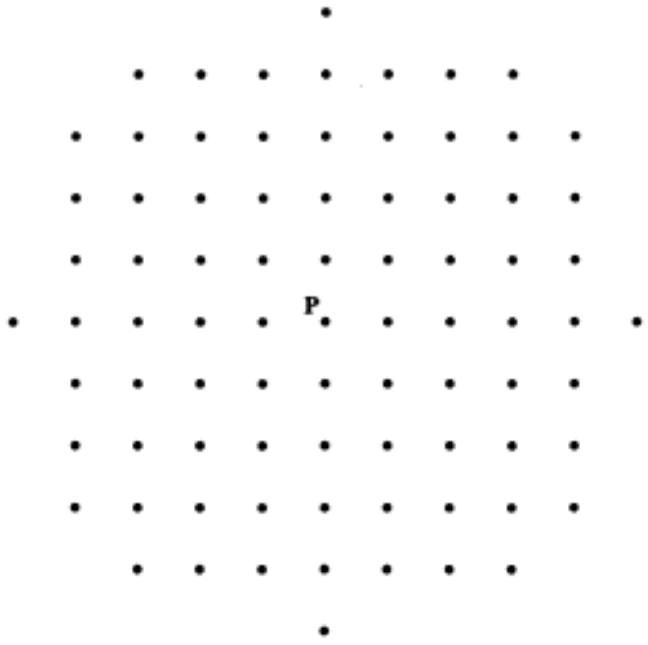



Figure 3.14:
A 5-ball about P, using the Euclidean metric.

acts as their representative in the traversal. Since the $r$-balls are identical, these representatives can be passed along in a lockstep fashion as they perform the traversal and return to their originating processor. No matter what the value of $r$, the modified $r$-ball has a perimeter of $O(n)$, so the algorithm is finished in $\Theta(n)$ time.

## 3.7 Further Remarks

In this chapter, optimal $\Theta(n)$ time algorithms have been presented to solve problems that involve matrices and digitized pictures on a mesh of size $n^2$. The algorithms are defined predominantly in terms of fundamental mesh operations that were introduced in Chapter 2. Many of the problems considered in this chapter involve combining information from processors far apart, in which case the use of sort-like data movements was crucial to the development of efficient algorithms. The general techniques demonstrated in this chapter, including divide-and-conquer, compression, expansion with cross-product, and data reduction techniques that include changing the form of the input, can be applied



to yield efficient solutions to a wide variety of problems on a mesh computer.

In fact, simple techniques, such as propagation, can be used to solve important problems. For example, an optimal mesh solution to the parallel visibility problem can be obtained by propagation. The *parallel visibility problem* can be defined as the problem of determining the portions of each figure that are illuminated by a given light source emitting rays of light parallel to a specified direction $r$. A simple propagation algorithm consists of partitioning the mesh into small strips parallel to $r$ and propagating the light source through each strip. Initially, the entire strip is considered visible. As each processor receives its strip's current visibility interval, it modifies the interval, if necessary (i.e., if the processor contains a black pixel that is still partially visible), and passes the interval on to the next processor in the strip. Details of choosing the proper size of the strips, determining strip predecessor and successor information, and general proofs of correctness are given in [DHSS87]. It should be noted that optimal mesh solutions to visibility problems are possible for assumptions other than a distant light source emitting rays of parallel light.

Dominance problems can also be solved using propagation techniques. Given a digitized picture $A = \{a_{i,j}\}$ stored one pixel per processor on a mesh of size $n^2$ so that processor $P_{i,j}$ contains $a_{i,j}$, pixel $a_{i,j}$ is said to *dominate* pixel $a_{i',j'}$, if and only if $i > i'$ and $j > j'$. A pixel $a_{i,j}$ is called *maximal* if there are no other pixels in $A$ that dominate it. The set of all maximal pixels is called the $1^{st}$ *contour of* $A$, and is denoted $MAX(A)$. The $k^{th}$ *contour of* $A$, denoted $MAX(A, k)$, $k$ an integer, is defined as

$$MAX(A,k) = \begin{cases} MAX(A) & \text{if } k = 1 \\ MAX(A - [MAX(A,1) \cup \ldots \cup MAX(A,k-1)]) & \text{if } k > 1. \end{cases}$$

A straightforward propagation algorithm, where processors send information down and to the left, can be designed to determine the $k^{th}$ contour of $A$ in optimal time on a mesh, simultaneously for all possible values of $k$. See [DHSS91] for details.

Since it takes $\Theta(n)$ time for data to travel across an $n \times n$ mesh, all of the algorithms presented in this chapter have optimal worst-case running times. However, there may be situations where the answer can be found faster. For example, suppose no figure (i.e., connected black component) of a digitized picture has an $l_\infty$ external diameter greater than $D$. Then by partitioning the mesh into disjoint subsquares of size $\Theta(D)$, and sharing data between adjacent squares, in $\Theta(D)$ time every

figure can determine its extreme points. Given the appropriate situation, this technique can be used with all of the image results, reducing them to $\Theta(D)$ time. One particularly interesting application of this technique occurs when it is combined with the algorithms of Theorem 3.34, while using the $l_\infty$ metric and a radius of $D$. In $\Theta(D)$ time, every processor will know the number of black pixels in a square centered at the processor. If a processor then becomes black if and only if more than half of the processors in its square were previously black, then the solution to the *bilevel median filtering problem* with a window of edgelength $2D + 1$ is known in $\Theta(D)$ time. It should be noted that median filtering with a window of edgelength $2D+1$ on an arbitrary greylevel picture can also be accomplished in $\Theta(D)$ time, but the algorithm is far more complicated [Stou83c].

The results of this chapter suggest many additional questions, most of which are still open. For instance, is there a $\Theta(n)$ time algorithm for locating all internal centers? For any $p$, are there $\Theta(n)$ time algorithms for locating $l_p$ planar centers and computing $l_p$ planar radii?

In this chapter, the concentration has been on 2-dimensional meshes. A *j-dimensional mesh of size* $n^j$, $j \geq 2$, has $n^j$ processors arranged in a j-dimensional cubic lattice. Processor $P_{s_1,\ldots,s_n}$ and processor $P_{t_1,\ldots,t_n}$ are connected if and only if $\sum_{i=1}^{n} |s_i - t_i| = 1$. In the $O$-notational analyses of algorithms for $j$-dimensional meshes it makes sense to consider $j$ as fixed. That is, there is no differentiation between a step needing a constant amount of time and one needing $2^j$ units. (A generic processor in a $j$-dimensional mesh has $2^j$ neighbors.) The reason for this is that a processor in a $j$-dimensional mesh is fundamentally different from one in a $k$-dimensional mesh when $j \neq k$.

When considering 3-dimensional (or higher) "pictures," then almost all questions are open. That is, suppose an $n \times n \times n$ (3-dimensional) picture is stored one pixel per processor in an $n \times n \times n$ (3-dimensional) mesh, how fast can figures be labeled, extreme points located, internal distances determined, diameters computed, and so forth? Notice that some methods, such as "shrinking," do not extend to higher dimensions (consider a pair of distinct interlocked solid rings in 3-space). Further, many of the convexity and external distance algorithms reduce the amount of data by one dimension, reducing an $n \times n$ picture to $\Theta(n)$ points, giving $\Theta(n)$ time algorithms. On a $j$-dimensional mesh, this would give $\Theta(n^{j-1})$ time algorithms, which is not necessarily optimal for $j > 2$.

Nassimi and Sahni [NaSa80] have extended their asymptotically optimal component labeling algorithm to all dimensions, and it is not hard to



design a $\Theta(n)$ time algorithm for finding the distance to the nearest processor with a different label, but for most other j-dimensional problems, $j > 2$, $\Theta(n)$ time algorithms are not known. It should be noted that while the first component labeling algorithm given in Section 3.3.2, as well as Nassimi and Sahni's [NaSa80] algorithm, both extend to asymptotically optimal algorithms in higher dimensions, it is not possible to extend the second algorithm given in Section 3.3.2 to higher dimensions. This is due to the fact that the algorithm reduces the digitized picture component labeling problem to the transitive closure problem, the solution of which requires $n^{2(j-1)}$ matrix entries for a picture with $n^j$ pixels. While the $j$-dimensional mesh of size $n^j$ is large enough to hold the matrix for $j = 2$, this is not true for $j > 2$.

Similar space problems arise when attempting to extend the internal distance algorithms given in this chapter to dimension greater than 2. The 2-dimensional algorithms considered $k \times k$ subsquares, ignoring all of the square except for the $\Theta(k)$ border elements, and constructed a distance matrix with $\Theta(k^2)$ entries. This matrix was able to fit in the original square. In 3-dimensions, for example, $k \times k \times k$ subcubes have $\Theta(k^2)$ border elements, which would require a distance matrix containing $\Theta(k^4)$ entries. This matrix will not fit in the original cube, so the method fails, as it would for any $j$-dimensional picture, $j > 2$, with $n^j$ pixels stored in a natural fashion on a $j$-dimensional mesh of size $n^j$. One could use a simple propagation algorithm, where each marked processor informs its neighbors that they are at distance one, each of which informs their neighbors they are at distance two, and so on, but this has a worst-case running time of $\Theta(n^j)$.

The graph algorithms given in Section 3.2 closely follow the solutions originally presented in [AtKo84]. Given an undirected graph $G = (V, E)$ and a directed breadth-first spanning tree $T = (V,A)$ of $G$, then if each vertex $v \in V$ has a data value $v.d$, define the *generalized x function* as a function that returns for every $v \in V$, the value * $\{v'.d \mid v'.d$ is an $x$ of $v$ in $T\}$, where $x$ can be *ancestor*, *descendant*, or *sibling*. Given that $G$ and $T$ are represented as adjacency matrices, as in Section 3.2, simple algorithms may be constructed to compute these functions for all $v \in V$ in $\Theta(n)$ time on a mesh of size $n^2$. Further, by putting together the breadth-first spanning tree algorithm of Section 3.2 with these generalized $x$ functions, many graph algorithms, including some of the ones presented in Section 3.2, can be solved in $\Theta(n)$ time.

The problems considered in Section 3.2 assume that the input is in matrix form. However, the most general form of input for a graph $G = (V, E)$, as defined in Section 1.3, is to allow the edges of $G$ to



be distributed in an arbitrary fashion no more than one per processor in a mesh of size $|E|$. Matrix and image input can be viewed as special cases of this unordered edge input. For unordered edge input, [ReSt] shows how to mark a spanning forest in $\Theta(|E|^{1/2})$ time, from which [Stou85a] shows how to determine whether or not $G$ is bipartite, mark the bridge edges and articulation points of $G$, determine whether or not $G$ is biconnected, and so on, in $\Theta(|E|^{1/2})$ time. Given a tree (or forest) $T = (V, E)$ represented as unordered edges, $\Theta(|E|^{1/2})$ time algorithms are presented in [AtHa85, Stou85a] to determine properties of $T$, such as the height, number of descendants, and preorder number of every node. Notice that if matrix input is given, as in Section 3.2, then $\Omega(|V|)$ time is required to solve any of these problems.

Finally, connections to mesh automata should be mentioned. As was noted in Section 1.2.3, for any given finite state automaton, once the mesh becomes large enough, the individual processors do not have enough memory to store their coordinates, distances to other processors, and so forth. This means that some of the problems solved in this chapter, such as determining the external diameter of each figure, will not map in a straightforward fashion to mesh automata. For example, one may take a black/white picture and want to compute the external diameter of the black pixels, where the answer is emitted by processor $P_{0,0}$ one bit at a time. Except for problems involving internal distances, the image problems considered in this chapter, or an appropriately modified version, can be solved in $\Theta(n)$ time on a mesh automaton by using clerks to simulate the solution given here. (Clerks appear in [Stou82b, Stou83a] and can be viewed as a systematic use of counters.) The problems involving internal distances cause difficulties because the solutions in this chapter create arrays having $\Theta(n^2 \log n)$ bits of information,

which cannot be held in an $n \times n$ mesh automaton. Beyer [Beye69] considered the problem of having a mesh automaton mark a minimal internal path between two given processors in the same figure, and it is still an open question as to whether or not there is a $\Theta(n)$ time solution to this problem.



# 4
# Mesh Algorithms for Computational Geometry

## 4.1 Introduction

The growing field of computational geometry has provided elegant and efficient serial computer solutions to a variety of problems. Particular attention has been paid to determining geometric properties of planar figures, such as determining the convex hull, and to determining a variety of distance, intersection, and area properties involving multiple figures. For a description of problems, efficient serial solutions, and applications of properties in computational geometry, the reader is referred to [PrLe84, PrSh85, Tous80].

Parallel algorithms were presented in Chapter 3 which computed geometric properties of digitized pictures, but such problems are significantly different from the problems that arise when the figures are represented as sets of points or line segments, as is the norm in most of computational geometry. Elegant serial solutions to many problems in computational geometry are based on being able to efficiently construct the planar Euclidean Voronoi diagram of a set of planar points, or use sophisticated data structures specifically designed for geometric problems [Sham78]. Although an optimal mesh algorithm is presented in this chapter for constructing the Voronoi diagram, the algorithms that are presented to solve other geometric problems do not rely on constructing the Voronoi diagram or manipulating sophisticated data structures. Instead, algorithms to solve problems in computational geometry are presented that rely on fundamental data movement operations.

Section 4.2 discusses fundamental data movement operations that are used in this chapter. General descriptions of these operations were given in Chapter 1, and detailed mesh algorithms were given in Chapter 2. However, the algorithms given in Chapter 2 assumed the processors were indexed by a snake-like ordering. In this chapter, many of the algorithms assume that the processors are indexed by a proximity ordering. In some instances, new algorithms for these operations are given that are significantly different from those presented in Chapter 2, in order to accommodate the proximity order index of the processors. The advantages of proximity order indexing are also discussed in Section 4.2.



For most of the problems considered in this chapter, the input is $n$ or fewer planar points, or pairs of points representing line segments or edges, arbitrarily distributed one per processor on a 2-dimensional mesh computer with $n$ processors. Convex figures are represented by the set of their vertices, and simple polygons are represented by the set of their edges. For problems involving multiple figures, each point or edge will have a label identifying its figure.

In Section 4.3, an algorithm is given for finding the convex hull of a set of planar points. In Section 4.4, algorithms are presented for determining smallest enclosing rectangles of sets of points. In Section 4.5, algorithms are presented to solve the all-nearest neighbor problem for a collection of points, to find the minimum distance between two sets of points, and to solve the all-nearest neighbor problem for collections of point sets. In Section 4.6, algorithms are given for finding nearest neighbors of line segments and for deciding whether or not line segments intersect. These algorithms are used to solve several problems involving simple polygons, including deciding whether or not simple polygons intersect and solving the all-nearest neighbor problem for simple polygons if there are no intersections. The algorithms in this section introduce an efficient mesh implementation of *multidimensional divide-and-conquer* [Bent80].

In Section 4.7, algorithms are given for deciding whether or not convex hulls intersect and for finding intersections of convex polygons and hyperplanes. In Section 4.8, the problem of computing the diameter of a set of planar points is considered. In Section 4.9, algorithms are given for determining area and intersection properties of iso-oriented rectangles, and the results are extended to circles and orthogonal polygons.

In Section 4.10, an optimal mesh algorithm is presented to construct the Voronoi diagram of a set of planar points. This construction allows for alternative solutions to many of the problems previously considered in this chapter. Section 4.11 discusses extensions to mesh computers of higher dimensions and to input data of higher dimensions.

It is important to note that in the preceding chapter, an optimal mesh algorithm finished in $\Theta(n)$ time, which is linear in the edgelength of a mesh of size $n^2$. An $n \times n$ mesh was used simply to remain consistent with the literature that typically considers matrices or images to be $n \times n$. In this chapter, however, it is most natural to consider problems involving $n$ objects, distributed one per processor on a mesh with $n$ processors. Therefore, optimal mesh algorithms in this chapter will finish in $\Theta(n^{1/2})$ time, which is (again) linear in the edgelength of the mesh. Except for the extensions in Section 4.11, every algorithm in this chapter finishes



in $\Theta(n^{1/2})$ time. Section 4.11 points out that straightforward changes produce optimal algorithms for meshes of higher dimensions and for some of the problems when the input is from a higher dimensional space.

## 4.2 Preliminaries

For problems in this chapter that involve distances between figures, the term *distance* is used to mean Euclidean distance. It should be noted that in most cases any reasonable metric will suffice. (Metrics were discussed in more detail in Section 3.6.) Let $d(x, y)$ denote the distance between points $x$ and $y$, and define the distance between two sets $S$ and $T$ to be $min\{d(s, t) \mid s \in S, t \in T\}$.

### *4.2.1 Initial Conditions*

For problems in this chapter, the data is initially distributed one piece per processor on a mesh of size $n$. For data involving points, it is typically assumed that no two distinct points have the same $x$-coordinate or $y$-coordinate. For data involving line segments, it is typically assumed that no two endpoints from distinct line segments have the same $x$-coordinate or $y$-coordinate, unless they are from line segments that intersect at an endpoint. These are common assumptions in computational geometry as it simplifies exposition by eliminating special cases. Furthermore, in $\Theta(n^{1/2})$ time, arbitrary input can be rotated to satisfy these assumptions by using sort steps to find the minimum difference in $x$-coordinates between points with different $x$-coordinates, the minimum difference in $y$-coordinates between points with different $y$-coordinates, the maximum difference in $x$-coordinates, and the maximum difference in $y$-coordinates, and then determining a small angle such that rotating by that much will eliminate duplicate coordinates and not introduce new ones.

### *4.2.2 Lower Bounds*

For all problems considered in this chapter, it is easy to create specific arrangements of data so that the solution cannot be obtained faster than the time it takes to combine information starting at opposite corners of the mesh. In a 2-dimensional mesh of size $n$, information starting at opposite corners cannot meet in any processor in less than $n^{1/2} - 1$ time steps. Therefore, all problems considered in this chapter must take $\Omega(n^{1/2})$ time on a mesh of size $n$.



### *4.2.3 Fundamental Operations on the Mesh*

Several of the data movement operations used in this chapter exploit an indexing of the processors based on a proximity ordering. The proximity ordering used in this book combines advantages of other orderings, as shown in Figure 1.2 of Section 1.2.3 on page 8. Proximity order is based on the concept of space-filling curves (c.f., Section 3.3 of [Wirt86]), in particular the *Peano-Hilbert scan curve* [KoVa79, LeZi86]. Notice that snake-like ordering has the useful property that processors with consecutive numbers in the ordering are adjacent in the mesh, while shuffled row-major ordering has the property that the first quarter of the processors form one quadrant, the next quarter form another quadrant, and so forth, with this property holding recursively within each quadrant. This property of shuffled row-major ordering is useful in many applications of a divide-and-conquer solution strategy.

Proximity ordering combines the advantages of the snake-like and shuffled row-major orderings. Given row and column coordinates of a processor $P$, in $O(\log n)$ time a single processor can compute the proximity order of $P$ by a binary search technique. Similarly, given a positive integer $i$, the row and column coordinates of processor $P_i$, that is, the processor with $i$ as its proximity order index, can be determined in $O(\log n)$ time by a single processor. Given any positive integers $i < j$, the shuffled row-major property of recursively dividing indices among quadrants gives the property that the distance from processor $P_i$ to processor $P_j$ is $O((j - i)^{1/2})$, and that a path of length $O((j - i)^{1/2})$ can be achieved using only processors numbered from $i$ to $j$. Further, the processors numbered from $i$ through $j$ contain a subsquare with more than $(j - i)/8$ processors.

The implementations for some of the data movement operations used in this chapter are altered in order to accommodate the proximity ordering of the processors. These operations are described in detail, while other necessary operations are briefly reviewed.

Many of these data movement operations will be performed in parallel on items stored in disjoint consecutively numbered (with respect to proximity ordering) processors, which will be referred to as (*ordered*) *intervals*. It should be noted that ordered intervals may be created by sorting data into proximity order so that related items reside in disjoint consecutively indexed processors.

1. *Sorting:* In Section 2.6.1, it was shown that $n$ elements, distributed one per processor on a mesh computer of size $n$, can be sorted into any predefined linear order in $\Theta(n^{1/2})$ time. It should be noted



that an algorithm that directly sorts into proximity order can be faster by a multiplicative factor. While such an algorithm would be useful, this does not affect the analysis of algorithms presented in this chapter.

2. *Broadcasting* and *Rotating Data within Intervals:* Suppose each processor contains a record with data, a label, and a Boolean flag called 'marked'. Further, assume that all processors containing records with the same value in the label field form an *ordered interval* with respect to the proximity ordering. Then the data in all records with marked = *true* can be sent to all other processors holding records with the same label in

$$O(max\{m(r) + i(r)^{1/2} \mid r \text{ a label}\})$$

time, where $m(r)$ is the number of marked records with label $r$, and $i(r)$ is the number of records with the label $r$. This is accomplished by building a breadth-first spanning tree, level by level, within each ordered interval, and then using this spanning tree to perform the desired data movement operation. It is first shown how to construct the breadth-first spanning tree within every ordered interval and then how to use the spanning tree to perform the desired data movement operations.

At time 0, the processor corresponding to the *root* of every tree is identified, with the root of a tree being defined to be at level 0. This is accomplished in $\Theta(1)$ time by having every processor $P_i$ examine the label of processor $P_{i-1}$, where the indices are with respect to the proximity order of the processors, and having processor $P_i$ identify itself as the root of the tree for its ordered interval of labels if the label of processor $P_{i-1}$ is different from the label of processor $P_i$. At time 1, the root of every tree sends a message to all of its neighbors with the same label informing them that it is their parent. The root records the identity of these processors as its children, and these neighbors record the identity of the root as their parent, as well as the fact that they are at level 1 of the tree.

At time $t$, processors at level $t - 1$ send a message to all neighbors with the same label that have not yet recorded a level. Each processor receiving one or more such messages at time $t$ records the fact that it is at level $t$ in the breadth-first spanning tree of its label. Each processor receiving one or more such messages also

picks one of the senders as its parent, records the identity of this chosen parent processor, and sends a message back to the chosen parent processor so that processors at level $t$ - 1 in the breadth-first spanning tree can record the identity of their children in this tree. Notice that the height of a breadth-first spanning tree for processors with label $r$ is $\Theta(i(r)^{1/2})$. Therefore, the breadth-first spanning tree for the processors labeled $r$ is constructed in $\Theta(i(r)^{1/2})$ time, since each step of the level by level construction takes $\Theta(1)$ time.

Once the spanning tree is constructed, the marked data is passed up the tree until it reaches the root, at which point it is passed down, with each parent passing the data item to all of its children. Using simple pipelining, the first item reaches all processors in its interval in $\Theta(i(r)^{1/2})$ time, and each subsequent item follows in $\Theta(1)$ time.

For the situation where one piece of data is circulated within each ordered interval, this operation is referred to as *broadcasting within* (*ordered*) *intervals*. For the situation where multiple pieces of data are circulated within ordered intervals, this operation is referred to as *rotating data within* (*ordered*) *intervals*.

3. *Reporting* and *Semigroup Computation within Intervals:* Suppose each processor has a record with data and a label, and all records with the same label form an ordered interval. Further, suppose a unit-time semigroup operation (i.e., an associative binary operation such as minimum, summation, or parity) is to be applied to all data items with the same label, with all processors receiving the answer for its label. Then this can be accomplished in $\Theta(max\{i(r)^{1/2} \mid r \text{ a label}\})$ time, where $i(r)$ is the number of records with label $r$. This is performed by forming a breadth-first spanning tree within every ordered interval, followed by having the leaves start passing their values up, where once a processor receives values from all of its children, it applies the semigroup operation to these values and its own, and passes the result up to its parent. Once the root processor of the spanning tree has computed the answer, the spanning tree is used to broadcast it to all processors in the interval.

The first phase of the semigroup operation that combines data to the root of the spanning tree within each interval is referred to as *reporting within* (*ordered*) *intervals*. Therefore, as discussed



in previous chapters, a *semigroup operation within intervals* can be viewed as a report followed by a broadcast within intervals.

4. *Concurrent Read and Concurrent Write:* The implementation of the concurrent read and concurrent write operations remains as before (c.f., Section 2.6.4), except that the sorting is performed with respect to the proximity order index. Assuming that each processor creates a fixed number of master records and a fixed number of request or update records, depending on whether the operation is a concurrent read or concurrent write, respectively, the concurrent read and concurrent write operations can be completed in $\Theta(n^{1/2})$ time on a mesh of size $n$.

5. *Compression:* Suppose that on a mesh of size $n$, $m$ pieces of data are randomly distributed one element per processor. Further, suppose that it is desirable to minimize the interprocessor communication time between the processors that contain these $m$ pieces of data. Then in $\Theta(n^{1/2})$ time, this information can be moved to a subsquare of size $\Theta(m)$, where the communication diameter is $\Theta(m^{1/2})$. The algorithm to perform this operation is as described in Section 2.6.5, with the exception that the sorting is done with respect to the proximity order index. Alternately, each processor containing one of the $m$ pertinent pieces of data may place the piece of data augmented with a key of 1 into its sort field, while all other processors place a dummy data entry with a key of $\infty$ into their sort field. After the sort fields are filled, the data is simply sorted into proximity order. This directly moves the $m$ pieces of data into a subsquare of size $\Theta(m)$ in $\Theta(n^{1/2})$ time.

6. *Searching and Grouping:* The searching problem is defined in Section 1.5, as are a variety of solutions to the problem that involve the grouping operation. The grouping operation will be used extensively in this chapter to solve a variety of search problems. Since the multiple parallel binary search, as well as both the one- and two-pass grouping operations, are described in terms of sorting, concurrent reads, concurrent writes, and operations within intervals, the searching problem can be solved in $\Theta(n^{1/2})$ time on a mesh of size $n$. Notice that for the two-pass algorithm, the parameter $k$ needs to be chosen appropriately. In this chapter, $k$ will typically be chosen to be $\Theta(n^{1/2})$.



## 4.3 The Convex Hull

The convex hull, a geometric structure of primary importance, has been well studied for the serial model of computation [PrSh85, Sham78, Tous80, Avis79, Yao81]. It has applications to normalizing patterns in image processing, obtaining triangulations of sets of points, topological feature extraction, shape decomposition in pattern recognition, and testing for linear separability, to name a few.

In this section, an asymptotically optimal $\Theta(n^{1/2})$ time algorithm is presented for marking the extreme points that represent the convex hull of a set of $n$ or fewer planar points, initially distributed one point per processor on a mesh of size $n$. The *convex hull* of a set $S$ of points, denoted *hull*($S$), is the smallest convex polygon $P$ for which each point of $S$ is in the interior or on the boundary of $P$, as shown in Figure 4.1. A point $p \in S$ is defined to be an *extreme point* of $S$ if $p \notin \text{hull}(S - \{p\})$. That is, $p$ is an extreme point of *hull*($S$) if and only if $p$ is on the boundary of *hull*($S$) at a point where a trace of the boundary results in a change of slope (i.e., $p$ is situated at a corner of the boundary). So, if $S$ is finite, then *hull*($S$) is a convex polygon, and the extreme points of $S$ are the corners of this polygon.

For several of the algorithms presented in this chapter, it will be useful to impose an ordering on the extreme points of $S$. The ordering will be in a counterclockwise fashion, starting with the easternmost point. (Recall from Section 4.2.1, that since the number of points is finite and no two points have the same $x$-coordinate, there must be a unique easternmost point.)

The edges of the convex hull of $S$ will be referred to as the *edges of the hull*($S$). In addition, it is said that *the extreme points of S have been identified*, and hence *hull*($S$) has been identified, if for every processor $P_i$ containing a point $p \in S$, the following hold.

1. $P_i$ has a Boolean variable 'extreme', and extreme is true if and only if $p$ is an extreme point of $S$.

2. $P_i$ stores the total number of extreme points of *hull*($S$).

3. If $p$ is an extreme point of $S$, then $P_i$ stores the position of $p$ in the counterclockwise ordering of extreme points.

4. If $p$ is an extreme point of $S$, then $P_i$ stores the Cartesian coordinates of the extreme points that precede and succeed $p$, as well as the ID of the processors that contain them.



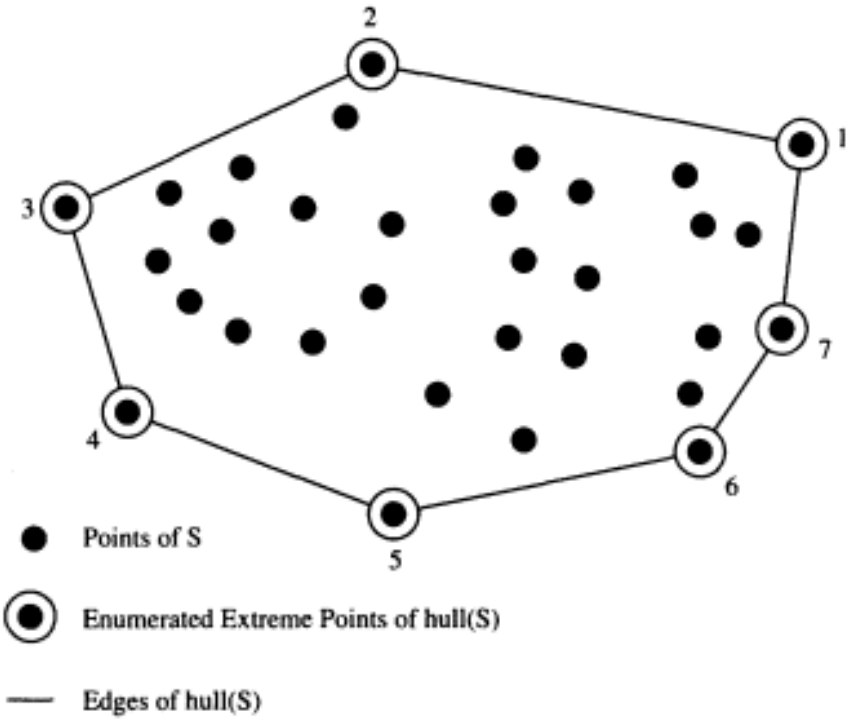



Figure 4.1:
Convex hull of $S$.

The mesh algorithm presented in this section to solve the convex hull problem follows directly from the generic Fixed Subset Division Algorithm of Section 1.6.2. The first (preprocessing) step of the algorithm is to sort the planar point data into proximity order by $x$-coordinate. After this step, the $x$-coordinates of all points in quadrant $q$ are less than the $x$-coordinates of all points in quadrant $q + 1$, $1 \leq q \leq 3$, with this property holding recursively within each quadrant. Next, the convex hull is determined simultaneously and recursively for each of these four sets of points. Finally, these linearly separable convex hulls are combined to form the convex hull of the entire set by determining the upper and lower common tangent lines between pairs of linearly separable convex hulls, and eliminating the points on the inside of the quadrilateral formed by these two tangent lines.

**Theorem 4.1** *Given a set $S$ of $n$ or fewer planar points, distributed one per processor on a mesh computer of size $n$, the extreme points of $S$ can be identified in $\Theta(n^{1/2})$ time.*

*Proof.* An algorithm to determine the extreme points of $S$ follows.

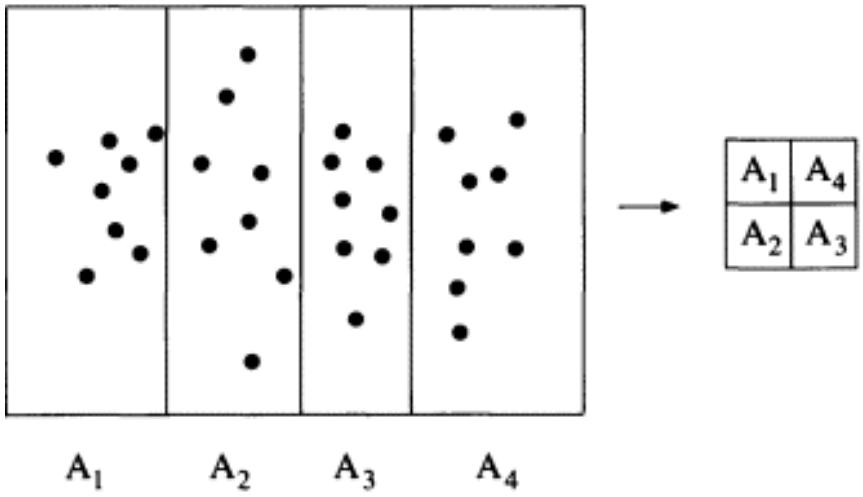



Figure 4.2:
Mapping points into the proper quadrants.

Initially, 'extreme' will be set to true for all points. As it is determined that a point is not an extreme point, this flag will be set to false.

1. *Preprocessing:* Sort the points into proximity order using the $x$-coordinate as the major key and the $y$-coordinate as the minor key.

2. If $n$, the number of points in the subsquare under consideration, is less than or equal to 2, then the convex hull of the points is determined in constant time. Otherwise, recursively solve the convex hull problem for the points in quadrants $A_1$, $A_2$, $A_3$, and $A_4$, of the subsquare under consideration. (See Figure 4.2.) Note that this is a recursive call to Step 2 and not Step 1.

3. From $hull(A_1)$ and $hull(A_2)$, identify $hull(A_1 \cup A_2)$. Denote the set of extreme points representing $hull(A_1 \cup A_2)$ as $B_1$.

4. From $hull(A_3)$ and $hull(A_4)$, identify $hull(A_3 \cup A_4)$. Denote the set of extreme points representing $hull(A_3 \cup A_4)$ as $B_2$.

5. From $hull(B_1)$ and $hull(B_2)$, identify $hull(B_1 \cup B_2)$.

Notice that in steps 3, 4, and 5, the convex hulls of two sets of points, say $A$ and $B$, are used to identify $hull(A \cup B)$. In each of these steps,



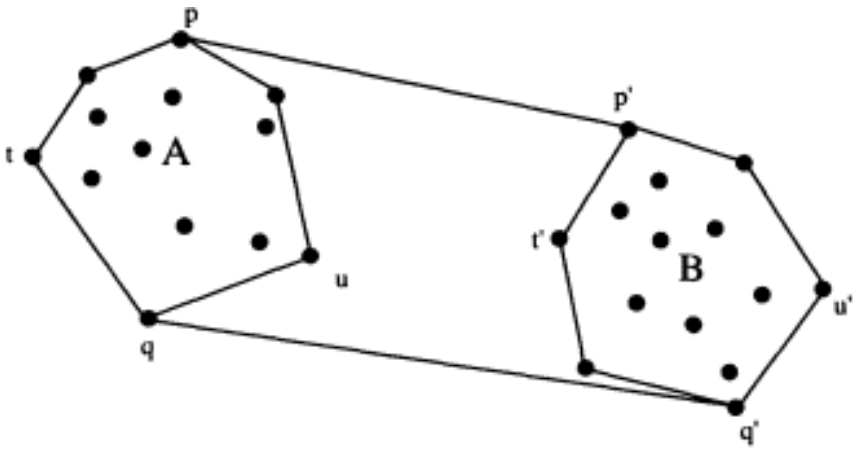
$A$ and $B$ can be picked so that $hull(A)$ lies to the left of $hull(B)$, and $hull(A)$ does not intersect $hull(B)$. This is due to the partitioning of points from step 1. An explanation of how to identify $hull(A \cup B)$ from $hull(A)$ and $hull(B)$ follows. (Refer to Figures 4.2 and 4.3.)

Without loss of generality, assume that the points of $A$ are in quadrant $A_1$, the points of $B$ are in quadrant $A_2$, and *hull*($A$) lies to the left of *hull*($B$). The most crucial phase of the algorithm is the identification of points $p, q \in$ *hull*($A$) and $p', q' \in$ *hull*($B$), such that $\overline{pp'}$ and $\overline{qq'}$ represent the upper and lower common tangent lines, respectively, between *hull*($A$) and *hull*($B$). (See Figure 4.3(a).) Let $t, u \in$ *hull*($A$) be the westernmost and easternmost extreme points of *hull*($A$), respectively. Then $p$ must lie on or above the line $\overline{tu}$, otherwise $\overline{pp'}$ would intersect *hull*($A$). Let $x, y \in$ *hull*($A$) be the extreme points immediately succeeding and preceding (in the counterclockwise ordering of extreme points) $p$, respectively. Referring to Figure 4.3(b), all points in *hull*($B$) must lie below $\overline{xp}$ and some points in *hull*($B$) must lie above $\overline{py}$. (Similar remarks can be made about the points $p'$, $q$, and $q'$.) The details of identifying the extreme points $p$ and $p'$ by a *binary search* technique, as described in step 4a of the Fixed Subset Division Algorithm on page 37, are now given. (A similar technique is used to identify $q$ and $q'$.)

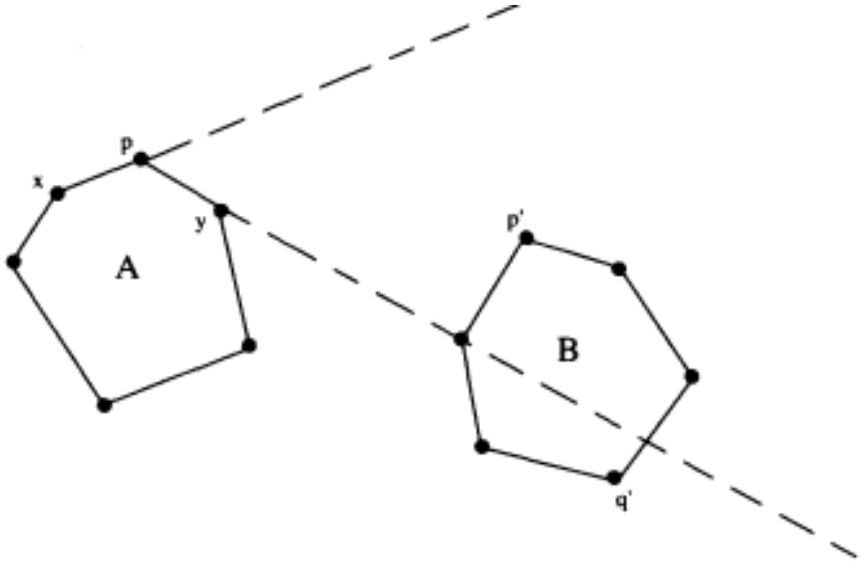
In $\Theta(n^{1/2})$ time, every processor of $A_1$ can know the Cartesian coordinates of $t$ and $u$, as well as the position of $t$ and $u$ in the counterclockwise ordering of the extreme points of $A$. Initially, every processor in $A_1$ containing an extreme point $a \in$ *hull*($A$) creates a record with the $x$-coordinate of $a$ as key, and the $y$-coordinate of $a$ and counterclockwise order of $a$ in *hull*($A$) as data. Then, a pair of semigroup (i.e., associative binary) operations is performed over these records so that all processors of $A_1$ know the identity of the easternmost and westernmost points of $A$, as well as their counterclockwise order. Without loss of generality, assume that point $u$ is numbered $n_0$, and point $t$ is numbered $n_1$, with respect to the counterclockwise ordering of extreme points of $A$.

Next, every processor in $A_1$ that contains an extreme point of *hull*($A$) decides if its point is above the line $\overline{tu}$. Notice that all such points above the line $\overline{tu}$ are numbered in counterclockwise order $n_0 + 1, n_0 + 2, \ldots, n_1 - 2, n_1 - 1$. The processor in $A_1$ containing the point above $tu$ and half way between $t$ and $u$ (i.e., the point numbered $\lfloor \frac{n_1+n_0}{2} \rfloor$), identifies this point as $p$. A $\Theta(n^{1/2})$ time semigroup operation is used to broadcast $p$ to all processors of $A_1$. The processors containing the succeeding and preceding neighbors of $p$ (in the counterclockwise ordering) create the equations of lines $\overline{xp}$ and $\overline{py}$, respectively. Similar computations in $A_2$ identify $p'$, $\overline{p'x'}$, and $\overline{y'p'}$ for $B$.






(a) Identifying the upper and lower common tangent lines between *hull*($A$) and *hull*($B$).




(b) All points in *hull*($B$) must lie below $\overline{xp}$ . Some points in *hull*($B$) must lie above $\overline{py}$.

Figure 4.3:
Stitching convex hulls together.



The coordinates of $\overline{xp}$ and $\overline{py}$ are broadcast from $A_1$ to those processors in $A_2$ that contain a point above $\overline{t'u'}$. This broadcast can be accomplished via a concurrent read. Next, every processor in $A_2$ that just received data decides whether or not its point is *i*) below $\overline{xp}$ and *ii*) above $\overline{py}$. By performing a concurrent write, this information can be collated and routed to the processor in $A_1$ that contains the point $p$. This processor can now determine if $\overline{xp}$ is above all of the extreme points in $B$, and if $\overline{py}$ is below some of the extreme points of $B$. If both conditions are satisfied, then $p$, $x$, and $y$ have been identified. If these conditions are not satisfied, then if $\overline{xp}$ is not above all of the extreme points of $B$, then assign the point $x$ to $u$, recompute $p$ as the point half way between $t$ and the new $u$, compress the data, and iterate the algorithm. If $\overline{xp}$ is above all of the extreme points of $B$, then assign the point $y$ to $t$, recompute $p$, compress the data, and iterate the algorithm. (The corresponding computations for $p'$, $q$, and $q'$ are similar.)

After O(log $n$) iterations, $p$, $p'$, $q$, and $q'$ will be identified since each iteration of the binary search for $p$ and $p'$ eliminates half the points in *hull*($A$) and half the points in *hull*($B$) from further inspection. At the end of each iteration of the binary search, the remaining data from both *hull*($A$) and *hull*($B$) is compressed into the smallest square set of processors that will hold this data. The $i^{th}$ iteration of the algorithm operates on $O(n/2^i)$ pieces of data at communication diameter $\Theta((n/2^i)^{1/2})$. Therefore, the $i^{th}$ iteration of the algorithm finishes in $\Theta((n/2^i)^{1/2})$ time. Notice that if the remaining data from *hull*($A$) was compressed to the smallest square set of processors in the upper-left corner of $A_1$ and the remaining data from *hull*($B$) was compressed to the smallest square set of processors in the upper-left corner of $A_2$, for example, then the during the $i^{th}$ iteration of the algorithm the remaining $O(n/2^i)$ pieces of data would be at communication diameter $\Theta(n^{1/2})$, and hence every iteration of the binary search would take $\Theta(n^{1/2})$ time. With the joint compression of data after each iteration of the algorithm, the time for the binary search to identify the desired points $p$, $p'$, $q$, and $q'$ is given by $\sum_{i=0}^{O(\log n)} \Theta((n/2^i)^{1/2})$, which is $\Theta(n^{1/2})$. Also, notice that the correctness of this binary search hinges on the fact that the points $p$ and $q$, for example, are never eliminated during the search and compression operation.

Finally, the positions of the extreme points of $hull(A \cup B)$ must be computed. First, a concurrent read is performed so that all processors know the number of points in $hull(A)$, the number of points in $hull(B)$, the position of $p$ and $q$ in $hull(A)$, and the position of $p'$ and $q'$ in $hull(B)$. Every processor can now compute the correct position of its



extreme point with respect to $hull(A \cup B)$, if indeed its point is an extreme point of $hull(A \cup B)$. Concurrent reads can be used so that every processor knows the total number of extreme points, as well as the extreme points that are adjacent to the extreme point that it contains. Therefore, the time to identify $hull(A \cup B)$ on a mesh of size $n$ from the extreme points of linearly separable sets $A$ and $B$, is dominated by the $\Theta(n^{1/2})$ time binary search procedure and the $\Theta(n^{1/2})$ time data movement operations used to compute the final position information.

The preprocessing sort step (step 1 of the algorithm, as described at the beginning of the proof) takes $\Theta(n^{1/2})$ time. Therefore, the running time of the algorithm is given by $T(n) = \Theta(n^{1/2}) + T'(n)$, where $T'(n)$ is the time required for the remaining steps of the algorithm. As described, steps 3, 4, and 5 each take $\Theta(n^{1/2})$ time. (Notice that steps 3 and 4 can be performed simultaneously.) Since Step 2 is a recursive call, steps 2 through 5 obey the recurrence $T'(n) = T'(n/4) + \Theta(n^{1/2})$, which is $\Theta(n^{1/2})$. Therefore, the running time of the entire algorithm is $\Theta(n^{1/2})$.

As an alternative to the binary search, a one pass grouping operation, as described on page 39 in step 4b of the Fixed Subset Division Algorithm, may be used to identify $p$, $p'$, $q$, and $q'$ in $\Theta(n^{1/2})$ time. Notice that the angles of incidence (refer to the definition on page 20) of the hull edges are monotonic with respect to both the *upper envelope* (i.e., the portion of the convex hull above the line determined by the westernmost and easternmost extreme points) and the *lower envelope* (i.e., the portion of the convex hull below the line determined by the westernmost and easternmost extreme points). Therefore, a fixed number of sort-like operations to create ordered intervals (groups), followed by a pipelined broadcast within intervals, followed by a sort-based operation to send the results back, will also solve the search problem for the upper and lower common tangent lines in $\Theta(n^{1/2})$ time.

Suppose that instead of being given a single set $S$ comprised of $n$ or fewer points, the input to the convex hull problem is $n$ or fewer *labeled* points representing multiple sets. If there are only a fixed number of labels, say $L$, then the previous algorithm could be performed $L$ times, once for each labeled set, and still enumerate the extreme points of every set in $\Theta(n^{1/2})$ time. If the relationship between $L$ and $n$ is not known, then a minor modification can be made to the previous algorithm so that work is done simultaneously on distinctly labeled sets of points. This modification consists of initially sorting data with respect to the label of the points.



**Corollary 4.2** *Given n or fewer labeled planar points, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time the extreme points of each labeled set can be identified.*

*Proof*. The previous algorithm needs to be modified only slightly. Modify the first sentence of step 1 to read, 'Sort the $n$ points using the label as primary key, the $x$-coordinate as secondary key, and the $y$-coordinate as tertiary key.' As was noted in Section 4.2.3, if a given label has $m$ points, then those points are now in an interval of processors which contains a square of size greater than $m/8$. The preceding algorithm is then executed inside each such square, where the processors in such a square simulate 16 processors of the original algorithm. (This last point insures that each simulated processor has at most one point.) ·

Once the extreme points of the convex hull have been identified, several properties of the convex hull can be quickly determined. For example, the area of the convex hull may be determined by triangulating the convex hull with respect the extreme points and a distinguished extreme point, as shown in Figure 4.4, computing the areas of the triangles, and finally summing over these areas. The centroid of the convex hull can be computed by similar local operations over these triangles, and the perimeter of the convex hull can be determined by simply summing the lengths of the hull edges.

**Corollary 4.3** *Given n or fewer labeled points, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time the area, perimeter, and centroid of the convex hull of each labeled set can be determined.*

*Sketch of Proof*. The area of the convex hull of every labeled set of points is computed as follows. Use the algorithm associated with Corollary 4.2 to determine the extreme points of every labeled set of points. Use sorting to gather all points with the same label together, where sorting is performed so that within each labeled set, all extreme points will be stored in counterclockwise fashion before all points interior to the convex hull. For each labeled set, A broadcast within ordered intervals can be used to send the easternmost extreme point $p_e$ of a labeled set to all processors containing points of the set. Every processor in the interval containing an extreme point $p_i$, computes the area of the triangle $p_i p_e p_{i+1}$, as shown in Figure 4.4. A semigroup (i.e., associative binary) operation within ordered intervals allows each processor to know the total area of the convex hull of the points with its label, and a concurrent write sends all points back to the processors where they initially resided,



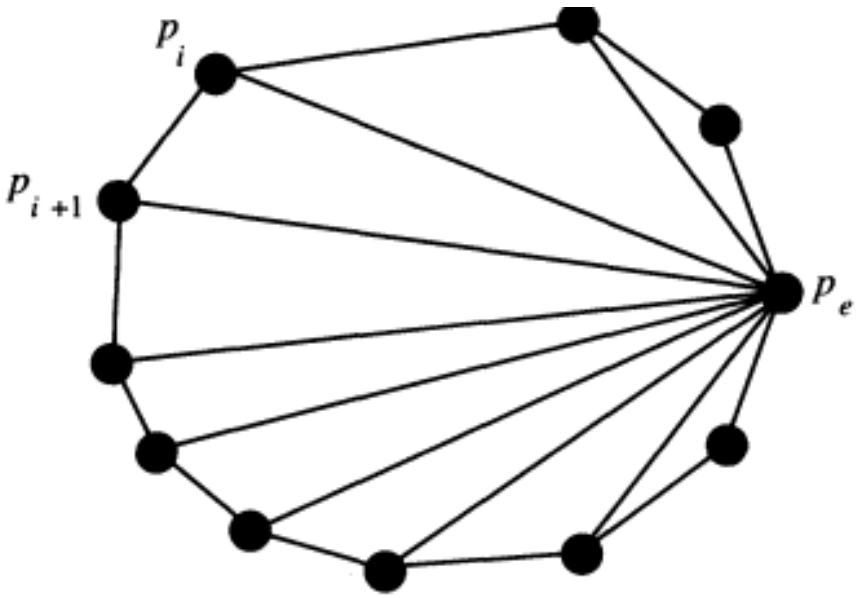



Figure 4.4:
Computing the area of a convex hull.

along with the total area of the labeled set that the point is a member of.

The perimeter of the convex hull for every labeled set of points is computed simply by determining the extreme points of each labeled set of points, gathering labeled sets of extreme points together, using a semigroup operation within ordered intervals to sum the lengths of the line segments $\overline{p_i p_{i+1}}$, for all extreme points $i$ in a labeled set, and then using a concurrent read to send the results back to the original processors.

The $x$-coordinate of the centroid of a figure is the total $x$-moment divided by the area, and the $y$-coordinate is the total $y$-moment divided by the area. To determine the centroid of each convex hull, form the triangles as in Figure 4.4, determine their moments and areas, and then add them to determine the moments and areas of the entire convex hull.

### 4.4 Smallest Enclosing Figures

Problems involving smallest enclosing figures have been studied extensively [Tous80, FrSh75, GVJK]. For certain packing and layout prob-



lems, it is useful to find a *minimum-area rectangle* (*smallest enclosing box*) that encloses a set $S$ of planar points. (Notice that while the area of this rectangle is unique, the rectangle itself need not be.) The algorithm presented in the proof of Theorem 4.4 exploits the facts that

1. any enclosing rectangle of $S$ must enclose $hull(S)$, and

2. a smallest enclosing box of $S$ must have one side collinear with an edge of the convex hull of $S$, and each of the other three sides must pass through an extreme point of $hull(S)$ [FrSh75].

After enumerating the extreme points representing the convex hull of $S$, the algorithm relies on one pass grouping operations so that for each hull edge $e$, a minimum enclosing box with one edge of the box containing $e$ is determined. The grouping operation follows the algorithm given in Section 1.5, operation 8a, on page 27. The final step of the algorithm uses a semigroup (i.e., associative binary) operation to determine the minimum area over all such boxes.

**Theorem 4.4** *Given a set $S$ of $n$ or fewer planar points, distributed one per processor on a mesh computer of size $n$, in $\Theta(n^{1/2})$ time a smallest enclosing box of $S$ can be identified.*

*Proof.* An algorithm for finding a smallest enclosing box of $S$ follows.

1. Identify $hull(S)$. Let $l$ represent the number of edges in $hull(S)$.

2. For each edge $e_i \in hull(S)$, $1 \le i \le l$, determine the minimum area enclosing box of $S$ that has one side collinear with $e_i$. Denote this box as $B_i$.

3. A smallest enclosing box of $S$ is $B_k$, where

$$area(B_k) = min\ area(B_i) \mid 1 \le i \le l\}.$$

The extreme points of *hull*(*S*) can be identified in $\Theta(n^{1/2})$ time by using the algorithm associated with Theorem 4.1. When the algorithm of Theorem 4.1 terminates, every processor containing an extreme point $x$ of $S$ also contains the preceding extreme point $w$ and the succeeding extreme point $y$, with respect to the counterclockwise ordering of extreme points of $S$. Each such processor now creates the hull edge $\overline{xy}$ of *hull*(*S*). In order to determine the minimum-area rectangle associated with such an edge, every processor containing an edge $\overline{xy}$ needs

Page 164

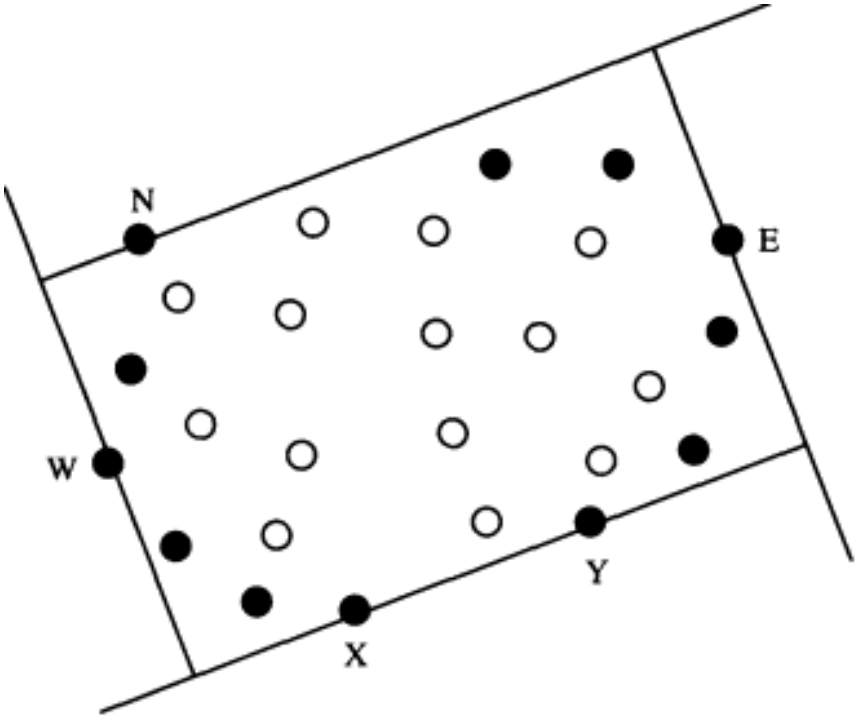



Figure 4.5:
Determining a smallest enclosing box.

to know three additional extreme points of the *S*. These points are *N*, the last extreme point of *S* encountered as a line parallel to $\overline{xy}$, starting collinear to $\overline{xy}$, passes through the *hull*(*S*); *W*, the last extreme point of *S* encountered as a line perpendicular to $\overline{xy}$ passes through all of the points of *hull*(*S*) from right to left (viewing $\overline{xy}$ as the southernmost edge of *hull*(*S*)); and *E*, the last extreme point of *S* encountered as a line perpendicular to $\overline{xy}$ passes through *hull*(*S*) from left to right. (See Figure 4.5.)

Each processor containing a hull edge $\overline{xy}$ can find the necessary points, *N*, *W*, and *E* simultaneously in $\Theta(n^{1/2})$ time by participating in a one pass grouping operation based on *angles of incidence* (*AOI*), as defined on page 20. The following is a description of how each such processor can determine its associated point *N* (with the determination of *W* and *E*, respectively, being similar). Every processor $P_i$ that is responsible for a hull edge $\overline{xy}$, creates a *query record* with the key defined as the angle $AOI(\overline{xy}) + \pi$ ($AOI(\overline{xy}) + 3\pi/2$ in the case of searching for *W*, and $AOI(\overline{xy}) + \pi/2$ in the case of searching for *E*) and with the index $i$ (i.e., the proximity order index of the processor) as data. Room

Page 165

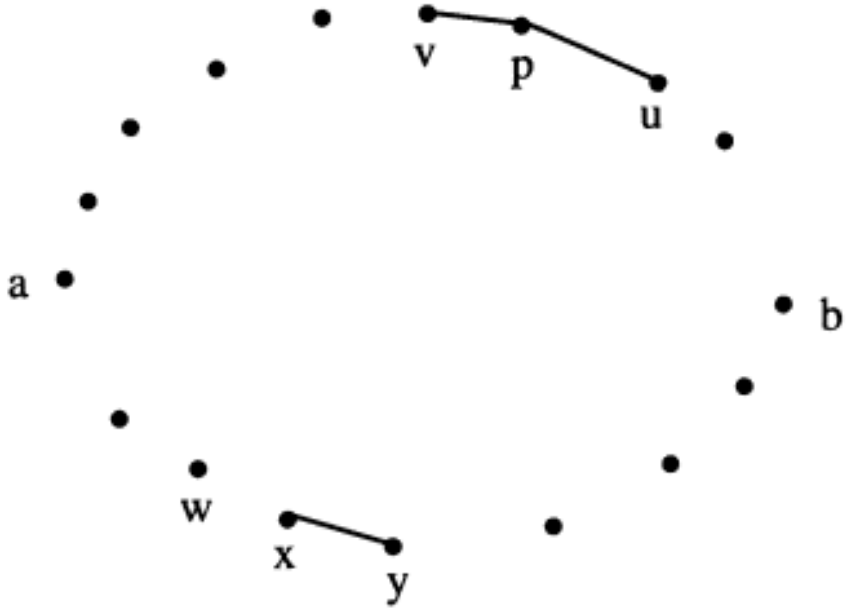



Figure 4.6:
Creating the *slope* and *interval* records.

is left in this record for a description of the point $N$ ($W$, $E$, respectively) that is to be determined. Every processor $P_j$ that is responsible for an extreme point $p$ in $hull(S)$, and contains the preceding extreme point $u$ and the succeeding extreme point $v$, with respect to the counterclockwise ordering of extreme points of $hull(S)$, creates a *master record* with the angle of incidence of $\overline{pu}$ as key, and $p$, $u$, $v$, and $j$ as data. (For technical reasons that will become apparent, the processor responsible for the extreme point with angle 0 contained in its angles of support, creates two additional master records, one with a key of 0 and one with a key of $2\pi$.) See Figure 4.6.

Sort the master and query records together by the key field, with ties broken in favor of master records. After sorting, every processor $P_i$ that contains a master record $(AOI(\overline{pu}), p, u, v, j)$ participates in a concurrent read so as to obtain the proximity order index of the processor containing the next master record. Notice that the key fields of every consecutive pair of master records correspond to the angles of support of the common extreme point. Given two consecutive master records stored in processors $P_i$ and $P_j$ , with $i < j$, the set of processors $P_i \ldots P_{j-1}$ are referred to as a *group*, and $P_i$ is considered to be the *leader* of



the group. Perform a group-restricted broadcast of the extreme point common to the pair of master records for the purpose of notifying all request records as to their desired point $N$ ($W$, $E$, respectively). This broadcast operation is almost identical to broadcasting within proximity ordered intervals, except that after the leader of a group initializes the creation of the breadth-first spanning tree within its group, data is only passed to those processors that are in the group. This can be done since the leader knows the index of the first processor (itself) of the group and the last processor of the group (found during the concurrent read).

A concurrent read completes the operation so that every processor containing a hull edge knows the coordinates of the appropriate $N$ ($W$, $E$, respectively). In 0(1) time, every processor can compute the area of the rectangle formed by its edge $xy$ and the three corresponding points, $N$, $W$, and $E$. Once a minimum area rectangle has been determined for every hull edge, a smallest enclosing box of $S$ can be determined by taking a minimum over the areas of these rectangles in $\Theta(n^{1/2})$ time. Therefore, the entire algorithm requires $\Theta(n^{1/2})$ time. ·

Simple modifications to the algorithm associated with Theorem 4.4, as in Corollary 4.2, allow labeled sets of points to be considered.

**Corollary 4.5** *Given n or fewer labeled planar points, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time a smallest enclosing box can be identified for each labeled set.* ·

### 4.5 Nearest Point Problems

In this section, problems are considered that involve nearest neighbors of planar points. A variety of nearest neighbor problems have been explored for the serial computer (c.f. [Sham78, Tous80, LiTa77, BWY78]). One of these problems is the nearest neighbor query. The *nearest neighbor query* requires that a nearest neighbor of a single query point be identified. Given $n$ or fewer points, distributed one per processor on a mesh computer of size $n$, the nearest neighbor query can be solved in $\Theta(n^{1/2})$ time by broadcasting a copy of the query point to all processors, having each processor compute the distance from its point to the query point, and then taking the minimum over these results.

A more interesting problem is the *all-nearest neighbor problem for points*. Given a set $S$ of points, the solution to the all-nearest neighbor problem for points consists of finding for every point $p \in S$, a point $q \in S$ such that $d(p, q) = \min_{r \in S} d(p, r)$. Notice that a nearest neighbor is not



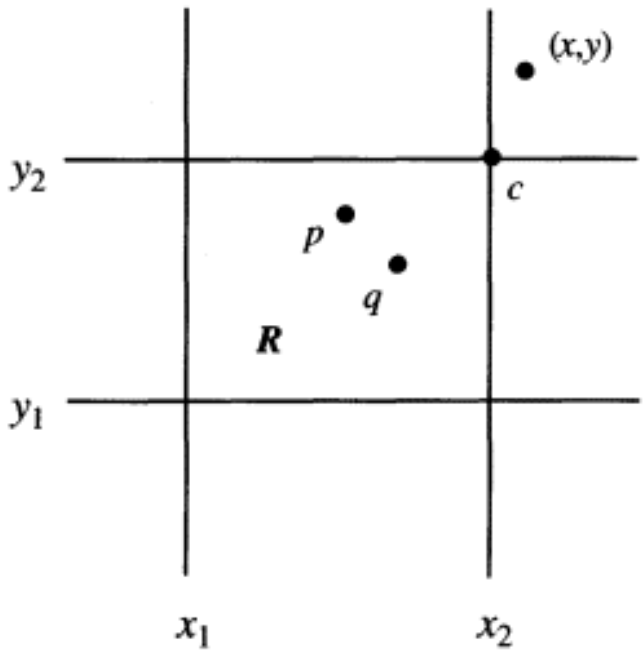



Figure 4.7:
Nearest neighbor in a corner.

necessarily unique, but that the distance to a nearest neighbor is unique. In this section, an asymptotically optimal algorithm is presented to solve the all-nearest neighbor problem for points. This solution easily yields an optimal mesh solution to the *closest-pair problem for points*, which requires that a closest pair of points from a given set be identified. This can be done on the mesh of size $n$ in $\Theta(n^{1/2})$ time by simply taking the minimum over the all-nearest neighbor distances.

Before considering the all-nearest neighbor problem for points, a useful lemma is presented. The nearest neighbor algorithms described in this section work by finding nearest neighbors in vertical 'strips,' and then in horizontal 'strips.' The lemma that follows shows that after these restricted solutions are determined, for any rectangular region that is determined by the intersection of a vertical and horizontal strip, there are only a few points in the region which have not yet found their global nearest neighbor. The reader should refer to Figure 4.7 during the statement and proof of the following lemma.

**Lemma 4.6** *Given a set $S$ of planar points, in two-dimensional space, and arbitrary real numbers $x_1 < x_2$ and $y_1 < y_2$, let $R = \{(x, y) \mid x_1 \le x \le x_2$ and $y_1 \le y < y_2\}$, let $D(p) = min\{d(p, q) \mid q \ne p, q \in S\}$, and*



*let $D'(p) = min\{d(p, q) \mid q \ne p, x_1 \le$ x-coordinate of $q \le x_2$ or $y_i \le$ y-coordinate of $q \le y_2$ $q \in S\}$. Then the following hold.*

*a) If $p$ is any element of $R \cap S$ such that $D(p) < D'(p)$, then there is a corner $c$ of the rectangle $R$ such that $d(p, c) < D'(p)$.*

*b) There are at most 8 points $p \in R \cap S$ such that there is a corner $c$ where $d(p, c) < D'(p)$.*

*Proof.* To prove a), notice that if $p \in R \cap S$ is such that $D(p) < D'(p)$, then there is a point $(x, y) \in S$ such that $D(p) = d(p, (x, y))$, $x$ is not in the interval $[x_1, x_2]$, and $y$ is not in the interval $[y_1, y_2]$. Assume $x > x_2$ and $y > y_2$, with all other cases being identical. In this case, if $c$ is the corner $(x_2, y_2)$, then $d(p, c) < d(p, (x, y)) = D(p) < D'(p)$, as was to be shown.

To show b), let $c$ be any corner, and suppose $p, q \in R \cap S$ are such that $d(p, c) < D'(p)$ and $d(q, c) < D'(q)$. It must be that the angle from $p$ to $c$ to $q$ is at least $\pi/3$ radians, for otherwise the further of $p$ and $q$ would be closer to the other than to $c$. Therefore there are at most 2 points of $R \cap S$ which are closer to $c$ than to any other point in $R$'s vertical or horizontal slab. Since there are only 4 corners, b) is proven.

The algorithm presented below to solve the all-nearest neighbor problem first uses sorting to partition the planar points into disjoint (linearly separable) vertical slabs, and then solves the restricted all-nearest neighbor problem so that all points know a nearest neighbor in their vertical slab. Next, the algorithm uses sorting to partition the planar points into disjoint horizontal slabs, and then solves the restricted all-nearest neighbor problem in each horizontal slab. At this point, it is known from the result of Lemma 4.6 that there are no more than 8 points in each rectangular region, as determined by the intersection of a vertical and horizontal slab, that may not know their true (global) nearest neighbor. The final step of the algorithm consists of passing these (fixed number of) points through the mesh so that each of these points views all other points, after which all points will know their nearest neighbor.

**Theorem 4.7** *Given $n$ or fewer planar points, distributed one per processor on a mesh computer of size $n$, the all-nearest neighbor problem for points can be solved in $\Theta(n^{1/2})$ time.*

*Proof*. The algorithm is recursive in nature. Initially, every processor $P_i$ containing a planar point $p_i$ creates a record with the $x$-coordinate of



$p_i$ as key. The data fields of this record include the $y$-coordinate of $p_i$, as well as the distance and identity to a nearest point found up to the current iteration of the algorithm. The distance field is initialized to $\infty$. Sort the points into proximity order by their $x$-coordinate. (Recall from Section 4.2.1 that no two unique points have the same $x$-coordinate.) After sorting, let $x_1$, $x_2$, $x_3$, and $x_4$ be the $x$-coordinates of the points in processors $P_{n/5}$, $P_{2n/5}$, $P_{3n/5}$, and $P_{4n/5}$ (in proximity ordering), respectively. These values divide the planar points into 5 vertical *slabs*, namely,

1. $(p \mid$ the $x$-coordinate of $p \leq x_1\}$,

2. $\{p \mid x_1 <$ $x$-coordinate of $p \leq x_2\}$,

3. $\{p \mid x_2 <$ $x$-coordinate of $p \leq x_3\}$,

4. $\{p \mid x_3 <$ $x$-coordinate of $p \leq x_4\}$, and

5. $\{p \mid x$-coordinate of $p > x_4\}$.

Recursively solve the restricted all-nearest neighbor problems so that every point determines a nearest neighbor (and the associated distance) in its slab. Now, repeat the process based on y-coordinates, determining for every point, a nearest neighbor in its horizontal slab.

The planar points can now be thought of as being in at most 25 rectangular regions, determined by $x_1, \ldots, x_4$ and $y_1, \ldots, y_4$. Sort the points by region to create ordered intervals of points corresponding to regions. Within each ordered interval, perform a semigroup (i.e., associative binary) operation to determine the, at most, 2 points (by Lemma 4.6) that are closer to each corner of the region than to their nearest neighbor found so far. All 2 * 4 * 25 (or fewer) such points are circulated to all $n$ processors by performing a rotation within the mesh, as described in Section 4.2.3, after which each processor $P_i$ knows the identity and distance from its planar point $p_i$ to a nearest neighbor. (It should be noted that the number of points that actually need to be circulated can be reduced to 128. Notice that the 9 interior squares each have 4 corners, 9 of the exterior squares each have 2 corners of concern, and the 4 outer squares each have 1 corner of concern. This is a total of 64 critical corners, each of which might have 2 points that need to be involved in the circulating step.) Sorting and semigroup operations within the ordered intervals corresponding to regions requires $\Theta(n^{1/2})$ time, as does circulating (rotating) a fixed number of points through



the mesh. Therefore, the time of the algorithm obeys the recurrence $T(n) = \Theta(n^{1/2}) + 2T(n/5)$, which is $\Theta(n^{1/2})$. ·

As stated previously, an efficient algorithm to solve the closest pair problem for planar points follows directly from the result just presented for the all-nearest neighbor problem. Simply apply the preceding $\Theta(n^{1/2})$ time algorithm and then in $\Theta(n^{1/2})$ time compute the minimum over the all-nearest neighbor distances, while keeping track of a pair that generates the minimal distance.

**Corollary 4.8** *Given n or fewer planar points, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time a closest pair of points can be identified.* ·

The next problem considered is the *all-nearest neighbor problem for point sets*. That is, for each labeled set of planar points, find the label and distance to a nearest distinctly labeled set of points. When the algorithm terminates, each processor that is responsible for a labeled point will know the nearest neighbor for the set that its point is a member of. It should be noted that a solution to the all-nearest neighbor problem for point sets will not, in general, provide a solution to the problem of detecting for each labeled point a nearest distinctly labeled point.

The solution to the all-nearest neighbor problem for point sets exploits an algorithm, given below in Lemma 4.9, that will determine the distance between two linearly separable sets of points in $\Theta(n^{1/2})$. time on a mesh of size $n$. The algorithm to determine the distance between two linearly separable sets works by conceptually partitioning a dividing line between the two sets into maximal subintervals such that for each subinterval there is a single point from each set closest to the interval. The minimum over the distances between these pairs of points represents the minimum distance between the two sets of points.

**Lemma 4.9** *Given n or fewer planar points, each labeled either A or B, distributed one per processor on a mesh computer of size n, and given the equation of a line L that separates A and B (i.e., all points labeled A lies on one side of L and all points labeled B lie on the other side of L), in* $\Theta(n^{1/2})$ *time every processor can determine the distance from A to B.*

*Proof.* In $\Theta(n^{1/2})$ time, a semigroup (i.e., associative binary) operation will determine if either $A$ or $B$ is empty, in which case the answer is infinity. Otherwise, the equation of $L$, along with a choice of orientation



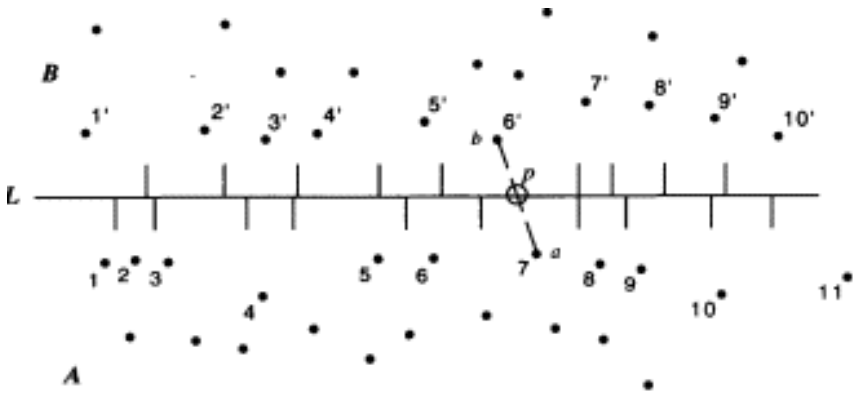



Figure 4.8:
Partitioning $L$ into maximal intervals. The labeled points correspond to the intervals.

and origin for $L$, is broadcast to all processors in $\Theta(n^{1/2})$ time. Suppose $a \in A$ and $b \in B$ are such that $d(a, b)$ equals the distance from $A$ to $B$. Since $A$ and $B$ are separated by $L$, then the line $L$ must intersect line segment $ab$ at some point $p$. Since $d(a, b)$ is the minimum distance between points in $A$ and points in $B$, it must be that a is a closest point in $A$ to $p$, and $b$ is a closest point in $B$ to $p$. This fact can be exploited, as follows (see Figure 4.8).

1. Partition $L$ into a set of maximal intervals such that for each interval there is a single element of $A$ which is a closest point of $A$ to each point of the interval.

2. Partition $L$ into a set of maximal intervals such that for each interval there is a single element of $B$ which is a closest point of $B$ to each point of the interval.

3. Perform an intersection operation on these sets of intervals to determine a closest pair $(a_i,b_i)$, $a_i \in A$,$b_i \in B$, for each interval $I_i$.

4. Determine $min\{d(a_i, b_i)l(a_i, b_i)$ is a closest pair of $l_i\}$.

Details of the algorithm follow. First sort the points into sets $A$ and $B$. The partitioning of $L$, as described in steps 1 and 2, now proceeds independently and identically. The partitioning is explained for set $A$. Every interval $[p_l,p_r]$ of $L$ will be represented by two *interval records*, one with $p_l$ as the key and the other with $p_r$ as the key. The interval



records take the form $(p_1, p_2,$ data), where $p_1$ is the key, and the data includes the Cartesian coordinates of the point of $A$ that determines the interval $[p_1, p_2]$. Notice that the intervals in the partition overlap only at their endpoints. If $A$ consists of a single point $x$, then there is only one interval, which is represented by the creation of interval records $(-\infty, \infty, x)$ and $(\infty, -\infty, x)$. If $A$ has more than one point stored in processors numbered 1 through $k$ (in proximity order), then simultaneously and recursively, find the intervals given by the set of points $H_1$ in processors numbered 1 through $\lfloor k/2 \rfloor$, and those given by the set of points $H_2$ in processors numbered $\lfloor (k/2) \rfloor + 1$ through $k$.

Notice that an interval from $H_1$ or $H_2$, can only shrink or disappear in the final set of intervals for $A$. Since an interval from $H_1$ ($H_2$) may overlap many intervals from $H_2$ ($H_1$), the intervals from $H_2$ ($H_1$) will be used to determine how much an interval from $H_1$ ($H_2$) shrinks. To shrink the intervals, shrink those that came from $H_1$ first, and then those that came from $H_2$, as follows.

First, generate interval records that also include a 1 or a 2 in the data field to indicate for each record whether it came from $H_1$ or $H_2$, respectively. Sort these representatives by their key (an endpoint). In case of ties, a left endpoint of $H_1$ precedes any endpoint of $H_2$, and a right endpoint of $H_1$ follows any endpoint of $H_2$. For each interval from $H_1$, the processors between the representatives of its left and right endpoints form an interval of processors that is called a *group*. Each processor holding an interval record of $H_2$ can determine which group the record is in, as follows. Every processor that contains a record representing an interval of $H_1$ keyed by its left endpoint is involved in a concurrent read to determine the proximity order index of the processor that contains the interval record keyed to the right endpoint of its interval. Viewing each group as an ordered interval, every processor containing a representative of the left endpoint of an interval of $H_1$ (the *leader* of each group) creates a spanning tree in its group, as described in Section 4.2.3 and in the algorithm of Theorem 4.4. While the spanning tree is created within each group, every processor representing an interval of $H_2$ is informed as to the point that defines the $H_1$ interval of the group that it is a member of. In a fixed amount of time, every processor containing an interval of $H_2$ determines which part of the intersection of its interval and the group's interval is closest to its point. By finding a minimum and maximum within intervals of processors (by computing an associative binary operation within intervals), each group can then determine the final interval (if any) corresponding to the group's point. Finally, repeat the process, interchanging the roles of $H_1$ and $H_2$ to



determine the final intervals.

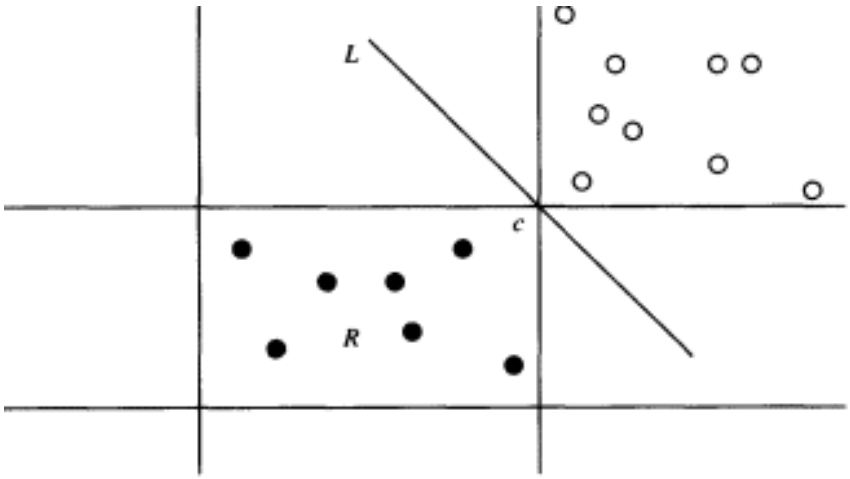
Once the partitions corresponding to $A$ and $B$ have been determined, the process of finding a nearest pair between them is similar. First, groups corresponding to intervals of $A$ find the nearest point of any interval of $B$ not properly containing the $A$ interval, and then groups corresponding to intervals of $B$ find the nearest point of any interval of $A$ not properly containing the $B$ interval. Finding the global minimum gives the answer. The running time of the algorithm is given by the recurrence $T(n) = T(n/2) + \Theta(n^{1/2})$, which is $\Theta(n^{1/2})$. ·

The solution to the all-nearest neighbor problem for point sets can be solved by using the underlying structure of the algorithm associated with Theorem 4.7. That is, the problem is first solved recursively for disjoint vertical slabs and then disjoint horizontal slabs. Within each rectangular region formed by the intersection of a vertical and horizontal slab, there will be points from at most 8 sets (2 with respect to each corner of the region) which may not know the correct global solution. For each such set of points in every rectangular region, the result of the previous lemma can be applied (sequentially) with respect to a corner of the rectangular region.

**Theorem 4.10** *Given n or fewer labeled points representing sets of points, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time the all-nearest neighbor problem for point sets can be solved.*

*Proof*. Each point will attempt to find a nearest point of a different label, quitting only when it determines that it cannot find a nearer neighbor than some other point in its set can find. The algorithm in Theorem 4.7 is used, resulting in the same conclusion that for each corner of every rectangular region (determined by the intersection of a vertical and horizontal slab) there are points from at most 2 sets which may be able to find closer points in the direction of the corner. The slight difference is that in each of these regions there may be O($n$) points from the same set considering possibilities in the same direction. For a given rectangular region $R$, assume that a set $A$ of labeled points is one of the, at most, two closest sets of labeled points to a corner $c$ of the region. A tilted line $L$ through c and tangent to $R$ is a separating line from $A \cap R$ and the points in $S - A$ on the other side of $L$ in the target direction, where $S$ represents the entire set of $n$ labeled points. (See Figure 4.9, where the direction is northeast.) Therefore, at most 128 applications of the algorithm associated with Lemma 4.9 are needed. A final concurrent write and concurrent read complete the operation. ·

Page 174




• Points in A ∩ R

○ Points of $S - A$ in northwest direction from $R$.

Figure 4.9:
Solution to the all-nearest neighbor problem for point sets.

Page 175

Given a collection $S$ of planar points, a spanning tree can be constructed by using the points as vertices and straight lines between them as edges. A *minimal-distance spanning tree* is a spanning tree of $S$ which minimizes the sum of the Euclidean lengths of the tree edges. A standard approach to building minimal spanning trees was given in Section 3.2.3. Start off with each point as being its own labeled component (club). Then each connected component (as a point set) merges with a nearest neighbor (in case of ties, the one of minimal label is chosen), and an edge corresponding to this minimal distance is added to the edge set of the minimal distance spanning tree. This occurs simultaneously for all components. Each iteration reduces the number of components by at least a factor of two, so at most $\log_2 n$ iterations are needed. Using Theorem 4.7 to find nearest neighbors, and the graph labeling algorithm of [ReSt] to label components, a minimal-distance spanning tree of a set of $n$ or fewer planar points, distributed one per processor on a mesh computer of size $n$, may be determined in $\Theta(n^{1/2}\log n)$ time.

Unfortunately, this algorithm is not optimal. In Section 4.10, an optimal $\Theta(n^{1/2})$ time mesh algorithm is given to solve the minimal-distance spanning tree problem for point data. This optimal solution is based on a $\Theta(n^{1/2})$ time mesh algorithm for constructing the Voronoi diagram of a set of points, coupled with the minimum-weight spanning tree algorithm for graphs appearing in [ReSt]. A definition of the Voronoi diagram, an optimal mesh algorithm for constructing it, and a number of applications of the Voronoi diagram are given in Section 4.10.

### 4.6 Line Segments and Simple Polygons

In this section, problems involving line segments and simple polygons are examined. The first problem considered is that of determining whether or not there is an intersection among sets of planar line segments. This is a fundamental problem in computational geometry [Sham78, BeOt79, PrSh85]. In fact, Preparata and Shamos [PrSh85] conjecture that in order to efficiently solve hidden-line problems, one must first be able to solve basic intersection problems.

The first algorithm presented in this section introduces the use of a paradigm known as *multidimensional divide-and-conquer* [Bent80]. In this approach, $k$-dimensional problems are solved by subdividing them into smaller $k$-dimensional problems, plus similar $(k - 1)$-dimensional problems. These pieces are solved recursively and are then glued together. When this paradigm is used on parallel computers, the smaller



pieces can be solved simultaneously. However, this raises the possibility that an initial object may be subdivided into several smaller objects, and if the recursion causes this to happen repeatedly, there can be an explosion in the amount of data. The algorithms of this section prevent this by insuring that any initial line segment never has more than two pieces representing it at the start of any level of recursion, no matter how many levels of recursion have occurred.

The algorithm that follows in the proof of Theorem 4.11 uses multidimensional divide-and-conquer to provide an optimal mesh solution to the problem of determining whether or not there exists an intersection between line segments with different labels, where line segments of the same label are assumed to be nonintersecting except possibly at endpoints. The algorithm works by creating vertical slabs, sending a representative of each line segment to every slab that it passes through or has an endpoint in, and then recursively solving the problem in each vertical slab. A pitfall to this approach is that if it is performed in a straightforward fashion, there may be an explosion in the amount of data that exists due to the fact that each line segment may recursively generate multiple representatives at each stage of the recursion. To avoid this, at every stage of the recursion, in each vertical slab, every representative associated with a line segment that passes completely through the slab is examined to determine whether or not it is intersected in the slab by a distinctly labeled line segment. If such an intersection exists, then the algorithm records that an intersection was detected and terminates. If there are no intersections within a slab involving these *spanning line segments* (i.e., line segments that pass completely through the slab), then the representatives corresponding to the spanning line segments may be discarded and the algorithm may proceed recursively within the slab. This guarantees that there will never be more than two representatives associated with any line segment at the beginning of any stage of recursion, and that there will never be more than some fixed number of representatives associated with each line segment in the entire mesh at any time during the algorithm.

**Theorem 4.11** *Given n or fewer labeled line segments, distributed one per processor on a mesh computer of size n, if no two line segments with the same label intersect other than at endpoints, then in* $\Theta(n^{1/2})$ *time it can be determined whether or not there are any intersections of line segments with different labels.*

*Proof.* Each processor with a labeled line segment $\overline{ab}$ creates two *line segment records*. One record has the $x$-coordinate of $a$ as key, with the



$y$-coordinate of $a$, coordinates of $b$, and label of $\overline{ab}$ as data, while the other record has the $x$-coordinate of $b$ as key, with the y-coordinate of $b$, coordinates of $a$, and label of $\overline{ab}$ as data. In $\Theta(n^{1/2})$ time, these $2n$ records are sorted into proximity order by the key field.

After sorting, the keys ($x$-coordinates) of the first record in processors $P_{n/4}$, $P_{n/2}$, and $P_{3n/4}$ (with respect to proximity order index) are used to partition the plane into 4 vertical *slabs*. These 3 values are then broadcast to all processors in $\Theta(n^{1/2})$ time. For each record representing the left endpoint $a$ of a line segment $\overline{ab}$, the processor holding the record determines if there are any slabs which the line segment crosses completely. For each such line segment and slab pair, the processor containing the line segment record generates a *spanning line record* equivalent to the line segment record except that the key is the left $x$-coordinate of the slab. The spanning line records will be used temporarily and then destroyed, which prevents the overaccumulation of data records. Sort the spanning line records by slab, breaking ties arbitrarily, and perform a semigroup (i.e., associative binary) operation within each ordered interval corresponding to a slab to enumerate the spanning line segments of each slab. Finally, using these numbers, a concurrent write is used to send the spanning line records to their slab. This is accomplished in $\Theta(n^{1/2})$ time.

Each slab is now stored in a quadrant of the mesh. Within each quadrant of processors, in $\Theta(n^{1/2})$ time it can be determined as to whether or not there is an intersection among the spanning line segments. This can be accomplished as follows. Sort the spanning line segments by $y$-intercept with the left boundary of the slab. This determines for each spanning line segment its position relative to the other spanning line segments with respect to the left boundary of the slab. Repeat the process to find the relative position of each spanning line segment with respect to the right boundary of the slab. If any spanning line segment has different order positions for the left and right boundary, or if there were any ties involving line segments with different labels, then there is an intersection within the slab, and the problem is solved.

Otherwise, in each slab, the spanning line segments divide the slab into nonoverlapping *regions*. (It should be noted that the property that line segments of the same label can only intersect at their endpoints is used here to guarantee that these regions are nonoverlapping. If arbitrary intersections were allowed among line segments with the same label, then spanning line segments of the same label could cross each other in the interior of the slab.) Any line segment not spanning this slab will intersect spanning line segments if and only if its endpoints lie

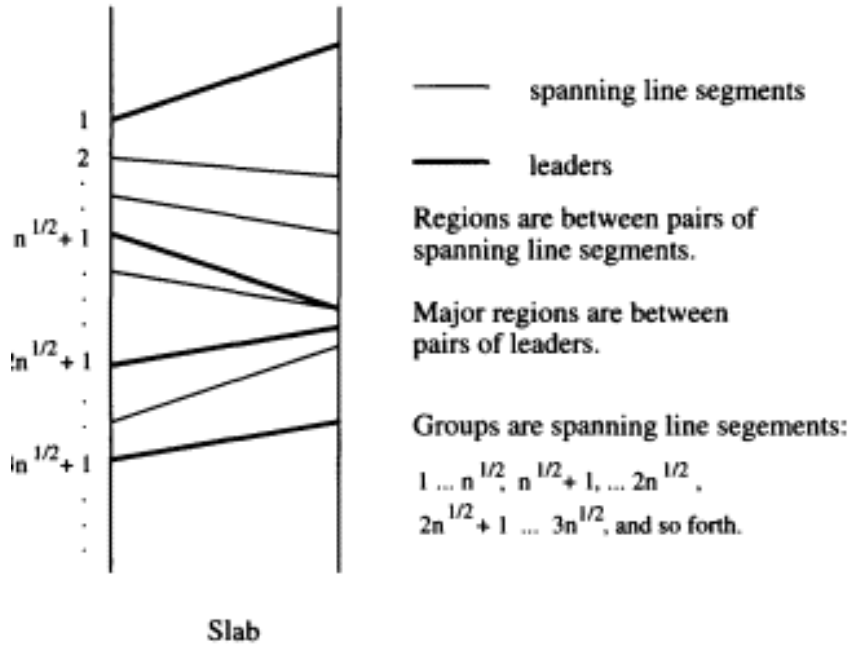



Figure 4.10:
Spanning line segments, leaders, regions, and major regions.

in different regions or on one of the spanning line segments. (If the line segment does not span, but does extend outside the slab, then one of its endpoints is temporarily treated as being the appropriate $y$-intercept.)

A 2-pass grouping operation, as described on page 27, may be used to determine whether or not spanning line segments are intersected in a given slab. Sort all spanning line segments by the $y$-intercept of the left boundary. The spanning line segments in each disjoint interval of $n^{1/2}$ processors form a group, and the first spanning line segment of each group is the *leader* of the group. See Figure 4.10. In $\Theta(n^{1/2})$ time, rotate the leaders through the processors of the slab (stored in a quadrant of the mesh), using the rotation algorithm of Section 4.2.3, where the number of processors is $i(r) = \Theta(n)$ and the number of records being rotated is $m(r) = O(n^{1/2})$. During the rotation, the major region that a nonspanning line segment lies in is recorded in its record, where a *major region* is determined by a consecutive pair of leaders, as shown in Figure 4.10..

With respect to the left boundary of the slab, use the $y$-intercept of the spanning line segments and the $y$-intercept of the top boundary of



the major region for each nonspanning line segment as keys, and sort the spanning line segment and nonspanning line segment records together, with ties broken in favor of spanning line segments. A concurrent read is performed so that the leader of each major region can determine the proximity order index of the next leader of a major region, forming a group. Within each group, in $\Theta(n^{1/2})$ time, the $O(n^{1/2})$ spanning line segments are rotated. During the rotation, any intersection between a spanning line segment and a nonspanning line segment of a different label can be detected, and the fact that such an intersection was detected can be recorded in the appropriate nonspanning line segment record. Finally, a semigroup (i.e., associative binary) operation determines if any of the spanning lines segments in the slab were intersected.

If such an intersection exists, then the algorithm is done, while otherwise the spanning line segments are discarded, and in each slab the problem is recursively solved to determine whether or not there are any intersections among line segments of different labels with endpoints in the slab.

The running time of a single step of the algorithm is dominated by a fixed number of data movement operations such as sorting, concurrent read, concurrent write, and (interval) grouping operations. Therefore, step $i$ of the algorithm operates on a subsquare of size $k = n/4^i$ and finishes in $\Theta(k^{1/2})$ time. Hence, the running time of the entire algorithm obeys the recurrence $T(n) = T(n/4) + \Theta(n^{1/2})$, which is $\Theta(n^{1/2})$. ·

With minor changes, the above algorithm can be modified to ignore intersections of line segments at common endpoints, or to ignore intersections involving the endpoint of one line segment but the middle of another.

The next problem considered is the *all-nearest neighbor problem for sets of line segments*. That is, for each set of line segments, find the label and distance to a nearest distinct set of line segments. When the algorithm terminates, each processor that is responsible for a labeled line segment will know the nearest neighbor for the set that its line segment is a member of.

The algorithm to solve this problem combines features of the algorithm presented in the preceding theorem with that of the algorithm of Theorem 4.10. The plane is divided into 5 vertical slabs and 5 horizontal slabs, and nearest neighbors within each are found. To find nearest neighbors in a slab, first each line segment with an endpoint in the slab finds the nearest spanning line segment, and each spanning line segment finds the nearest neighbor among the spanning line segments and line



segments with an endpoint in the slab. The spanning line segments use a concurrent write to report this back to the endpoint that generated them, and are then discarded.

As in the algorithm of Theorem 4.10, after the nearest neighbors in slabs have been found, in each rectangular region (determined by the intersection of a vertical and horizontal slab) there are at most 8 labels with endpoints of line segments in the region which may not yet have found their nearest neighbor. Lemma 4.9 can be straightforwardly extended to line segments, with the slight difference that $k$ nonoverlapping (except at endpoints) line segments may partition the separating line $L$ into as many as $2k - 1$ regions. A statement of the theorem follows.

**Theorem 4.12** *Given n or fewer labeled line segments, distributed one per processor on a mesh computer of size n, where line segments intersect at most at their endpoints, in* $\Theta(n^{1/2})$ *time the all-nearest neighbor problem for sets of line segments can be solved.* ·

An algorithm derived from the one associated with Theorem 4.12, where each line segment finds a nearest neighbor with a different label directly above it, will be useful in some of the algorithms that appear later in this section. Horizontal slabs are not needed, nor is there a final stage involving line segments close to corners of rectangular regions. Since the final stage is eliminated, each line segment can find a nearest neighbor in an upward direction, rather than just finding a nearest neighbor for each label.

**Corollary 4.13** *Given n or fewer labeled line segments, distributed one per processor on a mesh computer of size n, where line segments intersect at most at their endpoints, in* $\Theta(n^{1/2})$ *time every line segment can determine a nearest neighbor of a different label above it.* ·

A polygon is *simple* if it has the property that every two consecutive edges share only a common endpoint, and no two nonconsecutive edges intersect. While vertices can be used to uniquely represent a convex figure, a simple polygon cannot be represented by vertices unless they are given in an enumerated fashion. The input to problems in this section involving simple polygons will be in the form of line segments that represent the polygons.

Some of the algorithms that follow will make use of an efficient solution to the *connected component labeling problem for line segments*, where two line segments are connected if and only if they share a common endpoint. Proposition 4.14 is due to Reif and Stout [ReSt].



**Proposition 4.14** *Given n or fewer line segments* (*edges*), *distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time every processor containing a line segment can know the label of the connected component that its segment is a member of.*

Consider the problem of determining for each labeled set of line segments, whether or not it forms a simple polygon. The algorithm follows directly from the results of Proposition 4.14, which is used to determine for each labeled set whether or not it is connected, and Theorem 4.11, which is used to determine for each connected labeled set whether or not any of its edges intersect.

**Theorem 4.15** *Given n or fewer nondegenerate labeled line segments, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time it can be determined for each set whether or not the line segments form a simple polygon.*

*Proof.* Sort the line segments by labels into proximity order in $\Theta(n^{1/2})$ time. Using the algorithm associated with Proposition 4.14, for each set of line segments, simultaneously label all connected components in $\Theta(n^{1/2})$ time. Using a report and broadcast within each set of line segments, in $\Theta(n^{1/2})$ time discard those sets for which not all line segments received the same component label. Next, using a concurrent read within each set, in $\Theta(n^{1/2})$ time mark each line segment that does not satisfy the condition that each of its endpoints intersects exactly one endpoint from a distinct line segment of its component (set). Those components that contain marked line segments do not form simple polygons and are also discarded. For each of the remaining sets of line segments, apply the intersection algorithm of Theorem 4.11, treating each line segment as having a unique label, and ignoring intersections at common endpoints. Those components that contain intersections are not simple polygons, while the remaining nondiscarded polygons are simple. A final $\Theta(n^{1/2})$ time concurrent read returns the line segments to their original processors with the solution to the query.

Consider the problem of determining whether or not there is an intersection among a set of simple polygons. Before presenting a solution to this problem, a useful result that distinguishes the inside from the outside of each polygon will be given. The algorithm works by finding the point of each polygon with minimal $x$-coordinate, using the convex angle formed by the two edges incident on this point to identify the inside of the polygon, and then propagating interior/exterior information to all other edges of the polygon.

**Lemma 4.16** *Given multiple simple polygons, represented by n or fewer labeled line segments, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time each processor containing a line segment can determine which side of its line segment is towards the interior of its polygon.*

*Proof.* Every processor that contains a line segment creates a *line segment record* with the polygon label as major key and the $x$-coordinate of the leftmost of the two endpoints as minor key. Sort the line segment records by polygon labels (key), with ties broken in favor of minimum $x$-coordinate. After sorting, the first two line segments of each label intersect at the leftmost point of that polygon. Therefore, their interior angle must be towards the interior of the polygon. For each labeled polygon, conceptually eliminate the leftmost point from the bottommost of these two line segments. This serves to conceptually eliminate the link between these two line segments. Now, in this modified graph, select the leftmost point as root, and orient the edges to form an upward directed graph. (Algorithms appearing in [Stou85a, AtHa85] do this in the required time.) This graph represents a counterclockwise traversal of the polygon, so for each edge the inside is the left-hand side when going upward (in the tree). A final concurrent read allows every processor to know the orientation of the line segment that it initially contained, with respect to its simple polygon. ·

An algorithm to detect an intersection among a set of simple polygons follows directly from previous results. First, use the line segment intersection algorithm to determine whether or not there are any intersections among edges of polygons. If not, then the only possibility for intersection is via containment, i.e., if one polygon is completely contained within another. After each edge of a polygon distinguishes the outside from the inside of its polygon, every edge finds a nearest edge directly above it. Containment exists if and only if some edge is on the inside side of the nearest edge that it detects directly above it.

**Theorem 4.17** *Given multiple simple polygons, represented by n or fewer labeled line segments, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time it can be decided whether or not there is an intersection among the polygons.*

*Proof.* From Theorem 4.11, in $\Theta(n^{1/2})$ time it can be detected whether or not there is an intersection of line segments. It only remains to detect if one simple polygon contains another. If there is a containment



relationship among some polygons, then there is at least one line segment 1 for which the closest line segment to $l$, among the line segments directly above it, is a line segment $k$ of a polygon that contains 1. That is, $l$ is on the inside of $k$, and hence the polygon that $l$ is a member of is contained in the polygon that $k$ is a member of. Further, if no polygons are inside of others, then for every line segment $l$, the closest line segment $k$ directly above it either belongs to the same polygon as $l$, or else $l$ is on the outside of $k$, and hence the polygon that $l$ belongs to is not contained in the polygon that $k$ belongs to.

In $\Theta(n^{1/2})$ time, temporarily give each line segment its own label and use the modified version of the algorithm in Theorem 4.12 to find, for each line segment, the nearest neighbor directly above it (if any). Using Lemma 4.16 to determine orientations, in $\Theta(n^{1/2})$ time it can be decided as to whether or not any polygon is contained in another. The algorithm from Theorem 4.11, the modification of the algorithm from Theorem 4.12, and the algorithm from Lemma 4.16 all finish in $\Theta(n^{1/2})$ time. Therefore, the running time of the algorithm is as claimed.

The following result is an immediate corollary of Theorem 4.12.

**Corollary 4.18** *Given multiple nonintersecting simple polygons, represented by n or fewer labeled line segments distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time the all-nearest neighbor problem for simple polygons can be solved.*

The final problem considered in this section involves query points and a set of nonintersecting simple polygons. The problem is to determine for every query point, whether or not it is contained in a polygon, and if so, the label of such a polygon.

**Corollary 4.19** *Given multiple nonintersecting simple polygons, represented by labeled line segments, and given a collection of points, such that there are no more than n segments and points, stored no more than one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time each point can determine the label of a polygon it is in, if any.*

*Proof.* Assign to all points a label that is different from all of the polygons. Then use the modified version of the algorithm in Theorem 4.12 to find the nearest line segment above each point. If the point is on the inside side of this segment, then the point is in the polygon, while otherwise it is outside of it.



## 4.7 Intersection of Convex Sets

This section presents efficient mesh algorithms to determine intersection properties of convex sets and to solve the 2-variable linear programming problem. Intersection problems involving convex sets have been considered for the serial model (c.f., [OCON82, PrSh85, ShHo76]), since many of these problems solve classic pattern recognition queries.

The first result of this section shows that a mesh computer can be used to efficiently determine whether or not the convex hulls of two arbitrary sets of planar points intersect. A related problem is that of determining whether or not two sets of planar points $S_1$ and $S_2$, are linearly separable [Tous80], where $S_1$ and $S_2$ are *linearly separable* if and only if there exists a line in the plane such that all points in $S_1$ lies on one side of the line, and all points in $S_2$ lies on the other side. It is not hard to show that sets of planar points are linearly separable if and only if their convex hulls are disjoint.

The algorithm that follows in the proof of Theorem 4.20, to determine whether or not the convex hulls of two sets of points are linearly separable, relies heavily on the notion of angles of support, as defined on page 19. The algorithm exploits the fact that two convex sets, $S_1$ and $S_2$, are linearly separable if and only if there exists a pair of extreme points, $p \in S_1$ and $q \in S_2$, as shown in Figure 4.11, such that an angle of support of $p$ differs by $\pi$ from an angle of support of $q$, and that the half-planes that these angles represent do not intersect. Every extreme point attempts to find such an extreme point of the other set. This is accomplished by a one pass grouping operation based on the monotonicity of angles of support for extreme points.

**Theorem 4.20** *Given n or fewer labeled planar points, representing sets $S_1$ and $S_2$, distributed one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time it can be determined as to whether or not hull($S_1$) and hull($S_2$) intersect. Further, if they do not intersect, then a separating line between $S_1$ and $S_2$ can be determined.*

*Proof.* If $S_1$ and $S_2$ are separated by a line $L$, then there are extreme points $p \in S_1$ and $q \in S_2$ such that each of $p$ and $q$ has an angle of support parallel to $L$, and these angles of support differ by $\pi$ from each other (see Figure 4.11). Further, given extreme points $p' \in S_1$ and $q' \in S_2$, along with their angles of support, in constant time it can be determined if there is such a separating line. To locate a separating line, if one exists, representatives of the angles of support of the extreme



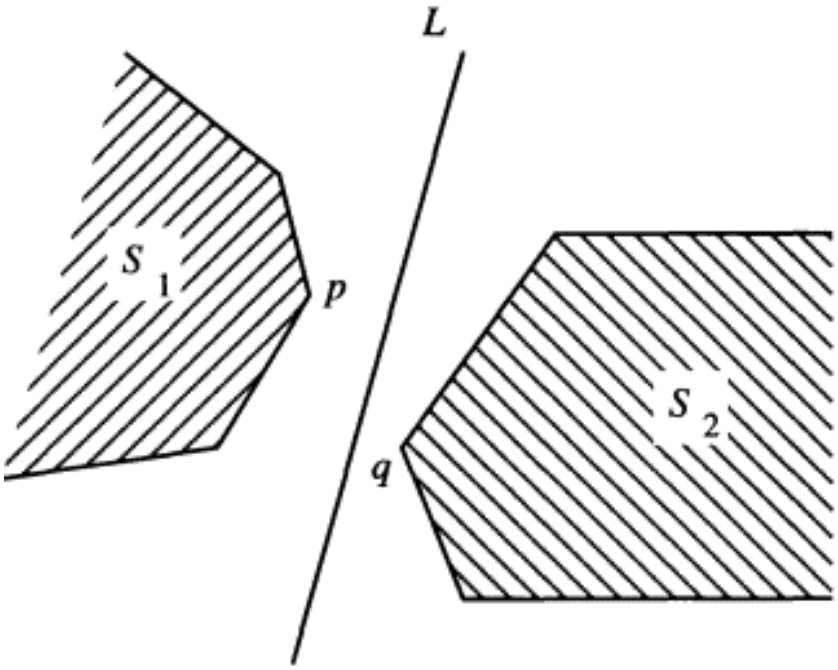



Figure 4.11:
Line $L$ separates $p$ and $q$.

points of $S_1$ and $S_2$ will use a grouping technique to locate extreme points of the other set with an angle of support differing by $\pi$.

Each extreme point $p \in S_1$ creates two records containing $p$'s coordinates and its range of supporting angles, along with an indicator that $p$ is in $S_1$. One of these records has $p$'s smallest supporting angle as its key, and the other has $p$'s largest supporting angle as its key. Each extreme point of $S_2$ creates two similar records, adding $\pi \pmod{2\pi}$ to each angle. Further, to convert the circular ordering of angles of support into a linear ordering, additional records are generated for an extreme point of $S_1$ having 0 as an angle of support and an extreme point of $S_2$ having $\pi$ as an angle of support. (These records will have keys of 0 and $2\pi$.) All records are then sorted by key, using smallest supporting angles as a secondary key.

Notice that if extreme points $p \in S_1$ and $q \in S_2$ have a separating line, then either an endpoint of $q$'s range of angles of support (plus $\pi$) is within the range of $p$'s angles of support, or vice versa, or both. For the first and third cases, if the records are viewed as grouped by intervals of angles of support determined by extreme points in $S_1$, then by circulating the information about each extreme point in $S_1$ throughout its interval,



every processor holding a record corresponding to an extreme point in $S_2$ can determine if there is a line separating them. Similarly, for the second case, the records can be viewed as grouped by intervals determined by extreme points in $S_2$.

The previous theorem presents an interesting algorithm to determine whether or not the convex hulls of two sets of planar points intersect. In fact, the problem of determining whether or not there is an intersection among the convex hulls of multiple sets of planar points can be solved in the same asymptotically optimal time, as follows. Use the algorithm associated with Corollary 4.2 to enumerate the extreme points of each labeled set. Then use a concurrent read to generate the edges of the convex hulls. Finally, use the polygon intersection algorithm of Theorem 4.17 to give the desired result.

**Theorem 4.21** *Given n or fewer labeled planar points, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time it can be determined whether or not any two labeled sets have convex hulls which intersect.* ·

The next problem considered is that of *constructing* the intersection of multiple half-planes, which is based on a straightforward bottom-up merging algorithm, where at each stage ordered intersections of half-planes are merged. Serial solutions to this problem appear in [PrSh85].

**Theorem 4.22** *Given the description of n or fewer half-planes, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time their intersection can be determined.*

*Proof.* Sort the half-planes into proximity order by their angles. Half-planes with the same angle can be combined into a single half-plane using simple prefix calculations. The core of the algorithm is a simple bottom-up merge procedure, where stage $i$ merges $2^i$ half-planes into their intersection in $\Theta(2^{i/2})$ time. At each stage, the result is a (perhaps infinite) convex figure, and when two figures are being merged, the initial sorting guarantees that at most one is noninfinite and either one is contained in the other, they have no intersection, their boundaries intersect in exactly one point, or their boundaries intersect in exactly two points. These cases can easily be determined and solved using, say, the algorithm in Theorem 4.11 to locate the intersections and the algorithm of Theorem 4.17 to determine if there is containment. ·



It was noted in [Gass69] that linear programming can be viewed as an intersection problem, determining the intersection of half-planes and evaluating the objective function at each extreme point. Corollary 4.23 follows directly from Theorem 4.22.

**Corollary 4.23** *Given n or fewer 2-variable linear inequalities, distributed one per processor on a mesh computer of size n, and a unit-time computable objective function to be maximized (minimized), then in* $\Theta(n^{1/2})$ *time the linear programming problem can be solved.* ·

Since each convex polygon is the intersection of the incident half-planes corresponding to its edges, the problem of *constructing* the intersection of multiple convex polygons can be solved by a simple application of the algorithm associated with Theorem 4.22. It should be noted that the following corollary can also be obtained by using a bottom-up merging approach which intersects pairs of convex polygons together.

**Corollary 4.24** *Given multiple labeled convex polygons, represented by n or fewer labeled planar points, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time the common intersection of the polygons can be constructed.* ·

## 4.8 Diameter

The problem of detecting a farthest pair and computing the diameter of a set of planar points is closely related to the convex hull problem. In fact, a farthest pair of points must be a pair of extreme points of the convex hull of the set of points [Sham78]. The distance between such a pair of points gives the diameter of the set. An optimal mesh algorithm for computing a farthest pair and the diameter of a set, or multiple labeled sets, of planar points follows directly from the techniques and algorithms presented in Section 4.7. Given a set $S$ of $n$ planar points, distributed one per processor on a mesh computer of size $n$, first mark the extreme points of hull($S$) using the algorithm associated with Corollary 4.2, then determine for each extreme point its angles of support as described in Section 4.7, and then based on these angles, perform a grouping operation to find a farthest point from every point. Performing a semigroup (i.e., associative binary) operation over the set of point-wise farthest pairs, gives a farthest pair and the diameter of $S$.



**Theorem 4.25** *Given n or fewer labeled planar points, distributed one per processor on a mesh computer of size n, a farthest pair and the diameter of every labeled set can be determined in* $\Theta(n^{1/2})$ *time.* ·

## 4.9 Iso-oriented Rectangles and Polygons

Problems involving rectangles have been well studied for the serial model of computation [MeCo79, PrSh85, McCr81], since they are important to many packing and layout problems. An important class of rectangles are the iso-oriented ones, where an *iso-oriented* (*planar*) *rectangle* is a planar rectangle with the property that one pair of opposite sides is parallel to the $x$-axis and the other pair is parallel to the y-axis. In this section, the input to the problems considered is $n$ or fewer iso-oriented planar rectangles, distributed one per processor on a mesh computer of size $n$. It is assumed that each iso-oriented rectangle is described by the Cartesian coordinates of its four planar vertices. To distinguish the rectangles during the course of an algorithm, each rectangle is labeled by the index of the processor that it is initially contained in. Multidimensional divide-and-conquer, as introduced in Section 4.6, will be used extensively.

Recall from Section 4.6 that for general simple polygons, it was only possible to detect whether or not an intersection exists. The first theorem of this section shows that when the polygons are restricted to iso-oriented rectangles, then in $\Theta(n^{1/2})$ time, every rectangle can determine whether or not it is intersected by another rectangle. The algorithm presented to solve this problem is based on the slab method, which has been used extensively in this chapter. Vertical slabs are created and every rectangle sends a representative to each slab that the rectangle passes through or terminates in. The problem is then solved for *spanning rectangles* within each slab, after which the representatives of the spanning rectangles are discarded. The algorithm then proceeds recursively within each slab. Processing and discarding the spanning rectangle representatives at the beginning of each stage of the recursion guarantees that there will not be an overaccumulation of data.

**Theorem 4.26** *Given n or fewer iso-oriented planar rectangles, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time every rectangle can determine whether or not it is intersected by another rectangle.*

*Proof.* Every processor initially storing a rectangle creates two representatives of the rectangle, one with the $x$-coordinate of the left side



as key, with the rest of the rectangle's description, proximity index of the processor (to be used as the label of the rectangle), and a flag set to 'left' as data, and the other with the $x$-coordinate of the right side as the key, with the description, proximity index of the processor, and 'right' as data. After this initialization step, there are at most $2n$ representatives. Each processor keeps track of the left and right *limits* of the region under consideration as the algorithm progresses, similar to other slab partitioning algorithms that have been presented in this chapter. Initially, each processor sets the left and right limits to $-\infty$ and $+\infty$, respectively.

Sort the representatives by the key field. The key (an $x$-coordinate) of the second representative in processors $P_{n/4}$, $P_{n/2}$, and $P_{3_n/4}$ (in proximity order) are broadcast to all processors. This serves to partition the region into 4 vertical slabs. Every processor holding a representative of a rectangle that spans one or more slabs generates a *special record* describing the rectangle for each slab the rectangle completely crosses. Initially, a rectangle can cross at most two vertical slabs, but in latter stages of recursion a rectangle may cross three slabs. The special records are then sent to the quadrant of the mesh that is responsible for maintaining representatives of the spanned slab. This is accomplished by sorting the special records with respect to slabs, performing a semigroup (i.e., associative binary) operation within ordered intervals corresponding to slabs to enumerate the special records of each slab, and performing a concurrent write to send special records to their appropriate slabs.

In each slab, these special records represent spanning rectangles. Notice that a spanning rectangle is intersected by another iso-oriented rectangle in the slab, if and only if their y-coordinates overlap. Therefore, spanning rectangle intersections have been reduced to a 1-dimensional intersection problem, and can be solved as follows. First, perform a sort step to eliminate duplicate entries that might have been created by a left and right representative of the same rectangle. For each spanning and nonspanning rectangle, two records are created, one corresponding to the y-coordinate of its top edge, and one corresponding to the $y$-coordinate of its bottom edge. Sort all of these records together. Use a parallel prefix operation to count the number of top and bottom spanning rectangle edges preceding every edge. Use a sort to reunite top and bottom records representing the same rectangle. For every rectangle, four important pieces of information are now known, namely, the number of top and bottom spanning rectangle edges that precede each of its two horizontal edges. For every rectangle, if all four pieces of data are identical then the rectangle is not intersected by a spanning



rectangle, while otherwise it is. The rectangles then report back to the representative that created them, and the spanning rectangles are then discarded.

Next, each processor updates the left and right limits of its slab and the algorithm proceeds recursively. Since the spanning rectangles are discarded before the recursive call, no rectangle can ever have more than 2 representatives (and up to 6 spanning rectangle representatives) at any one time. The time of the algorithm satisfies the recurrence $T(n) = \Theta(n^{1/2}) + T(n/4)$. Therefore, the algorithm finishes in the time claimed. ·

By applying a similar technique, this result can be extended to circles, assuming that each circle is represented by a record consisting of its center and radius.

**Corollary 4.27** *Given the descriptions of n or fewer circles, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time every circle can determine whether or not it is intersected by another circle.*

Consider the problem of determining the area covered by a set of iso-oriented planar rectangles. Notice that if the set of rectangles were nonintersecting, then this would be trivial. However, such a restriction is unnecessary. The algorithm presented below is a straightforward adaptation of the algorithm associated with Theorem 4.26, where at the beginning of each stage of recursion, the spanning rectangles are used to eliminate portions of the nonspanning rectangles that they intersect.

**Theorem 4.28** *Given n or fewer iso-oriented planar rectangles, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time all processors can know the total area covered by the rectangles.*

*Proof.* The algorithm is similar to the algorithm associated with Theorem 4.26. At each stage, when the spanning rectangles are sent to each slab, they first determine the total measure of the $y$-axis that they cover. The total area covered by the spanning rectangles is this measure times the width of the slab. Each representative of a nonspanning rectangle in the slab now 'eliminates' the portion of itself that overlaps spanning rectangles. That is, for a given nonspanning rectangle $R$ with top $y$-coordinate $y_1$ and bottom $y$-coordinate $y_2$, the total measure $M_1$ of the $y$-axis covered by spanning rectangles below $y_2$ and the total measure $M_2$ of the $y$-axis covered by spanning rectangles below $y_1$ is determined.

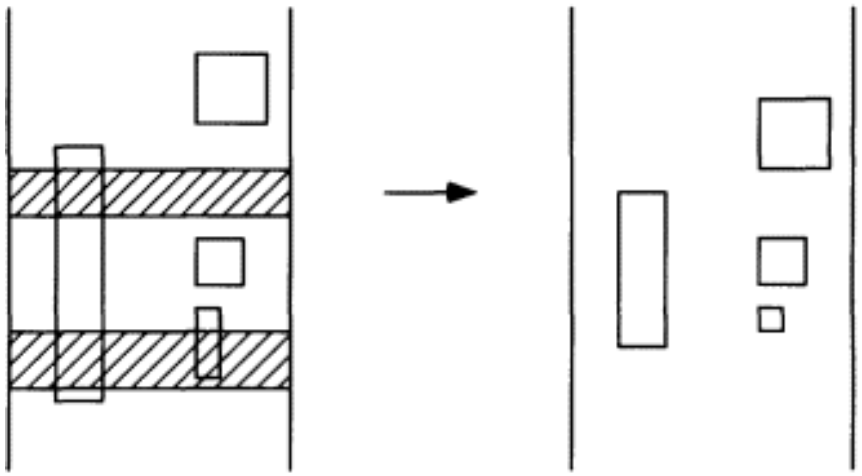


Figure 4.12:
'Cutting out' the spanning rectangles from a slab.

The processor responsible for $R$ subtracts $M_1$ from $y_2$ and $M_2$ from $y_1$. Figure 4.12 illustrates this. It has the effect of 'cutting out' the spanning rectangles and moving everything else down. The algorithms to determine these measures can be complete in $\Theta(n^{1/2})$ time and are left to the reader. A final semigroup (i. e., associative binary) operation will determine the total area covered by the rectangles.

In the remainder of this section, results are stated that either follow or are closely related to other results of this section. For certain packing and layout problems, it is often desirable to know the area of each rectangle that is covered by other rectangles. An algorithm to solve this problem is quite similar to the previous algorithm, with the exception being that the recursion takes place within the spanning rectangles, and the total area covered within the spanning rectangle is recorded, so as to be subtracted from the area of the spanning rectangle.

**Theorem 4.29** *Given n or fewer iso-oriented planar rectangles, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time every processor can know the area that its rectangle covers and that is covered by no other rectangle.*

A problem related to some of the rectangle intersection problems just described is that of determining the maximum number of overlapping



rectangles. A solution to this problem can be used to solve the *fixed-size rectangle placement problem*. That is, given a set of planar points and a rectangle, determine a placement of the rectangle in the plane so that the number of points covered by the rectangle is maximal. Optimal mesh algorithms solving these problems are presented in [LuVa86b]. The results are stated in the Theorem and Corollary that follow.

**Theorem 4.30** *Given n or fewer iso-oriented planar rectangles, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time every processor can know the maximum number of overlapping rectangles.* ·

**Corollary 4.31** *Given a set of planar points and a fixed rectangle, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time a placement of the rectangle in the plane that will cover a maximal number of points can be determined.* ·

By combining the basic technique of multidimensional divide-and-conquer with the use of spanning rectangles, and using the fact that the spanning rectangles have particularly simple properties, the following two results are obtained.

**Theorem 4.32** *Given n or fewer iso-oriented planar rectangles, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time every processor can know a nearest neighboring rectangle to the one that it contains.* ·

**Theorem 4.33** *Given a total of n or fewer iso-oriented planar rectangles and planar points, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time every processor containing a point can determine the number of rectangles containing the point, and every processor containing a rectangle can determine the number of points contained in the rectangle.* ·

A minor modification to Theorem 4.32 will yield an optimal mesh solution to the *all-nearest neighbor problem for circles*.

**Corollary 4.34** *Given n or fewer nonintersecting circles, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time every processor can know the nearest neighboring circle to the one that it contains.* ·



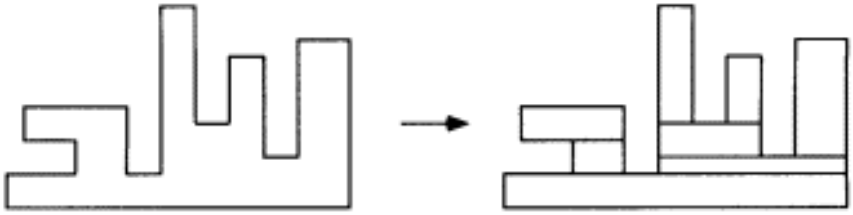


Figure 4.13:
Decomposing an orthogonal polygon into iso-oriented rectangles.

Problems in VLSI layout often involve more than iso-oriented rectangles. Frequently the objects that need to be considered are simple polygons with iso-oriented sides, referred to as *orthogonal polygons*. It is quite straightforward to add horizontal line segments which decompose each orthogonal polygon into a collection of rectangles overlapping only along their edges, where the number of rectangles is less than the number of initial edges, and where the process takes only $\Theta(n^{1/2})$ time. (See Figure 4.13.) Having done this, each of the results in Theorems 4.28, 4.29, 4.32, and 4.33 can be extended to orthogonal polygons, still requiring only $\Theta(n^{1/2})$ time. The only difference is that Theorem 4.32 must be extended to handle labeled rectangles, finding the nearest neighbor of a different label.

**Theorem 4.35** *a) Given multiple simple polygons with iso-oriented sides, represented by n or fewer labeled line segments, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time the total area covered by the polygons can be determined, a nearest neighbor of each polygon can be determined, and the area uniquely covered by each polygon can be determined.*

*b) Given a total of n or fewer labeled line segments (representing isooriented simple polygons) and planar points, distributed one per processor on a mesh computer of size n, in $\Theta(n^{1/2})$ time every processor containing a point can determine the number of polygons containing the point, and every processor containing a line segment of a polygon can determine the number of points contained in its polygon.*



### 4.10 Voronoi Diagram

The Voronoi diagram is a well known structure in computational geometry that is used to derive efficient serial solutions to a host of proximity queries, including many of those that have been considered in this chapter. Given a set $S$ of $n$ planar points, the *Voronoi diagram of S* is the union of $n$ convex polygons $V(p_i)$, $p_i \in S$, where $V(p_i)$ is the convex polygon associated with point *pi* that marks the region of the plane that is closer to point $p_i$ than to any of the other $n$ - 1 points of $S$. See Figures 4.14 - 4.16. The Voronoi diagram of $S$ has at most $2n$ - 5 vertices and at most $3n$ - 6 edges. In [ShHo75], an optimal $O(n \log n)$ serial algorithm is presented to compute the Voronoi diagram of a set of $n$ planar points by a divide-and-conquer solution. The reader is referred to [PrSh85], and the references contained therein, for an in-depth examination of Voronoi diagrams, serial algorithms to compute properties of Voronoi diagrams, and applications of Voronoi diagrams.

#### 4.10.1 Algorithm

In this section, an asymptotically optimal mesh algorithm to compute the Voronoi diagram of a set of planar points, based on the algorithm described in [JeLe90], is presented. This result is then used in Section 4.10.2 to give optimal mesh solutions to a number of geometric properties, some of which have already been solved by other methods in this chapter.

Given a set $S$ of $n$ planar points, if $p_i, p_j \in S$, then the set of points closer to $p_i$ than to $p_j$ is just the half-plane containing $p_i$ that is defined by the perpendicular bisector (*bisector*) of $\overline{p_i p_j}$. This half-plane will be denoted $H(p_i, p_j)$. The locus of points closer to $p_i$ than to any other point, is a convex region, given by $V(p_i) = \cap_{i \neq j} H(p_i, p_j)$, which is referred to as the *Voronoi polygon associated with* $p_i$. As stated previously, the Voronoi diagram of $S$ is the union of all $V(p_i)$, $1 \leq i \leq n$. See Figure 4.16. The edges of $V(p_i)$ are called *Voronoi edges*, and the end vertices of such as edge are referred to as *Voronoi vertices*. Each Voronoi edge is a continuous portion of the bisector of two points $p_i, p_j \in S$, denoted $B(p_i, p_j)$. Given a *directed* edge $e = B(p_i, p_j)$, where bisector point $p_i$ lies to the left of $e$ and bisector point $p_j$ lies to the right of $e$, then $p_i$ is called the *left bisector point of e* and $p_j$ is called the *right bisector point of e*. (Notice that *left* and *right* are in reference to the relationship between the bisector points and the directed edge in the plane, and have nothing to do with their relationship between the two

Figure 4.14:
A set $S$ of planar points.

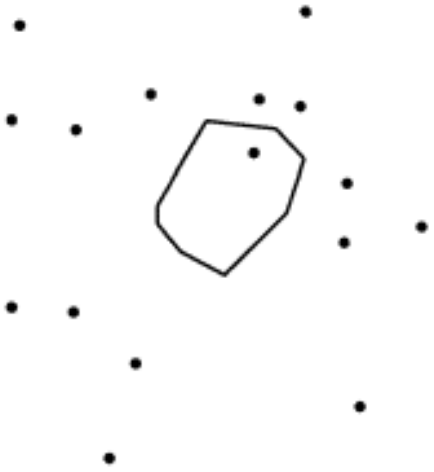


Figure 4.15:
The Voronoi polygon of a selected point in $S$.

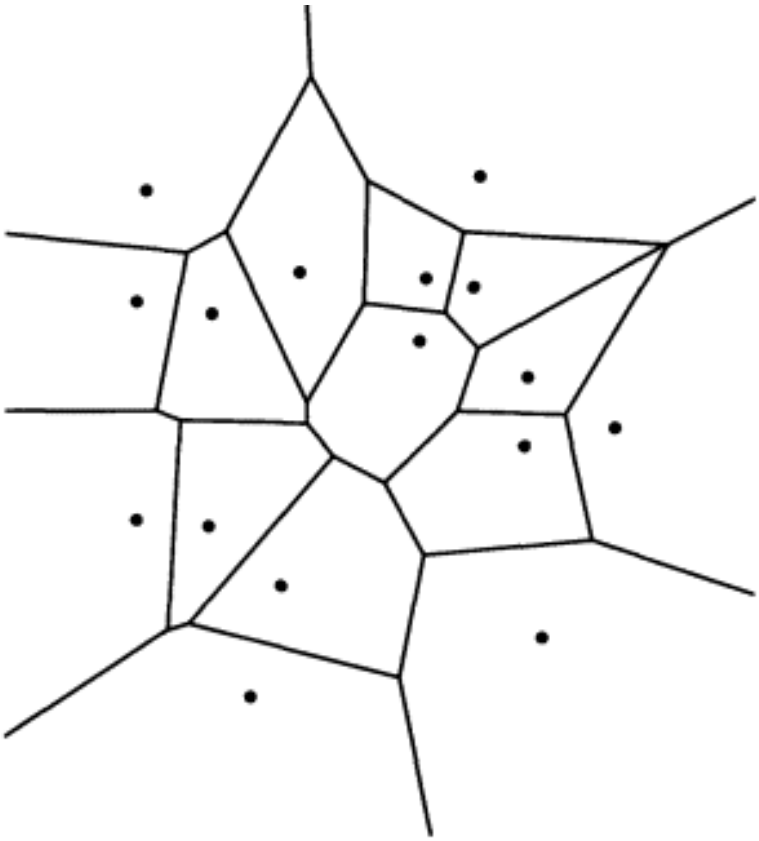


Figure 4.16:
The Voronoi diagram of $S$, denoted $V(S)$.

Page 197

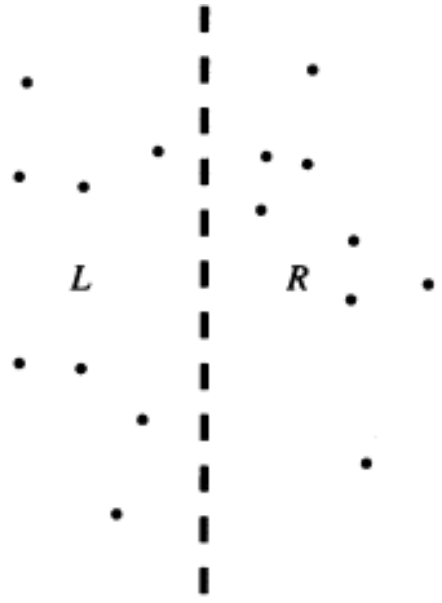



Figure 4.17:
Subsets $L$ and $R$ of $S$ are linearly separable, with each set containing half of the points.

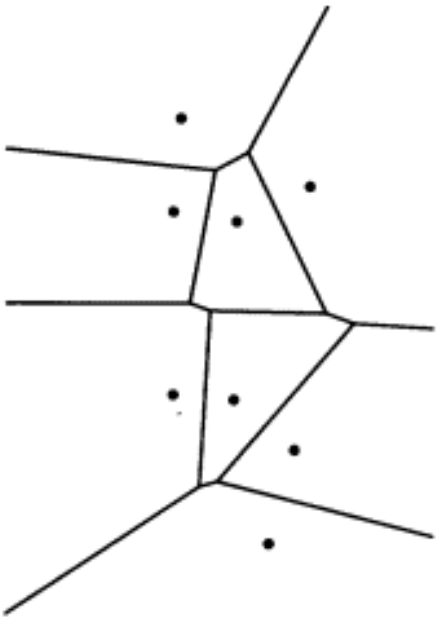


Figure 4.18:
The Voronoi diagram of $L$, denoted $V(L)$.

Page 198

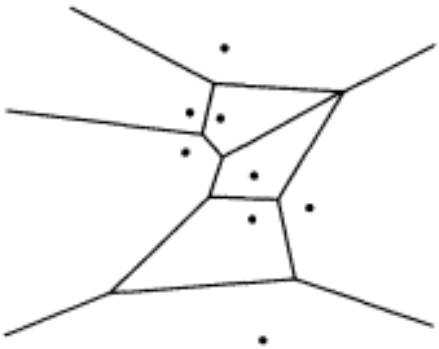


Figure 4.19:
The Voronoi diagram of $R$, denoted $V(R)$.

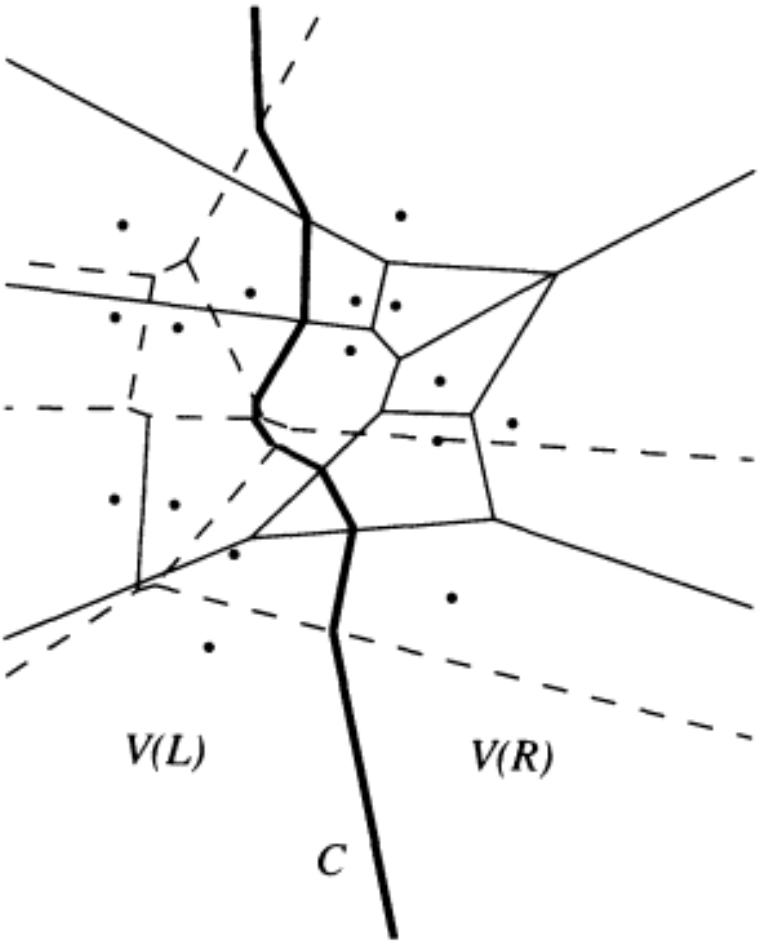



Figure 4.20:
The Voronoi diagram of $L$, the Voronoi diagram of $R$, and the dividing chain C.

Page 199

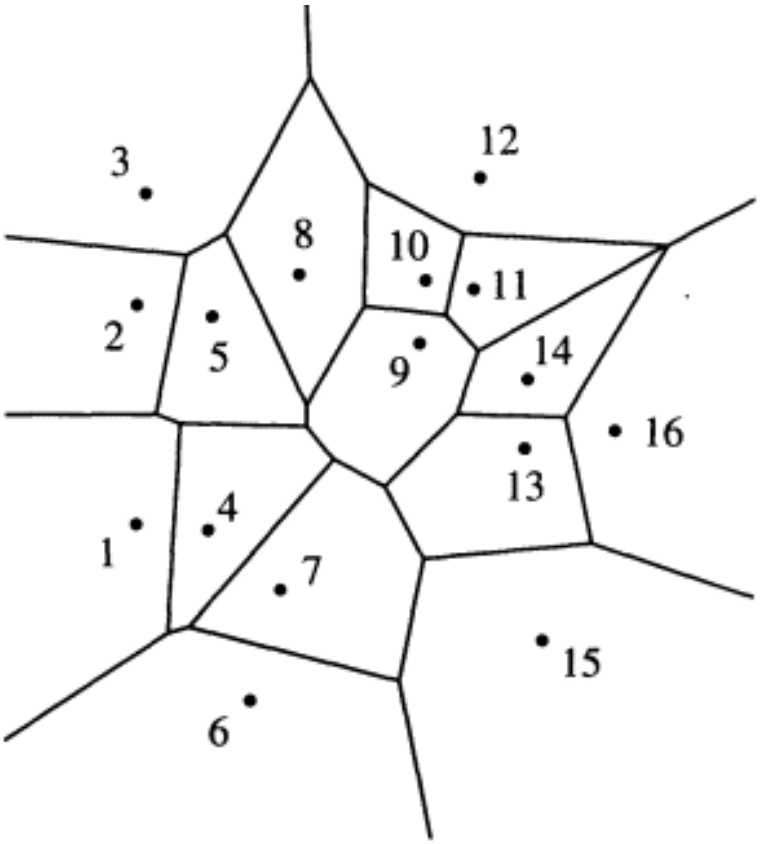


Figure 4.21:
The Voronoi diagram of $S$ with the points labeled.



bisector points.)

The mesh solution presented in this section is based on a divide-and-conquer paradigm. Consider a partition of $S$ into two subsets, $L$ and $R$, where all points of $L$ lie to the left of a separating line, and all points of $R$ lie to the right of the line, as shown in Figure 4.17. Let $C$ be the collection of Voronoi edges in $V(S)$ which are determined by polygons representing one point from $L$ and one point from $R$. Orient $C$ so that the points in $L$ and $R$ lie to the left and right of $C$, respectively, as the edges of $C$ are traversed from bottom up. $C$ will be referred to as the *dividing chain of $L$ and $R$*. See Figure 4.20.

A general mesh algorithm for constructing the Voronoi diagram of a set $S$ of $n$ planar points follows.

1. Partition a set $S$ of $n$ planar points into linearly separable sets $L$ and $R$, each of which has $n/2$ points, such that the points of $L$ lie to the left of the points of $R$.

2. Recursively construct $V(L)$ and $V(R)$. (See Figures 4.18 and 4.19.)

3. Merge $V(L)$ and $V(R)$ into $V(S)$, *as* follows. (See Figure 4.20.)

   (a) Construct $C$, the dividing chain of $L$ and $R$, as follows.

      i. Determine the set $B$ of edges from $V(L)$ and $V(R)$ that are intersected by $C$.

      ii. Arrange the edges in $B$ by the order in which they are intersected by $C$.

   (b) Discard any unnecessary portions of edges in $V(L)$ and $V(R)$, as determined by $C$.

The key to the algorithm is the merge step. For each edge e in either $V(L)$ or $V(R)$, it can be decided whether or not that edge is in $B$ (the set of edges intersected by the dividing chain $C$) by determining for each of its end vertices whether it is closer to $L$ or to $R$. Given an edge $e = B(q_1, q_2)$ of $V(L)$ with end vertex $v_1$ that lies in Voronoi region $V(q_j)$ of $V(R)$, it can be decided if $v_1$ is closer to $L$ or to $R$ by comparing the distances $d(q_j, v_1)$ and $d(q_1, v_1)$. In order to determine the Voronoi region of $V(R)$ that $v_1$ lies in, the algorithm associated with Corollary 4.19 may be used to determine the Voronoi edge of $V(R)$ that is directly above $v_1$. This information yields the Voronoi polygon of $R$ that contains $v_1$. This method can be used so that all Voronoi vertices of $L$ and all Voronoi vertices of $R$ can simultaneously determine if they are closer to $L$ or to $R$.



Let $E_l$ and $E_r$ be the set of Voronoi edges of $V(L)$ and $V(R)$, respectively. The set of edges $E = E_l \cup E_r$ can be partitioned into three sets, as follows.

1. $E_{ll}$: the set of edges both of whose end vertices are closer to $L$.

2. $E_{lr}$,: the set of edges one of whose end vertices is closer to $L$ and the other to $R$.

3. $E_{rr}$: the set of edges both of whose end vertices are closer to $R$.

In [JeLe90] it has been shown that for e $\in E_l$,

- if $e \in E_{ll}$ then $C$ does not intersect $e$,
- if e $\in E_{lr}$ then $C$ intersects $e$ exactly once, and
- if $e \in E_{rr}$ then *if* $C$ intersects e, it does so twice.

Similarly, for $e \in E_r$,

- if e $\in E_{rr}$ then C does not intersect $e$,
- if $e \in E_{lr}$ then C intersects $e$ exactly once, and
- if $e \in E_{ll}$ then *if* $C$ intersects e, it does so twice.

After identifying the edges $B$ that are intersected by $C$, they must be sorted by the order in which they intersect $C$, as $C$ is traversed from the bottom up. Let $B_l$ be the Voronoi edges of $V(L)$ intersected by $C$, and $B_r$ be the Voronoi edges of $V(R)$ intersected by $C$, where $B = B_l \cup B_r$. For any two edges $e_i$ $e_j \in B$, define $e_i < e_j$ if $e_i$ is intersected by $C$ before $e_j$ in the sense of traversing $C$ from the bottom up. For each $e_i \in B$, the end vertex closer to $L$ is denoted $lv(e_i)$ and the end vertex closer to $R$ is denoted $rv(e_i)$. Notice that $lv(e_i)$ lies to the left of $C$, while $rv(e_i)$ lies to the right of $C$.

For the situation where $e_i \in B$ is intersected twice by $C$, $e_i$ is divided into two parts, $e_{i1}$ and $e_{i2}$, such that each part is *i*) intersected by C exactly once and *ii*) has one end vertex closer to $L$ and the other closer to $R$. Specifically, $e_i = B(p_j, p_k) \in B_l \cap E_{rr}$ is divided into two parts, $e_{i1}$ and $e_{i2}$, such that $rv(e_{i1}) = lv(e_{i2}) = q_i$, where $q_i$ is the intersection of $B(p_j,p_k)$ and the horizontal line passing through $p_k$. Similarly, $e_i = B(p_j, P_k)$ $\in Br \cap E_{ll}$ is divided into two parts, $e_{i1}$, and $e_{i2}$, such that $rv(e_{i1}) = lv(e_{i2}) = q_i$, where $q_i$ is the intersection of $B(p_j, p_k)$ and the horizontal line passing through $p_j$.



Each $e_i \in B$ is assumed to be *directed* from $lv(e_i)$ to $rv(e_i)$. The left and right bisector points of $e_i$ are denoted by $lp(e_i)$ and $rp(e_i)$, respectively. The dividing chain $C$ intersects a sequence of restricted Voronoi polygons while crossing edges in $B$. (The reader is referred to [PrSh85] for details of constructing $C$ on a serial machine in time linear in the number of edges.) In [JeLe90], it is shown how to order any two edges $e_i$, $e_j$ *e* $B$ with respect to the order in which they are intersected by $C$.

Define $MINy(e)$ and $MAXy(e)$ of an edge $e \in B$ to be the minimum and maximum $y$-values of the two end vertices of $e$, respectively. Two edges $e_i$ and $e_j$ are said to be *y-disjoint* if $MINy(e_i) > MAXy(e_j)$ or $MINy(e_j) > MAXy(e_i)$. Further, $e_i$ is said to be *left* of $e_j$ if $e_i$ lies totally to the left of the directed line from $lv(e_j)$ to $rv(e_j)$ with respect to the direction. The notion of *right* is defined similarly. If $e_i$ and $e_j$ are $y$-disjoint, then it is easy to order $e_i$ and $e_j$ since $C$ is monotone with respect to $y$. However, if $e_i$ and $e_j$ are not $y$-disjoint, then $e_i$ and $e_j$ can be ordered according to the following. (See Figure 4.22 for intuitive arguments. The reader is referred to [JeLe90] for details and proof of correctness.)

1. If $e_i$ and $e_j$ intersect at a point m, then

   (a) If both $e_i$ and $e_j$ are in either $B_l$ or $B_r$, then $e_i < e_j$ if $e_i$ lies to the right of $e_j$, otherwise $e_j < e_i$.

   (b) If exactly one of $e_i$ and $e_j$ is in $B_1$, then

      i. $e_i < e_j$ if $m$ is closer to $L$ than to $R$ and $lv(e_j)$ lies to the right of $e_i$, otherwise

      ii. $e_i < e_j$ if $m$ is closer to $R$ than to $L$ and $lv(e_i)$ lies to the right of $e_j$, otherwise

      iii. $e_j < e_i$.

2. If $e_j$ lies to the left of $e_i$, then

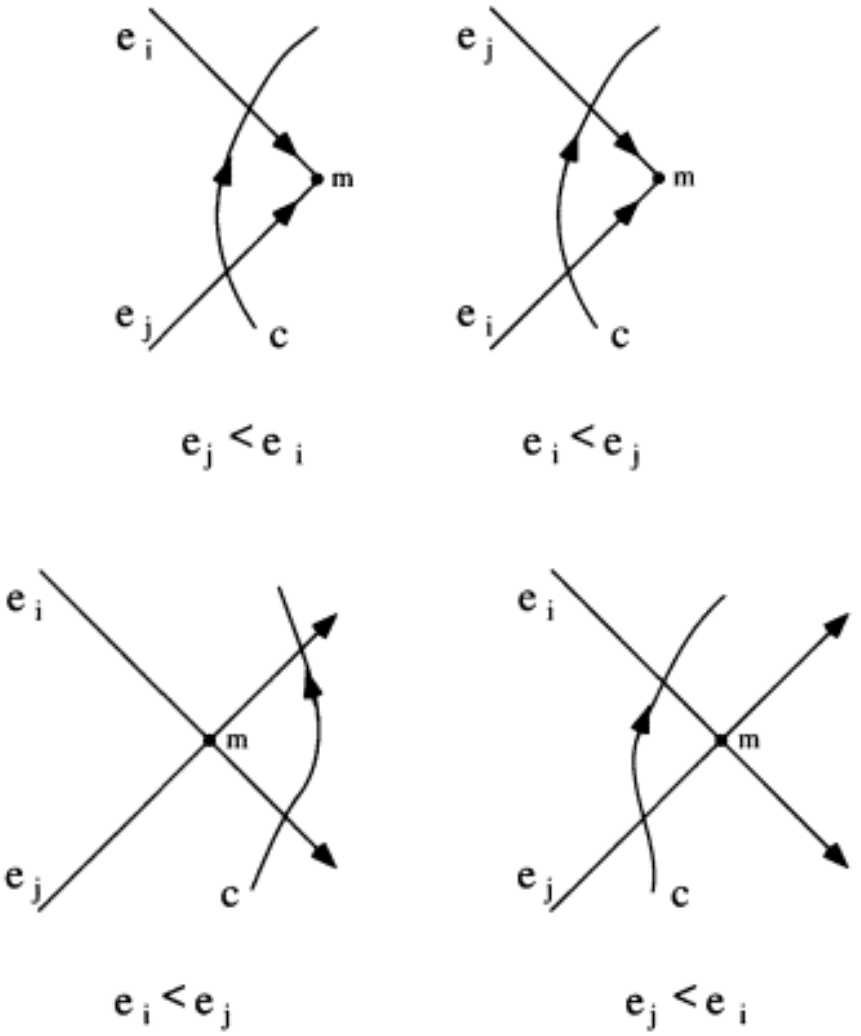
   (a) $e_j < e_i$ if $y(rv(e_j)) < y(lv(e_i)) < y(rv(e_i))$ or $y(lv(e_j)) < y(rv(e_i)) < y(lv(e_i))$, otherwise

   (b) $e_i < e_j$.

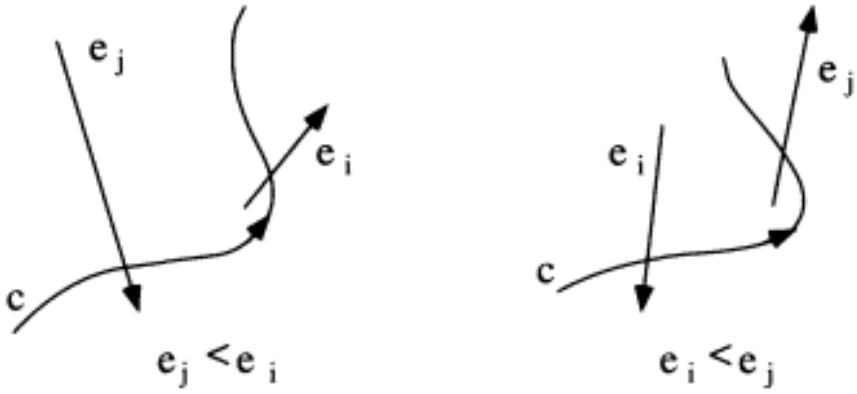
3. If $e_j$ lies to the right of $e_i$, then

   (a) $e_i < e_j$ if $y(lv(e_i)) < y(rv(e_i)) < y(lv(e_j))$ or $y(rv(e_i)) < y(lv(e_i)) < y(rv(e_j))$, otherwise

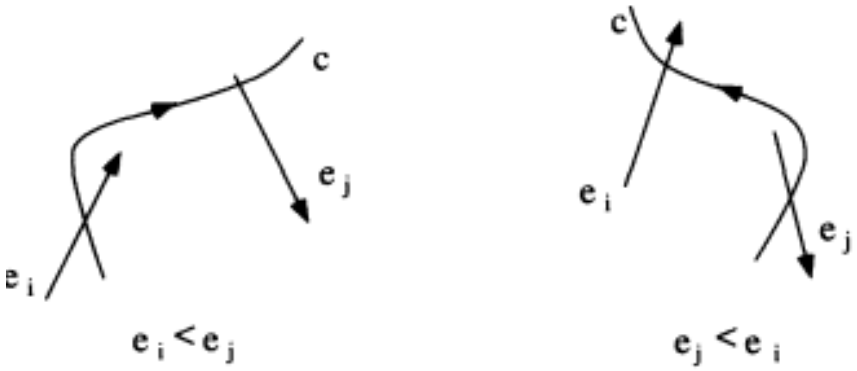
   (b) $e_j < e_i$.




(a) Examples for the situation where $e_i$ and $e_j$ intersect at a point $m$.




(b) Examples for the situation where $e_j$ lies to the left of $e_i$.




(c) Examples for the situation where $e_j$ lies to the right of $e_i$.

Figure 4.22:
Ordering $e_i$ and $e_j$ with respect to the traversal of $C$.

Since the set of edges $B$ that are intersected by $C$ can be ordered, the details of the merge step in the algorithm outlined at the beginning of the section is complete. An optimal $\Theta(n^{1/2})$ time mesh algorithm is now presented for constructing the Voronoi diagram of a set $S$ of $n$ planar points, initially distributed one per processor on a mesh of size $n$. Notice that no step of the algorithm requires more than $\Theta(n^{1/2})$ time.

1. Sort the points of $S$ by $x$-coordinate. Let $L$ be the set of points in processors $P_1, P_2, \ldots, P_{n/2}$, and $R$ be the set of points in processors $P_{n/2+1}, P_{n/2+2}, \ldots, P_n$.

2. Recursively find $V(L)$ and $V(R)$.

3. Merge $V(L)$ and $V(R)$ into $V(S)$ by constructing the dividing chain $C$ and discarding portions of Voronoi edges in $V(L)$ to the right of $C$ and portions of Voronoi edges in $V(R)$ to the left of $C$.

(a) Find the set of edges $B$ in $V(L)$ and $V(R)$ that are intersected by $C$.

i. Use the algorithm associated with Corollary 4.19 to compute for each Voronoi vertex in $V(L)$, the Voronoi polygon of $V(R)$ that the vertex is contained in. Similarly, for each Voronoi vertex in $V(R)$, determine the Voronoi polygon of $V(L)$ that the vertex is contained in. Using this information, determine for each such vertex whether it is closer to $L$ or to $R$.



ii. Based on the discussion preceding this algorithm, determine for each Voronoi edge in $V(L)$ and $V(R)$ whether or not it is intersected by $C$. These edges represent the set $B$.

iii. For the purpose of constructing the dividing chain, conceptually discard edges which lie totally to the left or totally to the right of $C$.

(b) Construct the dividing chain $C$.

i. If the number of edges in $B$ is less than 2, then

A. find the edges of $C$ directly from the starting and terminating bisectors. Notice that the starting and terminating bisectors correspond to the lower and upper tangent lines, respectively, between hull($L$) and hull($R$), and can be determined by applying the algorithm associated with Theorem 4.1 to $L$ and $R$. However, as noted in [JeLe90], in general this should not occur, unless the 11 metric is used as the distance metric.

B. Exit.

ii. Based on the discussion preceding this algorithm, sort the edges in $B$ according to the order in which they are intersected by $C$, viewing the traversal of $C$ from bottom up.

A. Let $B = \{e_1, e_2, \ldots, e_k\}$ be sorted so that $e_1 < e_2 < \ldots < e_k$.

B. Let $E_c = \{e_{0'}, e_{1'}, \ldots, e_{k'}\}$ denote the set of *edges of* C such that $e_{i'}$ is immediately above $e_i$ and such that $e_{o'}$ and $e_{k'}$ are the starting and terminating edges of C, respectively.

iii. Find the bisector points for each edge $e_i, \in E_c$ as follows. (Refer to Figure 4.20.)

A. Compute for each edge $e_i \in B_l$, the greatest edge $e_{gi} < e_i$, where $e_{gi} \in B_r$. (The case in which $e_i \in B_r$ is handled similarly.)

B. If there is no such greatest edge $e_{gi}$, then store the least edge in $B_r$ as $e_{gi}$. Record bisector points $lp(e_i)$ and $rp(e_{gi})$ for $e_i, \in E_c$.

C. If there is such a greatest edge $e_{gi}$, then record bisector points $lp(e_i)$ and $lp(e_{gi})$ for edge $e_{i'} \in E$, since



$e_{i'}$, lies in the region $V(lp(e_i))$ of $V(L)$ and $V(lp(e_{gi}))$ of $V(R)$.

iv. Find the end vertices of all edges in $E_c$ and modify, for each edge in $B$, its end vertices. Each processor containing an edge $e_i$ performs the following.

A. Locate one end vertex $c_i$, which is the intersection of $e_i$ and $e_{i'}$.

B. Obtain the other end vertex from an adjacent processor.

C. Discard the portion of the edges in $V(L)$ and $V(R)$ to the right and left of the dividing chain, respectively, by replacing the right and left end vertices, respectively, of $e_i$ in $B_l$ and $B_r$ with $c_i$.

(c) For each Voronoi edge $e = B(p_i, p_j)$, create two records. One record will have $p_i$ as the key and the other will have $p_j$ as the key. Sort all records by key fields so as to create ordered intervals out of the Voronoi polygons.

**Theorem 4.36** *Given a set S of n or fewer planar points, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time the Voronoi diagram of S can be constructed.*

### 4.10.2 Applications

Preparata and Shamos [PrSh85] have shown that the Voronoi diagram is useful for solving a number of proximity problems. They also cite examples from fields such as archaeology, ecology, and molecular modeling, for which the Voronoi diagram is an end in itself. In this section, optimal mesh solutions to a number of proximity problems are stated that are based on Theorem 4.36. It should be noted that some of these problems have been solved directly in previous sections of this chapter.

The first problem, which was previously discussed in Section 4.5 and solved in optimal time on a mesh by the algorithm associated with Theorem 4.7, is the *all-nearest neighbor problem*. Given a set $S$ of points, once $V(S)$ is constructed, every point $p \in S$ can find a nearest neighbor by examining the edges of $V(p)$. This can be done in the required time by performing a semigroup (i.e., associative binary) operation within ordered intervals. (When the algorithm associated with Theorem 4.36 terminates, the Voronoi polygon associated with each point is stored in an ordered interval of the mesh.)



**Corollary 4.37** *Given $n$ or fewer planar points, distributed one per processor on a mesh computer of size $n$, the all-nearest neighbor problem for points can be solved in $\Theta(n^{1/2})$ time.* ·

A semigroup (i.e., associative binary) operation over the all-nearest neighbor results, gives an asymptotically optimal algorithm to solve the closest pair problem.

**Corollary 4.38** *Given $n$ or fewer planar points, distributed one per processor on a mesh computer of size $n$, the closest pair problem for points can be solved in $\Theta(n^{1/2})$ time.* ·

Given a set of planar points $S$, a point $p$ is an extreme point of $S$ if and only if the Voronoi polygon of $p$ is unbounded. Therefore, given a set $S$ of planar points, the extreme points of $S$ that represent hull($S$) can be detected by constructing the Voronoi diagram of $S$ and then using a concurrent read within ordered intervals to detect for each point $p \in S$ whether or not $V(p)$ is bounded (c.f., the algorithm associated with Theorem 4.15).

**Corollary 4.39** *Given a set $S$ of $n$ or fewer planar points, distributed one per processor on a mesh computer of size $n$, the extreme points of $S$ can be identified in $\Theta(n^{1/2})$ time.* ·

The straight line dual of the Voronoi diagram is important for a number of applications. This diagram, called the *Delaunay diagram*, represents the graph embedded in the plane obtained by adding a straight-line segment between each pair of points of $S$ whose Voronoi polygons share an edge. Notice that the diagram may be unusual in that an edge and its dual do not necessarily intersect. One immediate property of the Delaunay diagram is that it gives a triangulation of $S$.

**Corollary 4.40** *Given a set $S$ of $n$ or fewer planar points, distributed one per processor on a mesh computer of size $n$, in $\Theta(n^{1/2})$ time the Delaunay triangulation of $S$ can be determined.* ·

In Section 4.5, a nonoptimal mesh solution to the minimal-distance spanning tree problem for planar point data was mentioned, referring the reader to this section for an optimal solution. Given a collection $S$ of planar points, a spanning tree can be constructed by using the points

as vertices and straight lines between them as edges. A *minimal-distance spanning tree* is a spanning tree of $S$ which minimizes the sum of the Euclidean lengths of the tree edges. A minimal-distance spanning tree of a set of planar points may be constructed in $\Theta(n^{1/2})$ time on a mesh by constructing the Delaunay diagram in $\Theta(n^{1/2})$ time, and then applying the $\Theta(n^{1/2})$ time minimum-weight spanning tree algorithm for graphs appearing in [ReSt].

**Corollary 4.41** *Given n or fewer planar points, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time a minimal-distance spanning tree can be constructed.* ·

In the remainder of this section, problems are discussed that have not been previously mentioned in this chapter. Given two sets of planar points, $S_1$ and $S_2$, a solution to the *all-nearest neighbor between two sets* problem requires finding *i*) for every point $p_i \in S_1$, a nearest neighboring point in $S_2$, and *ii*) for every point $P_2 \in S_2$, a nearest neighboring point in $S_1$. This problem is simply solved by constructing $V(S_1)$ and $V(S_2)$, and then using the algorithm associated with Corollary 4.19 to find, for every point, the Voronoi polygon of the other set that it resides in.

**Corollary 4.42** *Given n or fewer planar points representing sets* $S_1$ *and* $S_2$*, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time the all-nearest neighbor between two sets problem can be solved.* ·

Given a set $S$ of planar points, a *largest empty circle* in $S$ is a largest circle with the properties that *i*) its center is internal to *hull*($S$) and *ii*) it contains no points of $S$ in its interior. It can be shown that the center of a largest empty circle in $S$ lies either at a Voronoi vertex of $S$ or at the intersection of a Voronoi edge and an edge of hull($S$). Computing the largest empty circle centered at a Voronoi vertex of $S$ is straightforward, as is computing the largest empty circle for a point that lies along a Voronoi edge. Once the result is known for each such possible center, a simple semigroup (i.e., associative binary) operation determines the global largest. The challenge to this problem is finding the points of intersection between the Voronoi edges and the edges of the convex hull of $S$. It should be noted that a Voronoi edge may intersect at most 2 hull edges, whereas an edge on hull($S$) may intersect multiple Voronoi edges. A solution to this intersection problem is similar to other intersection algorithms presented previously in this chapter, and is left to the reader.



**Corollary 4.43** *Given a set n or fewer planar points, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time the largest empty circle problem can be solved.* ·

Additional problems can be solved by first constructing the Voronoi diagram of a set of points. The reader is referred to [PrSh85, PrLe84, Tous80], and the references contained therein, for a discussion of such problems. Solutions to some of these problems require constructing generalized Voronoi diagrams or farthest-neighbor Voronoi diagrams. The properties of the (nearest-neighbor) Voronoi diagram are similar enough to those of the farthest-neighbor Voronoi diagram, so that the farthest-neighbor Voronoi diagram can be constructed in asymptotically optimal time on a mesh by an algorithm similar to the one described in the previous section [JeLe90].

For instance, given a set $S$ of $n$ planar points, distributed one per processor on a mesh computer of size $n$, once the farthest-neighbor Voronoi diagram, denoted *FVor*($S$), is constructed in $\Theta(n^{1/2})$ time, a $\Theta(n^{1/2})$ time solution to the *all-farthest neighbor problem* is immediate.

**Corollary 4.44** *Given a set of n or fewer planar points, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time the all-farthest neighbor problem can be solved.* ·

In addition, once *FVor*(*S*) is constructed, the smallest enclosing circle of *S* can be determined in $\Theta(n^{1/2})$ time [JeLe90].

**Corollary 4.45** *Given a set S of n or fewer planar points, distributed one per processor on a mesh computer of size n, in* $\Theta(n^{1/2})$ *time the smallest enclosing circle of S can be determined.* ·

## 4.11 Further Remarks

In the early 1980's, efficient parallel algorithms to solve convex hull problems involving point data began to appear [Akl83, Chow81, NMB81]. In 1984, Miller and Stout [MiSt84b] published a paper that included efficient parallel algorithms to solve several problems involving geometric properties and distances on a mesh computer, and Chazelle [Chaz84] published a paper that gave efficient algorithms to solve some distance



and intersection problems on a 1-dimensional systolic computer. Subsequently, efficient parallel algorithms have been presented to solve additional problems on a variety of models [ACGO88, AkLy93, ADMR94, AtGo86a, Dehn86a, JeLe90, LuVa85, MiSt88b, MiSt89a].

Given $n$ or fewer planar points, distributed one per processor on a mesh computer of size $n$, algorithms were presented in this chapter to determine a number of formal geometric structures in $\Theta(n^{1/2})$ time. Since it takes $\Omega(n^{1/2})$ time for data to travel across a mesh computer of size $n$, all of the algorithms have optimal worst-case running times and are significantly faster than the $\Omega(n)$ time required for a serial computer to process $O(n)$ pieces of data. (In fact, many of these problems require $\Omega(n \log n)$ time on a serial computer.)

The algorithms presented in this chapter employ different approaches to solving geometric problems than those that have been explored for these problems on a serial computer. A variety of general techniques have been presented for the mesh computer, including multidimensional divide-and-conquer and grouping operations to solve parallel search problems. These techniques should be applicable to constructing efficient solutions to a wide variety of problems.

There may be situations in which the algorithms presented in this chapter can be slightly modified to produce even faster solutions to problems. For instance, when multiple sets of objects exist, each initially stored in a subsquare of size no more than $D$, then solutions to many of the problems addressed in this chapter may be extended so that the necessary result can be determined simultaneously for all such objects in $\Theta(D^{1/2})$ time.

In this chapter, the concentration was on 2-dimensional meshes, since they are the ones most commonly built. A *j-dimensional mesh of size n* (where *n* is the $j^{th}$ power of some integer) has *n* processors arranged in a *j*-dimensional cubic lattice. Processors $P_{s_1,\ldots,s_j}$ and $P_{t_1,\ldots,t_j}$ are connected if and only if $\sum_{i=1}^{j} |s_i - t_i| = 1$. That is, a generic processor in a *j*-dimensional mesh has $2^j$ neighbors. In the O-notational analyses of algorithms for *j*-dimensional meshes, it makes sense to consider *j* as fixed. That is, there is no differentiation between a step needing a constant amount of time and one needing $2^j$ units of time. The reason for this is that a processor in a *j*-dimensional mesh is fundamentally different from one in a *k*-dimensional mesh when $j \neq k$, since they have a different number of communication links.

A proximity ordering can be defined for a *j*-dimensional mesh, and all of the data movement operations described in Section 4.2.3 can be



extended to run in $\Theta(n^{1/j})$ time, which again is optimal. Therefore, all of the optimal 2-dimensional mesh algorithms written solely in terms of these data movement operations yield asymptotically optimal $\Theta(n^{1/j})$ time *j*-dimensional mesh algorithms. For a few algorithms, values of constants were chosen to make the recurrence yield the desired result. For *j*-dimensional mesh algorithms, these constants need to be chosen as a function of *j*. For example, in Theorem 4.7, for 2-dimensional meshes, 5 slabs were used in each direction, while for *j*-dimensional meshes, at least $1 + 2^j$ slabs should be used in each direction.

A number of algorithms given in this chapter exploit Bentley's [Bent80] paradigm of *multidimensional divide-and-conquer*. The mesh algorithms presenter in this chapter were careful to avoid possible pitfalls that exist when multidimensional divide-and-conquer is used naively on a parallel machine. The results presented in this chapter demonstrate that multidimensional divide-and-conquer can be applied more simply on mesh computers to solve geometric problems involving points than it can be to solve geometric problems involving polygonal figures. case in this chapter. A point *p* is said to *dominate* a point *q* if and only if the *x* and *y* coordinates of *p* are greater than the respective *x* and *y* coordinates of *q*. (This definition can be naturally extended to higher dimensional data.) By applying a straightforward multidimensional divide-and-conquer technique to *n* or fewer points on a *j*-dimensional mesh of size *n*, dominance problems can be solved in optimal $\Theta(n^{1/j})$ time. Such problems include determining for every point how many other points it dominates, and finding for every point whether or not it is a maxima (i.e., not dominated by any point). Serial algorithms for these problems appear in [Bent80] and optimal 2-dimensional mesh algorithms appear in [Dehn86a].

A variety of *visibility problems* can also be solved in asymptotically optimal time on a mesh computer by exploiting multidimensional divide-and-conquer. The *parallel visibility problem for line segments* can be defined as follows. Given a set of *n* or fewer line segments and a light source located at infinity which emits rays parallel to a given direction *r*, determine the portion of each line segment that is illuminated. Given one line segment per processor on a mesh of size *n*, partition the mesh into two submeshes, each of size *n*/2, and solve the visibility problem recursively for each of the two sets of line segments. The problem has now been reduced from a two dimensional problem to a one dimensional problem, in that the solution for each set is in the form of maximal disjoint intervals with respect to a line at infinity perpendicular to *r*, where each interval represents a portion of a line segment visible from

the light source or the fact that none of the line segments in the set are visible in the interval. A simple one dimensional merge operation involving intervals of line segments completes the solution. In addition to parallel visibility, problems involving perspective visibility (i.e., given a source $q$, determine all points $p$ such that $\overline{pq}$ does not intersect any object in the set) can also be solved in optimal time on a mesh. Details of visibility algorithms for line segments and simple polygons can be found in [Dehn87, Lu88].

Other problems can be solved in asymptotically optimal time on a mesh computer by exploiting multidimensional divide-and-conquer to reduce the problem to the same problem in lower dimensions, such as deciding which iso-oriented boxes are intersected be others. Some algorithms can be extended in a natural fashion to derive optimal algorithms for higher dimensional data, even though they do not use multidimensional divide-and-conquer. For example, for any fixed dimension $j$, the all-nearest neighbor problem for points can be solved in $\Theta(n^{1/j})$ time on a $j$-dimensional mesh by using a straightforward extension of the algorithm associated with Theorem 4.7. For a few problems, such as finding the convex hull, it should be possible to extend to 3-dimensional data in the same time bounds. (In fact, an optimal algorithm is presented in [ADMR94] for solving the 3-dimensional convex hull problem on a 2-dimensional mesh computer.) However, many of the remaining problems seem to either require too much data movement, or the generation of too much data, when the dimension of the input increases. For example, it is known that the convex hull of $n$ points in $d$-dimensional space may have $\Theta(n^{\lfloor d/2 \rfloor})$ faces, so for $d \geq 4$, any algorithm which generates and keeps all the faces will need $\Omega(n^{d/2})$ processors to store them, or else the memory available in each processor must be increased.



# 5
# Tree-like Pyramid Algorithms

## 5.1 Introduction

Pyramid-like parallel computers have long been proposed for performing high-speed low-level image processing [Dyer8la, Dyer8lb, Dyer82, MiSt84c, MiSt85c, Rose84, Stou82c, Stou83c, Tani81, Tani82a, TaK180, Uhr72, Uhr84], and a variety of such machines have been constructed [Buva87, CFLS85, ClMe87, FKLV83, Scha85, SHBV87, Tani82a]. The pyramid has a simple geometry that adapts naturally to many types of problems, and which may have ties to human vision processing. Furthermore, the pyramid can be projected into a regular pattern in the plane, which makes it ideal for VLSI implementation [Dyer81a].

The interconnection topology of the pyramid computer consists of a combination of tree and mesh connections, as outlined in Section 1.2.4. In Chapters 2-4, a variety of efficient mesh algorithms were presented. Notice that such algorithms can run directly on the base mesh of a pyramid. In contrast, in this chapter, pyramid algorithms are presented that use predominantly the child-parent links of the pyramid in order to obtain running times that are poly-logarithmic (i.e., that run in $O(\log^c n)$ time, for c a constant) in the number of processors. In Chapter 6, pyramid algorithms will be given that exploit the pyramid's combination of tree and mesh connections by combining tree and mesh algorithms to give efficient solutions to a variety of problems in image processing, graph theory, and digital geometry when the input is a digitized picture, adjacency/weight matrix, or a set of unordered edges that represent a graph.

A review of the pyramid computer is presented in Section 5.2. The focus of Section 5.3 is on lower bounds for problems on the pyramid. At the beginning of Section 5.4, an algorithm is presented that shows how to initialize the identity registers of all processors in the pyramid in time proportional to the height of the pyramid. The remainder of Section 5.4 concentrates on several standard (quad) tree-type algorithms, where the input is either a digitized black/white picture or a set of values, initially distributed one element per base processor.

In Section 5.5, algorithms are presented that are concerned with convexity properties of a given set of base processors. In Section 5.5.1, convexity algorithms are presented that range in running times from



$\Theta(\log n)$ to decide whether or not a digitized figure is convex and to enumerate the extreme points of a convex figure, to $\Theta(\log^2 n/\log \log n)$ to enumerate the extreme points of a (not necessarily convex) set of base processors. In Section 5.5.1, it is also shown that the extreme points of a figure can be used to solve problems such as determining whether or not two figures are linearly separable, as well as determining a smallest enclosing box, the smallest enclosing circle, and the diameter of a given set of base processors. In Section 5.5.2, some results from digital geometry are combined with convexity algorithms from Section 5.5.1 to show that it is possible to determine in $\Theta(\log n)$ time whether or not a digitized $n^{1/2} \times n^{1/2}$ black/white figure could have arisen as the digitization of a straight line segment.

### 5.2 Definitions

As a convenience, this section reviews the definition of a pyramid computer from Section 1.2.4. A *pyramid computer* (*pyramid*) *of size n* is a machine that can be viewed as a full, rooted, 4-ary tree of height $\log_4 n$, with additional horizontal links so that each horizontal *level* is a mesh. A pyramid of size $n$ has at its base a mesh of size $n$, and a total of $^4/_3 n - ^1/_3$ processors. The levels are numbered so that the base is level 0 and the apex is level $\log_4 n$. A processor at level $i$ is connected via bidirectional unit-time communication links to its 9 neighbors (assuming they exist): 4 siblings at level $i$, 4 children at level $i$ - 1, and a parent at level $i + 1$. (A sample pyramid is given in Figure 5.1.) It is assumed that each processor has a fixed number of words (registers), each of length $\Theta(\log n)$, and that all arithmetic, Boolean, and communication operations with a neighbor take unit time. Each processor contains registers with its level, row, and column coordinates, the concatenation of which are in the processor identification register. (In Section 5.4.1, an algorithm is given that shows how to initialize these registers in $\Theta(\log n)$ time.)

### 5.3 Lower Bounds

A pyramid computer of size $n$ has a communication diameter of $\Theta(\log n)$, meaning that any two processors can exchange messages in $O(\log n)$ time, by communicating via the apex, and some pairs of processors, such as those at opposite corners of the base mesh, require $\Omega(\log n)$ time to exchange messages. This gives a worst-case lower bound of $\Omega(\log n)$ time on any problem that may require information to be exchanged between

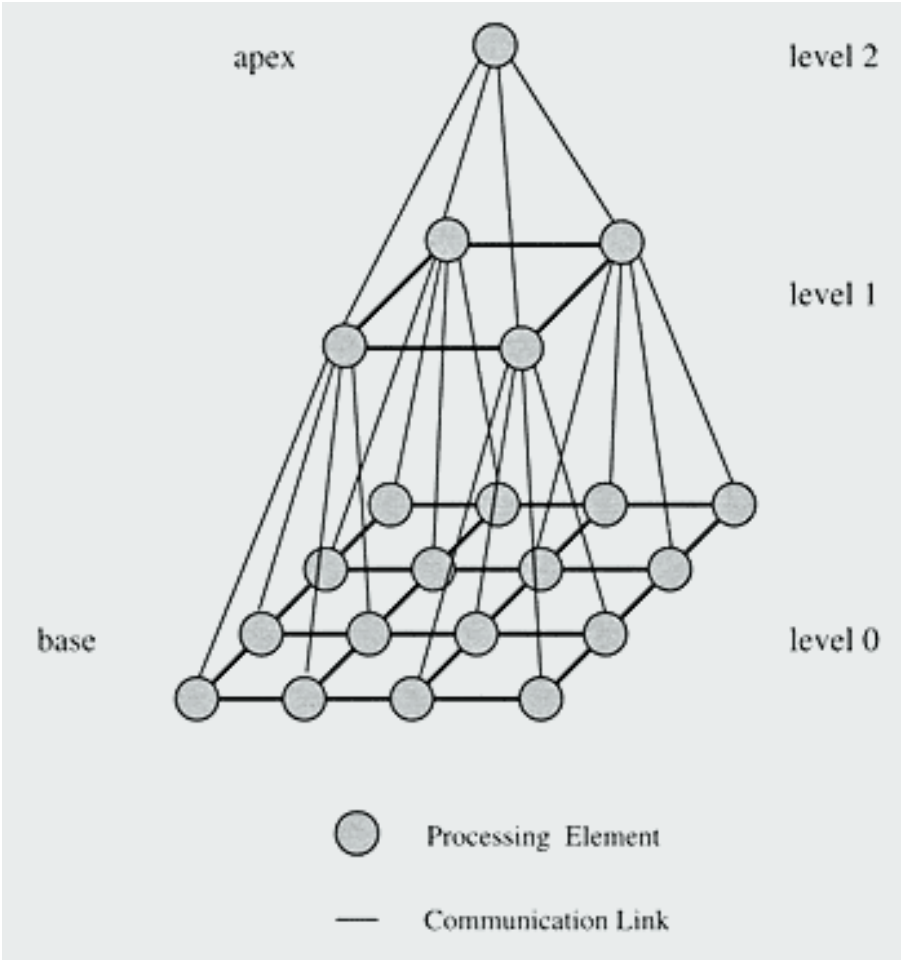



Figure 5.1:
A pyramid computer of size 16.



arbitrary processors. For problems such as counting the number of black pixels in the base, computing a semigroup operation (i.e., an associative binary operation such as minimum, summation, or parity) over a set of values stored in the base, determining certain convexity properties of a digitized figure, or deciding whether or not a digitized picture could have arisen as the digitization of a straight line segment, algorithms will be presented later in this chapter that finish in $\Theta(\log n)$ time.

However, in order to sort data that initially resides one item per base processor, $\Omega(n^{1/2})$ time is required in the worst case. This can be seen by using a wire-counting argument that compares the number of wires that cross the middle of the pyramid with the number of items that potentially must move from one half of the pyramid to the other. In the base of the pyramid there are $n^{1/2}$ wires that cross the middle of the pyramid, in the next level there are $\frac{n^{1/2}}{2}$ such wires, and so on, giving the total number of wires that cross the middle of a pyramid of size $n$ to be $\sum_{i=0}^{\log_4(n)-1} \frac{n^{1/2}}{2^i}$, which is $2n^{1/2} - 2$. Since all $n$ pieces of data that initially reside in the base of the pyramid may need to cross from one side of the base mesh to the other, then $\left\lceil \frac{n}{2n^{1/2}-2} \right\rceil$ time units, or $\Omega(n^{1/2})$ time is required just to get the data across the middle of the pyramid. This bound applies to many other problems, including most problems in computational geometry for which the input is a collection of points arbitrarily distributed one per processor in the base. Since the base mesh alone can sort in $\Theta(n^{1/2})$ time, as described in Section 2.6.1, the pyramid computer is a poor choice for problems involving extensive data movement.

For problems considered in this chapter, poly-logarithmic running times are attainable. However, for many of the problems considered in Chapter 6, neither the $\Omega(\log n)$ nor the $\Omega(n^{1/2})$ bounds are appropriate. In fact, for the problems considered in Chapter 6, the logarithmic bound is still true, but overly optimistic, while the $\Omega(n^{1/2})$ bound does not apply because not as much data needs to be moved. It will be shown in Chapter 6 that by exploiting properties of image data, many geometric problems for multiple figures can be solved in time approximately proportional to $n^{1/4}$.

## 5.4 Fundamental Algorithms

In this section, several fundamental pyramid computer algorithms are presented. These, and other tree-like pyramid algorithms, may be found



in [AhSw84, DWR81, Dyer79, Dyer80, Mill85a, Tani75, Tani76].

### *5.4.1 Initializing Identity Registers*

In this section, a $\Theta(\log n)$ time algorithm is presented to initialize the identity registers of all processors in a pyramid computer of size $n$. Initially, it is assumed that the processors do not know any of the dimensions of the pyramid, including the number of levels in the pyramid, the size of the base mesh, or their mesh level with respect to the pyramid. However, the assumption is made that those processors in the base of the pyramid can detect that they are base processors, by querying for nonexistent children, and that the processor at the apex of the pyramid can detect that it is the apex, by querying for a nonexistent parent.

The algorithm, as given in Figure 5.2, proceeds in two phases. During Phase 1, level information is propagated from the base towards the apex so that every processor can determine its mesh level within the pyramid. During Phase 2, the information flows through the pyramid from the apex to the base in order to allow every processor to determine its row and column coordinates with respect to its mesh level.

After completing both phases of the algorithm, every processor will know its correct *row*, *column*, and *level* coordinates. Recall that every processor can perform a fixed number of arithmetic and Boolean operations, as well as send or receive a fixed number of words from each of its nine neighbors, all in $\Theta(1)$ time. Therefore, step 1 of Phase 1 and steps 1 and 2 of Phase 2, can be performed in $\Theta(1)$ time. Every iteration of the loop in step 3 of Phase 1, and every iteration of the loop in step 3 of Phase 2, can also be performed in $\Theta(1)$ time. Since both phases of the algorithm require a constant amount of work per pyramid level, and since there are $\log_4(n) + 1$ levels in the pyramid, then each of the two phases of the algorithm requires $\Theta(\log n)$ time. Hence, the algorithm terminates after every processor has the required information, which takes $\Theta(\log n)$ time. Note that Phase 1 and Phase 2 may be performed simultaneously since they are independent.

**Proposition 5.1** *The identity registers of all processors of a pyramid computer of size n can be initialized in* $\Theta(\log n)$ *time.* ·

### *5.4.2 Bit Counting Problems*

In this section, it is assumed that a digitized black/white picture is initially stored one pixel per processor in the base of the pyramid. The

**Phase 1**

1. Every base processor sets its *level* register to 0.

2. While the apex has not determined its *level* do:

   (a) Every processor that just initialized its *level* register sends the value of *level* to its parent.

   (b) Every processor receiving 4 identical values from its children, call them *clevel*, sets its *level*←*clevel*+1.

**Phase 2**

1. The apex initializes (*row*, *column*) to (0,0).

2. The apex informs

   (a) its northwest child that its (*row*, *column*) is (0,0),

   (b) its northeast child that its (*row*, *column*) is (0,1),

   (c) its southwest child that its (*row*, *column*) is (1,0), and

   (d) its southeast child that its (*row*, *column*) is (1,1).

3. While the base processors are uninitialized do:

   Every processor that just received its (*row*, *column*) coordinates, informs its children as to their (*row*, *column*) coordinates, as follows.

   (a) northwest child: (2 * *row* - 1, 2 * *column* - 1)

   (b) northeast child: (2 * *row* - 1, 2 * *column*)

   (c) southwest child: (2 * *row*, 2 * *column* - 1)

   (d) southeast child: (2 * *row*, 2 * *column*)

Figure 5.2:
Initializing the identity registers.

interpretation is that the picture represents a single black figure on a white background. The area of the figure, that is, the number of black pixels in the figure, can be determined in $\Theta(\log n)$ time. During the first stage of the algorithm, every processor at level 1 obtains the values of the pixels stored in its four children (these children are base processors) and computes the number of these that are black, storing the result in register *local_count*. So, at the end of stage 1, every processor at level 1 will know the number of black pixels that exist in base of the subpyramid under it. At stage $i$, every processor at level $i$- 1 sends *local_count* to its parent. Every processor $P$ at level $i$ adds the 4 values sent from its children, which gives the total number of black pixels in the base of the subpyramid under $P$, and stores this value in *local_count*. At the conclusion of stage $\log_4 n$, the apex knows the total number of black pixels in the entire base. Since each of the $\log_4 n$ stages requires constant computation time, the algorithm runs in $\Theta(\log n)$ time.

**Proposition 5.2** *Given a digitized black/white picture, stored one pixel per base processor, in* $\Theta(\log n)$ *time all processors of a pyramid computer of size n can know the area of the digitized picture.*

*Proof.* As just described, in $\Theta(\log n)$ time the apex of the pyramid can know the area (number of black pixels) of the digitized picture. Further, any piece of information stored in the apex can be sent to all processors of the pyramid in $\Theta(\log n)$ time (starting with the apex and moving down to the base, at each stage a parent distributes the information obtained during the previous stage to its four children). ·

A number of additional queries for binary input, including majority, equality, and parity, can be answered in $\Theta(\log n)$ time by a straightforward bottom-up counting procedure, as just described.

**Proposition 5.3** *Given a digitized black/white picture, stored one pixel per base processor, in* $\Theta(\log n)$ *time all processors of a pyramid computer of size n can know the answers to the following queries.*

1. *Majority - Are there more black pixels than white pixels in the picture?*

2. *Equality - Are the number of black and white pixels in the picture the same?*

3. *Parity - Are there an odd number of black pixels in the picture?*



*Proof.* From Proposition 5.2, in $\Theta(\log n)$ time the apex can know the number of black (and similarly white) pixels present in the digitized black/white picture that resides in the base of a pyramid computer of size $n$. In (1) additional units of time, the apex can then answer the majority, equality, and parity queries. Finally, the apex can pass the answers down through the pyramid to every processor in $\Theta(\log n)$ time.

Given an arbitrary connected planar graph $G$, with $v$ vertices and e edges, the number of regions, including the unbounded region, is given by the topologically invariant *Euler number* $\Theta(G) = e - v + 2$. Given a single arbitrary digitized black *figure* (i.e., connected component) $F$ on a white background, Minsky and Papert [MiPa69] showed that the Euler number of $F$ can be computed by determining the cardinality of four sets of pixels, and then computing a fixed number of arithmetic operations on these values. Specifically, they showed that for a given figure $F$, $\Theta(F) = C_1 - C_2 - C_3 + C_4$, where $C_1$ is the number of black pixels, $C_2$ is the number of black pixels for which the eastern neighboring pixel is black, $C_3$ is the number of black pixels for which the southern neighboring pixel is black, and $C_4$ is the number of black pixels for which the eastern, southern, and southeastern neighboring pixels are black.

A pyramid algorithm follows.

1. In constant time, every base processor of a pyramid of size $n$ determines which, if any, of the four sets its pixel is a member of.

2. For $i := 1$ to $\log_4 n$ do,

   (a) Every processor at level $i$ obtains the running sum of the cardinality of the four sets of pixels from each of its four children at level $i - 1$.

   (b) Every processor at level $i$ sums the four values associated with each set, compressing the 16 pieces of data just received down to 4 pieces of data (1 piece of data for each of the four sets).

   (c) *Comment:* At this point, every processor at level $i$ knows the values of $C_1$, $C_2$, $C_3$, and $C_4$, with respect to the base processors in its subpyramid.

3. The apex of the pyramid computes $\Theta(F) = C_1 - C_2 - C_3 + C_4$.

4. $\Theta(F)$ is broadcast from the apex to all processors of the pyramid in a straightforward top-down fashion.



During the algorithm, notice that no processor ever needs more than a fixed number of registers, each of size $\Theta(\log n)$. Steps 1 and 3 each require $\Theta(1)$ time. Step 4 requires $\Theta(\log n)$ time. Each of the $\Theta(\log n)$ stages of step 2 require a fixed amount of computation time. Therefore, the Euler number of a single figure stored in the base of a pyramid can be computed and distributed to all processors of the pyramid of size $n$ in $\Theta(\log n)$ time.

**Proposition 5.4** *Given a digitized black/white picture containing a single figure (i.e., connected component), stored one pixel per base processor, in* $\Theta(\log n)$ *time all processors of a pyramid computer of size n can know the Euler number of the figure.* ·

### *5.4.3 Computing Commutative Associative Binary Functions*

Assume that every processor $P_i$ in the base of a pyramid of size $n$ contains a, not necessarily unique, value $v_i$. Suppose that there exists a commutative, associative, binary operation a defined on these values and that $\alpha(v_1, \ldots, v_n)$ is to be determined. (Common examples of a are minimum, maximum, and summation.) Notice that all processors can know the result of applying $\alpha$ to these values in $\Theta(\log n)$ time by combining the results, with respect to $\alpha$, from the base to the apex in $\Theta(\log n)$ time, and then distributing the result from the apex down to all processors in $\Theta(\log n)$ time. For example, the bit-counting algorithms of Section 5.4.2 correspond to the situation where a base processor with a black pixel has a 1, a base processor with a white pixel has a 0, and $\alpha$ is defined to be addition (+).

**Proposition 5.5** *Given that the processors in the base of a pyramid each contain a value, and that a commutative, associative, binary operation a can be computed in* $\Theta(1)$ *time, then in* $\Theta(\log n)$ *time all processors of a pyramid computer of size n can know the result of applying a over all of these base values.* ·

### *5.4.4 Point Queries*

Assume that every processor in the base of the pyramid contains a value (e.g., black or white for a digitized picture, a component label, or an integer value) and that the apex contains the coordinates of some base



processor. Then in $\Theta(\log n)$ time, numerous queries about the value contained in this base processor can be answered. Briefly, this is done by first informing all base processors as to the coordinates of the query point, and then using a bottom-up merging algorithm to obtain the desired result in $\Theta(\log n)$ time.

Certain queries may actually require a fixed number of applications of this top-down, bottom-up procedure. For instance, suppose that every base processor contains a, not necessarily unique, label, and that it is necessary to determine a nearest distinctly labeled processor to the query point (the coordinates of which are known to the apex of the pyramid). An efficient $\Theta(\log n)$ time algorithm to solve this nearest-neighbor query on a pyramid of size $n$ follows.

1. Use a standard $\Theta(\log n)$ time top-down algorithm to inform all base processors as to the coordinates of the query point.

2. Use a standard $\Theta(\log n)$ time bottom-up algorithm to send the label of the query point to the apex.

3. Use a standard $\Theta(\log n)$ time top-down algorithm to propagate this label down to all base processors.

4. Every base processor $P_{i,j}$ creates a distance record (dist, $i$, $j$), where *dist* is the distance to the query point, if the query point has a different label, or else it is set to $\infty$.

5. Now the minimum distance is determined in $\Theta(\log n)$ time using a nearly standard bottom-up computing algorithm, where the minimum is taken over the first field in these distance records, with ties broken arbitrarily. (Notice that while the minimum distance to a distinctly labeled processor is unique, there may be more than one processor at this distance. In the case of multiple nearest processors, this algorithm chooses one arbitrarily.)

Therefore, in $\Theta(\log n)$ time the apex of the pyramid knows a closest distinctly labeled point to the query point.

## 5.5 Image Algorithms

In this section, algorithms that use predominantly the child-parent links of the pyramid are presented to detect geometric properties of sets of base processors or digitized pictures that are stored in the base of the



pyramid. These algorithms solve problems such as enumerating the extreme points of a set of base processors, deciding whether or not a given figure is convex, deciding whether or not two labeled sets of base processors are linearly separable, and deciding whether or not a given figure could have arisen as the digitization of a straight line segment. In Chapter 6, specifically in Sections 6.4 and 6.5, additional pyramid algorithms are presented in conjunction with sophisticated data movement operations to provide efficient solutions to problems involving multiple figures, such as labeling figures, determining a nearest distinctly labeled figure for each figure, and solving a variety of convexity problems.

### *5.5.1 Convexity Properties*

As discussed in Section 1.4.2, for problems involving convexity, the base processor at position $(i, j)$ is identified with the integer lattice point $(i, j)$, and a set of base processors is said to be *convex* if and only if the corresponding set of integer lattice points is convex, i.e., the smallest convex polygon containing them contains no other integer lattice points. This is the proper notion of convexity for integer lattice points, but it does have the annoying property that some disconnected sets of points, such as $\{(1, 1), (3, 4)\}$, are convex.

Given a set $S$ of base processors, the convex hull of $S$, denoted $hull(S)$, is the smallest convex set of (base) processors that includes $S$. A processor $P \in S$ is defined to be an *extreme point* of $S$ if and only if $P \notin \text{hull}(S - \{P\})$. That is, the extreme points of $S$ are the corners of the smallest convex polygon containing $S$. It is said that the *extreme points of S have been marked* if every processor $P$ in the base of the pyramid has a Boolean variable that is true if and only if $P$ is an extreme point of $S$. It is said that the *extreme points of S have been enumerated* if for every (base) processor $P_i$ containing a point $p \in S$, the following hold. (See Figure 5.3.)

1. $P_i$ has a Boolean variable 'extreme', and extreme is true if and only if $p$ is an extreme point of $S$.

2. $P_i$ stores the total number of extreme points of $hull(S)$.

3. If $p$ is an extreme point of $S$, then $P_i$ stores the position of $p$ in the counterclockwise ordering of extreme points. (The rightmost extreme point is assigned the number 1. If there are two rightmost extreme points, then the lower one is assigned the number 1.)

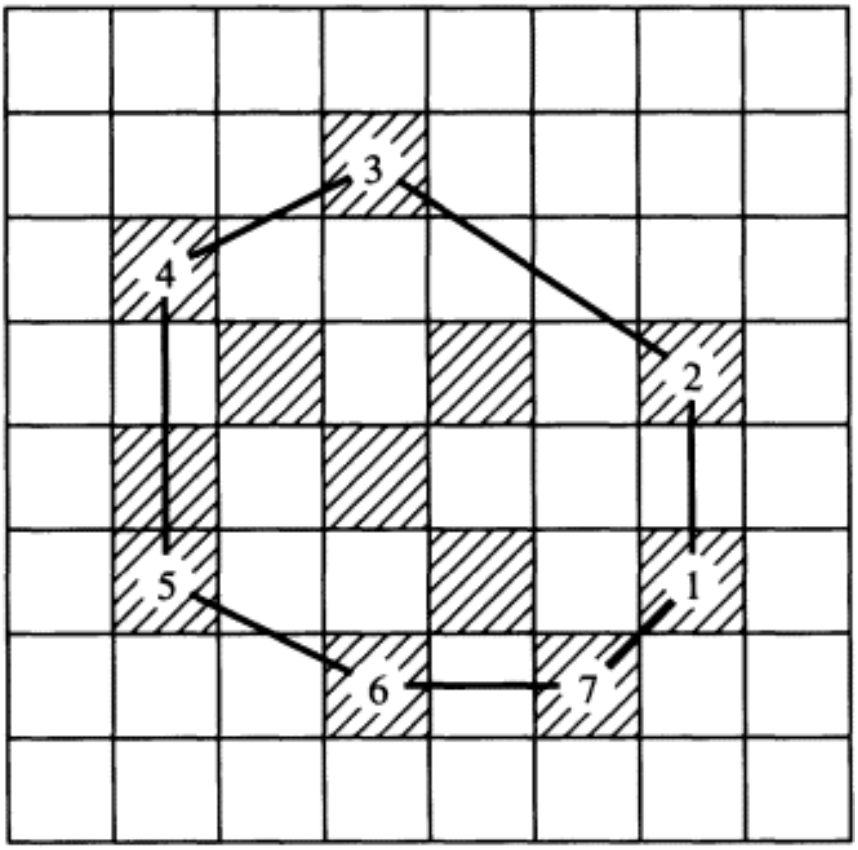



Figure 5.3:
Enumerated extreme points of $S$.

4. If $p$ is an extreme point of $S$, then $P_i$ stores the Cartesian coordinates of the extreme points that precede and succeed $p$, as well as the ID of the processors that contain them.

In this section, the input at the base of the pyramid consists of either a single digitized black figure on a white background (i.e., there is exactly one connected component in the base) or a set of consistently labeled processors (i.e., all base processors that contain a label, contain the same label). Before solving some general convexity problems for single figures, a useful lemma is presented. The algorithm associated with this lemma shows how to enumerate a marked set of extreme points in $\Theta(\log n)$ time. The algorithm works by using standard bottom-up and top-down tree-like operations, as described previously in Section 5.4. Initially, the set of (at most) 8 perimeter partition points is determined, as shown in Figure 5.4. After every processor determines the number of



extreme points in the base of its subpyramid in each of the eight triangular regions, the apex will recursively propagate intervals of numbers to children corresponding to the numbers that will be used in enumerating the extreme points of that child's base processors.

**Lemma 5.6** *In a pyramid computer of size $n$, suppose the extreme points of a set $S$ of base processors have been marked. Then the extreme points of $S$ can be enumerated in $\Theta(\log n)$ time.*

*Proof.* The algorithm requires that the processors determine certain basic information, as follows.

1. By using a bottom-up report, followed by a top-down broadcast operation, in $2\log_4 n$ steps all processors in the pyramid computer can know

(a) the total number of extreme points, and

(b) the coordinates of the rightmost-bottommost, rightmost-topmost, topmost-rightmost, topmost-leftmost, leftmost-topmost, leftmost-bottommost, bottommost-leftmost, and bottommost-rightmost extreme points, as shown in Figure 5.4.

2. In $2\log_4 n$ steps, every processor in the pyramid can know the total numbers of extreme points of $S$ in the base of its subpyramid that are in each of these 8 regions. Notice that the boundaries of these (possibly degenerate) triangular regions may be generated in unit-time by every processor of the pyramid once they are informed as to the locations of these 8 points.

Once this information has been determined, the extreme points of $S$ can be labeled in $\Theta(\log n)$ steps by having the apex recursively distribute ranges of the numbers to its children for each of the eight regions. Distributing the proper numbers to the children is straightforward since the extreme points represent a convex polygon. Notice that within each region, this is a prefix computation.

It only remains to show that each every processor containing an extreme point of $S$ can determine the location of the preceding and succeeding extreme points of $S$ in $\Theta(\log n)$ steps. During the numbering process, as every processor passes ranges of numbers to its children, it also determines if any of its children are responsible for extreme points that have a preceding or succeeding extreme point in another one of its



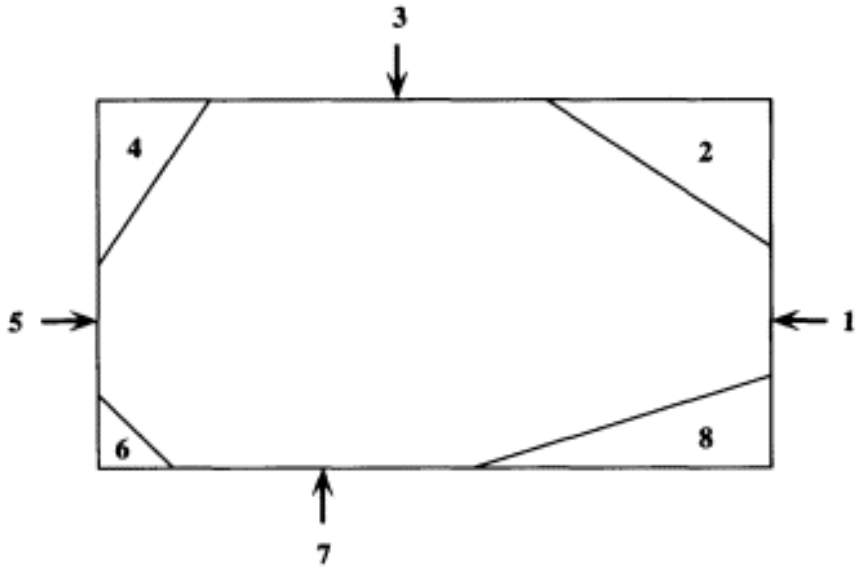



Figure 5.4:
The 8 perimeter points consist of the rightmost-bottommost, rightmost-topmost, topmost-rightmost, topmost-leftmost, leftmost-topmost, leftmost-bottommost, bottommost-leftmost, and bottommost-rightmost extreme points. These points partition the set of extreme points into 8 'triangular regions', which are labeled 1, 2,. . . , 8. Four of these regions (i.e., 1, 3, 5, and 7) are necessarily degenerate, while the remaining four (i.e., 2, 4, 6, and 8) might contain additional extreme points.

children. For each such case, the processor creates a *neighbor record*, which consists of the numbers of the extreme points involved, as well as the identity of the processor creating the record. When the numbering phase of the algorithm terminates, these neighbor records are sent down to the base in lockstep fashion. When a base processor receives a neighbor record, it is examined to determine if either of the numbers in the record correspond to its extreme point number. If there is a match, then the location of the extreme point is appended to the record, and the record is sent back up to the processor that generated it, while otherwise the record is discarded. Finally, the neighboring information is sent down to the base in lockstep fashion so that every base processor in $S$ knows its number, as well as the location of its predecessor and successor.

The algorithm requires a fixed number of $\Theta(\log n)$ time top-down and bottom-up tree-like operations. Therefore, the running time of the algorithm is $\Theta(\log n)$.

The next problem considered is that of enumerating the extreme points of a convex set of base processors. This is important in many processing applications that require a compact description of a single convex figure for storage or transmission purposes.

Before giving the result for enumerating the extreme points of a single convex figure, a simple technical lemma is presented. The lemma is concerned with the fact that it is possible to take a digitized convex figure, divide it into two parts by a straight line parallel to one of the grid axes, and have points which are extreme points of the parts but not of the entire figure. An important consequence of the following lemma is that there are only $O(\log n)$ such points.

**Lemma 5.7** *Given a convex figure $F$ on a grid, suppose the grid is divided vertically in half and the extreme points of the restriction of $F$ to the right half are determined. Suppose $p$ and $q$ are extreme points of the upper envelope of the righthand portion. Further, suppose $p$ and $q$ are not extreme points of $F$, and that $q$ is further from the dividing line than is $p$. Then $q$ is more than twice as far from the dividing line as $p$ is.*

*Proof.* Consider a convex figure $F$ partitioned into two pieces, $F_l$ and $F_r$, by a vertical line, where $F_1$ lies to the left of $F_r$. Let $e$ and $w$ be an easternmost and westernmost point, respectively, of $F_r$. Without loss of generality, let $p = (p_x, p_y)$ and $q = (q_x, q_y)$ be extreme points of $F_r$ that are on or above the line $\overline{ew}$. Further, assume $p$ and $q$ are not extreme



points of $F$. Then the line segment $L$ from $q$, passing through $p$, and continuing on to the dividing line, lies in the convex hull of $F$ when viewed as a figure in the real plane (rather than just on the grid). If $q$ were less than twice $p$'s distance to the dividing line, then the grid point $r = (2p_x - q_x, 2p_y - q_y)$ would lie on $L$ and be on the same side of the dividing line as $p$ and $q$. This means that $r$ would be in $F$ and, in fact, in $F_r$. However, since $p$ is halfway between $q$ and $r$, this contradicts the assumption that $p$ is an extreme point of $F_r$. ·

The result that follows shows that the extreme points of a *single convex set S* of base processors can be enumerated efficiently on a pyramid computer. The algorithm first marks the extreme points and then uses the algorithm associated with Lemma 5.6 to enumerate them. Notice that given a pair of adjacent subsquares in the base, there are at most two places along the border between the subsquares where elements of the convex set $S$ need to be considered when combining these subsquares. Unfortunately, if a straightforward bottom-up divide-and-conquer algorithm is used, Lemma 5.7 shows that in the worst case there are a logarithmic number of extreme points near the border of adjacent subsquares that need to be eliminated. While it may be possible to work out the details of such an algorithm, an interesting recursive bottom-up divide-and-conquer algorithm will be presented instead that has no such complications. Unlike previous algorithms presented in this chapter that rely predominantly on the child-parent links of the pyramid, the marking algorithm presented below takes advantage of the mesh connections in levels above the base in order to allow the apices over neighboring subsquares in the base to exchange information.

**Theorem 5.8** *In a pyramid computer of size $n$, suppose the base processors with a given label form a convex set $S$. Then in $\Theta(\log n)$ time the extreme points of $S$ can be enumerated.*

*Proof.* The algorithm proceeds in two phases. The first phase of the algorithm will mark the extreme points of $S$, and the second phase of the algorithm will enumerate them by applying the algorithm associated with Lemma 5.6. Therefore, only the details of the first phase need to be presented.

The algorithm for marking extreme points works in a bottom-up fashion, where at step $k$, $0 \leq k \leq \log_4 n$, decisions regarding extreme points are made by processors at level $k$. Consider the processors at level $k$ in the pyramid. These are the apices of disjoint subpyramids with bases of size $2^k \times 2^k$. Call the base of each of these disjoint subpyramids a *subsquare*. At the end of step $k$, suppose that



1. in each $2^k \times 2^k$ subsquare, those points that are not extreme points of the restriction of the figure to their subsquare have been marked as not being extreme points, while those that are extreme points in their subsquare remain as candidate extreme points for the entire figure, and

2. for every way of forming a $2^{k+1} \times 2^{k+1}$ square from four $2^k \times 2^k$ subsquares, each point that is not an extreme point in the restriction of the figure to a $2^{k+1} \times 2^{k+1}$ square has also been marked as not being an extreme point. (Notice that these larger squares overlap, and some correspond to bases of subpyramids of height $k + 1$, while others do not.)

Now, consider step $k + 1$, where for processors at level $k + 1$ the base of each of the corresponding subpyramids is called a *block*. Since each block is a $2^{k+1} \times 2^{k+1}$ square formed from four $2^k \times 2^k$ subsquares, it is known that at the beginning of step $k + 1$, those points that are not extreme points in the restriction of the figure to their block, have already been marked as not extreme points. The purpose of step $k + 1$ (i.e., the recursive step) is to identify those points that were candidate extreme points at the end of step $k$, but that are not extreme points in some square of 4 blocks. This must be done for all possible squares of 4 blocks, not just those corresponding to the base of a subpyramid of height $k + 2$.

Each square of 4 blocks can be formed by merging 2 pairs of blocks together (simultaneously), and then merging these rectangles together. Both merge steps are similar, so only the first will be described. Since all $2^{k+1} \times 2^{k+1}$ squares formed from four $2^k \times 2^k$ subsquares have been considered during step $k$, for a (current) candidate extreme point $p$ to be marked as not being a candidate during step $k + 1$, there must be a triangle containing $p$ with a pair of vertices, say $q$ and $r$, at least $2^k$ apart from each other, with one of them, say $q$, being at least $2^{k-1}$ from $p$. Further, if $q$ causes more than $p$ to be eliminated, then the second point it eliminates must be at least $2^k$ from $q$, the third must be at least $2^{k+1}$ from $q$, and the fourth point must be at least $2^k+2$ from $q$. Thus, by knowing for each subsquare only some small fixed number of candidate extreme points along the top, bottom, left, and right of a square, a processor can determine all false extreme points in the merger of the blocks.

The apex of each block maintains the necessary information about its block. By exchanging information with its neighbors at level $k + 1$, in constant time, simultaneously for every apex, an apex can determine for



each possible square of 4 blocks, which of its candidate extreme points should be eliminated from further consideration. To finish step $k + 1$, every processor at level $k + 1$ initiates a top-down broadcast message of the information to its block, and supplies its parent with the information necessary to start step $k + 2$.

The time between the start of step $k + 1$ and the start of step $k + 2$ is $\Theta(1)$. The time it takes to finish the final top-down broadcast after the last step is complete is $\Theta(\log n)$. Therefore, the total running time is $\Theta(\log n)$. ·

Given a set of enumerated extreme points, the next problem is concerned with marking a convex figure associated with the extreme points. If the extreme points correspond to a figure that was not convex, then the figure that is generated will be an approximation of the original figure. However, if the original figure was 'blob'-like, then this operation can be viewed approximately as the inverse operation to that of generating the extreme points of the figure. The algorithm is straightforward. It consists of first passing extreme point information up the pyramid, and then passing information down to base processors so that every base processor can decide whether or not it is in the convex hull represented by the set of enumerated extreme points.

**Theorem 5.9** *In the base of a pyramid computer of size $n$, suppose a set of extreme points has been enumerated. Then in $\Theta(\log n)$ time, the base processors corresponding to a convex figure that would yield such a set of extreme points, can be marked.*

*Proof*. Since the extreme points have been enumerated, every base processor containing an extreme point knows the location of the extreme points preceding and succeeding it, with respect to the counterclockwise ordering of extreme points. Assume that there are $p$ extreme points in the figure, where the base processor containing the $i^{th}$ extreme point is denoted $P_i$, $1 \leq i \leq p$. Each base processor $P_i$ assumes responsibility for the hull edge, call it $e_i$, between its (extreme) point and the extreme point that follows it in the counterclockwise ordering. Every processor $P_i$ can now determine in $\Theta(1)$ time, the processor in the pyramid at maximum level (i.e., closest to the apex), denoted $P_{i\,(m)}$, that is an ancestor of $P_i$ such that $e_i$ crosses the boundary between the subpyramids rooted at the children of $P_{i\,(m)}$. Every base processor $P_i$ now passes the hull edge $e_i$ that it is responsible for, as well as a flag indicating which side of $e_i$ is on the inside of the convex hull, up to $P_{i\,(m)}$. Notice that no processor in the pyramid will be responsible for more than 4 such



edges. After $\Theta(\log n)$ time, all processors in the pyramid will know the (at most) 4 edges in the base that cross the boundaries of the subpyramids of its children. This information is then passed down the pyramid in lockstep fashion from all $P_{i(m)}$ to their descendants. As each base processor receives such information, it decides in $\Theta(1)$ time whether or not it is in the convex hull. Since the algorithm consists of a straightforward bottom-up phase followed by a straightforward top-down phase, the running time is as claimed.

The next result provides an optimal solution to the problem of deciding whether or not a marked set of base processors is convex. The algorithm is straightforward, combining the results just presented in Theorem 5.8 and Theorem 5.9. First, use the algorithm associated with Theorem 5.8 to mark the "extreme points" of the set. Every processor containing an extreme point determines whether or not it can decide that the figure is not convex by examining the preceding pair of "extreme points" and the succeeding pair of "extreme points." Combining these results, it can be decided whether or not the "extreme points" are convex. If the set of "extreme points" are not convex, then the algorithm halts and it is known that the original marked set of processors is not convex. Otherwise, use the algorithm associated with Theorem 5.8 to mark the convex hull represented by the extreme points, and compare those marked processors with the original marked set of processors. This gives the following result.

**Corollary 5.10** *In a pyramid computer of size $n$, in $\Theta(\log n)$ time the set of base processors with a given label can decide whether or not they are convex.*

A set $A$ of base processors is *linearly separable* from a set $B$ of base processors if and only if there is a straight line in the plane such that all elements of $A$ lie on one side of the line, and all elements of $B$ lie on the other side. A well-known observation is that two such sets are linearly separable if and only if their convex hulls are disjoint. Given the enumerated extreme points of two sets of (not necessarily distinct) base processors, in $\Theta(\log n)$ time it can be determined whether or not these two sets are linearly separable, as follows. Mark the convex hull of $A$ such that a base processor has the value a if it is in the convex hull of $A$, and the convex hull $B$ such that a base processor has the value 3 if it is in the convex hull of $B$. This takes $\Theta(\log n)$ time by applying the algorithm associated with Theorem 5.9 once for $A$ and a second time for $B$. All base processors send to the apex a Boolean flag that is set

to 'true' if the processor is labeled $\alpha$ *and* $\beta$, and that is set to 'false' otherwise. As each processor in the pyramid receives the four Boolean values from its children, they are logically 'or'ed together and passed up. In $\Theta(\log n)$ the apex knows the answer to the query, which it propagates to all processors in the pyramid in $\Theta(\log n)$ time. Hence the algorithm is complete in $\Theta(\log n)$ time.

**Corollary 5.11** *In a pyramid computer of size n, suppose the extreme points corresponding to a set A of base processors have been enumerated, as have the extreme points of a set B of base processors. Then in* $\Theta(\log n)$ *time it can be decided whether or not A is linearly separable from B.* ·

The next problem considered is that of enumerating the extreme points of an arbitrary set of base processors. This extreme point generation algorithm degrades by a factor of $\Theta(\log n/\log\log n)$ over the convexity query algorithm in Corollary 5.10. This is counterintuitive in that the solution to the convexity query problem can be obtained faster than generating the extreme points of a given set of base processors. It should be noted that a $\Theta(\log n)$ time extreme point generation algorithm is an open problem.

The extreme point generation algorithm that is presented in the proof of Theorem 5.12 follows a top-down divide-and-conquer solution that exploits the following fact about extreme points. A point is an extreme point if and only if it is the first point of the figure contacted as some line is moved towards the figure from infinity. By way of an example, suppose that for a given digital figure embedded in an $n^{1/2} \times n^{1/2}$ grid, there exist unique topmost, bottommost, leftmost, and rightmost extreme points. Then the topmost point may be detected by finding the first point contacted as a line of slope 0 approaches the figure from the top, the bottommost point may be detected as a line of slope 0 approaches from the bottom, the leftmost point may be detected as a line of slope $\infty$ approaches from the left, and the rightmost point may be detected as a line of slope $\infty$ approaches from the right. In addition, for any extreme point $p$ of the figure that is between the topmost point and the leftmost point, there must be a slope in the range $(n^{-1/2}, n^{1/2})$ such that $p$ is the first point of the figure contacted as a line with this slope comes towards the figure from the upper-left direction. If the line with slope $\frac{n^{-1/2}+n^{1/2}}{2}$ is used to detect an extreme point between the topmost and leftmost extreme points, then



1. if the first point contacted is the topmost extreme point, then there are no extreme points of the figure between the topmost and leftmost extreme points that will be detected by slopes in the range $\left(n^{-1/2}, \frac{n^{-1/2}+n^{1/2}}{2}\right]$,while

2. if the first point contacted is the leftmost extreme point, then there are no extreme points of the figure between the topmost and leftmost extreme points that will be detected by slopes in the range $\left[\frac{n^{-1/2}+n^{1/2}}{2}, n^{1/2}\right)$, while

3. if a first point contacted was not the topmost or leftmost extreme point, then this first point (or, in the case of a multiple detection, the outermost points contacted) is an extreme point.

These situations define a recursive search procedure that is used to detect extreme points. Notice that if a single new extreme point is found in an interval, then this new extreme point is used to create two subintervals, both of which are searched for additional extreme points. These observations form the basis of the algorithm that follows.

**Theorem 5.12** *In a pyramid computer of size n, the extreme points of a labeled set of base processors can be enumerated in* $\Theta(\log^2 n/\log\log n)$ *time.*

*Proof.* The algorithm uses a top-down divide-and-conquer solution strategy. First, an algorithm requiring $O(\log^2 n)$ time will be given, after which it is shown how to modify this algorithm to reduce the running time to $\Theta(\log^2 n/\log\log n)$. Let $S$ be the set of base processors with a given label. Observe that as a line $I$ of fixed slope is brought towards $S$, then the first element of $S$ to come in contact with $I$ must be an extreme point of $S$. (If several elements of $S$ come in contact with $I$ simultaneously, then only the two extreme points of this 1-dimensional set of points are extreme points of $S$.) Notice that a processor $P$ that is an extreme point of $S$, with $P_1$ and $P_2$ the preceding and succeeding extreme points, respectively, will be detected as an extreme point of $S$ by a line $I$ that has a slope between $slope(\overline{P_1P})$ and $slope(\overline{PP_2})$, as it moves towards $S$ from the concave side of the angle formed by $P_1PP_2$, as shown in Figure 5.5.

The set $S$ of base processors is embedded in an $n^{1/2} \times n^{1/2}$ grid. Therefore, except for vertical lines, all lines through two processors have slopes between $-n^{1/2}$ and $n^{1/2}$. Further, only $\Theta(n)$ different slopes can actually occur. However, since it is simpler to consider slopes that are



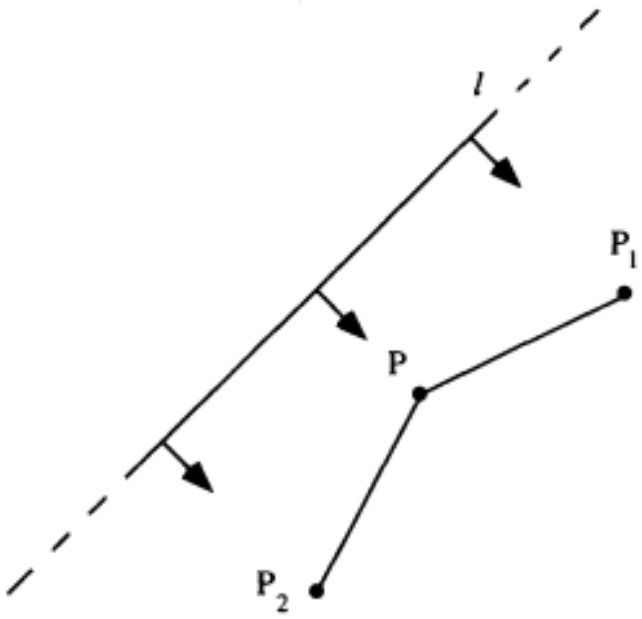



Figure 5.5:
Detecting $P$ as an extreme point.

multiples of $1/n$, and since this will not cause a significant time delay, the algorithm will, in fact, examine $O(n^{3/2})$ slopes.

In $\Theta(\log n)$ time, a straightforward bottom-up algorithm is used so that the apex knows the coordinates of the (not necessarily distinct) rightmost-bottommost, rightmost-topmost, topmost-rightmost, topmost-leftmost, leftmost-topmost, leftmost-bottommost, bottommost-leftmost, and bottommost-rightmost members of $S$. These are all extreme points of $S$, and they divide the perimeter of $S$ into 8 (or fewer) intervals, as shown in Figure 5.4 on page 226. Four of these intervals, e.g., between the topmost-rightmost and the topmost-leftmost points, contain no more extreme points, while the other four intervals, e.g., between the topmost-rightmost and the rightmost-topmost points, might contain more extreme points. For each of the four intervals that might contain more extreme points, there is a corresponding interval of line slopes that may be used to locate the extreme points in the interval. For example, in the interval between the rightmost-topmost and the topmost-rightmost extreme points, the slopes are in the range of $-n^{1/2}$ to $-n^{-1/2}$. Let an *interval* refer to a pair of endpoint coordinates, along with their associated interval of slopes. Notice that when a slope $m$ is being used, if each base processor computes the inner product of its $(x, y)$ position



with $(1/m, 1)$, then the base processor with the greatest inner product is the one that would be reached first. (If the line approaches from the opposite side then the base processor with the least inner product is the one reached first.)

Initially, the apex of the pyramid is responsible for the four intervals that may contain additional extreme points. (Referring to Figure 5.4, these are intervals 2, 4, 6, and 8.) The algorithm proceeds in stages, where a processor is responsible for at most 8 intervals during any stage. At the beginning of each stage, if both endpoints of an interval that a processor is responsible for lie in the base of the subpyramid of one of its children, then responsibility for that interval is passed on to that child (which may in turn pass it further down). Next, for each interval that a processor is responsible for, the processor creates a record corresponding to the interval's endpoints and the middle slope. In a top-down fashion, starting with the apex, copies of these records are then sent from every processor to each of its four children. Every processor receiving such a record ignores it if none of the base processors in its subpyramid could be an extreme point as discovered by that slope, while otherwise it passes the record down to its children, along with any such records it may generate. Notice that no processor passes more than 8 such records to any of its children.

When these records reach the base, each element of $S$ determines its inner product with the indicated slope and appends this to the record, along with the processor's coordinates, and passes this record back to its parent. This information is passed up through the pyramid, where when a parent receives multiple copies of an interval, it passes along only the one with the largest inner product. (If there are ties, then the two outermost extreme points among the ties are passed up.) When this information returns to the processor that generated the request, this generating processor will decide on the appropriate course of action. For example, if two new extreme points, say $N_1$ and $N_2$, were discovered between extreme points $P_1$ and $P_2$ , as in Figure 5.6, then the original $P_1P_2$ interval is divided into 3 new intervals, namely, $P_1N_1$ and $N_2P_2$, both of which have no more than half as many slopes as the original $P_1P_2$ interval, and $N_1N_2$ which requires no further work. Other scenerios are treated similarly. Finally, each time an extreme point is found, it is marked.

Each stage of the algorithm takes $\Theta(\log n)$ time. Since there are $O(n^{3/2})$ slopes considered, and since each stage subdivides an interval's slopes by at least half, then there are at most $\Theta(\log n^{3/2}) = \Theta(\log n)$ steps. When finished, all extreme points have been marked, and in

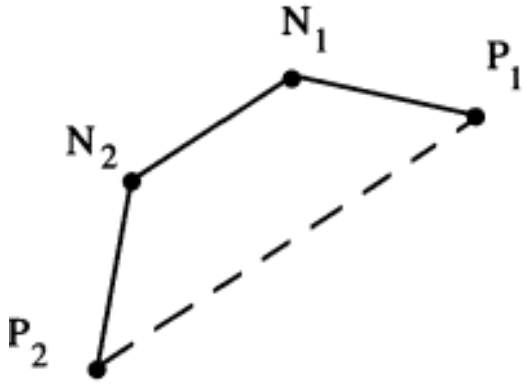



Figure 5.6:
Discovering 2 new extreme points in an interval.

an additional $\Theta(\log n)$ time, the extreme points can be enumerated by applying the algorithm of Lemma 5.6.

The algorithm as described requires $\Theta(\log^2 n)$ time. To reduce the time of the algorithm to $\Theta(\log^2 n/\log \log n)$, have each processor that is responsible for an interval divide that interval's slopes into $\log_2 n$ pieces, instead of 2 pieces. These records are sent down in serial fashion (i.e., pipelined), where no processor passes more than $8\log_2 n$ records to its children. Each stage still takes $\Theta(\log n)$ time, but because the intervals are being broken up faster, only $\Theta(\log n/\log \log n)$ stages are needed. Therefore, the algorithm finishes in the time indicated. ·

Given a ser of base processors, a variety of properties of the set can be determined once the extreme points have been enumerated. Algorithms for determining properties of a given set of processors, such as a smallest enclosing box, the smallest enclosing circle, and the diameter, are presented in Section 6.5. (Additional references for efficient pyramid computer algorithms that use extreme points to generate geometric properties of images include [MiSt84c, MiSt84d, MiSt85c, MiSt91].) The reason that such algorithms are not presented in this section is that they rely on advanced data movement operations that form the foundation of Chapter 6.

### *5.5.2 Digitized Straight Line Segments*

A major effort in digital image processing and pattern recognition has been on the fundamental problem of deciding whether or not a digitized



figure could have arisen as the digitization of a straight line segment (c.f., [Gaaf77, Kim82a, KiRo82b, Kim83, Rose74, Rose79, RoKi82]). In this section, a result about digital arcs [KiRo82b] is combined with Corollary 5.10 in order to prove that in optimal $\Theta(\log n)$ time, a pyramid computer can determine whether or not a digitized black/white figure could have arisen as the digitization of a straight line segment.

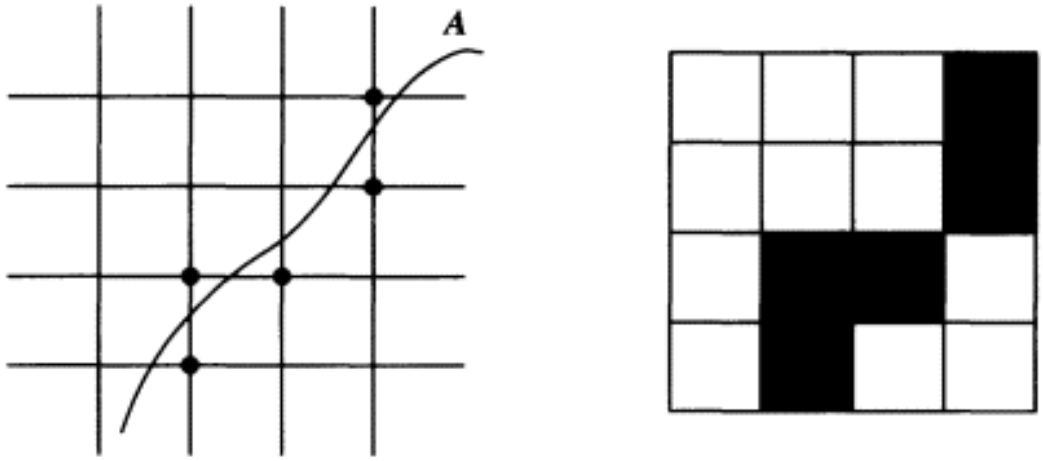
Digitization can make the detection of even basic properties of a figure nontrivial to determine. The digitization scheme that is used in this section is the standard *grid-intersection scheme* [Rose74] for digitizing arcs. (Section 3.5 mentions several digitization schemes, but for an in-depth discussion of digitization schemes, the reader is referred to [DoSm84, Gaaf77, Rose79, RoKi82, Kim81, Kim82a, Kim83, KiRo82b, KiSk82c], and the references contained therein.) Given a coordinate grid superimposed on an arc $A$, then as $A$ is traversed, a succession of grid lines will be crossed. Whenever $A$ crosses a grid line, the processor associated with the integer lattice point nearest to the crossing line becomes a part of $A$'s digitization. In the case where $A$ crosses a grid line halfway between two lattice points, the tie is resolved by choosing the processor associated with the lattice point that lies to the right of $A$ (in the sense that $A$ is being traversed) to be a member of the digitization of $A$. See Figure 5.7.

Define processors $P_{i\pm1,\,j}$, $P_{i,\,j\pm1}$, and $P_{i\pm1,\,j\pm1}$ to be the *8-neighbors* of processor $P_{i,\,j}$, assuming they exist. Given a set $S$ of processors, with processors $P_i, P_j \in S$, then $P_i$ and $P_j$ are said to be *8-connected* if and only if there exists a connected path of 8-neighbors in $S$ between $P_i$ and $P_j$ . A set $S$ of processors is an *8-connected set* if and only if for all processors $P_i, P_j \in S$, $P_i$ and $P_j$ are 8-connected.

An 8-connected set $D$ of processors is a *digital arc* if all but two of the processors in $D$ have exactly two 8-neighbors in $D$, and the exceptional two, called the *endpoints*, each have exactly one 8-neighbor in $D$ [KiRo82b]. Given two lattice points $p$ and $q$, corresponding to two processors in $D$, the line segment $\overline{pq}$ is defined to lie *near* $D$ if for any point $(x, y)$ of $\overline{pq}$ , $(x, y) \in R^2$, there exists a lattice point $(a, b)$ corresponding to a processor $P_{a,\,b} \in D$ such that $\max\{|a - x|, |b - y|\} < 1$. Finally, $D$ is said to have the *chord property* if for every $p, q \in D$, the line segment $\overline{pq}$ lies near $D$ [Rose74].

**Lemma 5.13** [Rose74] *A digital arc has the chord property if and only if it is the digitization of a straight line segment.* ·

Page 238




Coordinate grid superimposed on arc A  Processor digitization of A

Figure 5.7:
Grid-intersection scheme of digitization.

**Lemma 5.14** [KiRo82b] *A set S of base processors has the chord property if and only if S is convex.* ·

From Lemmas 5.13 and 5.14, notice that $D$ could have arisen as the digitization of a straight line segment if and only if it is a convex digital arc. From [KiRo82b], this implies that a convex set $D$ of two or more processors is the digitization of a straight line segment if and only if

1. all but two of the processors of $D$ have exactly two 8-neighbors in $D$, and the exceptional two have exactly one 8-neighbor in $D$, and

2. $D$ is 8-connected.

Further, it can be shown that if $D$ is convex and satisfies property 1, then it satisfies property 2 as well. (This is false for nonconvex sets, as can be seen by considering a disconnected set consisting of digitizations of a circle and a line.) Thus, a convex set $D$ of two or more processors is the digitization of a straight line if and only if it satisfies property 1.

This characterization yields an efficient algorithm to determine whether or not a set $D$ of lattice points could have arisen as the digitization of a straight line segment. (It is assumed that $D$ corresponds



to a set of labeled processors.) From Corollary 5.10, it can be decided in $\Theta(\log n)$ time, whether or not $D$ is convex. If $D$ is not convex, then the algorithm halts and it is known that $D$ could not have arisen as the digitization of a straight line segment, while otherwise the algorithm continues in an effort to determine whether or not $D$ is a digital arc (Property 1). To determine if Property 1 holds, each base processor that is a member of $D$ determines, in $\Theta(1)$ time, the number of its 8-neighbors that are members of $D$. By passing these results up to the apex and combining them at each level, in $\Theta(\log n)$ time the apex will know whether or not Property 1 holds, and hence knows whether $D$ could have arisen as the digitization of a straight line segment. This gives the following.

**Theorem 5.15** *Given a digitized black/white picture stored in a natural fashion one pixel per processor in the base of a pyramid computer of size $n$, in $\Theta(\log n)$ time it can be decided whether or not the set of black pixels could have arisen as the digitization of a straight line segment.* ·

### 5.6 Further Remarks

In this chapter, algorithms that use predominantly the child-parent links of a pyramid have been presented. These algorithms include solutions to a variety of fundamental problems, such as initializing the identity registers of all processors, solving bit counting problems, computing commutative associative binary functions, and answering point queries. All these algorithms have optimal $\Theta(\log n)$ running times.

Several optimal $\Theta(\log n)$ time algorithms were also presented to solve problems involving convexity properties of a single figure or set of base processors. These problems include determining whether or not an arbitrary set of base processors is convex and enumerating the extreme points of a convex set of base processors. For the problem of enumerating the extreme points of an arbitrary set of base processors, the running time of the algorithm presented in this chapter is $\Theta(\log^2 n/\log\log n)$, and the optimality of this algorithm was left as an open problem. Assuming that the extreme points of two figures have been enumerated, a $\Theta(\log n)$ time algorithm was presented to determine whether or not the two figures are linearly separable. Finally, a $\Theta(\log n)$ time algorithm was presented to decide whether or not a digitized picture could have arisen as the digitization of a straight line segment.

Additional algorithms that exploit the pyramid's child-parent communication links abound. Examples include the component labeling algorithm in [Dyer82, Tani82a, Tuck86], the feature extraction algorithm in [Reev80], the median filtering algorithm in [Tani82b], the selection algorithm in [Stou83c], and the polygon construction algorithm in [Sako81].

In the next chapter, algorithms and data movement operations will be presented to solve fundamental problems in image processing and graph theory, where the input is either an adjacency/weight matrix, a set of unordered edges distributed arbitrarily throughout the base of the pyramid, or a digitized picture representing multiple figures. These algorithms will require the use of intricate data movement operations that exploit the pyramid's mixture of child-parent and mesh-connected links.



# 6
# Hybrid Pyramid Algorithms

## 6.1 Introduction

Although the geometry of the pyramid makes it a natural architecture for image processing, there is no reason to limit pyramid computers to low-level image processing involving tree-like operations. The pyramid can be adapted to many other problems and should be considered as an alternative to the mesh computer. In this chapter, a variety of data movement operations are presented for the pyramid, some of which are customized for particular forms of input. These operations are incorporated into efficient pyramid computer algorithms to solve fundamental problems in graph theory, image processing, and digital geometry.

Much of the literature on pyramids consists of two classes of algorithms. The first concentrates on the tree structure, using predominantly child-parent links, as discussed in Chapter 5. These algorithms work efficiently only when the amount of data can be drastically reduced, for otherwise too much data must pass through the apex, creating a bottleneck. The second class of pyramid algorithms concentrates on the mesh, essentially ignoring everything above the base. Efficient mesh algorithms for a variety of problems and input formats were presented in Chapters 2, 3, and 4.

In this chapter, a third class of pyramid computer algorithms is considered. These algorithms utilize both the mesh and tree connections. The basic approach is as follows. Reduce $O(n)$ pieces of initial data, stored one piece per base processor, down to $O(n^{1/2})$ pieces of data. Move this data to a region of the pyramid where interprocessor communication is as fast as possible. Obtain the solution to the subproblem in this region, and move the results to their final locations. The region of the pyramid that the $O(n^{1/2})$ pieces of data will be moved to is the middle level of the pyramid, which is a mesh of size $\Theta(n^{1/2})$. The movement to and from the middle level will often be the most time-consuming part of the algorithm. Therefore, the focus of this chapter is on providing efficient techniques for reducing data and on providing a collection of efficient fundamental data movement operations.

Fundamental data movement operations are presented for several algorithmic strategies, such as divide-and-conquer, and for various formats of input data. These operations are used to give efficient solutions to



a variety of problems that involve a graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of edges. The graph can be expressed as a collection of unordered edges, an adjacency/weight matrix, or as a digitized black/white picture. Algorithms based on these data movement operations are presented to solve problems including component labeling, minimal spanning forest, nearest neighbor, transitive closure, bipartite graph, cyclic index, bridge edges, articulation points, biconnectivity, and various geometric problems involving convexity properties. Many of these problems were discussed in Section 1.4.1. The reader is referred to Chapter 1 for definitions of input formats and descriptions of the problems.

The algorithms presented in this chapter have running times that are about the square root of the running times for their mesh counterparts. Most of these algorithms are either optimal or very near optimal for the pyramid. In Section 6.2, the concurrent read and concurrent write operations are presented for the pyramid. These operations are then used in an algorithm to label the components of a graph, consisting of $\Theta(n^{1/2})$ vertices, in $\Theta(n^{1/4} \log n)$ time, where the graph is given as a set of unordered edges stored one per base processor. In Section 6.3, problems are considered for graphs represented as an adjacency matrix, stored in a natural fashion in the base of the pyramid. For this more structured input, pyramid matrix read and pyramid matrix write operations are introduced and used to reduce the time to solve some graph problems to $\Theta(n^{1/4})$.

In Section 6.4, problems are considered for digitized black/white pictures, such as labeling the black *figures* (i.e., connected components). Since each black pixel is a vertex, there may be $\Theta(n)$ vertices, but the geometry of the situation and the funnel read operation, which is introduced in Section 6.4, allows the labeling to be complete in $\Theta(n^{1/4})$ time. Section 6.4 also introduces the operation of reducing a function. This is used to solve the nearest neighbor problem for figures. This operation is somewhat unusual in that once the relevant data has been collected at the proper level of the pyramid, it is then spread downward to finish the calculations.

Section 6.5 concentrates on convexity problems involving *i*) multiple labeled sets of processors and *ii*) digitized black/white pictures that consist of multiple labeled figures. The operation of a sparse pyramid write is introduced, and problems for multiple figures, such as determining the extreme points, a smallest box, the smallest enclosing circle, and the diameter of each figure are examined. Section 6.6 presents details of the data movement operations, and Section 6.7 discusses the optimality of



the algorithms presented in this chapter.

Throughout the chapter, related problems, such as marking minimal-weight spanning forests, finding the transitive closure of a symmetric Boolean matrix, marking articulation points, and deciding if a graph is bipartite, are also solved.

## 6.2 Graphs as Unordered Edges

In this section, the input graph is given as a collection of unordered edges, arbitrarily distributed one per base processor, where edges may be represented more than once. This format is the most general of the input formats considered, including matrix input and digitized picture input as special cases.

### *6.2.1 Data Movement Operations*

The generic concurrent read and concurrent write operations are described in Section 1.5. In Section 2.6.4, algorithms were presented to perform the concurrent read and concurrent write operations in $\Theta(n^{1/2})$ time on a mesh of size $n$. On a pyramid, the concurrent read and concurrent write operations are extended to the pyramid read and pyramid write, respectively. These operations are now described, deferring the details of the algorithms to perform these operations until Section 6.6.1.

- In a *pyramid write*, all processors containing master records are on one level, and all processors generating update records are on the same level or some level below. (If both levels are the same, then a given processor might be responsible for a master record and also generate an update record.) As an example, consider the following "sample" call.

Pyramid write from level $L$ up to level $M$,
  For every processor on level $L$,
    if *test*1 then send($A_1, B_1, C_1$), send($A_2, B_2, C_2$);
  For every processor on level $M$,
    if *test*2 then receive($D, E, F$);

Since processors generating update records are descendents of processors maintaining master records, it must be that $L \leq M$. The terms *test*1 and *test*2 are used to represent arbitrary Boolean tests.



For a processor on level $L$, if *test*1 is true then the processor creates and sends two records (in this example), one with the key value $A_1$, with values $B_1$ and $C_1$ as data, and a second with key value $A_2$, with data items $B_2$ and $C_2$. (The key is always the first component.) If *test*1 is false, then the processor does not send any records. A processor on level $M$ will not try to receive a record if *test*2 is false. If *test*2 is true, a processor on level $M$ will try to receive a single record (in this example), where the value of the key goes into $D$, and copies of the data items go into $E$ and $F$. If *test*2 is true and no record is received, then the values of $D$, $E$, and $F$ become $\infty$.

- A *pyramid read* parallels the concurrent read in the same way that the pyramid write parallels the concurrent write. In a pyramid read, master records are maintained in processors at a given level, and request records are generated by processors on the same level or some level below. As an example, consider the following "sample" call.

Pyramid read at level $L$ from level M,
  For every processor on level M,
    if *test*1 then send($A$,$B$);
  For every processor on level $L$,
    if *test*2 then receive($C, D$);

Similar to the pyramid write, if a processor on level $L$ requests a key $C$ that has not been sent, then the data field $D$ will be set to $\infty$.

Section 6.6.1 gives implementations of the pyramid write and pyramid read. If the top level $M$ is a mesh of size m, and the bottom level $L$ is $i$ - 1 levels below, then the time for a pyramid write (read) from (at) level $L$ to (from) level $M$ is $\Theta(i + (mi)^{1/2})$.

**Lemma 6.1** *In a pyramid computer of size n, a pyramid read or pyramid write involving master records stored at level M, a mesh of size m, and request or update records, respectively, generated at level L, L ≤ M, takes* $\Theta(i + (mi)^{1/2})$ *time, where L is i - 1 levels below M.* ·

Page 245

### *6.2.2 Component Labeling*

Except for obvious differences in the computer model and data movements operations, the pyramid computer (connected) component labeling algorithm presented in this section is similar to those presented in [Hamb83, HCW79, Mill84a, NaSa80], Section 3.3.2, and the generic Component Labeling Algorithm of Section 1.6.1. Given an undirected graph $G = (V,E)$, the algorithm proceeds through a series of stages, where at each stage the vertices are partitioned into disjoint *clubs*. Vertices are in the same club only if they are in the same component of the graph. A club is called *unstable* if it is not an entire component.

Initially each vertex is its own club. During a single *stage* of the algorithm, unstable clubs are merged together to form larger clubs, and the number of unstable clubs decreases by at least half. This process is repeated until no unstable clubs remain. Since each stage of the algorithm reduces the number of unstable clubs by at least half, then at most $\lfloor \log_2 v \rfloor$ stages of the algorithm are needed to label a graph with $v$ vertices.

Every club has a unique label, which is defined to be the minimum label of any vertex in the club. During the algorithm, let $L(x)$ denote the current label of the club containing vertex $x$. Initially $L(x) = x$, indicating that each vertex is its own club. During a stage of the algorithm, clubs are merged as follows. Let $u$ be the label of an unstable club. Compute $M(u) = min\{L(y) \mid (x, y) \in E, L(x) = u\}$. The graph consisting of vertices that are labels of unstable clubs, and edges are of the form $(u, M(u))$, for all unstable clubs $u$, is called a *min-tree forest*. Merging clubs takes the form of relabeling the min-tree forest so that for each tree in the min-tree forest a new club is formed that is the union of all clubs in the tree. Given a min-tree $T = (V', E')$, the new club that is formed from $T$ is assigned the label $N(u)$, where $N(u) = \min_{u \in V'} M(u)$. Notice that in a min-tree forest, each unstable club is connected to at least one other unstable club, which guarantees that the number of unstable clubs is at least halved after each stage of the algorithm. (See the example given in Figure 6.1.)

The component labeling algorithm for a pyramid computer of size $n$ is given in Figure 6.2. It incorporates an integer function *count_keys* that counts the number of distinct keys in the base. The operation of *count_keys* is similar to that of the pyramid write, and is given in detail in Section 6.6.1.

Notice that the algorithm operates by moving the data to a place where the min-tree forest can be quickly relabeled. In Figure 6.2, the

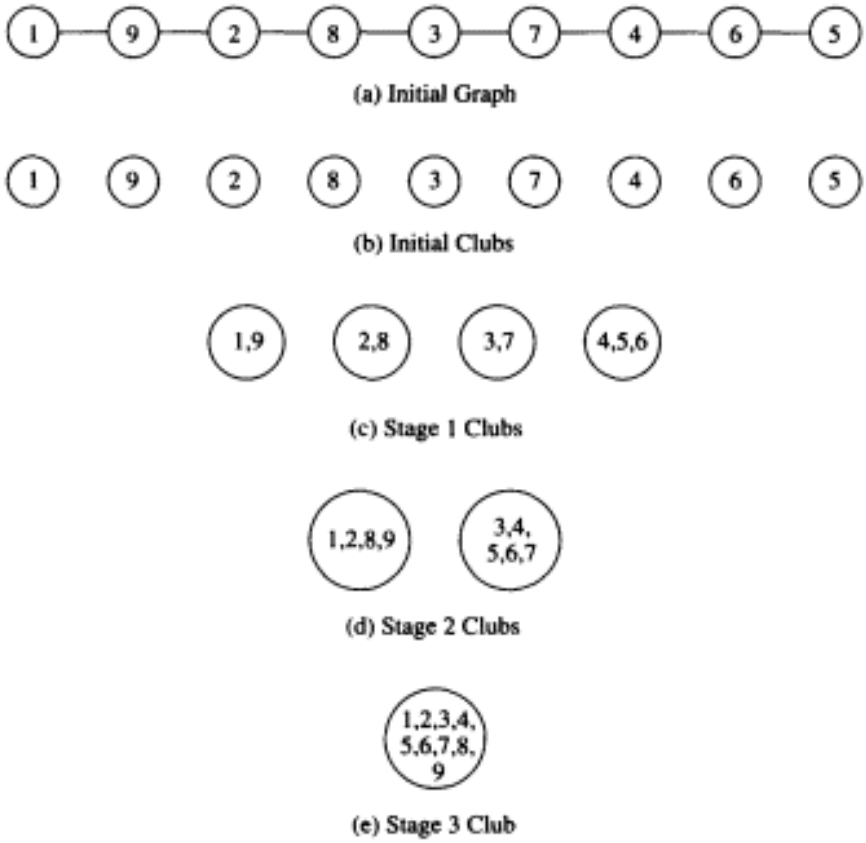



Figure 6.1:
An example of the component labeling algorithm.

term *forest_level* is used to indicate the level of the pyramid at which the forest is formed. Initially, this level must have at least $v$ processors. Each stage of the algorithm reduces the size of the forest by at least half, so after 2 stages, *forest_level* can be increased by 1. Without this upward movement, the time of the algorithm would increase by a factor of log $n$. Given a forest with $f$ vertices, the forest can be relabeled in $\Theta(f^{1/2})$ time on a mesh of size $\Theta(f)$ by the algorithm presented in [NaSa80]. This relabeling algorithm exploits the property that a min-tree forest is essentially upward directed, in that $M(u) \leq u$. Notice that if the forest data remained at the base of the pyramid, then the min-tree relabeling would take $\Theta(n^{1/2})$ time. However, by first moving the data up the pyramid, the relabeling step will take only $\Theta(v^{1/2})$ time.

**Theorem 6.2** *Given a pyramid computer of size n, if the base contains*

For every base PE ,
Label1:=Vertex1;
Label2:=Vertex2;

*v:=count_keys;* {*v* is the number of vertices}

*forest_level:=*$log_4(n/v)$*;*

for *stage*:=1 to $\log_2 v$ do

Pyramid write from the base upto level *forest_level*,
For every base PE ,
send(Label1, Label2), send(Label2, Label1);
For every PE $P_i$ at level *forest_level*,
receive($Vertex_{2_i}$, neighbor), receive($Vertex_{2_i+1}$, neighbor);

Relabel the min-tree forest, so that every PE $P_i$ at
level *forest_level* has
$Label_{2_i}$:=N($Vertex_{2_i}$) and $Label_{2_i}$+ :=N($Vertex_{2_i+1}$);

Pyramid read at the base from level *forest_level*,
For every PE $P_i$ at level *forest_level*,
if $Vertex_{2_i} < \infty$ then send($Vertex_{2_i}$,$Label_{2_i}$);
if $Vertex_{2_i+1} < \infty$ then send($Vertex_{2_i}$+ ,$Label_{2_i+1}$);
For every base PE
receive(Label1, templabel1),
receive(Label2, templabel2);

For every base PE ,
if templabel1$< \infty$ then Label1:=templabel1,
if templabel2$< \infty$ then Label2:=templabel2;

if (*stage* mod 2)=0 then *forest_level:=forest_level* + 1;

end{for};

Figure 6.2:
Component labeling algorithm.



*the unordered edges of an undirected graph with v vertices, then the above algorithm labels the components in* $\Theta(\log(n)+v^{1/2}[1+\log(n/v)]^{1/2})$ *time*.

*Proof*. Proposition 6.24 of Section 6.6.1 shows that *count_keys* finishes in the time claimed. Within the loop, at the start of an iteration, let $k$ be the number of processors at level *forest_level*. The pyramid read and pyramid write each take

$$\Theta(forest\_level + k^{1/2}[1 + forest\ level]^{1/2})$$

time, and the min-tree forest relabeling takes $\Theta(k^{1/2})$ time. Since $k = n/4^{forest\_level}$, the time for this iteration of the loop is

$$\Theta\left(forest\_level + \frac{n^{1/2}(1 + forest\_level)^{1/2}}{2^{forest\_level}}\right)$$

The initial value of *forest_level* is $\lfloor \log_4(n/v) \rfloor$, and *forest_level* increases every 2 iterations, so the total running time of the algorithm is

$$\Theta\left(\sum_{i=\lfloor \log_4(n/v) \rfloor}^{\log_4 n} \left[i + \frac{n^{1/2}(1+i)^{1/2}}{2^i}\right]\right),$$

which is

$$\Theta\left(\log(n) + v^{1/2}\left[1 + \log\left(\frac{n}{v}\right)\right]^{1/2}\right).$$

In the worst case, when the number of vertices in the graph is $v = \Theta(n)$, the pyramid algorithm takes $\Theta(n^{1/2})$ time, which is asymptotically equivalent to the running time of the mesh algorithm presented in [ReSt]. This situation arises, for example, when considering planar graphs, for which the number of vertices $v \leq 3e - 6$ edges. For smaller values of $v$, however, the pyramid algorithm exhibits significant improvement over the mesh algorithm. Any mesh algorithm to solve this problem must take $\Omega(n^{1/2})$ time, but for a dense graph with $v = \Theta(n^{1/2})$ vertices, the pyramid algorithm requires only $\Theta(n^{1/4}\log^{1/2} n)$ time.

Given a forest with $f$ vertices distributed one edge per base processor in a pyramid of size $f$, the algorithm presented in Figure 6.2 will label the connected components (i.e., the trees of the forest) in $\Theta(f^{1/2})$ time. Therefore, the mesh algorithm of [NaSa80] that is used to perform min-tree forest relabeling in the pyramid component labeling algorithm given in Figure 6.2, may be replaced by a recursive call with level *forest_level* viewed as the base of the pyramid.



### *6.2.3 Minimal Spanning Forests*

The strong similarities between component labeling algorithms and minimal spanning forest algorithms are well known. In particular, it has been noted that small changes to a component labeling algorithm for a parallel computer can give a minimal spanning forest algorithm for the same computer [CLC82, HaSi81, SaJa81]. There are two modifications that must be made to the pyramid computer component labeling algorithm, as presented in Figure 6.2, in order to arrive at a minimal spanning forest algorithm. First, a record must be kept of the edges that are used. Second, when clubs are being merged, each club must use an edge of minimal weight, rather than an edge to a club of minimal index. Notice that a club may have more than one minimal-weight edge, which may introduce cycles in the min-tree forest if the edges are not chosen in a consistent manner. In order to prevent cycles, the edges may be ordered in a consistent manner, as follows. An edge with weight $w$ between vertex $v_1$ and vertex $v_2$ is represented by the ordered triple $(w, v_1, v_2)$. Define the weighted edge $(w_1, x_1, y_1)$ to be less than weighted edge $(w_2, x_2, y_2)$ if *i*) $w_1 < w_2$, or *ii*) if $w_1 = w_2$ and $\min(x_1, y_1) < \min(x_2, y_2)$, or *iii*) if $w_1 = w_2$, $\min(x_1, y_1) = \min(x_2, y_2)$, and $max(x_1, y_1) < max(x_2, y_2)$.

Incorporating these changes is straightforward, giving the following result.

**Theorem 6.3** *Given a pyramid computer of size n, if the base contains the unordered weighted edges of an undirected graph with v vertices, then a minimal spanning forest may be determined in* $\Theta(\log(n) + v^{1/2}[1 + log(n/v)]^{1/2})$ *time.* ·

Even if the edges are unweighted, spanning forests can be quite useful. In order to decide whether or not an undirected graph $G = (V, E)$ is bipartite, let each edge have weight 1 and use the algorithm associated with Theorem 6.3 to select a spanning forest. Using a pyramid write, send the edges of the forest to level $\lfloor \log_4(n/v) \rfloor$. In each tree of the forest, select the vertex of minimum label as the root, and use the mesh algorithm in [Stou85a] at level $\lfloor \log_4(n/v) \rfloor$ to determine the depth of each vertex in its rooted tree. (This algorithm takes $\Theta(v^{1/2})$ time on a mesh of size $v$.) Say that a node is in $V_1$ if its depth is even, and is in $V_2$ if its depth is odd. It is easy to show that $G$ is bipartite if and only if this particular choice of $V_1$ and $V_2$ is such that every edge of $E$ joins a member of $V_1$ and a member of $V_2$. To check whether this property is true, perform a pyramid read so that every base processor can



determine the depths corresponding to the vertices of the edge that it contains. Finally, pass these results to the apex, combining them along the way.

This algorithm takes $\Theta(\log(n) + v^{1/2}[1 + \log(n/v)]^{1/2})$ time. Notice that a variety of graph-theoretic problems can be solved by using Theorem 6.3 to pick a spanning forest, moving the forest to level $\lfloor \log_4(n/v) \rfloor$, using a mesh algorithm at that level, and using pyramid reads and writes to move data up and down. Mesh algorithms for several graph-theoretic problems are given in [Stou85a].

**Corollary 6.4** *Given a pyramid computer of size n, if the base contains the unordered edges of an undirected graph G with v vertices, then in* $\Theta(\log(n) + v^{1/2}[1 + log(n/v)]^{1/2})$ *time the pyramid can be used to*

*a*) *decide if G is bipartite,*

*b*) *determine the cyclic index of G,*

*c*) *find all bridge edges of G*,

*d*) *find all articulation points of G, and*

*e*) *decide if G is biconnected*.

Note that some of the mesh algorithms presented in [Stou85a] are patterned after mesh algorithms appearing in [AtKo84], with the difference being that the algorithms in [AtKo84] require matrix input, while those in [Stou85a] use only unordered edge input. The algorithms from [AtKo84] are unsuitable because there may not be $v^2$ processors to hold the adjacency matrix. More importantly, the algorithms from [AtKo84] are too slow because they use matrix calculations that take $\Theta(v)$ time on a pyramid. (Algorithms from [AtKo84] appear in Section 3.2.)

## 6.3 Graphs as Adjacency Matrices

In this section, undirected graphs with $n^{1/2}$ vertices are considered, where the graph is given as an adjacency or weight matrix. The $(i, j)$ entry of the matrix is assumed to be stored in base processor $P_{i,j}$. Because the input is now more structured, algorithms that are slightly faster than those of Section 6.2 are possible.

### *6.3.1 Data Movement Operations*

The algorithms presented in this section require two new data movement operations, namely the *pyramid matrix write* and *pyramid matrix read*.



A pyramid matrix write performs the same basic action as a pyramid write and comes in two versions, one for rows and one for columns. In the row (column) version, update records are generated by base processors, where the base processors in the same row (column) generate update records with the same key. The pyramid matrix read performs the same basic action as a pyramid read, and also comes in two versions. For the row (column) version, request records are generated by base processors, where the base processors in the same row (column) request information regarding the same key.

Detailed implementations of these operations appear in Section 6.6.2, where it is shown that if the master records are maintained in a mesh of size $k$, $k \leq 2n^{1/2}$, then the pyramid matrix read and pyramid matrix write operations are complete in

$$\Theta(\log(n) + k^{1/2}[1 + log(n/k^2)]^{1/2})$$

time. (Though there will never be more than $n^{1/2}$ keys, $k = 2n^{1/2}$ is allowed since the highest level holding $n^{1/2}$ processors actually has $2n^{1/2}$ processors when $n > 256$ is an odd power of 4.)

### *6.3.2 Component Labeling and Minimal Spanning Forests*

Algorithms for graphs given as adjacency or weight matrices can be adapted from those algorithms presented in Section 6.2. for unordered edge input. This can be done by removing the call to *count_keys*, initializing $v$ to $n^{1/2}$, replacing pyramid read with pyramid matrix read, and replacing pyramid write with pyramid matrix write. The resulting algorithms are faster than those presented in Section 6.2 by a factor of $\Theta(\log^{1/2} n)$. This comes from the fact that the running times of both the unordered edge algorithms and the matrix algorithms sum as a geometric series, with the major term (ignoring *count-keys* for the moment) being dictated by the time to move data between the base and the middle level of the pyramid that contains $n^{1/2}$ processors. The pyramid read and pyramid write between the base and this level each require

$$\begin{aligned}\Theta\left(\log\left(\frac{n}{n^{1/2}}\right)+\left[n^{1/2}\log\left(\frac{n}{n^{1/2}}\right)\right]^{1/2}\right) &= \Theta(\log(n)+[n^{1/2}\log n]^{1/2}) \\ &= \Theta([n^{1/2}\log n]^{1/2}) \\ &= \Theta(n^{1/4}\log^{1/2} n)\end{aligned}$$

Page 252

time, as does *count_keys*, while the pyramid matrix read and pyramid matrix write between the base and this level only require

$$\begin{aligned}\Theta\left(\log(n)+(n^{1/2})^{1/2}\left[1+\log\left(\frac{n}{(n^{1/2})^2}\right)\right]^{1/2}\right) \\ = \Theta(\log(n)+n^{1/4}[1+\log(1)]^{1/2}) \\ = \Theta(n^{1/4})\end{aligned}$$

time. In both cases, it is assumed that $n$ is sufficiently large (i.e., the additive log $n$ term is insignificant in the $\Theta$-notation).

**Theorem 6.5** *Suppose the adjacency matrix of an undirected graph with $n^{1/2}$ vertices is stored in the base of a pyramid computer of size $n$. Then the connected components can be labeled in $\Theta(n^{1/4})$ time.* ·

**Theorem 6.6** *Suppose the weight matrix of a weighted undirected graph with $n^{1/2}$ vertices is stored in the base of a pyramid computer of size $n$. Then a minimal spanning forest can be marked in $\Theta(n^{1/4})$ time.* ·

**Corollary 6.7** *Suppose the adjacency matrix of an undirected graph $G$ with $n^{1/2}$ vertices is stored in the base of a pyramid computer of size $n$. Then in $\Theta(n^{1/4})$ time the pyramid can be used to*

*a) decide if $G$ is bipartite,*

*b) determine the cyclic index of $G$,*

*c) find all bridge edges of $G$,*

*d) find all articulation points of $G$, and*

*e) decide if $G$ is biconnected.*

Determining the transitive closure of a symmetric Boolean matrix stored in the base of a pyramid is a simple adaptation of component labeling. First, perform component labeling for matrix input. For processors that are storing off-diagonal entries (i.e., for which the row and column are different), the new entry is 1 if the row label equals the column label, while otherwise it remains 0. For processors on the diagonal, if the original entry was 1, it remains so, while if it was 0, then it becomes I only if some other entry in the row is 1. Pyramid matrix reads and writes can be used to determine the proper diagonal entries, as follows. Each base processor creates a record consisting of its row index as



the key, with its new entry stored in the data field. A pyramid matrix write to the middle level of the pyramid is used to combine records with the same key, breaking ties in favor of maximum data fields. A pyramid matrix read from this level is then used so that every diagonal entry of the matrix can determine whether or not there is a 1 in its row.

**Corollary 6.8** *Suppose an* $n^{1/2} \times n^{1/2}$ *symmetric Boolean matrix is stored in the base of a pyramid computer of size* $n$*. Then the transitive closure of this matrix can be determined in* $\Theta(n^{1/4})$ *time.* ·

### 6.4 Digitized Pictures

In this section, a bottom-up divide-and-conquer approach is used to solve a variety of geometric problems involving digitized black/white pictures stored in the base of the pyramid. In the previous chapter, it was shown that divide-and-conquer can be used on the pyramid to produce efficient algorithms to solve problems by quickly eliminating data as partial solutions are merged from the base to the apex. Unfortunately, for the problems considered in this section, the data movement requirements are substantially greater than for the problems considered in Chapter 5. In this section, new data movement operations are introduced and efficient implementations of a divide-and-conquer strategy are demonstrated for the pyramid. It will also be shown, in Section 6.7, that the results obtained in this section are at most a factor of $\Theta(\log^{1/2} n)$ from optimal for the pyramid.

Throughout this section, the mesh at some level will be divided into squares of some size $t$. This means that the mesh will be completely partitioned into disjoint squares of size $t$, where $t$ is a power of 4. Using this partitioning, the concept of *the square of size t at level l containing processor P* is well-defined, assuming that level $l$ is of size $t$ or greater. The term *picture square* will be used to refer to such a square in the base.

The computations will proceed in a bottom-up fashion. The first *stage* of the algorithm will involve analyzing picture squares of size $4c$, for some constant $c$ that depends upon the particular problem. In general, at the end of stage $i$, $i \geq 1$, picture squares of size $4^{c+i-1}$ have been analyzed, where the *analysis* is with respect to the particular problem being solved (e.g., labeling figures or determining nearest neighbors). During stage $i + 1$, results from stage $i$ are combined so as to analyze picture squares of size $4^{c+i}$.

An important property of the solution strategy presented in this section is to reduce the amount of data from an amount proportional to the area of the picture square to an amount proportional to the perimeter of the picture square under examination. So at the end of stage $i$, every picture square of size $4^{c+i-1}$ will be reduced to $O(2^i)$ pieces of data, from which stage $i + 1$ can produce the analysis for picture squares of size $4^{c+i}$. The algorithms presented in this section proceed rapidly by moving the perimeter amount of data that is used to represent a picture square up through the subpyramid over that picture square.

For a picture square $S$ of size $4^{c+i-1}$, the square of size $4^{c+\lceil i/2 \rceil - 1}$ at level $\lfloor i/2 \rfloor$ of the pyramid that contains the ancestors of $S$ is called the *data square corresponding to the picture square of S*. Intuitively, the data square corresponding to a picture square $S$ of size $m$ will be a square $\hat{S}$ of size $O(m^{1/2})$. Further, this data square $\hat{S}$ will be located at the middle level of the subpyramid that has $S$ as its base. This means that a data square contains enough processors to store a perimeter amount of data from its respective picture square. For the algorithms presented in this section, the focal point of work to be performed on a picture square $S$ is at the data square $\hat{S}$ corresponding to the picture square $S$.

Note that the data square corresponding to a picture square is either the union of the data squares corresponding to the picture square's quadrants, or else it is the union of the parents of the quadrants' data squares. This means that the data used during stage $i$ of the algorithm is either already in place, or must move up only one level in the pyramid, in order to be where it is required for processing during stage $i + 1$.

The last stage of the bottom-up divide-and-conquer algorithm is stage $\log_4(n) - c + 1$, which is responsible for an analysis over the entire picture. During the course of the algorithm, intermediate results will not be sent down to picture squares. Therefore, at the end of the algorithm, a final step is needed to move these results back down to the base. This final data movement is accomplished with a *funnel read*, which is described in Section 6.4.1. Section 6.4.1 also introduces a data movement operation called *reducing a function*. This operation allows data squares to perform some calculations (such as computing a nearest neighbor for each point from a set of points) in time proportional to the edgelength of the square, even though mesh algorithms that finish in this time are not currently available. The operation of reducing a function uses processors below the data square to help perform the calculations in the desired time.



### *6.4.1 Data Movement Operations*

In this section, data movement operations are described that will be used in some of the bottom-up divide-and-conquer algorithms presented later in this chapter.

- *Funnel Read:* Assume every base processor knows the key corresponding to data it wishes to read from its stage 1 data square. Further, assume that for a stage $i$ data square that is responsible for supplying the data for a given key, either

    1. one of its processors has the data, or

    2. it must read the data from its stage $i + 1$ data square (where a stage $i + 1$ data square means the data square it supplies data to), or

3. one of its processors has an alias for the key and must read the data for the alias from its stage $i + 1$ data square.

Notice that if $i$ is the last stage of the algorithm, then the data square must have the desired data. Finally, assume a data square of size $m$ never receives more than $\Theta(m)$ such requests. Then, the funnel read ultimately obtains the data for all base processors in $\Theta(m^{1/2})$ time, where $m$ is the size of the data squares at the final stage. Figure 6.3 gives a picture of a funnel read. The details of the funnel read are deferred to Section 6.6.3.

- *Reducing a function:* Given sets $Q$, $R$, and $S$, let $g$ be a function mapping $Q \times R$ into $S$, and let * be a commutative, associative, binary operator over $S$. Define a map $f$ from $Q$ into $S$ by $f(q) = *\{g(q, r) \mid r \in R\}$, where $f$ is said to be the *reduction of* $g$. For example, if $Q$ and $R$ are sets of points in some metric space, if $S$ is the real numbers, if $g(q, r)$ is the distance from $q$ to $r$, and if * is minimum, then $f(q)$ is the distance from $q$ to the nearest point in $R$.

  Suppose the elements of $Q$ are stored one per processor in a square of size $m$ at level $i$ of the pyramid, and the elements of $R$ are also stored one per processor in the square. (A processor may contain an element of $Q$ and an element of $R$.) Suppose $g$ and * can both be computed in $\Theta(1)$ time. Then the operation of reducing a function will compute $f(q)$ and store the result in the processor containing $q$, for all $q \in Q$, in $\Theta(m^{1/2} + m/4^i)$ time. The details of this operation appear in Section 6.6.3.

Page 256

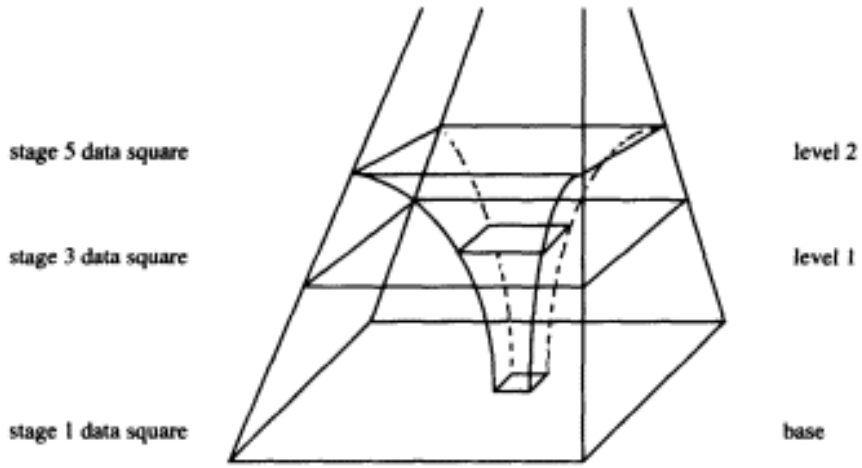



Figure 6.3:
A single processor's view of a funnel read.

### *6.4.2 Component Labeling*

A digitized picture can be viewed as an undirected graph, where the black pixels are vertices, and adjacent black pixels have an (undirected) edge between them. Upon termination of the digitized picture component labeling algorithm, every base processor containing a black pixel must contain the label of the pixel's figure (i.e., connected component), which will correspond to the minimum index of any processor containing a pixel in its figure.

The component labeling algorithm presented in this section follows the basic divide-and-conquer strategy outlined at the beginning of Section 6.4, and is similar to the mesh algorithm of Section 3.3.2. However, the algorithm presented in this section for the pyramid computer is significantly faster than the optimal mesh algorithm presented in Section 3.3.2.

To initialize the algorithm, every base processor that contains a black pixel will generate an edge record corresponding to each of its, at most four, neighboring base processors that also contains a black pixel. So, every black processor $P$ (i.e., every base processor $P$ containing a black pixel) will generate an *edge record* $(p, q, \infty)$ corresponding to each neighboring black processor $Q$, where $p$ is the index of $P$, $q$ is the index of $Q$, and the third component of the record will be used to store the com-



ponent label of $p$ and $q$ generated during stage 1. Therefore, every base processor may generate as many as four edge records, one corresponding to each of its neighbors.

Picture squares of size 256 are labeled during stage 1.. (That is, $c = 4$ in the generic divide-and-conquer strategy.) For every picture square of size 256, a labeling algorithm is applied to the subset of records $(x, y, \infty)$ for which both $x$ and $y$ are in the picture square. This means that those records for which $x$ is on the border of one such picture square and $y$ is on the border of an adjacent picture square are omitted from the stage 1 labeling process. Notice that since $x$ and $y$ are concatenated coordinates of processors, that in $\Theta(1)$ time a processor containing a record of the form $(x, y, \infty)$ can decide whether or not $x$ and $y$ are in the same picture square of size 256, and therefore can decide whether or not this record is to be included in the stage 1 labeling. Since the size of the square to be labeled is a constant, stage 1 labeling can be performed in $\Theta(1)$ time simultaneously for every picture square of size 256. A simple propagation algorithm, or the unordered edge component labeling algorithm of Section 6.2.2, is all that is needed.

After completing the stage 1 labeling, notice that records of the form $(x, y, \infty)$ represent edges between distinctly labeled figures in adjacent picture squares. (These records were not included in the stage 1 labeling algorithm.) For every such record $(x, y, \infty)$, perform a mesh concurrent read within $x$'s picture square to determine the (possibly) new label of $x$. Store this new label, call it $l_x$, in the third field of the record, so that the record now has the form $(x, y, l_x)$. Since these records represent edges between distinctly labeled figures of adjacent picture squares, the correct labels for the figures that these edges are incident on have yet to be determined (even after the mesh concurrent read). However, for all other figures (i.e., for those that do not span at least two picture squares), their final labels have already been determined by the stage 1 unordered edge labeling algorithm.

All original records (with updated third fields) are kept in their stage 1 data squares, and every processor containing a record $(x, y, l_x)$ with y outside of $x$'s picture square generates a record $(x.orig, y.orig, x.label, y.label)$, where $x.orig = x$, $y.orig = y$, $x.label = l_x$, and $y.label = \infty$, for use in the next stage of the algorithm. There are at most 64 such records generated within a single picture square of size 256. (Each corner pixel may generate 2 such records, and all other border pixels may generate 1.) These records are now spread out in their stage 1 data squares so that no processor holds more than one such record. This concludes stage 1.

During stage $i$, the algorithm labels picture squares of size $4^{3+i}$, using data squares of size $4^{3+\lceil i/2 \rceil}$ at level $\lfloor i/2 \rfloor$, as follows. If $i$ is odd, then all of the necessary data (i.e., the records generated at the end of stage $i$ - 1) is already present, while if $i$ is even, then the necessary data is in the four data squares one level below. In the latter case, in $\Theta(2^{i/2})$ time, the data is moved up and distributed so that no processor has more than one record.

Next, perform a mesh concurrent read within each stage $i$ data square so that every record ($x.orig$, $y.orig$, $x.label$, $\infty$) can fill in the fourth field that corresponds to the label of y from the end of stage $i$ - 1. Every processor containing a record ($x.orig$, $y.orig$, $x.label$, $y.label$) now creates a record ($x.label$, $y.label$, $\infty$), and component labeling is performed for this unordered edge input in the stage $i$ data square, again using only edges for which both vertices lie in the same stage $i$ picture square. When finished, a processor containing a record with a vertex outside of the picture square generates a record (with 4 fields) for the next stage. Since work is being performed on picture squares of size $4^{3+i}$, at most $2^{5+i}$ records can be generated for the next stage.

After stage $\log_4(n)$ - 3, the labels of all figures have been decided. The data square for the last stage is a level of the pyramid consisting of a mesh of size $\Theta(n^{1/2})$, for which the mesh unordered edge component labeling algorithm of [ReSt] takes $\Theta(n^{1/4})$ time. The running time of the algorithm, as described, is given by the recurrence $T(n) = T(n/4) + \Theta(n^{1/4})$, which is $\Theta(n^{1/4})$. Unfortunately, at the end of the algorithm that has been described, the labels remain scattered throughout the pyramid. A collation step is needed that will enable every base processor to obtain the final label of its pixel.

Notice that if $P$'s figure extends outside of $P$'s stage 1 picture square, then the labeling information in the stage 1 data square may be incorrect, and $P$ would need to consult data squares of later stages in order to obtain the correct component label. The figure may extend outside of $P$'s picture square for many stages, so in advance $P$ does not know which data square has the needed labeling information. This is where a funnel read is used, moving labels from the data squares of the last stage back down towards the base, taking $\Theta(n^{1/4})$ time, and completing the algorithm substantially faster than the optimal $\Theta(n^{1/2})$ mesh algorithm of Section 3.3.2.

**Theorem 6.9** *Given a digitized picture stored one pixel per processor in a natural fashion in the base of a pyramid computer of size $n$, in $\Theta(n^{1/4})$ time, the connected components can be labeled.* ·



A pyramid computer component labeling algorithm designed to operate efficiently in a more restrictive domain was introduced in [Tani82a]. The algorithm is designed to consistently label every "convex blob" with the label of a distinguished member of the blob (component). The algorithm uses predominantly the child-parent links in the pyramid, but is somewhat different from the tree-like pyramid algorithms presented in Section 5.4.4, in that data 'bounces' between levels instead of just traveling directly between the base and the apex. The algorithm finishes labeling a "convex blob" of diameter $D$ in $\Theta(\log D)$ time by continually propagating the label of a distinguished black base processor to neighboring processors that cover an entirely black piece of the blob. Therefore, the algorithm terminates in time proportional to the logarithm of the "convex blob" with largest diameter. This algorithm is not intended for arbitrary digitized pictures. In fact, it would require $\Theta(n^{1/2})$ time to label a $D \times n^{1/2}$ rectangle, for any constant $D$. In contrast, the algorithm presented in this section will label any digitized picture containing multiple figures of any shape in $\Theta(n^{1/4})$ time.

### *6.4.3 Nearest Neighbors*

Given digitized picture input in the base of the pyramid, an efficient solution to the all-nearest neighbor problem for figures can be obtained quite simply from the solution just presented to the digitized picture component labeling problem. Therefore, in this section, only those aspects of the algorithm that change will be described in detail.

In the all-nearest neighbor problem for figures, it is required that the kin of each figure be detected, where the *kin* of a figure is the label/distance pair representing a nearest distinctly labeled figure. (In case of ties, the figure of smallest label is chosen.) In this section, input to the all-nearest neighbor problem is a digitized picture with its figures already labeled, and at the conclusion of the algorithm, every base processor containing a black pixel will know the kin of its pixel's figure.

The bottom-up divide-and-conquer algorithm is based on the following observation. Assume that the 4 quadrants within a picture square have been analyzed. Then the only figures that might not have their correct kin information with respect to the entire picture square, are those figures that have pixels that are either the topmost or bottommost black pixel in its column, or the leftmost or rightmost black pixel in its row. (In fact, it is possible to restrict the set of candidates even further, but this is not necessary.) A pixel that is either the topmost or bottommost black pixel in its column, or the leftmost or rightmost black



pixel in its row, is called a *special pixel*. Within a quadrant, figures with no special pixels must have determined their kin during earlier stages of the algorithm since they are totally surrounded by other figures within their quadrant.

Stage 1 of the algorithm analyzes picture squares of size 256. Within every picture square, for each figure $C$, a closest figure within the square is determined and stored in a record $(C, kin(C))$. (This kin information maybe incorrect globally, but the final funnel read will bring the correct global information down from data squares above.) For every column $i$ in the picture square, form the records $(1, i, tr(i), tl(i))$ and $(2, i, br(i), bl(i))$, where $tr(i)$ is the row of the topmost black pixel in the column restricted to the square, $br(i)$ is the row of the bottommost black pixel in the column restricted to the square, and $tl(i)$ and $bl(i)$ are the labels of the pixels at locations $(tr(i), i)$ and $(br(i), i)$, respectively. (If the column has no black pixel, then set the coordinates to $\infty$.) Similarly, for every row $j$, form records $(3, j, lc(j), ll(j))$ and $(4, j, rc(j), rl(j))$, corresponding to the leftmost and rightmost black pixels in the row, respectively. These are the records needed for the next stage of the algorithm.

The purpose of stage $i + 1$ is to find for every black pixel represented in a stage $i + 1$ data square, a nearest black pixel of a different label within that data square. This is accomplished by using the operation of reducing a function, where $Q$ and $R$ are the records, $S$ is the real numbers, * is minimum, and $g$ is distance, with the exception that $g$ gives an infinite distance if the two points have the same label. When the operation is finished, a mesh concurrent read is used to form a record $(C, kin(C))$ for every figure $C$ represented by one or more pixels. To generate the records for the next stage, notice that for every column in the stage $i + 1$ picture square there are two type 1 records. The one representing the topmost pixel is passed to the next stage, and similar reductions occur for records of types 2, 3, and 4.

Finally, after the last stage of the algorithm, a funnel read brings the correct kin information back to the base.

**Theorem 6.10** *Given a digitized picture stored one pixel per processor in a natural fashion in the base of a pyramid computer of size $n$, in $\Theta(n^{1/4})$ time, the all-nearest neighbor problem for figures can be solved.*

It should be noted that every black pixel can determine the location of a nearest black pixel in $\Theta(\log n)$ time [Stou85b].



## 6.5 Convexity

In this section, problems are considered that involve convexity of multiple figures or multiple labeled sets of base processors. The problems include enumerating extreme points, deciding convexity, and using extreme points to solve problems such as determining diameter, smallest enclosing rectangles, and smallest enclosing circles for every figure. Since there may be $\Theta(n)$ disjoint sets of base processors in a pyramid computer of size $n$, applying the tree-like algorithms of Section 5.5.1 to one set of processors at a time would yield substantially suboptimal running times in the worst case. In order to efficiently determine convexity properties for multiple sets of base processors, it appears that the algorithms must be designed to work on multiple sets simultaneously. Further, since $\Omega(n^{1/2})$ time is required if only the base mesh of the pyramid is used, faster algorithms must use both the parent-child and mesh links that are available in the pyramid. (Refer to Section 3.5 for optimal mesh algorithms concerning convexity of multiple sets of processors.) Finally, the algorithms must avoid having many figures trying to send data through the apex, for then the apex becomes a bottleneck.

The running times of algorithms presented in this section are slower than the running times of algorithms from Section 5.5.1 that involved single figures. Nevertheless, the results presented in this section are at most a logarithmic factor from optimal for the pyramid.

### *6.5.1 Data Movement Operations*

The pyramid write, which was introduced in Section 6.2.1, is used to move data up the pyramid from a given mesh level to a desired level that contains enough processors to hold all of the distinct pieces of data being sent. For the algorithms in this section, only a restricted version of the pyramid write is needed. The restriction is for the situation where only $Cn^p$ base processors wish to send a piece of data to the highest mesh level that can hold all of this data, for fixed constants $p$ and $C$. This restricted version of the pyramid write can be performed in $\Theta(n^{p/2}\log^{1/2} n)$ time. The details of the pyramid write are given in Section 6.6.1, and the restricted version of the pyramid write is derived directly from that.

**Lemma 6.11** *Fix constants $p$ and $C$, where $0 < C$ and $0 < p < 1$. Given a pyramid computer of size $n$, suppose there are no more than $Cn^p$ processors in the base that have a piece of data to be sent to level*



$\log_4 \left(\frac{n^{1-p}}{C}\right)$. *(This level is the highest one with at least $Cn^p$ processors.) The pyramid write will move the data to its proper location in $\Theta(n^{p/2}\log^{1/2} n)$ time.* ·

A closely related data movement operation is the *sparse pyramid write*, which is an extension of the pyramid write operation, and which is crucial to some of the algorithms in this section. The sparse pyramid write again assumes that for fixed constants $p$ and $C$, only $Cn^p$ base processors have a piece of data to be sent to the mesh at the highest level of the pyramid that can hold all of this data. However, one further restriction applies to the sparse pyramid write, namely, the assumption that in each subsquare of size $k$ in the base of the pyramid, $0 \leq k \leq n$, there are no more than $Ck^p$ processors sending data. With this additional constraint, the time for the sparse pyramid write is reduced by a factor of $\log^{1/2} n$ over the restricted pyramid write given in Lemma 6.11. The details are given in Section 6.6.1.

**Lemma 6.12** *Fix constants $p$ and $C$, where $0 < C$ and $0 < p < 1$. Given a pyramid computer of size $n$, suppose there are no more than $Cn^p$ processors in the base that have a piece of data to be sent to level $\log_4(\frac{n^{1-p}}{C})$. Further, in each subsquare of size $k$ in the base of the pyramid, $0 \leq k \leq n$, assume that there are no more than $Ck^p$ processors sending data. Then a sparse pyramid write will move the data to level $\log_4(\frac{n^{1-p}}{C})$ $n$ $\Theta(n^{p/2})$ time.*

The algorithms of this section will also use the operation of reducing a function, as described in Section 6.4.1, as well as an extended reduction operation. This extended reduction operation is performed for the situation where there are three sets $A_1$, $A_2$, and $A_3$, a function $g$ mapping $A_1 \times A_2 \times A_3$ into $C$, and an associative, commutative operation * on $C$. The *extended reduction of $g$* is the function $f$ mapping $A_1$ to $C$ given by

$$f(a) = *\{g(a, x, y) \mid x \in A_2, y \in A_3\},$$

for $a \in A_1$. Details of this operation are presented in Section 6.6.

**Lemma 6.13** *Suppose that $g$ and * can be computed in unit time, and that $A_1$, $A_2$, and $A_3$ are stored one item per processor at a level with $m$ processors, $1 \leq m \leq n^{1/3}$. Then the reduction of $g$ can be computed in $\Theta(m^{1/2})$ time, storing $f(a)$ in the processor storing $a$.* ·



### *6.5.2 Enumerating Extreme Points*

The first algorithm presented in this section is used to enumerate the extreme points for each figure (i.e., connected component) in a digitized picture. The algorithm uses a bottom-up divide-and-conquer strategy, as follows.

For each figure, first enumerate the extreme points of the restriction of the figure to each of the 4 quadrants of the picture. Next for each figure in two or more quadrants, as shown in Figure 6.4, determine which points are extreme points in the quadrant, but are not extreme points in the entire figure. These form an interval, e.g., in Figure 6.4 they are the ones between the dotted lines. To find these dotted lines, use a binary search on the convex hull edges of the (at most 4) pieces of the figure. This binary search follows the generic Fixed Subset Division Algorithm given in Section 1.6.2. For example, in Figure 6.4, the topmost dotted line can be found as follows. Find a leftmost and rightmost extreme point of the restriction of the figure to the right subimage. Using this information, find and send the upper convex hull edge that is in the middle of these two extreme points in the enumeration ordering (as restricted to the right subimage) to the left subimage. Next, determine if the line collinear with this edge passes above the restriction of the figure to the left subimage, passes through or below it, or is tangent to it (and hence is the dotted line). In the first case, the edge and all convex hull edges preceding it in the counterclockwise ordering (with respect to the restriction of the figure to the right subimage) are eliminated from further consideration, while in the second case the edge and all convex hull edges following it are eliminated.

Next, the lefthand piece sends over its middle edge, and a similar check eliminates half of the convex hull edges. A binary search for the top dotted line continues in a natural fashion, alternating between the halves. Eventually, either an edge on the dotted line is found, or else both pieces locate a processor representing an extreme point such that the edge on one side is too high, and the edge on the other side is too low. In this case the dotted line passes through the processor. Once the intervals of extreme points between the dotted lines have been determined, it is easy to enumerate the remaining points using their old enumeration information.

There may be $\Theta(n^{1/2})$ figures merging pieces together, so for each step of the binary search, for all figures simultaneously, a hull edge is moved up to a level of size $\Theta(n^{1/2})$, across the level, and down to the piece on the other side. A sparse pyramid write, with $p = 1/2$, may



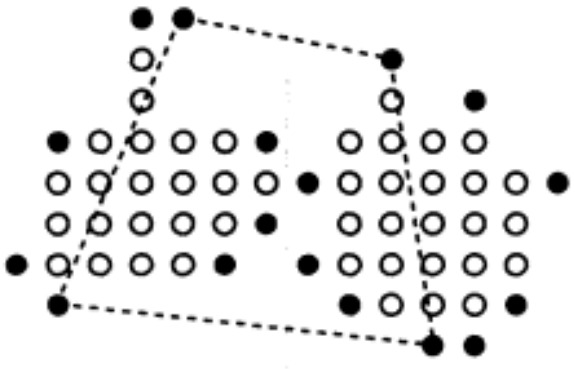


Figure 6.4:
Not all extreme points of a quadrant are extreme points of the figure.

be used to move the data up. This sparse pyramid write can be used since, in any subsquare of size $k$, if a piece of data is being moved up, then it is in a figure crossing the border of the subsquare, and there are $O(k^{1/2})$ such figures. A similar operation moves the data down. The time obeys a recurrence equation of the form $T(n) = T(n/4)+cn^{1/4}\log n$, c a constant, which has a solution of $T(n) = \Theta(n^{1/4}\log n)$.

**Theorem 6.14** *Given a digitized black/white picture in the base of a pyramid computer of size n, in* $\Theta(n^{1/4}1\log n)$ *time the extreme points of every figure can be enumerated.* ·

Suppose it is known that all of the figures in the digitized picture are convex. Then by incorporating the approach of Theorem 5.8, the time of the previous theorem for enumerating the extreme points of each figure can be reduced by a factor of $\Theta(\log n)$.

**Corollary 6.15** *In a pyramid computer of size n with a digitized picture in its base, suppose all the figures are convex. Then the extreme points of every figure can be enumerated in* $\Theta(n^{1/4})$ *time.* ·

In Section 5.5.1, the algorithm associated with Corollary 5.10 can be used to decide whether or not a figure in a digitized picture is convex.



This algorithm was designed by making a minor modification to the algorithm associated with Theorem 5.8, which enumerates the extreme points of a convex figure. A similar modification can be made so that for each figure in a digitized picture, it can be detected whether or not the figure is convex.

**Corollary 6.16** *Given a digitized picture stored one pixel per processor in a natural fashion in the base of a pyramid computer of size n, in* $\Theta(n^{1/4})$ *time every figure can decide whether or not it is convex.* ·

Suppose that an arbitrary number of (not necessarily connected) labeled sets of processors are given in the base of the pyramid, and that the extreme points of each such labeled set are to be enumerated. A wire counting argument shows that in the worst case, $\Theta(n)$ messages may have to cross from the left half of the pyramid to the right half of the pyramid. Therefore, any pyramid computer algorithm to solve this problem will require $\Omega(n^{1/2})$ time. Since a mesh algorithm to solve this problem in $\Theta(n^{1/2})$ time was presented in Section 3.5, the pyramid structure above the base mesh may be ignored, and the mesh computer algorithm may be used to enumerate the extreme points of every set of base processors.

**Proposition 6.17** *In a pyramid computer of size n, in* $\Theta(n^{1/2})$ *time the extreme points of the processors with the same label can be enumerated, simultaneously for all labels.* ·

### *6.5.3 Applications of Extreme Points*

Given multiple labeled sets of base processors, the algorithms presented in this section make use of enumerated extreme points. The problems considered include determining a smallest enclosing box, the smallest enclosing circle, and the diameter for every labeled set of base processors.

Given a metric $d$ and a set $S$ of base processors, the *diameter of S with respect to d* is $\max\{d(P, Q) | P, Q \in S\}$. Assume that $d$ is one of the $l_p$ metrics, such as the $l_1$ (taxi-cab) metric, the $l_\infty$ (chessboard) metric, or the $l_2$ (Euclidean) metric. The $l_p$ distance from $(a, b)$ to $(c, d)$ is $(|a - c|^p + |b - d|^p)^{1/p}$, for $1 \le p < \infty$, and the $l_\infty$ distance from $(a, b)$ to $(c, d)$ is $\max(|a - c|, |b - d|)$. These metrics can be computed in unit-time, and for them the diameter is $max\{d(P, Q) | P$ and $Q$ are extreme points of $S\}$. It should be noted that metrics other than the $l_p$ metrics could also be used, and although

a discussion of appropriate metrics is outside the focus of this book, the reader might care to review the discussion of metrics that was distributed throughout Section 3.6.

Given a set $S$ of points in the plane, a *smallest enclosing rectangle* (also known as a *smallest box*) is a rectangle of least area containing $S$. (If rectangles of zero area contain $S$, then the smallest such line segment is used as the smallest enclosing box of $S$.) If $S$ is finite, then it can be shown that a smallest enclosing rectangle must contain an extreme point of $S$ on each side, and at least one side must contain two consecutive extreme points [FrSh75] (i.e., an edge of the convex hull of $S$). The *smallest enclosing circle* is the circle of least area containing $S$. Smallest enclosing rectangles and smallest enclosing circles appear in [FrSh75, MiSt85b, Tous80] and were discussed in Chapters 3 and 4 for the mesh.

Algorithms to solve these problems for a *single* set of processors rely on the number-theoretic fact that for a set of lattice points in a square of size $k$, there are $O(k^{1/3})$ extreme points [VoK182]. Therefore, in $\Theta(n^{1/6})$ time, a sparse pyramid write can be used to move the extreme points of a labeled set of processors to a level in the pyramid that consists of a mesh of size $\Theta(n^{1/3})$.

To determine diameter, let $E$ be the set of extreme points and let $d$ compute the given metric. Let $g(e)$ represent the maximum distance from $e \in E$ to any other processor in set $E$. Then $g$ is defined on $E$ as $g(e) = max\{d(e, x) \mid x \in E\}$. Using the operation of reducing a function, $g$ can be computed in $\Theta(n^{1/6})$ time for all $e \in E$. Once this is accomplished, the diameter of $E$, which is just $max\{g(e) \mid e \in E\}$, can be computed in $\Theta(\log n)$ time.

A smallest enclosing rectangle can be found in a similar manner. For each hull edge, assume an orientation of the points that has this edge as the southernmost edge parallel to the $x$-axis, and use the reduction operator to find the northernmost, westernmost, and easternmost points. For each hull edge, these three points determine the minimum-area enclosing rectangle that includes the edge. A smallest enclosing rectangle of the entire set is found by taking a minimum over these rectangles (ties broken arbitrarily).

The smallest enclosing circle is the largest circle either passing through 3 of the extreme points or having 2 of the extreme points as a diameter. Thus, the smallest enclosing circle can be found by using an extended reduction of a function, which is complete in $\Theta(n^{1/6})$ time.

The results are summarized in the following theorem.

**Theorem 6.18** *In a pyramid computer of size n, suppose the extreme*



points of a labeled set of processors have been marked. Then in $\Theta(n^{1/6})$ time, the diameter (measured with any given $l_p$ metric), smallest enclosing circle, and a smallest enclosing rectangle can be determined.
.

An interesting open problem is extending the results of Theorem 6.18 to the situation where multiple figures exist (possibly with their extreme points enumerated) in the digitized picture stored in the base of a pyramid of size $n$. While [MiSt84c] did not consider finding the diameter, a smallest enclosing box, and the smallest enclosing circle for multiple figures, it is straightforward to modify the algorithms in that paper to do so in $\Theta(n^{1/3})$ time. However, the optimality of these results is open.

In Section 5.5.2, Theorem 5.15 shows that determining whether or not a figure in a digitized picture is convex, can be used to decide whether or not the figure could have arisen as the digitization of a straight line segment. Using the algorithm of Corollary 6.16 to decide whether or not each figure is convex, and following the general procedure outlined in Theorem 5.15, in $\Theta(n^{1/4})$ time, it can be decided whether or not each of these figures could have arisen as the digitization of a straight line segment.

**Corollary 6.19** *Given a digitized picture stored one pixel per processor in a natural fashion in the base of a pyramid computer of size n, in* $\Theta(n^{1/4})$ *time it can be decided for every figure whether or not it could have arisen as the digitization of a straight line segment.* ·

Consider the problem of detecting whether or not the convex hull of each figure is intersected by the convex hull of some other figure. An algorithm similar to those presented in Section 6.5.2 that makes use of a sparse pyramid write, a grouping operation for the mesh, as described in Chapter 4, and a funnel read, will provide an efficient pyramid solution.

**Theorem 6.20** *In a pyramid computer of size n, if the extreme points of each figure have been determined, then in* $\Theta(n^{1/4})$ *time each figure can determine whether or not its convex hull intersects the convex hull of any other figure.* ·

## 6.6 Data Movement Operations

In this section, details of the data movement operations used throughout this chapter are presented.



### *6.6.1 Pyramid Read, Pyramid Write, and Count_keys*

The pyramid read and pyramid write involve master records stored at some level $i$, with request or update records, respectively, generated at some level $j$, $j \leq i$. The description of the pyramid read algorithm, which follows, is somewhat counterintuitive. Instead of request records traveling from level $j$ to level $i$, obtaining their data, and returning to level $j$, the algorithm works by sending the master records from level $i$ down to level $j$. For example, suppose level $i$ is a mesh of size $m$ that contains the master records. At iteration $t$, $1 < t < i - j$, squares of size $m$ are copied from level $i - t + 1$ to level $i - t$ in unit time, where they are decoupled in $\Theta(m^{1/2})$ time so that every square of size $m$ at level $i - t + 1$ creates four squares of size $m$ at level $i - t$. After $t$ iterations, $4^t$ disjoint squares of size $m$ exist at level $j$, each of which is a duplicate of the master records as they are stored at level $i$. In order to obtain the desired information, request records at level $j$ simply perform a mesh concurrent read within their square of size $m$. This algorithm is complete in $\Theta(i - j + 1 + (i - j + 1)m^{1/2})$ time. A pyramid write algorithm that finishes in the same time may be performed similarly. Instead of data moving down the pyramid, data flows up the pyramid from level $j$ to level $i$, where each iteration consists of combining squares of size $m$. When the data arrives at level $i$, a final mesh concurrent write is performed to complete the operation.

Notice that during each iteration of the algorithm, work is only performed at one level of the pyramid. The running times of these algorithms can be improved by incorporating pipelining, so that work is performed concurrently at multiple levels of the pyramid, as follows.

Let $m = n/4^i$ and $S = 4^{\lceil \log_4 \lceil m/(i-j+1) \rceil \rceil}$. Conceptually, level $i$ is a mesh of size $m$, and levels $j \ldots i$ are divided into disjoint squares of size $S$. The squares at level $i$ are numbered from 1 to $m/S$ using a snake-like ordering, as in Figure 1.2. All of the data starting in square $k$ at level $i$ is called *packet k*.

Define a *cycle* to be $cS^{1/2}$ time units, where the constant $c$, independent of $n$ and $S$, is chosen so that in one cycle a square can perform all of the following.

1. Exchange packets with the next square on the same level (where next is with respect to the snake-like ordering).

2. Make a copy of the packet in each of the four descendant squares at the level below.



3. Perform a mesh concurrent read.

A description of an improved pyramid read algorithm follows. Packets are first passed backwards along level $i$ towards square 1, using the snake-like ordering, one square per cycle. Once at square 1, a packet is moved forwards along level $i$, again using the snake-like ordering. Each time that a square at level $i$ receives a packet moving forwards, it first creates a copy of the packet in each of its four descendant squares at level $i$ - 1, before passing it along. Each square at level $(j + 1) \ldots (i - 1)$ that receives a packet, makes a copy of the packet in each of the four squares at the level below. Finally, each time a square at level $j$ receives a packet, it performs a mesh concurrent read so that the processors in the square can read information from the current packet, after which, the packet can be discarded.

**Proposition 6.21** *In a pyramid computer of size $n$, a pyramid read at level $j$ from level $i$ takes* $\Theta(i - j + 1 + [m(i - j + 1)]^{1/2})$ *time, where* $m = n/4^i$

*Proof.* The operation is finished when packet $m/S$ has moved backwards to square 1, forwards to square $m/S$, down to level $j$, and all level $j$ squares beneath square $m/S$ have done a mesh concurrent read. This takes $2m/s\text{-}1 + i - j$ cycles, or $\Theta(i - j + 1 + [m(i - j + 1)]^{1/2})$ time. ·

For the pyramid write, assume that master records are maintained by processors on level $i$ and update records are generated by processors are on level $j$, $j \leq i$, and $m$ and $S$ are defined as above. The pyramid write is basically performed by running the pyramid read in reverse. Slight differences arise because several processors at level $j$ can send records with the same key, but perhaps different data parts, in which case it is necessary to take a minimum (or some other appropriate commutative associate binary function). Also, it is not initially known which packet a given record will end up in.

In general, a square $Z$ will have a packet's worth of data from each square feeding into it (either the four squares below, or, for squares at level $i$, the four squares below and the preceding square in the snake-like ordering). From this, $Z$ has enough to make at least one packet's worth of data. However, since the square it is feeding data to may have some left-over data from the previous cycle, the square it is feeding informs $Z$ as to the number of records required. In one cycle, $Z$ supplies the necessary data and informs each square feeding into it as to how many records need to be replaced. Since it takes one cycle to receive the data,



and one cycle for $Z$ to pass on data after the new data is received, each step of the pyramid write takes two cycles.

Making these minor changes to the pyramid read, the following is obtained.

**Proposition 6.22** *In a pyramid computer of size n, a pyramid write from level j up to level i can be performed in* $\Theta(i\text{-}j+1+[m(i-j+1)]^{1/2})$ *time, where* $m = n/4^i$. •

Section 6.5 makes use of a data movement operation that is closely related to the pyramid write. The operation is the *sparse pyramid write*, and it is used in circumstances where a rigid relationship exists between the amount of data in the base that needs to be sent and the amount of data that needs to be sent from each square of the base.

**Proposition 6.23** *Fix constants p and C, where* $0 < C$ *and* $0 < p < 1$. *Given a pyramid computer of size n, suppose ther*$\left(\frac{n^{1-p}}{C}\right)$*o more than* $Cn^p$ *processors in the base which have a piece of data that is to be sent to level* $\log_4\left(\frac{n^{1-p}}{C}\right)$. *(This level is the highest one with at least* $Cn^p$ *processors.) Further, in each subsquare of size k in the base of the pyramid,* $0 \le k \le n$, *assume that there are no mor*$\left(\frac{n^{1-p}}{C}\right)Ck^p$ *processors sending data. Then a sparse pyramid write will move the data to level* $\log_4\left(\frac{n^{1-p}}{C}\right)$ *in* $\Theta(n^{p/2})$ *time.*

*Proof.* To perform a sparse pyramid write, fix $p$ and $C$, and in parallel perform a sparse pyramid write in each quadrant of the pyramid. The level that the data is written to is either the same as the desired final level, or else it is one level below. Merge the data together using a mesh computer operation such as a concurrent read (see Section 2.6.4), and move up one level if necessary. For fixed $p$ and C, the time obeys a recurrence of the form $T(n) = T(n/4) + dn^{p/2}$, $d$ a constant, which has a solution of $\Theta(n^{p/2})$. ·

The function *count-keys* is responsible for counting the number of distinct keys present in the base of a pyramid of size $n$. If each key were represented only once, then *count_keys* could finish in $\Theta(\log n)$ time. However, keys may be duplicated, so *count-keys* uses the pyramid write to eliminate duplicates. It first tries to determine if the number of unique keys is less than or equal to $K$, where $K = 4^{\lceil \log_4 \log_4 n \rceil}$. This is accomplished by performing a pyramid write of the keys from the base to level $L = \log_4(n/K)$, where each processor at level $L$ acts as if it



is maintaining one master record. When finished, a pyramid read and a tree-like semigroup operation (i. e., an associative binary operation) are used to determine whether or not all keys reached level $L$. Since each processor at level $L$, a mesh of size $K$, permitted only one record to be viewed, then if there are more than $K$ distinct keys in the base, not all keys would have reached level $L$. If it is determined that all keys have reached level $L$, then the number of distinct keys in the base is the number of processors at level $L$ that actually received a record. Otherwise, *count_keys* determines if the number of keys is less than or equal to $4K$ by performing a new pyramid write to level $L \leftarrow L - 1$. The algorithm continues by multiplying the number of keys by 4 at each stage, and decrementing the level $L$ representing the location of the master records, until it reaches a stage where the pyramid write succeeds in moving all distinct keys to level $L$. At this point, since exactly one copy of each key has reached level $L$, the total number of keys can be counted and distributed to all processors in $\Theta(\log n)$ time by performing a semigroup operation.

**Proposition 6.24** *If there are k different keys present in the base of a pyramid computer of size n, then in* $O(\log(n) + k^{1/2}[1 + \log(n/k)]^{1/2})$ *time,* count_keys *will count them.*

### *6.6.2 Pyramid Matrix Read and Pyramid Matrix Write*

The pyramid matrix read and pyramid matrix write both assume that a matrix $M = \{m_{i,j}\}$ is stored in the pyramid so that base processor $P_{i,j}$ stores matrix entry $m_{i,j}$. A pyramid matrix write performs the same basic action as a pyramid write, with the exception that all processors in the same row of the matrix send update records corresponding to the same key. (A column version of the pyramid matrix write can be defined similarly.) Assume that update records for the pyramid matrix write are generated by base processors, and that master records are maintained by processors at level $i$, and let $m = n/4^i$. (Recall from Section 6.3 that $m \leq 2n^{1/2}$.) Also, assume that ties are broken in favor of the minimum record. The pyramid matrix write has two steps, namely

1. move the data to level $j = \log_4 m$, and then

2. move the data from level $j$ to level $i$.

(Note: if $m = 2n^{1/2}$ then set $j = i$ instead of $i + 1$.)



To perform the first step of the row version of the pyramid matrix write (the column version is similar), partition the processors at level $j$ into disjoint *strings* of $k = 2^j$ processors all in the same row, and call such a string and all descendant processors of the string a *prism*. Notice that a prism includes $k^2$ columns of $k$ rows in the base. Therefore, a prism sits over no more than $k$ different keys. In each prism, at time $j$, the first string processor receives the minimum record sent from any of its $k$ base descendants that are in the first row of its prism. This processor passes the record on to the next processor in its string. In general, the computations are pipelined so that at time $j + r - 1 + p - 1$, the $p^{th}$ processor in the string of each prism receives the minimum record sent from any descendant base processor in the $r^{th}$ row of the prism, and also receives from the preceding string processor, the minimum record sent from any base processor in the $r^{th}$ row beneath any of the preceding string processors. The $p^{th}$ processor in the string takes the minimum of these two values and passes it to the $(p + 1)^{st}$ processor in its string.

At time $j + k - 1$, the last processor in each string forms the minimum sent by any base processor in the first row of its prism, and this value is sent back towards the first processor of its string. These reverse messages are passed simultaneously with the previously mentioned ones. Finally, at time $j + 2k - 2$, the last string processor (the $k^{th}$ one) finds the minimum record sent by any base processor in the $k^{th}$ row of the prism. Simultaneously, the minimum record sent by any base processor in the first row of a prism has moved back to the first processor of its string, and the first step of the algorithm is finished.

The second step is just a pyramid write from level $j$ to level $i$. This gives the following result.

**Proposition 6.25** *In a pyramid computer of size $n$, a pyramid matrix write to level $i$, $i \geq \left\lfloor \frac{\log_4 n}{2} \right\rfloor$, takes* $\Theta(\log(n) + m^{1/2}[2 + \log(n/m^2)]^{1/2})$ *time, where* $m = 4^i$.

*Proof.* If $m \leq n^{1/2}$, then the time for the first step is $\Theta(m^{1/2})$, and the time for the second step is $\Theta(i - j + 1 + m^{1/2}[i - j + 1]^{1/2})$. Since $j = \log_4 m$ and $i = \log_4(n/m)$, the result is as claimed. Otherwise, if $m = 2n^{1/2}$, then the time is $\Theta(m^{1/2})$. In this case, $\log_4(n/m^2) = -1$, which is why there is a 2 instead of the usual 1 inside the brackets. ·

The pyramid matrix read performs the same basic action as a pyramid read, and also comes in a row and column version. The discussion will again be for the row version, with the column version being similar.

Assume that all request records are generated by base processors, that those in the same row request information about the same key, and that master records are maintained by processors at level $i$, where $m = n/4^i$. The pyramid matrix read takes 3 steps. The first step uses prisms of height $j$, where $j = \log_4 m$. By using the first step of the pyramid matrix write, in $\Theta(m^{1/2})$ time, the top row (string) of each prism contains the keys needed by the rows beneath. The second step is a pyramid read at level $j$ from level $i$. The third step reverses the first one, taking data to the base.

**Proposition 6.26** *In a pyramid computer of size $n$, a pyramid matrix read from level $i$, $i \geq \left\lfloor \frac{\log_4 n}{2} \right\rfloor$, takes $\Theta(log(n)+m^{1/2}\ [2+\log(n/m^2)]^{1/2})$ time, where $m = 4^i$.* ·

### *6.6.3 Funnel Read and Reducing a Function*

The funnel read was initially discussed in Section 6.4.1. It is useful for the situation where an algorithm leaves intermediate results in data squares scattered throughout the pyramid that must be collated and moved to the base. Its implementation is straightforward. Suppose the final stage of an algorithm is stage $f$. Then every stage $f$ - 1 data square uses a pyramid read to obtain the necessary data from its stage $f$ data square. (Notice that this runs in time proportional to the edgelength of a stage $f$ data square.) Continuing downwards, every stage $i$ - 1 data square uses a pyramid read to obtain its data from its stage $i$ data square. As data moves down the pyramid, the squares get smaller by a factor of 4 at each stage (see Figure 6.3). Therefore, if the final stage of an algorithm produces data in a square of size $m$ at level $k$, the running time of the funnel read is $c\sum_{i=0}^{k-1} \frac{m^{1/2}}{2^i}$, for c a constant, which is $\Theta(m^{1/2})$.

**Proposition 6.27** *Assume that the final stage of an algorithm is stage $f$, and that a stage $f$ data square is a mesh of size $m$. Then a funnel read runs in $\Theta(m^{1/2})$ time.* ·

The operation of reducing a function was initially discussed in Section 6.4.1. Given sets $Q$, $R$, and $S$, let $g$ be a function mapping $Q \times R$ into $S$, and let * be a commutative, associative, binary operator over $S$. Define $f$ to be the *reduction of* $g$, where $f$ maps $Q$ into $S$ by $f(q) = *\{g(q, r) \mid r \in R\}$. For example, if $Q$ and $R$ are sets of points



in some metric space, if $S$ is the real numbers, if $g(q, r)$ is the distance from $q$ to $r$, and if * is minimum, then $f(q)$ is the distance from $q$ to a nearest point in $R$.

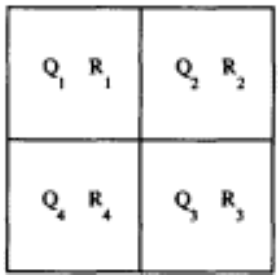
Assume that the elements of $Q$ are stored one per processor in a square, call it $T$, of size $m$ at level $i$, and the elements of $R$ are also stored one per processor in $T$. (A processor of $T$ may contain an element of $Q$ and an element of $R$.) Further, assume $g$ and * can both be computed in $\Theta(1)$ time.

In order to describe the algorithm, some notation is in order. Let $G(Q, R)$ denote the reduction, where for $q \in Q$, $G(Q, R)(q) = *\{g(q, r) \mid r \in R\}$. The term *computing $G(Q, R)$ in $T$*, means that for a set $T$ of processors, for each element $q \in Q$, there is a processor in $T$ that computes and stores the value of $G(Q, R)(q)$. Notice that if a set $A \subseteq Q$ is partitioned into subsets $A_1$, $A_2$, $A_3$, and $A_4$, then $G(A, R) = G(A_1, R) \cup G(A_2, R) \cup G(A_3, R) \cup G(A_4, R)$. Also, if a set $B \subseteq R$ is partitioned into subsets $B_1$, $B_2$, $B_3$, and $B_4$, then $G(Q, B)(q) = G(Q, B_1)(q)* G(Q, B_2)(q) * G(Q, B_3)(q) * G(Q, B_4)(q)$, for any $q \in Q$.

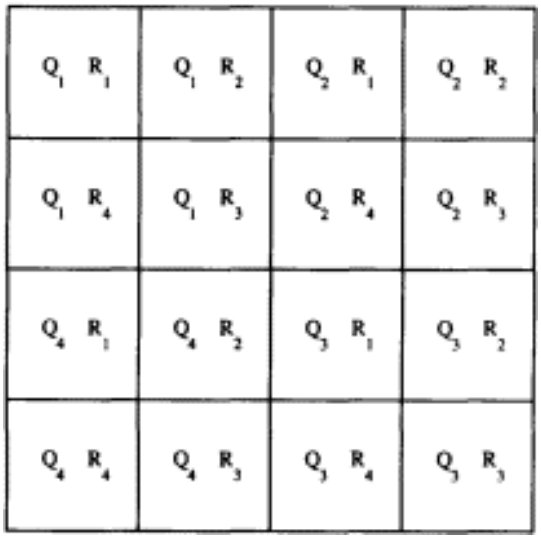
Using these observations, the operation of reducing a function can now be presented in a straightforward fashion. If $m = 1$ (i.e., the number of processors in $T$ is 1), then the single processor just computes the required value in $\Theta(1)$ time. If $i = 0$ (i.e., if all of the data is at the base), then in $\Theta(m)$ time, all values of $R$ are circulated among all processors holding members of $Q$, and each such processor calculate its associated $f$ value. Otherwise, $Q$ is partitioned at level $i$ by quadrants into 4 subsets, namely, $Q_1$, $Q_2$, $Q_3$, and $Q_4$, as in the top of Figure 6.5.

The quadrant storing $Q_j$ assumes responsibility for computing $G(Q_j, R)$. In order to compute $G(Q_j, R)$, each processor that contains an element of $Q_j$ first sends a copy of this element to its four children. A mesh sort-like step is used at level $i$ -1 to create 4 copies of $Q_j$, one in each quadrant of the descendant processors of $Q_j$ from level $i$. Next, the processors at level $i$ - 1 that are descendants of $Q_j$ from level $i$, perform a pyramid read from level $i$ in order to obtain a copy of $R$. (See the bottom of Figure 6.5.) A square of size $m/4$ on level $i$ - 1 holding $Q_j$ and $R_k$ is now responsible for computing $G(Qj, R_k)$, which it does recursively. When this is finished, beneath each quadrant of level $i$, the four squares of size $m/4$ at level $i$ - 1 use a pyramid write to send up their results so that a processor at level $i$ that is responsible for an element $q \in Q$ will receive $G(Q, R_1)(q)$, $G(Q, R_2)(q)$, $G(Q, R_3)(q)$, and $G(Q, R_4)(q)$. By taking the * of these values, the processor responsible for $q$ will compute $G(Q, R)(q)$, and the operation is complete.






level i




level i-1

Figure 6.5:
Reduction of a function.

**Proposition 6.28** *Suppose the elements of Q are stored one per processor in a square of size m at level i, and the elements of R are also stored one per processor in this square, and suppose g and * can be computed in* $\Theta(1)$ *time. Then the reduction of g can be computed in* $\Theta(m^{1/2} + m/4^i)$ *time.*

*Proof.* It takes $\Theta(m)$ time to copy the values of $Q$ and $R$ from level $i$ to level $i - 1$. Since the size of the squares reduces by a factor of 4 at each level, it takes $\Theta(m)$ time to copy the values all the way down to the base. (If $m < n^{1/2}$, then the data does not even reach the base, instead only moving down $\log_4 m$ levels until the problem has been broken into squares of size 1.) The base squares are of size $m/4^i$, so they take $\Theta(m/4^i)$ time to compute their values. Moving data back up the pyramid, and combining results along the way, again takes $\Theta(m)$ time. Hence, the running time is as claimed. ·

The preceding operation may be extended to the following situation. Given sets $P$, $Q$, $R$, and $S$, let g be a function mapping $P \times Q \times R$ into $S$, and let * be a commutative, associative, binary operator over $S$. Define $f$ to be the *extended reduction of g*, where $f$ maps $P$ into $S$ by

$$f(p) = *\{g(p, q, r) \mid q \in Q, r \in R\}.$$

If $g$ and * can be computed in unit time, and if $P$, $Q$, and $R$ are stored one element per processor at a mesh level in the pyramid with $m$ processors, $1 \le m \le n^{1/3}$, then this reduction of $g$ can be computed in $\Theta(m^{1/2})$ time. The algorithm for this reduction is analogous to the one of Proposition 6.28, except that the partitioning and merging of the data now follows the pattern illustrated in Figure 6.6.

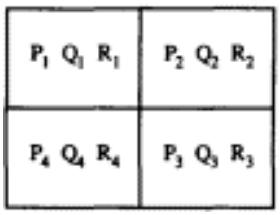
**Proposition 6.29** *Suppose the elements of* $P$, $Q$, and $R$ *are stored one per processor in a square of size* $m$, $1 \le m \le n^{1/3}$, *and suppose * and g can be computed in* $\Theta(1)$ *time. Then the extended reduction of g over* $P \times Q \times R$ *can be computed in* $\Theta(m^{1/2})$ *time.* •

## 6.7 Optimality

This section is concerned with the optimality of the results presented in this chapter.

**Proposition 6.30** *In a pyramid computer of size n, the time needed to move* $B > 1$ *bits of data from the first column of the base to the last column of the base is* $\Omega(\log(n) + [B/\log n]^{1/2})$.






level i

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $P_1 Q_1 R_1$ | $P_1 Q_1 R_2$ | $P_1 Q_2 R_1$ | $P_1 Q_2 R_2$ | $P_2 Q_1 R_1$ | $P_2 Q_1 R_2$ | $P_2 Q_2 R_1$ | $P_2 Q_2 R_2$ |
| $P_1 Q_1 R_4$ | $P_1 Q_1 R_3$ | $P_1 Q_2 R_4$ | $P_1 Q_2 R_3$ | $P_2 Q_1 R_4$ | $P_2 Q_1 R_3$ | $P_2 Q_2 R_4$ | $P_2 Q_2 R_3$ |
| $P_1 Q_4 R_1$ | $P_1 Q_4 R_2$ | $P_1 Q_3 R_1$ | $P_1 Q_3 R_2$ | $P_2 Q_4 R_1$ | $P_2 Q_4 R_2$ | $P_2 Q_3 R_1$ | $P_2 Q_3 R_2$ |
| $P_1 Q_4 R_4$ | $P_1 Q_4 R_3$ | $P_1 Q_3 R_4$ | $P_1 Q_3 R_3$ | $P_2 Q_4 R_4$ | $P_2 Q_4 R_3$ | $P_2 Q_3 R_4$ | $P_2 Q_3 R_3$ |
| $P_4 Q_1 R_1$ | $P_4 Q_1 R_2$ | $P_4 Q_2 R_1$ | $P_4 Q_2 R_2$ | $P_3 Q_1 R_1$ | $P_3 Q_1 R_2$ | $P_3 Q_2 R_1$ | $P_3 Q_2 R_2$ |
| $P_4 Q_1 R_4$ | $P_4 Q_1 R_3$ | $P_4 Q_2 R_4$ | $P_4 Q_2 R_3$ | $P_3 Q_1 R_4$ | $P_3 Q_1 R_3$ | $P_3 Q_2 R_4$ | $P_3 Q_2 R_3$ |
| $P_4 Q_4 R_1$ | $P_4 Q_4 R_2$ | $P_4 Q_3 R_1$ | $P_4 Q_3 R_2$ | $P_3 Q_4 R_1$ | $P_3 Q_4 R_2$ | $P_3 Q_3 R_1$ | $P_3 Q_3 R_2$ |
| $P_4 Q_4 R_4$ | $P_4 Q_4 R_3$ | $P_4 Q_3 R_4$ | $P_4 Q_3 R_3$ | $P_3 Q_4 R_4$ | $P_3 Q_4 R_3$ | $P_3 Q_3 R_4$ | $P_3 Q_3 R_3$ |

level i-2

Figure 6.6:
Extended reduction of a function.



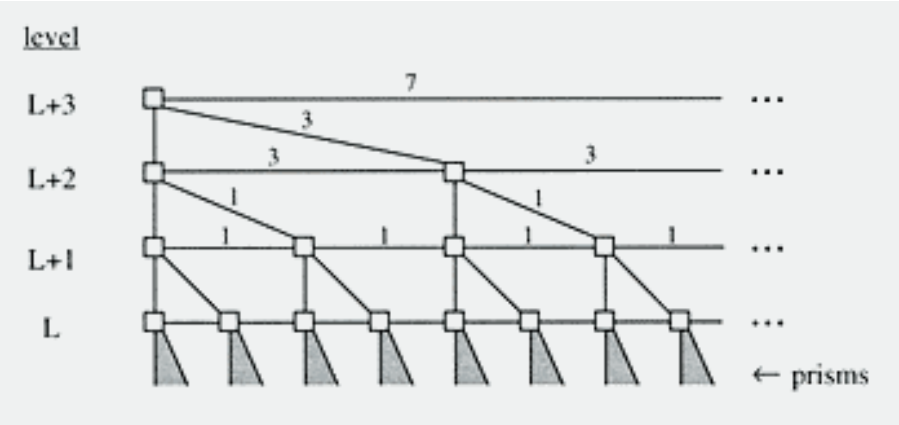



Figure 6.7:

Another view of the pyramid computer.

*Proof*. Assume $B \geq \log_2 n$, and let $L = \left\lfloor \log_4 \left( \frac{n}{B} \log_2 n \right) \right\rfloor$ and $E = \frac{n^{1/2}}{2^L}$. For each column of processors at level $L$, call the entire column and all of its descendants a *prism*. The data initially resides in the leftmost prism and must move to the rightmost one. If a bit only moves along communication links involving processors at level $L$ or below, then at least $E$ - 1 communication links must be traversed, since there are $E$ prisms, and each communication link either keeps the bit in the same prism or moves it to an adjacent one.

Figure 6.7 shows a side view of the processors at level $L$ and above. The usual way of drawing the pyramid has been slightly altered so that all processors in the same column and level are represented as a single processor. The labels along the wires (communication links) indicate the number of steps that could be saved in moving data from the leftmost prism to the rightmost prism by using the wire. The time spent traversing vertical wires is ignored since these wires do not provide any savings. As such, vertically drawn wires are not labeled. Notice that there are $^{E}/_{2}$ ($^{E}/_{2}$ - 1) horizontal wires labeled 1 ,2 ($^{E}/_{4}$)$^{2}$ slanted wires labeled 1, ($^{E}/_{4}$) ($^{E}/_{4}$ - 1) horizontal wires labeled 3, and so forth. Since each wire can carry at most $C\log_2 n$ bits per unit time, for some constant C, in 1 unit of time the maximum number of bits moved by the nonvertical



wires above level $L$ is

$$C\log_2(n)\left[\sum_{i=1}^{\log_4(n)-L}(2^i-1)\left(\frac{E^2}{4^i}-\frac{E}{2^i}\right)+\sum_{i=1}^{\log_4(n)-L-1}2(2^i-1)\frac{E^2}{4^{i+1}}\right],$$

which is less than $CE^2 1\log_2 n$. Therefore, in $t$ units of time the maximum number of bits moved by wires above level $L$, i.e., the maximum savings in moving data from the leftmost prism to the rightmost prism by using wires above level $L$, is less than $CE^2 t\log_2 n$.

Conversely, a bit of data that reaches the rightmost prism in t units of time must have crossed labeled wires above level $L$ with a total weight of at least $E$ - 1 - $t$. Furthermore, if all $B$ bits of data reach the rightmost prism in t units of time, the total savings by using wires (i.e., the cumulative weight of the wires) above level $L$ must be at least $B(E$ - 1 - $t)$. Therefore, t must be such that

$$B(E - 1 - t) < CE^2 t\log_2 n,$$

or

$$t > \frac{B(E-1)}{B+CE^2\log_2 n} = \Theta\left(\left[\frac{B}{\log n}\right]^{1/2}\right).$$

Since the pyramid computer of size $n$ has a communication diameter of $2\log_4 n$, then $t \geq \log_4 n$. Hence, the desired result is obtained. ·

For many of the problems considered in this chapter, inputs can be devised for which Proposition 6.30 applies. For example, consider the problem of labeling the connected components of a digitized picture stored in the base of a pyramid, where the input is of the form shown in Figure 6.8, where an *X* indicates a pixel that may or may not be black and a *Y* indicates a pixel that is always white. Notice that if the two black processors neighboring a processor marked *Y* end up with the same label, then the processor marked *X* that is in *Y*'s row must be black. Since a *Y* can determine if its black neighbors have the same label in $\Theta(1)$ time after the component labeling algorithm is finished, the component labeling algorithm requires at least as much time as it takes to transmit $\Theta(n^{1/2})$ bits from one edge to the opposite edge of the pyramid. By Proposition 6.30, this is $\Omega(n^{1/4}/\log^{1/2} n)$. This lower bound is a factor of $\Theta(\log^{1/2} n)$ smaller than the time achieved in Theorem 6.9, which shows that Theorem 6.9 is at most $\Theta(\log^{1/2} n)$ times optimal. For many of the problems considered in this chapter, Proposition 6.30 can be used to show that the algorithms presented to solve these problems are not



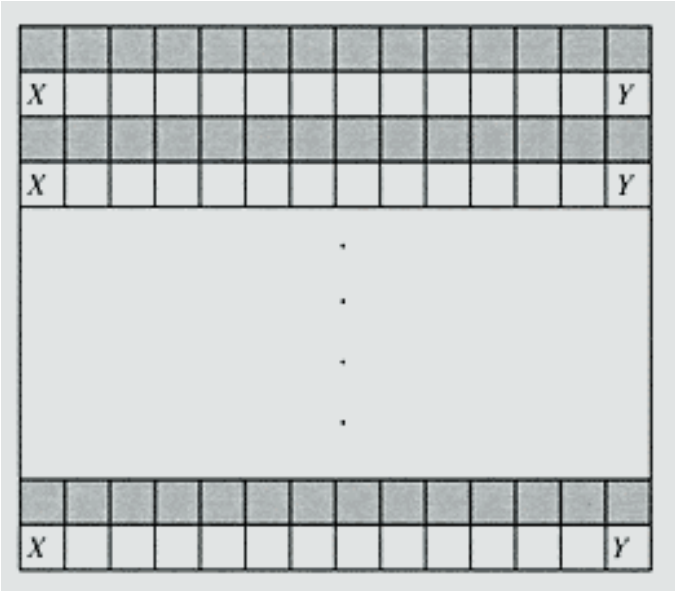



Figure 6.8:
An image requiring extensive data movement.

far from optimal. However, the optimality of the results presented in this chapter remains open.

*Conservative data movement* is defined to be the situation under which data must be moved as separate packets that may not be combined. Note that with respect to the conservative data movement model, several of the algorithms given in this chapter are optimal, including, for example, the digitized picture component labeling algorithm associated with Theorem 6.9.

## 6.8 Further Remarks

Because of its similarity to some animal optic systems, its similarity to the (region) quadtree structure, and its natural use in multiresolution image processing, the pyramid computer has long been suggested for lowlevel image processing [Dyer8la, Dyer8lb, Dyer82, MiSt84c, MiSt85c, Rose84, Stou82c, Stou83c, Tani81, Tani82a, TaK180, Uhr72, Uhr84]. In fact, several pyramid computers are in various stages of construction [Buva87, CFLS85, ClMe87, FKLV83, Scha85, SHBV87, Tani82a]. This chapter demonstrates that the pyramid computer can be used for more complex tasks than originally considered. For example, efficient pyramid computer algorithms were presented in this chapter to solve higher-level



problems in image analysis, as well as problems in graph theory and digital geometry.

In this chapter, fundamental data movement operations for the pyramid computer were presented for a variety of standard input formats. The algorithms that were presented relied heavily on these data movement operations, as well as on fundamental solution strategies, such as divide-and-conquer. Note that these data movement operations intermingle the use of both the child-parent and mesh-connected links. They also make extensive use of intermediate levels of the pyramid to do calculations, store results, and communicate data. Furthermore, the algorithms presented in this chapter show how to exploit lower levels of the pyramid to aid in the computation of functions being performed at higher levels.

In Section 5.3, lower bounds on the running times of algorithms on a pyramid were discussed. The communication diameter of a pyramid of size $n$ gives a lower bound of $\Omega(\log n)$ for problems in which information must be exchanged between arbitrary processors. In Chapter 5, a number of pyramid algorithms were given that have running times that are poly-logarithmic in the size of the input. In this chapter, it was shown that for many problems on a pyramid of size $n$, the $\Omega(\log n)$ bound is overly optimistic, and that lower bounds for these problems are closer to $\Omega(n^{1/4})$. While algorithms that are within a logarithmic factor of this lower bound were presented, the general question of optimality for the problems considered remains open.

In this chapter, the concentration was on 2-dimensional pyramids (i.e., pyramids over 2-dimensional meshes at the base) since they are the ones most commonly built. However, it is interesting to consider higher dimensional pyramids, especially for situations involving higher dimensional data. A *j-dimensional pyramid* (*j-pyramid*) *of size n* is a machine viewed as a full $2^j$-ary tree with additional horizontal links. The base of the $j$-pyramid of size $n$ is a $j$-dimensional mesh of size $n$, as discussed in Section 4.11. Each level of the pyramid is a $j$-dimensional mesh with $1/2^j$ as many processors as the previous level. A processor at level $i$ is connected to its neighbors (assuming they exist): $2j$ adjacent processors at level $i$, $2^j$ children at level $i$ - 1, and a parent at level $i$ + 1. In [MiSt87a], it is shown that several of the data movement operations and algorithms presented in this chapter may be extended to $j$-dimensional pyramids.

It is interesting to compare the pyramid computer to other parallel architectures. Using the standard VLSI model in which processors are separated by at least unit distance and a wire has unit width, it has been

shown that a pyramid computer of size $n$ can be laid out in $\Theta(n)$ area by a simple modification of the standard "H tree" layout scheme [Dyer81a]. The space of a layout for an interconnection scheme is one measure of its cost, as is the regularity of the layout. A mesh computer of size $n$ also requires $\Theta(n)$ area with an extremely regular layout, but because it has a communication diameter of $\Theta(n^{1/2})$, it requires $\Omega(n^{1/2})$ time to solve all of the problems considered in this and the previous chapter, compared to, say, $\Theta(n^{1/4})$ time needed by the pyramid computer to label the figures of an image. (Mesh computer algorithms taking $\Theta(n^{1/2})$ time to solve problems presented in this chapter appear in Chapters 2, 3, and 4.)

Another model that can be easily laid out in $\Theta(n)$ area is the *quadtree machine*, which is simply a pyramid computer without the nearest neighbor (mesh) links. Like the pyramid, the quadtree has a logarithmic communication diameter, but unlike the pyramid, the apex often acts as a bottleneck. For example, it is easy to show that the quadtree needs $\Omega(n^{1/2})$ time to label figures or find nearest neighbors of an image, even if higher processors have additional memory (as suggested in [AhSw84]). On the pyramid, nearest neighbor connections may be used at the intermediate levels to circumvent this bottleneck.

General-purpose interconnection schemes such as the *shuffle-exchange*, *butterfly*, and *cube-connected cycles* can be used to provide poly-logarithmic time solutions to all the problems considered in this and the previous chapter. Unfortunately, these interconnection schemes require area that is nearly proportional to the square of that required to lay out the pyramid computer [Ullm84].

A more interesting model is the *orthogonal trees* or *mesh-of-trees* [Ullm84]. This model has a mesh-connected base of size $n$, augmented so that each row and column of the base mesh has a binary tree over it, with these trees being disjoint except at their leaves. In this model, $\Theta(n^{1/2}1og^2\ n)$ bits can be moved from the leftmost log $n$ columns to the rightmost log $n$ columns in $\Theta(\log n)$ time. This is a significant improvement over the pyramid computer bound presented in Proposition 6.30, though not enough to provide poly-logarithmic time sorting. The mesh-of-trees has not received much consideration as an image processing machine, but for all of the problems considered in this and the previous chapter involving images or adjacency matrices, orthogonal trees can be used to solve them in poly-logarithmic time.

Orthogonal trees do have some drawbacks, however. While the pyramid computer can be laid out in linear area, orthogonal trees need a factor of $\log^2 n$ more area [Ullm84]. Further, orthogonal trees seem to have few ties to other objects of interest for researchers in image pro-



cessing, as opposed to the neural, data structure, and multiresolution ties mentioned above for the pyramid computer. Additional models which are closer to the pyramids, and which solve all of the image processing problems considered herein in poly-logarithmic time, have been suggested [Stou87].

Finally, many of the problems that have been considered for the pyramid have poly-logarithmic time solutions on a *hypercube*. The major disadvantage of a hypercube, however, is that as the number of processors is doubled, each processor of the hypercube is required to add an additional communication link. That is, the number of bidirectional communication links required for each processor of the hypercube is not $\Theta(1)$, as it is for the mesh, pyramid, and mesh-of-trees, but rather $\Theta(\log n)$.

# A
# Order Notation

Intuitively, $\Theta$ is used to mean 'order exactly', O is used to mean 'order at most', $\Omega$ is used to mean 'order at least', $o$ is used to mean 'order less than', and $\omega$ is used to mean 'order greater than'. Let $f$ and $g$ be nonnegative functions defined on the positive integers.

1. The notation $f = \Theta(g)$ (read as "f is theta of g") may be used if and only if there are positive constants $C_1$, $C_2$, and a positive integer $N$ such that $Ci*g(n) \leq f(n) \leq C_2*g(n)$, whenever $n > N$.

2. The notation $f = O(g)$ (read as "f is oh of g") may be used if and only if there is a positive constant $C$ and an integer $N$ such that $0 \leq f(n) \leq C * g(n)$, for all $n > N$.

3. The notation $f = \Omega(g)$ (read as "f is omega of g") may be used if and only if there is a positive constant $C$ and an integer $N$ such that $0 \leq C * g(n) \leq f(n)$, for all $n > N$.

4. The notation $f = o(g)$ (read as "f is little-oh of g") may be used if and only if there is a positive constant $C$ and an integer $N$ such that $0 \leq f(n) < C * g(n)$, for all $n > N$.

5. The notation $f = \omega(g)$ (read as "f is little-omega of g") may be used if and only if there is a positive constant $C$ and an integer $N$ such that $0 \leq C * g(n) < f(n)$, for all $n > N$.

When using order notation, the simplest function possible should be used within the $\Theta$, O, $\Omega$, $o$, or $\omega$. The idea is to use the notation to reduce complicated functions to simpler ones whose behavior is easier to understand. Intuitively, the order notation seeks to capture the dominant term of the function, so as to represent its asymptotic growth rate. The reader should be aware that when using order notation, the symbol '=' should be read as 'is' and not 'equals.' In fact, since order notation is used to describe set membership, it would have been better if the symbol $\in$ were used instead of the symbol =. However, since the symbol '=' has become a defacto standard in the literature, it will be used throughout the book. Using $\Theta$ arbitrarily for an example, it should now be clear that $f = \Theta(g)$ is not the same as $\Theta(g) = f$. In fact, $\Theta(g) = f$ is meaningless.



Notice from the definition of $\Theta$ that if $f = \Theta(g)$, then $g = \Theta(f)$, since if there are positive constants $C_1$,$C_2$, and $N$ such that $C_1 *g(n) \leq f(n) \leq C_2 * g(n)$ for all $n > N$, then $\frac{1}{C_2} * f(n) \leq g(n) \leq \frac{1}{C_1} * f(n)$, for all $n > N$. Some examples of $\Theta$ are $2 + \sin(n) = \Theta(n)$, $3n + 2 = \Theta(n)$, $10n^2 + 4n + 2 = \Theta(n^2)$, $10n^2 + 4n + 2 \neq \Theta(n)$, and $10 * \log(n) + 4 = \Theta(\log n)$.

It is not always possible to determine the behavior of an algorithm using Θ-notation. For example, given a problem with $n$ inputs, it may be that a given algorithm takes $Dn^2$ time when $n$ is even and $Cn$ time when $n$ is odd, or one may only be able to prove that some given algorithm never uses more than $En^2$ time and never less than $Fn\log n$ time (as is the case for the serial version of quicksort). In the first case, one can claim nexther $\Theta(n)$ nor $\Theta(n^2)$ to represent the running time of the algorithm, and in the second case one can claim nexther $\Theta(n)$ nor $\Theta(n\log n)$ to represent the running time of the algorithm. O and Ω notation allow for partial descriptions of functions.

Some examples of O notation are $3n + 2 = O(n)$, since $3n + 2 \le 4n$, for all $n \ge 2$, $100n + 6 = O(n)$, since $100n + 6 < 101n$, for all $n \ge 10$, $10n^2 + 4n + 2 = O(n^2)$, since $10n^2 + 4n + 2 \le 11n^2$, for all $n \ge 5$, $3n+2 \ne O(1)$, since $3n + 2$ is not less than or equal to $c$ for any constant $c$ and all $n \ge N$, $10n^2 + 4n + 2 \ne O(n)$, and $10n^2 + 4n + 2 = O(n^4)$. Of course a more desirable relationship to represent the last function is $10n^2 + 4n + 2 = O(n^2)$, but since O-notation is used to represent an upper bound, $10n^2 + 4n + 2 = O(n^4)$ is also technically correct.

Examples of Ω notation are $3n + 2 = \Omega(n)$, since $3n + 2 \ge 3n$ for all $n >= 1$, $100n + 6 = \Omega(n)$, since $100n + 6 \ge 100n$ for all $n >= 1$, $10n^2 + 4n+2 = \Omega(n^2)$ since $10n2+4n+2 \ge n^2$ for all $n \ge 1$, $10n^2+4n+2 = \Omega(n)$, and $10n^2 + 4n + 2 = \Omega(1)$. Again, the simplest and most accurate representation of the last function would be $10n^2 + 4n + 2 = \Omega(n^2)$.

Examples of $o$ notation are $3n^2 \ne o(n^2)$, since $3n^2 = \Theta(n^2)$, $7n = o(n^2)$, and $7n^{1/2} = o(n^2)$.

Examples of $w$ notation are $3n^2 \ne w(n^2)$, since $3n^2 = \Theta(n^2)$, $7n^3 = w(n^2)$, and $7nr = w(n^3)$

For many of the algorithms in the book, it will be shown that the running time is $O(T(n))$ on a particular machine model, for some function $T(n)$. Further, if it is known that the problem requires $\Omega(T(n))$ time on that model, then it can be concluded that the running time of the algorithm is in fact optimal at $\Theta(T(n))$.



## B
## Recurrence Equations

The running times for many of the algorithms presented in this book involve recurrences of the form

$$T(n) = \begin{cases} d & \text{if } n = 1 \\ aT(n/b) + cn^i & \text{if } n > 1, \end{cases}$$

where $n$ is a power of $b$. The growth rate of $T(n)$, which expresses the asymptotic running time of the algorithm, is given by

$$T(n) = \begin{cases} \Theta(n^i) & \text{if } a < b^i \\ \Theta(n^i \log n) & \text{if } a = b^i \\ \Theta(n^{\log_b a}) & \text{if } a > b^i. \end{cases}$$

Further, for most of the algorithms presented in this book, $i$ takes on the value 0 or 1, which further simplifies the solution to the recurrence.

For example, many of the mesh algorithms in the book operate on $n^2$ pieces of data distributed one piece per processor, and have running times $T(n^2)$ that are expressed as $T(n^2) = T(n^2/4)+cn$, or equivalently as $T(n^2) = T(n^2/4) + \Theta(n)$. From the above solution to the general recurrence, one can see that $i = 1/2$ and $1 = a < b^i = 4^{1/2} = 2$, so the asymptotic running time of such an algorithm is $T(n^2) = \Theta(n)$.



## Bibliography

### A

[APP85] F. Afrati, C. Papadimitriou, and G. Papageorgiou, The complexity of cubical graphs, *Information and Control* **66** (1985), 53-60.

[ACGO88] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap, Parallel computational geometry, *Algorithmica* **3** (1988), 293-327. (A preliminary version appeared in *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, 1985, 468-477.)

[AhSw84] N. Ahuja and S. Swamy, Multiprocessor pyramid architecture for bottom-up image analysis, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **6** (1984), 463-474.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi, An O($n$ log $n$) sorting network, *Proceedings of the 15th ACM Symposium on Theory of Computing*, 1983, 1-9.

[Akl83] S.G. Akl, Optimal parallel algorithms for computing convex hulls and for sorting, *Computing* **33** (1984), 1-11.

[Akl85] S.G. Akl, *Parallel Sorting Algorithms*, Academic Press, Inc., New York, 1985.

[AkLy93] S. G. Akl and K. A. Lyon, *Parallel Computational Geometry*, Prentice-Hall, 1993.

[Amet86] Ametek, Inc., *Ametek System 14 User's Guide*, August, 1986.

[ADMR94] M.J. Atallah, F. Dehne, R. Miller, A. Rau-Chaplin, and J.-J. Tsay, Multisearch techniques for implementing data structures on a mesh-connected computer, *Journal of Parallel and Distributed Computing* **20** (1994), 1-13.

[AtGo86a] M.J. Atallah and M.T. Goodrich, Efficient parallel solutions to some geometric problems, *Journal of Parallel and Distributed Computing* **3** (1986), 492-507.



[AtGo86b] M.J. Atallah and M.T. Goodrich, Parallel algorithms for some functions of two convex polygons, *Proceedings of the 24th Allerton Conference on Communications, Control, and Computation*, 1986, 758-767.

[AtHa85] M.J. Atallah and S.E. Hambrusch, Solving tree problems on a mesh-connected processor array, *Proceedings of the 26th Annual IEEE Symposium on the Foundations of Computer Science*, 1985, 222-231.

[AtKo84] M.J. Atallah and S.R. Kosaraju, Graph problems on a mesh-connected processor array, *Journal of the ACM* **31** (1984), 649-667.

[Avis79] D. Avis, On the complexity of finding the convex hull of a set of points, Tech. Rept. SOCS 79.2, School of Computer Science, McGill University, 1979.

[AwShi83] B. Awerbuch and Y. Shiloach, New connectivity and MSF algorithms for ultracomputer and PRAM, *Proceedings of the 1983 International Conference on Parallel Processing*, 175-179.

**B**

[BBKK68] G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.A. Stokes, The ILLIAC IV computer, *IEEE Transactions on Computers* **17** (1968), 746-757.

[Batc68] K.E. Batcher, Sorting networks and their applications, *Proceedings of the AFIPS Spring Joint Computer Conference* **32**, 1968, 307-314.

[Batc81] K.E. Batcher, Design of a Massively Parallel Processor, *IEEE Transactions on Computers* **29** (1981), 836-840.

[Bene64] V.E. Benes, Optimal rearrangeable multistage connection networks, *Bell System Technical Journal* **43** (1964), 1641-1656.

[Bent80] J.L. Bentley, Multidimensional divide-and-conquer, *Communications of the ACM* **23** (1980), 214-229.

[BeOt79] J.L. Bentley and T.A. Ottman, Algorithms for counting and reporting geometric intersections, *IEEE Transactions on Computers* **28** (1979), 643-647.



[BWY78] J.L. Bentley, B.W. Weide, and A.C. Yao, Optimal expected-time algorithms for closest point problems, *Proceedings of the 16th Allerton Conference on Communication, Control, and Computing*, 1978.

[Beye69] W. T. Beyer, *Recognition of Topological Invariants by Iterative Arrays*, Ph .D. thesis, M.I.T., 1969.

[BCLR92] S.N. Bhatt, F.R.K. Chung, F.T. Leighton, and A.L. Rosenberg, Efficient embeddings of trees in hypercubes, *SIAM Journal on Computing* **21** (1992), 151-162.

[BhIp85] S.N. Bhatt and I.C.F. Ipsen, How to embed trees in hypercubes, Yale University Research Report YALEU/DCS/RR-443, 1985.

[BoRa90] R. Boppana and C.S. Raghavendra, Optimal self-routing of linear-complement permutations in hypercubes, *Proceedings of the 5th Distributed Memory Computing Conference*, 1990, 800-808.

[Buva87] P.J. Burt and G.S. van der Wal, Iconic image analysis with the pyramid vision machine (PVM), *Proceedings of the IEEE 1987 Workshop on Pattern Analysis and Machine Intelligence*, 137-144.

**C**

[CFLM85] V. Cantoni, M. Ferretti, S. Levialdi, and F. Maloberti, A pyramid project using integrated technology, in *Integrated Technology for Parallel Image Processing*, S. Levialdi, ed., Academic Press, 1985, 121-132.

[CFLS85] V. Cantoni, M. Ferretti, S. Levialdi, and R. Stefanelli, Papia: pyramidal architecture for parallel image analysis, *Proceedings of the 7th Symposium on Computer Arithmetic*, 1985, 237-242.

[Chan88] M.Y. Chan, Dilation-2 embeddings of grids into hypercubes, *Proceedings of the 1988 International Conference on Parallel Processing*, vol. III, 295-298.

[Chan91] M.Y. Chan, Embedding of grids into optimal hypercubes, *SIAM Journal on Computing* **20** (1991), 834-864.

[Chaz84] B. Chazelle, Computational geometry on a systolic chip, *IEEE Transactions on Computers* **33** (1984), 774-785.



[CSK90] M.-S. Chen, K.G. Shin, and D.D. Kandlur, Addressing, routing, and broadcasting in hexagonal mesh multiprocessors, *IEEE Transactions on Computers* **39** (1990), 10-18.

[CLC82] F.Y. Chin, J. Lam and I.-N. Chen, Efficient parallel algorithms for some graph problems, *Communications of the ACM* **25** (1982), 659-665.

[Chow81] A. Chow, A parallel algorithm for determining convex hulls of sets of points in two dimensions, *Proceedings of the 19th Allerton Conference on Communication, Control, and Computing*, 1981, 214-233.

[ClMe87] Ph . Clermont and A. Merigot, Real time synchronization in a multi-SIMD massively parallel machine, *Proceedings of the IEEE 1987 Workshop on Pattern Analysis and Machine Intelligence*, 131-136.

[Cohn89] E. Cohn, Implementing the multi-prefix operation efficiently, Stanford Univ., Computer Science Tech. Report, 1989.

[Cole88] R. Cole, Parallel merge sort, *SIAM Journal of Computing* **17** (1988), 770-785.

[CLR92] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, McGraw-Hill Book Company, New York, 1992.

[CyP193] R. Cypher and C.G. Plaxton, Deterministic sorting in nearly logarithmic time on the hypercube and related computers, *Journal of Computer and System Sciences* **47**, 1993, 501-548.

[CySa88] R. Cypher and J. Sanz, Optimal sorting on reduced architectures, *Proceedings of the 1988 International Conference on Parallel Processing*, vol. III, 308-311.

[CySa89] R. Cypher and J. Sanz, Data reduction and fast routing: a strategy for efficient algorithms for message-passing parallel computers, *Algorithmica* **7** (1992), 77-89.

[CSS87] R. Cypher, J. Sanz, and L. Snyder, Hypercube and shuffle-exchange algorithms for image component labeling, *Proceedings of the IEEE 1987 Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence*, 5-9.



[CSS89] R. Cypher, J. Sanz, and L. Snyder, EREW PRAM and mesh connected computer algorithms for image component labeling, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **11** (1989), 258-262.

**D**

[DaLe81] P. E. Danielsson and S. Levialdi, Computer architectures for pictorial information systems, *IEEE Computer* **14** (1981), 53-67.

[Dehn86a] F. Dehne, O(*n*1/2) algorithms for the maximal elements and ECDF searching problem on a mesh-connected parallel computer, *Information Processing Letters* **22** (1986), 303-306.

[Dehn86b] F. Dehne, *Parallel Computational Geometry and Clustering Methods*, Ph .D. thesis, Tech. Rept. SCS-TR-104, School of Computer Science, Carleton University, 1986.

[Dehn87] F. Dehne, *Solving visibility and separability problems on a mesh-of-processors*, Tech. Rept. SCS-TR-123, School of Computer Science, Carleton University, 1987.

[DFR93] F. Dehne, A. Fabri and A. Rau-Chaplin, Scalable Parallel Computational Geometry for Multicomputers, *Proceedings of the 1993 ACM Symposium on Computational Geometry*, 298-307.

[DHSS87] F. Dehne, A. Hassenklover, J.-R. Sack, and N. Santoro, Parallel visibility on a mesh-connected parallel computer, *Proceedings of the International Conference on Parallel Processing and Applications*, 1987, 173-178.

[DHSS91] F. Dehne, A. Hassenklover, J.-R. Sack, and N. Santoro, Computational geometry on a systolic screen, *Algorithmica* **6** (1991), 734-761.

[DoSm84] L. Dorst and W.M. Smeulders, Discrete representation of straight lines, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **6** (1984), 450-463.

[DWR81] T. Dubitzki, A.Y. Wu, and A. Rosenfeld, Parallel region property computation by active quadtree networks, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **3** (1981), 626-633.



[DuWa77] M.L.B. Duff and D.M. Watson, The cellular logic array image processor, *Computer Journal* **20** (1977), 68-72.

[Dyer79] C.R. Dyer, *Augmented Cellular Automata for Image Analysis*, Ph .D. thesis, University of Maryland, 1979.

[Dyer80] C.R. Dyer, Computing the Euler number of an image from its quadtrees, *Computer Graphics and Image Processing* **13** (1980), 270-276.

[Dyer81a] C.R. Dyer, A VLSI pyramid machine for hierarchical parallel image processing, *Proceedings of the IEEE Computer Society Conference on Pattern Recognition and Image Processing*, 1981, 381-386.

[Dyer81b] C.R. Dyer, A Quadtree Machine for Parallel Image Processing, Tech. report KSL 51 (1981), U. of Ill. at Chicago Circle.

[Dyer82] C.R. Dyer, Pyramid algorithms and machines, *Multicomputers and Image Processing Algorithms and Programs*, K. Preston and L. Uhr, eds., Academic Press, New York, 1982, 409-420.

[DyRo81] C.R. Dyer and A. Rosenfeld, Parallel image processing by memory augmented cellular automata, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **3** (1981), 29-41.

**F**

[Fisc80] M. A. Fischler, Fast algorithms for two maximal distance problems with applications to image analysis, *Pattern Recognition* **12** (1980), 35-40.

[FoFu88] G.C. Fox and W. Furmanski, Optimal communication algorithms for regular decompositions on the hypercube, *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications*, 1988, 648-713.

[FrSh75] H. Freeman and R. Shapira, Determining the minimal-area encasing rectangle for an arbitrary closed curve, *Communications of the ACM* **18** (1975), 409-413.

[FKLV83] G. Fritsch, W. Kleinoeder, C.U. Linster, and J. Volkert, EMSY85-The Erlanger multi-processor system for a broad



spectrum of applications, *Proceedings of the 1983 International Conference on Parallel Processing*, 325-330.

**G**

[Gaaf77] M. Gaafar, Convexity verification, block-chords, and digital straight lines, *Computer Graphics and Image Processing* **6** (1977), 361-370.

[Gass69] S.I. Gass, *Linear Programming*, McGraw-Hill, New York, 1969.

[Gent78] W.M. Gentleman, Some complexity results for matrix computations on parallel processors, *Journal of the ACM* **25** (1978), 112-115.

[GeKu81] W.M. Gentleman and H.T. Kung, Matrix triangularization by systolic arrays, *Proceedings of SPIE Vol. 298 Real-Time Signal Processing IV*, 1981, 19-26.

[Gola69] M.J.E. Golay, Hexagonal parallel pattern transformations, *IEEE Transactions on Computers* **18** (1969), 733-740.

[Gord90] J.M. Gordon, Analysis of minimal path routing schemes in the presence of faults, *Discrete Applied Mathematics*, 1990.

[Gray71] S. B. Gray, Local properties of binary images in two dimensions, *IEEE Transactions on Computers* **20** (1971), 551-561.

[GVJK] F.C.A. Groen, P.W. Verbeek, N.d. Jong, and J.W. Klumper, The smallest box around a package, Tech. Rept., Institute of Applied Physics, Delft University of Technology.

[GHS86] J.L. Gustafson, S. Hawkinson, and K. Scott, The architecture of a homogeneous vector supercomputer, *Proceedings of the 1986 International Conference on Parallel Processing*, 649-652.

**H**

[Habe72] A.N. Habermann, Parallel neighbor sort, Tech. Rept., Carnegie-Mellon University, 1972.

[Hamb83] S.E. Hambrusch, VLSI algorithms for the connected component problem, *SIAM Journal on Computing* **12** (1983), 354-365.



[HaSi81] S.E. Hambrusch and J. Simon, Solving undirected graph problems on VLSI, Tech. rep. CS-81-23, Computer Science, Penn. State Univ., 1981.

[HaLi72] I. Havel and P. Liebl, Embedding the dichotomic tree into the *n*-cube, *Casopis Pro Pestorani Matematiky* **97** (1972), 201-205.

[HaLi73] I. Havel and P. Liebl, Embedding the polytomic tree into the *n*-cube, *Casopis Pro Pestorani Matematiky* **98** (1973), 307-314.

[HaMo72] I. Havel and J. Moravek, B-valuations of graphs, *Czechoslovak Mathematics Journal* **22** (1972), 338-351.

[HMSC86] J. Hayes, T.N. Mudge, Q.F. Stout, S. Colley, and J. Palmer, A microprocessor-based hypercube, *IEEE Micro* **6** (1986), 6-17.

[Hill85] D. Hillis, *The Connection Machine*, The MIT Press, Cambridge, Mass., 1985.

[HCW79] D.S. Hirschberg, A.K. Chandra and D.V. Sarwate, Computing connected components on parallel computers, *Communications of the ACM* **22** (1979), 461-464.

[HoJo86] C.-T. Ho and S.L. Johnsson, Distributed routing algorithms for broadcasting and personalized communication in hypercubes, *Proceedings of the 1986 International Conference on Parallel Processing*, 640-648.

[HoJo88] C.-T. Ho and S.L. Johnsson, Optimal algorithms for stable dimension permutations on Boolean cubes, *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications*, 1988, 725-736.

[Huan85] M.D.A. Huang, Solving some graph problems with optimal or near-optimal speedup on mesh-of-trees networks, *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, 1985, 232-240.

[HKW82] A. Hubler, R. Klette, and G. Werner, Shortest path algorithms for graphs of restricted in-degree and out-degree, *Elektronische Informationsverarbeitung und Kybernetik (Journal of Information Processing and Cybernetics - EIK)* **18** (1982), 141-151.



[HwFu82] K. Hwang and K-S. Fu, Integrated computer architectures for image processing and database management, *IEEE Computer* **15** (1982), 51-60.

[HyMu86] *Hypercube Multiprocessors 1986*, M.T. Heath, ed., SIAM, 1986.

[HyMu87] *Hypercube Multiprocessors 1987*, M.T. Heath, ed., SIAM, 1987.

**I**

[Inte86] Intel Corporation, *iPSC System Overview*, January 1986.

**J**

[JaJa92] J. Ja'Ja', *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA., 1992.

[JGD87] L.H. Jamieson, D.B. Gannon, and R.J. Douglas, eds., *The Characteristics of Parallel Algorithms*, The MIT Press, Cambridge, Mass., 1987.

[JeLe90] C.S. Jeong and D.T. Lee, Parallel geometric algorithms on a mesh connected computer, *Algorithmica* **5** (1990), 155-178.

[John84] S.L. Johnsson, Combining parallel and sequential sorting on a Boolean *n*-cube, *Proceedings of the 1984 International Conference on Parallel Processing*, 444-448.

**K**

[KaRa90] R.M. Karp and V. Ramachandran, A survey of parallel algorithms for shared-memory machines, *Handbook of Theoretical Computer Science*, North-Holland, 1990, 869-941.

[Kim81] C.E. Kim, On the cellular convexity of complexes, *IEEE Transactions on Pattern Analysis and Machine Intelligence* 3 (1981), 617-625.

[Kim82a] C.E. Kim, Digital convexity, straightness, and convex polygons, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **4** (1982), 618-626.

[Kim83] C.E. Kim, Three-dimensional digital line segments, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **5** (1983), 231-234.

[KiRo82b] C. E. Kim and A. Rosenfeld, Digital straight lines and convexity of digital regions, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **4** (1982), 149-153.

*To Brian, for making it all worthwhile.*
*Russ Miller*

*To my teachers in the public schools of Euclid, Ohio, for encouraging play that adults call research.*
*Quentin F. Stout*