Spring 1993

Time-Optimal Algorithms on Meshes With Multiple Broadcasting

Dharmavani Bhagavathi
Old Dominion University

Time-Optimal Algorithms on Meshes with Multiple Broadcasting

by

Dharmavani Bhagavathi
B.E. July 1987, Osmania University, India
M. Tech. June 1989, University of Hyderabad, India

DISSERTATION

submitted to the faculty of Old Dominion University in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
August 1993

APPROVED BY
SUPERVISORY COMMITTEE

Time-Optimal Algorithms on Meshes with Multiple Broadcasting

Dharmavani Bhagavathi
Old Dominion University, 1993

Advisors : Profs. Chester E. Grosch and Stephan Olariu

The mesh-connected computer architecture has emerged as a natural choice for solving a large number of computational tasks in image processing, computational geometry, and computer vision. However, due to its large communication diameter, the mesh tends to be slow when it comes to handling data transfer operations over long distances. In an attempt to overcome this problem, mesh-connected computers have recently been augmented by the addition of various types of bus systems. One such system known as the mesh with multiple broadcasting involves enhancing the mesh architecture by the addition of row and column buses. The mesh with multiple broadcasting has proven to be feasible to implement in VLSI, and is used in the DAP family of computers. In recent years, efficient algorithms to solve a number of computational problems on meshes with multiple broadcasting have been proposed in the literature.

The problems considered in this thesis are semigroup computations, sorting, multiple search, various convexity-related problems, and some tree problems. Based on the size of the input data for the problem under consideration, existing results can be broadly classified into *sparse* and *dense*. Specifically, for a given $\sqrt{n} \times \sqrt{n}$ mesh with multiple broadcasting, we refer to problems involving $m \in O(\sqrt{n})$ items as sparse, while the case $m \in O(n)$ will be referred to as dense. Finally, the case corresponding to $2 \leq m \leq n$ is be termed general. The motivation behind the current work is twofold. First, time-optimal solutions are proposed for the problems listed above. Secondly, an attempt is made to remove the artificial limitation of problems studied to sparse and dense cases.

To establish the time-optimality of the algorithms presented in this work, we use some existing lower bound techniques along with new ones that we develop. We solve the semigroup computation problem for the general case and present a novel lower bound argument. We solve the multiple search problem in the general case and present some surprising applications to computational geometry. In the case of sorting, the general case is defined to be slightly different. For the specified range of the size of input, we present a time and VLSI-optimal algorithm. We also present time lower bound results and matching algorithms for a number of convexity related and tree problems in the sparse case.

Acknowledgements

First of all, I would like to thank my family and Dr. A. K. Pujari for encouraging me to take up doctoral studies. Special thanks to Gourinath for his love and support throughout my work. I also thank Ramesh and many other friends for taking the place of my family and for being by my side throughout my doctoral program. I would like to thank Prof. Larry Wilson and Prof. James L. Schwing for giving me encouragement and support. I thank Prof. Chester E. Grosch for his advice and encouragement. I thank our chairman, Prof. Kurt Maly for providing the facilities and an environment conducive to high quality research. Most of all, I owe it all to Prof. Stephan Olariu for making the hard work worthwhile and for being my friend and guide all through my stay at Old Dominion University.

Contents

1 Introduction **1**

1.1 Introduction 1

1.2 State of the Art 2

2 Basics **6**

2.1 Model of Computation 6

2.2 Motivation 7

2.3 Lower Bounds 8

2.4 Data Movement 10

3 Semigroup Computations **15**

3.1 Introduction 15

3.2 The Lower Bound 16

3.3 The Algorithm 19

3.4 Conclusions 22

4 Sorting **25**

4.1 Introduction 25

4.2 The Lower Bound 26

4.3 The Algorithm 27

4.4 Conclusions 32

5 Multiple Search **33**

5.1 Introduction 33

5.2 The Algorithm - Preliminaries 34

5.3 The Algorithm - Details 36

5.4 Applications 41

5.5 Conclusions 45

6 Convexity Problems **47**

6.1 Introduction 47

6.2 Preliminaries 48
6.3 Lower Bounds 52
6.4 Algorithms Involving one Convex Polygon 58
6.5 Computations on Two Convex Polygons 64
6.6 Conclusions 67

7 Tree Problems 69
7.1 Introduction 69
7.2 Lower Bounds 70
7.3 Time-Optimal Parentheses Algorithms 75
7.4 Encoding and Decoding Trees 77
7.5 Reconstruction of Trees from Traversals 79
7.6 Conclusions 84

8 Conclusions 85
8.1 Concluding Remarks 85
8.2 Open Problems 86

List of Figures

1.1 A regular 4×4 mesh and various enhanced meshes 5

2.1 Illustrating the data movement of Lemma 2.4.9 13

3.1 Step 0 of semigroup computations. 20
3.2 Step 1 of semigroup computations. 21
3.3 Step 2 in the first submesh $\mathcal{R}_1$ for semigroup computations. 22
3.4 Final data movements in Step 3 of semigroup computations. 23

4.1 Lower bound argument for sorting 26
4.2 Diagonal submeshes and leaders in stage i of sorting 28

5.1 Stage 0 of multiple search 35
5.2 Lower bound argument for multiple search 40
5.3 Stage 1 of the multiple ray problem 43
5.4 Stage 2 of the multiple ray problem 44

6.1 An instance of the point overlap problem 51
6.2 Illustrating antipodal pairs 59

7.1 A binary tree and its encoding 72
7.2 An ordered tree and its encoding 74
7.3 Example of binary tree reconstruction 81
7.4 Example of ordered tree reconstruction 83

Chapter 1

Introduction

1.1 Introduction

Recent advances in VLSI have made it possible to build parallel machines featuring tens of thousands of processors [63]. Yet, practice indicates that this increase in raw computational power does not, as a rule, translate into increased performance of the same order of magnitude. The reason seems to be twofold: first, not all problems are known to admit efficient parallel solutions; second, parallel computation requires interprocessor communications and simultaneous memory accesses which often act as bottlenecks in present-day parallel machines.

The mesh-connected computer architecture has emerged as one of the most natural choices for solving a large number of computational tasks in image processing, computational geometry, and computer vision [3, 7, 11, 38, 42, 45, 60]. Its regular structure and simple interconnection topology makes the mesh particularly well suited for VLSI implementation [63]. However, the mesh tends to be slow when it comes to handling data transfer operations over long distances. In many problems, the worst case running time of the solution is constrained by the communication diameter. As the running times of most algorithms on a mesh connected computer are constrained by data movement considerations, an attempt was made to design an architecture which retained the natural mesh configuration but was augmented with a a faster mechanism for data movement. Gentleman [29] was the first to talk about broadcasting in this context. In broadcasting, a processor sends a data item on a bus taking O(1) time, but the restriction is that only one processor can broadcast on a bus at one time. All the processors connected by the same bus, instantaneously receive the data item that is broadcast on the bus. Jordan and Sawyer [34] took this approach in designing their mesh connected computer with broadcasting. Broadcasting at infinite speed is physically unrealistic, but for practical situations, this is a reasonable working hypothesis. Though broadcasting speeds up the computation of a single subproblem it prohibits the simultaneous computation of subproblems in many cases. In an attempt to overcome this problem, mesh-connected computers have recently been augmented by the addition of different types of bus systems [2, 18, 37, 43]. For example, Aggarwal [2] and Bokhari [18] have considered mesh-connected machines enhanced by the addition of k global buses. Stout [62] has considered a mesh connected machine with only row buses. In this model, the processors in the mesh retain their nearest-neighbor local links, and processors belonging to each row are connected to a bus.

Yet another such system that is commercially available [55] involves enhancing the mesh architecture by the addition of row and column buses. In [37] an abstraction of such a system is referred to as mesh with multiple broadcasting. The mesh with multiple broadcasting has proven to be feasible to implement in VLSI, and is used in the DAP family of computers [55]. The DAP machine is an array of bit-organized processing elements, or PE's, each with its own bit-oriented memory. A PE has connections to each of its nearest neighbors and to a bus system which interconnects the PE's by row and column. It is these row and column busses which allow fast data fetching and broadcasting.

1.2 State of the Art

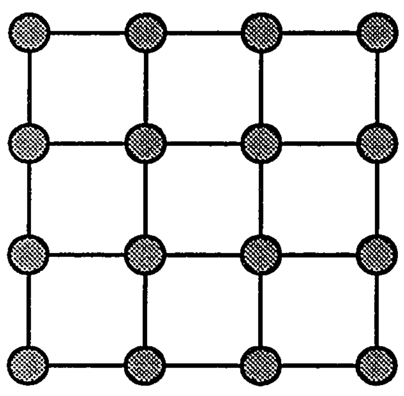
As mentioned earlier, most algorithms on regular mesh-connected computers are restricted in running time by the communication diameter of the mesh. Specifically, for an $n \times n$ mesh, many algorithms exhibit a running time of $\Theta(n)$. In [61], Stout has shown that semigroup computations and selection can be done faster when the meshes are enhanced with a single bus connecting all processors. Specifically, for a 2-dimensional mesh connected computer of size $\sqrt{n} \times \sqrt{n}$, the running time for semigroups operations of n data items is shown to be $O(n^{\frac{1}{3}})$. Later, problems like finding the OR of n given bits stored one per processor, in an $\sqrt{n} \times \sqrt{n}$ mesh with a global bus has been shown to exhibit in $O(n^{\frac{1}{3}})$ time. In [62] it was shown that the problem of finding the leftmost one in each row of a mesh with row buses, with each processor holding a one or a zero, can be determined in $O(n^{\frac{1}{6}})$ time for an $\sqrt{n} \times \sqrt{n}$ mesh. However, in [62], each processor required unbounded memory. In [30], the problem was solved using processors with a constant number of registers.

Being of theoretical interest as well as commercially available, the mesh with multiple broadcasting has attracted a great deal of well-deserved attention. In recent years, efficient algorithms to solve a number of computational problems on meshes with multiple broadcasting have been proposed in the literature. These include image processing [55, 56], computational geometry [14, 15, 37, 49], semigroup computations [2, 8, 18, 21, 37], selection [13, 21, 37], among others. In [56], Kumar and Raghavendra have proposed solutions to a number of problems including, semigroup computations, the median problem, convex hull of an image, and the nearest-one problem. Their solutions were designed for n items stored one per processor in a $\sqrt{n} \times \sqrt{n}$ mesh with multiple broadcasting. The algorithms for semigroup computations, convex hull of an image and the nearest-one problem for n bits run in $O(n^{\frac{1}{6}})$ time, whereas their median finding algorithm takes $O(n^{\frac{1}{6}}(\log n)^{\frac{2}{3}})$ time. The model of computation adopted throughout [37] was a square mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. Later, it has been shown that rectangular meshes are better for problems such as selection, semigroup computations, to name a few [8, 13, 21]. In the above cases, the size of the input data is n, and the number of processors in the square or the rectangular mesh also equal n. Bar-Noy and Peleg [8], and Chen *et al.* [21], have shown independently that semigroup computations can be faster on suitably chosen rectangular meshes, whose dimensions depend on the input size n. Specifically, they have shown that semigroup computations take $O(n^{\frac{1}{8}})$ time on a rectangular mesh with multiple broadcasting of size $n^{\frac{5}{8}} \times n^{\frac{3}{8}}$. In addition to this Chen *et al.* [21], have also proposed an algorithm for prefix semigroup computations for data given in blocked-row-major order, that runs in $O(n^{\frac{1}{8}})$ time on a rectangular mesh with multiple broadcasting of size $n^{\frac{5}{8}} \times n^{\frac{3}{8}}$. Chen *et al.* [22] have also shown that selection can be done faster on a rectangular mesh. In fact, they proposed an algorithm that runs in $O(n^{\frac{1}{8}} \log n)$ time on a mesh with multiple broadcasting of size $n^{\frac{5}{8}} \times n^{\frac{3}{8}}$. Recently, Bhagavathi *et al.* [13] have improved on the running time and presented a selection algorithm that runs in $O(n^{\frac{1}{8}}(\log n)^{\frac{3}{4}})$ on a mesh with multiple broadcasting of size $n^{\frac{5}{8}}/(\log n)^{\frac{1}{4}} \times n^{\frac{3}{8}}(\log n)^{\frac{1}{4}}$. In [56], Kumar and Raghavendra have proposed solutions to some image processing problems. Specifically, they have presented $O(n^{\frac{1}{4}})$ time algorithms for finding connected components and component labeling. Olariu *et al.* [51] have proposed a $O(n^{\frac{1}{6}}(\log n)^{\frac{2}{3}})$ time algorithm for finding the convex hull of points in the plane sorted by x-coordinate. Later, Bhagavathi *et al.* [17], have shown that in this case also, rectangular meshes are better. They presented an algorithm that runs in $O(n^{\frac{1}{8}}(\log n)^{\frac{3}{4}})$ on a mesh with multiple broadcasting of size $n^{\frac{5}{8}}/(\log n)^{\frac{1}{4}} \times n^{\frac{3}{8}}(\log n)^{\frac{1}{4}}$. For all the problems with an exception of semigroup computations, no known lower bounds exist until now. A part of our work in this volume involves proving lower bounds to establish time-optimality of our algorithms.

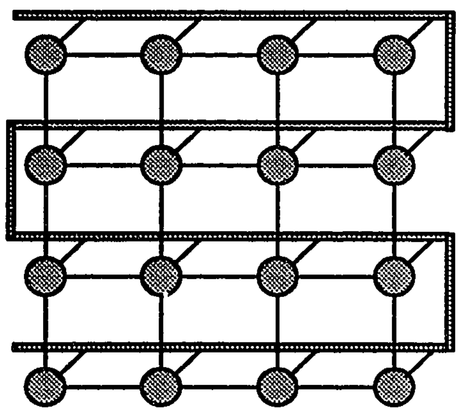
Kumar and Raghavendra [37] have also considered problems involving $\sqrt{n}$ data items in one column of a $\sqrt{n} \times \sqrt{n}$ mesh with multiple broadcasting. They have shown that $\sqrt{n}$ elements given in one row of a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$ can be sorted in $O(\log^2 n)$ time. Quite recently, Lin *et al.* [40] have solved a number of problems including sorting, semigroup computations, convex hull of points in the plane, involving $\sqrt{n}$ data items in one column of a $\sqrt{n} \times \sqrt{n}$ mesh with multiple broadcasting presenting $O(\log n)$ time algorithms. They have also exhibited lower bounds for their algorithms, proving their time-optimality.

The fundamental purpose of this thesis is to propose time-optimal solutions to a number of problems on meshes with multiple broadcasting. In addition to using the existing techniques for proving lower bounds, we develop some novel techniques to establish the time-optimality of our algorithms. To put existing results in perspective, we shall refer to problems involving $m \in O(\sqrt{n})$ data items as *sparse*, while the case $m \in O(n)$ will be referred to as *dense*. Finally, the case corresponding to $2 \leq m \leq n$ will be termed *general*. In this terminology, up to now, lower bounds and matching algorithms for various problems have been obtained only for the sparse and dense cases, but nothing is known about the general case. A part of the present work was motivated by an attempt to remove the artificial limitation of the problem to the sparse and dense cases.

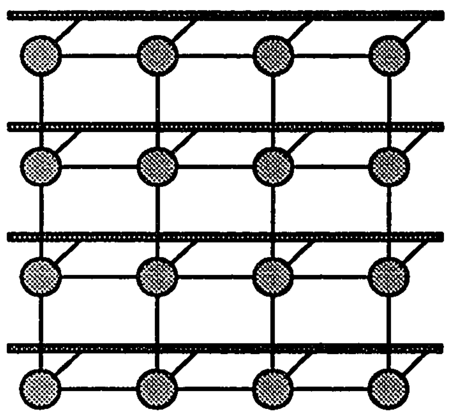
We present time-optimal algorithms for semigroup computations in general case, sorting in general case, multiple search where initial data is dense and the queries are of general case, convexity related problems in sparse case and some tree problems in sparse case. The remainder of the work is organized as follows: Chapter 2 presents the model of computations in more detail and some important fundamental concepts that play a key role in the remaining chapters; Chapter 3 discusses the semigroup computation problem and presents a time-optimal algorithm; Chapter 4 contains a discussion of the time-optimal sorting algorithm; Chapter 5 presents the multiple search algorithm; Chapter 6 involves the time-optimal algorithms for convexity related problems; Chapter 7 involves the tree-problems and the time-optimal algorithms for the same and finally, chapter 8 gives a discussion of further directions of research and poses some open problems.



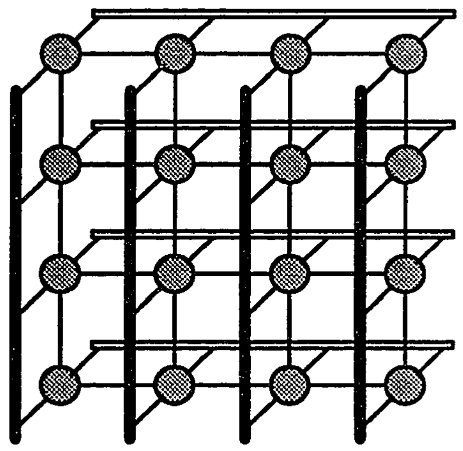
(ii) A regular 4 x 4 mesh



(ii) A 4 x 4 mesh with global bus



(iii) A 4 x 4 mesh with row buses



(iv) A 4 x 4 mesh with row and column buses

Figure 1.1: A regular 4 $\times$ 4 mesh and various enhanced meshes

Chapter 2

Basics

2.1 Model of Computation

Throughout this work our model of computation is a mesh with multiple broadcasting (MMB for short). We shall present the details of the model and discuss some basic assumptions made regarding the working of the model. In general, a MMB of size $M \times N$ consists of MN identical processors positioned on a rectangular array overlaid with a bus system. In every row of the mesh the processors are connected to a horizontal bus; similarly, in every column the processors are connected to a vertical bus (refer to Figure 1.1).

The processor $P(i,j)$ is located in row i and column j $(1 \leq i \leq M; 1 \leq j \leq N)$ with $P(1,1)$ in the north-west corner of the mesh. Every processor $P(i,j)$ is connected by local links to its four neighbors, $P(i-1,j)$, $P(i+1,j)$, $P(i,j-1)$, and $P(i,j+1)$, provided they exist. Throughout this work we assume that the MMB operates in SIMD mode: in each time unit, the same instruction is broadcast to all processors, which execute it and wait for the next instruction. Each processor is assumed to know its own coordinates within the mesh and to have a constant number of registers of size $O(\log MN)$; in unit time, every processor performs some arithmetic or boolean operation, communicates with one of its neighbors using a local link, broadcasts a value on a bus or reads a value from a specified bus. These operations involve handling at most $O(\log MN)$ bits of information. For practical reasons, only one processor is allowed to broadcast on a given bus at any one time. By contrast, all the processors on the bus can simultaneously read the value being broadcast. In accord with other researchers [8, 18, 21, 37, 43, 55, 59], we assume that communications along buses take $O(1)$ time. Although inexact, recent experiments with the DAP [55] and the YUPPIE multiprocessor array system [43], seem to indicate that this first approximation is a reasonable working hypothesis. In further chapters, for problems involving sparse data, we assume that the input consists of n data items in the first row or column of an $n \times n$ MMB. For problems involving data in dense and general cases, our model of computation is an $\sqrt{n} \times \sqrt{n}$ mesh with multiple broadcasting. In the case of dense data, the mesh is completely filled with data and in the general case, the data is given in the leftmost columns.

2.2 Motivation

We consider a number of fundamental problems on the model described above and present time-optimal algorithms. In many cases, we also present relevant applications.

Semigroup computations are a fundamental algorithmic tool, with numerous applications to all branches of parallel processing [3, 38]. The semigroup computation problem has been well studied in the literature. Aggarwal [2] and Bokhari [18] have studied the problem in the context of mesh-connected computers augmented by a global bus. Kumar and Raghavendra [37] showed that on a MMB of size $\sqrt{n} \times \sqrt{n}$ the semigroup computation problem with $m = n$ has a lower bound of $\Omega(n^{\frac{1}{6}})$. At the same time, they exhibited an $O(n^{\frac{1}{6}})$ and therefore optimal algorithm for semigroup computation. Later, Chen *et al.* [21] and Bar-Noy and Peleg [8] have shown that semigroup computations can be performed faster on a *rectangular* MMB. Specifically, they showed that the semigroup computation of n items can be solved in $O(n^{\frac{1}{8}})$ time on a mesh of size $n^{\frac{3}{8}} \times n^{\frac{5}{8}}$. Chen *et al.* [21] also showed that if every processor can hold $n^{\frac{1}{9}}$ items from the input sequence, then the computation can be performed in $O(n^{\frac{1}{9}})$ time on a mesh of size $n^{\frac{1}{3}} \times n^{\frac{5}{9}}$. In Chapter 3, we propose a unifying look at semigroup computations on MMBs.

Sorting is unquestionably one of the most extensively investigated topics in computer science. It is well known [3, 38] that n data items can be sorted in $O(\sqrt{n})$ time on a mesh-connected machine of size $\sqrt{n} \times \sqrt{n}$. Furthermore, this result is easily shown to be both time-optimal and VLSI-optimal. Recently, a number of sorting algorithms have been proposed for enhanced meshes [56, 40]. An easy information transfer argument shows that for meshes of area n, even when enhanced with multiple broadcasting or with a dynamically reconfigurable bus system, $\Omega(\sqrt{n})$ is a time lower bound for sorting n items. This somewhat counter-intuitive result motivated us to look at the following problem. Suppose that we are given a mesh or enhanced mesh of size $\sqrt{n} \times \sqrt{n}$ and the goal is to sort m ($\sqrt{n} \leq m \leq n$) items stored in the first $\lceil \frac{m}{\sqrt{n}} \rceil$ columns of the machine. How fast can this task be performed? It is easy to show that $\Omega(\sqrt{n})$ is a time lower bound for the regular mesh. Clearly, no algorithm can be VLSI-optimal in this case. One of our contributions is to show that we can do better on MMBs. Specifically, we show that once we fix a positive integer constant c, we can sort m items ($n^{\frac{1}{2}+\frac{1}{2c}} \leq m \leq n$) in $O(\frac{m}{\sqrt{n}})$ time. We show that this is time-optimal for this architecture. It will also be shown that this achieves the VLSI lower bound in the word model.

The multiple search problem can be considered to be a fundamental algorithmic problem [4, 72] and finds applications to query processing in databases, computer graphics, image processing, and computational geometry, to name just a few. Recently, Akl and Meijer [4] as well as Wen [72] have studied the multiple search problem in the PRAM model of computation. To the best of our knowledge, this important problem has found no solution in the context of MMBs. In this thesis, we propose a time-optimal algorithm for the multiple search problem on enhanced meshes and show that a number of problems in computer graphics, image processing, robotics, and computational geometry reduce to the multiple search problem or a variant thereof.

Convexity and related computations are a recurring theme in pattern recognition, image processing, computer vision, operations research, robotics, computational geometry, and computational morphology. In pattern recognition, for instance, convexity appears in clustering, and computing similarities between sets [7]. In image processing and computer vision, convexity is used as a natural shape descriptor and classifier for objects in the image space [2,20,22]. In operations research, convexity is a fundamental tool in linear programming and convexity analysis [66]. In robot navigation, one of the fundamental heuristics involves approximating real-world objects by convex sets [41]. In computational geometry, convexity is often a valuable tool in devising efficient algorithms for a number of seemingly unrelated problems [57, 66]. In computational morphology, convexity has played a central role in analyzing relevant features of the shape of a set of points [68]. Further, one of the fundamental features that contributes to a morphological description useful in shape analysis is the Euclidian distance function among vertices of the polygon [68]. In this work we present simple time-optimal algorithms for a number of convexity-related problems on MMBs.

Various applications, including integrated circuit design and automated theorem proving, require efficient handling of binary and ordered trees – encoding, decoding, and reconstruction of trees from their traversals being of particular interest. It is therefore not surprising that these problems have been studied extensively on various architectures. Our contribution is to provide time-optimal algorithms for these problems on the MMB.

2.3 Lower Bounds

We consider problems in the sparse, dense as well as general case. We establish the time-optimality of each problem with the help of various lower bound arguments.

First of all, we would like to present an argument that helps us import lower

bounds from other models of computation. For the same purpose, we state a fundamental result of Cook *et al.* [24] asserting that the time lower bound for the OR problem on the CREW-PRAM is $\Omega(\log n)$ regardless of the number of processors used. For the sake of completeness, we define the problem and state the relevant result from [24].
OR: given n bits $b_1, b_2, \ldots, b_n$, compute their logical OR.

Proposition 2.3.1 The time lower bound for computing the OR of n bits on the CREW-PRAM is $\Omega(\log n)$, independent of the number of processors and memory cells used. □

In addition, we shall rely on the following recent result of Lin *et al.* [40].

Proposition 2.3.2 Any computation that takes $O(t(n))$ computational steps on an n-processor mesh with multiple broadcasting can be performed in $O(t(n))$ computational steps on an n-processor CREW-PRAM with $O(n)$ extra memory. □

It is important to note that Proposition 2.3.2 guarantees that if $T_M(n)$ is the execution time of an algorithm for solving a given problem on an n-processor MMB, then there exists a CREW-PRAM algorithm to solve the same problem in $T_P(n) = T_M(n)$ time using n processors and $O(n)$ extra memory. In other words, "too fast" an algorithm on the mesh with multiple broadcasting implies "too fast" an algorithm for the CREW-PRAM. This observation is exploited in [40] to transfer known computational lower bounds for the PRAM to the MMB.

In some cases, the enhanced mesh does not perform any better than the regular mesh. Since each bus can broadcast only one item at a time, whenever a large amount of data has to be moved around in the mesh, the algorithm tends to be slow due to data movement. A well-established argument known as the data transfer argument is as follows. Consider a submesh S connected by $O(\sqrt{m})$ buses and local links to the remainder of the original mesh. It is important to note that, as a consequence, in one time unit at most $O(\sqrt{m})$ pieces of information/data can "cross" the boundary between S and the remainder of the original mesh. For problems where a lot of data movement is required, transfer of data items across the boundary between S and the remainder of the original mesh clearly imposes a lower bound on the running time.

We also present a novel technique based on counting the amount of information that is possibly obtained by any processor at the end of a time unit. For instance, consider m data items to start with in a $\sqrt{n} \times \sqrt{n}$ MMB. We view the information available to a processor as consisting of two distinct types: on the one hand, the processor may have access to information that has been broadcast on buses and, on the other, the processor may have access to information received from local communication only. Clearly, the problem at hand can not be solved without getting the information about all the data elements. Thus, this imposes a bound on the minimum number of time units that must elapse before any one processor in the mesh gets all the information.

2.4 Data Movement

The purpose of this section is to review a number of fundamental results for the MMB that will be instrumental in the design of our algorithms.

The problem of list ranking is to determine the rank of every element in a given list, stored as an unordered array, that is, the number of elements following it in the list. Recently, Bhagavathi *et al.,* [13] have proposed a time-optimal algorithm for list ranking on MMB's.

Proposition 2.4.1 [13] The task of ranking an n-element linked list stored in one row of an MMB of size $n \times n$ can be performed in $O(\log n)$ time. Furthermore, this is time-optimal. $\square$

Recently, Olariu *et al.* [50] have proposed a constant time algorithm to merge two sorted sequences of total length $\sqrt{n}$ stored in one row of a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. More precisely, the following result was established in [50].

Proposition 2.4.2 Let $S_1 = (a_1, a_2, \ldots, a_r)$ and $S_2 = (b_1, b_2, \ldots, b_s)$, with $r+s = \sqrt{n}$, be sorted sequences stored in the first row of a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$, with $P(1,i)$ holding a_i $(1 \leq i \leq r)$ and $P(1, r+i)$ holding b_i $(1 \leq i \leq s)$. The two sequences can be merged into a sorted sequence in $O(1)$ time. $\square$

Since merging is an important ingredient in our algorithms, we now give the details of the merging algorithm [50]. To begin, using vertical buses, the first row is replicated in all rows of the mesh. Next, in every row i $(1 \leq i \leq r)$, processor $P(i,i)$ broadcasts a_i horizontally on the corresponding row bus. It is easy to see that for every i, a unique processor $P(i,j)$ $(r+1 \leq j \leq \sqrt{n})$, will find that $b_{j-1} < a_i \leq b_j$. Clearly, this unique processor can now use the horizontal bus to broadcast j back to $P(i,i)$. In turn, this processor has enough information to compute the position of a_i in the sorted sequence. In exactly the same way, the position of every b_j in the sorted sequence can be computed in $O(1)$ time. Knowing their positions in the sorted sequence, the data items can be moved to their final positions in $O(1)$ time.

Next, we consider the problem of merging multiple sorted sequences with a common length. Let a sequence of $\sqrt{n}$ items a_1, a_2, ..., $a_{\sqrt{n}}$ be stored, one per processor, in the first row of a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. Suppose that the sequence consists of k sorted subsequences and each subsequence consists of $\frac{\sqrt{n}}{k}$ consecutive elements of the original sequence. The goal is to sort the entire sequence.

For definiteness, we assume that subsequence j $(1 \leq j \leq k)$, contains the items $a_{\frac{(j-1)\sqrt{n}}{k}+1}, \ldots, a_{\frac{j\sqrt{n}}{k}}$. Our sorting algorithm proceeds by merging the subsequences two at a time into longer and longer subsequences. The details are as follows. We set aside submeshes of size $\frac{2\sqrt{n}}{k} \times \frac{2\sqrt{n}}{k}$ and every pair of consecutive subsequences is merged in each one of these submeshes. Specifically, the first pair of subsequences is allocated the submesh with $P(1,1)$ in i ts north-west corner; the next pair of subsequences is allocated the submesh with processor $P(\frac{2\sqrt{n}}{k}+1, \frac{2\sqrt{n}}{k}+1)$ in its north-west corner, and so on. Note that moving the subsequences to the corresponding submeshes amounts to a simple broadcast operation on vertical buses. Now in each submesh, the corresponding subsequences are merged using the algorithm described above. By Proposition 2.4.2, this operation takes constant time. Repeating the merging operation $\lceil \log k \rceil$ times, the entire sequence is sorted. Consequently, we have the following result.

Lemma 2.4.3 A sequence consisting of k equal-sized sorted subsequences stored in the first row of a MMB of size $\sqrt{n} \times \sqrt{n}$ can be sorted in O$(\log k)$ time. □

Proposition 2.4.4 [50] An n-element sequence of items from a totally ordered universe stored one item per processor in the first row of an MMB of size $n \times n$ can be sorted in O$(\log n)$ time. Furthermore, this is time-optimal. □

It is an easy observation that the above result can be extended as follows.

Corollary 2.4.5 For an arbitrary constant c, a sequence of cn items stored c per processor in one row of a MMB of size $n \times n$ can be sorted in O$(\log n)$ time. Furthermore, this is time optimal. □

All Nearest Smaller Values problem (ANSV, for short) has been introduced in [10], where it is argued that this is a fundamental problem of parallel processing, as a number of other problems reduce to it. The ANSV problem is formulated as follows: given a sequence of n real numbers $a_1, a_2, \ldots, a_n$, for each a_i $(1 \leq i \leq n)$, find the nearest element to its left and the nearest element to its right. Recently, Bhagavathi *et al.,* [13] has proposed a time-optimal algorithm for the ANSV problem.

Proposition 2.4.6 [13] An arbitrary instance of size n of the ANSV problem can be solved in O$(\log n)$ time on an MMB of size $n \times n$. Furthermore, this is time-optimal. □

The *prefix sums* problem is a key ingredient in many parallel algorithms. The problem is stated as follows: given a sequence a_1, a_2, ..., a_n of items, compute all the sums of the form a_1, $a_1 + a_2$, $a_1 + a_2 + a_3$, ..., $a_1 + a_2 + \cdots + a_n$.

Proposition 2.4.7 [37, 50] The prefix sums (also maxima or minima) of a sequence of n real numbers stored in one row of an MMB of size $n \times n$ can be computed in $O(\log n)$ time. Furthermore, this is time-optimal. $\square$

For later reference, we now describe the details of a simple data movement that allows to compact a list by eliminating some of its elements. For definiteness, suppose that the processors in the first row of the mesh store a sequence of n items, a_1, a_2, ..., a_n with some of the items marked. Assume further that every marked item knows its rank among the marked items. We wish to obtain a sublist consisting of the marked elements stored in the leftmost positions of the first row of the mesh. This task can be performed as follows. Suppose that a_i is the k-th marked element in the sequence; processor $P(1,i)$ will broadcast a_i vertically to processor $P(k,i)$ which in turn will broadcast a_i horizontally to $P(k,k)$. Finally, $P(k,k)$ will broadcast a_i vertically to $P(1,k)$, as desired. Consequently, we have the following result.

Lemma 2.4.8 Consider a sequence a_1, a_2, ..., a_n of items stored in the first row of an MMB of size $n \times n$, one item per processor, with some of the items marked. If every marked item knows its rank among the marked items, then a sublist consisting of the marked elements stored in the leftmost positions of the first row of the mesh can be obtained in $O(1)$ time. $\square$

Finally, we look at a data movement technique on a MMB of size $\sqrt{n} \times \sqrt{n}$ involving the reorganization of data items in the first x columns of the mesh sorted in row-major order to column-major order (see Figure 2.1(a) to 2.1(d)). This can be accomplished by a series of simple data movement operations whose details follow. To simplify the notation we shall assume that $\frac{\sqrt{n}}{x}$ is an integer. The first x columns of the mesh are moved one at a time as follows. Suppose that column s is being moved. In constant time, every processor $P(r,s)$ $(1 \leq r \leq \sqrt{n})$ broadcasts the item it holds to processor $P(r,k)$ where $k = (r \bmod \frac{\sqrt{n}}{x} - 1)x + s$ (Figure 2.1.(b)).

We now view the mesh as consisting of horizontal submeshes $R_1, R_2, \ldots, R_x$, each of size $\frac{\sqrt{n}}{x} \times \sqrt{n}$. In a submesh R_p $(1 \leq p \leq x)$, each processor $P(l,j)$ $(1 \leq j \leq \sqrt{n}$ and $l = \lceil \frac{j}{x} \rceil)$ broadcasts its value along the column bus j and $P(j,j)$ records it (Figure 2.1(c)). Again, in constant time, each processor $P(j,j)$ broadcasts its value along row bus j to processor $P(p,j)$. The above can be repeated for

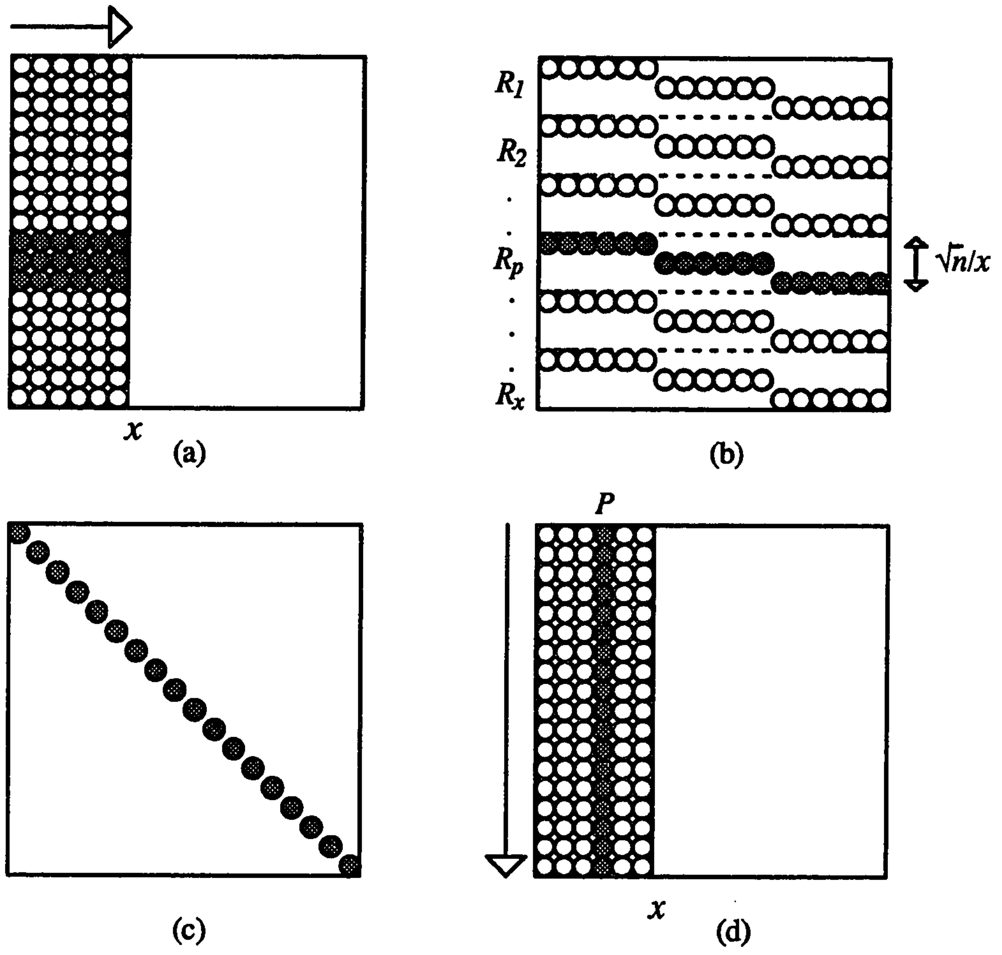


Figure 2.1: Illustrating the data movement of Lemma 2.4.9

each submesh R_p $(1 \leq p \leq x)$, thus accomplishing the required data movement in O(x) time. To summarize our findings we state the following result.

Lemma 2.4.9 Given a $\sqrt{n} \times \sqrt{n}$ mesh with multiple broadcasting, with input elements stored in the first x columns in sorted row-major order, the data can be moved into a sorted column-major order in the first x columns, in O(x) time. □

We now state a simple data movement that is useful in many of our algorithms. Consider a $\sqrt{n} \times \sqrt{n}$ MMB that we perceive as consisting of submeshes $R_1, R_2, \ldots, R_{\frac{\sqrt{n}}{x}}$, each of size $\sqrt{n} \times x$. We are interested in replicating the contents of R_1 in all the other submeshes. This task can be achieved as follows.

The columns of R_1 are handled one by one. In one time unit, all the elements belonging to a column j of R_1 can be replicated in the jth column of $R_2, R_3, \ldots, R_{\frac{\sqrt{n}}{x}}$ as follows. Every processor $P(i,j)$ in this column broadcasts the value it stores horizontally along the row bus i and every processor $P(i, (k-1)x+j)$ $(1 \leq k \leq \frac{\sqrt{n}}{x})$, records the value on the bus. Repeating the above for every column of R_1, its contents are replicated as desired. Thus we have the following result.

Lemma 2.4.10. The elements stored in the first x columns of a MMB of size $\sqrt{n} \times \sqrt{n}$ can be replicated throughout the mesh in O(x) time. □

Corollary 2.4.11 Given a $\sqrt{n} \times \sqrt{n}$ MMB, with input elements stored in the first x columns, the data can be moved to the first x rows of the mesh in O(x) time. □

Chapter 3

Semigroup Computations

3.1 Introduction

Semigroup computations are a fundamental algorithmic tool, with numerous applications to all branches of parallel processing [3, 38]. Formally, the *semigroup computation* problem [3, 38] is defined as follows: given a sequence $a_1, a_2, \ldots, a_m$ of items from a semigroup S with an associative operation $\oplus$, compute $a_1 \oplus a_2 \oplus \ldots \oplus a_m$. The purpose of this chapter is to propose a unifying look at semigroup computations on MMBs.

The semigroup computation problem has been well studied in the literature. However, up to now, lower bounds and matching algorithms for semigroup computations have been obtained only for the sparse and dense cases, but nothing is known about the general case. The present work was motivated by an attempt to remove the artificial limitation of the problem to the sparse and dense cases. We present a time-optimal algorithm for the semigroup computation in the general case. This algorithm unifies the previous special case results. In this context, we first derive a lower bound for semigroup computations in the general case. More precisely, we show that every algorithm that solves the semigroup computation involving m $(2 \le m \le n)$ items must take $\Omega(\max\{\min\{\log m, \log \frac{n^{\frac{2}{3}}}{m^{\frac{1}{3}}}\}, \frac{m^{\frac{1}{3}}}{n^{\frac{1}{6}}}\})$ time on a MMB of size $\sqrt{n} \times \sqrt{n}$. As expected, our result matches the known lower bounds for the sparse and dense cases.

We also show that the lower bound is tight by providing an algorithm for the same that runs in $O(\max\{\min\{\log m, \log \frac{n^{\frac{2}{3}}}{m^{\frac{1}{3}}}\}, \frac{m^{\frac{1}{3}}}{n^{\frac{1}{6}}}, \})$ time. Our algorithm contains the algorithms in [21, 37, 40] as special cases. Further, we establish that the semigroup algorithms in [8, 21] remain optimal even if "slightly" more processors are available.

The remainder of the chapter is organized as follows: section 3.2 gives the lower bound arguments for the general case; section 3.3 discusses the details of a time-optimal algorithm for semigroup computation. finally, section 3.4 summarizes the results and poses a number of open problems.

3.2 The Lower Bound

The purpose of this section is to establish a non-trivial lower bound on semigroup computations on MMBs. For this purpose, consider a MMB $\mathcal{R}$ of size $\sqrt{n} \times \sqrt{n}$ storing m $(2 \leq m \leq n)$ items $a_1, a_2, \ldots, a_m$ from a semigroup S with an associative operation $\oplus$. The items are stored in column-major order, one per processor, in the first $\lceil \frac{m}{\sqrt{n}} \rceil$ columns of the mesh. We shall distinguish several ranges for m.

First, in case $2 \leq m < \sqrt{n}$, the lower bound of $\Omega(\log m)$ of [40] applies. For further reference we state the following result.

Lemma 3.2.1 (Lin *et al.* [40]) Any algorithm that solves the semigroup computation problem involving m $(1 < m \leq \sqrt{n})$ items stored consecutively, one per processor, in the first column of a MMB of size $\sqrt{n} \times \sqrt{n}$ must take $\Omega(\log m)$ time. □

Therefore, from now on we shall assume that $m \geq \sqrt{n}$, which implies

$$\frac{n^{\frac{2}{3}}}{m^{\frac{1}{3}}} \leq \sqrt{n}. \tag{3.1}$$

For the remainder of the lower bound argument, we assume for simplicity that $\frac{m}{\sqrt{n}}$ is an integer.

Lemma 3.2.2 Any algorithm that solves the semigroup computation problem involving m $(\sqrt{n} \leq m \leq n)$ items stored one per processor in the first $\frac{m}{\sqrt{n}}$ columns of a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$ must take at least $\Omega(\log \frac{n^{\frac{2}{3}}}{m^{\frac{1}{3}}})$ time.

Proof. We construct a sequence of m items as follows. The first group consists of $\frac{n^{\frac{2}{3}}}{m^{\frac{1}{3}}}$ items that we place consecutively in the first column of the mesh. Note that by virtue of (3.1) this is always possible. The remaining items are chosen to be the null element with respect to $\oplus$ (note that even if the semigroup has no null element, we can augment the structure, for lower bound purposes, by adding such an element). Now the result of the semigroup computation is known as soon as the partial result involving the $\frac{n^{\frac{2}{3}}}{m^{\frac{1}{3}}}$ items in the first column is available, and conversely. The lower bound of [40] applies, guaranteeing that $\Omega(\log \frac{n^{\frac{2}{3}}}{m^{\frac{1}{3}}})$ time is needed to complete the computation. □

We shall complement the result of Lemma 3.2.2 by presenting a different lower bound argument that is based on *counting*. Our argument can be seen as a generalization of lower bound arguments in [2, 8, 18]. The semigroup computation terminates at the end of t computational steps, when some processor in $\mathcal{R}$ has enough information to compute $a_1 \oplus a_2 \oplus \ldots \oplus a_m$.

We view the information available to a processor as consisting of two distinct types: on the one hand, the processor may have access to information that has been broadcast on buses and, on the other, the processor may have access to information received from local communication only. To make this statement precise, let $\mathcal{B}(i)$ be the *set* of input items that have contributed to any information which has been broadcast on some bus at the end of time unit i, and write $b(i) = |\mathcal{B}(i)|$. Further, for each processor P of the mesh, let $\mathcal{L}(i, P)$ be the *set* of all input elements initially stored in processors whose Manhattan distance from P is at the most i. It follows that $\mathcal{L}(0, P) = \{\text{item stored initially by } P\}$. The following result follows directly from the definition of Manhattan distance.

Lemma 3.2.3 With $l(i)$ standing for $\max_{P \in \mathcal{R}} |\mathcal{L}(i, P)|$ we have

$$\begin{cases} l(i) \leq 1 + 2i(i+1) & \text{for } i \geq 1 \\ l(0) = 1 \end{cases}$$

Lemma 3.2.4 Let t be the least integer for which $b(t) + l(t) \geq m$. Then t is a lower bound on the number of computational steps needed by any algorithm to correctly solve the semigroup problem.

Proof. Suppose not; let $j < t$ be the number of computational steps after which some processor P of the mesh $\mathcal{R}$ has enough information to know the result of the semigroup computation.

Note, however, that the number of input items that have contributed to the information known by P is bounded by $b(j) + l(j)$. By our choice of t, it must be that $b(j) + l(j) < m$, implying that P can complete the semigroup computation without having all input items contribute. But now we have reached a contradiction since one may change the items that have not contributed, invalidating the algorithm. □

Lemma 3.2.5 For all i ($i \geq 0$),

$$\begin{cases} b(i) \leq b(i-1) + 2\sqrt{n} * l(i-1) & \text{for } i \geq 1 \\ b(0) = 0 \end{cases}$$

Proof. First, the fact that $b(0) = 0$ follows from the definition of $\mathcal{B}(0)$. Further, observe that by definition, the set of input items that have contributed to the information that has been broadcast on any bus of the mesh can only be enriched by having some processor broadcast, in the i-th time unit, information that it has received locally by the end of the first $i-1$ computational steps. Since there are $2\sqrt{n}$ buses available for broadcasting, the conclusion follows. □

We are now in a position to state the following result that provides yet another component of the overall lower bound.

Lemma 3.2.6 Any algorithm that solves the semigroup computation problem involving m ($\sqrt{n} \le m \le n$) items stored one item per processor in the first $\frac{m}{\sqrt{n}}$ columns of a MMB of size $\sqrt{n} \times \sqrt{n}$ must take at least $\Omega(\frac{m^{\frac{1}{3}}}{n^{\frac{1}{6}}})$ time.

Proof. Let t denote the least number of steps needed for any semigroup algorithm to terminate. The recurrence of Lemma 3.2.5 along with Lemma 3.2.3 yields:

$$\textstyle\sum_{i=1}^{t} b(i) \le \sum_{i=1}^{t} b(i-1) + 2\sqrt{n}\sum_{i=1}^{t}(1 + 2i(i-1))$$

which telescopes to

$$b(t) \le 2t\sqrt{n} + 4\sqrt{n}\frac{t(t+1)(2t+1)}{6} - 4\sqrt{n}\frac{t(t+1)}{2}.$$

Now after appropriately upperbounding the right hand size of this inequality we obtain

$$b(t) \le \frac{4}{6}\sqrt{n}t(t+1)(2t+1) \le 4\sqrt{n}t^3.$$

Now Lemma 3.2.3 and Lemma 3.2.4 combined with the above inequality guarantee that

$$m \le b(t) + l(t) \le 4\sqrt{n}t^3 + 1 + 2t(t+1).$$

By suitably upperbounding the right hand side, we get

$$m \le 4\sqrt{n}t^3 + 2t^3 \le 8\sqrt{n}t^3$$

and so,

$$t \ge \frac{1}{2}\left(\frac{m}{\sqrt{n}}\right)^{\frac{1}{3}}$$

which completes the proof of Lemma 3.2.6. □

To unify the results in Lemma 3.2.1, Lemma 3.2.2, and Lemma 3.2.6 we make the following observation, the justification for which is routine.

Observation 3.2.7 Let m be an arbitrary integer in the range $2 \leq m \leq n$. Then

- for $2 \leq m < \sqrt{n}$, $\frac{m^{\frac{1}{3}}}{n^{\frac{1}{6}}} < \log m < \log \frac{n^{\frac{2}{3}}}{m^{\frac{1}{3}}}$;
- for $\sqrt{n} \leq m \leq n$, $\log \frac{n^{\frac{2}{3}}}{m^{\frac{1}{3}}} \leq \log m$.

Thus, by Lemma 3.2.1, Lemma 3.2.2, Lemma 3.2.6, and Observation 3.2.7 combined we have the following lower bound result for semigroup computations on MMBs.

Theorem 3.2.7 Any algorithm that solves the semigroup problem involving m $(2 \leq m \leq n)$ items in column-major order in the first $\lceil \frac{m}{\sqrt{n}} \rceil$ columns of a MMB of size $\sqrt{n} \times \sqrt{n}$ must take at least $\Omega(\max\{\min\{\log m, \log \frac{n^{\frac{2}{3}}}{m^{\frac{1}{3}}}\}, \frac{m^{\frac{1}{3}}}{n^{\frac{1}{6}}},\})$ time. □

3.3 The Algorithm

The purpose of this section is to exhibit an algorithm for semigroup computation whose running time matches the lower bound derived in the previous section.

Consider a mesh $\mathcal{R}$ with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. The m $(2 \leq m \leq n)$ data items are stored in the first $\lceil \frac{m}{\sqrt{n}} \rceil$ columns of $\mathcal{R}$, in column-major order. At various steps of the algorithm, the mesh $\mathcal{R}$ may be dynamically partitioned into submeshes to suit computational needs. We will describe these partitions as they become necessary. Our algorithm consists of the following sequence of computational steps. In case $m < \sqrt{n}$, Step 0 below is executed and the algorithm terminates. Otherwise, Steps 1 through 3 are executed. The details of these steps follow.

Step 0. In case the number of input items, m, is less than $\sqrt{n}$, we proceed recursively. Consider the submesh $\mathcal{M}$ determined by the first m rows and m columns of $\mathcal{R}$ (refer to Figure 3.1).

We partition the submesh $\mathcal{M}$ into submeshes of size $\frac{m}{2} \times \frac{m}{2}$ each, and recursively solve the semigroup computation problem. Specifically, the semigroup computation involving the first half of the input is solved in the north-west submesh of $\mathcal{M}$, while the semigroup computation involving the second half of the input is solved in the south-east submesh of $\mathcal{M}$. Clearly, this is possible and no broadcasting conflict will arise. Furthermore, it is easy to confirm that the running time of this step is bounded by $O(\log m)$.

Step 1. We shall refer to the submesh consisting of the first $\frac{m}{\sqrt{n}}$ columns[1] as $\mathcal{R}'$.

[1]For simplicity we assume that $\frac{m}{\sqrt{n}}$ is an integer

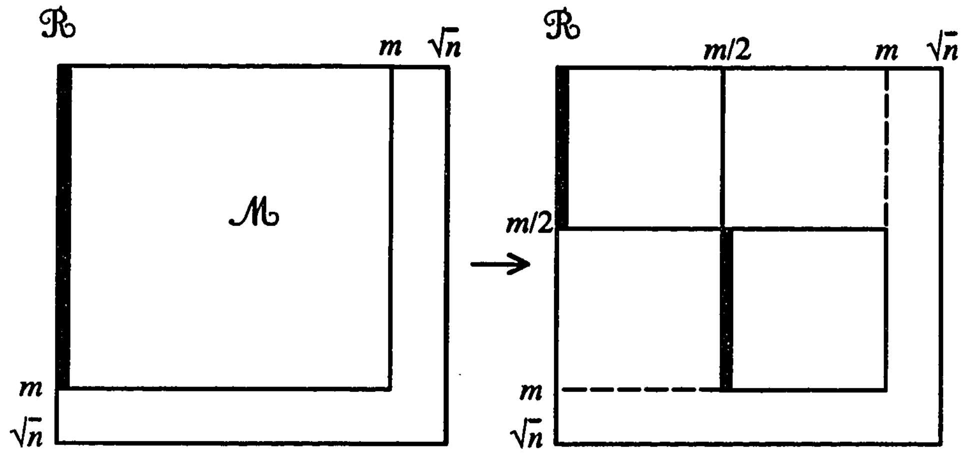


Figure 3.1: Step 0 of semigroup computations.

To make the presentation easier to follow, we let

$$x = \frac{m^{\frac{1}{3}}}{n^{\frac{1}{6}}}. \tag{3.2}$$

In this notation, we view the mesh $\mathcal{R}'$ as consisting of submeshes $R_{i,j}$ $(1 \le i \le \frac{\sqrt{n}}{x}, 1 \le j \le \frac{m}{x\sqrt{n}})$ of size $x \times x$, with each $R_{i,j}$ involving processors $P(r,s)$ with $1+(i-1)x \le r \le ix, 1+(j-1)x \le s \le jx$. The leftmost processor in the top row of every such submesh is termed the *leader* of the submesh.

For the sake of simplicity, we assume that $\frac{m}{x\sqrt{n}}$ and $\frac{\sqrt{n}}{x}$ are integers. In each of the submeshes $R_{i,j}$, perform the semigroup computation in $O(x)$ time, using local communication only [38], and store the partial result in the leader of the submesh, that is, processor $P(1+(i-1)x, 1+(j-1)x)$ (refer to Figure 3.2).

To summarize, at the end of Step 1 the leader of every submesh $R_{i,j}$ contains the partial result $\alpha_{i,j}$ of the semigroup computation involving the items in $R_{i,j}$. Furthermore, Step 1 takes $O(x)$ time.

Step 2. We now view the mesh $\mathcal{R}'$ as a collection of submeshes R_i $(1 \le i \le \frac{\sqrt{n}}{x})$, with R_i containing the submeshes $R_{i,1}, R_{i,2}, \ldots, R_{i,\frac{m}{x\sqrt{n}}}$. The objective of this step is to perform the semigroup computation on the partial results in each of the submeshes R_i. The new partial result will be stored by processor $P(1+(i-1)x, 1)$, the leader of the submesh R_i.

The idea is to compact the $\alpha_{i,j}$'s in R_i into the first $\frac{m}{x^2\sqrt{n}}$ columns of R_i and to perform the semigroup computation after compaction. Note that by virtue of (3.2), $\frac{m}{x^2\sqrt{n}} = x$ confirming that the data movement is feasible. The details are presented next.

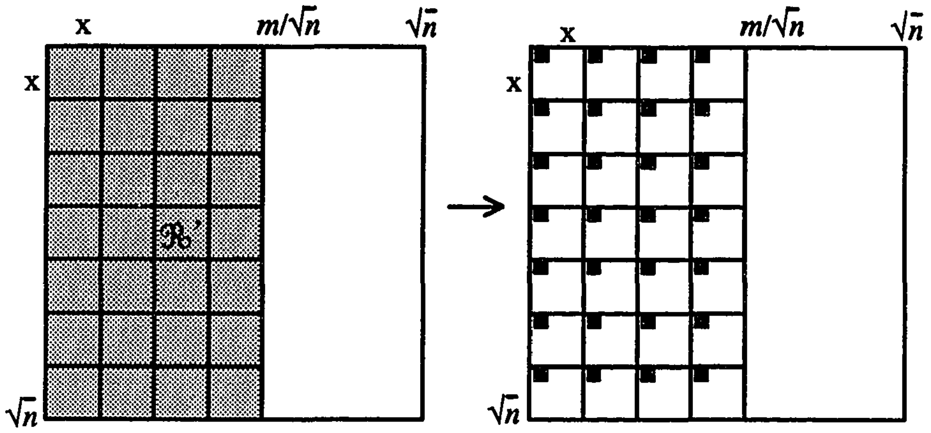


Figure 3.2: Step 1 of semigroup computations.

Partition the $R_{i,j}$'s into x groups such that the r^{th} group involves submeshes $R_{i,1+(r-1)x}, \ldots, R_{i,rx}$ (refer to Figure 3.3).

Next, compact all the $\alpha_{i,j}$'s in the r^{th} group into column r of R_i. To avoid broadcasting conflicts, every processor $P(1+(i-1)x, 1+(j-1)x)$ $(1 \leq i \leq \frac{\sqrt{n}}{x}; 1 \leq j \leq \frac{m}{x\sqrt{n}})$, sends the value $\alpha_{i,j}$ it contains vertically to processor $P(1+(i-1)x, 1+(j-1)x+j\text{mod}x)$, using local links only. This can be done in $O(x)$ time. Finally, in each R_i, groups of x $\alpha_{i,j}$'s are moved horizontally to their final destination. Note that by virtue of (3.2) the data movement described above is completed in $O(x)$ time.

As a result of the previous data movement the partial results of Step 1 are contained in the first x columns of the mesh. Now in every $R_{i,1}$ perform the semigroup computation using local communications only. Clearly, (3.2) guarantees that this can be done in $O(x)$ time. Once computed, the new partial results are stored in the leader of every submesh R_i $(1 \leq i \leq \frac{\sqrt{n}}{x})$.

Step 3. We now have one partial result per submesh R_i. To complete the algorithm, we need to perform the semigroup computation on these $\frac{\sqrt{n}}{x}$ partial results.

We begin by compacting the partial results in the first $\frac{\sqrt{n}}{x}$ positions in the first column of $\mathcal{R}$. This is done as follows: every processor $P(1+(i-1)x, 1)$ that holds a partial result β_i at the end of Step 2, broadcasts β_i horizontally to processor $P(1+(i-1)x, i)$, which in turns broadcasts β_i vertically to $P(i,i)$, and finally to $P(i,1)$ (refer to Figure 3.4).

Clearly, the above data movement takes $O(1)$ time. Finally, using the optimal algorithm of Lin *et al.* [40], we perform the semigroup computation on the β_i's in $O(\log \frac{\sqrt{n}}{x})$ time, and store the final result in processor P(1,1).

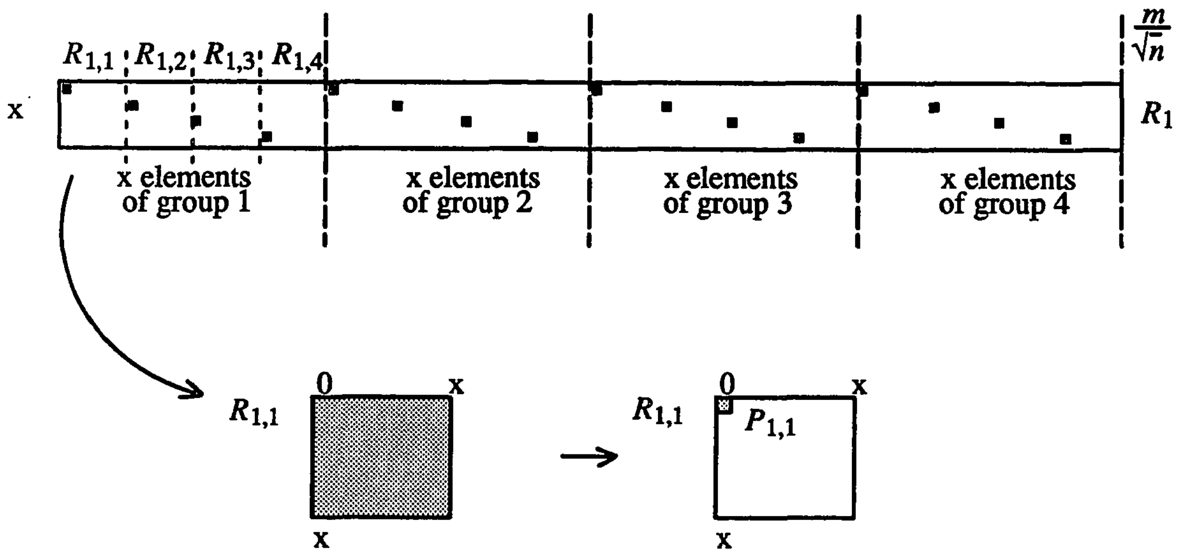


Figure 3.3: Step 2 in the first submesh $\mathcal{R}_1$ for semigroup computations.

To analyze the running time of our algorithm, note that in case $m < \sqrt{n}$ only Step 0 is executed and the running time is bounded by $O(\log m)$. In case $m \geq \sqrt{n}$, Steps 1 through Step 3 are executed. By (3.2), Step 1 and Step 2 are performed in $O(\frac{m^{\frac{1}{3}}}{n^{\frac{1}{6}}})$ time, while Step 3 takes $O(\log \frac{n^{\frac{2}{3}}}{m^{\frac{1}{3}}})$ time.

Finally, the analysis of the running times of Steps 0 through 3, combined with Observation 3.2.7, allow us to state the following result.

Theorem 3.3.8. The semigroup computation involving m $(2 \leq m \leq n)$ items stored in column-major order in the first $\lceil \frac{m}{\sqrt{n}} \rceil$ columns of a MMB of size $\sqrt{n} \times \sqrt{n}$ takes $O(\max\{\min\{\log m, \log \frac{n^{\frac{2}{3}}}{m^{\frac{1}{3}}}\}, \frac{m^{\frac{1}{3}}}{n^{\frac{1}{6}}}\})$ time being, therefore, time-optimal. □

3.4 Conclusions

In this chapter we have addressed the semigroup computation problem involving m $(2 \leq m \leq n)$ items on a MMB of size $\sqrt{n} \times \sqrt{n}$. Our contribution is to have presented the first lower bound and the first time-optimal algorithm which apply to the entire range of m $(2 \leq m \leq n)$. We provided a unifying look at semigroup computations on MMBs by solving the problem for the general case rather than treating only the sparse and dense cases. First, we provided a lower bound by showing that any algorithm which solves the semigroup computation

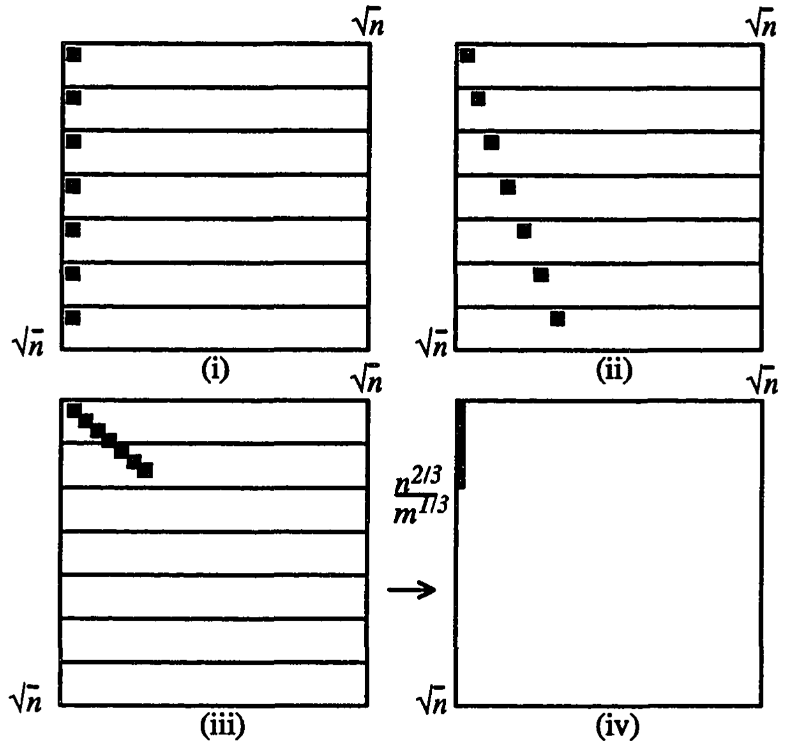


Figure 3.4: Final data movements in Step 3 of semigroup computations.

problem must take at least Ω $(\max\{\min\{\log m, \log \frac{n^{\frac{2}{3}}}{m^{\frac{1}{3}}}\}, \frac{m^{\frac{1}{3}}}{n^{\frac{1}{6}}}\})$ time. Second, we have shown that our lower bound is tight by designing an algorithm whose running time matches the lower bound. These results generalize all semigroup algorithms known to date.

In [8, 21] it was argued that rectangular meshes lead to faster algorithms for semigroup computations than square meshes. It is interesting to note that our lower bound implies that the semigroup algorithms in [8, 21] are optimal *even* if a larger number of processors are available. Specifically, our result shows that semigroup computations involving n items stored one item per processor in a mesh of size $n^{\frac{5}{8}} \times n^{\frac{5}{8}}$ takes as long as the same computation on a rectangular mesh of size $n^{\frac{3}{8}} \times n^{\frac{5}{8}}$. Therefore, in this case, the additional processors cannot speed up the computation.

A number of important problems remain open. Similar unifying results are desirable for other important algorithmic problems on meshes with multiple broadcasting. Candidate problems include, prefix computation, selection, sorting, and list ranking among many others.

Chapter 4

Sorting

4.1 Introduction

This chapter is concerned with sorting on a mesh with multiple broadcasting. Sorting is unquestionably one of the most extensively investigated topics in computer science. It is well known [3, 38] that n data items can be sorted in $O(\sqrt{n})$ time on a mesh-connected machine of size $\sqrt{n} \times \sqrt{n}$. Furthermore, this result is easily shown to be both time-optimal and VLSI-optimal. Recently, a number of sorting algorithms have been proposed for enhanced meshes [40, 56]. An easy information transfer argument shows that for meshes of area n, even when enhanced with multiple broadcasting or with a dynamically reconfigurable bus system, $\Omega(\sqrt{n})$ is a time lower bound for sorting n items.

This somewhat counter-intuitive result motivated us to look at the following problem. Suppose that we are given a mesh or enhanced mesh of size $\sqrt{n} \times \sqrt{n}$ and the goal is to sort m ($\sqrt{n} \leq m \leq n$) items stored in the first $\lceil \frac{m}{\sqrt{n}} \rceil$ columns of the machine. How fast can this task be performed? It is easy to show that the $\Omega(\sqrt{n})$ is a time lower bound for the unenhanced mesh. Clearly, no algorithm can be VLSI-optimal in this case.

The contribution of this work is to show that we can do better on meshes with multiple broadcasting. Specifically, we show that once we fix a positive integer constant c, we can sort m items ($n^{\frac{1}{2}+\frac{1}{2c}} \leq m \leq n$) in $O(\frac{m}{\sqrt{n}})$ time. We show that this is time-optimal for this architecture. It is also easy to see that this achieves the VLSI lower bound in the word model.

The remainder of this chapter is organized as follows: section 4.2 presents our lower bound arguments; section 4.3 presents the details of our optimal sorting algorithm. Finally, section 4.4 summarizes the results and poses a number of open problems.

4.2 The Lower Bound

The purpose of this section is to show that every algorithm that sorts m ($m \le n$) items given in the first $\lceil \frac{m}{\sqrt{n}} \rceil$ columns of a mesh with multiple broadcasting must take $\Omega(\frac{m}{\sqrt{n}})$ time.

Our argument is of *information transfer* type [38, 70]. Consider the submesh $\mathcal{M}$ consisting of processors $P(r,s)$ with $1 \le r \le \frac{m}{2\sqrt{n}}$ and $1 \le s \le \frac{m}{\sqrt{n}}$. The input will be constructed in such a way that every element initially input into $\mathcal{M}$ will find its final position in the sorted order outside of $\mathcal{M}$. To see that this is possible, note that $m \le n$ guarantees that the number of elements in $\mathcal{M}$ satisfies:

$$\frac{m}{\sqrt{n}} * \frac{m}{2\sqrt{n}} = \frac{m^2}{2n} \le \frac{m}{2}.$$

Since at most $O(\frac{m}{\sqrt{n}})$ items can leave or enter $\mathcal{M}$ in $O(1)$ time, it follows that any algorithm that correctly sorts the input data must take at least $\Omega(\frac{m}{\sqrt{n}})$ time. Thus, we have the following result.

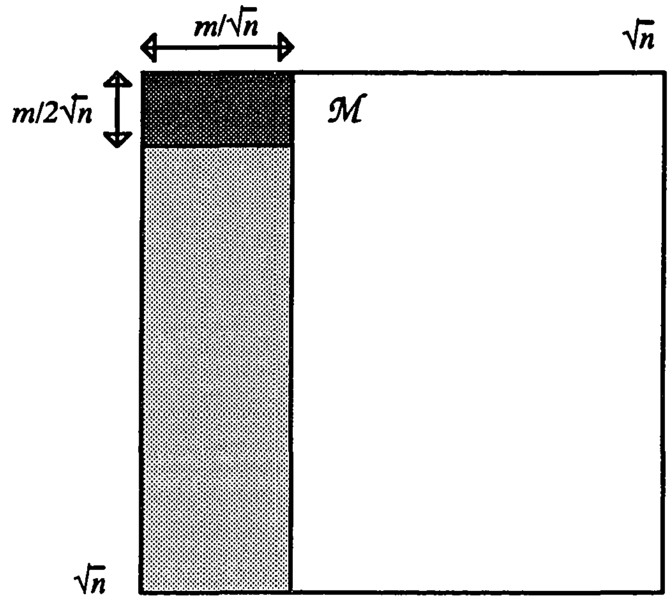


Figure 4.1: Lower bound argument for sorting

Theorem 4.2.1 Every algorithm that sorts m ($m \le n$) items in the first $\frac{m}{\sqrt{n}}$ columns of a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$ must take $\Omega(\frac{m}{\sqrt{n}})$ time.

In addition to time-lower bounds for algorithms solving a given problem, one is often interested in designing algorithms that feature a good VLSI performance

[64]. One of the most used metrics is the product AT^2, where A is the area of the chip and T is the time taken for the problem. A time lower bound based on this metric is strong because it is not based on memory requirements or input/output rate, but on the requirements for information flow within the chip. It is well-known [38, 64, 70] that in the word model the lower bound for sorting m elements on a VLSI chip is m^2. In our case, the size of the set of input elements m varies, while the area of the mesh n is constant. Hence, for any algorithm to be VLSI-optimal, we have $nT^2 = m^2$, where T is the running time. Thus, in this case, the time lower bound of $\Omega(\frac{m}{\sqrt{n}})$ also translates to VLSI-optimality.

The purpose of this chapter is show that the bounds derived above are tight by providing an algorithm with a matching running time.

4.3 The Algorithm

We are now in a position to present our time- and VLSI-optimal sorting algorithm for meshes with multiple broadcasting. Essentially, our algorithm implements the well-known bucket sort strategy. Throughout, we assume a mesh with multiple broadcasting $\mathcal{R}$ of size $\sqrt{n} \times \sqrt{n}$. We also assume that the processors in the first column of the mesh also serve as I/O ports.

Fix an arbitrary positive integer constant c. The input is assumed to be a set S of m items

$$n^{\frac{1}{2}+\frac{1}{2c}} \leq m \leq n \tag{4.1}$$

from a totally ordered universe[1] stored in the first $\frac{m}{\sqrt{n}}$ columns of $\mathcal{R}$. To avoid tedious but inconsequential details, we assume that $\frac{m}{\sqrt{n}}$ is an integer. The goal is to sort these items in column major order, so that they can be output from the mesh in $O(\frac{m}{\sqrt{n}})$ time. We propose to show that with the above assumptions the entire task of sorting can be performed in $O(\frac{m}{\sqrt{n}})$ time. Thus, from our discussion in section 4.2, we can conclude that our algorithm is both time- and VLSI-optimal.

To make the presentation more transparent and easier to follow we refer to the submesh consisting of the first $\frac{m}{\sqrt{n}}$ columns of $\mathcal{R}$ as $\mathcal{M}$ (In other words, $\mathcal{M}$ is the submesh that initially contains the input). Further, a *slice* of size k of the input consists of the items stored in k consecutive rows of $\mathcal{M}$.

We will first present an outline of our algorithm and then proceed to the details. Starting with slices of size $\frac{m}{\sqrt{n}}$ sorted in row major order, we use bucket sort to

[1]We assume O(1) time comparisons among the items in the universe

merge consecutive $\frac{m}{\sqrt{n}}$ of these into slices of size $(\frac{m}{\sqrt{n}})^2$ sorted in row major order. Using the same strategy, these slices are again merged into larger slices sorted in row major order. We proceed with the merging process until we have one slice of size $\sqrt{n}$, sorted in row major order. Finally, employing the data movement discussed in Lemma 2.4.9, the data is converted into column major order.

We proceed to show that the merging of $\frac{m}{\sqrt{n}}$ consecutive sorted slices of size $(\frac{m}{\sqrt{n}})^i$ into sorted slices of size $(\frac{m}{\sqrt{n}})^{i+1}$ requires $O(\frac{m}{\sqrt{n}})$ time. It is convenient to view the original mesh $\mathcal{R}$ as consisting of submeshes $R_{j,k}$ of size $(\frac{m}{\sqrt{n}})^{i+1} \times (\frac{m}{\sqrt{n}})^{i+1}$ with $R_{j,k}$ involving processors $P(r,s)$ such that $(j-1)(\frac{m}{\sqrt{n}})^{i+1} < r \leq j(\frac{m}{\sqrt{n}})^{i+1}$ and $(k-1)(\frac{m}{\sqrt{n}})^{i+1} < s \leq k(\frac{m}{\sqrt{n}})^{i+1}$.

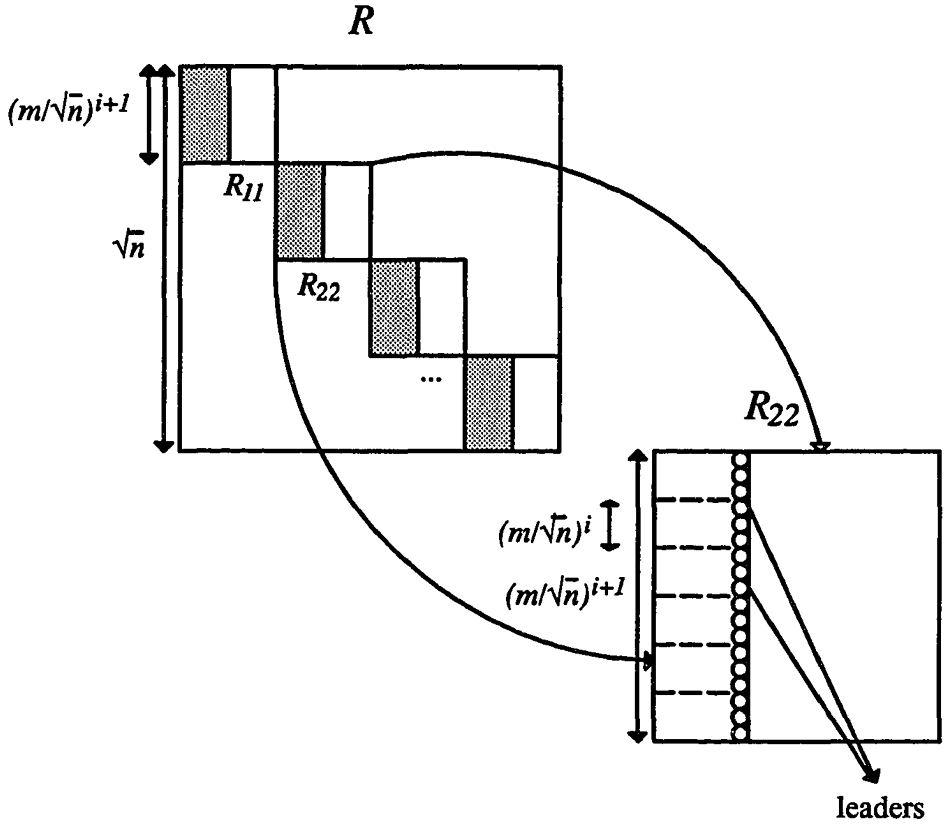


Figure 4.2: Diagonal submeshes and leaders in stage i of sorting

We refer to submeshes $R_{k,k}$ as *diagonal*. The diagonal submeshes can be viewed as independent meshes, since the same task can be performed, in parallel, in all of them without broadcasting conflict. The algorithm begins by moving the data items in every $R_{k,1}$ to the diagonal submesh $R_{k,k}$. This can be accomplished column by column in $O(\frac{m}{\sqrt{n}})$ time. We now present the details of the processing that takes place in parallel in every diagonal submesh $R_{k,k}$.

The rightmost item in every row of $R_{k,k}$ will be referred to as the *leader* of that row (see Figure 4.2). To begin, the sequence of leaders $q_1, q_2, \ldots, q_{(\frac{m}{\sqrt{n}})^{i+1}}$ in $R_{k,k}$ is sorted in increasing order. Note that by virtue of our grouping, the sequence of leaders consists of $\frac{m}{\sqrt{n}}$ sorted subsequences, and so, by Lemma 2.4.3, the sequence of leaders can be sorted in $O(\log \frac{m}{\sqrt{n}})$ time. Let this sorted sequence be a_1, a_2, ..., $a_{(\frac{m}{\sqrt{n}})^{i+1}}$. For convenience, we assign $a_0 = -\infty$.

Next, in preparation for bucket sort, we define a set of $(\frac{m}{\sqrt{n}})^i$ buckets B_1, B_2, ..., $B_{(\frac{m}{\sqrt{n}})^i}$, such that for every j $(1 \leq j \leq (\frac{m}{\sqrt{n}})^i)$,

$$B_j = \{x \in S \mid a_{\frac{(j-1)m}{\sqrt{n}}} < x \leq a_{\frac{jm}{\sqrt{n}}}\} \tag{4.2}$$

By definition, the leaders $a_{\frac{(j-1)m}{\sqrt{n}}+1}$ through $a_{\frac{jm}{\sqrt{n}}}$ belong to bucket B_j. This observation motivates us to call a row in $R_{k,k}$ *regular* with respect to bucket B_j if its leader belongs to B_j. Similarly, a row of $R_{k,k}$ is said to be *special* with respect to bucket B_j if its leader belongs to a bucket B_t with $t > j$, while the leader of the previous row belongs to a bucket B_s with $s \leq j$. To handle the boundary case, we also say that a row is *special* with respect to B_j, if it is the first row in a slice and its leader belongs to B_t with $t > j$. Note that, all items in B_j must be in either regular rows or special rows.

Let us make a crucial observation.

Observation 4.3.1 With respect to every bucket B_j, there exist $\frac{m}{\sqrt{n}}$ regular rows and at most $\frac{m}{\sqrt{n}}$ special rows in $R_{k,k}$.

Proof. The number of regular rows follows directly from the definition of bucket B_j in (4.2). The claim concerning the number of special rows follows from the assumed sortedness of the $\frac{m}{\sqrt{n}}$ slices of size $(\frac{m}{\sqrt{n}})^i$ in $R_{k,k}$. □

In order to process each of the $(\frac{m}{\sqrt{n}})^i$ buckets individually, we view the mesh $R_{k,k}$ as consisting of submeshes $T_1, T_2, \ldots, T_{(\frac{m}{\sqrt{n}})^i}$ of size $(\frac{m}{\sqrt{n}})^{i+1} \times \frac{m}{\sqrt{n}}$. Each submesh T_l is dedicated to bucket B_l, in order to accumulate and process the elements belonging to that bucket.

In $O(\frac{m}{\sqrt{n}})$ time, we replicate the contents of T_1 in T_j $(2 \leq j \leq (\frac{m}{\sqrt{n}})^i)$. Using simple data movements, in each of the submeshes T_l, the values of $a_{(l-1)\frac{m}{\sqrt{n}}}$ and $a_{l\frac{m}{\sqrt{n}}}$ are broadcast to all the elements in it in $O(\frac{m}{\sqrt{n}})$ time so that all the elements that belong to B_l mark themselves. All the unmarked elements change their values to ∞.

Now the mesh $R_{k,k}$ is viewed as consisting of submeshes $Q_{l,j}$ $(1 \leq l \leq (\frac{m}{\sqrt{n}})^i$, $1 \leq j \leq (\frac{m}{\sqrt{n}})^i)$, of size $(\frac{m}{\sqrt{n}}) \times (\frac{m}{\sqrt{n}})$. The processor $P(r, s)$ is in $Q_{l,j}$ if $(l-1)\frac{m}{\sqrt{n}} < r \leq l\frac{m}{\sqrt{n}}$ and $(l-1)\frac{m}{\sqrt{n}} < s \leq l\frac{m}{\sqrt{n}}$. The objective of this step is to move all the elements belonging to bucket B_j in submesh T_j into submesh $Q_{j,j}$. Let q_k be the leader of a regular row. The rank r of this row is given by $r = v \bmod \frac{m}{\sqrt{n}}$, where $q_k = a_v$. Hence, in the order of their ranks, each of the regular rows is moved to the rth row of $Q_{j,j}$ $(j = \lceil \frac{v}{m/\sqrt{n}} \rceil)$, where r is its rank. Thus, all the regular rows with respect to B_j can be moved into the submesh $Q_{j,j}$ in $O(\frac{m}{\sqrt{n}})$ time.

A special row u in T_j with respect to bucket B_j is assigned a rank s, $s = \lfloor \frac{u}{(\frac{m}{\sqrt{n}})^i} \rfloor + 1$. Note that no two special rows can have the same rank. In the order of their ranks, special rows are moved to the rows corresponding to their rank in $Q_{j,j}$. As the number of special rows is at most $\frac{m}{\sqrt{n}}$, the time taken to broadcast all the special rows to $Q_{j,j}$ is $O(\frac{m}{\sqrt{n}})$.

Now each processor in $Q_{j,j}$ holds at most one element from a regular row and one from a special row. We would now like to sort the elements in each of the submeshes $Q_{j,j}$, in an overlaid row major order. In case the number of elements in $Q_{j,j}$ is less than or equal to $\frac{m^2}{n}$, after sorting, the elements can be placed one per processor. If the number of elements exceeds $\frac{m^2}{n}$, the first $\frac{m^2}{n}$ of them are said to belong to generation-1 and the remaining elements are said to belong to generation-2. We would like to place the elements belonging to generation-1 one per processor in row major order and overlay this with those from generation-2, also in the same order. This is done as follows.

Using optimal sorting algorithm for meshes [44, 65], sort the elements from regular rows in $Q_{j,j}$ in $O(\frac{m}{\sqrt{n}})$ time and repeat the same for the elements from special rows. Merging the two sorted sequences thus obtained can be accomplished in another $O(\frac{m}{\sqrt{n}})$ time.

Now in each $Q_{j,j}$, all the elements know their ranks in bucket B_j. Our next goal is to compute the final rank of each of the elements in $R_{k,k}$. Before we give the details of this operation, we let S_1, S_2, ..., $S_{\frac{m}{\sqrt{n}}}$ be the sorted slices of size $(\frac{m}{\sqrt{n}})^i$ in $R_{k,k}$. Let m_j be the largest element in bucket B_j. In parallel, using simple data movement, each m_j is broadcast to all the processors in T_j in $O(\frac{m}{\sqrt{n}})$ time. Next, we determine the rank of m_j in each of the S_l's as follows: in every S_l we identify the smallest item (if any) strictly larger than m_j. Clearly, this can be done in at most $O(\frac{m}{\sqrt{n}})$ time, since every processor only has to compare m_j with the item it holds and with the item held by its predecessor. Now the rank of m_j among the items in $R_{k,k}$ is obtained by simply adding up the ranks of m_j in all the S_l's. Once these ranks are known, in at most $O(\frac{m}{\sqrt{n}})$ time they are broadcast to the first row of $Q_{j,j}$, where their sum is computed in $O(\log \frac{m}{\sqrt{n}})$ time. Observe that once m_j knows its rank in $R_{k,k}$, every item in bucket B_j finds its rank in $R_{k,k}$ by using its rank in the bucket, the size of the bucket and the rank of m_j in O(1) time. Consequently we have proved the following result.

Lemma 4.3.2 The rank in $R_{k,k}$ of every element in every bucket can be determined in $O(\frac{m}{\sqrt{n}})$ time. □

Finally, we need to move all the elements into the first $\frac{m}{\sqrt{n}}$ columns of $R_{k,k}$ in row major order. In O(1) time, each element determines its final position from its rank r as follows. The row number x is given by $\lceil \frac{r}{(\frac{m}{\sqrt{n}})} \rceil$ and the column number y by $((r-1) \bmod \frac{m}{\sqrt{n}})+1$. In every submesh T_j, each element belonging to generation-1 is moved to the row x it belongs to (after sorting) by broadcasting the $\frac{m}{\sqrt{n}}$ rows of $Q_{j,j}$, one at a time. This takes $O(\frac{m}{\sqrt{n}})$ time. Now that every row of $R_{k,k}$ contains at most $\frac{m}{\sqrt{n}}$ elements. Knowing the columns they belong to, in another $\frac{m}{\sqrt{n}}$ time all the elements can be broadcasted to their positions along the row buses. This is repeated for the generation-2 elements. In parallel, every diagonal submesh $R_{k,k}$ moves back its data into the first $\frac{m}{\sqrt{n}}$ columns of submesh $R_{k,1}$. Thus, in an overall time of $O(\frac{m}{\sqrt{n}})$, all the elements are moved to the first $\frac{m}{\sqrt{n}}$ columns of $\mathcal{R}$. Now $\mathcal{R}$ contains slices of size $(\frac{m}{\sqrt{n}})^{i+1}$, each sorted in row major order.

To summarize our findings we state the following result.

Lemma 4.3.3 Merging $\frac{m}{\sqrt{n}}$ consecutive sorted slices of size $(\frac{m}{\sqrt{n}})^i$ into sorted slices of size $(\frac{m}{\sqrt{n}})^{i+1}$ can be done in $O(\frac{m}{\sqrt{n}})$ time. □

Let $T(i+1)$ be the worst-case running time of the basic step described above. It is easy to confirm that the recurrence describing the behavior of $T(i+1)$ is

$$\begin{cases} T(i+1) = T(i) + O(\frac{m}{\sqrt{n}}) & \text{for } i \geq 1 \\ T(1) = \frac{m}{\sqrt{n}} \end{cases}$$

The algorithm terminates at the end of t iterations, when

$$(\frac{m}{\sqrt{n}})^{t+1} = \sqrt{n}. \tag{4.3}$$

This implies that

$$t+1 = \frac{\log\sqrt{n}}{\log\frac{m}{\sqrt{n}}}. \tag{4.4}$$

By virtue of (4.1), (4.4) yields

$$t = c. \tag{4.5}$$

Thus the total running time of our algorithm is given by

$$T(t+1) = O(c\frac{m}{\sqrt{n}}) \tag{4.6}$$

which is obtained by solving the above recurrence. Since c is a constant, we have proved the following result.

Theorem 4.3.4 For every choice of a positive integer constant c, m items ($n^{\frac{1}{2}+\frac{1}{2c}} \leq m \leq n$) stored in the first $\lceil\frac{m}{\sqrt{n}}\rceil$ columns of a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$ can be sorted in $O(\frac{m}{\sqrt{n}})$ time. Furthermore, this is both time- and VLSI-optimal. □

4.4 Conclusions

In this chapter we have presented a time- and VLSI-optimal sorting algorithm for meshes with multiple broadcasting. Specifically, we have shown that once we fix a positive integer constant c, we can sort m items in the range $n^{\frac{1}{2}+\frac{1}{2c}} \leq m \leq n$ in $O(\frac{m}{\sqrt{n}})$ time.

A number of problems remain open. First, it would be of interest to see whether the bucketing technique used in this chapter can be applied to the problem of selection. To this day, no time-optimal selection algorithms for this architecture are known. Also, it is not known whether the technique used in this chapter can be extended to meshes enhanced by the addition of k global buses [2, 18]. Further, we would like to completely resolve these issues concerning optimal sorting over the entire range $\sqrt{n} \leq m \leq n$. Note that the results of Lin and others [40] show that for m near $\sqrt{n}$, $\Omega(\log n)$ is the time lower bound for sorting in this architecture. Their results imply that a sorting algorithm cannot be VLSI-optimal for m near $\sqrt{n}$.

Chapter 5

Multiple Search

5.1 Introduction

The purpose of this chapter is to propose a time-optimal algorithm for the multiple search problem on meshes with multiple broadcasting and to show that a number of problems in computer graphics, image processing, robotics, and computational geometry reduce to the multiple search problem or a variant thereof. The *multiple search* problem [4, 72] is defined as follows: given a sorted sequence $A = a_1, a_2, \ldots, a_n$ of items from a totally ordered universe along with an arbitrary sequence $Q = q_1, q_2, \ldots, q_m$ $(1 \leq m \leq n)$ of *queries* from the same universe, determine for every q_j $(1 \leq j \leq m)$ the unique a_i for which $a_{i-1} \leq q_j < a_i$. To handle boundary conditions, we augment the sequence A by the addition of two "dummy" elements, namely $a_0 = -\infty$ and $a_{n+1} = \infty$. The multiple search problem is considered to be a fundamental algorithmic problem [4, 72] and finds applications to query processing in databases, computer graphics, image processing, and computational geometry, to name just a few. Recently, Akl and Meijer [4] as well as Wen [72] have studied the multiple search problem in the PRAM model of computation. To the best of our knowledge, this important problem has found no solution in the context of meshes with multiple broadcasting.

Our algorithm runs in $O(\sqrt{m})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh with multiple broadcasting. We also show that in this model $\Omega(\sqrt{m})$ is a lower bound for the multiple search problem. An interesting feature of our algorithm is that for any given n and m $(1 \leq m \leq n)$, the running time is independent of n, thus the algorithm is adaptive. The remainder of the chapter is organized as follows: section 5.2 presents preliminaries for the proposed multiple search algorithm along with the preprocessing stage; section 5.3 discusses the details of the remaining stages of the algorithm; section 5.4 gives a number of surprising applications to problems in computer graphics, image processing, robotics and computational geometry; finally, section 5.5 summarizes the results, defines the generalized multiple search problem which can be solved by our algorithm, and poses a number of open problems.

5.2 The Algorithm - Preliminaries

Consider a $\sqrt{n} \times \sqrt{n}$ mesh $\mathcal{R}$ with multiple broadcasting. The elements of the sequence A are stored in $\mathcal{R}$, one item per processor, in row-major order. The queries q_j $(1 \leq j \leq m)$ are stored initially in the first $\frac{m}{\sqrt{n}}$ columns[1] of $\mathcal{R}$ one query per processor. The elements of sequence A will be referred to as *items*; the elements of sequence Q will be called *queries.*

For further reference, we note that since A is stored in row-major order, row i $(1 \leq i \leq \sqrt{n})$ of the mesh contains the items

$$a_{(i-1)\sqrt{n}+1}, a_{(i-1)\sqrt{n}+2}, \ldots, a_{i\sqrt{n}} \tag{5.1}$$

The *solution* of q_j is defined to be the item a_i such that $a_{i-1} \leq q_j < a_i$ $(1 \leq i \leq n)$ or if no such a_i exists the solution is taken to be $+\infty$. For each query q_j smaller than a_n we define its *row rank* to be the row in the mesh $\mathcal{R}$ that contains the solution of q_j, otherwise the row rank is $\sqrt{n} + 1$.

Let q be a query in Q whose row rank is r $(1 \leq r \leq \sqrt{n})$. We describe a simple data movement that allows us to solve the query in O(1) time. Begin by broadcasting q to all the processors in row r: this is accomplished by moving q vertically to row r, then horizontally to all processors in that row. Upon receiving q, every processor $P(r,j)$ $(1 \leq j \leq \sqrt{n})$ in row r compares q with the item $a_{(r-1)\sqrt{n}+j}$ it stores and marks itself 0 or 1 depending on whether or not $a_{(r-1)\sqrt{n}+j}$ is *strictly* larger than q. Let $P(r,k)$ be the leftmost processor in row r that is marked 0. Note that $P(r,k)$ always exists and can be identified in O(1) time by mandating every processor to check the mark of its left neighbor. Our marking scheme guarantees that $a_{(r-1)\sqrt{n}+k}$ is the solution of q. Finally, this solution can be moved in O(1) time to the original location of the query. This simple data movement will be used repeatedly in the remainder of the chapter without further explanation.

The algorithm proceeds in four stages: in Stage 0 the queries are moved into a square submesh of size $\sqrt{m} \times \sqrt{m}$; in Stage 1 we determine the row rank of every query in Q; Stage 2 has two goals: after sorting the queries by row rank we solve a subset of the queries and extract further information that will be used in Stage

[1]To avoid tedious but inconsequential details we assume that $\frac{m}{\sqrt{n}}$ is an integer.

3; finally, the goal of Stage 3 is to use the information computed in Stages 1 and 2 to solve all the remaining queries.

We perceive Stage 0 as a preprocessing stage and we proceed to show how the goal of this stage can be achieved. The details of subsequent stages will be given in the section 5.3 this chapter.

Stage 0

The purpose of this stage is to move the sequence Q of queries into the $\sqrt{m} \times \sqrt{m}$ submesh consisting of the intersection of the first $\sqrt{m}$ rows and columns of $\mathcal{R}$. Let B denote the submesh of $\mathcal{R}$ consisting of processors $P(i,j)$ with $\sqrt{m}+1 \leq i \leq \sqrt{n}$ and $1 \leq j \leq \frac{m}{\sqrt{n}}$; similarly, let C be the submesh of $\mathcal{R}$ consisting of processors $P(i',j')$ with $1 \leq i' \leq \sqrt{m}$ and $\frac{m}{\sqrt{n}} + 1 \leq j' \leq \sqrt{m}$ (see Figure 5.1).

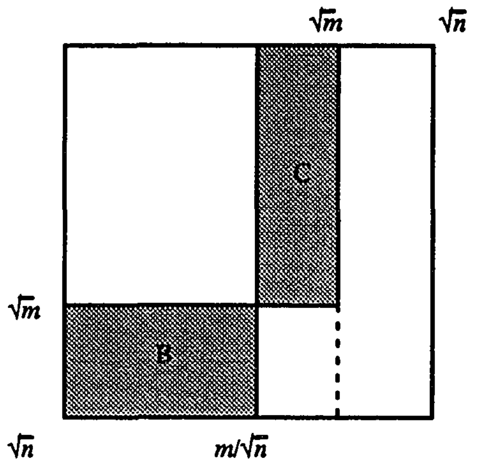


Figure 5.1: Stage 0 of multiple search

To achieve the goal of this stage we need only move the queries stored by processors in the submesh B to the processors in the submesh C, one query per processor. For this purpose, we handle the columns of B one by one, from left to right. To make our exposition more transparent and easier to follow, we assume that $\sqrt{m} \geq \sqrt{n} - \sqrt{m}$; the case $\sqrt{m} < \sqrt{n} - \sqrt{m}$ is handled analogously. We further assume that for some positive integer c, $\sqrt{m} = c(\sqrt{n} - \sqrt{m})$.

Our plan is to move the queries in every column j $(1 \leq j \leq \frac{m}{\sqrt{n}})$ of B to processors $P([(j-1) \bmod c](\sqrt{n}-\sqrt{m})+1, \frac{m}{\sqrt{n}} + \lceil \frac{i}{c} \rceil)$ through $P([(j-1) \bmod c + 1](\sqrt{n} - \sqrt{m}), \frac{m}{\sqrt{n}} + \lceil \frac{i}{c} \rceil)$ in column $\frac{m}{\sqrt{n}} + \lceil \frac{i}{c} \rceil$ of the submesh C.

We now give the details of this data movement. To begin, every processor $P(\sqrt{m} + i, j)$ $(1 \leq i \leq \sqrt{n} - \sqrt{m})$ broadcasts the query it holds horizontally to processor $P(\sqrt{m}+i, \frac{m}{\sqrt{n}}+i)$. In turn, processor $P(\sqrt{m}+i, \frac{m}{\sqrt{n}}+i)$ broadcasts the query received vertically to processor $P([(j-1) \bmod c](\sqrt{n}-\sqrt{m})+i, \frac{m}{\sqrt{n}}+i)$. Finally, every processor $P([(j-1) \bmod c](\sqrt{n}-\sqrt{m})+i, \frac{m}{\sqrt{n}}+i)$ broadcasts the query horizontally to processor $P([(j-1) \bmod c](\sqrt{n}-\sqrt{m})+i, \frac{m}{\sqrt{n}}+\lceil\frac{i}{c}\rceil)$.

To argue about the feasibility of this data movement, we only need observe that our initial assumption that $1 \leq m \leq n$ implies that

$$\frac{m}{\sqrt{n}} \leq \sqrt{m}. \tag{5.2}$$

It follows that all the subscripts used in the data movement are within the limits specified for $\mathcal{R}$.

Finally, (5.2) also guarantees that the data movement just described can be carried out in $O(\sqrt{m})$ time. We have proved the following result.

Lemma 5.2.1 The sequence Q of queries initially stored in the first $\frac{m}{\sqrt{n}}$ columns of $\mathcal{R}$ can be moved into a $\sqrt{m} \times \sqrt{m}$ submesh in $O(\sqrt{m})$ time. □

Remark: For further reference we note that if the output is assumed to take place along the first column of the mesh only, then after all the queries have been solved, we sort them by their initial position in the submesh $\mathcal{R}'$. Finally, we reverse the data movement described above and have the queries and their solutions moved to the first $\frac{m}{\sqrt{n}}$ columns of the mesh. From there, they can be output in $O(\sqrt{m})$ time.

5.3 The Algorithm - Details

It is important to remember that at the end of the preprocessing stage all the queries have been moved into the $\sqrt{m} \times \sqrt{m}$ submesh $\mathcal{R}'$ consisting of the intersection of the first $\sqrt{m}$ rows and columns of $\mathcal{R}$. In the remainder of this chapter, the columns of $\mathcal{R}'$ will be referred to as *query-columns*.

Stage 1.

The purpose of this stage is to determine the row rank of every query in Q. To accomplish the task specific to this stage, we let every processor $P(i, \sqrt{n})$ $(1 \leq i \leq \sqrt{n})$ broadcast the item $a_{i\sqrt{n}}$ horizontally to processor $P(i,i)$.

Next, $P(i,i)$ broadcasts the item $a_{i\sqrt{n}}$ vertically to the whole column i of the mesh. Note that (5.1) guarantees that as a result of this data movement, every processor in column i $(1 \leq i \leq \sqrt{n})$ becomes aware of the last item in row i.

To determine the row rank of every query in Q, the query-columns of $\mathcal{R}'$ are handled sequentially. More precisely, for every j $(1 \leq j \leq \sqrt{m})$, each query in column j of $\mathcal{R}'$ is broadcast horizontally along its row bus. For every i ($1 \leq i \leq \sqrt{m}$), every processor $P(i,k)$ in row i of $\mathcal{R}$ compares the query it receives with $a_{k\sqrt{n}}$ (i.e., the last item in row k), and marks itself 0 or 1 depending on whether or not $a_{k\sqrt{n}}$ is strictly larger than the query.

In case processor $P(i,\sqrt{n})$ is marked 1, the row rank of the corresponding query is $\sqrt{n}+1$ and $P(i,\sqrt{n})$ will broadcast this information to $P(i,j)$ (i.e. the processor that has broadcast the query). Otherwise, let $P(i,k)$ be the leftmost processor in row i that is marked 0. Now it is easy to confirm that k must be the row rank of the query and $P(i,k)$ sends the appropriate information to processor $P(i,j)$.

Note that computing the row ranks of queries in every query-column takes O(1) time using broadcasting on buses. Since there are $\sqrt{m}$ query-columns altogether, we have the following result.

Lemma 5.3.1 The row ranks of all queries in Q can be determined in $O(\sqrt{m})$ time. □

Stage 2

Stage 2 involves performing two basic tasks: the first one involves solving a part of the queries in Q; the second involves further processing the query-submesh $\mathcal{R}'$ to extract information that will be needed to solve the remaining queries in the next stage.

To begin, using an optimal sorting algorithm for meshes [44, 65], the sequence Q of queries is sorted in column-major order by row rank.

A sorted query-column of $\mathcal{R}'$ is called *pure* if all the queries in the column have the same row rank. Otherwise, the query-column is termed *impure*. We begin by identifying every query-column as pure or impure. For this purpose, every processor $P(\sqrt{m},j)$ ($1 \leq j \leq \sqrt{m}$) broadcasts the row rank of the query it holds vertically to $P(1,j)$. Now $P(1,j)$ has enough information to determine whether column j is pure or impure.

First, we note that the solution is $+\infty$ for all queries whose row rank is $\sqrt{n}+1$. Next, we process pure query-columns with row ranks at most $\sqrt{n}$, one by one. Specifically, let query-column j be pure and assume that the row rank of every query in j is r. Using column buses, row r is replicated through the entire mesh $\mathcal{R}$. Further, each query in column j is broadcast horizontally using its row bus. Note that in every row i ($1 \leq i \leq \sqrt{m}$) of $\mathcal{R}$ there exists precisely one processor that determines and returns the solution of the corresponding query.

Impure query-columns of $\mathcal{R}'$ are handled differently. In preparation for this, the second task of Stage 2 is to determine which rows of the mesh $\mathcal{R}$ are dense and which ones are sparse as we are about to define.

A row r of $\mathcal{R}$ is said to be *sparse* if the number of queries in impure columns of $\mathcal{R}'$ whose row rank is r is not greater than $\sqrt{m}$. Otherwise, row r is termed *dense*. To complete Stage 2, we identify sparse and dense rows of $\mathcal{R}$; in addition, we build a linked list consisting of the dense rows.

Let j be an arbitrary impure query-column of $\mathcal{R}'$ and let $r_1, r_2, \ldots, r_t$ $(t \geq 2)$ be the row ranks of the queries in column j. Since the sequence of queries was sorted in column-major order, the queries having the same row rank occur consecutively in j. For further reference, such a set of queries is termed a *run*. It is important to note that for any r $(1 \leq r \leq \sqrt{n})$, at most two impure query-columns contain queries whose row rank is r. Note further that if $t \geq 3$ then all rows $r_2, \ldots, r_{t-1}$ must be sparse, since queries having such row ranks cannot occur in a different query-column.

Consequently, any potential dense row r must straddle two impure columns. By the above observation, a dense row can only correspond to the bottomost run in some impure column and the topmost run in the closest impure column to its right. It is an easy matter to compute the number of queries in every such run, and using column buses to send to processor $P(1,j)$ in every impure column j ordered pairs consisting of the row number and the number of queries in the topmost and bottomost runs in column j.

Finally, traversing the first row of $\mathcal{R}'$ sequentially, dense rows can be identified and a linked list containing all the dense rows in increasing order can be built: in fact, we only need inform every dense row of the identity of the next dense row in $\mathcal{R}$.

To argue about the complexity of Stage 2, we note that solving queries whose row rank is $\sqrt{n}+1$ takes O(1) time; the task of identifying pure and impure columns takes O(1) time. Similarly, solving the queries in a pure column involves only broadcasting and is done in O(1) time. Since there are at most $O(\sqrt{m})$ pure columns solving the queries in these columns takes at most $O(\sqrt{m})$ time. Similarly, it is easy to see that the second task of Stage 2 can be done in $O(\sqrt{m})$ time.

Consequently, we have proved the following result.

Lemma 5.3.2 The task of solving all the queries with row ranks $\sqrt{n}+1$ and those in pure columns of $\mathcal{R}'$, as well as that of identifying sparse and dense rows of $\mathcal{R}$ can be performed in $O(\sqrt{m})$ time. □

Stage 3. The goal of this stage is to solve the remaining queries in Q. In the first step, every query in an impure column is moved to the row of the mesh that equals its row rank. Let j be an impure query-column. To make our exposition more transparent, we now describe the data movement involving a query q stored by processor $P(i,j)$. Assume, without loss of generality that the row rank of q is r. To begin, processor $P(i,j)$ broadcasts the query q along with its row rank

r to processor $P(i,i)$ using the bus in row i. Next, using the bus in column i, processor $P(i,i)$ sends q to processor $P(r,i)$. Note that as a consequence of this data movement, every query in column j of $\mathcal{R}'$ is sent to the row of the mesh that equals its row rank. We note that each processor $P(r,i)$ receives at most two queries. Once the queries are solved, as we are about to explain, by reversing the previous data movement, the solution of q is returned to processor $P(i,j)$ in $\mathcal{R}'$.

Our algorithm proceeds by first solving all queries in sparse rows, and then in all the dense rows. The details are spelled out as follows. In each sparse row, the solution for each query is determined one by one, by broadcasting its value across the row, and having a unique processor identify and return the corresponding solution.

Let r be a dense row of $\mathcal{R}$. Begin by replicating row r throughout the mesh $\mathcal{R}$ by using vertical buses. Note that every processor in row r will broadcast the item and any possible queries it holds. Consider the diagonal processors $P(i,i)$ of the mesh. By a previous observation, each of them contains at most two queries. For all values of i $(1 \leq i \leq \sqrt{n})$ if processor $P(i,i)$ contains queries it will use row i to solve them in $O(1)$ time as described in section 2, and will return the solutions to row r.

Since there are at most $\sqrt{m}$ queries in each sparse row and there are at most $\sqrt{m}$ dense rows in the linked list of dense rows previously constructed, it is easy to see that the computation in Stage 3 can be carried out in $O(\sqrt{m})$ time. We have proved the following result.

Lemma 5.3.3 The task of solving all the queries in impure columns of $\mathcal{R}'$ can be performed in $O(\sqrt{m})$ time. □

We now establish the complexity and time-optimality of our algorithm for the multiple search problem.

Theorem 5.3.4 Given a sorted sequences $A = a_1, a_2, ..., a_n$ of items from a totally ordered universe and a sequence $Q = q_1, q_2, ..., q_m$ $(1 \leq m \leq n)$ of queries, the corresponding multiple search problem can be solved in $O(\sqrt{m})$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. Furthermore, this is time-optimal.

Proof. The correctness of the algorithm is easily seen; by Lemmas 5.2.1, 5.3.1, 5.3.2, and 5.3.3 combined, the running time is bounded by $O(\sqrt{m})$.

To argue for the time optimality, we show that every algorithm that solves the multiple search problem must take $O(\sqrt{m})$ time in the worst case. For this purpose we assume that the sorted sequence A consists of distinct items and is stored in row-major order in a $\sqrt{n} \times \sqrt{n}$ mesh $\mathcal{R}$ with multiple broadcasting.

Let $\mathcal{S}$ be a $\frac{\sqrt{m}}{2} \times \frac{\sqrt{m}}{2}$ submesh of $\mathcal{R}$ consisting of the processors $P(i,j)$ with $1 \leq$

$i \leq \frac{\sqrt{m}}{2}$ and $\sqrt{n} - \frac{\sqrt{m}}{2} + 1 \leq j \leq \sqrt{n}$, containing a subsequence $A' = a'_1, a'_2, \ldots, a'_{\frac{m}{4}}$ of A (see Figure 5.2).

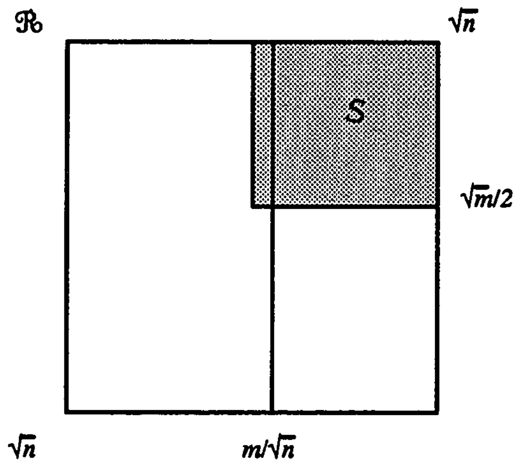


Figure 5.2: Lower bound argument for multiple search

The queries are assumed to be in the first $\frac{m}{\sqrt{n}}$ columns of $\mathcal{R}$. We construct the sequence $Q = q_1, q_2, \ldots, q_m$ $(1 \leq m \leq n)$ of queries satisfying the following conditions:

- for a subsequence $Q' = q_{i_1}, q_{i_2}, \ldots, q_{i_{\frac{m}{4}}}$ of Q, the solution of q_{i_j} $(1 \leq j \leq \frac{m}{4})$ is a'_j;
- every query in Q but not in Q' is larger than a_n;
- all the queries in Q' are stored by processors outside $\mathcal{S}$.

It is easy to see that such a sequence Q of queries can be constructed; to clarify the feasibility of the last constraint, note that since the processors in $\mathcal{S}$ contain at most $\frac{m}{4}$ queries, at least $\frac{3m}{4}$ of the queries are stored by processors outside $\mathcal{S}$.

Our proof of the lower bound relies on a very simple information transfer argument. Note that the submesh $\mathcal{S}$ is connected by $O(\sqrt{m})$ buses and local links to the remainder of $\mathcal{R}$. It is important to note that, as a consequence, in one time unit at most $O(\sqrt{m})$ pieces of information (queries and/or items in A') can "cross" the boundary between $\mathcal{S}$ and the remainder of $\mathcal{R}$. In addition, our construction of the sequence Q guarantees that to solve the queries q_i with $1 \leq i \leq \frac{m}{4}$, $O(m)$ pieces of information must cross the boundary of $\mathcal{S}$. Consequently, $\Omega(\sqrt{m})$ time is needed to solve all the queries, completing the proof of the theorem. □

5.4 Applications

The purpose of this section is to show that the multiple search problem finds a number of surprising applications to problems in computer graphics, image processing, robotics, and computational geometry.

Throughout the remainder of this chapter we assume an underlying convex polygon $P = p_1, p_2, \ldots, p_n$ in standard form (see [3, 57] for details), stored in row-major order in a $\sqrt{n} \times \sqrt{n}$ mesh $\mathcal{R}$ with multiple broadcasting.

Let $Q=q_1, q_2, \ldots, q_m$ $(1 \leq m \leq n)$ be a sequence of points in the plane. The *multiple point inclusion* problem asks for determining for every subscript j $(1 \leq j \leq m)$ whether the query point q_j is inside P. Just as the point inclusion problem, the multiple point inclusion problem finds application to distributed database design, computer graphics, and image processing [11, 38, 57].

As it turns out, the problem at hand can be solved by reducing it to the multiple search problem. To begin, choose an arbitrary point ω inside P and convert the vertices of P as well as the query points in Q to polar coordinates with pole ω and polar axis ωp_1.

It is well-known that the vertices of P occur in sorted angular order about ω [57]. Now solving the corresponding instance of the multiple search problem, we determine for every query point q_j the wedge $p_{i-1}\omega p_i$ within which q_j lies. Finally, in one more comparison, it can be decided whether p_j and ω lie on the same side of the segment $p_{i-1}p_i$. Consequently, we have the following result.

Theorem 5.4.1 Given a convex polygon $P = p_1, p_2, \ldots, p_n$ and a sequence $Q=q_1, q_2, \ldots, q_m$ $(1 \leq m \leq n)$ of points in the plane, the multiple point inclusion problem can be solved in $O(\sqrt{m})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh with multiple broadcasting. □

Note that Theorem 5.4.1 holds for star-shaped polygons, provided that a point ω in the kernel is known.

A related problem arising in computer graphics and image processing is known as the *convex polygon containment* problem. Here, we are given convex polygons $P = p_1, p_2, \ldots, p_n$ and $Q = q_1, q_2, \ldots, q_m$ $(1 \leq m \leq n)$ and we are interested in finding out whether Q is contained in P. Note that this problem can be easily solved by reducing it to the multiple point inclusion problem. We have the following simple corollary of Theorem 5.4.1.

Corollary 5.4.2. The convex polygon containment problem for $P = p_1, p_2, \ldots, p_n$ and $Q = q_1, q_2, \ldots, q_m$ $(1 \leq m \leq n)$ can be solved in $O(\sqrt{m})$ time on a $\sqrt{n} \times \sqrt{n}$ enhanced mesh. □

Next, we address a problem that arises frequently in computer graphics, image processing, and path planning. We are given a convex polygon $P = p_1, p_2, \ldots, p_n$ and a sequence $Q=q_1, q_2, \ldots, q_m$ $(1 \leq m \leq n)$ of points in the plane. For a point q_j *exterior* to P the two supporting rays for P emanating from q_j are referred to as *right* and *left* depending on whether the interior of P lies to the left or to the right of the ray. Our task is to compute for every point q_j exterior to P the corresponding left and right rays. For definiteness, we refer to this as the *multiple ray* problem. Surprisingly, the solution to the multiple ray problem reduces to a variant of the multiple search problem.

Recall that, by assumption, the polygon P is stored in row-major order in a $\sqrt{n} \times \sqrt{n}$ mesh $\mathcal{R}$ with multiple broadcasting. We assume that the points in Q are stored in the $\sqrt{m} \times \sqrt{m}$ submesh $\mathcal{R}'$ consisting of the first $\sqrt{m}$ rows and columns of $\mathcal{R}$.

We begin by using the solution to the multiple point inclusion problem to determine which points of Q are exterior to P. We assume, without loss of generality, that all vertices of Q are exterior to P. We now show how to reduce the problem of computing the *left* supporting ray for every point in Q to a variant of the multiple search problem. (Handling right supporting rays is done by a symmetric argument.)

For this purpose, consider the convex polygon $P' = p_{\sqrt{n}}, p_{2\sqrt{n}}, \ldots, p_n$ consisting of the vertices of P whose subscripts are multiples of $\sqrt{n}$ (see Figure 5.3). Note that P' partitions the boundary of P into chains $C_1, C_2, \ldots, C_{\sqrt{n}}$ such that $C_i = p_{(i-1)\sqrt{n}+1}, p_{(i-1)\sqrt{n}+2}, \ldots, p_{i\sqrt{n}}$ $(1 \leq i \leq \sqrt{n})$.

It is easy to see that the vertices of P' are stored by the processors in the last column of $\mathcal{R}$, and that every chain C_i $(1 \leq i \leq \sqrt{n})$ defined above involves points stored by processors in row i of the mesh.

Our algorithm proceeds in two stages. In Stage 1, we solve the multiple ray problem for Q and P'; in Stage 2, we extend the solution to P. We note that the instance of the multiple ray problem involving P' and Q can be solved in $O(\sqrt{m})$ time. As in the multiple search problem, we let every processor $P(i, \sqrt{n})$ holding $p_{i\sqrt{n}}$ broadcast the point horizontally to processor $P(i,i)$ which, in turn, broadcasts $p_{i\sqrt{n}}$ to the entire column i of the mesh. Next, the columns of $\mathcal{R}'$ are processed sequentially as follows.

To process query-column j $(1 \leq j \leq \sqrt{m})$, each point in that column is broadcast horizontally to all the processors in its own row. To make the exposition easier to follow, consider a generic point q in Q stored in some processor $P(i,j)$ of $\mathcal{R}'$. Note that exactly *two* processors in row i determine that the ray emanating from q and passing through the points of P' they contain, are supporting rays for P'. However, only one of them detects that the corresponding ray is a left ray. The processor in row i that detects this condition broadcasts its identity to processor $P(i,j)$.

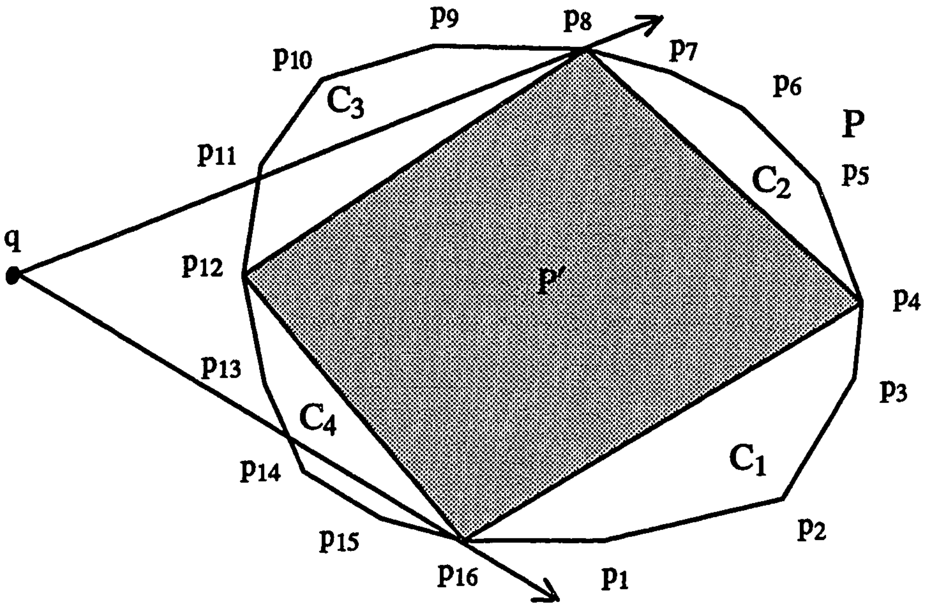


Figure 5.3: Stage 1 of the multiple ray problem

Observe that if the left supporting ray for P' determined by some point q in Q and some point $p_{t\sqrt{n}}$ is a supporting ray for P, then no further action is needed. Otherwise, it is easy to confirm that the ray $qp_{t\sqrt{n}}$ intersects *precisely* one of the chains C_{t-1} or C_t. Furthermore, the chain intersected by the ray $qp_{t\sqrt{n}}$ can be determined in O(1) time by checking the edges of P incident to $p_{t\sqrt{n}}$ (see Figure 5.4).

Consider points q in Q for which the left supporting ray for P' is not a supporting ray for P. For every such point q, we define its ***chain rank*** to be the subscript of the chain which the left supporting ray to P' from q intersects.

Further, we sort the points in Q in column-major order by their chain ranks. Assume without loss of generality that the chain rank of q is t. It is now easy to confirm that in order to find a left supporting ray for P emanating from q we only need find a left supporting ray for the convex polygon determined by $p_{(t-1)\sqrt{n}}, p_{(t-1)\sqrt{n}+1}, \ldots, p_{t\sqrt{n}}$. This can be done in $O(\sqrt{m})$ time by a slight modification of the Stages 2 and 3. Consequently, we have the following result.

Theorem 5.4.3. Given a convex polygon $P = p_1, p_2, \ldots, p_n$ and a sequence $Q=q_1, q_2, \ldots, q_m$ $(1 \leq m \leq n)$ of points in the plane, the corresponding multiple ray problem can be solved in $O(\sqrt{m})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh with multiple broadcasting. □

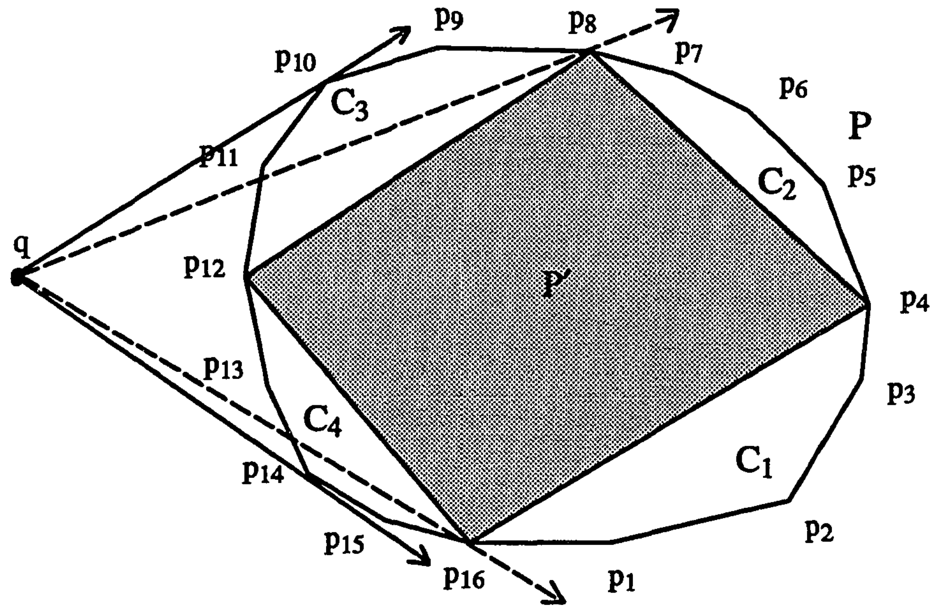


Figure 5.4: Stage 2 of the multiple ray problem

As a further application, consider the following problem that arises frequently in computer graphics, robotics, and image processing. A convex polygon $P = p_1$, p_2, ..., p_n is given along with a sequence $L=l_1, l_2, \ldots, l_m$ $(1 \leq m \leq n)$ of lines in the plane. The *multiple stabbing* problem involves answering queries of the type "does line l_j intersect P?".

As we are about to explain, the multiple stabbing problem reduces to the multiple ray problem. To begin, for every j $(1 \leq j \leq m)$ choose a point q_j on the line l_j. What results is a sequence $Q=q_1, q_2, \ldots, q_m$ $(1 \leq m \leq n)$ of points in the plane. In $O(\sqrt{m})$ time we solve the multiple point inclusion problem.

Clearly, if some point q_j is inside P then the line l_j intersects the polygon. For the points in Q that are outside P we solve the multiple ray problem. Finally, we determine whether l_j is within the wedge bounded by the left and right supporting rays at q_j. We have proved the following result.

Theorem 5.4.4. Given a convex polygon $P = p_1$, p_2, ..., p_n and a sequence $L=l_1, l_2, \ldots, l_m$ $(1 \leq m \leq n)$ of lines in the plane, the corresponding multiple stabbing problem can be solved in $O(\sqrt{m})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh with multiple broadcasting. □

Finally, consider again two convex polygons $P = p_1$, p_2, ..., p_n and $Q =$

$q_1, q_2, \ldots, q_m$ $(1 \leq m \leq n)$ both in standard form. An important problem in computational geometry is to determine whether P and Q are linearly separable and, if so, to construct a line that separates them.

As it turns out, the solution to the multiple ray problem affords us an efficient solution to the problem at hand. Specifically, assume that a solution to the multiple ray problem is available. For definiteness, we assume that the polygon Q is stored in column-major in the submesh $\mathcal{R}'$ consisting of the first $\sqrt{m}$ rows and columns of $\mathcal{R}$. Note that the polygons are separable if and only if some common supporting line for the two polygons is separating.

We proceed as follows: every processor in $\mathcal{R}'$ determines whether any of its supporting rays for P is also a supporting ray for Q. This, of course can be done in O(1) time by verifying whether or not the left and right neighbors of a vertex in Q are to the same side of the supporting ray. Note that either the polygons are found not to be separable or else rays that are common supporting lines for P and Q will be identified in the process described above. It is now an easy matter to identify those supporting rays that are separating. Therefore, we have the following result.

Theorem 5.4.5. Given convex polygons $P = p_1, p_2, \ldots, p_n$ and $Q = q_1, q_2, \ldots, q_m$ $(1 \leq m \leq n)$ in the plane, the task of determining whether P and Q are linearly separable can be performed in $O(\sqrt{m})$ time on a $\sqrt{n} \times \sqrt{n}$ enhanced mesh. Furthermore, in case the polygons are separable a separating line can be identified in the same time bounds. □

Note also that with a trivial modification, the algorithm for computing a separating line for two convex polygons can determine common tangents in case neither polygon contains the other. The following result summarizes this finding.

Corollary 5.4.6 Given convex polygons $P = p_1, p_2, \ldots, p_n$ and $Q=q_1, q_2, \ldots, q_m$ $(1 \leq m \leq n)$ in the plane, the common tangents to P and Q, if any, can be computed in $O(\sqrt{m})$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. □

5.5 Conclusions

Given a sorted sequence $A = a_1, a_2, \ldots, a_n$ of items from a totally ordered universe, along with an arbitrary sequence $Q = q_1, q_2, \ldots, q_m$ $(1 \leq m \leq n)$ of *queries*, the *multiple search* problem involves computing for every q_j $(1 \leq j \leq m)$ the unique a_i for which $a_{i-1} \leq q_j < a_i$. In this chapter we have proposed a time-optimal algorithm to solve the multiple search problem on meshes with multiple broadcasting. More specifically, our algorithm runs in $O(\sqrt{m})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh with multiple broadcasting. We also showed that this achieves the theoretical lower bound for the problem. Note that the running time of our algorithm is independent of n, and thus the algorithm is adaptive.

We have also shown that the multiple search problem finds surprising applications to computer graphics, image processing, robotics, and computational geometry. At this time it is not known whether these solutions are time-optimal.

As it turns out, our algorithm for the multiple search problem works for a more general problem that we are about to define. For this purpose, consider a sequence $A = a_1, a_2, \ldots, a_n$ of items and a sequence $Q = q_1, q_2, \ldots, q_m$ $(1 \leq m \leq n)$ of queries such that every ordered pair (q_j, a_i) $(1 \leq i \leq n; 1 \leq j \leq m)$ forms a decision problem.

We assume that this collection of decision problems satisfies at least one of the following:

- For each (q_j, a_i) $(1 \leq i \leq n; 1 \leq j \leq m)$, if the answer is "yes", then the answer is "yes" for all (q_j, a_k) $(i \leq k \leq n)$;
- For each q_j $(1 \leq j \leq m)$, there exists at most one value of i $(1 \leq i \leq n)$ such that the answer to (q_j, a_i) is "yes".

For each q_j $(1 \leq j \leq m)$, the corresponding *solution* is either the item a_i with minimal i for which the answer to (q_j, a_i) is "yes" or if no such i exists the solution is $+\infty$. Now the *generalized multiple search* problem asks for the solution of every query in Q. The algorithm which we have proposed to solve the multiple search problem on a $\sqrt{n} \times \sqrt{n}$ enhanced mesh can be easily modified to solve the generalized multiple search problem in $O(\sqrt{m})$ time, provided every decision problem (q_j, a_i) $(1 \leq i \leq n; 1 \leq j \leq m)$ can be answered by a single processor in $O(1)$ time using only local information.

In this chapter we have solved some instances of the generalized multiple search problem. It would be interesting to see what other problems can be solved by the same paradigm. This promises to be a challenging area for future research.

Chapter 6

Convexity Problems

6.1 Introduction

Convexity and related computations are a recurring theme in pattern recognition, image processing, computer vision, operations research, robotics, computational geometry, and computational morphology. In pattern recognition, for instance, convexity appears in clustering, and computing similarities between sets [7]. In image processing and computer vision convexity is used as a natural shape descriptor and classifier for objects in the image space [2,20,22]. In operations research convexity is a fundamental tool in linear programming and convex analysis [66]. In robot navigation, one of the fundamental heuristics involves approximating real-world objects by convex sets [41]. In computational geometry, convexity is often a valuable tool in devising efficient algorithms for a number of seemingly unrelated problems [57, 66]. In computational morphology, convexity has played a central role in analyzing relevant features of the shape of a set of points [68]. Further, one of the fundamental features that contributes to a morphological description useful in shape analysis is the Euclidian distance function among vertices of the polygon [68].

The purpose of this chapter is to study a number of convexity-related problems on meshes with multiple broadcasting. First, given an n-vertex convex polygon P, we address the problems of:

- computing the area of P,
- computing the diameter of P,
- computing the width of P,
- computing the modality of P,
- computing a smallest-area rectangle enclosing P, and
- computing the largest-area inscribed triangle sharing an edge with P.

Next, given two n-vertex convex polygons P and Q we address the problems of:

- detecting whether P lies in the interior of Q,
- computing the largest distance between P and Q,
- computing the smallest distance between P and Q, assuming that the two polygons are separable.

These tasks are motivated by, and find applications to, problems in pattern recognition, computer graphics, computational morphology, image processing, computer vision, and VLSI design. Some examples follow. The diameter of a convex polygon is of import in clustering [9, 27, 58], computer graphics [53], path planning [41, 66], and in a number of facility location problems [57]. The smallest area enclosing rectangle arise in image processing [58] as well as in the compaction process in VLSI [54]. The problem of computing the largest vertex distance between two convex polygons arises in clustering [9, 27, 58], computer graphics [53], and image understanding [9, 71]. The problems of computing the width of a convex polygon and smallest distance between two convex polygons are central to path planning [41], morphology [68], and in a number of facility location problems [57].

One of the contributions of this thesis is to show that $\Omega(\log n)$ is a lower bound for any instance of size n of the problems mentioned above (with the exception of the smallest distance between two convex polygons) both in the CREW-PRAM and in the mesh with multiple broadcasting, regardless of the number of processors and memory cells used. We prove that these lower bounds are tight by providing $O(\log n)$ time algorithms to solve these problems on a mesh with multiple broadcasting of size $n \times n$. We also show that the task of computing the smallest distance between two convex polygons can be computed in $O(1)$ time. This result is likely to bring some aspects of robot path-planning within the realm of real-time computations.

The remainder of the chapter is organized as follows: section 6.2 presents a number of basic results that will be useful in the design of our algorithms; section 6.3 discusses our lower bound results; section 6.4 presents algorithms for problems involving one convex polygon; section 6.5 proposes algorithms for computational problems involving two polygons; finally, section 6.6 summarizes our findings and proposes a number of open questions.

6.2 Preliminaries

The purpose of this section is to review a number of fundamental geometric definitions and concepts along with data movement techniques for implementing basic algorithms on meshes with multiple broadcasting.

Specifying an n-vertex polygon P in the plane amounts to enumerating its vertices in *clockwise* order as $p_1, p_2, \ldots, p_n$ $(n \geq 3)$, in such a way that p_ip_{i+1} $(1 \leq i \leq n-1)$ and p_np_1 define the edges of P. This representation is also known as *vertex* representation of P. We note that the vertex representation of a polygon can be easily converted into an *edge* representation: namely, P is represented by a sequence $e_1, e_2, \ldots, e_n$ of edges, specified in clockwise order, with e_i $(1 \leq i \leq n-1)$ having p_i and p_{i+1} as its endpoints, and e_n having p_n and p_1 as its endpoints. To avoid tedious but inconsequential details, in our algorithms we assume that all our polygons have distinct vertices with no three vertices collinear.

A polygon P is termed *simple* if no two of its non-consecutive edges intersect. Recall that Jordan's Curve Theorem [57] guarantees that a simple polygon partitions the plane into two disjoint regions, the *interior* (bounded) and the *exterior* (unbounded) that are separated by the *boundary* of the polygon. A simple polygon is *convex* if its interior is a convex set [57]. The *diameter* of a convex polygon [57] is the largest Euclidian distance between any pair of its vertices. The *width* of a convex polygon [32] is the least distance between pairs of antipodal vertices. A vertex p_i of a polygon P is termed *unimodal* with respect to the Euclidian distance to the remaining vertices, if there exists a subscript j $(1 \leq j \leq n)$ such that $\mathrm{d}(p_i, p_k)$ is non-decreasing for $k = i+1, i+2, \ldots, j$ and non-increasing for $k = j+1, j+2, \ldots, i-1$[1]. More generally, the *modality* of a vertex u of a polygon P (with respect to the Euclidian distance) [6, 67] is defined to be the number of maxima of the Euclidian distance function $\mathrm{d}(u, v)$ when v traverses the boundary of P. The *modality* of a polygon is defined as the sum of the modalities of its vertices.

Recently, Toussaint [67] pointed out that the notions of convexity and unimodality are quite different: convex polygons need not be unimodal, and unimodal polygons need not be convex. Furthermore, Toussaint [67, 68] argues that unimodality is one of the key factors in obtaining very efficient algorithms for a large number of problems in computational morphology and pattern recognition. It is not surprising, therefore, that unimodality and multimodality have received considerable attention in the literature [6, 46, 47, 67, 68]. In particular, Aggarwal and Melville [6] have obtained a linear time sequential algorithm to determine whether a convex polygon is unimodal.

One of the fundamental heuristics in pattern recognition, image processing, and robot navigation, involves approximating real-world objects by convex sets. For obvious reasons, one is typically interested in the smallest convex region that contains some object in the space of interest. The convex hull of a set of points

[1]Throughout this chapter, subscript arithmetic is modulo n.

in the plane is defined as the smallest convex set that contains the original set [57, 66]. Our arguments rely, in part, on the following recent result proved in [49].

Proposition 6.2.1 [49] The convex hull of planar set of n points stored in the first row of a mesh with multiple broadcasting of size $n \times n$ can be computed in $O(\log n)$ time. Furthermore, this is time-optimal □

A fundamental geometric problem [57] is referred to as *point inclusion.* The problem can be stated as follows: given a convex polygon $P = p_1, p_2, \ldots, p_n$ and a point q in the plane, does q lie inside P? For definiteness, we assume that P is stored one vertex per processor in row i of a mesh with multiple broadcasting of size $n \times n$, and that q is stored by $P(i, 1)$. We propose to show that the point inclusion problem for q can be solved in $O(1)$ time, using the processors in row i only.

Let ω be a point inside P, obtained by computing the centroid of three vertices of P. In $O(1)$ time partition the plane into n wedges each of the form $p_i \omega p_{i+1}$ by broadcasting the coordinates of ω to all the processors in row i. Clearly, q belongs to exactly one of these wedges. In addition, this particular wedge, say $p_k \omega p_{k+1}$, can be determined in $O(1)$ time by simply broadcasting the coordinates of q to all the processors in row i. Finally, by checking q against the edge $p_k p_{k+1}$ we determine whether or not q lies in the interior of P. To summarize our discussion we state the following result.

Lemma 6.2.2 Let $P = p_1, p_2, \ldots, p_n$ be a convex polygon stored in one row of a mesh with multiple broadcasting of size $n \times n$, one vertex per processor, and let q be an arbitrary point in the plane. The point inclusion problem for q can be solved in $O(1)$ time using the processors in this row only. □

For further reference we now state and solve the following problem which is of independent interest. Consider a circle $\mathcal{C}$ in the plane and n equally spaced points on the boundary of $\mathcal{C}$, numbered for convenience as $1, 2, \ldots, n$ in clockwise order, and refer to Figure 6.1.

Let $C = \{C_i = [a_i, b_i] \mid 1 \leq i \leq n\}$ be a family of circular arcs in $\mathcal{C}$ such that:

- the endpoints of every circular arc are integers in the range $\{1, 2, \ldots, n\}$;
- the endpoints of every arc are specified in clockwise order;
- no arc is self-overlapping, that is, the angle subtended by every arc is less than 2π.

The family C is stored one arc per processor in the first row of a mesh with multiple broadcasting of size $n \times n$. The *point overlap* problem asks to determine for every integer i $(1 \leq i \leq n)$ the number of intervals in which it appears.

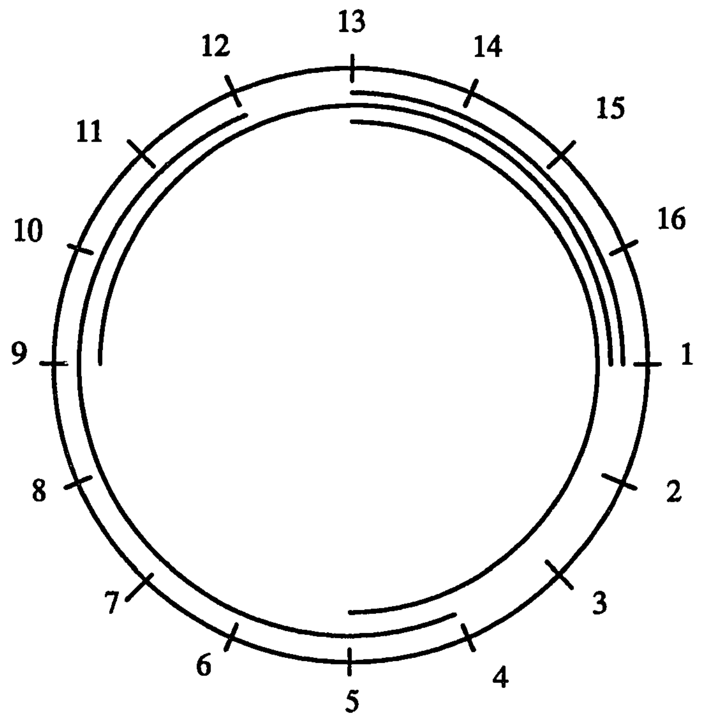


Figure 6.1: An instance of the point overlap problem

First, we convert the family C of circular arcs to a family of half-closed intervals in the following natural way. Every circular arc $[a_i,b_i]$ for which $1 \leq a_i \leq b_i \leq n$ will be regarded as the interval $[a_i,b_i+1)$ (in case $b_i = n$ we take $b_i+1 = n+1$). Every circular arc $[a_i,b_i]$ which contains the arc $[n,1]$ will be replaced by the intervals $[a_i,n+1)$ and $[1,b_i+1)$. Note that the number of intervals thus obtained is at most $2n$.

It is easy to see that for every i $(1 \leq i \leq n)$, the number of circular arcs in which it appears is exactly the number of intervals that begin before or at i minus the number of intervals that end before or at i. This observation suggests the following simple algorithm.

Step 1. Determine for every i $(1 \leq i \leq n)$ the number d_i of intervals that begin at i.

Step 2. Compute the prefix sums of the sequence $d_1, d_2, \ldots, d_n$, and let the result be $e_1, e_2, \ldots, e_n$.

Step 3. Determine for every i $(1 \leq i \leq n)$ the number c_i of intervals that end at

i, that is the number of intervals of the form $[j, i)$.
Step 4. Compute the prefix sums of the sequence $c_1, c_2, \ldots, c_n$, and let the result be $f_1, f_2, \ldots, f_n$.
Step 5. For every i $(1 \leq i \leq n)$ compute $e_i - f_i$.

We now show how the above steps can be implemented in $O(\log n)$ time on a mesh with multiple broadcasting. As noted before, the collection of circular arcs has been converted into a family of at most $2n$ intervals. These intervals are stored, at most two per processor, in the first row of a mesh with multiple broadcasting of size $n \times n$. First, we sort all the left-endpoints in increasing order. By Corollary 3.4, this operation takes $O(\log n)$ time. Once this is done, we process the left-endpoints in two similar stages. In the first stage we process the first n left-endpoints in sorted order and in the second stage the remaining (at most n) left-endpoints. Assume that the first n left-endpoints are stored in left-to-right order by the processors in the first row of the mesh.

Using vertical buses, replicate the contents of the first row to all the rows of the mesh. In every row i $(1 \leq i \leq n)$ the processors that store a left-endpoint whose value is i occur consecutively. By checking the values stored by their immediate neighbors, the first and last processor in row i storing the value i are identified in $O(1)$ time. In two more broadcasts, these processors send their column numbers to processor $P(i,i)$. In turn, $P(i,i)$ computes the number d_i of intervals that begin at i. This information is then broadcast to processor $P(1,i)$.

Using the prefix sums algorithm of Proposition 3.0, the prefix sums of $d_1, d_2, \ldots, d_n$ is computed in $O(\log n)$ time. The computation of the number c_i of intervals ending at i $(1 \leq i \leq n)$, as well as the computation of the prefix sums $f_1, f_2, \ldots, f_n$ is carried out in a perfectly similar way. To summarize our findings we state the following result.

Lemma 6.2.3 The point overlap problem of n arcs stored one per processor in the first row of a mesh with multiple broadcasting of size $n \times n$ can be solved in $O(\log n)$ time. □

6.3 Lower Bounds

The purpose of this section is to derive lower bounds for the following problems.
AREA: given an n-vertex convex polygon P, compute its area.
DIAMETER: given an n-vertex convex polygon P, compute its diameter.
WIDTH: given an n-vertex convex polygon P, compute its width.
MODALITY: given an n-vertex convex polygon P, compute its modality.

ENCLOSING RECTANGLE: given an n-vertex convex polygon P, determine an enclosing rectangle of minimum area.
INSCRIBED TRIANGLE: given an n-vertex convex polygon P, determine an inscribed triangle of maximum area sharing an edge with P.
MAX DISTANCE: given two n-vertex convex polygons P and Q, compute the largest Euclidian distance between a point on P and a point on Q.
CONTAINMENT: given two n-vertex convex polygons P and Q, is every vertex of P contained in the interior of Q?

Our optimality arguments will be stated first in the Parallel Random Access Machine model (PRAM, for short). This is based on Propositions 2.3.1 and 2.3.2 of Chapter 2. We first show that the time lower bound for AREA, DIAMETER, WIDTH, MODALITY, ENCLOSING RECTANGLE, and INSCRIBED TRIANGLE is $\Omega(\log n)$ on the CREW-PRAM, by reducing the OR problem to each of these problems. In all the derivations in this section we use polar coordinates for convenience[2]. Before proving time lower bounds, we show a way of associating with an arbitrary n-bit sequence a unique n-vertex convex polygon. We shall refer to this as the *standard* construction, as all our subsequent constructions are variations thereof.

For this purpose let $b_1, b_2, \ldots, b_n$, be an arbitrary bit sequence. Consider the unit circle C centered at ω and let $u_1, u_2, \ldots, u_n$ be equally spaced points on the boundary of C. Further, let δ be a number satisfying

$$\cos\frac{2\pi}{n} = \frac{1}{1+\delta}, \tag{6.1}$$

and draw the circle C' centered at ω and radius $1+\epsilon$, with $0 < \epsilon < \delta$. For every i $(1 \le i \le n)$ let v_i be the intersection between the boundary of C' with the extension of the line segment ωu_i.

To complete the construction, let P be the polygon with vertices $p_1, p_2, \ldots, p_n$, such that $p_i = u_i$ or $p_i = v_i$ depending on whether or not b_i is a 0. The reader will have no difficulty confirming that the resulting polygon P is always convex.

Lemma 6.3.1 AREA has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, independent of the number of processors and memory cells used.
Proof. We shall reduce OR to AREA. For this purpose, we assume that the input to OR consists of n bits, $b_1, b_2, \ldots, b_n$. Let δ satisfy (1) and let ϵ be a positive real less than δ. We use the standard construction described above to associate a convex polygon P with the input sequence. Note that the area of this polygon is exactly $\frac{n}{2}\sin\frac{2\pi}{n}$ if and only if the OR of the input sequence is 0. Since the

[2]As pointed out in [57] this is not really necessary.

construction of P takes O(1) time, the conclusion follows from Proposition 2.3.1 □

Lemma 6.3.2 DIAMETER has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, independent of the number of processors and memory cells used.
Proof. We shall reduce OR to DIAMETER. For this purpose, we assume that the input to OR is n bits, $b_1, b_2, \ldots, b_n$. Let δ satisfy

$$\cos\frac{\pi}{n} = \frac{1}{1+\delta} \tag{6.2}$$

and let ϵ be a positive number less than δ. The input is mapped to a convex polygon with $2n$ vertices $p_1, p_2, \ldots, p_{2n}$, with every bit b_i associated to points p_i and p_{n+i} defined as follows: $p_i = (1 + \epsilon b_i, \frac{i\pi}{n})$ and $p_{n+i} = (1 + \epsilon b_i, \frac{(n+i)\pi}{n})$.

It is easy to see that the construction is such that all points corresponding to 0-bits lie on the unit circle, while all the others lie on the circle of radius $1 + \epsilon$. Note that ϵ has been chosen in such a way that the polygon P determined by the points $p_1, \ldots, p_{2n}$ is always convex. Further, note that the diameter of P is exactly 2 if and only if the OR of the input bits is 0. Since the construction of P takes O(1) time using n processors on the CREW-PRAM, the conclusion follows from Proposition 4.1. □

Lemma 6.3.3 WIDTH has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, independent of the number of processors and memory cells used.
Proof. We shall reduce OR to WIDTH. For this purpose, assume the input to OR to consist of n bits $b_1, b_2, \ldots b_n$. Further, let δ satisfy the condition

$$\cos\frac{\pi}{2n} = \frac{1}{1+\delta}, \tag{6.3}$$

and chose a positive ϵ less than δ. Consider the unit circle C centered at ω and let $u_1, u_2, \ldots, u_{4n}$ be equally spaced points on C such that for all i $(1 \le i \le 4n)$, $u_i = (1, \frac{i\pi}{2n})$. Let C' be the circle centered at ω and with radius $1 + \epsilon$. For every i $(1 \le i \le 4n)$ let v_i be the intersection of C' with the extension of the line segment ωu_i.

The input is mapped to a convex polygon P with $4n$ vertices $p_1, p_2, \ldots, p_{4n}$, with every bit b_i associated to points p_{2i-1}, p_{2i}, $p_{2n+2i-1}$ and p_{2n+2i} defined as follows:

- $p_{2i-1} = (1 + \epsilon - b_i\epsilon, \frac{(2i-1)\pi}{2n})$,
- $p_{2i} = (1 + \epsilon - b_i\epsilon, \frac{2i\pi}{2n})$,
- $p_{2n+2i-1} = (1 + \epsilon - b_i\epsilon, \frac{(2n+2i-1)\pi}{2n})$, and
- $p_{2n+2i} = (1 + \epsilon - b_i\epsilon, \frac{(2n+2i)\pi}{2n})$.

It is easy to confirm that in this construction all points corresponding to 1-bits lie on C, while all the other points lie on the circle C'. In addition, ϵ has been chosen in such a way that the polygon P determined by the points $p_1, \ldots, p_{4n}$ is always convex. Further, note that the width of P is $2\cos\frac{\pi}{4n}$ if and only if the OR of the input bits is 1. Since the construction of P takes O(1) time using n processors on the CREW-PRAM, the conclusion follows from Proposition 2.3.1 □

Lemma 6.3.4 MODALITY has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, independent of the number of processors and memory cells used.
Proof. We shall reduce OR to MODALITY. Let $b_1, b_2, \ldots b_n$ be an arbitrary input to the OR problem. First, if b_1=1 or b_n=1, then the answer to the OR problem is 1. We shall, therefore, assume that $b_1 = b_n = 0$. We convert the sequence $b_1, b_2, \ldots, b_n$ to a new bit-sequence $d_1, d_2, \ldots, d_n$, in two stages. In the first stage the sequence $b_1, b_2, \ldots, b_n$ is converted to a sequence $c_1, c_2, \ldots, c_n$ defined by setting $c_i = 0$ if $b_i = 0$ or if $b_{i-1} = b_i = b_{i+1} = 1$, and by setting $c_i = 1$ otherwise. In the second stage, we negate every bit in the sequence $c_1, c_2, \ldots, c_n$, that is, for every i $(1 \leq i \leq n)$ set $d_i = \bar{c}_i$.

For further reference we take note of the following property of the sequence $d_1, d_2, \ldots d_n$.
Observation 6.3.5 If $b_t = 1$ for some subscript t $(2 \leq t \leq n-1)$, then there exist subscripts i, j, k with $1 \leq i < j < k \leq n$ such that $d_i = d_k = 1$ and $d_j = 0$.
Proof. Let p $(p \geq 2)$ be the first subscript for which $b_p = 1$, and let q be the last subscript for which $b_p = b_{p+1} = \ldots = b_q = 1$. Since, by assumption $b_1 = b_n = 0$ we have $2 \leq p \leq q \leq n-1$.

Note that in the sequence $c_1, c_2, \ldots c_n$ we have $c_{p-1} = 0$, $c_p = 1$, $c_{p+1} = \ldots = c_{q-1} = 0$, $c_q = 1$, and $c_{q+1} = 0$. Consequently, our construction guarantees that $d_{p-1} = 1$, $d_p = 0$, $d_{p+1} = \ldots = d_{q-1} = 1$, $d_q = 0$, and $d_{q+1} = 1$. Now setting $i = p-1$, $j = p$, $k = q+1$, the conclusion follows. □

Having constructed the sequence $d_1, d_2, \ldots, d_n$, we proceed to construct a convex polygon P with $n+1$ vertices as follows. Consider the unit circle C centered at ω and let $u_1, u_2, \ldots, u_n$ be equally spaced points on the first quadrant of C such that for all i $(1 \leq i \leq n)$, $u_i = (1, \frac{i\pi}{2n})$. Let δ be a number satisfying (3), and ϵ be a positive number less than δ. Let C' be the circle centered at ω and with radius $1+\epsilon$. For every i $(1 \leq i \leq n)$ let v_i be the intersection between the first quadrant of C' with the extension of the line segment ωu_i. Now P is a polygon with vertices $p_0, p_1, \ldots, p_n$ such that $p_0 = \omega$, and for every i $(1 \leq i \leq n)$ $p_i = u_i$ or $p_i = v_i$ depending on whether or not d_i is a 0. It is an easy matter to confirm that the polygon P is always convex.

We claim that the modality of P is exactly $n+1$ if and only if the input sequence contains no $b_t = 1$ with $2 \leq t \leq n$. To show that this is the case, note that if no such t exists, then every vertex of P is unimodal, and so the modality of

P equals $n+1$. Conversely, if such a t exists, then Observation 6.3.5 guarantees that p_0 is not unimodal since $\mathrm{d}(p_0, p_i) = \mathrm{d}(p_0, p_k) = 1+\epsilon$, while $\mathrm{d}(p_0, p_j) = 1$. It follows that the overall modality of P is larger than $n+1$. The conclusion follows. □

Lemma 6.3.6 Enclosing rectangle has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, independent of the number of processors and memory cells used.
Proof. We shall reduce OR to ENCLOSING RECTANGLE. For this purpose, assume the input to OR to consist of n bits $b_1, b_2, \ldots b_n$. Let δ be a number satisfying

$$\cos\frac{\pi}{4n} = \frac{1}{1+\delta}, \tag{6.4}$$

and chose a positive ϵ less than δ. The input is mapped to a convex polygon P with $8n$ vertices $p_1, p_2, \ldots, p_{nn}$, with every bit b_i associated to points p_i, p_{i+1}, p_{2n+i}, p_{2n+i+1}, p_{4n+i}, p_{4n+i+1}, p_{6n+i}, and p_{6n+i+1} defined as follows:
- $p_i = (1+\epsilon(1-b_i), \frac{i\pi}{4n})$,
- $p_{i+1} = (1+\epsilon(1-b_i), \frac{i+1)\pi}{4n})$,
- $p_{2n+i} = (1+\epsilon(1-b_i), \frac{(2n+i)\pi}{4n})$,
- $p_{2n+i+1} = (1+\epsilon(1-b_i), \frac{(2n+i+1)\pi}{4n})$,
- $p_{4n+i} = (1+\epsilon(1-b_i), \frac{(4n+i)\pi}{4n})$,
- $p_{4n+i+1} = (1+\epsilon(1-b_i), \frac{(4n+i+1)\pi}{4n})$,
- $p_{6n+i} = (1+\epsilon(1-b_i), \frac{(6n+i)\pi}{4n})$,
- $p_{6n+i+1} = (1+\epsilon(1-b_i), \frac{(6n+i+1)\pi}{4n})$.

It is easy to confirm that in this construction all points corresponding to 1-bits lie on the unit circle, while all the other points lie on the circle of radius $1+\epsilon$. In addition, ϵ has been chosen in such a way that the polygon P determined by the points $p_1, \ldots, p_{8n}$ is always convex. Further, note that the smallest enclosing rectangle has area $(1+\epsilon)^2 4\cos^2\frac{\pi}{8n}$ if and only if the OR of the input bits is 0. Since the construction of P takes $O(1)$ time using n processors on the CREW-PRAM, the conclusion follows from Proposition 2.3.1 □

Lemma 6.3.7 INSCRIBED TRIANGLE has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, independent of the number of processors and memory cells used.
Proof. We shall reduce OR to INSCRIBED TRIANGLE. For this purpose, assume the input to OR is n bits $b_1, b_2, \ldots, b_n$. Let δ be a real number satisfying (6.3) and chose a positive number ϵ less than δ. The input is mapped to a convex polygon P with $4n$ vertices $p_1, p_2, \ldots, p_{4n}$, with every bit b_i associated to points p_{2i-1}, p_{2i}, $p_{2n+2i-1}$ and p_{2n+2i} defined as follows:
- $p_{2i-1} = (1+b_i\epsilon, \frac{(2i-1)\pi}{2n})$,

- $p_{2i} = (1 + b_i\epsilon, \frac{2i\pi}{2n})$,
- $p_{2n+2i-1} = (1 + b_i\epsilon, \frac{(2n+2i-1)\pi}{2n})$, and
- $p_{2n+2i} = (1 + b_i\epsilon, \frac{(2n+2i)\pi}{2n})$.

Observe that our construction guarantees that all points corresponding to 0-bits lie on the unit circle, all the others lie on the circle of radius $1 + \epsilon$, and that the resulting polygon is always convex. Further, note that the largest inscribed triangle has area $\sin\frac{\pi}{2n}$ if and only if the OR of the input bits is 0. Since the construction of P takes O(1) time using n processors on the CREW-PRAM, the conclusion follows from Proposition 2.3.1 □

Combining Proposition 2.3.2 with Lemmas 6.3.1, 6.3.2, 6.3.3, 6.3.4, 6.3.6, and 6.3.7, we have the following result.

Theorem 6.3.8 AREA, DIAMETER, WIDTH, MODALITY, ENCLOSING RECTANGLE, and INSCRIBED TRIANGLE have a time lower bound of $\Omega(\log n)$ on meshes with multiple broadcasting of size $n \times n$. □

Next, we show that MAX DISTANCE, and CONTAINMENT have a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, by reducing the OR problem to each of these problems.

Lemma 6.3.9 MAX DISTANCE has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, independent of the number of processors and memory cells used.
Proof. We shall reduce OR to MAX DISTANCE. For this purpose, assume that the input to OR consists of n bits $b_1, b_2, \ldots, b_n$. Let δ satisfy (3) and let ϵ be a positive number less than δ. The polygon P is constructed by associating with every bit b_i ($1 \le i \le n$), the point $p_i = (1 + \epsilon b_i, \frac{i\pi}{2n})$. The polygon Q is constructed by taking q_i to be the point on the same circle as p_i and diametrically opposite to p_i. It is easy to see that both P and Q are convex. Furthermore, the maximum distance between P and Q is exactly 2 if and only if the OR of the input sequence is 0. Since the construction of P and Q takes O(1) time using n processors on the CREW-PRAM, the conclusion follows from Proposition 2.3.1 □

Lemma 6.3.10 CONTAINMENT has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, independent of the number of processors and memory cells used.
Proof. We shall reduce OR to CONTAINMENT. For this purpose, assume the input to OR is n bits $b_1, b_2, \ldots, b_n$. We use the standard construction to obtain a polygon P. Polygon Q is obtained again by the standard construction from the sequence $\bar{b}_1, \bar{b}_2, \ldots, \bar{b}_n$.

It is easy to see that P is contained in the interior of Q if and only if the OR of the input sequence is 0. Since the construction of P and Q takes O(1) time using

n processors on the CREW-PRAM, the conclusion follows from Proposition 2.3.1 □

To derive time lower bounds for MAX DISTANCE and CONTAINMENT on meshes with multiple broadcasting, we combine Proposition 2.3.2 with Lemmas 6.3.9 and 6.3.10 To summarize our findings we state the following result.

Theorem 6.3.11 MAX VERTEX DISTANCE and CONTAINMENT have a time lower bound of $\Omega(\log n)$ on meshes with multiple broadcasting of size $n \times n$. □

6.4 Algorithms Involving one Convex Polygon

The purpose of this section is to show that the time lower bounds derived in Theorems 6.3.8 are tight. We propose algorithms for AREA, DIAMETER, WIDTH, MODALITY, ENCLOSING RECTANGLE, and INSCRIBED TRIANGLE running in $O(\log n)$ time on meshes with multiple broadcasting of size $n \times n$.

Throughout this section we assume a mesh $\mathcal{M}$ with multiple broadcasting of size $n \times n$. The input to all our algorithms is a convex polygon $P = p_1, p_2, \ldots, p_n$, with p_j $(1 \leq j \leq n)$ stored by processor $P(1, j)$ of $\mathcal{M}$.

To solve the AREA problem, we fix an arbitrary vertex, say p_1, of P. Now broadcasting the coordinates of p_1 to all the vertices of P, every processor in the first row of the mesh can determine the area of the triangle determined by p_1, p_i, and p_{i+1}. Once this is done, it is a simple matter to compute the sum of all the partial results. By Proposition 2.4.4 this takes $O(\log n)$ time which is the best possible. Thus we have the following result.

Theorem 6.4.1 The area of an n-vertex convex polygon stored in the first row of a mesh with multiple broadcasting of size $n \times n$ can be computed in $O(\log n)$ time. Furthermore, this is time-optimal. □

Our solution to the DIAMETER problem relies on the the notion of *antipodal pairs* citePRE. Vertices p_i and p_j of a convex polygon P are an antipodal pair if P admits parallel supporting lines through p_i and p_j (see Figure 6.2). It is well known [57] that the diameter of a convex polygon is the largest distance between antipodal pairs. It is also well-known that the number of antipodal pairs in a convex polygon with n sides is bounded by $3n/2$ [57].

Our DIAMETER algorithm begins by replicating the information in the first row throughout the mesh. This is done by mandating every processor $P(1, j)$ $(1 \leq j \leq n)$, to broadcast the coordinates of the point it stores on the vertical bus

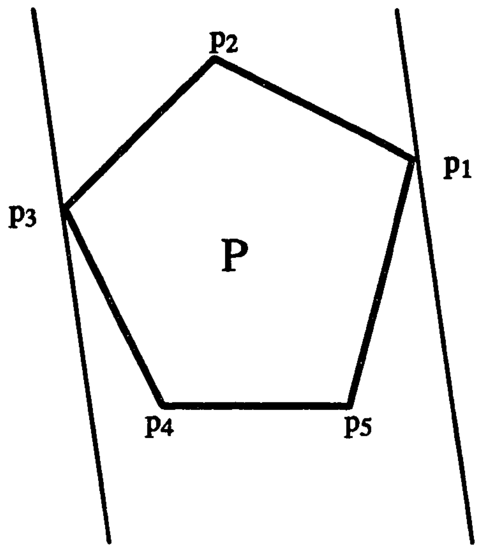


Figure 6.2: Illustrating antipodal pairs

in its own column.

Next, every processor $P(i,i)$, $(1 \leq i \leq n)$, broadcasts a packet consisting of (p_{i-1}, p_i, p_{i+1}) horizontally on the bus in row i. Every processor $P(i,j)$ with $i < j$ can now detect whether the points p_i and p_j are antipodal. If they are, $P(i,j)$ marks itself. It is easy to confirm ([57] page 180) that in every row of the mesh, the marked processors occur consecutively. Therefore, for every i $(1 \leq i \leq n)$, detecting the leftmost and the rightmost marked processor, along with the number of marked processors in row i, called row rank, can be done in O(1) time. In one broadcast operation this information is sent to $P(i,1)$. Once this information has been gathered in the first column of the mesh, Proposition 2.4.4 guarantees that it takes $O(\log n)$ time to compute the prefix sums of these items. The corresponding value of the prefix sum is then broadcast throughout every row. As a consequence, every marked processor knows its rank among the marked processors in the mesh $\mathcal{M}$.

It is an easy observation that in every column of the mesh the marked processors occur consecutively. The rank of each processor in every column of the mesh, termed the column rank, is easily computed in O(1) time. It is an easy observation that either the row rank or column rank of a marked processor must be smaller than or equal to three. Now the marked processors can be divided into two groups. The first group consists of the marked processors with column rank at most three. The second group consists of the marked processors, with column rank larger than three, but with row rank at most three. Our previous observation guarantees that all marked processors are thus accounted for.

We only show how the marked processors with column rank at most three are handled, for handling the marked processors with row rank at most three is similar. We shall refer to the top-most marked processor in every column as belonging to the first generation, the second marked processor in every column is said to be of the second generation, and finally, the third marked processor (if any) belongs to the third generation. We bring the marked processors to the first two rows of the mesh: note that since the total number of antipodal pairs does not exceed $\frac{3n}{2}$, this is possible. The details of this data movement follow. In a first step a generic processor $P(i,j)$ of the first generation holding an item of rank c, sends the item vertically to the diagonal processors $P(j,j)$. In turn, $P(j,j)$ broadcasts the item to $P(j,c)$ which broadcasts the item to $P(1,c)$. The same data movement is then repeated two more time to move the items stored by second and third generation processors to the first two rows. In one time unit, every processor in the second row sends the item it holds northbound to the corresponding processor in the first row, using the local connection. Finally, the largest Euclidian distance between the antipodal pairs is computed in $O(\log n)$ time. To summarize our findings, we state the following result.

Theorem 6.4.2 The DIAMETER problem can be solved in $O(\log n)$ time on a mesh with multiple broadcasting of size $n \times n$. Furthermore, this is time-optimal. □

Given a set S of n points in the plane, the problem of identifying a pair of vertices in S that are farthest apart is central to a number of applications in image processing, robotics, and pattern recognition. It is well-known [57] that such a pair of points is obtained by computing the diameter of the convex hull of S. By Proposition 6.2.1, the convex hull of a set of n points in the plane can be computed in $O(\log n)$ time on a mesh with multiple broadcasting of size $n \times n$. Therefore, we have the following result.

Corollary 6.4.3. The diameter of a set of n points in the plane can be computed in $O(\log n)$ time on a mesh with multiple broadcasting of size $n \times n$. Furthermore, this is time-optimal. □

We note that the width of a convex polygon [32] is the least distance between pairs of antipodal pairs. This observation suggests the following simple algorithm to compute the width of a convex polygon. As in the proof of Theorem 6.4.2, all the pairs of antipodal pairs are determined, ranked and brought to the first row of the mesh. Once there, we only need compute the minimum of the corresponding distances between antipodal pairs. By Proposition 2.4.4, this task can be performed in $O(\log n)$ time. Consequently, we have the following result.

Theorem 6.4.4. The WIDTH problem can be solved in $O(\log n)$ time on a mesh with multiple broadcasting of size $n \times n$. Furthermore, this is time-optimal. □

The result of Theorem 6.4.4 can be extended to handle the width of a set S of n points in the plane [32]. This is defined as the width of the convex hull of S. By Proposition 6.2.1, the convex hull of S can be computed in $O(\log n)$ time. By virtue of Theorem 6.2.4, the width of the convex hull can also be determined in $O(\log n)$ time. Thus, we have proved the following result.

Corollary 6.4.5 The width of a set of n points in the plane can be computed in $O(\log n)$ time on a mesh with multiple broadcasting of size $n \times n$. Furthermore, this is time-optimal. □

We are now interested in devising a time-optimal algorithm to compute the modality of a convex polygon. Our algorithm is a parallelization of the sequential algorithm in [6]. To make the presentation easier to follow, we need to introduce a number of new terms. Let $P = e_1, e_2, \ldots, e_n$ be an n-vertex convex polygon in edge representation. For every i $(1 \leq i \leq n)$, we let $\text{pb}(e_i)$ denote the perpendicular bisector of edge e_i. We also let $\text{op}(e_i)$ stand for the *unique* edge of P intersected by $\text{pb}(e_i)$. Recall that the endpoints of e_i, specified in clockwise order, are p_i and p_{i+1}. We now briefly sketch the modality-finding algorithm in [6].

For every pair of consecutive edges e_k, e_{k+1}, let w_k be the intersection point of $\text{pb}(e_k)$ and $\text{pb}(e_{k+1})$. Let $e_s = \text{op}(e_k)$ and let $e_t = \text{op}(e_{k+1})$. We distinguish between the following cases.

Case 1. The point w_k lies inside P.
Increment the modality counter of all the vertices lying between p_s and p_t, but not including the last vertex if it is the intersection of $\text{bp}(e_{k+1})$ and e_t.

Case 2. The point w_k lies outside P and $\text{pb}(e_k)$ intersects the edge e_{k+1} closer to p_{k+1} than to p_{k+2}.
In this case, increment the modality counter of all the vertices between e_{k+1} and $\text{op}(e_{k+1})$ that are not adjacent to p_{k+1}. The vertex p_t is not counted if $e_t = e_k$.

Case 3. The point w_k lies outside P and $\text{pb}(e_k)$ intersects the edge e_{k+1} closer to

p_{k+2} than to p_{k+1}.
Increment the modality counter of all the vertices strictly between e_s and e_k that are not adjacent to p_{k+1}. The first vertex in e_s is not counted if it is the intersection point of e_s and bp(e_k).
Case 4. The point w_k lies outside P and neither pb(e_k) nor pb(e_{k+1}) intersect the edges e_{k+1} and e_k, respectively.
In this case, increment the modality counter of all the vertices between p_{t+1} and p_s.

We now show that the above algorithm can be implemented in O($\log n$) time on a mesh with multiple broadcasting of size $n \times n$. The input is an n-vertex convex polygon $P = e_1, e_2, \ldots, e_n$, stored in the first row of the mesh, one edge per processor. Using the vertical buses, the first row is replicated throughout the mesh. The intention is to process the pair of edges e_k, e_{k+1} $(1 \leq k \leq n-1)$ in the k-th row of the mesh, and to process the pair e_n, e_1 in the n-th row. We now describe the processing that takes place in a generic row k of the mesh. Of course, the same processing is carried out, in parallel, in all other rows. Processor $P(k,k)$ computes the intersection point w_k of the perpendicular bisectors pb(e_k) and pb(e_{k+1}). Next, using the data movement described in Lemma 6.2.2, we determine in O(1) time whether or not w_k is interior to P. Further, having broadcast the equations of pb(e_k) and pb(e_{k+1}), we determine op(e_k) and op(e_{k+1}). Clearly, this operation takes O(1) time since exactly one processor[3] determines that the edge it holds is intersected by pb(e_k) and another one will detect that the edge it holds is cut by pb(e_{k+1}).

Consequently, in O(1) time the processors in row k of the mesh detect which of the four cases above applies. Once this is known, every processor whose vertex has its modality incremented is marked. It is important to note that in every row, the marked processors occur consecutively. Further, in every row the first and last marked processor send the identity of the edges (resp. vertices) to $P(k,k)$. By the previous observation, the marked vertices form a consecutive chain C_k in P.

We now convert the modality finding problem to the point overlap problem in the following natural way. With every edge e_k we associate the circular arc corresponding to chain C_k. By virtue of Lemma 6.2.2, the corresponding instance of the point overlap can be solved in O($\log n$) time. With vertex p_i of the polygon the modality of p_i is the number of arcs that overlap point i. Consequently, we have the following result.

Theorem 6.4.6. The MODALITY problem can be solved in O($\log n$) time on a mesh with multiple broadcasting of size $n \times n$. Furthermore, this is time-optimal. □

[3]The handling of degenerate cases is similar

As before, consider a convex polygon $P = p_1, p_2, \ldots, p_n$, stored in the first row of a mesh $\mathcal{M}$ with multiple broadcasting of size $n \times n$, with p_j stored by processor $P(1, j)$ $(1 \leq j \leq n)$. Our solution to the ENCLOSING RECTANGLE problem relies on the following technical result proved in [28].

Proposition 6.4.7. The minimum area rectangle enclosing a convex polygon has one side collinear with one of the edges of the polygon. □

Motivated by Proposition 6.4.7, we begin by having every processor $P(1, j)$ $(1 \leq j \leq n)$ broadcast the coordinates of the point it stores on the corresponding vertical bus. Next, every processor $P(i, i)$, $(1 \leq i \leq n)$, broadcasts the equation of the edge determined by p_i and p_{i+1} to all the processors in row i. In each row, at most two adjacent processors detect that the point they hold are farthest away from the edge $p_i p_{i+1}$. We retain the leftmost such processor in each row. Along similar lines, precisely two processors detect that P admits a line of support perpendicular to $p_i p_{i+1}$ through the point they store. It is easy to confirm that in $O(1)$ time processor $P(i, i)$ can compute the area of the enclosing rectangle having one edge collinear with $p_i p_{i+1}$.

Finally, what remains to be done is to compute the minimum of all the areas stored by processors $P(i, i)$ $(1 \leq i \leq n)$. This can be done as follows. First, every processor $P(i, i)$ $(1 \leq i \leq n)$ broadcasts the value of the area it stores vertically to processor $P(1, i)$. Once this information is available in the first row of the mesh, by Proposition 2.4.1, computing the minimum of these values can be performed in $O(\log n)$ time. To summarize our findings we state the following result.

Theorem 6.4.8. The ENCLOSING RECTANGLE problem can be solved in $O(\log n)$ time on a mesh with multiple broadcasting of size $n \times n$. Furthermore, this is time-optimal. □

Now Proposition 6.2.1 and Theorem 6.4.8 combined imply the following result.

Corollary 6.4.9. Given a set S of n points in the plane, the smallest-area rectangle that contains all the points in S can be computed in $O(\log n)$ time on a mesh with multiple broadcasting of size $n \times n$. Furthermore, this is time-optimal. □

Again, consider a convex polygon $P = p_1, p_2, \ldots, p_n$, stored in the first row of a mesh M with multiple broadcasting of size $n \times n$, with p_j stored by processor $P(1, j)$ $(1 \leq j \leq n)$. Our solution to the INSCRIBED TRIANGLE problem relies on the following technical result.

Proposition 6.4.10. ([26]) If $p_i p_k$ is a chord of a convex polygon the function, area$(p_i p_j p_k)$ is unimodal as p_j assumes values on the perimeter of the polygon between p_i and p_k. □

We begin by having every processor $P(1,j)$ $(1 \leq j \leq n)$ broadcast the coordinates of the point it stores on the corresponding vertical bus. Next, every processor $P(i,i)$, $(1 \leq i \leq n)$, broadcasts the equation of the edge $p_i p_{i+1}$ to all the processors in row i. At most two adjacent processors detect that the point they hold are farthest away from the edge $p_i p_{i+1}$. We retain the leftmost such processor in each row and mandate it to send the coordinates of the point it holds to $P(i,i)$ along the bus in row i. It is easy to see that in O(1) time processor $P(i,i)$ can compute the area of the inscribed triangle sharing the edge $p_i p_{i+1}$ with the original polygon. Finally, every processor $P(i,i)$ $(1 \leq i \leq n)$ broadcasts the value of the area it stores vertically to processor $P(1,i)$. Once this information is available in the first row of the mesh, computing the maximum of these values can be performed in $O(\log n)$ time. Furthermore, the correctness of our algorithm is guaranteed by Proposition 5.10. Hence we have the following result.

Theorem 6.4.11. The INSCRIBED TRIANGLE problem can be solved in $O(\log n)$ time on a mesh with multiple broadcasting of size $n \times n$. Furthermore, this is time-optimal. □

6.5 Computations on Two Convex Polygons

The purpose of this section is to show that the time lower bounds derived in Theorem 6.3.11 are tight. We propose algorithms for MAX DISTANCE and CONTAINMENT problems running in $O(\log n)$ time on meshes with multiple broadcasting of size $n \times n$. We also exhibit an O(1) time algorithm for the MIN DISTANCE problem.

Throughout this section we assume a mesh $\mathcal{M}$ with multiple broadcasting of size $n \times n$. The input to all the algorithms we present are convex polygons $P = p_1, p_2, \ldots, p_n$ and $Q = q_1, q_2, \ldots, q_n$, with p_j and q_j stored by processor $P(1,j)$ $(1 \leq j \leq n)$.

To solve the MAX DISTANCE in optimal time, we parallelize the algorithm of Bhattacharya and Toussaint [19]. Specifically, they preprocess P and Q as follows. We shall present the preprocessing on P only. Find p_{xmax}, p_{xmin}, p_{ymax} and p_{ymin}

the points of maximum and minimum x and y coordinates in P (for simplicity we assume that they are unique). Let R be the rectangle with sides parallel to the x and y axes through the extreme points of P determined above. The rectangle R is partitioned into nine subrectangles R_1, R_2, ..., R_9 each of length $\frac{x_{xmax}-x_{xmin}}{3}$ and width $\frac{y_{ymax}-y_{ymin}}{3}$.

In a perfectly similar way, Q is partitioned into nine rectangles $R'_1, R'_2, \ldots, R'_9$. Further, let S_i $(1 \leq i \leq 9)$ be the set of vertices of P belonging to R_i, and let S'_i $(1 \leq i \leq 9)$ be the set of vertices of Q belonging to R'_i. A key result in [19] states that

Proposition 6.5.1. [19] The maximum vertex distance between P and Q is achieved by $\max_{1\leq i,j\leq 9}\{\text{diam}(S_i \cup S'_j)\}$. □

Our MAX DISTANCE algorithm begins by computing the convex hulls of the two polygons. By Proposition 6.2.2 this can be done in $O(\log n)$ time. Further, it is not hard to see that each of the pairs $S_i \cup S'_j$ $(1 \leq i,j \leq 9)$ can be generated. Now Theorem 6.3.1 and Proposition 6.4.2 guarantee that the diameter of $S_i \cup S'_j$ can be computed in $O(\log n)$ time. Since there are a constant number of such pairs we have the following result.

Theorem 6.5.2. The MAX DISTANCE between two n-vertex convex polygons can be computed in $O(\log n)$ time on a mesh with multiple broadcasting of size $n \times n$. Furthermore, this is time-optimal. □

It is important to note that the algorithm of Bhattacharya and Toussaint [19] solves the problem of computing the maximum distance between two arbitrary planar sets of points. The only additional step involved is the compute the convex hulls of the original sets of points. By Proposition 6.2.1 this can be done in $O(\log n)$ time on a mesh with multiple broadcasting of size $n \times n$. Therefore, we have the following result.

Corollary 6.5.3. The maximum distance between two arbitrary n-point sets in the plane can be computed in $O(\log n)$ time on a mesh with multiple broadcasting of size $n \times n$. Furthermore, this is time-optimal. □

As before, consider convex polygons $P = p_1, p_2, \ldots, p_n$ and $Q = q_1, q_2, \ldots, q_n$, with p_j and q_j, stored in the first row of a mesh M with multiple broadcasting of size $n \times n$, with p_j and q_j stored by processor $P(1,j)$ $(1 \leq j \leq n)$. To check whether P is contained in Q we only need check whether every vertex of P belongs to the interior of Q. By Lemma 6.2.2, to solve this task, every vertex of P needs a row of the mesh and O(1) time. Finally, a simple maximum prefix establishes whether the containment holds for all vertices. To summarize our findings we state the following result.

Theorem 6.5.4. The CONTAINMENT problem for two n-vertex convex polygons can be solved in $O(\log n)$ time on a mesh with multiple broadcasting of size $n \times n$. Furthermore, this is time-optimal. $\square$

Consider separable convex polygons $P = p_1, p_2, \ldots, p_n$ and $Q = q_1, q_2, \ldots, q_n$, with P and Q stored in the first column and the first row, respectively, of a mesh $\mathcal{M}$ with multiple broadcasting of size $n \times n$. For definiteness, we assume that for all i $(\leq i \leq n)$, $P(1, i)$ stores q_i and $P(i, 1)$ stores p_i. We propose to show that in this setup, the task of computing the smallest distance between P and Q can be solved in O(1) time.

To make the exposition easier to follow, we assume without loss of generality that the two polygons are separable in the x direction, with P to the left of Q. Let T_u and T_l be the upper and lower common tangent of P and Q. Let T_u touch P and Q at p_r and q_s and let T_l touch P and Q at p_t and q_w. Let C_P and C_Q be the mutually visible chains in P and Q respectively. In other words, C_Q involves vertices $q_s, q_{s+1}, \ldots, q_w$, while the chain C_P involves the vertices $p_t, p_{t+1}, \ldots, p_r$. Simple geometric considerations confirm that to compute the minimum distance between P and Q we only need examine the distance from vertices in C_P to C_Q. Now a result in [23] guarantees that the distance function of vertices in these chains is unimodal. Specifically, for every vertex u in C_P, the distance to any point v (not necessarily a vertex) of C_Q first decreases and then increases, as v moves from q_s to q_w, with the minimum achieved by either a vertex of Q or by the perpendicular projection of u on the boundary of Q. A mirror property holds for points in C_Q and their distance function to C_P. Once the minimum distance is computed for every point in the chains C_P and C_Q, the minimum distance between the two polygons can be determined in O(1) time by exploiting the convexity of the two polygons.

We now present the details of our algorithm. To begin, we show how the upper tangent T_u is computed. First, using vertical buses, the contents of the first row is replicated throughout the mesh. In every row i $(1 \leq i \leq n)$ processor $P(i, 1)$ broadcasts p_i horizontally to the whole row. In row i $(1 \leq i \leq n)$, a unique processor $P(i, j)$ will find that the line determined by p_i and q_j is the upper supporting line to Q from p_i. In the next time unit, this processor broadcasts q_j back to $P(i, 1)$. In turn, by checking whether both p_{i-1} and p_{i+1} are below the line $p_i q_j$, processor $P(i, 1)$ detects whether $p_i q_j$ is the upper tangent of P and Q. Clearly, *exactly* one processor in the first column of $\mathcal{M}$ detects this condition. The lower common tangent is determined similarly. Note that the whole computation runs in constant time. Therefore, we have the following result.

Lemma 6.5.5 The common tangents of two n-vertex convex polygons stored in one row/column of a mesh with multiple broadcasting of size $n \times n$ can be computed in O(1) time. □

Once T_u and T_l have been computed, identifying the chains C_P and C_Q is achieved by a simple broadcasting and marking operation. We now show how the minimum distance from a vertex p_i in C_P to C_Q is computed in one row of $\mathcal{M}$. Of course, the same computation is performed, in parallel, in all rows in which $P(i,1)$ stores a vertex in C_P. Processor $P(i,1)$ broadcasts p_i horizontally to the whole row i. Every processor in row i ($1 \le i \le n$), storing a vertex q_j in C_Q computes the distance $\mathrm{d}(p_i, q_j)$. Notice that a unique processor $P(i,k)$ detects that $\mathrm{d}(p_i, q_k) < \mathrm{d}(p_i, q_{k-1})$ and that $\mathrm{d}(p_i, q_k) \le \mathrm{d}(p_i, q_{k+1})$. In addition, this processor computes the intersection points of each of the edges $q_{k-1}q_k$ and q_kq_{k+1} with the perpendiculars from p_i to these two edges. If one of these points is interior to one of the edges $q_{k-1}q_k$ or q_kq_{k+1}, then $P(i,k)$ reports the corresponding perpendicular distance back to $P(i,1)$. Otherwise, $P(i,k)$ reports $\mathrm{d}(p_i, q_k)$.

Now every processor in the first column of the mesh that contains a vertex in C_P compares the minimum distance achieved by its own vertex with the minimum distances achieved by the vertices stored by its two neighbors. Convexity guarantees that exactly one of them will detect the minimum distance. The previous steps are then repeated for every vertex in C_Q, thus obtaining the minimum distance between a vertex in C_Q and a point (not necessarily vertex) in C_P. Once this information is available, a simple comparison establishes the minimum distance between P and Q. To summarize our findings we state the following result.

Theorem 6.5.6. The MIN DISTANCE problem involving two separable n-vertex polygons stored in one row/column of a mesh with multiple broadcasting of size $n \times n$ can be solved in O(1) time. □

6.6 Conclusions

For two n-vertex convex polygons P and Q, we have established $\Omega(\log\ n)$ time lower bounds for the following problems:
- computing the area of P;
- computing the diameter of P;
- computing the width of P;

- computing the modality of P;
- computing the smallest area rectangle enclosing P;
- computing a maximum-area inscribed triangle sharing an edge with P;
- computing the maximum distance between P and Q;
- detecting whether or not P is contained in the interior of Q.

We then have showed that the bounds are tight by providing $O(\log n)$ algorithms to accomplish these tasks on meshes with multiple broadcasting of size $n \times n$. We have also shown that the problem of computing the minimum distance between two separable n-vertex polygons stored in one row/column of a mesh with multiple broadcasting of size $n \times n$ can be solved in $O(1)$ time.

Other problems seem to be harder. First, we don't know how to determine the largest inscribed triangle in a given convex polygon. In [26] an elegant $O(n)$ time sequential algorithm is presented but it does not seem to be parallelizable to run in $O(\log\ n)$ time. Further, it would be nice to solve the symmetric problems of computing the smallest-area enclosing triangle as well as the largest enscribed circle and rectangle.

Chapter 7

Tree Problems

7.1 Introduction

Encoding the shape of an ordered tree is a basic step in a number of algorithms in integrated circuit design, automated theorem proving, and game playing [25, 73]. The common characteristic of these applications is that the information stored at the nodes is irrelevant, as one is only interested in detecting whether two ordered trees have the same "shape". As it turns out, if we ignore the contents of the nodes of an n-node tree T, then the remaining shape information can be uniquely captured by a string of $2n$ bits, referred to as the *encoding* of T [50, 73]. Conversely, given a string of $2n$ bits, a number of practical applications ask to recover the unique n-node ordered tree (if any) corresponding to this encoding.

One of the contributions of this thesis is to provide time-optimal tree algorithms on the MMB architecture. Specifically, we show that the following tasks can be solved in $\Theta(\log n)$ time on an MMB of size $n \times n$:

• Encode an n-node binary tree into a $2n$-bitstring;
• Encode an n-node ordered tree into a $2n$-bitstring;
• Recover an n-node binary tree from its $2n$-bit encoding;
• Recover an n-node ordered tree from its $2n$-bit encoding.

We also show that the following tasks can be performed in $O(1)$ time:

• Reconstruct an n-node binary tree from its preorder and inorder traversals;
• Reconstruct an n-node ordered tree (forest) from its preorder and postorder traversals.

Our algorithms rely on novel time-optimal algorithms involving sequences of parentheses, that we also develop. Specifically, we show that each of the tasks can be solved in $\Theta(\log n)$ time:

• determining whether a sequence of n parentheses is well-formed;

• finding all the matching pairs in a well-formed sequence of parentheses;
• determining the closest enclosing pair for every matching pair in a well-formed sequence.

The remainder of the chapter is organized as follows. Section 7.2 presents our lower bound arguments. Section 7.3 discusses the details of our parentheses algorithms. Section 7.4 presents time-optimal algorithms for encoding and decoding binary and ordered trees. Section 7.5 addresses the problems or reconstructing binary and ordered trees from their traversals. Finally, Section 7.6 concludes with some final remarks and open problems.

7.2 Lower Bounds

LEFTMOST ONE: Given a sequence τ of n bits, determine the position of the leftmost 1 in τ.
It is a trivial observation that OR reduces to LEFTMOST ONE in O(1) time. Therefore, Proposition 2.3.1 implies the following result.

Corollary 7.2.1 LEFTMOST ONE has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, regardless of the number of processors and memory cells used. □

We are now in a position to present our lower bound arguments. A sequence $\sigma = x_1x_2 \ldots x_n$ of parentheses is said to be *well-formed* if it contains the same number of left and right parentheses and in every prefix of σ the number of right parentheses does not exceed the number of left parentheses. The WFS decision problem is stated as follows.
WFS: Given a sequence $\sigma = x_1x_2 \ldots x_n$ of parentheses, is σ well-formed?

Lemma 7.2.2 WFS has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, regardless of the number of processors and memory cells used.
Proof. We reduce OR to WFS. Assume that the input to OR is a sequence $b_1, b_2, \ldots, b_n$. We construct a sequence $c_1c_2 \ldots c_{2n}$ of parentheses by setting for every j $(1 \leq j \leq n)$:
• c_{2j-1} = '(' and c_{2j} = ')', whenever $b_j = 0$;
• c_{2j-1} = ')' and c_{2j} = '(', whenever $b_j = 1$.
Our construction guarantees that the solution to OR is 0 if and only if the sequence $c_1c_2 \ldots c_{2n}$ is well-formed. Thus, the conclusion follows by Proposition 2.3.1. □

Next we define the classic parentheses matching problem.

MATCHING: Given a well-formed sequence $\sigma = x_1x_2\dots x_n$ of parentheses, for each parenthesis in σ, find its match.

Lemma 7.2.3 MATCHING has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, regardless of the number of processors and memory cells used.
Proof. We reduce OR to MATCHING. For this purpose, let the input to OR consist of n bits $b_1, b_2, \dots, b_n$. We convert this input to a sequence $c_0c_1c_2\dots c_{2n+1}$ of parentheses by writing $c_0 =$ '(' and $c_{2n+1} =$ ')', and by setting for all j $(1 \le j \le n)$:
• $c_{2j-1} =$ '(' and $c_{2j} =$ ')', whenever $b_j = 0$;
• $c_{2j-1} =$ ')' and $c_{2j} =$ '(', whenever $b_j = 1$.
An easy inductive argument on the number of 1's in $b_1, b_2, \dots, b_n$ shows that the sequence $c_0c_1c_2\dots c_{2n+1}$ is always well-formed. Furthermore, the matching pair of c_0 is c_{2n+1} if and only if the answer to the OR problem is 0. The conclusion follows by Proposition 2.3.1. □

Given a well-formed sequence of parentheses in which every parenthesis knows its matching pair, it is often of interest to determine the solution to the following problem.
ENCLOSING PAIR: Given a well-formed sequence $\sigma = x_1x_2\dots x_n$ of parentheses, for every matching pair of parentheses in σ, find the closest pair of parentheses that encloses it.

Lemma 7.2.4 ENCLOSING PAIR has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, regardless of the number of processors and memory cells used.
Proof. We reduce LEFTMOST ONE to ENCLOSING PAIR. Assume that the input to LEFTMOST ONE is $b_1, b_2, \dots, b_n$. Construct a sequence $c_1c_2\dots c_{4n+2}$ of parentheses as follows:
• $c_{2n+1} =$ '('; $c_{2n+2} =$ ')';
furthermore, for all j $(1 \le j \le n)$ set
• $c_{2n-2j+1} =$ '('; $c_{2n-2j+2} =$ ')'; $c_{2n+2j+1} =$ '('; $c_{2n+2j+2} =$ ')', whenever $b_j = 0$;
• $c_{2n-2j+1} =$ '('; $c_{2n-2j+2} =$ '('; $c_{2n+2j+1} =$ ')'; $c_{2n+2j+2} =$ ')', whenever $b_j = 1$.
Our construction guarantees that the sequence is well-formed and that every parenthesis knows its match; in particular, c_{2n+1} and c_{2n+2} are a matching pair. Furthermore, the closest enclosing parentheses for the pair (c_{2n+1}, c_{2n+2}) is $(c_{2n-2k+2}, c_{2n+2k+1})$, if and only if k is the position of the leftmost 1 in $b_1, b_2, \dots, b_n$. Now the conclusion follows by Corollary 7.2.1 □

A binary tree T is either empty or consists of a root and two disjoint binary trees, called the left subtree, T_L and the right subtree, T_R. In many contexts, it is desirable to encode the shape of T as succinctly as possible. Many encoding schemes are known [50, 73]. In this chapter, we are interested in one such encoding scheme recursively defined as follows:

$$\sigma(T) = \begin{cases} \epsilon & \text{if } T \text{ is empty;} \\ 1\sigma(T_L)0\sigma(T_R) & \text{otherwise.} \end{cases} \tag{7.1}$$

It is not hard to see that under (7.1) an arbitrary n-node binary tree is encoded into $2n$ bits. Refer to Figure 7.1 for an example.

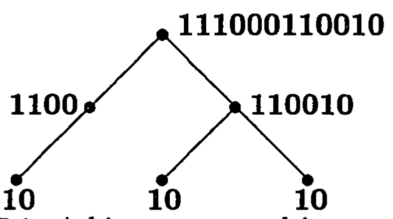


Figure 7.1: A binary tree and its encoding

BINARY TREE ENCODING: Given an n-node binary tree, encode it into $2n$ bits.

Lemma 7.2.5 BINARY TREE ENCODING has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, regardless of the number of processors and memory cells used.

Proof. We reduce OR to BINARY TREE ENCODING. Assume that $b_1, b_2, \ldots, b_n$ is an arbitrary input to OR. Convert this bit sequence to an n-node binary tree T with nodes $1, 2, 3, \ldots, n$. Specifically, we associate with every bit b_j $(1 \leq j \leq n)$ the node j of T, such that:

- 1 is the root of T;
- for every i $(1 \leq i \leq n-1)$, node $i+1$ is the unique child of i. Moreover, $i+1$ is the left child of i if $b_i = 1$ and the right child otherwise.

Clearly the construction of T takes O(1) time. Let $\sigma(T) = c_1c_2\ldots c_{2n}$ be the $2n$-bit encoding of T. We rely on the following simple observation.

Observation 7.2.6 $c_{2n-1} = 1$ if and only if the answer to OR is 0.

Proof. Observe that if all the bits in the string $b_1, b_2, \ldots, b_n$ are 0 then, by our construction and (7.1) combined, $c_{2n-1} = 1$. Conversely, let j be the position of the leftmost 1 in $b_1, b_2, \ldots, b_n$. Clearly, (7.1) guarantees that the encoding of the subtree rooted at j is $1\sigma(T_{j+1})0$ and, since by construction node j has no right child, this is a suffix of the encoding of T. Since $\sigma(T_{j+1})$ must end in a 0, it follows that $c_{2n-1} = 0$ and the conclusion follows. □

By virtue of Observation 7.2.6, once the encoding is available, one can determine in O(1) time the answer to the OR problem. Therefore, by Proposition 2.3.1, the encoding algorithm must take $\Omega(\log n)$ time. □

The converse operation involving the recovery of a binary tree from its encoding is of interest in a number of practical applications. We assume that every node in a binary tree maintains pointers to its left and right children. We state the problem as follows.
BINARY TREE DECODING: Recover an n-node binary tree from its $2n$-bit encoding.

Lemma 7.2.7 BINARY TREE DECODING has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, regardless of the number of processors and memory cells used.
Proof. We reduce OR to BINARY TREE DECODING. Let $b_1, b_2, \ldots, b_n$ be an arbitrary input to OR. First, if $b_1 = 1$, then the answer to OR is 1. We may, therefore, assume that $b_1 = 0$. Construct a well-formed sequence of parentheses, $c_0 c_1 \ldots c_{2n+1}$ as described below:

- $c_0 =$ '(' ; $c_{2n+1} =$ ')'.
- $c_{2i-1} =$ '('; $c_{2i} =$ ')', whenever $b_i = 0$;
- $c_{2i-1} =$ ')' and $c_{2i} =$ '(', whenever $b_i = 1$ and $b_{i-1} = 0$;
- $c_{2i-1} =$ '(' and $c_{2i} =$ ')', whenever $b_i = 1$ and $b_{i-1} = 1$.

It is easy to verify that the resulting sequence is well-formed and so, interpreting every '(' as a 1 and every ')' as a 0, $c_0 c_1 \ldots c_{2n+1}$ is the encoding of a binary tree T with $n+1$ nodes. Notice that root(T) has two children if and only if the OR of the input sequence is 1. Therefore, once the decoding is available, one can solve the OR problem in O(1) time. Now Proposition 2.3.1 implies that any algorithm that performs the decoding must take $\Omega(\log n)$ time. □

An ordered tree T is either empty or it contains a root and disjoint ordered subtrees, $T_1, T_2, \ldots, T_k$. The encoding $\sigma(T)$ of T, is defined as follows:

$$\sigma(T) = \begin{cases} \epsilon & \text{if } T \text{ is empty} \\ 1\sigma(T_1)\sigma(T_2)\ldots\sigma(T_k)0 & \text{otherwise.} \end{cases} \qquad (7.2)$$

An easy inductive argument shows that the encoding of an n-node ordered tree is a sequence of $2n$ bits. Refer to Figure 7.2 for an example.

ORDERED TREE ENCODING: Given an n-node ordered tree, encode it into $2n$ bits.
Lemma 7.2.8 ORDERED TREE ENCODING has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, regardless of the number of processors and memory cells used.

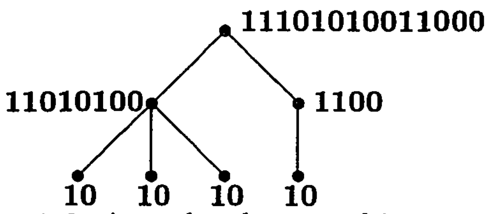


Figure 7.2: An ordered tree and its encoding

Proof. We reduce OR to ORDERED TREE ENCODING. Let $b_1, b_2, \ldots, b_n$ be an arbitrary input to OR. We add two bits $b_0 = 1$ and $b_{n+1} = 0$. The new sequence $b_0, b_1, b_2, \ldots, b_{n+1}$ is converted to an ordered tree T as follows:
• b_0 is the root;
• for all i $(1 \leq i \leq n+1)$, the parent of b_i is b_0 if $b_i = 0$ and b_{n+1} otherwise.
Let $c_1c_2 \ldots c_{2n+4}$ be the $2(n+2)$-bit encoding of T. We rely on the following result.

Observation 7.2.9 $c_{2n+2} = 1$ if and only if the OR of the input sequence is 0.
Proof. If all bits in $b_1, b_2, \ldots, b_n$ are 0 then, by (7.2) and our construction, $c_{2n+2} = 1$. Conversely, if there exist 1's in the sequence $b_1, b_2, \ldots, b_n$, then the node of T corresponding to b_{n+1} is not a leaf. Therefore, (7.2) guarantees that $c_{2n+2} = c_{2n+3} = c_{2n+4} = 0$ and the conclusion follows. □
By Observation 7.2.9, once the encoding is available, the answer to OR can be obtained in O(1) time. Therefore, by Proposition 2.3.1, the encoding algorithm must take $\Omega(\log n)$ time. □

The converse operation requires recovering an ordered tree from its encoding. Specifically, the tree is assumed specified in parent pointer representation. We state the problem as follows.
ORDERED TREE DECODING: Recover an n-node ordered tree from its $2n$-bit encoding.

Lemma 7.2.10 ORDERED TREE DECODING has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, regardless of the number of processors and memory cells used.
Proof. We reduce ENCLOSING PAIR to ORDERED TREE DECODING. Let the input to ENCLOSING PAIR be $s_1s_2 \ldots s_{2n}$. Augment this sequence with s_0 = '(' and s_{2n+1} = ')'. Thus, interpreting every '(' as a 1 and every ')' as a 0 we obtain the valid encoding of some ordered tree T under (7.2). Now, consider any algorithm that correctly recovers T from the encoding above. It is easy to see that

the setting of parent pointers gives exactly the solution to the ENCLOSING PAIR problem for the augmented sequence. The conclusion follows from Lemma 7.2.4. □

Recall that Proposition 2.3.2 guarantees that if $T_M(n)$ is the execution time of an algorithm for solving a given problem on an n-processor MMB, then there exists a CREW-PRAM algorithm to solve the same problem in $T_P(n) = T_M(n)$ time using n processors and $O(n)$ extra memory. Now Proposition 2.3.2 together with Lemmas 7.2.2, 7.2.3, 7.2.4, 7.2.5, 7.2.7, 7.2.8, and 7.2.10 imply the following result.

Theorem 7.2.11 WFS, MATCHING, ENCLOSING PAIR, BINARY TREE ENCODING, BINARY TREE DECODING, ORDERED TREE ENCODING, and ORDERED TREE DECODING have a time lower bound of $\Omega(\log n)$ on an MMB of size $n \times n$. □

7.3 Time-Optimal Parentheses Algorithms

The purpose of this section is to present three time-optimal algorithms involving sequences of parentheses on an MMB of size $n \times n$. In addition to being interesting in their own right, these algorithms are instrumental in our subsequent tree algorithms.

Consider a sequence $\sigma = x_1x_2 \dots x_n$ of parentheses stored one item per processor in the first row of an MMB of size $n \times n$, with x_k $(1 \leq k \leq n)$ stored by $P(1, k)$.

Our first algorithm decides whether σ is well-formed. Begin by assigning a 1 to every left parenthesis and a -1 to every right parenthesis, and compute the prefix sums of the resulting sequence. Note that σ is well-formed if and only if the total sum is zero and all prefix sums are non-negative. Every processor for which the prefix sum is negative marks itself. Using the prefix sums algorithm of Proposition 2.4.7 we rank all the marks. Trivially, we can move all the marks to the left most positions of the first row of the mesh in $O(1)$ time. To check whether σ is well-formed we only need verify that $P(1, 1)$ does not contain a mark. Note that the entire computation is performed in $O(\log n)$ time on an MMB of size $n \times n$. Theorem 7.2.11 guarantees that this is the best possible. We have the following result.

Theorem 7.3.1 Checking whether a sequence σ of n parentheses is well-formed takes $O(\log n)$ time on an MMB of size $n \times n$. Furthermore, this is time-optimal.

□

Next, assuming that the sequence $\sigma = x_1 x_2 \ldots x_n$ is well-formed, we present an algorithm to find all the matching pairs. The idea is as follows. First, we compute a sequence w_1, w_2, ..., w_n obtained from σ by setting w_1=0 and by defining w_k $(2 \leq k \leq n)$ as follows:

$$w_k = \begin{cases} 1 & \text{if both } x_{k-1} \text{ and } x_k \text{ are left parentheses;} \\ -1 & \text{if both } x_{k-1} \text{ and } x_k \text{ are right parentheses;} \\ 0 & \text{otherwise.} \end{cases}$$

We now compute the prefix sums of $w_1, w_2, \ldots, w_n$ and let the result be $e_1, e_2, \ldots, e_n$. By Proposition 2.4.7, this operation is performed in $O(\log n)$ time. It is easy to see that left and right parentheses x_i and x_j are a matching pair if and only if x_j is the first right parenthesis to the right of x_i for which $e_i = e_j$.

Further, with each parenthesis x_k $(1 \leq k \leq n)$, we associate the tuple (e_k, k). On the set of these tuples we define a binary relation $\prec$ by setting

$$(e_i, i) \prec (e_j, j) \text{ whenever } (e_i < e_j) \text{ or } [(e_i = e_j) \text{ and } (i < j)].$$

It is an easy exercise to show that $\prec$ is a linear order. Now, sort the sequence $(e_1, 1), \ldots, (e_n, n)$ in increasing order of $\prec$. By Proposition 2.4.4, sorting the ordered pairs can be done in $O(\log n)$ time. The key observation is that, as a result of sorting, the matching pairs occur next to one another. Consequently, we have the following result.

Theorem 7.3.2 Given a well-formed sequence of n parentheses as input, all matching pairs can be found in $O(\log n)$ time on an MMB of size $n \times n$. Furthermore, this is time-optimal. □

Finally, we are interested in finding a time-optimal solution to the ENCLOSING PAIR problem stated in section 7.2. For this purpose, consider a well-formed sequence $\sigma = x_1 x_2 \ldots x_n$ of parentheses, stored one item per processor in the first row of the mesh. We assume that every parenthesis in σ knows its match. The details of the algorithm follow.

Step 1. Find the match of every parenthesis in σ; every processor $P(1, i)$ stores in a local variable the position j of the match x_j of x_i.

Step 2. Solve the corresponding instance of the ANSV problem.

It is not hard to see that at the end of Step 2 every processor knows the identity of the closest enclosing pair. By Proposition 2.4.6 and Theorem 7.3.2, the running time of this simple algorithm is bounded by $O(\log n)$. By Theorem 7.2.13, this is the best possible on this architecture. Thus, we have proved the following result.

Theorem 7.3.3 Given a well-formed sequence $\sigma = x_1 x_2 \dots x_n$ parentheses stored one item per processor in the first row of an MMB of size $n \times n$, the ENCLOSING PAIR problem can be solved in $O(\log n)$ time. Furthermore, this is time-optimal. □

7.4 Encoding and Decoding Trees

The purpose of this section is to show that the task of encoding n-node binary and ordered trees into a $2n$-bitstring can be carried out in $O(\log n)$ time on an MMB of size $n \times n$. By virtue of Theorem 7.2.11, this is time-optimal.

Consider an n-node binary tree T with left and right subtrees T_L and T_R, respectively. We assume that the nodes of T are stored, one item per processor, in the first row of an MMB of size $n \times n$. We further assume that every node in T maintains pointers to its left and right children. First, we show how to associate with T the unique encoding $\sigma(T)$ defined in (7.1).

Our encoding algorithm can be seen as a variant of the classic Euler-tour technique [69]. We proceed as follows. Replace every node u of T by 3 copies, u^1, u^2, and u^3. If u has no left child, then set link$(u^1) \leftarrow u^2$, else if v is the left child of u, set link$(u^1) \leftarrow v^1$ and link$(v^3) \leftarrow u^2$. Similarly, if u has no right child, then set link$(u^2) \leftarrow u^3$ else if w is the right child of u then set link$(u^2) \leftarrow w^1$ and link$(w^3) \leftarrow u^3$. It is worth noting that the processor associated with node u can perform the pointer assignments in $O(1)$ time. What results is a linked list starting at $root(T)^1$ and ending at $root(T)^3$, with every edge of T traversed exactly once in each direction. It is easy to confirm that the total length of the linked list is $O(n)$. Finally, assign to every u^1 a 1, to every u^2 a 0 and delete all elements of the form u^3. It is now an easy matter to show that what remains represents the encoding of T specified in (7.1).

The correctness of this simple algorithm being easy to see, we turn to the complexity. Computing the Euler tour amounts to setting pointers. Since all the information is available locally, this step takes $O(1)$ time. The task of eliminating every node of the form u^3 can be reduced to list ranking, prefix computation, and then compacting in the obvious way. By virtue of Propositions 2.4.1, and 2.4.7 these tasks can be performed in $O(\log n)$ time. By Theorem 7.2.13, this is the best possible. Consequently, we have the following result.

Theorem 7.4.1 The task of encoding an n-node binary tree into a $2n$-bitstring can be performed in $O(\log n)$ time on an MMB of size $n \times n$. Furthermore, this is time-optimal. □

It is worth noting here that the encoding algorithm described above is quite general and can be used for other purposes as well. For example, the *preorder-inorder* traversal of T is obtained by replacing for every node u of T, u^1 and u^2 by the label of u (see [52] for details). We will further discuss properties of the preorder-inorder traversal in the context of reconstructing binary trees from their preorder and inorder traversals in section 7.5.

Our encoding algorithm for ordered trees is very similar to the one described for binary trees. Consider an n-node ordered tree T. It is well-known [50] that for the purpose of getting the encoding (7.2) of T we only need to convert T into a binary tree BT as in [35] and then to encode BT using (7.1). It is easy to confirm that the resulting encoding is exactly the one defined in (7.2). The conversion of T into BT can be performed in O(1) time since it amounts to resetting pointers only. By Theorem 7.4.1, the encoding of BT takes $O(\log n)$ time. By Theorem 7.2.11 this is the best possible. Consequently, we have the following result.

Theorem 7.4.2 The task of encoding an n-node ordered tree into a $2n$-bitstring can be performed in $O(\log n)$ time on an MMB of size $n \times n$. Furthermore, this is time-optimal. □

Before addressing the task of recovering binary and ordered trees from their encodings, we introduce some notation and review a few technical results. Let T be a binary tree and let v be a node of T. We let T^v stand for the subtree of T rooted at v. A bitstring τ is termed *feasible* if it contains the same number of 0's and 1's and in every prefix the number of 0's does not exceed the number of 1's. Recently, Olariu *et al.* [50] have shown that every feasible bitstring is the encoding of some binary tree. For later reference, we state the following technical result [50].

Proposition 7.4.3 A nonempty bitstring τ is feasible if and only if for every 1 in τ there is a *unique* matching 0 such that τ can be written as $\tau_1 1 \tau_2 0 \tau_3$, with both τ_2 and $\tau_1\tau_3$ feasible. □

Proposition 7.4.3 motivates us to associate with every 1 and its matching 0, a node v in T. The following simple observation [50] will justify our decoding procedure.

Observation 7.4.4. The corresponding decomposition of τ as $\tau_1 1 \tau_2 0 \tau_3$ has the property that $\sigma({T_L}^v) = \tau_2$, and $\sigma({T_R}^v)$ is a prefix of τ_3. □

Observation 7.4.4 motivates our algorithm for recovering a binary tree from its encoding. Let τ be a feasible bitstring. For every 1 in τ we find the unique matching 0 guaranteed by Proposition 7.4.3. The corresponding (1,0) pair is associated with a node v in the binary tree T corresponding to τ. We then compute the left and right children of v. The details of the algorithm are spelled out as follows. Begin by ranking the 1's of τ and use the ranks as indices in T. For every 1, find its unique matching 0. Let v_i be the node of T corresponding to the 1 of rank i and to its matching 0; let p_i and q_i denote the positions in τ of the 1 of rank i and that of its matching 0, respectively. The processor in charge of v_i sets pointers as follows:

- left(v_i) $\leftarrow$ **nil** in case $q_i = p_i+1$, and left(v_i) $\leftarrow v_{i+1}$ otherwise;
- right(v_i) $\leftarrow v_j$ if $p_j = q_i + 1$, and **nil** otherwise.

The correctness follows immediately from Proposition 7.4.3 and Observation 7.4.4. Therefore, we turn to the complexity. Note that to rank all the 1's we need to compute their prefix sum. By Proposition 2.4.7, this task can be performed in $O(\log n)$ time. By Theorem 7.3.2, the matching takes $O(\log n)$ time. Finally, the setting of pointers can be done in $O(1)$ time. Thus, we have the following result.

Theorem 7.4.5 The task of recovering an n-node binary tree from its $2n$-bit encoding can be performed in $O(\log n)$ time on an MMB of size $n \times n$. Furthermore, this is time-optimal. □

The task of recovering an n-node ordered tree T from its $2n$-bit encoding is similar. We begin by perceiving the encoding of T as the encoding of a binary tree BT. Once, this tree has been recoved as we just described, we proceed to convert BT to T using the classic ordered-to-binary conversion [35]. As it turns out, this latter task can be carried out in $O(\log n)$ time using the sorting algorithm of Proposition 2.4.4. To summarize our findings, we state the following result.

Theorem 7.4.6 The task of recovering an n-node ordered tree from its $2n$-bit encoding can be performed in $O(\log n)$ time on an MMB of size $n \times n$. Furthermore, this is time-optimal. □

7.5 Reconstruction of Trees from Traversals

The purpose of this section is to present $O(1)$ time algorithms for reconstructing binary and ordered trees from their traversals.

It is well-known that a binary tree can be reconstructed from its inorder traversal along with either its preorder or its postorder traversal [35]. Our first goal is to show that this task can be performed in $O(1)$ time on the MMB. The main idea of our algorithm is borrowed from Olariu *et al.* [52], where the reconstruction process was reduced to that of merging two sorted sequences.

Let T be an n-node binary tree. For simplicity, we assume that the nodes of T are $\{1, 2, \ldots, n\}$. Let c_1, c_2, ..., c_n and d_1, d_2, ..., d_n be the preorder and inorder traversals of T, respectively. We may think of $c_1, c_2, \ldots, c_n$ as $1, 2, \ldots, n$, the case where $c_1, c_2, \ldots, c_n$ is a permutation of $1, 2, \ldots, n$ reducing easily to this case [52]. In preparation for merging, we construct two sequences of triples. The first sequence is $(1, j_1, c_1), (1, j_2, c_2), \ldots, (1, j_n, c_n)$ such that $d_{j_i} = c_i$, $(i = 1, 2, \ldots, n)$. In other words, the second coordinate j_i of a generic triple represents the position of c_i in the inorder sequence $d_1, d_2, \ldots, d_n$. The second sequence consists of the triples $(2, 1, d_1), (2, 2, d_2), \ldots, (2, n, d_n)$ (Refer to Figure 7.3).
Denote by $\prod$ the set of triples

$$\{(1, j_1, c_1), (1, j_2, c_2), \ldots, (1, j_n, c_n), (2, 1, d_1), (2, 2, d_2), \ldots, (2, n, d_n)\},$$

and define a binary relation $\prec$ on $\prod$ as follows: for arbitrary triples (α, β, γ) and $(\alpha', \beta', \gamma')$ in $\prod$ we have:
Rule 1. $((\alpha = 1) \wedge (\alpha' = 1)) \rightarrow (((\alpha, \beta, \gamma) \prec (\alpha', \beta', \gamma')) \leftrightarrow (\gamma < \gamma'))$;
Rule 2. $((\alpha = 2) \wedge (\alpha' = 2)) \rightarrow (((\alpha, \beta, \gamma) \prec (\alpha', \beta', \gamma')) \leftrightarrow (\beta < \beta'))$;
Rule 3. $((\alpha = 1) \wedge (\alpha' = 2)) \rightarrow (((\alpha, \beta, \gamma) \prec (\alpha', \beta', \gamma')) \leftrightarrow ((\beta \leq \beta') \vee (\gamma \leq \gamma')))$.
In view of the rather forbidding aspect of Rules 1, 2 and 3, an explanation is in order. First, note that Rules 1 and 2 confirm that with respect to the relation $\prec$ both sequences

$$(1, j_1, c_1), (1, j_2, c_2), \ldots, (1, j_n, c_n) \text{ and } (2, 1, d_1), (2, 2, d_2), \ldots, (2, n, d_n)$$

are *sorted.* Intuitively, Rule 3 specifies that in the preorder-inorder traversal any pair of distinct labels u and v must occur in the order "...u...v...v...u..." or "...u...u...v...v..." [52] (refer to Figure 7.3).

Consider the sequence e_1, e_2, ..., e_{2n} obtained by extracting the third coordinate of the triples in the sequence resulting from merging the two sequences above. The following fundamental result proved in [52] will justify our algorithmic approach.

Proposition 7.5.1 [52] The sequence $e_1, e_2, \ldots, e_{2n}$ is the preorder-inorder traversal of T. □

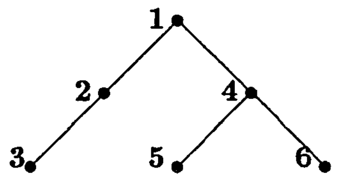
We now present the details of an algorithm that, starting with the preorder and inorder traversals of a binary tree T, produces the preorder-inorder traversal e_1, e_2, ..., e_{2n} of T. As we shall see, once this sequence is available, T can be reconstructed in O(1) time.

Let $c_1 = 1$, $c_2 = 2$, ..., $c_n = n$ and d_1, d_2, ..., d_n be the preorder and inorder traversals of a binary tree. We assume that these sequences are stored in the first row of an MMB of size $n \times 2n$ in left to right order, with the c_i's stored to the left of the d_i's. It is easy to modify the algorithm to work on a mesh of size $n \times n$. To construct the sets of triples discussed above, every processor storing c_i needs to determine the position of the second copy of c_i in the inorder traversal. Notice that every processor storing a d_j can construct the corresponding triple without needing any further information. The details follow.

Step 1. Begin by replicating the contents of the first row throughout the mesh. This is done by tasking every processor $P(1,i)$ to broadcasts the item it stores on the bus in its own column. Every processor reads the bus and stores the value broadcast.

Step 2. Every processor $P(i,i)$ $(1 \leq i \leq n)$ broadcasts c_i on the bus in row i. The unique processor storing the second copy of label c_i will inform $P(i,i)$ of its position in the inorder sequence. A simple data movement now sends this information to $P(1,i)$. Clearly, at the end of Step 2, every processor in the first row of the mesh can construct the corresponding triple.

Step 3. Merge the two sequences of triples using Proposition 3.4 and store the result in the first row of the mesh. Finally, every processor retains the third coordinate of the triple it receives by merging.



Preorder Traversal: 1,2,3,4,5,6

Inorder Traversal: 3,2,1,5,4,6

Sequence 1: (1,3,1),(1,2,2),(1,1,3),(1,5,4),(1,4,5),(1,6,6)

Sequence 2: (2,1,3),(2,2,2),(2,3,1),(2,4,5),(2,5,4),(2,6,6)

Merged Sequence: (1,3,1),(1,2,2),(1,1,3),(2,1,3),(2,2,2),(2,3,1)

(1,5,4),(1,4,5),(2,4,5),(2,5,4),(1,6,6),(2,6,6)

Preorder-Inorder Traversal: 1,2,3,3,2,1,4,5,5,4,6,6

Figure 7.3: Example of binary tree reconstruction

The correctness of the algorithm follows immediately from Proposition 6.1. Since all steps take O(1) time, we have proved the following result.

Lemma 7.5.2 Given the preorder and inorder traversals of an n-node binary tree, the corresponding preorder-inorder traversal can be constructed in O(1) time on an MMB of size $n \times n$. □

Our next goal is to show that once the preorder-inorder traversal $e_1, e_2, \ldots, e_{2n}$ is available, the corresponding binary tree can be reconstructed in O(1) time. Recall, that every label of a node in T occurs twice in the preorder-inorder traversal. Furthermore, by virtue of Step 2 above, the first copy of a label knows the position of its duplicate, and vice-versa.

We associate a node u with every pair of identical labels in $e_1, e_2, \ldots, e_{2n}$. Let e_i and e_j be the first and second copy of a given label. The processor holding e_i assigns children pointers as follows:

• if e_{i+1} is the first copy of a label v, then left(u) $\leftarrow v$; otherwise, left(u) $\leftarrow$ **nil**;
• if e_{j+1} is the first copy of a label w, then right(v_i) $\leftarrow w$; otherwise, right(v_i) $\leftarrow$ **nil**.

The setting of pointers takes O(1) time. Therefore, Lemma 7.5.2 implies the following result.

Theorem 7.5.3 An n-node binary tree can be reconstructed from its preorder and inorder traversals in constant time on an MMB of size $n \times n$. □

An ordered tree is an object that is either empty, or it consists of a root along with a possibly empty list $T_1, T_2, \ldots, T_k$ of subtrees, enumerated from left to right. Every node in an ordered tree stores a pointer to its leftmost child along with a pointer to its right sibling. The purpose of this section is to show that given its preorder and postorder traversals, an n-node ordered tree can be reconstructed in O(1) time on an MMB of size $n \times n$. We are presenting a slightly more general result, namely we show how to reconstruct an ordered forest from its preorder and postorder traversals.

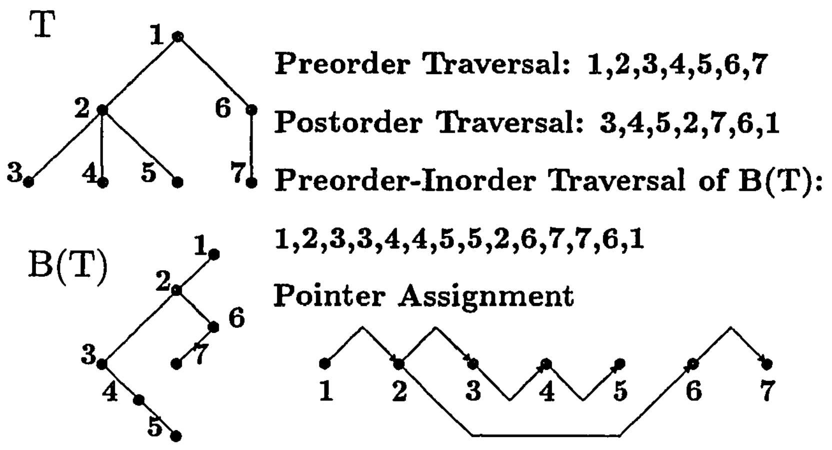
Our algorithm relies on the well-known one-to-one correspondence between n-node ordered forests and n-node binary trees [35]. Specifically, let $F = (T_1, T_2, \ldots, T_m)$ be an ordered forest. The binary tree $B(F)$ corresponding to F is either empty (in case F is empty), or else is defined as follows:

• the root of $B(F)$ is $root(T_1)$;
• the left subtree of $B(F)$ is $B(T_{11}, T_{12}, \ldots, T_{1k})$, where $T_{11}, T_{12}, \ldots, T_{1k}$ are the subtrees of $root(T_1)$;
• the right subtree of $B(F)$ is $B(T_2, \ldots, T_m)$.

The following result is well-known [35].

Proposition 7.5.4. F and $B(F)$ have the same preorder traversal. Furthermore, the postorder traversal of F is precisely the inorder traversal of $B(F)$. □
Proposition 7.5.4 motivates the following natural approach to reconstruct an ordered forest F from its preorder and postorder traversals. First, interpret the two traversals of F as the preorder and inorder traversals of the corresponding binary tree $B(F)$. Using the algorithm discussed in the previous section reconstruct $B(F)$. Finally, convert $B(F)$ to F.

We now present the details of the implementation of our forest reconstruction algorithm on an MMB of size $n \times 2n$. It is easy to modify the algorithm to work on an MMB of size $n \times n$. We assume that the preorder and postorder traversals of an ordered forest F are stored in the first row of the mesh in left to right order. Our algorithm proceeds as follows.
Step 1. Reconstruct the binary tree $B(F)$ having the same preorder traversal as F and whose inorder traversal corresponds to the postorder traversal of F;
Step 2. Let u be a generic node in $B(F)$; the processor in charge of u reinterprets pointers as follows:
- if left(u)=v then set l_child(u) $\leftarrow v$;
- if right(u)=v then set r_sibling(u) $\leftarrow v$.



Preorder Traversal: 1,2,3,4,5,6,7
Postorder Traversal: 3,4,5,2,7,6,1
Preorder-Inorder Traversal of B(T):
1,2,3,3,4,4,5,5,2,6,7,7,6,1
Pointer Assignment

Figure 7.4: Example of ordered tree reconstruction

Figure 5 illustrates the reconstruction of an ordered tree from its traversals. The upper arrows indicate l_child pointers and the lower arrows indicate r_sibling pointers. The correctness of our algorithm is easy to see. Furthermore, by Theorem 7.5.3 the running time is O(1). Consequently, we have proved the following result.

Theorem 7.5.5. An n-node ordered forest stored in the first row of an MMB of size $n \times n$ can be reconstructed from its preorder and postorder traversals in

O(1) time. □

7.6 Conclusions

In this chapter, we have presented a number of time-optimal tree algorithms on meshes with multiple broadcasting. Specifically, we have shown that the following tasks can be solved in $\Theta(\log n)$ time:
• Encode an n-node binary tree into a $2n$-bitstring;
• Encode an n-node ordered tree into a $2n$-bitstring;
• Recover an n-node binary tree from its $2n$-bit encoding;
• Recover an n-node ordered tree from its $2n$-bit encoding.
We have also shown that the following tasks can be performed in O(1) time:
• Reconstruct an n-node binary tree from its preorder and inorder traversals;
• Reconstruct an n-node ordered tree (forest) from its preorder and postorder traversals.

Our algorithms rely heavily on time-optimal algorithms for sequences of parentheses that we developed. Specifically, we have shown that each of the following tasks can be solved in $\Theta(\log n)$ time:
• determining whether a sequence of n parentheses is well-formed;
• finding all the matching pairs in a well-formed sequence of parentheses;
• determining the closest enclosing pair for every matching pair in a well-formed sequence.

A number of problems are open. In particular, it is not known whether reconstructing an ordered tree in parent-pointer format can be done in less than $O(\log n)$ time. It is clear that such an algorithm using the closest enclosing pair can be devised. However, it is not known whether one can do better.

Chapter 8

Conclusions

8.1 Concluding Remarks

In this thesis, we considered a number of fundamental problems on the MMB's and presented time-optimal algorithms. In many cases, we also presented relevant applications.

In Chapter 1 and Chapter 2 we presented the details of the model of computation and relevant background.

In Chapter 3, we proposed a unifying look at semigroup computations on meshes with multiple broadcasting. Specifically, we provided a lower bound by showing that any algorithm which solves the semigroup computation problem must take at least $\Omega\left(\max\{\min\{\log m, \log \frac{n^{\frac{2}{3}}}{m^{\frac{1}{3}}}\}, \frac{m^{\frac{1}{3}}}{n^{\frac{1}{6}}}\}\right)$ time. We also have shown that our lower bound is tight by designing an algorithm whose running time matches the lower bound.

In Chapter 4 we addressed the problem of sorting in general case. We have presented a time- and VLSI-optimal sorting algorithm for meshes with multiple broadcasting. Specifically, we have shown that once we fix a positive integer constant c, we can sort m items in the range $n^{\frac{1}{2}+\frac{1}{2c}} \leq m \leq n$ in $O(\frac{m}{\sqrt{n}})$ time.

In Chapter 5, we proposed a time-optimal algorithm for the multiple search problem and showed that a number of problems in computer graphics, image processing, robotics, and computational geometry reduce to the multiple search problem or a variant thereof. More specifically, our algorithm runs in $O(\sqrt{m})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh with multiple broadcasting. We also showed that this achieves the theoretical time lower bound for the problem. Note that the running time of our algorithm is independent of n, and thus the algorithm is adaptive.

In chapter 6, we addressed a number of convexity related problems in sparse case. For two n-vertex convex polygons P and Q, we have established $\Omega(\log\ n)$ time lower bounds for the following problems: computing the area of P; computing the diameter of P;computing the width of P; computing the modality of P; computing the smallest area rectangle enclosing P; computing a maximum-area inscribed triangle sharing an edge with P; computing the maximum distance between P and Q; detecting whether or not P is contained in the interior of Q. We have shown that the bounds are tight by providing $O(\log n)$ algorithms to accomplish these tasks on meshes with multiple broadcasting of size $n \times n$. We have also shown that the problem of computing the minimum distance between two separable n-vertex polygons stored in one row/column of a mesh with multiple broadcasting of size $n \times n$ can be solved in $O(1)$ time.

In chapter 7, we presented time-optimal algorithms for a number of parentheses and tree problems. Specifically, we have shown that each of the following tasks can be solved in $\Theta(\log n)$ time: determining whether a sequence of n parentheses is well-formed; finding all the matching pairs in a well-formed sequence of parentheses; determining the closest enclosing pair for every matching pair in a well-formed sequence; encoding an n-node binary tree into a $2n$-bitstring; encoding an n-node ordered tree into a $2n$-bitstring; recovering an n-node binary tree from its $2n$-bit encoding; recovering an n-node ordered tree from its $2n$-bit encoding. We have also shown that the following tasks can be performed in $O(1)$ time: reconstruct an n-node binary tree from its preorder and inorder traversals; reconstruct an n-node ordered tree (forest) from its preorder and postorder traversals.

However, a number of open problems remain. They are discussed in the following section.

8.2 Open Problems

As we have done for the semigroup computation problem, similar unifying results are desirable for important algorithmic problems on meshes with multiple broadcasting. Candidate problems include, prefix computation, selection, sorting, and list ranking among many others. As far as sorting is concerned we would like to completely resolve these issues concerning optimal sorting over the entire range $\sqrt{n} \leq m \leq n$. Note that the results of Lin and others [40] show that for m near $\sqrt{n}$, $\Omega(\log n)$ is the time lower bound for sorting in this architecture. Their results imply that a sorting algorithm cannot be VLSI-optimal for m near $\sqrt{n}$. It would be of interest to know whether the multiple search problem can be solved optimally even when the input data is not sorted.

In the sparse case, we have addressed some problems related to convexity, and some parentheses and tree problems. This leads to many interesting questions thus opening up an entire area for research.

As the work presented in this thesis addresses the general case for the first time, this opens an area for further work. In both general and dense cases, it would be of interest to prove more lower bounds. It would be interesting to see if the counting argument developed in this thesis for proving lower bounds can be used in other contexts as well.

Bibliography

[1] M. J. Atallah and M. T. Goodrich, Parallel Algorithms for Some Functions of Two Convex Polygons, *Algorithmica* 3, 1988, 535–548.

[2] A. Aggarwal, Optimal Bounds for Finding Maximum on Array of Processors With k Global Buses, *IEEE Trans. on Computers*, C-35, 1986, 62–64.

[3] S. G. Akl, The Design and Analysis of Parallel Algorithms, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[4] S. G. Akl and J. Meijer, Parallel Binary Search, *IEEE Trans. on Parallel and Distributed Systems*, 1, 1990, 247–250.

[5] G. S. Almasi and A. Gottlieb, Highly Parallel Computing, Benjamin/Cummings, Redwood City, California, 1990.

[6] A. Aggarwal and R. C. Melville, Fast Computation of the Modality of Polygons, *Journal of Algorithms*, 7, 1986, 369–381.

[7] M. Atallah and S. E. Hambrush, Solving Tree Problems on a Mesh Connected Processor Array, *Proc.* 26th *Symp. Foundations of Computer Science*, 1985, 222–231.

[8] A. Bar-Noy and D. Peleg, Square Meshes are not Always Optimal, *IEEE Trans. on Computers*, C-40, 1991, 196–204.

[9] D. H. Ballard and C. M. Brown, Computer Vision, Prentice-Hall, 1982.

[10] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin, Highly Parallelizable Problems, *Proc.* 21st *Annual ACM Symp. on the Theory of Computing*, 1989, 770–785.

[11] W.-E. Blanz, D. Petkovic, and J. L. C. Sanz, Algorithms and Architectures for Machine Vision, in C. H. Chen, ed., Signal Processing Handbook, M. Dekker, New York, 1989.

[12] K. E. Batcher, Design of Massively Parallel Processor, *IEEE Trans. on Computers*, C-29, 1980, 836–840.

[13] D. Bhagavathi, P. J. Looges, S. Olariu, J. L. Schwing, and J. Zhang, A Fast Selection Algorithm on Meshes with Multiple Broadcasting, *Proc. International Conference on Parallel Processing*, St-Charles, Illinois, 1992, III-10-17.

[14] D. Bhagavathi, S. Olariu, W. Shen, and L. Wilson, A Time-Optimal Multiple Search Algorithm on Enhanced Meshes, with Applications, *Proc. Fourth Canadian Computational Geometry Conference*, St-Johns, August 1992, 359–364.

[15] D. Bhagavathi, S. Olariu, J. L. Schwing, and J. Zhang, Convex Polygon Problems on Meshes with Multiple Broadcasting, *Parallel Processing Letters*, to appear.

[16] D. Bhagavathi, S. Olariu, W. Shen, and L. Wilson, A Time-Optimal Multiple Search Algorithm on Enhanced Meshes, with Applications, *Journal of Parallel and Distributed Computing*, to appear.

[17] D. Bhagavathi, H. Gurla, S. Olariu, J. L. Schwing, and J. Zhang, Square Meshes or not Optimal for Convex Hull Computation, *Proc. International Conference on Parallel Processing*, St-Charles, Illinois, 1992, III-10-17.

[18] S. H. Bokhari, Finding Maximum on an Array Processor with a Global Bus, *IEEE Trans. on Computers* C-33, no. 2, Feb. 1984, 133–139.

[19] B. K. Bhattacharya and G. T. Toussaint, Efficient Algorithms for Computing the Maximum Distance Between two Finite Planar Sets, *Journal of Algorithms*, 4, 1983, 121–126.

[20] R. Cahn, R. Poulsen, and G. Toussaint, Segmentation of Cervical Cell Images, *Journal of Histochemistry and Cytochemistry*, 25, 1977, 681–688.

[21] Y. C. Chen, W. T. Chen, G. H. Chen and J. P. Shen, Designing Efficient Parallel Algorithms on Mesh Connected Computers with Multiple Broadcasting, *IEEE Trans. Parallel and Distributed Systems,* 1, no. 2, Apr. 1990.

[22] Y. C. Chen, W. T. Chen, G. H. Chen, Efficient Median Finding and its Applications to Two-Variable Linear Programming on Mesh Connected Computers

with Multiple Broadcasting, *Journal of Parallel and Distributed Computing*, 15, 1992, 79–84.

[23] F. Chin and C. A. Wang, Optimal Algorithms for the Minimum Distance Between two Separated Convex Polygons, University of Alberta, Tech. Report, Jan. 1983.

[24] S. A. Cook, C. Dwork, and R. Reischuk, Upper and Lower Time Bounds for Parallel Random Access Machines Without Simultaneous Writes, *SIAM Journal on Computing*, 15, 1986, 87–97.

[25] E. R. Dougherty and C. R. Giardina, Mathematical Methods for Artificial Intelligence and Autonomous Systems, Prentice-Hall, Englewood Cliffs, 1988.

[26] D. P. Dobkin and L. Snyder, On a General Method for Maximizing and Minimizing Among Certain Geometric Problems, *Proc. 20th Annual Symp. on Foundations of Computer Science*, 1979, 9–17.

[27] R. O. Duda and P. E. Hart, Pattern Classification and Scene Analysis, Wiley and Sons, New York, 1973.

[28] H. Freeman and R. Shapira, Determining the Minimum Area Encasing Rectangle for an Arbitrary Closed Curve, *Communications of the ACM*, 18, 1975, 409–413.

[29] W. M. Gentleman, Some Complexity Results for Matrix Computations on Parallel Processors, *Journal of ACM*, 25, 1978, 112–115.

[30] H. Gurla, Leftmost One Computation of Meshes with Broadcasting, *Info. Processing Letters*, to appear.

[31] J. L. Hennessy and D. A. Patterson, Computer Architecture, A Quantitative Approach, Morgan Kaufmann Publishers, San Manteo, 1990.

[32] M. E. Houle and G. T. Toussaint, Computing the width of a set, *Proc. First Annual ACM Symp. on Computational Geometry*, 1985, 1–7.

[33] C. S. Jeong and D. T. Lee, Parallel Convex Hull Algorithms in 2- and 3-dimensions on Mesh-Connected Computers, *Algorithmica*, 5, 1990, 155–177.

[34] H. F. Sawyer and P. L. Sawyer, A Multiprocessor System for Finite Element Structural Analysis, *Comput. Struct.* , (10) 1979, 21–29

[35] D. Knuth, The Art of Computer Programming: Fundamental Algorithms, 1, 2nd edition, Addison-Wesley, Reading, 1973.

[36] R. M. Karp and V. Ramachandran, A Survey of Parallel Algorithms for Shared Memory Machines, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., North-Holland, 1990, 869–941.

[37] V. P. Kumar and C. S. Raghavendra, Array Processor with Multiple Broadcasting, *Journal of Parallel and Distributed Computing,* 2, 1987, 173–190.

[38] F. Thomson Leighton, Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, Morgan Kaufmann Publishers, San Mateo, 1992.

[39] H. Li and M. Maresca, Polymorphic-Torus Network, *IEEE Trans. on Computers,* C-38, no. 9, 1989, 1345–1351.

[40] R. Lin, S. Olariu, J. L. Schwing, and J. Zhang, Simulating Enhanced Meshes, with Applications, *Parallel Processing Letters*, to appear.

[41] T. Lozano-Perez, Spatial Planning: A Configurational Space Approach, *IEEE Trans. on Computers*, C-32, 1983, 108-119.

[42] M. Lu, Constructing the Voronoi Diagram on a Mesh-Connected Computer, *Proc. of the International Conference on Parallel Processing,* 1985, 806-811.

[43] M. Maresca and H. Li, Connection Autonomy and SIMD Computers: a VLSI Implementation, *Journal of Parallel and Distributed Computing,* 7, 1989, 302-320.

[44] D. Nassimi and S. Sahni, Bitonic Sort on a Mesh-Connected Parallel Computer, *IEEE Trans. on Computers*, C-27, 1979.

[45] D. Nassimi and S. Sahni, Finding Connected Components and Connected Ones on a Mesh-Connected Parallel Computer, *SIAM Journal on Computing*, 9, 1980, 744–757.

[46] S. Olariu, On the Unimodality of Convex Polygons, *Information Processing Letters*, 29, 1988, 289–292.

[47] S. Olariu, A Simple Linear-Time Algorithm for Computing the RNG and MST of Unimodal Polygons, *Information Processing Letters*, 31, 1989, 243–247.

[48] S. Olariu, The Morphology of Convex Polygons, *Computers and mathematics, with Applications*, 24, 1992, 59–68.

[49] S. Olariu, J. L. Schwing, and J. Zhang, Time-Optimal Sorting and Applications on $n \times n$ Enhanced Meshes, *Proc. IEEE International Conf. on Computer Systems and Software Engineering*, The Hague, May 1992.

[50] S. Olariu, J. L. Schwing, and J. Zhang, Optimal Parallel Encoding and Decoding Algorithms for Trees, *International Journal of Foundations of Computer Science*, 3, 1992, 1–10.

[51] S. Olariu, J. L. Schwing, and J. Zhang, Convex Hull Computation on Enhanced Meshes, Department of Computer Science, Old Dominion University, Technical Report TR-92-23, June 1992.

[52] S. Olariu, C. M. Overstreet, and Z. Wen, Parallel Reconstruction of Binary Trees, *Journal of Parallel and Distributed Computing*, to appear.

[53] T. Pavlidis, Computer Graphics, Computer Science Press, Potomac, MD, 1978.

[54] B. Preas and M. Lorenzetti, Eds., Physical Design and Automation of VLSI Systems, Benjamin/Cummings, Menlo Park, 1988.

[55] D. Parkinson, D. J. Hunt, and K. S. MacQueen, The AMT DAP 500, 33rd *IEEE Comp. Society International Conferrence*, Feb. 1988, 196–199.

[56] V. K. P. Kumar and D. I. Reisis, Image Computations on Meshes with Multiple Broadcast, *IEEE Trans. Pattern Analysis and Machine Intelligence,* 11, No. 11, 1989, 1194–1202.

[57] F. P. Preparata and M. I. Shamos, Computational Geometry, An Introduction, Springer-Verlag, New York, Berlin, 1988.

[58] A. Rosenfeld and A. Kak, Digital Picture Processing, Academic Press, 1-2, 1982.

[59] J. Rothstein, Bus Automata, Brains, and Mental Models, *IEEE Trans. on Systems Man Cybernetics* 18, 1988.

[60] H. S. Stone, High-Performance Computer Architecture, Second, Edition, Addison-Wesley, Reading, MA, 1990.

[61] Q. F. Stout, Mesh Connected Computers with Broadcasting, *IEEE Trans on Computers,* C-32, 826-830, Sept. 1983.

[62] Q. F. Stout, Meshes with Multiple Buses, in *Proc. 27th IEEE Symp. Foundations of Comput. Sciences* 1986, 264–273.

[63] D. Tabak, Multiprocessors, Prentice-Hall, Englewood Cliffs, New Jersey, 1990

[64] C. D. Thompson, The VLSI Complexity of Sorting, *IEEE Trans. on Computers*, C-32, 1983, 1171–1184.

[65] C. D. Thompson and H. T. Kung, Sorting on a Mesh-Connected Parallel Computer, *Comm. of the ACM*, 20, 1977, 263–271.

[66] G. T. Toussaint Ed., Computational Geometry, Elsevier Science Publishers, North-Holland, 1985.

[67] G. T. Toussaint, Complexity, Convexity and Unimodality, *International Journal of Computing and Information Sciences* 13, 1984, 197–217.

[68] G. T. Toussaint, Computational Morphology, North-Holland, Amsterdam, 1988.

[69] R. E. Tarjan and U. Vishkin, An Efficient Parallel Biconnectivity Algorithm, *SIAM Journal on Computing*, 14 1985, 862–874.

[70] J. Ullman, Computational Aspects of VLSI, Computer Science Press, Potomac, MD, 1984.

[71] D. Vernon, Machine Vision, Automated Visual Inspection and Robot Vision, Prentice-Hall, 1991.

[72] Z. Wen, Parallel Multiple Search, *Info. Proc. Letters*, 37, Feb. 1991, 181–186.

[73] S. Zaks, Lexicographic Generation of Ordered Trees, *Theoretical Computer Science*, 10 (1980), 63–82.

VITA

I was born in Warangal, India, in August 1966. I was raised in Hyderabad. I received a State Merit Award and a national Merit Award for outstanding performance in high school and junior college. I obtained my B.E. in Electrical Engineering, in 1987 from Osmania University and my M.Tech in Computer Science from University of Hyderabad, in 1989. I worked with the Real-Time Systems Group in CMC, Hyderabad before joining the Ph.D. program at Old Dominion University in 1990.

I worked with Dr. Stephan Olariu in the area of Parallel Algorithms. For my dissertation, I developed Time-Optimal Algorithms on Meshes with Multiple Broadcasting. My other research interests include image processing, Computational Geometry and Parallel Algorithms for the PRAM model. During my Ph.D. program I worked as a Teaching Assistant for the first two years and was a Special Doctoral Research Assistant for one year. During my Ph.D. program I published a number of papers in various conferences and journals.