Computer Science Theses & Dissertations   Computer Science

Winter 1992

High Performance Issues in Image Processing and Computer Vision

Jingyuan Zhang
Old Dominion University

Follow this and additional works at: https://digitalcommons.odu.edu/computerscience_etds

Part of the Computer Sciences Commons

HIGH PERFORMANCE ISSUES IN IMAGE PROCESSING

AND COMPUTER VISION

by

Jingyuan Zhang
B.S., July 1984, Shandong University, China
M.S., July 1987, Zhejiang University, China

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
December, 1992

Approved by:

Stephan Olariu (Advisor)

James L. Schwing (Advisor)

Larry Wilson

Chester E. Grosch

Przemysław Bogacki

ABSTRACT

HIGH PERFORMANCE ISSUES IN IMAGE PROCESSING AND COMPUTER VISION

Jingyuan Zhang

Old Dominion University

Advisors: Drs. Stephan Olariu and James L. Schwing

Typical image processing and computer vision tasks found in industrial, medical, and military applications require real-time solutions. These requirements have motivated the design of many parallel architectures and algorithms. Recently, a new architecture called the reconfigurable mesh has been proposed. This thesis addresses a number of problems in image processing and computer vision on reconfigurable meshes.

We first show that a number of low-level descriptors of a digitized image such as the perimeter, area, histogram and median row can be reduced to computing the sum of all the integers in a matrix, which in turn can be reduced to computing the prefix sums of a binary sequence and the prefix sums of an integer sequence. We then propose a new computational paradigm for reconfigurable meshes, that is, identifying an entity by a bus and performing computations on the bus to obtain properties of the entity. Using the new paradigm, we solve a number of mid-level vision tasks including the Hough transform and component labeling. Finally, a VLSI-optimal constant time algorithm for computing the convex hull of a set of planar points is presented based on a VLSI-optimal constant time sorting algorithm.

As by-products, two basic data movement techniques, computing the prefix sums of a binary sequence and computing the prefix maxima of a sequence of real numbers, and a VLSI-optimal constant time sorting algorithm have been developed. These by-products are interesting in their own right. In addition, they can be exploited to obtain efficient algorithms for a number of computational problems.

ACKNOWLEDGEMENTS

This work could not have been completed without the help of many individuals, to whom I would like to express my appreciation. First and foremost, I would like to thank my advisors, Drs. Stephan Olariu and James Schwing, who have put a great deal of time and effort into the guidance of this work including read the first draft and suggesting a number of changes that resulted in a better presentation.

Next, I would like to convey my sincere thanks to the other members of my dissertation committee, Drs. Larry Wilson, Chester Grosch and Przemyslaw Bogacki. Their expertise, thorough reviewing and valuable suggestions have also led to a greatly improved dissertation.

I wish to extend my appreciation to the faculty of this department, especially Dr. Kurt Maly, the department chair, and Dr. Michael Overstreet, the graduate program director, for providing a stimulating research environment. I also wish to thank many of my fellow students for their assistance.

Special thanks go to Dr. Rong Lin at SUNY Geneseo for his help and advice during the development of this work.

Finally, I thank my wife, Ping, and my son, Mike, for their understanding and patience during the many evenings and weekends required to complete this work.

ii

TABLE OF CONTENTS

iv

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Recent advances in VLSI have made it possible to build massively parallel machines featuring many thousands of cooperating processors. Among these, the mesh-connected computer architecture has emerged as a natural choice for solving image processing and computer vision tasks. Its regular structure and simple interconnection topology makes the mesh particularly well suited for VLSI implementation. In addition, matrices and digitized images map naturally onto the mesh. It comes as no surprise, therefore, that many image processing and computer vision algorithms have been reported on mesh-connected computers [7,8,11,28,33,35,37,52,53].

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 [1,4,46,58,59]. A number of efficient image processing and computer vision algorithms on meshes with a bus system have been reported in the recent literature [12,46,47,51]. A common feature of these bus structures is that they are static in nature, which means that the communication patterns among processors cannot be modified during the execution of the algorithm.

Typical computer and robot vision tasks found currently in industrial, medical, and military applications involve digitized images featuring millions of pixels. The large amount of data contained in these images, combined with real-time processing requirements have motivated researchers to consider adding reconfigurable features to high-performance computers. This has motivated a number of bus systems whose configuration can change, under program control, to be proposed in the literature: such

1

a bus system is referred to as *reconfigurable*. The *bus automaton* |56|, the *reconfigurable mesh* [30,31], and the *polymorphic torus* [21,29] are examples of architectures with a reconfigurable bus system. Quite recently a number of image processing and computer vision algorithms on reconfigurable architectures have been proposed [14-16,22,32].

In this thesis, we will solve a number of image processing and computer vision problems on reconfigurable meshes. Among the low-level vision tasks, we will address the problem of computing the perimeter, area, histogram and median row of a digitized image. Among the mid-level vision tasks, we will address the Hough transform and component labeling. We will also address the problem of computing the convex hull of a set of planar points.

1.1. The Computational Model

The computational model used throughout this work is the *reconfigurable mesh*.[*] An $M \times N$ reconfigurable mesh consists of MN identical processors positioned on a rectangular array (refer to Figure 1.1). The processor located in row i and column j $(1 \leq i \leq M; 1 \leq j \leq N)$[**] is referred to as $P(i,j)$. Every processor has 4 ports denoted by N, S, E, and W. In each processor, ports can be dynamically connected in pairs to suit computational needs. Our computational model only allows two connections to be set in each processor. Furthermore, these two connections must involve disjoint pairs of ports (see Figure 1.1). In the absence of these local connections, the reconfigurable mesh is functionally equivalent to the mesh connected computer.

We assume that the processing elements have a constant number of registers of $O(\log MN)$ bits and a very basic instruction set which allows a processor to perform standard arithmetic and boolean operations in unit time. We assume a SIMD model: in each time unit the same instruction is broadcast to all processors, which execute it

---

[*] When no confusion is possible a reconfigurable mesh will be referred to simply as a mesh.
[**] For convenience, the row (column) number sometime ranges from 0 to $M-1$ $(N-1)$.

2

and wait for the next instruction. Each instruction can consist of performing an arithmetic or boolean operation, setting local connections, broadcasting a value on a bus, or reading a value from a specified bus. The regular structure of the reconfigurable mesh makes it suitable for VLSI implementation [30,31]. In fact, it has been argued [30] that the reconfigurable mesh can be used as a universal chip capable of simulating any equivalent-area architecture without loss of time.
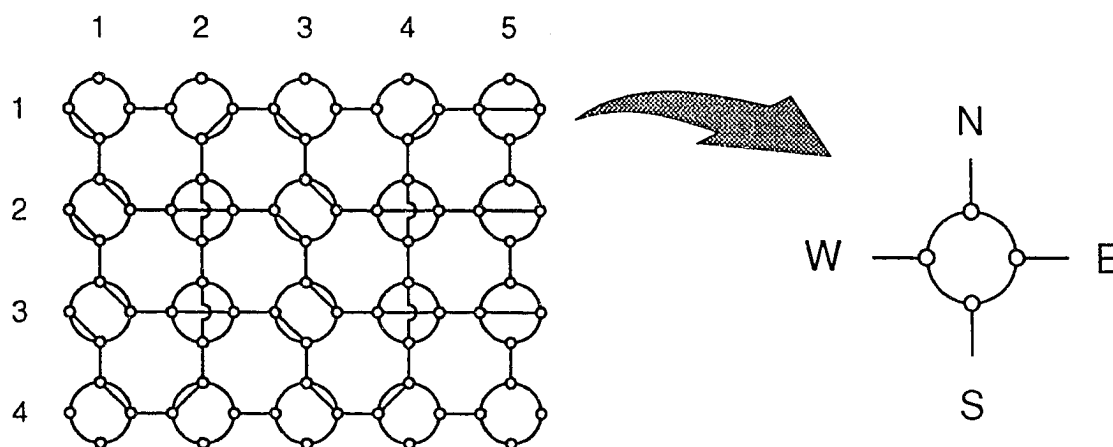
Figure 1.1: A reconfigurable mesh of size 4×5

By adjusting the local connections within each processor, several subbuses can be established. We assume that the setting of local connection is destructive in that setting a new pattern of connections destroys the previous one. At any given time, only one processor can broadcast a value onto a bus. Processors, if instructed to do so, can read any bus which passes through one of the processor's ports. In accord with other researchers [4,14-16,21,22,30-32,46,47,58,59], we assume that communications along buses take O(1) time. This seems to be a reasonable assumption in the light of recent experiments with the YUPPIE system [29].

3

1.2. Problems of Interest

We are interested in addressing a number of image processing and computer vision problems on reconfigurable meshes. Computer vision deals with scene analysis to obtain results similar to those obtained by man. A simplified computer vision paradigm consists of two computational stages [20]. The first, which is concerned with low-level and mid-level techniques, is referred to as image processing. The second, which is concerned with high-level techniques, is termed image interpretation and provides a symbolic output which describes the contents of the scene. In this thesis, we study the following image processing and computer vision problems on reconfigurable meshes: computing the perimeter, area, histogram and median row among the low-level vision tasks, the Hough transform and component labeling among the mid-level vision tasks as well as computing the convex hull of a set of planar points.

Chapter 2 is devoted to the problems of computing the perimeter, area, histogram and median row of a digitized image. These seemingly unrelated low-level computer vision tasks become related since they can be solved using the same technique. Specifically, these problems can be reduced to the problem of computing the sum of all values of a binary matrix. To compute the sum of all values of a binary matrix, we first show how to compute the prefix sums of a binary sequence, and then how to compute the prefix sums of an integer sequence.

In Chapter 3, we study the problem of the Hough transform and component labeling in the mid-level vision tasks. These problems are solved by using a novel computational paradigm, i.e. identifying entities such as lines and regions with buses and performing computations on these buses to obtain properties of the entities. The computations on buses include finding the maximum on an open bus (therefore, electing a leader of a closed bus), ranking an arbitrary open bus, and computing the prefix maxima (sums) on a bus.

4

Chapter 4 describes a VLSI-optimal constant time algorithm for computing the convex hull of a set of planar points, which was listed as one of the tasks of the first DARPA image understanding benchmark for parallel computers. This VLSI-optimal convex hull algorithm is based on the VLSI-optimal constant time sorting algorithm and is refined from our sub-optimal convex hull algorithm. In turn, the VLSI-optimal sorting algorithm is obtained from the multiple selection algorithm and a sub-optimal sorting algorithm.

Finally, Chapter 5 summarizes the results and proposes a number of open problems. The research results presented in this thesis can also be found in [23,38-45].

CHAPTER 2

LOW-LEVEL VISION TASKS

In this chapter, we will show that a number of low-level vision tasks such as computing the perimeter, area, histogram and median row of a digitized image can be performed in doubly logarithmic time on reconfigurable meshes. For this purpose, we first argue that the prefix sums of a binary sequence of size N can be computed in $O(\frac{\log N}{\log M})$ time on a reconfigurable mesh of size $M \times N$ with $2 \leq M \leq N$. This allows us to compute the prefix sums of a sequence of N integers in the range from 0 to N in $O(1)$ time on a reconfigurable mesh of size $N \times N$. Next, we show that the number of 1's in a binary matrix of size $N \times N$ can be computed in $O(\log\log N)$ time on a reconfigurable mesh of size $N \times N$. Finally, we reduce the problems of computing the perimeter, area, histogram and median row to computing the number of 1's in a binary matrix.

2.1. Prefix Sums of a Binary Sequence

Given a binary sequence $b_1, b_2, \ldots, b_N$, the prefix sum problem involves computing the sums of all the prefixes of that sequence, that is, b_1, $b_1 + b_2$, $b_1 + b_2 + b_3, \ldots, b_1 + b_2 + \cdots + b_N$.

For definiteness, we let z_j $(1 \leq j \leq N)$ denote the j-th prefix sum, that is, $z_j = b_1 + b_2 + \cdots + b_j$. We first show how to compute z_j in constant time on a reconfigurable mesh of size $(N+1) \times N$ by using the *configurational computation* paradigm introduced by Wang *et al.* [61]. We then extend the idea to solve the entire problem in $O(\frac{\log N}{\log M})$ time on a reconfigurable mesh of size $(M+1) \times 2N$ with

6

$2 \le M \le N$. Finally, we show that the same problem can be solved in $O(\frac{\log N}{\log M})$ time on a reconfigurable mesh of size $M \times N$ with $3 \le M \le N$.

To begin, assume a reconfigurable mesh of size $(N+1) \times N$ with the input sequence $b_1, b_2, \ldots, b_N$ stored by the processors in the first row, with $P(1,j)$ storing b_j for all j. If b_j is 0 then all processors in column j connect their ports W and E; if b_j is 1 then all processors in column j connect their ports W and S, as well as their ports N and E. Now $P(1,1)$ broadcasts a signal on the bus through its W port; it is easy to confirm that for $1 \le j \le N$, the row number of the unique processor in column j that observes the signal on its E port, equals $z_j + 1$. Refer to Figure 2.1 for an example: here, $N = 4$, the binary sequence is $(1,0,1,1)$, and the highlighted bus is the one on which the signal is broadcast.
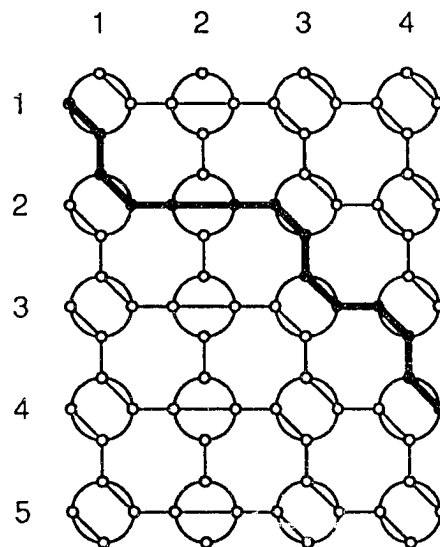
Figure 2.1: The buses formed for $(1,0,1,1)$ on a 5×4 mesh

We now assume a reconfigurable mesh R of size $(M+1) \times 2N$ $(2 \le M \le N)$. Notice that in this setup we may not have sufficient height to store the prefix sums as

7

described above. We will therefore adapt the previous algorithm by emulating modulo M arithmetic. We assume that the input is stored in the first row of R, with $P(1,2j-1)$ storing b_j for all $j=1, 2, \ldots, N$.

We now outline the basic idea of the algorithm. Begin by setting connections in the odd columns as described above. For the even columns, except for the top and bottom rows, connections are set as a cross (i.e. connect N to S, and E to W, with no intersection). Connections in row $M+1$ are set as follows: if $b_j=1$ then $P(M+1,2j)$ connects its ports W and N; if $b_j=0$, then no connections are set in $P(M+1,2j)$. The connections in the first row of R are set as follows: if $b_j=1$ then $P(1,2j)$ connects ports S and E; if $b_j=0$ then $P(1,2j)$ connects ports W and E. Figure 2.2 shows such an example for $M=2$, $N=4$, and input sequence $(1,0,1,1)$.
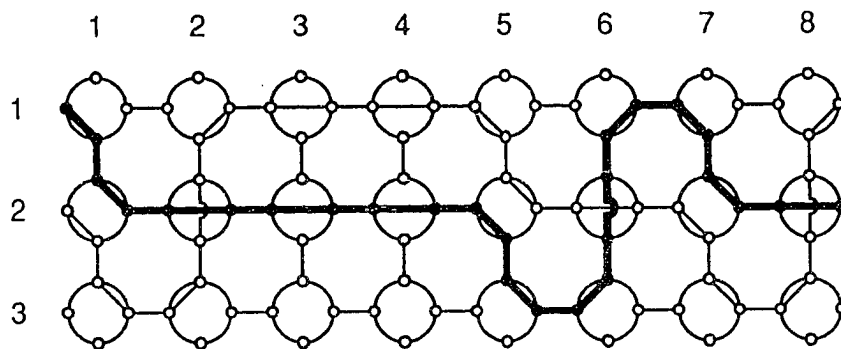
Figure 2.2: The buses formed for $(1,0,1,1)$ on a 3×8 mesh

As before, processor $P(1,1)$ broadcasts a signal along the bus containing its W port. It is easy to confirm that there exists a unique processor in column $2j$ ($1 \leq j \leq N$) that receives the broadcast signal from its ports E. Let r_j be the row number of this processor. Now set, for every j,

$$y_j^1 \leftarrow r_j - 1.$$

Moreover, it is clear that

$$y_j^1 = z_j \bmod M.$$

Next, mark each processor $P(1,2j)$ if a "wrap-around" occurred in column $2j$. We determine a new set of binary values by letting every processor $P(1,2j)$ set b_j^1 to 1 or 0 depending on whether or not $P(1,2j)$ is marked. It is worth to note that

$$z_j = (b_1^1 + \cdots + b_j^1)M + y_j^1.$$

The objective now becomes the computation of the prefix sums of the sequence of b_1^1, $b_2^1, \ldots, b_n^1$. It is further interesting to note that the number of times 1 occurs in the derived sequence $b_1^1, b_2^1, \ldots, b_n^1$ is a factor of M times less than in the original sequence. In addition, computing all the values y_j^1 as well as the new binary sequence takes O(1) time.

This process can be carried out iteratively, say t times, until there are less than M 1's in the new binary sequence, and therefore no "wrap-around" will occur. It is obvious from the above construction that

$$z_j = (\cdots (((0)M + y_j^t)M + y_j^{t-1}) \cdots)M + y_j^1.$$

Expanding we get

$$z_j = y_j^t M^{t-1} + \cdots + y_j^2 M + y_j^1$$

a base M expansion of z_j. This last equation yields the final insight into this algorithm. Namely, for the first iteration set

$$z_j^1 \leftarrow y_j^1$$

and thereafter set

$$z_j^k \leftarrow y_j^k M^{k-1} + z_j^{k-1}.$$

It is easy to confirm that z_j^t is exactly z_j.

To argue for the running time of the algorithm, observe that t is at most $\left\lfloor \dfrac{\log N}{\log M} \right\rfloor + 1$. To summarize our findings we state the following result.

Theorem 2.1. The prefix sums of a binary sequence of size N can be computed in $O(\dfrac{\log N}{\log M})$ time on a reconfigurable mesh of size $(M+1) \times 2N$ with $2 \leq M \leq N$. $\square$

9

We next illustrate the above concepts with the example featured in Figure 2.2. Here, $N=4$, $M=2$, and the input sequence is $(b_1, b_2, b_3, b_4) = (1,0,1,1)$. The highlighted bus in Figure 2.2 is the one on which the signal is broadcast. After the first iteration, the updated binary sequence $(b_1^1, b_2^1, b_3^1, b_4^1)$ reads $(0,0,1,0)$, the sequences $(y_1^1, y_2^1, y_3^1, y_4^1)$ and $(z_1^1, z_2^1, z_3^1, z_4^1)$ are both $(1,1,0,1)$. The buses formed in the second iteration are demonstrated in Figure 2.3. After the second iteration, the updated binary sequence $(b_1^2, b_2^2, b_3^2, b_4^2)$ is $(0,0,0,0)$, $(y_1^2, y_2^2, y_3^2, y_4^2)$ is $(0,0,1,1)$ and $(z_1^2, z_2^2, z_3^2, z_4^2)$ is $(1,1,2,3)$. At this point the computation ends and the prefix sums returned are $(z_1, z_2, z_3, z_4) = (1,1,2,3)$.
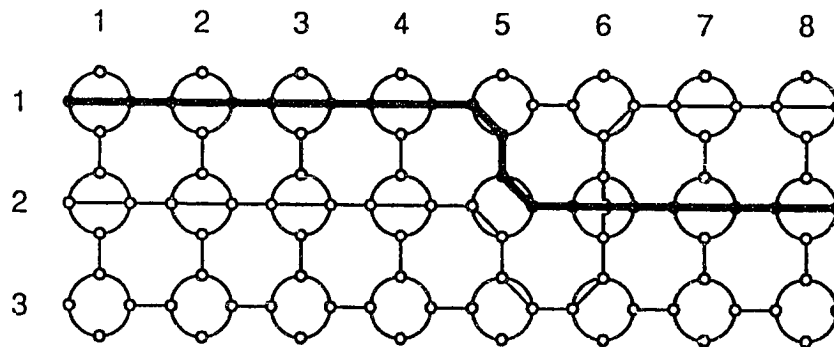
Figure 2.3: The buses formed for (0,0,1,0) on a 3×8 mesh

Note that the prefix sums of a binary sequence of size N can be computed in two stages on a reconfigurable mesh of size $(M+1)\times N$: we deal with the even and odd subsequences separately and combine the results in the obvious way. Note, further, that the computation can, in fact, be carried out on a reconfigurable mesh of size $M\times N$ by replacing modulo M arithmetic by modulo $(M-1)$ arithmetic.

In [30] it is argued that the prefix sums of an arbitrary sequence of N real numbers can be computed in $O(\log N)$ time in one row of a reconfigurable mesh of size $N\times N$. Now combining the result in [30] with our findings, we get the following

10

important theorem.

Theorem 2.2. On a reconfigurable mesh of size $M \times N$ with $M \leq N$ the prefix sums of an N-element binary sequence can be computed in $O(\log N)$ time if $M = 1$, and in $O(\frac{\log N}{\log M})$ time if $M > 1$. $\square$

The following results can be easily derived from our Theorem 2.2.

Corollary 2.3. On a reconfigurable mesh of size $M \times N$ the prefix sums of a binary sequence $b_1, b_2, \ldots, b_N$ with at most M 1's can be computed in $O(1)$ time. $\square$

Corollary 2.4. The prefix sums of a binary sequence $b_1, b_2, \ldots, b_N$ can be computed in $O(1)$ time on a reconfigurable mesh of size $\sqrt{N} \times N$. $\square$

Corollary 2.5. The prefix sums of an integer sequence $x_1, x_2, \ldots, x_{\sqrt{N}}$ with $0 \leq x_i \leq \sqrt{N}$ can be computed in $O(1)$ time on a reconfigurable mesh of size $\sqrt{N} \times N$. $\square$

2.2. Prefix Sums of an Integer Sequence

In this section, we present a constant time algorithm to compute the prefix sums of a sequence of n integers in the range from 0 to $n-1$ on an $n \times n$ reconfigurable mesh. We then show that the range of integers to be processed by the prefix sum algorithm can be extended to n^c, where c is a positive integer. In this case, the complexity of our algorithms is $O(c)$ time on this mesh.

To begin, we show how to compute the prefix sums of $a_0, a_1, \ldots, a_{n-1}$ with $0 \leq a_j \leq n-1$ in $O(1)$ time on a reconfigurable mesh R of size $n \times n$. For this purpose, it is assumed that the sequence is stored in the first row of R, with a_j being stored by processor $P(0,j)$ for all j. At the end of the computation, every processor $P(0,j)$ will store $a_0 + \cdots + a_j$.

First, for every a_j, the processors in column j can compute the tuple $<x_j, y_j>$ such that $a_j = x_j \times \sqrt{n} + y_j$, with $0 \leq x_j \leq \sqrt{n} - 1$ and $0 \leq y_j \leq \sqrt{n} - 1$ in $O(1)$ time. Note that

$$x_j = \left\lfloor \frac{a_j}{\sqrt{n}} \right\rfloor \text{ and } y = a_j \bmod \sqrt{n}.$$

11

The computation proceeds as follows. Every processor $P(i,j)$ $(0 \le i \le n-1)$ in column j computes $i \times \sqrt{n}$ and compares $i \times \sqrt{n}$ with a_j. Notice that in each column there must exist a unique row number i for which the following two conditions hold:

- $i \times \sqrt{n} \le a_j$;
- $(i+1) \times \sqrt{n} > a_j$.

Now this unique row number i is the largest integer smaller than or equal to $\dfrac{a_j}{\sqrt{n}}$, that is, $\left\lfloor \dfrac{a_j}{\sqrt{n}} \right\rfloor$, and, therefore, $a_j \bmod \sqrt{n} = j - \left\lfloor \dfrac{a_j}{\sqrt{n}} \right\rfloor \times \sqrt{n}$.

Once the tuple $<x_j,y_j>$ has been obtained from a_j $(0 \le j \le n-1)$, computing the prefix sums of $a_0, a_1, \ldots, a_{n-1}$ amounts to computing the prefix sums of the sequences $x_0, x_1, \ldots, x_{n-1}$ and $y_0, y_1, \ldots, y_{n-1}$. We only demonstrate how to compute the prefix sums of $x_0, x_1, \ldots, x_{n-1}$, for the prefix sums of $y_0, y_1, \ldots, y_{n-1}$ can be computed in the same way.

To compute the prefix sums of $x_0, x_1, \ldots, x_{n-1}$, we partition the $n \times n$ mesh R into $\sqrt{n}$ submeshes $M_0, M_1, \ldots, M_{\sqrt{n}-1}$ of size $n \times \sqrt{n}$, with M_i involving the columns $i\sqrt{n}$ through $(i+1)\sqrt{n} -1$ of R, and compute the prefix sums of the elements stored in the first row of every M_i. Now Corollary 2.5 guarantees that this can be done in O(1) time in every submesh M_i. Let $c_0, c_1, \ldots, c_{\sqrt{n}-1}$ be a new sequence of integers with c_i $(0 \le i \le \sqrt{n}-1)$ standing for $x_{i\sqrt{n}} + x_{i\sqrt{n}+1} + \cdots + x_{(i+1)\sqrt{n}-1}$. Put differently, every c_i is the sum of the elements in the first row of M_i.

The purpose of the next data movement operation is to move all the c_i's to the first row of M_0, with $P(0,i)$ storing c_i for all i. We proceed as follows: every processor $P(0,(i+1)\sqrt{n}-1)$ holding c_i, broadcasts c_i to $P(i,(i+1)\sqrt{n}-1)$; in turn, $P(i,(i+1)\sqrt{n}-1)$ broadcasts c_i to processor $P(i,i)$; finally, $P(i,i)$ broadcasts c_i to $P(0,i)$.

It is important to note that every c_i is an integer in the range 0 to $n-1$. As

12

before, every processor $P(0,i)$ $(0 \le i \le \sqrt{n} - 1)$ computes the tuple $<u_i, v_i>$ such that $c_i = u_i \times \sqrt{n} + v_i$, with $0 \le u_i \le \sqrt{n} - 1$ and $0 \le v_i < \sqrt{n} - 1$; this can be done in $O(1)$ time on M_0. Consequently, computing the prefix sums of $c_0, c_1, \ldots, c_{\sqrt{n}-1}$ reduces to computing the prefix sums of $u_0, u_1, \ldots, u_{\sqrt{n}-1}$ and $v_0, v_1, \ldots, v_{\sqrt{n}-1}$. By Corollary 2.5, this operation, performed in two stages, takes $O(1)$ time altogether.

Now performing in reverse the data movement operation detailed above, we can arrange for $c_0 + c_1 + \cdots + c_i$ $(0 \le i \le \sqrt{n} - 1)$ to be moved to $P(0,(i+1)\sqrt{n} - 1)$. All that remains to be done is to add, for all i $(0 \le i \le \sqrt{n} - 2)$, the value just received by $P(0,(i+1)\sqrt{n} - 1)$ to all the processors in the first row of M_{i+1}. This, of course, is done in a straightforward way in $O(1)$ time.

To summarize our findings we state the following result.

Theorem 2.6. The prefix sums of a sequence of integers $a_0, a_1, \ldots, a_{n-1}$ with $0 \le a_j \le n-1$ can be computed in $O(1)$ time on a reconfigurable mesh of size $n \times n$. $\square$

To extend the result in Theorem 2.6 to integers larger than n, consider a sequence $a_0, a_1, \ldots, a_{n-1}$ with $0 \le a_j \le n^c$ for a positive integer c. We begin by representing every a_j in radix n form. This can be done by successive divisions in $O(c)$ time. We apply the above algorithm to every sequence consisting of the digits of the same rank in all a_j's. By Theorem 2.6, handling every such sequence takes $O(1)$ time. Since there are a total of $O(c)$ sequences, the whole processing including combining the results from all sequences takes $O(c)$ time, hence we have the following.

Theorem 2.7. The prefix sums of a sequence of integers $a_0, a_1, \ldots, a_{n-1}$ with $0 \le a_j \le n^c$, here c is a positive integer. can be computed in $O(c)$ time on a reconfigurable mesh of size $n \times n$. $\square$

2.3. Sum of all the Integers in a Matrix

The goal of this section is to show how the constant time integer prefix sums algorithm developed in the previous section can be used to devise fast algorithms for

13

computing the sum of all values in a square matrix. We first show the number of 1's in a binary matrix of size $n \times n$ can be computed in $O(\log\log n)$ time on a reconfigurable mesh of size $n \times n$. We then show the sum of all the entries in an $n \times n$ matrix all of whose elements are integers in the range 0 to $\log n$ can be also computed in $O(\log\log n)$ time.

We assume a reconfigurable mesh R of size $n \times n$ and a matrix $A[1..n, 1..n]$ with $P(i,j)$ storing $A[i,j]$ for all $1 \le i, j \le n$. It is, furthermore, assumed that every entry $A[i,j]$ is either 0 or 1. We view R as consisting of submeshes $R_1, R_2, \ldots, R_{\sqrt{n}}$ of size $\sqrt{n} \times n$ with R_i ($1 \le i \le \sqrt{n}$) involving rows $(i-1)\sqrt{n}+1$ through $i\sqrt{n}$ of R. In turn, every R_i ($1 \le i \le \sqrt{n}$) is further subdivided into submeshes $R_{i1}, R_{i2}, \ldots, R_{i\sqrt{n}}$ with R_{ij} ($1 \le j \le \sqrt{n}$) consisting of columns $(j-1)\sqrt{n}+1$ through $j\sqrt{n}$ of R_i.

Next, we compute the sum of all the entries stored by processors in R_{ij} recursively and let the result, denoted by a_{ij}, be stored by processor $P(i\sqrt{n}, j\sqrt{n})$. Notice that the sum of all numbers in the original matrix is given by $\sum_{1 \le i, j \le \sqrt{n}} a_{ij}$.

Finally, every processor $P(i\sqrt{n}, j\sqrt{n})$ broadcasts a_{ij} to $P((i-1)\sqrt{n}+j, j\sqrt{n})$; in turn, processor $P((i-1)\sqrt{n}+j, j\sqrt{n})$ broadcasts a_{ij} to $P((i-1)\sqrt{n}+j, 1)$. Note that as a result of the previous data movement operation, column 1 of every R_i contains in top-down order $a_{i1}, a_{i2}, \ldots, a_{i\sqrt{n}}$. Now applying Theorem 2.7, we compute the (prefix) sums of all the elements in column 1 of R in $O(1)$ time. Clearly, the value of the prefix sum stored by $P(n,1)$ is exactly the sum of all the entries in the matrix.

Let $T(n)$ denote the worst-case running time of our algorithm. By our previous discussion, $T(n)$ satisfies the recurrence:

$$T(n) = T(\sqrt{n}) + O(1).$$

with the boundary condition $T(2) = O(1)$. It is easy to confirm that the solution of this recurrence is $T(n) = O(\log\log n)$.

The following theorem captures our findings.

14

Theorem 2.8. Let A be an $n \times n$ matrix all of whose elements are either 0 or 1. The sum of all the elements of A can be computed in $O(\log\log n)$ time on a reconfigurable mesh of size $n \times n$. $\square$

Now consider the matrix $A[1..n, 1..n]$ all of whose entries are integers in the range from 0 to $\log n$ and let R be an $n \times n$ reconfigurable mesh with $P(i,j)$ storing $A[i,j]$ for all i, j. We try to compute the sum of all the entries in A on R by divide and conquer as before, but once the size of the submeshes is smaller than $\log n \times \log n$, the method in [30] will be used to compute the sums of all entries in the corresponding matrices, and it takes $O(\log\log n)$. The overall complexity is still $O(\log\log n)$, so we have the following result.

Theorem 2.9. Let A be an $n \times n$ matrix all of whose elements are integers in the range from 0 to $\log n$. The sum of all the elements of A can be computed in $O(\log\log n)$ time on a reconfigurable mesh of size $n \times n$. $\square$

2.4. Computing Low-Level Descriptors of a Digitized Image

In this section, we will use the results developed in the previous section to compute a number of low-level descriptors of a digitized image, including the perimeter, area, histogram, and median row.

The *area* of a gray-level image is defined to be the number of pixels whose gray-level intensity exceeds a certain threshold. For a binary image, the area corresponds to the number of 1 pixels in an image. The boundary of the image is the set of all pixels whose gray-level intensity exceeds the threshold and all of whose neighbors do not exceed the threshold. It is worth noting that the boundary of a gray-level image can be identified by checking a constant number of neighbors of each pixel. The *perimeter* of an image is the length of its boundary, i.e. the number of pixels on the boundary. Correspondingly, the area of a component is the number of pixels within that component, and the perimeter of a component is the number of pixels

15

on the boundary of that component.

Recently, Jenq and Sahni [14] presented an algorithm for computing the area and perimeter of components of an $n \times n$ image in $O(\log n)$ time on a reconfigurable mesh of size $n \times n$, if the components are labeled. We will show how to compute the perimeter and the area of an an $n \times n$ image in $O(\log\log n)$ time on a reconfigurable mesh of size $n \times n$.

In fact, computing the area and perimeter of a digitized image can be reduced to instances of computing the sum of all values of an $n \times n$ matrix discussed in the previous section. To see that this is the case, we let every processor in the mesh write a 1 or a 0 into a local variable depending on whether or not the gray level intensity of the pixel it holds is above or below the threshold. Therefore, we get a binary matrix of size $n \times n$, and the number of 1's in that matrix is exactly the area of the image. The problem of computing the perimeter is similar. Consequently, we have the following result.

Theorem 2.10. Both the area and the perimeter of a digitized image of size $n \times n$ can be computed in $O(\log\log n)$ time on a reconfigurable mesh of size $n \times n$. □

The *histogram* of an image quantized to k gray levels is a vector H, where $H(z_i)$ is the number of pixels with gray level z_i ($1 \le i \le k$). The histogram tells us how often each gray level occurs in an image. It indicates the overall brightness and contrast of an image. Further, the dynamic range of the gray levels that make up an image is readily apparent. As such, histograms are valuable tools for image processing work both qualitatively and quantitatively [25]. It is not surprising that algorithms for computing the histogram have been designed and implemented in many parallel architectures [2,12,24,26,27,49].

On a reconfigurable mesh of size $n \times n$, Jenq and Sahni [16] presented an algorithm for computing the histogram of an $n \times n$ image quantized to k gray levels in $O(\sqrt{k} \log(n/\sqrt{k}))$ time. Their algorithm assumes that each processor has $O(\sqrt{k})$

16

memory. In practice, each processor can only have O(1) memory and the number of gray levels is a constant. Under this more realistic assumption, their algorithm takes $O(\log n)$ time. This result can be improved if we reduce computing the histogram to computing the sum of all values of an $n \times n$ binary matrix. To compute the histogram of a digitized image, we proceed to compute the number of pixels of each gray level and so, by the result in the previous section, the overall complexity is $O(k \log\log n)$. Since for all practical purposes, the number k of gray-level intensities is a constant, we have the following result.

Theorem 2.11. The histogram of a digitized image of size $n \times n$ can be computed in $O(\log\log n)$ time on a reconfigurable mesh of size $n \times n$. $\square$

The *median row* of a binary image is defined to be the row with the property that about half the 1's are above it and about half are below it [46]. More specifically, for a binary matrix $A[1..n, 1..n]$, let s_i be the sum of all entries in row i of A, and let S stand for the sum of all the entries in the matrix. The *median row* m of A is defined by the condition

$$\sum_{i=1}^{m-1} s_i < \frac{S}{2} \text{ and } \sum_{i=1}^{m} s_i \geq \frac{S}{2}.$$

In [46], Prasanna-Kumar and Raghavendra gave an $O(n^{1/3})$ algorithm to compute the median row on $n \times n$ meshes with multiple broadcasting. To the best of our knowledge, there is no algorithm reported in the literature to compute the median row on a reconfigurable mesh. We will present such an algorithm next.

To compute the median row, we proceed along the same lines as for computing the sum of all the elements in a binary matrix in the previous section: we view the mesh R as consisting of submeshes $R_1, R_2, \ldots, R_{\sqrt{n}}$ of size $\sqrt{n} \times n$ with R_i ($1 \leq i \leq \sqrt{n}$) involving rows $(i-1)\sqrt{n}+1$ through $i\sqrt{n}$ of R. In addition, every R_i ($1 \leq i \leq \sqrt{n}$) is further subdivided into submeshes $R_{i1}, R_{i2}, \ldots, R_{i\sqrt{n}}$, with R_{ij} ($1 \leq j \leq \sqrt{n}$) consisting of columns $(j-1)\sqrt{n}+1$ through $j\sqrt{n}$ of R_i.

17

We compute, recursively, the sum of all the entries stored by processors in R_{ij} and move all these partial results to the first column of R. Specifically, if we let a_{ij} stand for the sum of all the entries in R_{ij}, then as a result of the above data movement operation, column 1 of every R_i contains in top-down order $a_{i1}, a_{i2}, \ldots, a_{i\sqrt{n}}$.

Next, having computed the prefix sums of all the entries in the first column of R, we can determine the unique subscript i for which $P((i-1)\sqrt{n},1)$ stores a value smaller than $\frac{S}{2}$, while $P(i\sqrt{n},1)$ stores a value larger then or equal to $\frac{S}{2}$. Clearly, this tells us that the median row occurs among the rows from $(i-1)\sqrt{n}+1$ to $i\sqrt{n}$. This observation motivates us to search for the median row in the submesh R_i.

We now proceed as follows: R_i is viewed as consisting of submeshes M_1, $M_2, \ldots, M_{n^{1/4}}$ of size $n^{1/4} \times n$ with M_j $(1 \le j \le n^{1/4})$ involving rows $(j-1)n^{1/4}+1$ through $jn^{1/4}$ of R_i (note that for simplicity the rows of R_i are numbered from 1 to $\sqrt{n}$). We further subdivide every M_j into square submeshes M_{jk} of size $n^{1/4} \times n^{1/4}$, with M_{jk} involving columns $(k-1)n^{1/4}+1$ through $kn^{1/4}$ of M_j.

The sum of all entries in every M_{jk} is determined: note that this takes O(1) time as these sums were computed as part of the recursive call that determined the sum in all square submeshes R_{ij} discussed above. The only remaining task is to determine which processor stores this intermediate result. This, however, is done in a straightforward way whose details are omitted.

We are now in a position to determine which M_j contains the median row. For this purpose, we move the sum of all entries in every M_{jk} to the first column of R in the obvious order, and compute the prefix sums of these values. This process continues until a median row is found. It is easy to show that there is a total of O(loglog n) iterations and each iteration takes O(1) time. To summarize, we have the following result.

18

Theorem 2.12. The median row of a binary image of size $n \times n$ can be computed in $O(\text{loglog } n)$ time on a reconfigurable mesh of size $n \times n$. □

19

CHAPTER 3

MID-LEVEL VISION TASKS

The reconfigurability of a bus system allows us to associate an entity with a bus (either closed or open) and have all relevant elements of the entity on that bus (or enclosed by that bus if the bus is closed). Once such a bus is established, many properties of these elements can be determined by performing computations on the bus. In this chapter, we first present a new computational paradigm, i.e., identifying entities such as lines and regions with buses and performing computations on these buses to obtain properties of the entities. Computations on these buses include finding the maximum on an open bus (that leads to electing a leader of a closed bus), ranking an arbitrary open bus, and computing the prefix maxima (sums) on a bus. We then use this paradigm to develop the algorithms to solve the problem of Hough transform and component labeling,

3.1. Basic Computations on Buses

On reconfigurable meshes, buses are created dynamically, under program control, to fulfill computational needs. Every bus will have a *positive* direction and a *negative* direction. We assume that every processor that belongs to a bus can determine the positive (resp. negative) direction on the corresponding bus by checking local conditions only. Corresponding to each direction each bus will have a first and a last processor (for a closed bus, we can elect a first processor, i.e., a leader).

A bus may be thought of as a doubly-linked list of processors, with every processor on the bus being aware of its immediate neighbors, if any, in the positive and negative direction. When restricted to a given bus, the processors will be assumed to

20

have two ports: one is the positive port, the other the negative port. The *positive* (resp. *negative*) *rank* of a processor on a bus is taken to be one larger than the number of processors preceding the given processor when the bus is traversed in the positive (resp. negative) direction from the first processor. A bus is said to be *ranked* when every processor on the bus knows its positive and negative rank. The *length* of a bus coincides with the highest rank of a processor on that bus.

Quite often, the dynamic setting of connections within a reconfigurable mesh will result in establishing a number of buses, some of them being closed. Such is the case in some image processing applications where buses are "wrapped" around regions of interest in the image at hand. Subsequent computation often calls for selecting an ID for every region under consideration. This can be done by labeling every region with the identity of one of its pixels (typically, the pixel with the largest row and least column number within the region).

Put differently, it is normally necessary to elect a *leader* on the bus that was created around the region and broadcast the identity of this leader to all the pixels concerned. Therefore, we shall formulate the following general problem.

Problem 3.1. *Computing the maximum on an unranked bus.* Consider an unranked bus B of N processors, with every processor holding an item from a totally ordered universe. Identify the processor that holds the largest of these items.

Let P be an arbitrary processor on the bus, and $v(P)$ be the item stored by the processor P*. The algorithm begins by having all processors disconnect their positive and negative ports. We equate *active* processors with those on the bus which have set no connections. Hence, initially, all processors are active. During the course of the algorithm certain processors will become inactive, as we are about to explain. For a processor P on the bus, its (current) active neighbors are the closest active processors

---

* For simplicity we assume that all items on the bus are distinct.

21

in both positive and negative directions, provided they exist. An active processor P is said to be a *local maximum* if with P' and P'' standing for the active neighbors of P,

$$v(P) > v(P') \text{ and } v(P) > v(P'').$$

The algorithm comprises at most $\lfloor \log N \rfloor$ iterations. Specifically, for all i ($1 \le i \le \lfloor \log N \rfloor$), iteration i involves the following sequence of steps:

Step 1. By probing its active neighbors in each direction, every active processor determines whether or not it is a local maximum;

Step 2. Every processor that is not a local maximum on the bus connects its positive and negative ports, thus becoming inactive;

The algorithm terminates when there is only one active processor left on the bus. It is easy to see that the algorithm has at most $\lfloor \log N \rfloor$ iterations: this is because no two active neighbors on the bus can be local maxima in a given iteration. Consequently, every iteration eliminates at least half of the active processors. Therefore we have the following result.

Lemma 3.1. Given an unranked bus of length N with every processor holding an item from a totally ordered universe, the processor that holds the largest of these items can be identified in $O(\log N)$ time. $\square$

Since electing a leader is equivalent (in our formulation) to computing the maximum of the items on the bus we have proved the following result.

Corollary 3.2. A leader can be elected on an unranked closed bus of length N in $O(\log N)$ time. $\square$

Consider an arbitrary bus B. Applying the divide-and-conquer paradigm on B assumes that every processor on the bus knows its (positive) rank. Therefore, an important task is to determine the rank of every processor on B. Specifically, we state this as the following fundamental problem.

Problem 3.2. *Ranking an arbitrary bus.* Let B be an arbitrary open bus of length N.

compute the rank of every processor on B.

Obviously, in case the bus B is closed, our first task is to elect a leader on this bus and to transform B into a open one. Our algorithm for ranking a bus is an adaptation of an elegant algorithm of Cole and Vishkin [5]. They define the r-ruling set problem as follows. Let $G = (V, E)$ be a directed graph such that the in-degree and the out-degree of every vertex is exactly one. For obvious reasons such a graph is termed a *ring*. A subset U of V is an r-ruling set if (1) no two vertices in U are adjacent, and (2) for each vertex v in V there is a directed path from v to some vertex in U whose edge length is at most r. Cole and Vishkin proved the following surprising result.

Proposition 3.3 ([5]) A 2-ruling set of a ring with N vertices can be obtained in $O(\log^* N)$ time using N processors in the EREW-PRAM. □

As noted by Cole and Vishkin the same algorithm applies to models of computations where only local communications between successive nodes in the ring are allowed. This is precisely the case of an unranked bus. To rank an arbitrary bus B of length N we let every processor store a 1. Now repeatedly find a 2-ruling set in B and eliminate the nodes not in the current ruling set after having added the value they contain to the next node of the ruling set. Once the rank of the node in the current ruling set is known, the rank of the node eliminated can be derived in $O(1)$ time. Clearly, this process terminates in $O(\log N)$ iterations. Consequently we have the following result.

Lemma 3.4. An arbitrary open bus of length N can be ranked in $O(\log N \log^* N)$ time using the processors on the bus only. □

Another fundmental problem is performing semi-group computations on a bus

Problem 3.3. *Computing the prefix maxima (sums) on a bus.* Consider an arbitrary open bus B (either ranked or unranked) of length n, with every processor holding an item. The problem of interest is to compute the prefix maxima (sums) on the bus.

23

It is easy to see that a ranked bus of length N is equivalent to a reconfigurable mesh of size $1 \times n$. By applying the prefix maxima (sums) algorithm in [30], it follows that the prefix maxima (sums) on a ranked bus of length N can be computed in $O(\log N)$ time. For an unranked bus of length N, we can first rank the bus, then apply the prefix maxima (sums) algorithm in [30]. Hence we have that the prefix maxima (sums) on an unranked bus of length N can be computed in $O(\log N \log^* N)$ time.

Although our general bus ranking algorithm takes $O(\log N \log^* N)$, in many practical (special) cases, bus ranking takes only $O(\log N)$ time or even $O(1)$ time. Therefore under such circumstances, computing the prefix maxima (sums) on an unranked bus of length N takes $O(\log N)$ time. Next we will show that, even on an arbitrary unranked bus of length N, the prefix maxima problem can be solved in $O(\log N)$ time.

To make the problem precise, we need to introduce some terminology. As before, let P be an arbitrary processor on the bus, and $v(P)$ be the item stored by the processor P. We let $\text{Neg}(P)$ stand for the set of processors preceding P in the positive direction of B, and we let $\text{Pos}(P)$ stand for the set of processors following P in the positive direction of B. Now the prefix maxima problem involves computing for every processor P on the bus the value

$$\max_{Q \in \text{Neg}(P) \cup P} v(Q).$$

Initially, every processor on the bus marks itself "active". During the course of the algorithm certain processors will become inactive. For a processor P on the bus, its (current) active neighbors are the closest active processors in $\text{Neg}(P)$ and $\text{Pos}(P)$, provided they exist. An active processor P is said to be a *local maximum* if with P' and P'' standing for the active neighbors of P,

$$v(P) > v(P') \text{ and } v(P) > v(P'').$$

The algorithm begins by having all processors on the bus disconnect their positive and negative ports. We equate active processors with those on the bus which have set no connections (thus, initially, all processors on the bus are active). The algorithm

24

comprises at most $\lfloor \log n \rfloor$ iterations. Specifically, for all i ($1 \le i \le \lfloor \log n \rfloor$), iteration i involves the following sequence of steps:

Step 1. Every active processor checks its two active neighbors to detect whether or not it is a local maximum;

Step 2. Every processor that is not a local maximum connects its positive and negative ports (and, thus, becomes "inactive");

Step 3. Every processor that is a local maximum in the current iteration broadcasts the value it holds in the positive direction on the bus;

Step 4. Every inactive processor updates the prefix maximum by taking the maximum of the value received and the current maximum it stores.

It is easy to see that this simple algorithm terminates in $\lfloor \log N \rfloor$ iterations: this is because no two active neighbors on the bus can be local maxima in a given iteration. Consequently, every iteration eliminates at least half of the active processors, and the claim follows.

To summarize our findings we state the following result.

Lemma 3.5. The prefix maxima problem on an arbitrary unranked bus of length N can be solved in $O(\log N)$ time. $\square$

3.2. The Hough Transform

One of the fundamental problems in computer vision and image processing is the detection of shape. An important subproblem involves detecting straight lines and curves in binary or gray-level images. The task of detecting lines is often accomplished by a computational method referred to as the Hough transform [10,13,20,55]. We assume an image of size $N \times N$, and for the purpose of computing the Hough transform, we assume that the image has been binarized by assigning to every possible edge pixel a 1 and to every pixel that cannot be an edge pixel a 0. An instructive discussion of this binarization process can be found in [53]. It is well known that a

25

straight line L in the plane can be represented by two parameters θ and ρ, where θ is the angle determined by the normal to L and the positive direction of the x axis, and ρ is the signed distance from the origin to the line L, with the points on the line satisfying $x\cos\theta+y\sin\theta=\rho$ (refer to Figure 3.1). For an image of size $N \times N$, the Hough transform involves building a matrix $H[1..n,-\sqrt{2}N..\sqrt{2}N]$ such that for every i, $H[i,\rho]$ contains the number of the edge pixels (x,y) for which $\left\lfloor x\cos\theta_i + y\sin\theta_i \right\rfloor=\rho$, here θ_1, $\theta_2, \ldots, \theta_n$ are the possible values of the θ-parameter.
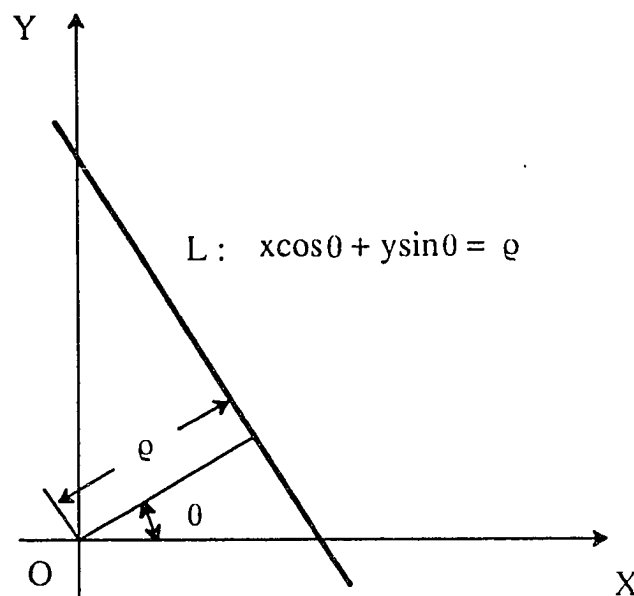
Figure 3.1: A line L in the normal form

Recently, a number of parallel algorithms for the Hough transform have been designed for different parallel architectures. Cypher *et al.* [7], and Guerra and Hambrush [11] show that the Hough transform can be computed in $O(N+n)$ time on a mesh connected computer of size $N \times N$ with each processor storing a pixel. Ranka and Sahni [50] presented $O(n+\log N)$ algorithms on an N^2 hypercube. Jolion and Rosenfeld [18] present an $O(\log n)$ Hough transform algorithm on pyramid. Jenq and Sahni [15] give an $O(n \log(N/n))$ algorithm on a reconfigurable mesh of size $N \times N$ with each

26

processor storing a pixel by token passing. We will present another $O(n \log(N/n))$ algorithm for computing the Hough transform by using a bus to identify a potential line and computing how many edge pixels are on the bus. Although our algorithm has the same complexity as Jenq and Sahni's, ours is simpler and more natural.

We begin by reviewing basics concerning the Hough transform. The interested reader is referred to [13,20,55] for more information. Let L be a straight line in the plane. It is easy to see that L can be represented by the following two parameters:

- θ_L, the angle determined by the normal to L and the positive direction of the x axis;

- ρ_L, the signed distance from the origin of the cartesian coordinate system to L.

When no confusion is possible we simplify the notation by writing θ and ρ, respectively. It is obvious that if we restrict θ to the range $[0..\pi]$, the ordered pair (θ,ρ) uniquely determines the line L. In addition, a point (x, y) of the plane belongs to L whenever

$$x \cos\theta + y \sin\theta = \rho. \tag{3.1}$$

Equation (3.1) is at the heart of the practical approach for detecting straight lines in an image.

The process begins by identifying the pixels in the image space that have a chance of belonging to an edge (i.e. straight line) in the image. Appropriately, these pixels are referred to as *edge pixels* (see [53] for details). It is convenient to associate with every edge pixel in the image the value of 1 and with all the remaining pixels the value of 0. Next, the θ-space is quantized: let $\theta_1, \theta_2, \ldots, \theta_n$ be the angles in the quantization. Naturally, the number n of different angles in the quantization is dictated by the desired accuracy of the output as well as by the resources available. It is customary, although not necessary, to chose the angles in the quantification such that for all i, $\theta_i = \frac{\pi i}{n}$. For every angle θ_i in the quantization, all the edge pixels (x, y) that have the same ρ-value* in (3.1) define a possible edge in the image space. For an

---

* Truncated to an integer; in this section we shall always truncate by applying the floor function.

27

image of size $N \times N$, the Hough transform involves building a matrix $H[1..n, -\sqrt{2}N..\sqrt{2}N]$ such that for every i, $H[i,\rho]$ contains the number of the edge pixels (x, y) for which $\lfloor x\cos\theta_i + y\sin\theta_i \rfloor = \rho$

We are now in a position to describe how the proposed Hough transform algorithm on reconfigurable meshes works. For this purpose, a reconfigurable mesh R of size $N \times N$ is assumed, with every processor $P(i,j)$ of R storing pixel $IM[i,j]$ whose x and y coordinates are i and j respectively. We proceed to detect edge pixels in the image: as pointed out in [53] this can be carried out in O(1) time on R by applying a Sobel gradient operator to the image.

We assume the θ-space quantized as described above. To make our exposition more transparent and easier to follow, we restrict ourselves to angles in the range $[0,\pi/4]$. As it turns out, the remainder of the range is handled similarly. We shall, therefore, consider a *generic* angle θ satisfying

$$0 \leq \theta \leq \pi/4. \tag{3.2}$$

It is easy to confirm that this choice of θ implies that

$$\frac{\sqrt{2}}{2} \leq \cos\theta \leq 1 \text{ and } 0 \leq \sin\theta \leq \frac{\sqrt{2}}{2,} \tag{3.3}$$

and that

$$\cos\theta \geq \sin\theta \text{ and } \sin\theta + \cos\theta \geq 1. \tag{3.4}$$

For every pixel $IM[i,j]$ (not necessarily an edge pixel) in the image space we write

$$\rho_{i,j} = \lfloor i\cos\theta + j\sin\theta \rfloor \tag{3.5}$$

The following results will be instrumental in understanding how our algorithm works and can be easily derived from observation (3.3)-(3.5) above using straightforward trigonometric manipulation.

Lemma 3.6. For all j $(0 \leq j \leq N-1)$ we have $\rho_{0,j} \leq \rho_{1,j} \leq \cdots \leq \rho_{N-1,j}$. Furthermore, no three consecutive elements in row j have the same value of ρ. $\square$

28

Lemma 3.7. For all i, j ($0 \leq i, j \leq N-1$), $0 \leq \rho_{i+1,j} - \rho_{i,j} \leq 1$. That is, the ρ-values of consecutive pixels in row j differ by no more than 1. $\square$

Lemma 3.8. For all values of i, j ($0 \leq i, j \leq N-1$), $\rho_{i,j} \neq \rho_{i+1,j+1}$. $\square$

Lemma 3.9. For all values i, j ($0 \leq i, j \leq N-1$), $\rho_{i,j-1} = \rho_{i,j+1}$ implies $\rho_{i,j-1} = \rho_{i,j} = \rho_{i,j+1}$. $\square$

For every i ($0 \leq i \leq N-1$), and for every value of ρ, let I_i^ρ stand for the set of pixels $IM[i,j]$ in column i satisfying

$$\lfloor i\cos\theta + j\sin\theta \rfloor = \rho. \tag{3.6}$$

In this notation, Lemma 3.9 tells us that I_i^ρ is a vertical *interval* in column i of the image space. Denote I_i^ρ as $[b_i^\rho, t_i^\rho]$, with t_i^ρ and b_i^ρ standing for the top and bottom row numbers of pixels in I_i^ρ, respectively. Now Lemmas 3.7 and 3.8 combined indicate that

$$b_{i-1}^\rho - t_i^\rho \leq 1. \tag{3.7}$$

Refer to Figure 3.2.a for an illustration. Let R_ρ be the set of all the pixels in the image that satisfy (3.6). Note that R_ρ is the collection of vertical intervals $\{I_i^\rho \mid 0 \leq i \leq N-1\}$.

Next, compute the number of pixels in R_ρ which are edge pixels. We begin by constructing a bus to connect pixels of R_ρ and then count the number of edge pixels on that bus. Figure 3.2.b illustrates informally how the buses are formed for the data of Figure 3.2.a, where pixels marked by Δ or $*$ are at either end of a bus for a given value of ρ. The general principle for constructing the bus follows. For each ρ, the pixels in R_ρ are ordered by increasing x coordinate; when the pixels have the same x value, they are ordered by increasing y coordinate; The bus then connects processors together according to this order.

More specifically, link the processors corresponding to each I_i^ρ vertically. Then we consider two cases for linking processors in I_i^ρ to processors in I_{i-1}^ρ. First, if $t_i^\rho =$

29

b_{i-1}^{ρ}, processor $P(i-1,b_{i-1}^{\rho})$ can be directly linked to processor $P(i,t_i^{\rho})$. Second, if $t_i^{\rho}+1 = b_{i-1}^{\rho}$, processor $P(i-1,b_{i-1}^{\rho})$ is linked to processor $P(i,t_i^{\rho})$ through processor $P(i,t_i^{\rho}+1)$. Notice in the second case, $P(i,t_i^{\rho}+1)$ corresponds to a pixel in $R_{\rho+1}$. To avoid two connections in one processor, the computation proceeds in two steps. One step is for even values of ρ, the other for odd values of ρ. Notice for each ρ one end of the bus must in the bottom row or the rightmost column. The even values of ρ can be distinguished from the odd values of ρ in O(1) time as follows: consider the sequence beginning with the bottom row (from left to right) followed by the rightmost column (from bottom to top). Assign a 1 to the processors marked as the end of a bus with Δ and 0 to others; perform prefix sums on that binary sequence. Theorem 2.6 guarantees that this prefix sums can be computed in O(1) time on an $N \times N$ reconfigurable mesh. Figure 3.2.c and Figure 3.2.d show how the buses are formed for even values of ρ and odd values of ρ respectly using the data in Figure 3.2.a. It is obvious that the buses can be constructed in O(1) time since every processor need only check the ρ-value of its direct neighbors. Note, further, that by construction every bus has length O(N).

Finally, the technique for computing sums in one row described in [30] can be applied if the rank of the processors on a bus can be determined. To determine the rank, the coordinates of the processors marked Δ are put on the bus. The rank of each processor is the sum of the absolute difference of its coordinates with those broadcast. We now assign a 1 to processors storing edge pixels in R_{ρ}, and 0 to other processors. The technique of [30] can now be applied to compute the sum in O(log k) time, where k is the length of the bus connecting R_{ρ}.

Since the sums for all ρ at a given θ are computed in two steps, and each step takes O(log$N$), the row of values in H corresponding to a θ can be computed in O(log$N$) time. Thus, the complete set of values for H can be computed in O($n \times$log$N$) time. To summarize our findings we state the following result.
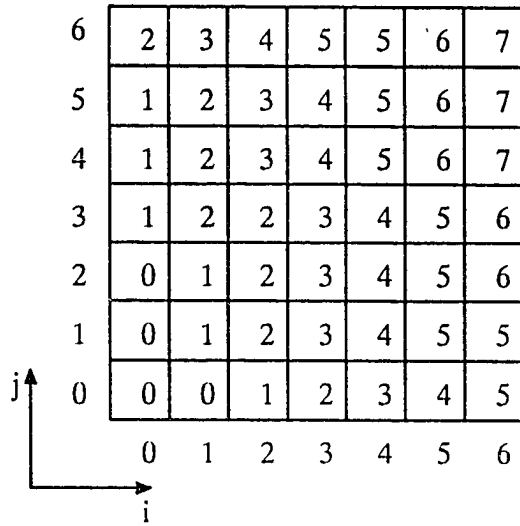
30

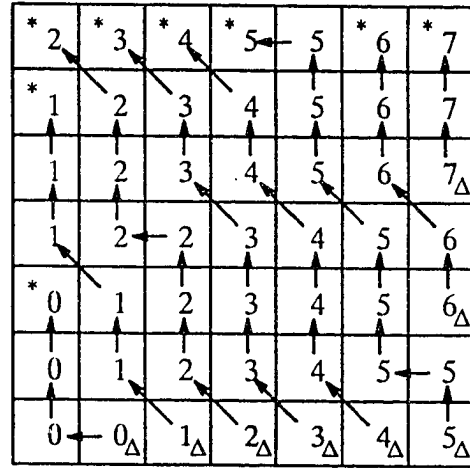Figure 3.2.a: A 7×7 grid with ρ-value computed for θ=π/8
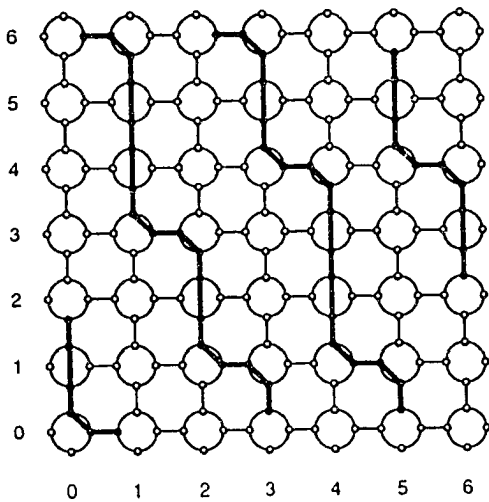
Figure 3.2.b: Buses formed for each ρ
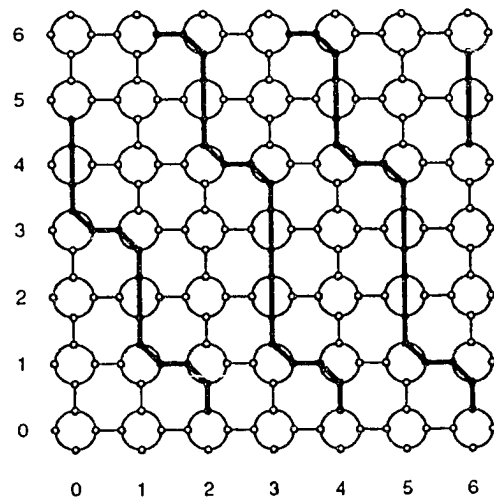
Figure 3.2.c: Buses formed for even ρ's

Figure 3.2.d: Buses formed for odd ρ's

31

Theorem 3.10. The Hough transform matrix H of an $N \times N$ digitized image stored one pixel per processor by a reconfigurable mesh of size $N \times N$ can be computed in $O(n \times \log N)$ time, where n is the number of angles in the quantization of the θ-space. $\square$

In fact, we can do better. To see this, we consider the $N \times N$ reconfigurable mesh M as consisting of n submeshes $S_1, S_2, \ldots, S_n$ of size $(N/n) \times N$ with S_i ($1 \le i \le n$) consisting of the rows $(i-1)N/n$ through $iN/n-1$ of R, here we assume that n is smaller than N. The idea of the (improved) algorithm is to pipeline the n angles θ_1, $\theta_2, \ldots, \theta_n$ as follows. Initially, the mesh S_1 processes angle θ_1 as described above for $\log(N/n)$ time units. In the next stage, the submesh S_2 receives the partial result computed by S_1 and continues processing θ_1; at the same time S_1 starts processing the angle θ_2, and so on.

It is easy to confirm that the whole computation is finished in $(2n-1) \times \log (N/n)$ time units. Consequently, we have the following important result.

Theorem 3.11. The Hough transform of an $N \times N$ digitized image stored one pixel per processor by a reconfigurable mesh of size $N \times N$ can be computed in $O(n \times \log (N/n))$ time, where n is the number of angles in the quantization of the θ-space. $\square$

3.3. Component Labeling

Another fundamental mid-level vision task involves detecting, counting, and labeling the various connected components present in an image. The task at hand is commonly referred to as *component labeling*. Given a binary image of size $\sqrt{n} \times \sqrt{n}$, two 1-valued pixel are adjacent if they share a horizontal or vertical edge. Two pixels are connected if there exists a path of adjacent 1-valued pixels from one to the other. The component labeling problem is to label each 1-valued pixel such that any two 1-valued pixels receive the same label if and only if they are connected. A set of pixels that

32

receive the same label is a connected component of the image.

The component labeling problem has been extensively studied in the parallel settings. Nassimi and Sahni [37] showed an $O(\sqrt{n})$ algorithm on a mesh connected computer of size $\sqrt{n} \times \sqrt{n}$. Cypher *et al.* [6] gave an $O(\log^2 n)$ algorithm on a hypercube or shuffle-exchange architecture of size n. In [8], the same authors presented an $O(\log N)$ algorithm on the EREW (Exclusive Read, Exclusive write) PRAM model. Prasanna-Kumar and Reisis [76] gave an $O(n^{1/4})$ algorithm on meshes with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. Recently, Miller *et al.* [32] proposed an $O(\log^2 n)$ time algorithm on a reconfigurable mesh of size $\sqrt{n} \times \sqrt{n}$. Their algorithm is based on divide-and-conquer and on an $O(\log n)$ time algorithm to label the connected components of a graph [57]. We will present an $O(\log n)$ time component labeling algorithm using buses to identify connected components. For ease of understanding, we first give a preview of the algorithm, then give the details of the implementation.

3.3.1. The Algorithm — a Preview

The task of *labeling* the connected components of an image can be broken down into the following two subtasks:

- assigning a unique "name" to every component;

- informing every 1-pixel in the image of the name of the component it belongs to.

The bulk of the component labeling algorithms in the literature are divide-and-conquer based, or else proceed along lines originating in Levialdi [19]. Levialdi's algorithm is, basically, a two-stage algorithm. In the first stage, every component in the image is shrunk to a single pixel and then eliminated. The second stage is the expansion stage: here, a unique label is generated for each component and then every pixel in the component is informed about this label.

The divide-and-conquer approach [3,9,18] involves solving the component labeling problem in larger and larger sub-images, until the whole problem is solved.

33

Our approach is different: we exploit the dynamic reconfigurability of the bus system of the mesh at our disposal to "wrap" a bus around every component of the image. In case the component contains holes, buses will be created around the holes as well. As it turns out, once these buses have been created, the subtasks of uniquely identifying every component and of informing every 1-pixel of the name of the component it belongs to can be performed efficiently.

To avoid the need of handling tedious special cases, we subject the input image I to the following simple morphological transformation, resulting in a new image I' of size $2\sqrt{n} \times 2\sqrt{n}$: for all i, j ($1 \leq i, j \leq \sqrt{n}$), we map pixel($i,j$) of I to the 2×2 group of pixels consisting of pixel($2i-1, 2j-1$), pixel($2i-1, 2j$), pixel($2i, 2j-1$), and pixel($2i, 2j$) of I'.

It is easy to see that the mapping from I to I' that we just defined preserves a number of important topological properties, including connectivity. In addition, I' has a property that we refer to as the *odd-even* property, namely,

$$\text{for every } i \ (1 \leq i \leq \sqrt{n}) \text{ rows } 2i-1 \text{ and } 2i \text{ are identical.} \qquad (3.8)$$

Similarly,

$$\text{for every } j \ (1 \leq j \leq \sqrt{n}) \text{ columns } 2j-1 \text{ and } 2j \text{ are identical.} \qquad (3.9)$$

We further assume that the new image I' has been pretiled in a reconfigurable mesh of size $2\sqrt{n} \times 2\sqrt{n}$ such that for every i, j, processor $P(i,j)$ stores pixel(i,j) of I'.

Before we give a high-level description of the algorithm, we need to define some new terms. A 1-pixel pixel(i,j) is termed a *boundary* pixel if the 3×3 window centered at pixel(i,j) contains both 0 and 1-pixels. By convention, the portion of the 3×3 window that exceeds the borders of the image is assumed to contain 0-pixels. A processor storing a boundary pixel will be termed a *boundary processor*. A *hole* in a component is a maximal connected region of 0-pixels within the component. In the presence of holes, it makes sense to distinguish between *external* boundaries and *internal*

34

boundaries, with one internal boundary surrounding every hole in a connected component of 1-pixels.

On every boundary, external or internal, we elect a *leader*: this is a boundary processor $P(i,j)$ for which j is minimized and i is maximized, among all the processors on the same boundary as $P(i,j)$.

Our component labeling algorithm involves the following sequence of computational steps. We only give a brief overview of these steps here. A detailed description is presented in the next subsection.

Step 1. The purpose of this step is to detect boundary pixels in the image and to construct buses that will connect the corresponding boundary processors of the mesh; these buses will be referred to as *boundary* buses;

Step 2. The goal of this step is to elect a leader on each boundary bus constructed in Step 1; furthermore, every leader identifies its bus as external or internal;

Step 3. The task specific to this step is to have every leader of an internal bus determine the identity of the corresponding external bus;

Step 4. Finally, using the information computed in Step 3, every 1-pixel in the image is informed about the identity of the component it belongs to.

3.3.2. The Algorithm — Detailed Description

We are now in a position to give a detailed description of the computational steps in our component labeling algorithm.

Step 1. The first goal of this step is to identify boundary pixels; once this is done. the corresponding boundary processors connect a certain pair of ports, thus establishing buses in the reconfigurable mesh. Specifically, every processor $P(i,j)$ holding a 1-pixel inspects the 3×3 window centered at pixel(i,j). If this window contains both 0 and 1-pixels, $P(i,j)$ identifies itself as a boundary processor.

35

Note that by the odd-even property specified in (3.8) and (3.9), every such 3×3 window must contain two identical rows and two identical columns. Furthermore, the identical rows (resp. columns) are adjacent in the window. It is now an easy task to confirm that there are exactly 24 distinct window configurations such that pixel(i,j) is a boundary pixel (for the reader's benefit, these configuration are featured in Figure 3.3; the * stands for the "don't care" pixel).

Group 1

*	0	0
0	1	1
0	1	1

1.1

0	0	*
1	1	0
1	1	0

1.2

1	1	0
1	1	0
0	0	*

1.3

0	1	1
0	1	1
*	0	0

1.4

Group 2

*	0	0
1	1	1
1	1	1

2.1

0	0	*
1	1	1
1	1	1

2.2

1	1	1
1	1	1
0	0	*

2.3

1	1	1
1	1	1
*	0	0

2.4

1	1	*
1	1	0
1	1	0

2.5

1	1	0
1	1	0
1	1	*

2.6

*	1	1
0	1	1
0	1	1

2.7

0	1	1
0	1	1
*	1	1

2.8

Group 3

0	1	1
1	1	1
1	1	1

3.1

1	1	0
1	1	1
1	1	1

3.2

1	1	1
1	1	1
1	1	0

3.3

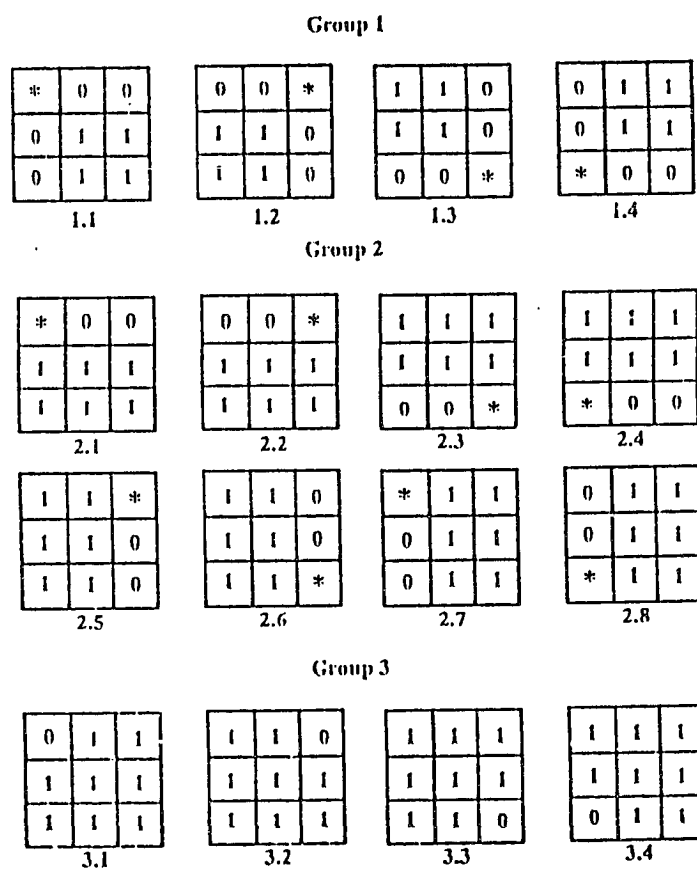| 1	1	1
1	1	1
0	1	1

3.4

Figure 3.3: The various 3×3 windows centered at a boundary pixel

Now every boundary processor connects two of its ports as follows:

- Every boundary processor $P(i,j)$ in Group 1 connects:

36

SE if $P(i,j)$ is in subgroup (1.1);

SW if $P(i,j)$ is in subgroup (1.2);

NW if $P(i,j)$ is in subgroup (1.3);

NE if $P(i,j)$ is in subgroup (1.4).

- Every boundary processor $P(i,j)$ in Group 2 connects:

  EW if $P(i,j)$ is in subgroups (2.1)-(2.4);

  NS if $P(i,j)$ is in subgroups (2.5)-(2.8).

- Every boundary processor $P(i,j)$ in Group 3 connects:

  NW if $P(i,j)$ is in subgroup (3.1);

  NE if $P(i,j)$ is in subgroup (3.2);

  SE if $P(i,j)$ is in subgroup (3.3);

  SW if $P(i,j)$ is in subgroup (3.4).

What results is a collection of disjoint buses, referred to as boundary buses, satisfying a number of properties that we present next. The reader can find an illustration in Figure 3.4 about how the boundary buses are created: shaded processors are assumed to contain 1-pixels.

Lemma 3.12. Every boundary processor belongs to exactly one boundary bus.

Proof. First, every 1-pixel can determine whether or not it is a boundary pixel by examining the 3×3 window discussed above.

Consequently, every boundary processor will belong to at least one boundary bus. However, since every boundary processor only connects one pair of ports, it belongs to at most one boundary bus. The conclusion follows. ☐

Lemma 3.13. Two boundary buses either coincide or else are disjoint.

Proof. Suppose not; now, some boundary processor must belongs to at least two distinct buses, contradicting Lemma 1.2. ☐

Lemma 3.14. Every boundary bus is a closed curve.

37
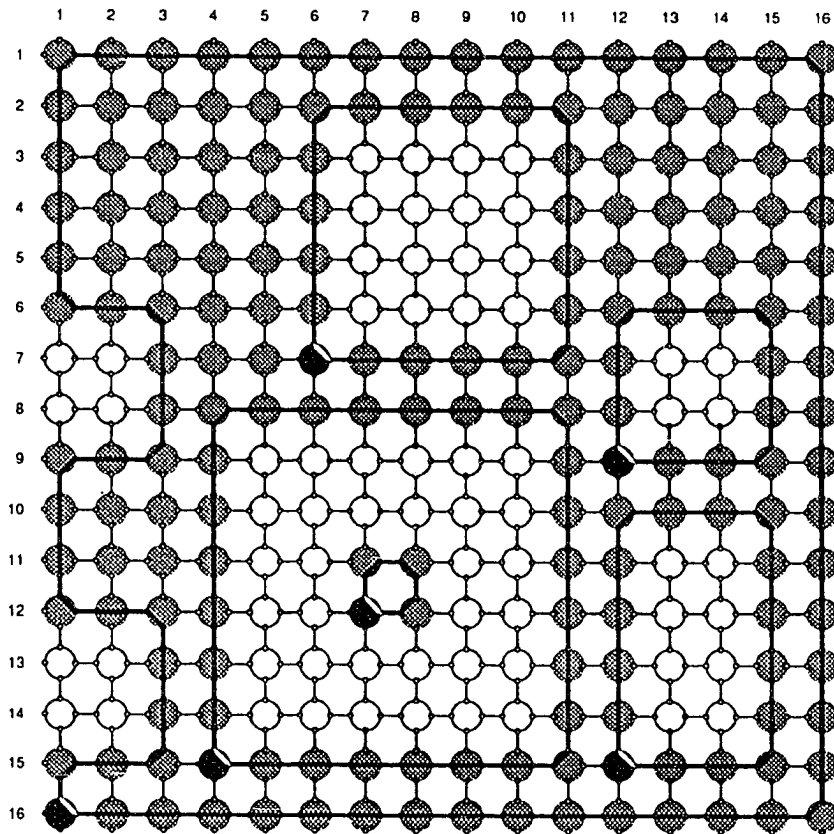
Figure 3.4: The resulting buses after Step 1

Proof. Follows immediately from the way boundary processors set their local connections. □

In summary, Step 1 runs in $O(1)$ time since it amounts to setting local connections by checking 3×3 windows only.

Step 2. As mentioned in the previous section, the goal of this step is to determine the leader of every boundary bus created in Step 1. Let B be an arbitrary bus created in Step 1. Recall that the leader of the bus B is the boundary processor $P(i, j)$ for which the value of j is minimized and i is maximized, among all boundary processors on the

38

bus B. By Lemma 3.1, the leader can be found in $O(\log n)$ time.

To complete Step 2, the leader of every bus identifies its bus as external or internal. This task is easy as soon as we note that a bus is external if the 3×3 window centered at its leader has a 1-pixel in its north-east corner; similarly, a bus is internal if the 3×3 window centered at its leader has a 0-pixel in its north-east corner.

Note: For the reader's benefit, the leaders of the buses in Figure 3.4 are featured in black.

Step 3. The purpose of this step is to associate every internal bus with the corresponding external bus. To carry out this task, we extend slightly the definition of a leader of an internal bus. Let B stand for an arbitrary internal bus created in Step 1, and let $P(i,j)$ be the leader of B. Note that the odd-even property specified in (3.8) and (3.9) guarantees that i is odd. For the purpose of this step, we shall refer to processor $P(i,j)$ as the *odd* leader of B and to processor $P(i-1,j)$ as the *even* leader of B.

To begin, every non-boundary processor in the reconfigurable mesh sets its local connection to EW. Both odd and even leaders of an internal bus broadcast a signal westbound on the horizontal buses created above. Every boundary processor reads its port E; boundary processors that receive a signal from their E port will be referred to as *special*. It is important to note that the data movement described above allows us to distinguish between *even* and *odd* special processors. Clearly, if $P(i',j')$ is an *odd* special processor, then $P(i'-1,j')$ is an *even* special processor. Further, note that no special processor can be the leader of its own bus. Now leaders (even and odd) of internal buses as well as special processors (even and odd) proceed to reset their local connection as follows:

(3.10) every odd leader of an internal bus sets its local connection to EW;

(3.11) every even leader of an internal bus sets its local connection to NW;

(3.12) every special processor in an odd row sets its connection to EW or to NE

depending on whether or not its original setting was SW or NS;

39

(3.13) every special processor in an even row sets its connection to EW or to SE

depending on whether or not its original setting was NW or NS.

What follows as a result of (3.10) - (3.13) above is a bus system that originates at the leader of every external bus, traverses exactly once every boundary processor of the corresponding connected component and returns to the leader (see Figure 3.5 for an illustration).
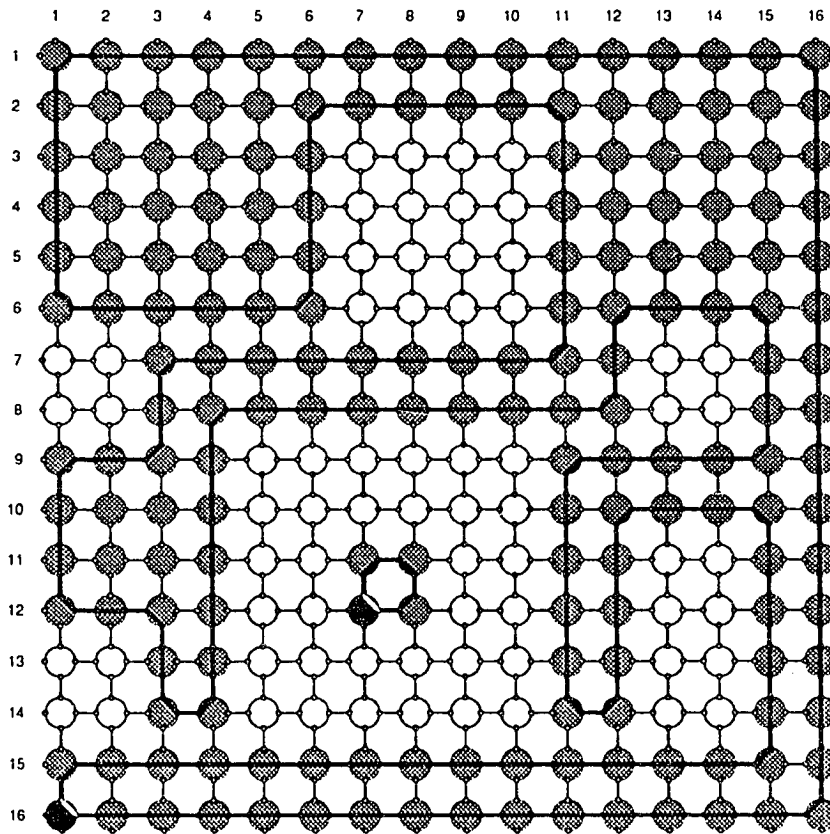

Figure 3.5: The resulting buses after Step 3

To conclude Step 3, every leader of an external bus broadcasts its identity on the bus created above; an obvious result of this operation is that every boundary processor associated with a connected component in the image becomes aware of the identity of

40

the leader of the external bus of that component. It is obvious that Step 3 can be carried out in time O(1).

Step 4. The goal of this last step is to inform every non-boundary processor holding a 1-pixel of the identity of the leader of the corresponding external bus. Note that since the leader of every external bus uniquely determines the corresponding component, once this step is completed, the task of labeling the connected components in the image is also complete.

To achieve the goal of this step, we let every non-boundary processor storing a 1-pixel set its local connection to EW; boundary processors as well as processors storing 0-pixels set no connection. Next, every boundary processor broadcasts westbound the identity of the leader of the external bus of the component. This, obviously, achieves the desired result, and Step 4 is complete.

It is important to note that Step 4 can be carried out in time O(1). To summarize our findings we state the following result.

Theorem 3.15. The component labeling problem of a binary image of size $\sqrt{n} \times \sqrt{n}$ can be solved in $O(\log n)$ time on a reconfigurable mesh of size $2\sqrt{n} \times 2\sqrt{n}$ □

CHAPTER 4

CONVEX HULL OF A SET OF PLANAR POINTS

The convex hull of a set of points in the plane is defined as the smallest convex set that contains the original set [48]. The problem of computing the convex hull of points in the plane is central in a variety of problems in pattern recognition and image processing [32-35,47,48]. It comes as no surprise, therefore, that this problem was included among the tasks of the first DARPA image understanding benchmark for parallel computers [54]. In addition, the convex hull of an $n \times n$ digitized image can be reduced to the convex hull of planar points of size $O(n)$.

Quite recently, Miller and Stout [34] proposed efficient parallel convex hull algorithms on many different parallel architectures. In particular, they showed that the convex hull of n planar points can be computed in $O(\log^2 n)$ time on a reconfigurable mesh of size $\sqrt{n} \times \sqrt{n}$ if the points are sorted by x-coordinate. It is worth noting that sorting n points takes $\Omega(\sqrt{n})$ time on a reconfigurable mesh of size $\sqrt{n} \times \sqrt{n}$. Thus, the reconfigurable bus system does not seem helpful when dealing with the dense case, i.e., computing the convex hull of n planar points on a $\sqrt{n} \times \sqrt{n}$ reconfigurable mesh. We will show a constant time algorithm to deal with the sparse case, i.e., computing the convex hull of n planar points stored in one row of an $n \times n$ reconfigurable mesh.

In preparation for this result We first present two important data movement techniques. First, we argue that the prefix maxima of n real numbers can be computed in $O(\frac{\log n}{\log m})$ time on a reconfigurable mesh of size $m \times n$. Second, we show that n real numbers stored in one row of a reconfigurable mesh of size $n \times n$ can be sorted in $O(1)$ time. We then use these two techniques to construct a sub-optimal convex hull algo-

42

rithm. This sub-optimal algorithm will be later refined into an optimal one. Our result shows that the convex hull of n planar points can be computed in $O(1)$ time on a reconfigurable mesh of size $n \times n$.

4.1. Prefix Maxima of a Sequence of Real Numbers

Given a sequence of n real numbers $a_0, a_1, \ldots, a_{n-1}$, the problem is to compute $z_j = \max\{a_0, a_1, \ldots, a_j\}$ for all j $(0 \leq j \leq n-1)$. To begin, we demonstrate an $O(1)$ time algorithm for computing the prefix maxima of n reals on an $n \times n$ reconfigurable mesh. We then use divide-and-conquer to solve the same problem in $O(\frac{\log n}{\log m})$ time on a $m \times n$ reconfigurable mesh with $2 \leq m \leq n$. In both algorithms, we assume that the input a_j $(0 \leq j \leq n-1)$ is stored in the processor $P(0,j)$, and the output z_j $(0 \leq j \leq n-1)$ will also be stored in the processor $P(0,j)$.

Our first algorithm can compute the prefix maxima of n reals in $O(1)$ time on an $n \times n$ reconfigurable mesh. It consists of three stages. In the first stage, we let every processor $P(i,j)$ with $0 \leq i < j \leq n-1$ know a_i and a_j. For this purpose, we establish a vertical bus in column j $(1 \leq j \leq n-1)$ from $P(0,j)$ to $P(j,j)$, and let $P(0,j)$ $(1 \leq j \leq n-1)$ broadcasts a_j southbound along the vertical subbus in column j; we then establish a horizontal subbus in row i $(0 \leq i \leq n-2)$ from $P(i,i)$ to $P(i,n-1)$, and let $P(i,i)$ $(0 \leq i \leq n-2)$ broadcast a_i eastbound along the horizontal subbus in row i. In the second stage, we check whether a_j equals the maximum of $a_0, a_1, \ldots, a_j$ in column j $(1 \leq j \leq n-1)$. To do so, we compare a_j with $a_0, a_1, \ldots, a_{j-1}$ in column j, and record a 1 if a_j is smaller than a_i $(0 \leq i \leq j-1)$ and a 0 otherwise. What we get in column j is a binary sequence of length $j-1$, and a_j equals the maximum of $a_0, a_1, \ldots, a_j$ if and only if the resulting binary sequence is all 0. It is easy to see that, if a_j equals the maximum of $a_0, a_1, \ldots, a_j$, the j-th prefix maximum is exactly a_j. Hence, we mark these a_j's. On the other hand, if a_j does not equal the maximum of $a_0, a_1, \ldots, a_j$, the j-th prefix maximum is equal to the nearest marked a's to its

43

left. Therefore all we need to do in the third stage is to let every non-marked processor obtain the value stored by the nearest marked processor to its left. This can be done by establishing the buses between two consecutive processors and letting every marked processor broadcast the value it stores westbound. It is obvious that all these three stages take constant time. To summarize, we state the following result.

Theorem 4.1. The prefix maxima of n real numbers stored in one row of a reconfigurable mesh of size $n \times n$ can be computed in $O(1)$ time. $\square$

Next, we show how to compute the prefix maxima of n reals $a_0, a_1, \ldots, a_{n-1}$ on an $m \times n$ mesh with $2 \leq m \leq n$. For simplicity, we assume that n is a power of m. Begin by partitioning the original mesh into submeshes of size $m \times m$, and compute the prefix maxima on each such submesh of size $m \times m$.

We further combine groups of m consecutive submeshes of size $m \times m$ into a submesh of size $m \times m^2$, then combine groups of m consecutive submeshes of size $m \times m^2$ into a submesh of size $m \times m^3$, and continue until the original mesh is obtained. Note that if the prefix maxima of m consecutive submeshes are known individually then the prefix maxima of their combination can be computed as follows. For convenience, let M_k^t represent the k-th submesh of size m^t, involving the columns from km^t to $(k+1)m^t - 1$; similarly, let $\max(M_k^t)$ represent the maximum of the reals originally stored in row 0 of M_k^t. Now we show how to combine $M_{sm}^t, M_{sm+1}^t, \ldots, M_{(s+1)m-1}^t$ into M_s^{t+1}. Consider the submesh M_{sm+k}^t of this group, the prefix maxima of this submesh can be updated by taking the maximum of the current prefix maximum of the submesh M_{sm+k}^t and the overall maximum of the preceding submeshes in the group, i.e. $\max\{\max(M_{sm}^t), \max(M_{sm+1}^t), \ldots, \max(M_{sm+k-1}^t)\}$. Note that there will be $O(\frac{\log n}{\log m})$ iterations and that each iteration takes $O(1)$ time.

Theorem 4.2. The prefix maxima of n reals stored in one row of a reconfigurable mesh of size $m \times n$ with $2 \leq m \leq n$ can be computed in $O(\frac{\log n}{\log m})$ time. $\square$

44

4.2. A VLSI-Optimal Sorting Algorithm

In this section, we first present several basic algorithms based on the binary prefix sum algorithm developed in Section 2.1, which include a simple sub-optimal sorting algorithm and a ranking algorithm. These algorithms then will be used to devise an O(1) time selection algorithm on a reconfigurable mesh. Further, it turns out that we can exploit our selection algorithm to solve a more general problem, the multi-selection problem, in O(1) time. Finally, the solution to the multi-selection problem yields a VLSI-optimal sorting algorithm: we show that sorting n items takes O(1) time on a reconfigurable mesh of size $n \times n$.

4.2.1. Preliminaries

In Section 2.1 we showed that, on a reconfigurable mesh of size $M \times N$ with $M \leq N$ the prefix sums of an N-element binary sequence can be computed in O(log N) time if $M = 1$, and in O($\frac{\log N}{\log M}$) time if $M > 1$. The binary prefix problem has far-reaching applications. We will proceed to illustrate some of these applications. Consider an arbitrary set $X = \{x_1, x_2, \ldots, x_N\}$ with M elements of X marked. The *compaction* problem asks for a permutation $x_1', x_2', \ldots, x_N'$ of X with all the marked elements occurring before the non-marked ones.

We now demonstrate how the prefix sum of a binary sequence can be used to solve the compaction problem efficiently on a reconfigurable mesh R of size $M \times N$. For this purpose, we assume that X is stored in the first row of R, with processor $P(1,i)$ storing x_i for all i.

Begin by partitioning the mesh R into two submeshes R_1 and R_2, with R_1 involving the first M columns, and R_2 consisting of the last $N-M$ columns of R. Every processor $P(1,i)$ in R_1 writes a 1 into a local variable b_i whenever x_i is *non-marked*; otherwise, $P(1,i)$ writes a 0 into b_i. Similarly, processor $P(1,i)$ in R_2 writes a 1 into a local variable c_i when x_i is marked and a 0 otherwise. Now compute the prefix sums

45

of the two binary sequences $b_1, b_2, \ldots, b_M$ and $c_1, c_2, \ldots, c_{N-M}$ on R_1 and R_2, respectively. By Corollary 2.3, this takes $O(1)$ time on both R_1 and R_2.

It is easy to verify that for a non-marked processor $P(1,i)$ in R_1 if $b_1 + b_2 + \cdots + b_i = p$, then $P(1,i)$ stores the p-th non-marked element among $x_1, x_2, \ldots, x_M$. Likewise, for a marked processor $P(1,M+j)$ in R_2 if $c_1 + c_2 + \cdots + c_j = q$, then $P(1,M+j)$ stores the q-th marked element among $x_{M+1}, x_{M+2}, \ldots, x_N$. Note also that since there are M marked elements altogether, $b_1 + b_2 + \cdots + b_M = c_1 + c_2 + \cdots + c_{N-M}$.

To complete the compaction operation, we only need *swap* the t-th non-marked element among $x_1, x_2, \ldots, x_M$ with the t-th marked element among $x_{M+1}, x_{M+2}, \ldots, x_N$. This can be achieved by a simple data movement operation as follows. Every processor $P(1,i)$ ($1 \le i \le M$) storing a non-marked element broadcasts x_i to $P(p,i)$ with $p = b_1 + b_2 + \cdots + b_i$; similarly, every processor $P(1,M+j)$ ($1 \le j \le N-M$) storing a marked element broadcasts x_{M+j} to $P(q,M+j)$ with $q = c_1 + c_2 + \cdots + c_j$. After this operation, every row of R contains two elements: one marked, the other non-marked. In two broadcasting steps these two elements are interchanged. All that remains to be done is to broadcast these elements back to the first row of R.

Further, it is not hard to figure out that the relative order of the marked elements can be preserved by rearranging all the marked elements on R_1. To summarize our findings we state the following result.

Theorem 4.3. On a reconfigurable mesh of size $M \times N$ ($M \le N$), the compaction problem on an N-element set with M elements marked can be solved in $O(1)$ time. $\square$

Consider a set $X = \{x_1, x_2, \ldots, x_N\}$ with elements from a totally ordered universe,* and let Y be an M-element subset of X. For every x_j ($1 \le j \le N$), we let

---

* To avoid tedious bookkeeping details we assume that all elements of X are distinct, and that two elements of X can be compared in $O(1)$ time.

46

rank(x_j) stand for the position of x_j in the sorted version of X. The *ranking* problem involves computing the ranks in X of all the elements in Y. As we are about to explain, the ability to compute the (prefix) sum of a binary sequence, together with the compaction algorithm described above, affords us an efficient ranking algorithm.

Let R be an $N \times N$ reconfigurable mesh with X stored in the first row of R, with processor $P(1,i)$ storing x_i for all i. To begin, compact the elements in Y into the first M positions of row 1, and let $x'_1, x'_2, \ldots, x'_N$ be the corresponding permutation of X. Note that by Theorem 4.3 this operation can be done in O(1) time on R.

Partition the mesh R into M submeshes $R_1, R_2, \ldots, R_M$, with R_i ($1 \le i \le M$) consisting of rows $1+(i-1)\dfrac{N}{M}$ through $i\dfrac{N}{M}$ of the original mesh*. Further, having established vertical buses in all columns of R, every processor $P(1,i)$ ($1 \le i \le N$) broadcasts x'_i to the whole column i.

Next, processor $P(1+(i-1)\dfrac{N}{M},i)$ ($1 \le i \le M$) broadcasts x'_i horizontally to the whole row $1+(i-1)\dfrac{N}{M}$. Every processor $P(1+(i-1)\dfrac{N}{M},j)$ ($1 \le j \le N$) sets a local variable b_{ij} to 1 whenever x'_i is larger than or equal to x'_j; otherwise the variable is set to 0. Clearly, the number of 1's in the binary sequence $b_{i1}, b_{i2}, \ldots, b_{iN}$ is precisely the rank of x'_i in X. In turn, finding the number of 1's in the sequence $b_{i1}, b_{i2}, \ldots, b_{iN}$ is an instance of the binary prefix problem. By Theorem 2.2, using the processors in the mesh R_i this takes O(log N) time if $M=N$, and O($\dfrac{\log N}{\log N - \log M}$) time if $M \ne N$.

Theorem 4.4. Let X be an arbitrary set of N elements chosen from a totally ordered universe, and let Y be a subset of X of size M ($M \le N$). On a reconfigurable mesh of size $N \times N$ the ranking problem for Y can be solved in O(log N) time if $M=N$ and in O($\dfrac{\log N}{\log N - \log M}$) time if $M \ne N$. $\square$

---

* For simplicity of exposition we assume that $\dfrac{N}{M}$ is an integer.

47

The solution to the ranking problem presented above can be easily extended to yield an efficient sorting algorithm. To see this, consider a reconfigurable mesh R of size $MN \times N$ with $M \leq N$, and let $x_1, x_2, \ldots, x_N$ be an arbitrary sequence of elements from a totally ordered universe. We assume that the sequence is stored in the first row of R, with $P(1,i)$ storing x_i for all i. The output is also stored in the first row of R in the usual way.

To begin, we create vertical buses in every column of R, and let processor $P(1,i)$ ($1 \leq i \leq N$) broadcast x_i to the whole column i. We now view the original mesh as consisting of submeshes $S_1, S_2, \ldots, S_M$ of size $N \times N$, with S_i ($1 \leq i \leq M$) involving rows $1+(i-1)N$ through iN of R. We further partition the input sequence into groups of consecutive $M' = \dfrac{N}{M}$ items, and let the i-th group be ranked in S_i. By Theorem 4.4, this takes O($\log N$) time if $M=1$, and in O($\dfrac{\log N}{\log N - \log M'}$)=O($\dfrac{\log N}{\log M}$) time if $M>1$. It is now a straightforward data movement operation to move the item of rank j ($1 \leq j \leq N$) to processor $P(1,j)$. We therefore have the following result.

Theorem 4.5. On a reconfigurable mesh of size $MN \times N$ with $M \leq N$ an N-element sequence chosen from a totally ordered universe can be sorted in O($\log N$) time if $M=1$, and in O($\dfrac{\log N}{\log M}$) time if $M>1$. $\square$

Note that Theorem 4.5 implies the following result that will be used again and again in the remaining part of this work.

Corollary 4.6. An N-element sequence chosen from a totally ordered universe can be sorted in O(1) time on a reconfigurable mesh of size $N^{3/2} \times N$. $\square$

The following result is a well-known gem of the computer science folklore. In addition, it turns out to be useful in our algorithms, so we state it for further use.

Folklore Theorem. Let B be a two-dimensional array. Begin by sorting the elements in each row of B in ascending order; next, sort the elements in each column of B in

48

ascending order. After this the rows of B are still sorted in ascending order. $\square$

4.2.2. The Selection Algorithm — a Preview

Given a collection $A = \{a_1, a_2, \ldots, a_n\}$ of n elements chosen from a totally ordered universe and an integer k ($1 \le k \le n$), our task is to return the k-th smallest element in A. As before, for all j ($1 \le j \le n$), we let $\text{rank}(a_j)$ stand for the position of a_j in the sorted version of A. In this terminology, we need to identify the element a_t of A with $\text{rank}(a_t) = k$.

Assume that the elements of A have been placed in a matrix $B[1..n^{1/3}, 1..n^{2/3}]$ in such a way that all rows and all columns are sorted in increasing order. The basic idea of our algorithm is to exploit the structure of B to reduce the problem of finding the k-th smallest element in A to that of computing the k'-th smallest element in a subset A' of A containing at most $n^{2/3}$ elements. On a reconfigurable mesh of size $n \times n$, both computing the matrix B with the properties specified above and selecting the k'-th smallest element in a set A' containing at most $n^{2/3}$ elements can be done in constant time, yielding, as we shall see, a constant time algorithm for selection.

Our arguments rely, in part, on the following simple result whose proof follows immediately from the properties of matrix B.

Lemma 4.7. Let $b_{i,j}$ be an arbitrary element of B. For every choice of subscripts i', j' with $1 \le i' < i$; $1 \le j' < j$, $\text{rank}(b_{i',j'}) < \text{rank}(b_{i,j})$; similarly, for every choice of subscripts i'', j'' with $i < i'' \le n^{1/3}$; $j < j'' \le n^{2/3}$, $\text{rank}(b_{i,j}) < \text{rank}(b_{i'',j''})$. $\square$

Subdivide B into square matrices $B_1, B_2, \ldots, B_{n^{1/3}}$ with B_i ($1 \le i \le n^{1/3}$) consisting of columns $1+(i-1)n^{1/3}$ through $in^{1/3}$. For all i ($1 \le i \le n^{1/3}$) let D_i stand for the (set of elements on the) main diagonal of B_i. Note that the elements of D_i are $b_{1,1+(i-1)n^{1/3}}$, $b_{2,2+(i-1)n^{1/3}}, \ldots, b_{n^{1/3}, in^{1/3}}$. To make the notation less involved, we let the elements of D_i be $d_1^i, d_2^i, \ldots, d_{n^{1/3}}^i$, with d_j^i standing for $b_{j,j+(i-1)n^{1/3}}$.

49

Lemma 4.7 implies the following easy result.

Lemma 4.8. For all i ($1 \leq i \leq n^{1/3}$), rank(d_1^i)<rank(d_2^i)< $\cdots$ < rank($d_{n^{1/3}}^i$). $\square$

Assume that none of the diagonal elements has rank k. Under this assumption, we define for every diagonal D_i ($1 \leq i \leq n^{1/3}$) the parameter s_i as follows:

$$s_i = \begin{cases} 0 & \text{if rank}(d_1^i) > k; \\ n^{1/3} & \text{if rank}(d_{n^{1/3}}^i) < k; \\ t & \text{if rank}(d_t^i) < k \text{ and rank}(d_{t+1}^i) > k. \end{cases} \qquad (4.1)$$

Notice that Lemma 4.8 implies that s_i is unique for every i ($1 \leq i \leq n^{1/3}$). We now introduce a result that tells us that some elements of B can be eliminated, as they are known to have ranks either *smaller* than k or larger than k and, consequently, they do not qualify for the k-th smallest element in A. The proof follows immediately from Lemma 4.7 and is, therefore, omitted.

Lemma 4.9. If $s_i = t$ ($0 \leq t \leq n^{1/3}$) then

$$\text{rank}(b_{p,q}) < k \quad \text{whenever } 1 \leq p \leq t, \ 1 \leq q \leq t + (i-1)n^{1/3}$$

and

$$\text{rank}(b_{p,q}) > k \quad \text{whenever } t+1 \leq p \leq n^{1/3}, \ t+1+(i-1)n^{1/3} \leq q \leq n^{2/3}. \quad \square$$

We call an element of B a *candidate* if it cannot be eliminated by virtue of Lemma 4.9. Let C stand for the set of candidates. We propose to show that C contains no more than $n^{2/3}$ elements. For this, we begin by defining the sets of elements eliminated by virtue of Lemma 4.9. Informally, let U_i and L_i ($1 \leq i \leq n^{1/3}$) stand for the rectangular sets eliminated in the upper left and lower right regions determined by $d_{s_i}^i$ and $d_{s_i+1}^i$, respectively (refer to Figure 4.1).
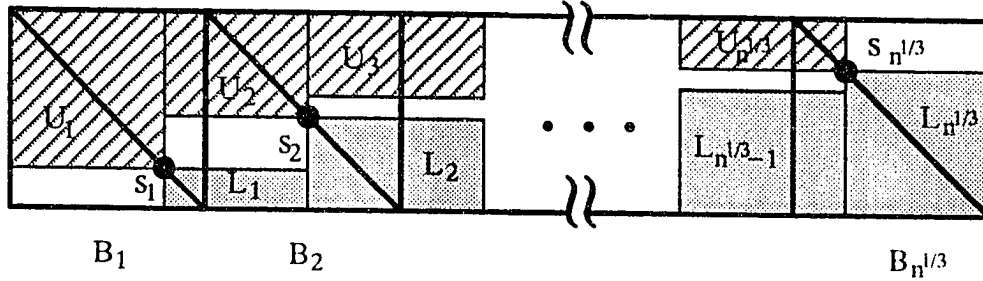
50

Figure 4.1: The layout of matrix B

More formally,

- $U_1 = \{b_{p,q} \mid 1 \le p \le s_1;\ 1 \le q \le s_1\}$;

- $L_{n^{1/3}} = \{b_{p,q} \mid s_{n^{1/3}} + 1 \le p \le n^{1/3};\ s_n^{1/3} + 1 + (n^{1/3} - 1)n^{1/3} \le q \le n^{2/3}\}$;

- for all i $(2 \le i \le n^{1/3})$,

$$U_i = \{b_{p,q} \mid 1 \le p \le s_i;\ s_{i-1} + 1 + (i-2)n^{1/3} \le q \le s_i + (i-1)n^{1/3}\};$$

- for all i $(1 \le i \le n^{1/3} - 1)$,

$$L_i = \{b_{p,q} \mid s_i + 1 \le p \le n^{1/3};\ s_i + 1 + (i-1)n^{1/3} \le q \le s_{i+1} + in^{1/3}\}.$$

It is important to note U_i is the set of elements whose ranks are smaller than k, and L_i is the set of elements whose ranks are greater than k. It is tedious but straightforward to show that

$$\sum_{i=1}^{n^{1/3}} (|U_i| + |L_i|) \ge n - n^{2/3}. \tag{4.2}$$

The interested reader can find the details of the derivation in Appendix A.

Observe that with C standing for the set of candidates as above, (4.2) translates as

$$|C| \le n^{2/3}.$$

51

Finally, note that we have reduced the problem of computing the k-th smallest element in A to the problem of computing the $(k - \sum_{i=1}^{n^{1/3}} |U_i|)$-th smallest element in C.

In the next section we show how these ideas can be implemented efficiently on an $n \times n$ reconfigurable mesh.

4.2.3. The Selection Algorithm — Implementation Details

Consider an $n \times n$ reconfigurable mesh R; the input to the algorithm is a collection $A = \{a_1, a_2, \ldots, a_n\}$ of n elements chosen from a totally ordered universe. The input is assumed stored in the first row of R, with $P(1,i)$ storing a_i for all i ($1 \le i \le n$). During the course of the algorithm, the mesh R will be viewed as consisting of submeshes in a way that suits various computational needs. However, every time such a view is taken, the implied partition will be made explicit to avoid confusion.

The first stage of our algorithm involves constructing the matrix $B[1..n^{1/3}, 1..n^{2/3}]$ featured in Section 4.2.2: recall that B contains all the elements of A and has all rows and columns sorted in increasing order.

To begin, partition the mesh R into $n^{1/3}$ submeshes $M_1, M_2, \ldots, M_{n^{1/3}}$ of size $n \times n^{2/3}$ with M_i ($1 \le i \le n^{1/3}$) involving the columns $1 + (i-1)n^{2/3}$ through $in^{2/3}$ of R. Sort the elements in the first row of each M_i and let $a'_1, a'_2, \ldots, a'_n$ be the corresponding permutation of A. It is worth noting that if the sorting algorithm discussed in Section 4.2.1 is used, Corollary 4.6 guarantees that the previous operation can be performed in $O(1)$ time in each submesh M_i.

Next, every processor in R connects its ports N and S creating vertical buses in every column. Now processor $P(1,i)$ ($1 \le i \le n$) broadcasts a'_i southbound to $P(x_i + y_i n^{1/3}, i)$ with $x_i = \left\lceil \dfrac{i}{n^{2/3}} \right\rceil$ and $y_i = (i-1) \bmod n^{2/3}$.

Further, every processor connects its E and W ports thus creating horizontal buses in every row of R. For all i ($1 \le i \le n$), processor $P(x_i + y_i n^{1/3}, i)$ broadcasts a'_i

52

westbound to $P(x_i + y_i n^{1/3}, 1)$.

At this moment it is convenient to view the original mesh as a collection of $n^{2/3}$ submeshes $S_1, S_2, \ldots, S_{n^{2/3}}$ of size $n^{1/3} \times n$ with S_j $(1 \le j \le n^{2/3})$ involving rows $1 + (j-1)n^{1/3}$ through $jn^{1/3}$ of R. It is easy to verify that, as a result of the previous data movement operation, the first column of each S_j $(1 \le j \le n^{2/3})$ contains in top-down order $a'_j, a'_{j+n^{2/3}}, \ldots, a'_{j+n-n^{2/3}}$.

Again, using the sorting algorithm discussed in Section 4.2.1, sort the first column of every S_j $(1 \le j \le n^{2/3})$ in O(1) time. After this operation we refer to the elements in the first column of S_j as $b_{j,1}, b_{j,2}, \ldots, b_{j,n^{1/3}}$. Now repeating in reverse the data movement operation detailed above, we obtain a one-dimensional row-major layout of matrix B in the first row of the original mesh. The fact that both rows and columns of B are sorted is implied by the Folklore Theorem stated in Section 4.2.1. Furthermore, the layout of B is such that the first row of M_i $(1 \le i \le n^{1/3})$, contains $b_{i,1}$, $b_{i,2}, \ldots, b_{i,n^{2/3}}$. At this moment, it is helpful to note that in the row-major layout of B,

$$\text{processor } P(1,i) \text{ stores } b_{p,q} \text{ with } p = \left\lceil \frac{i}{n^{2/3}} \right\rceil \text{ and } q = (i-1) \bmod n^{2/3} + 1. \quad (4.3)$$

The second stage of our algorithm involves identifying the diagonals $D_1, D_2, D_3, \ldots, D_{n^{1/3}}$ as discussed in Section 4.2.2. Note that it is an easy task for every processor in row 1 of R to mark itself if it contains an element of such a D_i. Note also that row 1 of R contains exactly $n^{2/3}$ marked elements. Now using the ranking procedure specified in Section 4.2.1, we can compute in O(1) time the rank in A of every element of B stored by a marked processor.

If some rank obtained above equals k then the algorithm terminates, returning the desired element. We may, therefore, assume that all the ranks computed above are distinct from k.

53

Now (4.3) will be used to store the matrix B in a different form in all M_i's. Specifically, the rows of B will be stored in rows 1, $1+n^{2/3}$, $1+2n^{2/3}$, . . . , $1+n-n^{2/3}$ of all M_i's.

To achieve this, we first establish vertical buses in every column of R and let processor $P(1,i)$ ($1 \le i \le n$) broadcast $b_{p,q}$ southbound to $P(i,i)$. Next, establishing a horizontal bus in every row of R, we let processor $P(i,i)$ broadcast $b_{p,q}$ to the entire row i. Finally, every processor $P(i,j)$ with $j=q+rn^{2/3}$ ($0 \le r \le n^{1/3}-1$) broadcasts $b_{p,q}$ northbound to $P(1+(p-1)n^{2/3},j)$. It is easy to confirm that after this data movement operation, processor $P(1+(p-1)n^{2/3},q)$ in M_1 stores $b_{p,q}$. Note that from now on we only use the submesh M_1 (recall that M_1 contains columns 1 through $n^{2/3}$ of the original mesh).

The third stage of our algorithm involves computing for all i ($1 \le i \le n^{1/3}$), the parameter s_i (see Section 4.2.2). We proceed as follows: having established a vertical bus in each column of M_1, we let processors storing elements d_j^i of D_i broadcast the corresponding elements northbound to the processors in row 1 (of M_1). Observe that the purpose of this data movement operation is to have all the elements of D_i in positions $1+(i-1)n^{1/3}$ through $in^{1/3}$ of the first row of M_1.

Next, we further subdivide M_1 into submeshes $M_{11}, M_{12}, \ldots, M_{1n^{1/3}}$ each containing $n^{1/3}$ contiguous columns of M_1. By the previous observation, the first row of M_{1i} ($1 \le i \le n^{1/3}$) contains in left-to-right order the elements $d_1^i, d_2^i, \ldots, d_{n^{1/3}}^i$ of the diagonal D_i.

We now describe how to compute s_i in the first row of M_{1i}. First, the processor holding d_1^i broadcasts to the whole row $s_i=0$ or $s_i=-\infty$ depending on whether or not rank$(d_1^i)>k$. In case the processor holding $d_{n^{1/3}}^i$ receives $-\infty$, it will broadcast to the whole row $s_i=n^{1/3}$ or $s_i=+\infty$ depending on whether or not rank$(d_{n^{1/3}}^i)<k$. In case $+\infty$ was broadcast previously, every processor in row 1 reads the rank of the item held by

its right neighbor (provided it exists) and the unique processor that identifies the condition in (1) broadcasts s_i to the whole row.

We now turn to the last stage of our algorithm, namely computing the set C of candidates and identifying the $(k - \sum_{i=1}^{n^{1/3}} |U_i|)$-th smallest element of C. As noted in Section 4.2.2, the k-th smallest element in A is precisely the $(k - \sum_{i=1}^{n^{1/3}} |U_i|)$-th smallest element in C.

First, to compute C we only need mark for all $1 \leq i \leq n^{1/3}$ the elements belonging to U_i and L_i (refer to Section 4.2.2 for definitions). We only demonstrate how the elements of U_i are marked, since marking the elements of L_i is a perfectly similar operation.

Let $s_1, s_2, \ldots, s_{n^{1/3}}$ be the parameters computed above. If $s_i = 0$ then there is nothing to be done, as $U_i = \varnothing$. Otherwise, for all i, let $P(1, j_i)$ be the processor that has identified s_i. Now every processor $P(1, j_i)$ broadcasts s_i to the whole column j_i. Next, horizontal buses are established in all rows $1 + (t-1)n^{2/3}$ $(1 \leq t \leq n^{1/3})$. Further, every processor $P(1 + (t-1)n^{2/3}, j_i)$ with $t \leq s_i$ splits the horizontal bus in row $1 + (t-1)n^{2/3}$ and broadcasts $-\infty$ westbound on its own subbus. Every processor that receives $-\infty$, including the sender itself, marks itself "U-removed". Clearly, after having removed all the elements in U_i and L_i $(1 \leq i \leq n^{1/3})$ what remains are precisely the elements of C.

Before actually computing the set C, it is convenient to compute $\sum_{i=1}^{n^{1/3}} |U_i|$. At this point, we view M_1 as a collection of $n^{1/3}$ submeshes $N_1, N_2, \ldots, N_{n^{1/3}}$ of size $n^{2/3} \times n^{2/3}$, each containing $n^{2/3}$ contiguous rows of M_1. Specifically, for every i $(1 \leq i \leq n^{1/3})$, N_i contains rows $1 + (i-1)n^{2/3}$ through $in^{2/3}$ of M_1. Every processor that has been marked "U-removed" in the previous computational step broadcasts the element of A it holds to the diagonal of the mesh N_j that it belongs to; in turn this diagonal processor broadcasts the item to the first column of N_j. The net result of this data

55

movement operation is that the first column of M_1 contains all the elements of $\bigcup_{i=1}^{n^{1/3}} U_i$ in some order. What remains to be done is to count the elements in $\bigcup_{i=1}^{n^{1/3}} U_i$. For this, we let every processor in column one of M_1 write a 1 or a 0 into a local variable depending on whether or not it stores a "U-removed" element. Now computing the (prefix) sum of the corresponding binary sequence we obtain the desired result.

To compute the set C, every processor that has not been marked "U-removed" or "L-removed" broadcasts the element of A it holds to the diagonal of the mesh N_j that it belongs to; in turn this diagonal processor broadcasts the item to the first column of N_j. The net result is that the first column of M_1 contains all the elements of C in some order. What remains to be done is to sort the elements in C using the submesh M_1 and to pick the $(k-\sum_{i=1}^{n^{1/3}} |U_i|)$-th element in the sorted version of C. To sort C we need to move the elements of C to the first row of M_1. For this movement, we first compact the elements of C in the first column of M_1 (as discussed in Section 4.2.1) and then broadcast every element of C to a unique slot in the first row of M_1. The details of this simple operation are left to the reader.

To summarize our findings we state the following result.

Theorem 4.10. Selecting the k-th smallest element of a collection of n elements chosen from a totally ordered universe can be done in $O(1)$ time on a reconfigurable mesh of size $n \times n$. $\square$

The problem of multi-selection arises frequently in databases. Here, given an unordered set A of n records and a sequence of m integers $1 \le q_1 < q_2 < \cdots < q_m \le n$, we are interested in answering queries of the type "find the q_i-th smallest element in A". The problem is to answer all these queries as fast as possible. We propose to show that our selection algorithm affords us a constant time multi-selection algorithm as long as $m \le n^{1/3}$.

56

Our multi-selection algorithm proceeds exactly as the selection algorithm up to the point where only M_1 was used to return the k-th smallest element of the collection. Notice that, in fact, we have answered *one* query in M_1. Clearly, we could have, just as well, answered, in parallel all $m \leq n^{1/3}$ queries by using all the submeshes M_1, $M_2, \ldots, M_{n^{1/3}}$. Consequently, we have the following result.

Theorem 4.11. The multi-selection problem involving $m \leq n^{1/3}$ queries can be solved in $O(1)$ time on a reconfigurable mesh of size $n \times n$. $\square$

4.2.4. The VLSI-Optimal Sorting Algorithm

We are now in a position to show how the pieces fit together to produce a very efficient sorting algorithm for reconfigurable meshes. For this purpose, consider an arbitrary set $X = \{x_1, x_2, \ldots, x_n\}$ of elements chosen from a totally ordered universe. Let R be a reconfigurable mesh of size $n \times n$ storing the input sequence in the first row, with $P(1,i)$ containing x_i for all i.

To begin, we solve the multi-selection problem on X involving the set of queries $Q = \{q_1, q_2, \ldots, q_{n^{1/3}}\}$ with q_i $(1 \leq i \leq n^{1/3})$ asking for the $in^{2/3}$-th smallest element in X. Note that the multi-selection algorithm developed in the previous section allows us to solve this instance of the multi-selection problem in $O(1)$ time of the mesh R.

Let $X' = \{x'_1, x'_2, \ldots, x'_{n^{1/3}}\}$ be the subset of X with x'_i $(1 \leq i \leq n^{1/3})$ the $in^{2/3}$-th smallest element in X. Note that having obtained X' amounts to having $n^{1/3}$ implicit buckets of *exactly* $n^{2/3}$ elements each. Our plan is to place the elements of X into the corresponding buckets and, using the sorting algorithm discussed in Section 4.2.1, to sort each such bucket. This will clearly amount to sorting X itself.

The only point that needs to be clarified is how the elements of X are placed into those buckets in $O(1)$ time. For this purpose, establish vertical buses in all columns of R, and let every processor in the first row broadcast the value it holds to the whole column. The unique processor in row $in^{2/3}$ $(1 \leq i \leq n^{1/3})$ that holds x'_i broadcasts x'_i to

57

the whole row $in^{2/3}$.

In every column j of R we do the following. Processor $P(in^{2/3},j)$ $(1 \le i \le n^{1/3})$ writes a 1 into a local variable whenever x_j is larger than or equal to x'_i; otherwise, it writes a 0. Furthermore, in every column j of R it is easy to identify the unique t for which

$$P((t-1)n^{2/3},j) \text{ stores a 0 and } P(tn^{2/3},j) \text{ stores a 1.} \qquad (4.4)$$

We now view the mesh R as being composed of $n^{1/3}$ submeshes S_1, $S_2, \ldots, S_{n^{1/3}}$ of size $n^{2/3} \times n$, with S_i $(1 \le i \le n^{1/3})$ involving rows $1+(i-1)n^{2/3}$ through $in^{2/3}$ of R. In our scheme, for all i, the submesh S_i will play the role of the i-th bucket discussed above.

Observe that the purpose of the previous data movement operation was to establish for every element x_j of X the bucket it belongs to. Specifically, we let x_j belong to the t-th bucket whenever the condition of (4.4) above holds for x_j.

As noted above, our choice of the query-set guarantees that every bucket contains precisely $n^{2/3}$ elements. It is now a straightforward operation to sort the elements in every bucket in $O(1)$ time using the sorting algorithm discussed in Section 4.2.1.

To summarize our findings we state the main result.

Theorem 4.12. An n-element sequence chosen from a totally ordered universe can be sorted in $O(1)$ time on a reconfigurable mesh of size $n \times n$. $\square$

4.3. A Sub-Optimal Convex Hull Algorithm

In this section, we will exhibit a sub-optimal convex hull algorithm. This algorithm will be later refined into an optimal one. Let $S = \{p_1, p_2, \ldots, p_N\}$ be a set of points in the plane; here, p_i $(1 \le i \le N)$ is represented by its cartesian coordinates (x_i, y_i). To avoid tedious details we assume, without loss of generality, that the points in S are in *general* position, with no three collinear and no two having the same x or y coordinates. The details of our sub-optimal convex hull algorithm follow.

58

Step 1. Find the four extreme points in S in the x and y direction, and let them be without loss of generality, p_1, p_2, p_3, and p_4. Specifically, $x_1 = \max_{1 \le j \le N} \{x_j\}$, $y_2 = \max_{1 \le j \le N} \{y_j\}$, $x_3 = \min_{1 \le j \le N} \{x_j\}$, and $y_4 = \min_{1 \le j \le N} \{y_j\}$.

Step 2. Compute the sets (refer to Figure 4.2 for an illustration)

$$S_1 = \{p_i \mid x_2 \le x_i \le x_1; \ y_1 \le y_i \le y_2\},$$

$$S_2 = \{p_i \mid x_3 \le x_i \le x_2; \ y_3 \le y_i \le y_2\},$$

$$S_3 = \{p_i \mid x_3 \le x_i \le x_4; \ y_4 \le y_i \le y_3\},$$

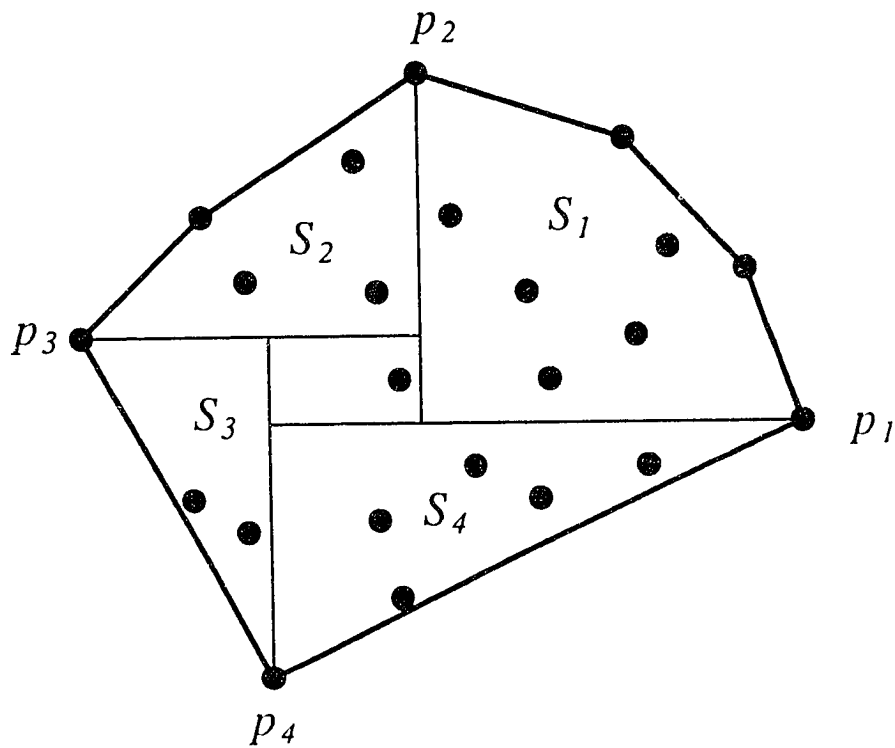$$S_4 = \{p_i \mid x_4 \le x_i \le x_1; \ y_4 \le y_i \le y_1\}.$$

Figure 4.2: Illustrating S_1, S_2, S_3 and S_4

For further reference, we take note of the following result.

59

Lemma 4.13. The upper hull $U(S)$ of S is the concatenation of the upper hulls U_1 and U_2 of S_1 and S_2, respectively. Similarly, the lower hull $L(S)$ of S is the concatenation of the lower hulls L_3 and L_4 of S_3 and S_4, respectively.

Proof. Suppose not. Enumerate the points on $U(S)$ in counterclockwise order as $p_1 = u_1, u_2, \ldots, u_s = p_3$ $(s \geq 3)$; similarly, enumerate the points on the concatenation of U_1 and U_2 in counterclockwise order as $p_1 = v_1, v_2, \ldots, u_t = p_3$ $(t \geq 3)$. Let i be the first subscript for which the two enumerations differ. Specifically, u_i is different from v_i and for all $1 \leq j < i$, $u_j = v_j$.

Since u_i belongs to $U(S)$, u_i must lie outside of the triangle determined by p_1, p_2, and p_3. This observation restricts u_i to either S_1 or S_2. Without loss of generality, assume that u_i belongs to S_1. Consequently, $u_{i-1}(=v_{i-1})$ and v_i also belong to S_1. Since U_1 is the upper hull of S_1, it must be that u_i lies in the closed left halfplane determined by the directed line $\overrightarrow{v_{i-1}v_i}$. On the other other hand, since $U(S)$ is the upper hull of S, v_i is in the closed left halfplane determined by the directed line $\overrightarrow{u_{i-1}u_i}$, it must be that $u_{i-1}(v_{i-1})$, u_i, and v_i are collinear. It contradicts either that u_i is in $U(S)$ or that v_i is in U_1.

The proof of the fact that the lower hull $L(S)$ is the concatenation of the lower hulls L_3 and L_4 of S_3 and S_4 is similar. $\square$

Note that by virtue of Lemma 4.13, we only need to compute the upper hulls of S_1, S_2 and the lower hulls of S_3 and S_4. For simplicity, we shall deal with S_1 only, the others being similar.

Step 3. Sort the points in S_1 by increasing y-coordinate, and let $L = (p_1 = q_1, q_2, \ldots, q_l = p_2)$ be the resulting sorted sequence;

Step 4. For every j $(1 \leq j \leq t)$, find the subscript $f(j)$ $(j < f(j) \leq t)$ such that the angle determined by $q_{f(j)}$, q_j, and the negative direction of the x-axis is as large as possible; (see Figure 4.3)
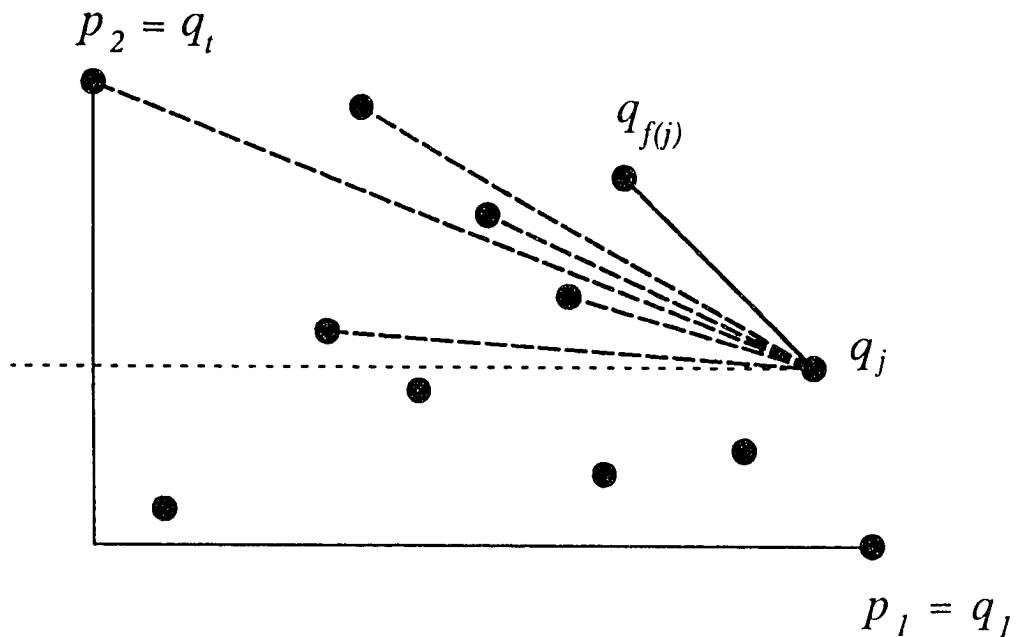
$$p_2 = q_t$$

$$q_{f(j)}$$

$$q_j$$

$$p_1 = q_1$$

Figure 4.3: Computing $f(j)$ of q_j

Step 5. Compute the prefix maxima of the values $f(j)$'s in L; specifically. let $g(j) = \max_{1 \le t \le j-1} \{f(t)\}$. (see Figure 4.4, where for every point q_j the pair $(f(j), g(j))$ is featured)

Step 6. Eliminate all the points q_j for which $f(j) \le g(j)$. Report all the remaining points in L, including q_1 and q_t.

The following result argues for the correctness of our algorithm.

Theorem 4.14. At the end of Step 6, all the remaining points in L belong to the upper hull of S_1.

Proof. The conclusion implied by the following stronger statement. Let U_1 stand for the upper hull of S_1.

$$q_j \ (2 \le j \le t-1) \text{ belongs to } U_1 \text{ if and only if } f(j) > g(j). \tag{4.5}$$

61

First, if q_j ($2 \leq j \leq t-1$) belongs to the upper hull U_1 then let q_i and q_k be its neighbors on the upper hull. We note that since q_1 and q_t trivially belong to the upper hull U_1, and that the points q_i and q_k are well defined. Clearly, $f(i)=j$ and so $g(j)=j<k=f(j)$, as claimed.

Conversely, if q_j does not belong to U_1 then let q_i and q_k be the closest points on U_1 such that q_j lies on the chain from q_i to q_k. Since q_i and q_k are neighbors on the upper hull U_1, we have $f(i)=k$; furthermore, $f(j) \leq k = g(j)$, and the conclusion follows. □
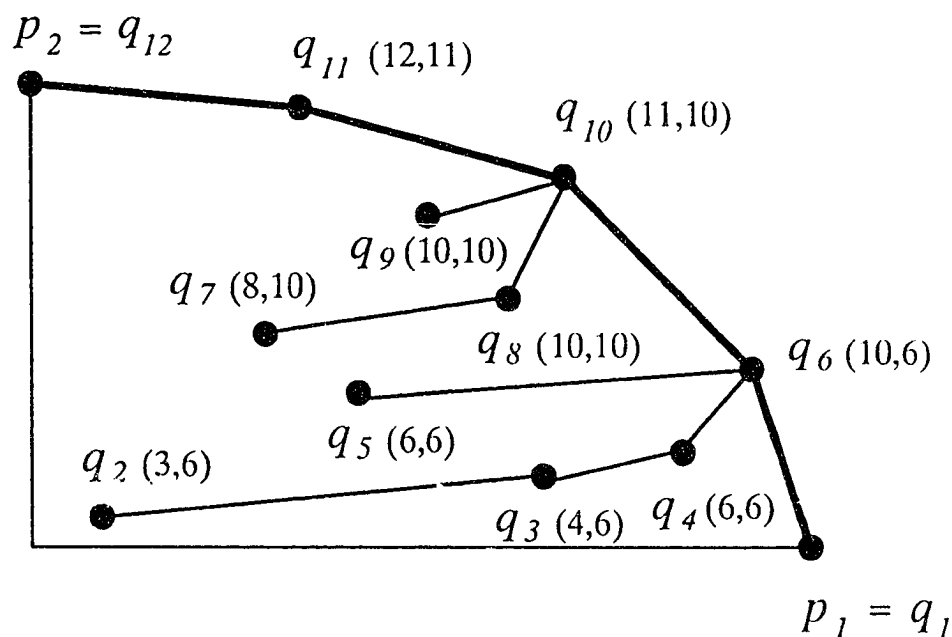

Figure 4.4: The pair $(f(j), g(j))$ of every q_j

Since the correctness is settled, we now proceed to analyze the running time. We assume a reconfigurable mesh of size $NM \times N$ with $2 \leq M \leq N$. Step 1 can be implemented to run in $O(1)$ time on a reconfigurable mesh of size $N \times N$ since we only need compute $\max_{1 \leq j \leq N} \{z_j\}$ and $\min_{1 \leq j \leq N} \{z_j\}$ with $z_j = x_j$ and $z_j = y_j$.

Step 2 is demonstrated for S_1 only; computing S_i with $i=2,3,4$ is similar. All that is needed is to establish a subbus running through the entire row 1. The processors storing p_1 and p_2 broadcast, in two computational steps, their cartesian coordinates to all processors in row 1; every processor that stores a point in S_1 marks itself. Thus Step 2 runs in O(1) time.

Step 3 can be implemented as follows. First, all unmarked processors change the y-coordinate of the point that they store to $+\infty$. Now the sorting algorithm in Section 4.2 is invoked: this runs in O(1) time and needs a mesh of size $N \times N$. It is helpful to assume that at the end of Step 3, processors $P(1,1)$, $P(1,2)$, . . . , $P(1,t)$ contain $L=(p_1=q_1, q_2, \ldots, q_t=p_2)$ in sorted order.

Step 4 can be implemented to run in O(log N/log M) time on a reconfigurable mesh of size $MN \times N$ as follows. Assume that, initially, for all $1 \leq j \leq t$, $P(1,j)$ stores q_j. Establish vertical subbuses in each column and let $P(1,j)$ broadcast the cartesian coordinates of q_j along the subbus in column j $(1 \leq j \leq t)$. Next, establish horizontal buses running from $P((j-1)M+1,j)$ to $P((j-1)M+1,t)$ $(1 \leq j \leq t)$. For all j, $P((j-1)M+1,j)$ broadcasts the cartesian coordinates of q_j eastbound on the horizontal subbus in row $(j-1)M+1$. Finally, every processor $P((j-1)M+1,k)$ with $j<k \leq t$ computes the angle determined by q_k, q_j and the negative direction of the x axis.

We now subdivide the mesh into submeshes of size $M \times N$ as follows. The first $M \times N$ submesh involves the first M rows, the second the next M rows and so on. Now Theorem 4.2 guarantees that for all j $(1 \leq j \leq t)$, f(j) can be computed in O(log N/log M) time. (Actually, to compute the f(j)'s only the maximum is needed, not the prefix maxima.) It is easy to arrange for f(j) in row $(j-1)M+1$ to be sent back to $P(1,j)$. This, clearly takes O(1) time since only the appropriate buses have to be established and the information broadcast along them.

Step 5 can be implemented to run in O(1) time on a reconfigurable mesh of size $N \times N$ by using the prefix maxima algorithm discussed in Section 4.1. Finally, Step 6

63

involves marking every $P(1,j)$ that contains a point of the convex hull. To complete the algorithm, the points of the convex hull are compacted into the leftmost position in the first row of the mesh. By Theorem 4.3, this takes $O(1)$ time on a reconfigurable mesh of size $N \times N$.

To summarize our discussion we state the following result.

Theorem 4.15. The convex hull of a set of N planar points can be computed in $O(\log N / \log M)$ time on a reconfigurable mesh of size $NM \times N$ with $2 \leq M \leq N$. $\square$

In particular, with $M = N$, Theorem 4.15 implies the following result.

Corollary 4.16. The convex hull of a set of N planar points can be computed in $O(1)$ time on a reconfigurable mesh of size $N^2 \times N$. $\square$

4.4. An Optimal Convex Hull Algorithm

To make our presentation more transparent and easier to understand, we first present the details of a simple routine that finds the supporting line of two upper hulls U and V that do not overlap in the x direction.

We assume that both U and V have size $\sqrt{n}$ and that the points in U and V are sorted by x coordinate; since U and V are non-overlapping, we may assume without loss of generality that all the points in U have smaller x coordinates than those in V. To merge these two hulls we shall use a reconfigurable mesh of size $\sqrt{n} \times \sqrt{n}$ with every processor $P(i,j)$ containing a point of U and a point of V. More precisely, let $u_1, u_2, \ldots, u_{\sqrt{n}}$ and $v_1, v_2, \ldots, v_{\sqrt{n}}$ be the points of U and V in left to right order. Now for every i, j ($1 \leq i, j \leq \sqrt{n}$), processor $P(i,j)$ of the mesh stores u_i and v_j.

For every fixed i ($1 \leq i \leq \sqrt{n}$) every processor $P(i,j)$ in row i checks whether the hull points v_{j-1} and v_{j+1} (provided they exist) lie below the line determined by u_i and v_j. Clearly, in every row of the mesh, exactly one processor detects this condition. By using suitably constructed horizontal buses, the processor $P(i,j)$ detecting the condition above sends v_j to $P(i,i)$. In turn, $P(i,i)$ broadcasts the ordered pair (u_i, v_j)

64

vertically, to all the processors in column i of the mesh: recall that these processors store the points of the hull U. Now every processor in column i checks whether the hull point in U it stores lies below the line determined by u_i and v_j. It is easy to see that $\overline{u_i v_j}$ is a supporting line if and only if every processor in column j detects this condition. Clearly, there exists exactly one such supporting line for the hulls U and V. Note that the entire computation is performed in $O(1)$ time. Therefore, we can state the following intermediate result.

Lemma 4.17. Consider a reconfigurable mesh of size $\sqrt{n} \times \sqrt{n}$ and let U and V be two non-overlapping upper hulls of size at most $\sqrt{n}$. If the elements of U (V) are replicated to all the rows (columns) of the mesh, then the supporting line of the two hulls can be computed in $O(1)$ time. $\square$

We are now in a position to show how the sub-optimal convex hull algorithm developed in the previous section and the simple merging algorithm detailed above can be used for obtaining a constant time convex hull algorithm for reconfigurable meshes.

Consider, again, an arbitrary set S of points $p_1, p_2, \ldots, p_n$ in general position with every p_i specified by its cartesian coordinates (x_i, y_i). The set is assumed stored, one point per processor, in the first row of a reconfigurable mesh of size $n \times n$.

As in the case of meshes with multiple broadcasting, our strategy involves computing the *upper* and *lower hull* of the set S. Symmetry allows us to restrict ourselves to describing how the upper hull is computed: the computation leading to the lower hull is similar.

Our optimal convex hull algorithm begins by sorting the set of points in increasing order of their x coordinates: this takes $O(1)$ time if the algorithm in Section 4.2 is used. Next, we view the original mesh as consisting of submeshes of size $n \times \sqrt{n}$: the first such submesh involves the first $\sqrt{n}$ columns, the second involves the next $\sqrt{n}$ columns, and so on. It is important to note that in every submesh constructed above, we can determine, using the sub-optimal algorithm of the previous section, the upper

65

hull of $\sqrt{n}$ of the original points in the set S. For further reference, we refer to these upper hulls as $H_1, H_2, \ldots, H_{\sqrt{n}}$ in left to right order.

In preparation for the next step, we view the original mesh as a collection of submeshes $R_{i,j}$ of size $\sqrt{n} \times \sqrt{n}$ such that for all i, j ($1 \le i, j \le \sqrt{n}$), $R_{i,j}$ involves the processors $P(a,b)$ with $1+(i-1)\sqrt{n} \le a \le i\sqrt{n}$ and $1+(j-1)\sqrt{n} \le b \le j\sqrt{n}$. As a result of the previous computational step, every $R_{1,j}$ contains, in its first row, the upper hull of the points $p_{(j-1)\sqrt{n}+1}$ through $p_{j\sqrt{n}}$.

Our next step involves merging the $\sqrt{n}$ upper hulls. This task begins by replicating the information in the first row of the original mesh (i.e. the $\sqrt{n}$ upper hulls) to all the rows in the mesh by using appropriately constructed vertical buses. For every i ($1 \le i \le \sqrt{n}$), every processor on the diagonal of $R_{i,i}$ storing a point of H_i, broadcasts its coordinates horizontally, along the bus in its row. Note that this data movement has the effect of replicating the points of the upper hulls in a way consistent with the hypothesis of Lemma 4.17. Now in every $R_{i,j}$ ($1 \le i < j \le \sqrt{n}$) the supporting line of H_i and H_j is computed in O(1) time as previously described.

Our next task is to determine whether the supporting line between upper hulls H_i and H_j is a supporting line for the set $\bigcup_{i=1}^{\sqrt{n}} H_i$. This task is accomplished as follows: for every i ($1 \le i \le \sqrt{n}$), the previous computational step produces $\sqrt{n} - i$ supporting lines. The ordered pair consisting of a point in H_i and a point in H_j with $i < j$ is assigned to row j of $R_{i,j}$, and then broadcast horizontally to the whole row of the original mesh. As before, every processor detects whether the hull point it stores lies below the line determined by the ordered pair it receives. If some point lies above this line, then the line is not a supporting line for $\bigcup_{i=1}^{\sqrt{n}} H_i$.

It is easy to see that the operation described above produces all the supporting lines of the set $\bigcup_{i=1}^{\sqrt{n}} H_i$. Note that for each supporting line detected, all the points that lie

66

in the x direction between the endpoints of the supporting line cannot be on the convex hull and can, therefore, be eliminated. Finally, the remaining convex hull points are compacted into the leftmost positions in the first row of the mesh. By Theorem 4.3, this can be done in O(1) time.

To summarize our findings we state the following result.

Theorem 4.18. The convex hull of an n-element set of points in the plane, stored one item per processor in the first row of a reconfigurable mesh of size $n \times n$ can be computed in O(1) time. $\square$

CHAPTER 5

CONCLUSIONS AND OPEN PROBLEMS

Typical image processing and computer vision tasks found currently in industrial, medical, and military applications require real-time solutions. This requirement has motivated the design of many parallel architectures and algorithms. Recently, a new architecture called the reconfigurable mesh has been proposed. It has been claimed that the reconfigurable mesh is suitable for VLSI implementation because of its regular structure. Further, it has been argued that the reconfigurable mesh can be used as a universal chip capable of simulating any equivalent-area architecture without loss of time. In this thesis we have addressed a number of image processing and computer vision problems on reconfigurable meshes. Among the low-level vision tasks, we have addressed the problem of computing the perimeter, area, histogram and median row of a digitized image. Among the mid-level vision tasks, we have addressed the Hough transform and component labeling. We have also addressed the problem of computing the convex hull of a set of planar points.

In Chapter 2, we first showed how to compute the prefix sums of a binary sequence and the prefix sums of an integer sequence by using the reconfigurability of the bus systems. We then showed how to compute the sum of all the integers in a matrix. Finally, we showed that a number of low-level descriptors of a digitized image such as the perimeter, area, histogram and median row can be reduced to computing the sum of all the integers in a matrix. Another important class of descriptors of a digitized image is the *moments*. Specifically, the (p,q)-th moment of a binary image $IM[1..n,1..n]$ is given by

$$m_{pq} = \sum_{i=1}^{n} \sum_{j=1}^{n} i^p j^q IM[i,j].$$

(5.1)

68

As it turns out, m_{00} is precisely the area of the image. The *center* of an image is a point of coordinates x_c, y_c defined as

$$x_c = \frac{m_{10}}{m_{00}} \text{ and } y_c = \frac{m_{01}}{m_{00}}.$$

Finally, m_{02} and m_{20} are the moments of inertia around the i and j axes, respectively. Furthermore, $m_{02}+m_{20}$ yields the moment of inertia around the origin. It is easy to see that computing the double sum in (5.1) amounts to computing the sum of all the integers in a matrix, the integers being in the range from 0 to $O(n^{p+q})$. However, in our algorithm for computing the sum of all the integers in a matrix, the ranges of the integers is restricted to $[0,\log n]$. We do not know how to compute the moments in $O(\log\log n)$ time.

In Chapter 3, we first proposed a new computational paradigm, that is, identifying an entity with a bus and using computations on the bus to obtain properties of the entity. The computations that we introduced include finding the maximum on a bus (which allows us to elect a leader of a closed bus), ranking an arbitrary open bus, and computing the prefix maxima (sums) on a bus. To illustrate our paradigm, we showed how to solve the problems of computing the Hough transform and component labeling. However, our algorithm for ranking a bus of length N takes $O(\log N \log^* N)$ time. It would be of interest to see whether an $O(\log N)$ time bus ranking algorithm can be obtained. Such an algorithm would reduce the overall complexities of many algorithms. In our component labeling algorithm, every step except for electing a leader takes $O(1)$ time. It is also interesting to see whether electing a leader can be avoided or whether our algorithm for electing a leader can be improved.

In Chapter 4, we have developed a VLSI-optimal constant time sorting algorithm and a VLSI-optimal constant time convex hull algorithm. In both algorithms, we only handled the sparse case. For the dense case, i.e., computing the convex hull of n planar points on a reconfigurable mesh of size $\sqrt{n} \times \sqrt{n}$, it is easy to get an $O(\sqrt{n})$

69

algorithm by combining the sorting algorithm on a mesh [36,60] and the result of Miller *et al.* [34]. It is also time-optimal or VLSI-optimal. It is of interest to develop a time-optimal and VLSI-optimal algorithm for handling the general case, that is, computing the convex hull of m ($\sqrt{n} \leq m \leq n$) planar points on a reconfigurable mesh of size $\sqrt{n} \times \sqrt{n}$.

Further, in this thesis, we have developed two basic data movement techniques for reconfigurable meshes. One is computing the prefix sums of a binary sequence of size n. The other is computing the prefix maxima of a sequence of n real numbers. Both algorithms take $O(\frac{\log n}{\log m})$ time on a reconfigurable mesh of size $m \times n$ with $2 \leq m \leq n$. Therefore, they are *adaptive* in the sense that they can be executed on a reconfigurable mesh featuring a number of rows independent of the size of the input. These two techniques are interesting in their own right. In addition, they can be exploited to obtain efficient algorithms for a number of computational problems.

BIBLIOGRAPHY

[1] A. Aggarwal, Optimal Bounds for Finding Maximum on an Array Processor with K Global Buses, *IEEE Trans. Computers*, Vol. C-35, No. 1, 62-64, 1986.

[2] T. Bestul and L. S. Davis, On Computing Complete Histograms of Images in Log(n) Steps Using Hypercubes, *IEEE Trans. Pattern Anal. Mach. Intell.*, Vol. PAMI-11, No. 2, 212-213, 1981.

[3] 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.

[4] S. H. Bokhari, Finding Maximum on an Array Processor with a Global Bus, *IEEE Trans. Computers*, Vol. C-33, No. 2, 62-64, 1986.

[5] R. Cole and U. Vishkin, Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking, *Information and Control*, 70, 32-53, 1986.

[6] R. Cypher, J. L. C. Sanz and L. Snyder, Hypercube and Shuffle-Exchange Algorithms for Image Component Labeling, *Proc. IEEE Workshop on Comput. Arch. for Pattern Anal. Mach. Intell.*, 1987, 5-10.

[7] R. Cypher, J. L. C. Sanz and L. Snyder, The Hough Transform Has O(N) Complexity on SIMD NXN Mesh Array Architectures, *Proc. IEEE Workshop on Comput. Arch. for Pattern Anal. Mach. Intell.*, 1987, 115-121.

[8] R. Cypher, J. L. C. Sanz and L. Snyder, EREW PRAM and Mesh Connected Computer Algorithms for Image Component Labeling, *Proc. IEEE Workshop on Comput. Arch. for Pattern Anal. Mach. Intell.*, 1987, 122-128.

[9] M. Fairhurst, Computer Vision and Robotic Systems, Prentice-Hall, Englewood

Cliffs, NJ, 1988.

[10] K. S. Fu, R. C. Gonzales and C. S. G. Lee, Robotics, McGraw-Hill, 1987.

[11] C. Guerra and S. Hambrush, Parallel Algorithms for Line Detection on a Mesh, *Proc. IEEE Workshop on Comput. Arch. for Pattern Anal. Mach. Intell.*, 1987, 99-106.

[12] K. Hwang, H. M. Alnuweiri, V. K. Prasanna Kumar and D. Kim, Orthogonal Multiprocessor Sharing Memory with an Enhanced Mesh for Integrated Image Understanding, *CVGIP: Image Understanding*, Vol. 53, No. 1, 31-45, 1991.

[13] J. Illingworth and J. Kittler, A Survey of the Hough Transform, *Computer Vision, Graphics, and Image Processing*, 44, 87-116, 1988.

[14] J.-F. Jenq and S. Sahni, Reconfigurable Mesh Algorithms for the Area and Perimeter of Image Components, *Proc. International Conference of Parallel Processing*, 1991, III, 280-281.

[15] J.-F. Jenq and S. Sahni, Reconfigurable Mesh Algorithms for the Hough Transform, *Proc. International Conference of Parallel Processing*, 1991, III, 34-41.

[16] J.-F. Jenq and S. Sahni, Histogramming on a Reconfigurable Mesh Computer, *Proc. 6th International Parallel Processing Symposium*, Beverly Hill, 1992.

[17] J. Jolion and A. Rosenfeld, An O(log$n$) Pyramid Hough Transform, *Pattern Recognition Letters*, 9, 343-349, 1989.

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

[19] S. Levialdi, On Shrinking Binary Picture Patterns, *Comm. of the ACM*, 15, 7-10, 1972.

[20] M. D. Levine, Vision in Man and Machine, McGraw-Hill, New York, 1985.

72

[21] H. Li and M. Maresca, Polymorphic-Torus Network, *IEEE Trans. Computers*, Vol. C-38, No. 9, 1345-1351, 1989

[22] H. Li and M. Maresca, Polymorphic-Torus Architecture for Computer Vision, *IEEE Trans. Pattern Anal. Mach. Intell.*, Vol. 11, No. 3, 233-243, 1989.

[23] R. Lin, S. Olariu, J. L. Schwing and J. Zhang, Sorting in O(1) Time on an $N \times N$ Reconfigurable Mesh, *Parallel Computing: From Theory to Sound Practice, Proc. 9th European Workshop on Parallel Computing*, Spain, 1992, 16-27, IOS Press.

[24] W.-N. Lin and V. K. Prasanna Kumar, Efficient Histogramming on Hypercube SIMD Machines, *Computer Vision, Graphics, and Image Processing*, 49, 104-120, 1990.

[25] C. A. Lindley, Practical Image Processing in C, John Wiley & Sons, New York, 1991.

[26] J. J. Little, G. Blelloch and T. Cass, Parallel Algorithms of Computer Vision on the Connection Machine, *Proc. Int. Conf. on Computer Vision*, 1987.

[27] J. J. Little, G. Blelloch and T. Cass, Algorithmic Techniques for Computer Vision on a Fine-Grained Parallel Machine, *IEEE Trans. Pattern Anal. Mach., Intell.*, Vol. PAMI-11, No. 3, 244-259, 1989.

[28] M. Manohar and H. K. Ramapriyan, Connected Component Labeling of Binary Images on a Mesh Connected Massively Parallel Processor, *Computer Vision, Graphics, and Image Processing*, 45, 133-149, 1989.

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

[30] R. Miller, V. K. Prasanna-Kumar, D. Reisis and Q. F. Stout, Meshes with Reconfigurable Buses, *Proc. 5th MIT Conf. on Advanced Research in VLSI*,

73

1988, 163-178.

[31] R. Miller, V. K. Prasanna-Kumar, D. Reisis and Q. F. Stout, Data Movement Operations and Applications on Reconfigurable VLSI Arrays, *Proc. International Conference on Parallel Processing,* 1988, I, 205-208.

[32] R. Miller, V. K. Prasanna-Kumar, D. Reisis and Q. F. Stout, Image Computations on Reconfigurable VLSI Arrays, *Proc. IEEE Conf. on Computer Vision and Pattern Recognition,* 1988, 925-930.

[33] R. Miller and Q. Stout, Geometric Algorithms for Digitized Pictures on a Mesh-Connected Computer, *IEEE Trans. Pattern Anal. Mach. Intell.,* Vol. 7, 216-228, 1985.

[34] R. Miller and Q. Stout, Efficient Parallel Convex Hull Algorithms, *IEEE Trans. Computers,* Vol. 37, 1605-1616, 1988.

[35] R. Miller and Q. Stout, Mesh Computer Algorithms for Computational Geometry, *IEEE Trans. Computers,* Vol. 38, 321-340, 1989.

[36] D. Nassimi and S. Sahni, Bitonic sort on a mesh-connected parallel computer, *IEEE Trans. Computers,* Vol. C-27, 1979.

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

[38] S. Olariu, J. L. Schwing, and J. Zhang, Fundamental Data Movement Algorithms for Reconfigurable Meshes, *Proc. 11th annual International Phoenix Conference on Computers and Communications,* 1992, 472-479.

[39] S. Olariu, J. L. Schwing, and J. Zhang, Fundamental Algorithms on Reconfigurable Meshes, *Proc. 29th Annual Allerton Conference on Communications, Control, and Computing,* 1991, 811-820.

[40] S. Olariu, J. L. Schwing and J. Zhang, Integer Problems on Reconfigurable

Meshes, with Applications, to appear in *Journal of Computer and Software Engineering*.

[41] S. Olariu, J. L. Schwing and J. Zhang, Fast Computer Vision Algorithms on Reconfigurable Meshes, *Image and Vision Computing*, Vol. 10, 610-616, 1992.

[42] S. Olariu, J. L. Schwing and J. Zhang, Computing the Hough Transform on Reconfigurable Meshes, *Proc. Vision Interface '92*, Vancouver, Canada.

[43] S. Olariu, J. L. Schwing and J. Zhang, Fast Component Labeling on Reconfigurable Meshes, *Computing and Information, Proc. International Conference on Computing and Information*, Toronto, 1992, 121-124.

[44] S. Olariu, J. L. Schwing, and J. Zhang, A Fast Adaptive Convex Hull Algorithm on Two-Dimensional Processor Arrays with a Reconfigurable Bus System, *Proc. 3rd NASA Symp. on VLSI Design*, 1991, 13.2.1-13.2.9.

[45] S. Olariu, J. L. Schwing and J. Zhang, Time-Optimal Sorting and Applications on nxn Enhanced Meshes, *Proc. IEEE International Conference on Computer Systems and Software Engineering*, The Netherlands, 1992.

[46] V. K. Prasanna-Kumar and C. S. Raghavendra, Array Processor with Multiple Broadcasting, *Journal of Parallel and Distributed Processing*, 4, 173-190, 1987.

[47] V. K. Prasanna-Kumar and D. I. Reisis, Image Computations on Meshes with Multiple Broadcast, *IEEE Trans. Pattern Anal. Mach. Intell.*, Vol. 11, No. 11, 1194-1202, 1989.

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

[49] D. V. Ramanamurthy, N. J. Dimopoulos, K. F. Li, R. V. Patel and A. J. Al-Khalili, Parallel Algorithms for Low Level Vision on the Homogeneous Multiprocessor, *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, 1986.

[50] S. Ranka and S. Sahni, Computing Hough Transform on Hypercube Multicomputers, *Journal of Supercomputing*, 4, 169-190, 1990.

[51] A. P. Reeves, Parallel Computer Architecture for Image Processing, *Computer Vision, Graphics, and Image Processing*, 25, 68-88, 1984.

[52] A. Rosenfeld, Parallel Image Processing Using Cellular Arrays, *IEEE Computer*, 16, 14-20, 1983.

[53] A. Rosenfeld, J. Ornelas and Y. Hung, Hough Transform Algorithms for Mesh-Connected SIMD Parallel Processors, *Computer Vision, Graphics, and Image Processing*, 41, 293-305, 1988.

[54] A. Rosenfeld, A Report on the DARPA Image Understanding Architecture Workshop, *Proc. 1987 DARPA Image Understanding Workshop*, Morgan Kaufmann, Los Altos, CA, 1987, 298-302.

[55] A. Rosenfeld and A. Kak, Digital Picture Processing, Academic Press, 1982.

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

[57] Y. Shiloach and U. Vishkin, An O(log$n$) Parallel Connectivity Algorithm. *J. Algorithms*, 3, 57-67, 1982.

[58] Q. F. Stout, Mesh Connected Computers with Broadcasting, *IEEE Trans. Computers*, Vol. 32, 826-830, 1983.

[59] Q. F. Stout, Meshes with Multiple Buses, *Proc. 27th IEEE Symp. on the Foundations of Computer Science*, 1986, 264-273.

[60] C. D. Thompson and H. T. Kung, Sorting on a Mesh-Connected Parallel Computer, *Comm. of the ACM*, 20, 263-270, 1977.

[61] B.-F. Wang, G.-H. Chen and H. Li, Configurational Computation: a New Computation Method on Processor Arrays with Reconfigurable Bus Systems, *Proc. International Conference on Parallel Processing*, 1991, III, 42-49.

76

APPENDIX A

In this appendix we propose to prove that

$$\sum_{i=1}^{n^{1/3}}(|U_i| + |L_i|) \geq n - n^{2/3}. \tag{A.1}$$

To begin, we note the following simple observation that follows directly from Lemma 4.8.

$$\text{For all } i, j \ (1 \leq i \leq j \leq n^{1/3}), \ s_i \geq s_j. \tag{A.2}$$

Observe that the definitions in Section 4.2.2 immediately imply that

- $|U_1| = s_1 s_1$;

- $|L_{n^{1/3}}| = (n^{1/3} - s_{n^{1/3}})(n^{1/3} - s_{n^{1/3}})$;

- for $i = 2, 3, \ldots, n^{1/3}$

$$|U_i| = s_i(s_i - s_{i-1} + n^{1/3});$$

- for $i = 1, 2, \ldots, n^{1/3} - 1$

$$|L_i| = (n^{1/3} - s_i)(s_{i+1} - s_i + n^{1/3});$$

For technical reasons, we find it convenient to augment the notation by defining

$$s_0 = n^{1/3} \text{ and } s_{n^{1/3}+1} = 0. \tag{A.3}$$

Note that the definitions in (A.3) are consistent with (A.2); furthermore, in this notation we can write for all $i = 1, 2, \ldots, n^{1/3}$

- $|U_i| = s_i(s_i - s_{i-1} + n^{1/3})$;

and

- $|L_i| = (n^{1/3} - s_i)(s_{i+1} - s_i + n^{1/3})$.

To settle (A.1), we propose to show that

$$\sum_{i=1}^{n^{1/3}}(|U_i| + |L_i|) = n - n^{2/3} + \sum_{i=0}^{n^{1/3}}(s_i - s_{i+1})^2. \tag{A.4}$$

77

For this purpose, note that we can write for all $i = 1, 2, \ldots, n^{1/3}$

$$|U_i| + |L_i| = n^{2/3} - n^{1/3}(s_i - s_{i+1}) + 2s_i^2 - s_i s_{i-1} - s_i s_{i+1}.$$

Now

$$\sum_{i=1}^{n^{1/3}} (|U_i| + |L_i|) = n - n^{1/3} \sum_{i=1}^{n^{1/3}} (s_i - s_{i+1}) + \sum_{i=1}^{n^{1/3}} (s_i - s_{i+1})^2 + s_1^2 - s_1 s_0$$

which can be written as

$$\sum_{i=1}^{n^{1/3}} (|U_i| + |L_i|) = n - n^{1/3} s_1 - s_1 s_0 + s_1^2 + \sum_{i=1}^{n^{1/3}} (s_i - s_{i+1})^2.$$

By (A.3) we can write $n^{1/3} s_1$ as $s_0 s_1$; at the same time, we add and subtract $s_0^2 = n^{2/3}$ to get

$$\sum_{i=1}^{n^{1/3}} (|U_i| + |L_i|) = n - n^{2/3} + s_1^2 - 2s_1 s_0 + s_0^2 + \sum_{i=1}^{n^{1/3}} (s_i - s_{i+1})^2 = n - n^{2/3} + \sum_{i=0}^{n^{1/3}} (s_i - s_{i+1})^2,$$

as claimed.

The conclusion follows. $\square$

AUTOBIOGRAPHICAL STATEMENT

Jingyuan Zhang was born in Ningbo, China on July 8, 1964. He received his BS degree in Computer Science from Shandong University in 1984, and his MS degree in Computer Science from Zhejiang University in 1987.

From June 1987 to August 1989, he was an instructor with Department of Electrical Engineering and Computer Science at Ningbo University, China. Since September 1989, he has pursued his PhD Degree in Compute Science at Old Dominion University, Virginia. In August 1992, he joined the faculty of the Mathematics and Computer Science Department at Elizabeth City State University, where he is currently an assistant professor of computer science.

Jingyuan Zhang is a member of the Association of Computing Machinery and the IEEE Computer Society.

79