UNIVERSITY OF CENTRAL FLORIDA

COLLEGE OF ARTS AND SCIENCES

DISSERTATION APPROVAL

DATE: *May 20, 1994*

BASED ON THE CANDIDATE'S SUCCESSFUL ORAL DEFENSE, IT IS RECOMMENDED THAT THE DISSERTATION PREPARED BY *Amit Jain*

ENTITLED "*Multiselection and Multisearch: Parallel Algorithms*"

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF *Doctor of Philosophy*

FROM THE DEPARTMENT OF *Computer Science*

Narsingh Deo, Professor
Department of Computer Science

Ronald D. Dutton
Graduate Program Coordinator

Terry J. Frederick, Chair
Department of Computer Science

Ben B. Morgan, Jr., Associate Dean

Edward P. Sheridan, Dean

The material presented within this report does not necessarily reflect the opinion of the committee, the College of Arts and Sciences, or the University of Central Florida

UNIVERSITY OF CENTRAL FLORIDA

COLLEGE OF ARTS AND SCIENCES

DEFENSE OF DISSERTATION

THE UNDERSIGNED VERIFY THAT THE FINAL ORAL DEFENSE OF THE
DOCTOR OF PHILOSOPHY DISSERTATION OF *Amit Jain*
HAS BEEN SUCCESSFULLY COMPLETED ON *May 20, 1994*
TITLE OF DISSERTATION: "*Multiselection and Multisearch: Parallel Algorithms*"
MAJOR FIELD OF STUDY: *Computer Science*

COMMITTEE:

Narsingh Deo, Chair

Ronald D. Dutton

Charles E. Hughes

Brian E. Petrasko

Udaya B. Vemulapati

APPROVED:

Ronald D. Dutton Date
Graduate Program Coordinator

Ben B. Morgan, Jr. Date
Associate Dean

Terry J. Frederick, Chair, Date
Department of Computer Science

Edward P. Sheridan Date
Dean

The material presented within this report does not necessarily reflect the opinion of the committee, the College of Arts and Sciences, or the University of Central Florida

MULTISELECTION AND MULTISEARCH: PARALLEL ALGORITHMS

BY

AMIT JAIN
B.Tech. Indian Institute of Technology, New Delhi, 1987

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the graduate studies program
of the College of Arts and Sciences
University of Central Florida
Orlando, Florida

Summer Term
1995

Major Professor: Narsingh Deo

ABSTRACT

Parallel computing is the practice of employing several processors to solve a problem substantially faster than is possible on a conventional single-processor machine. There are many applications that can benefit from parallel computing. However, often the best sequential algorithms cannot routinely be converted to fast parallel algorithms that make efficient use of processors. Generally, the design of fast and efficient parallel algorithms requires techniques quite different from those used in the design of sequential algorithms. Another benefit of designing parallel algorithms is that it usually provides new insights into the problem that could lead to simpler or better sequential algorithms. This dissertation presents new parallel algorithms for performing multiple selections and multiple searches in sets. These problems arise in many applications. The model of parallel computation used is the Exclusive-Read Exclusive-Write (EREW) Parallel Random Access Machine (PRAM).

A fundamental problem, encountered in many applications, is merging two sorted arrays to produce a single sorted array. We will introduce a technique called *multiselection* that captures the "essence" of merging. An efficient solution to the multiselection problem allows us to partition the merging problem so as to obtain an optimal parallel algorithm for merging. The analysis of the number of comparisons, performed by the parallel algorithm, also suggests an optimal sequential algorithm for multiselection.

Next, we will use the partitioning as well as the chaining technique to design parallel algorithms for search and multisearch on $n \times n$ matrices with sorted rows

and sorted columns. The amount of work done in our parallel algorithms matches the best-known sequential time bounds. We also consider the more general problem of searching in an $m \times n$ matrix with only the columns sorted. We obtain optimal parallel algorithms that employ techniques quite different from those required for searches in matrices with both rows and columns sorted. The execution times of these algorithms have a non-trivial dependence on the rank of the required elements. Searching in both kinds of sorted matrices has applications in diverse areas.

Finally, our multiselection and merging algorithms are employed, together with the technique of tree contraction, to design a fast and efficient parallel algorithm, first of its kind, for computing the value of the minimum-cost flow in series-parallel networks. Our algorithm is not only optimal with respect to the best-known sequential algorithm but it also suggests a simpler sequential algorithm.

ACKNOWLEDGEMENTS

I would like to thank Narsingh Deo, my doctoral advisor, for his encouragement and patience. His insistence as well as willingness to spend many hours on improving my clarity in oral and written communication is much appreciated. I would also like to thank the members of my research committee: Ronald Dutton, Charles Hughes, Udaya Vemulapati and Brian Petrasko for their input and advice.

I am grateful for my involvement with the NSF project, on the introduction of parallel computing to undergraduates, for giving me a desirable taste of the life afterwards. In connection with the project, I would like to thank Ratan Guha, Terry Fredericks, Narsingh Deo, Charles Hughes, Amar Mukherjee, Udaya Vemulapati, Benjamin Goldfarb and lastly, but perhaps more important, all of the undergraduate students for their support.

It was a pleasure to do research with Murali. For brain storming and for his input, Chandra was always available. Udaya was always patient enough to answer even obscure questions about TEX. David was instrumental in rescuing me from the tentacles of INS. Thanks also to Sanjay for his "long distance" faith in my abilities. Jennie, Connie, Dolores and Betsye helped me in untangling the not so occasional bureaucratic knot. I must thank Ben, Don and Tony for going out of their way on many a occasion to provide the systems support that I often demanded.

And finally, this thesis would surely have been abandoned if it were not for the inspiration from Anita.

TABLE OF CONTENTS

LIST OF TABLES viii

LIST OF FIGURES ix

1 INTRODUCTION **1**

1.1 Motivation 1

1.2 Preliminaries 2

1.2.1 Parallel Random Access Machine 2

1.2.2 Presentation and Performance of Parallel Algorithms 4

1.3 Outline 6

2 PARALLEL MULTISELECTION AND MERGING **8**

2.1 Introduction 8

2.2 Selection in Two Sorted Arrays 9

2.3 Parallel Multiselection 12

2.4 Analysis of the Multiselection Algorithm 16

2.5 Parallel Merging 20

2.5.1 Previous Results 20

2.5.2 Parallel Merging using Multiselection 22

2.6 Summary 25

3 PARALLEL SEARCH AND MULTISEARCH IN SORTED MATRICES **27**

3.1 Introduction 27

3.2 Search in Matrices with Sorted Rows and Columns 28
3.2.1 Related Results 29
3.2.2 Parallel Search in Square Sorted Matrices 30
3.3 Multisearch in Matrices with Sorted Rows and Columns 34
3.3.1 A Simple Pipelined Algorithm for Multisearch 34
3.3.2 The Main Parallel Algorithm 35
3.4 Multisearch in X+Y 40
3.5 Search in Matrices with Sorted Columns 42
3.6 Multisearch in Matrices with Sorted Columns 48
3.6.1 Sequential Algorithm 49
3.6.2 Parallel Algorithm 52
3.7 Faster Parallel Search in Matrices with Sorted Columns 54
3.8 Summary 57

4 PARALLEL MIN-COST FLOW 59

4.1 Introduction 59
4.1.1 Previous Work 60
4.1.2 Definitions of Max-Flow and Min-Cost Max-Flow Problems . . 62
4.2 Min-Cost Flow on Series-Parallel Networks 64
4.2.1 Flow Feasibility 64
4.2.2 Min-Cost Flow Value and Flow List Computations 66
4.2.3 Obtaining Flow Lists using Multiselection and Merging 75
4.3 Summary 80

5 CONCLUSION AND OPEN PROBLEMS 82

5.1 Conclusions 82

5.2 Open Problems 86

REFERENCES **88**

LIST OF TABLES

3.1 Various cases for the sequential time-complexity of the multisearch algorithm 52

LIST OF FIGURES

2.1 Reduction of selection to median finding. 10

2.2 An example to illustrate the parallel merging algorithm. In Step 1, the 4th, 8th, 12th, and 16th smallest elements are selected. Step 2 completes the partitioning and the corresponding sections (shown by braces) of the two arrays A and B can then be merged independently. 23

3.1 Sequential algorithm for search in a sorted matrix M. 31

3.2 Sequential algorithm for ranking in a sorted matrix M. 31

3.3 Parallel algorithm for search in a sorted matrix. 33

3.4 Paths followed by two searches in a sorted matrix. 35

4.1 A series-parallel network and its decomposition tree. The edges are labeled by 3-tuples $(l(e), u(e), c(e))$. 63

4.2 SHUNT on a parallel composition node for computing the max-flow. . 67

4.3 SHUNT on a series composition node for computing the max-flow. . . 67

4.4 Flow lists for the series-parallel network in Figure 4 (special pairs are shown in square brackets). 71

4.5 SHUNT on a parallel composition node for computing the min-cost flow. 73

4.6 SHUNT on a series composition node for computing the min-cost flow. 74

Chapter 1

INTRODUCTION

1.1 Motivation

Parallel computation is the practice of employing multiple processors to solve a problem substantially faster than is possible on a conventional single-processor machine. Why do we need parallel computation? Because of fundamental physical limitations, imposed by speed of light and device sizes, the processing speed of a single processor machine is not expected to keep on improving at the same rate. Nevertheless, the need for faster solutions to ever larger problems is growing rapidly. A viable alternative is the exploitation of parallelism.

Just as the fastest cycle times are approaching the fundamental physical barriers, we have new generations of parallel machines appearing with more mature technology. Very large scale integrated (VLSI) circuit technology has advanced to the point that is has become feasible to build systems with thousands of processors at a reasonable cost. Most high-performance computation is already being targeted principally at the exploitation of parallelism. There are many important applications that can benefit from parallelism. But how can one design a fast parallel algorithm for a specific application without having some algorithmic paradigms that can be followed? Unfortunately, it turns out that often the best sequential algorithms cannot routinely be

converted into fast parallel algorithms. Generally, the design of fast and efficient parallel algorithms requires new techniques quite different from the design of sequential algorithms. Another benefit of designing parallel algorithms is that it usually provides new insights into the problems that could lead to simpler or better sequential algorithms.

In this dissertation, we will design parallel algorithms to perform multiple selections (multiselection) and multiple searches (multisearch) in structured sets that arise in many applications. We will also see that multiselection and multisearch are good building blocks for solving other important problems. In the process of designing parallel algorithms for these problems we will see the use of several general techniques. We shall also have a chance to investigate if parallel algorithms can suggest simple or even better sequential algorithms for the problems under consideration.

In the next section we will define the model of parallel computation that will be used as well as mention some important issues in the presentation and the performance of parallel algorithms. In the last section an outline of the dissertation is given.

1.2 Preliminaries

1.2.1 Parallel Random Access Machine

The Parallel Random Access Machine (PRAM) is a natural generalization of the sequential model of computation. The PRAM consists of several processors and a shared memory that each processor can access in unit time. Each processor is uniquely identified by an index, called a *processor number* or a *processor id*. Thus,

a PRAM with p processors has processors numbered $1, 2, \ldots, p$. All the processors operate synchronously under the control of a common clock. The PRAM algorithms that we will present are of the type *single instruction multiple data (SIMD)*. That is, all processors execute the same program and in a single time unit all active processors execute the same instruction, but with different data in general.

There are several variations of the PRAM model based on the assumption regarding the handling of the simultaneous access of several processors to the same location of the shared memory. The *exclusive-read exclusive-write* (EREW) PRAM does not allow any simultaneous access to a single memory location. The *concurrent-read exclusive-write* (CREW) PRAM allows simultaneous access for a read instruction only. In the *concurrent-read concurrent-write* (CRCW) PRAM simultaneous access to a location is allowed for both read and write instructions. There are different variations of the CRCW PRAM depending on how concurrent writes are handled. The CREW PRAM is more powerful than the EREW PRAM and the CRCW PRAM is the most powerful. For more details, see the book by JáJá [41], or the survey by Karp and Ramachandran [44] and the references found there.

We will use the EREW PRAM model for all of the algorithms presented in this dissertation. The EREW PRAM model is the least powerful PRAM model and, arguably, the closest to real parallel computers. The PRAM model has gained widespread acceptance for the design of parallel algorithms. The issue of modeling parallel computation, however, is by no means a closed topic. For more discussion see JáJá [41], Sanz [54], Valiant [60], and Vishkin [62].

1.2.2 Presentation and Performance of Parallel Algorithms

In the design of parallel algorithms, progress during the last decade has redirected the research focus from an effort to classify problems that can be solved fast, that is, in $O(\log^k n)$ time on n^l processors, where l and k are constants (NC algorithms, NC stands for Nick's Class), to a growing body of research on how to design algorithms that run fast but use processors efficiently. To discuss this issue, we need to first define a few concepts.

Let P be a given problem and n be its input size. Denote the sequential complexity, if known, of P by $T^*(n)$. Otherwise, let $T^*(n)$ be the worst-case time bound of the best known sequential algorithm. Suppose we have a parallel algorithm that runs in $T(n)$ time using $P(n)$ processors. The time-processor product $C(n) = T(n)P(n)$ represents the *cost* of the parallel algorithm. The parallel algorithm can be converted into a sequential algorithm that runs in $O(C(n))$ time. If we have p processors, where $p \leq P(n)$, then we can have each processor simulate the $P(n)$ processors in $O(P(n)/p)$ substeps. Overall the simulation takes $O(C(n)/p)$ time.

A parallel algorithm is said to be *cost-optimal* if the total cost of the parallel algorithm $C(n)$ equals $T^*(n)$. A parallel algorithm is said to be *time-optimal* if the running time $T(n)$ can be shown to be the fastest possible for the problem on the model under consideration, usually by a matching lower bound on that particular model. A parallel algorithm is said to be *optimal* if it is be both *cost-optimal* and *time-optimal.*

An alternative way to describe PRAM algorithms is the Work-Time paradigm, which is described in the book by JáJá [41]. The Work-Time paradigm provides

informal guidelines for a two level top-down description of parallel algorithms. At the upper level, we describe the algorithm in terms of a sequence of time units, where each time unit may include any number of concurrent operations. Suppose that the upper level description results in a parallel algorithm that runs in $T(n)$ time units while performing a total of $W(n)$ operations, where $W(n)$ is called the *work* performed by the parallel algorithm. Using the general Work-Time scheduling principle (also known as Brent's scheduling [17]), we can almost always adapt this algorithm to run on a p-processor PRAM in $\leq \lfloor W(n)/p \rfloor + T(n)$ parallel steps. The success of this principle depends on being able to calculate the number of operations performed during each time unit and the allocation of each processor to the appropriate tasks to be performed by that processor. For most problems, the allocation is straightforward, but for some problems it may not be obvious.

Suppose we have a parallel algorithm that runs in time $T(n)$ and uses a total of $W(n)$ operations. Then using the Work-Time scheduling principle this algorithm can be simulated on a p-processor PRAM in $O(W(n)/p + T(n))$ time and the parallel algorithm is optimal if $W(n) = T^*(n)$ and $T(n)$ is as small as possible. The corresponding cost is $O(W(n) + pT(n))$ and the previous definition of cost-optimality still holds. Note that $W(n) \leq C(n)$ for any number of processors. The Work-Time scheduling principle allows a succinct presentation of parallel algorithms that emphasizes structural parallelism in the problem.

1.3 Outline

The remainder of the dissertation is organized as follows.

In the next chapter, we first consider the problem of multiselection in two sorted arrays and propose an efficient parallel algorithm on the EREW PRAM model. The multiselection algorithm, based on a novel application of the technique of chaining, is used as a building block for solving other problems. Next, we consider the problem of merging two sorted lists, a fundamental problem in sequential and parallel computing. We propose an optimal parallel algorithm for merging using our multiselection algorithm for partitioning the merging problem.

In Chapter 3, we look at the problem of search in sets that possess certain structure. In particular, we are interested in matrices with sorted rows and sorted columns and matrices with sorted columns only. Searching in such matrices has received considerable attention due to their application in statistics, operations research and combinatorics, among others. We first consider search and multisearch in matrices with sorted rows and sorted columns, and Cartesian sets of the form $X + Y$ where X and Y are sorted arrays of size n, a special case of matrices with sorted rows and sorted columns. Next we consider the more general case of multiple sorted arrays or matrices with sorted columns. The algorithms for search and multisearch in matrices with sorted rows and sorted columns use the techniques of partitioning and chaining while for matrices with sorted columns we use the technique of accelerated cascading and chaining.

In Chapter 4, the problem of finding the minimum cost of a feasible flow in directed series-parallel networks with real-valued lower and upper bounds for the flows on edges

is addressed. The technique of tree contraction is combined with our multiselection and merging algorithms to tackle the min-cost flow problem in parallel. Although there have been some algorithms for the max-flow problem on restricted classes of networks, there are no known efficient NC algorithms for the min-cost flow problem. We will present one such algorithm in Chapter 4, which is optimal with respect to the best known sequential algorithm.

The final chapter contains the conclusions and some open problems arising from this dissertation.

Chapter 2

PARALLEL MULTISELECTION AND MERGING

2.1 Introduction

A fundamental problem that is used in many applications is the merging of two sorted arrays to produce a single sorted array. There is a simple sequential algorithm for merging two sorted arrays, which takes linear time in the length of the two arrays. We are interested in merging substantially faster by using several processors together on the EREW PRAM. However, designing a fast parallel algorithm for merging is not at all obvious. We will introduce a technique of *multiselection* that captures the "essence" of merging. A fast solution to the multiselection problem gives us an optimal solution to merging. Multiselection and the techniques involved in its solution are also useful for solving other problems in parallel. Interestingly, the parallel algorithm for multiselection also suggests an optimal sequential algorithm for multiselection.

Formally, the problem of selection in two sorted arrays can be stated as follows. Given two ordered multisets A and B of sizes m and n, where $m \leq n$, the problem is to select jth smallest element in A and B. The problem can be solved sequentially in $O(\log(min\{j, m\}))$ time without explicitly merging A and B [26, 32]. Finding the median of A and B is a well-known special case of this problem. Multiselection, a generalization of selection, is the problem where given a sequence of r integers

$1 \leq K_1 < K_2 < \ldots < K_r \leq (m+n)$, all the K_ith, $1 \leq i \leq r$, smallest elements in A and B are to be found. In the rest of the chapter we use the term selection, for convenience, to mean selection in two sorted arrays. Without loss of generality, we also assume that the two arrays A and B contain no repeated elements.

First, we present a new sequential algorithm for selection in two sorted arrays and then use it to develop a parallel algorithm for multiselection. The algorithm uses r processors, on the EREW PRAM, to perform multiselection for the set of queries $\{K_1, \ldots, K_r\}$ in $O(\log m + \log r)^\dagger$ time, where we assume that $m \leq n$. Next, we analyze the number of comparisons performed by the parallel algorithm for multiselection.

We will show that an efficient solution for multiselection leads to an elegant parallel merging algorithm which is optimal in time and cost. The merging problem is introduced in Section 2.5.1, where we also relate our work to previous results on merging. The merging algorithm itself is presented and analyzed in Section 2.5.2. The chapter ends with a summary.

2.2 Selection in Two Sorted Arrays

The median of $2r$ elements is defined to be the rth smallest element (the left median), while that of $2r+1$ elements is defined to be the $(r+1)$th element. Finding the jth smallest element can be reduced to selecting the median of the appropriate subarrays of A and B as follows: When $1 \leq j \leq m$ and the arrays are in nondecreasing order, the required element can only lie in the subarrays $A[1..j]$ and $B[1..j]$. Thus,

†For clarity in presentation, we use $\log x$ to mean $\max\{1, \log_2 x\}$.

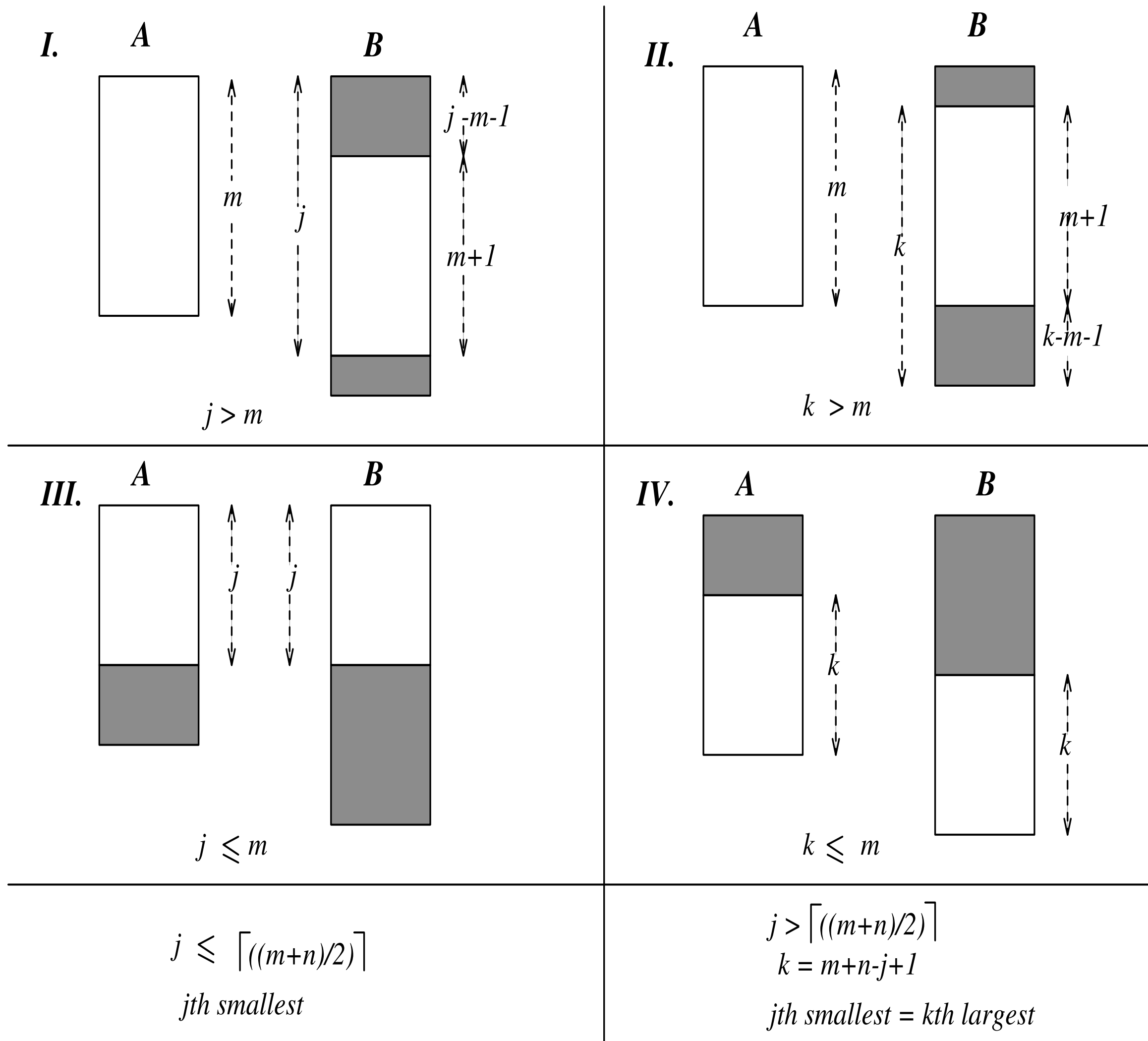

Figure 2.1. Reduction of selection to median finding.

the median of the $2j$ elements in these subarrays is the jth smallest element. This reduction is depicted as Case III in Figure 2.1. On the other hand, when $m < j \leq \lceil (m+n)/2 \rceil$, the jth selection can be reduced to finding the median of the subarrays $A[1..m]$ and $B[(j-m)..j]$, which is shown as Case I in Figure 2.1. When $j > \lceil (m+n)/2 \rceil$, we can view the problem as that of finding the kth largest element, where $k = m + n - j + 1$. This gives rise to Cases II and IV which are symmetric to Cases I and III, respectively, in Figure 2.1. For Cases II and IV, the median of $2r$ elements is defined to be the $(r+1)$st element, which is the right median. From now on, these subarrays will be referred to as *windows*.

The median can be found by comparing the individual median elements of the current windows and suitably truncating the windows to half, until the window in A has no more than one element. The middle elements of the windows will be referred to as *probes*. A formal description of this median-finding algorithm follows.

```
procedure select_median(A[lowA,...,highA], B[lowB,...,highB])
{       A[p] < A[p+1], lowA ≤ p ≤ highA, A[m+1] = ∞
        B[q] < B[q+1], lowB ≤ q ≤ highB, B[n+1] = ∞
        highA − lowA ≤ highB − lowB ≤ highA − lowA + 1
        [lowA, highA], [lowB, highB]: current windows in A and B
        probeA, probeB : next position to be examined in A and B }
1.      while (highA > lowA)
2.              probeA ← ⌊ (lowA + highA)/2 ⌋; sizeA ← (highA − lowA + 1)
3.              probeB ← ⌊ (lowB + highB)/2 ⌋; sizeB ← (highB − lowB + 1)
4.              case  (A[probeA] < B[probeB]) :
5.                              lowA ← probeA; highB ← probeB + 1
6.                              if (sizeA = sizeB) and (sizeA is odd)   highB ← probeB
7.                      (A[probeA] > B[probeB]) :
8.                              highA ← probeA; lowB ← probeB
9.                              if (sizeA = sizeB) and (sizeA is even)  highA ← probeA + 1
10.             endcase
11.     endwhile
12.     merge the remaining (at most 3) elements from A + B and return their median
endprocedure
```

When the procedure *select_median* is invoked, there are two possibilities: (i) the size of the window in A is one less than that of the window in B (ii) or the sizes of the windows are equal. Furthermore, considering whether the size of the window in A is odd or even, the reader can verify (examining Steps 4 through 9) that an equal number of elements are being discarded from above and below the median. Hence, the scope of the search is narrowed to at most three elements (1 in A and at most 2 in B) in the two arrays; the median can then be determined easily in Step 12, which will be denoted as the postprocessing phase. The total time required for selecting the jth smallest element is $O(\log(\min\{j, m\}))$. With this approach, r different selections, $\{K_1, \ldots, K_r\}$, in A and B can be performed in $O(r \log m)$ time. Note that the information-theoretic lower bound for the problem of multiselection is $\binom{m+r}{r}$ which turns out to be $O(r \log m/r)$ when $r \leq m$ and $O(m \log r/m)$ when $r > m$. A parallel algorithm for r different selections based on the above sequential algorithm is presented next.

2.3 Parallel Multiselection

Let the selection positions be $(K_1, K_2, \ldots, K_r)$, where $1 \leq K_1 < K_2 < \ldots < K_r \leq (m+n)$. Our parallel algorithm employs r processors with the ith processor assigned to finding the K_ith element, $1 \leq i \leq r$. The distinctness and the ordered nature of the K_is are not significant restrictions on the general problem. If there are duplicate K_is or if the selection positions are unsorted, both can be remedied in $O(\log r)$ time using r processors [21]. On a CREW PRAM this problem admits a trivial solution, as each processor can carry out the selection independently. On an EREW PRAM, however,

this problem becomes interesting because the read conflicts have to be avoided. In the following, we will outline how multiselections can be viewed as multiple searches in a search tree. Hence, we can exploit the well-known technique of *chaining* introduced by Paul, Vishkin and Wagener [53].

Let us first consider only those K_is that fall in the range $[m+1..\lceil(m+n)/2\rceil]$, that is, those for which Case I, in Figure 2.1, holds. All of these selections initially share the same probe in array A. Let $m < K_l < K_{l+1} < \ldots < K_h \leq \lceil(m+n)/2\rceil$ be a sequence of K_is that share the same probe in A. Following the terminology of Paul, Vishkin and Wagener [53], we refer to such a sequence of selections as a *chain*. Note that these selections will have different probes in array B. Let the common probe in array A be x for this chain, and the corresponding probes in array B be $y_l < y_{l+1} < \ldots < y_h$. The processor associated with K_lth selection will be active for the chain. This processor compares x with y_l and y_h, and based on these comparisons the following actions take place:

- $x < y_l$: The chain stays intact.
- $x > y_l$
    - ⋆ $x < y_h$: The chain is split into two subchains.
    - ⋆ $x > y_h$: The chain stays intact.

Note that at most two comparisons are required to determine if the chain stays intact or has to be split. When the chain stays intact, the window in array A remains common for the whole chain. Processor P_l computes the size of the new common window in array A. The new windows in the array B can be different for the selections

in the chain, but they all shrink by the same amount, and hence the size of the new window in B and the offset from the initial window in B are the same for all the selections. The two comparisons made by the active processor determine the windows for all the selections in the chain (when the chain stays intact). The chain becomes inactive when it is within 7 elements to compute the required median for all the selections in the chain. The chain does not participate in the algorithm any more, except for the postprocessing phase.

When a chain splits, processor P_l remains in charge of the chain $K_l, \ldots, K_{\lceil (l+h)/2 \rceil - 1}$ and activates processor $p_{\lceil (l+h)/2 \rceil}$ to handle the chain $K_{\lceil (l+h)/2 \rceil}, \ldots, K_h$. It also passes the position and value of the current probe in array A, the offsets for the array B, and the parameter h. During the same stage, both these processors again check to find whether their respective chains remain intact. If the chains remain intact they move on to a new probe position. Thus, only those chains that do not remain intact stay at their current probe positions to be processed in the next stage. It can be shown that at most one chain remains at a probe position after any stage. Moreover, there can be at most two new chains arriving at a probe position from the previous stages. The argument is the same as the one used in the proof of Claim 1 in Paul, Vishkin and Wagener [53]. All of this processing within a stage can be performed in $O(1)$ time on an EREW PRAM, as at most three processors may have to read a probe. One of the chains always comes from the left and the other comes from the right while the chain that stays at a probe is always in the center. When a chain splits into two, their windows in A will overlap only at the probe that splits them. Any possible read conflicts at this common element can happen only during the postprocessing

phase (which can be handled as described in the next paragraph). Hence, all of the processing can be performed without any read conflicts.

At each stage a chain is either split into two halves or its window size is halved. Hence after at most $O(\log m + \log r)$ stages each selection process must be within seven elements of the required position. At this point, each processor has a window of size at most 3 in A and 4 in B. If the windows of different selections have any elements in common, the values can be broadcasted in $O(\log r)$ time, such that each processor can then carry out the required postprocessing in $O(1)$ time. However, we may need to sort the indices of the elements in the final windows in order to schedule the processors for broadcasting. But this requires only integer sorting as we have r integers in the range $1 \ldots n$ which can surely be done in $O(\log r)$ time [21]. Thus, the total amount of data copied is only $O(r)$.

Of the remaining selections, those K_is falling in Case II can be handled in exactly the same way as the ones in Case I. The chaining concept can be used only if $O(1)$ comparisons can determine the processing for the whole chain. In Cases III and IV, different selections have windows of different sizes in both the arrays. Hence, chaining cannot be directly used as in Cases I and II. However, we can reduce Case III (IV) to I (II). To accomplish this reduction, imagine array B to be padded with m elements of value $-\infty$ in locations $-m+1$ to 0 and with m elements of value ∞ in locations $n+1$ to $n+m$. Let this array be denoted as C (which need not be explicitly constructed). Selecting the jth smallest element, $1 \leq j \leq m+n$, in $A[1..m]$ and $B[1..n]$ is equivalent

to selecting the $(j+m)$th element, $m+1 \leq (j+m) \leq 2m+n$, in the arrays $A[1..m]$ and $C[1..2m+n]$. Thus, selections in Case III (IV) in the arrays A and B become selections in Case I (II) in the arrays A and C.

For clarity of presentation, we have increased the window sizes in Case III (IV) to m in array A and $m+1$ in C. For efficiency, it is better to truncate array A and B depending upon the maximum (minimum) value of a selection in Case III (IV) before using the reduction. If all selection positions fall either in interval $[1..m]$ (Case III) or $[n+1..n+m]$ (Case IV) then r selections can be performed by r processors in time

$$O(\log(\max\{\max_{K_i \in [1...m]} K_i,\ \ m+1-\min_{K_j - n \in [1...m]}\{K_j - n\}\})).$$

Any selection in the interval $[m..n]$ dominates the time complexity. In such a case, we note that all of the selections in different cases can be handled by one chain with the appropriate reductions. Hence we have the following result.

Theorem 2.3.1 *Given r selection positions $\{K_1, \ldots, K_r\}$, all of the selections can be made in $O(\log m + \log r)$ time using r processors on the EREW PRAM.*

2.4 Analysis of the Multiselection Algorithm

We want to analyze the number of comparisons required, in the worst-case, in the parallel multiselection algorithm, since the number of comparisons dominates other operations used in the algorithm. The estimation of the number of comparisons is somewhat involved. First we prove the following lemma.

Lemma 2.4.1 *Suppose we have a chain of size r, $r \geq 2$. The worst-case number of comparisons required to process the chain completely is greater if the chain splits at the current probe than if it stays intact.*

Proof: We can envisage the multiselection algorithm as a specialized search in a binary tree with height $O(\log m)$. Let $T(r, l)$ be the total number of comparisons needed, in the worst case, to process a chain of size r, which is at a probe corresponding to a node at height l in the search tree. We proceed by induction on the height of the node. The base case, when the chain is at a node of height 1, can be easily verified. Suppose the lemma holds for all nodes at height $\leq l-1$. Consider a chain of size r at a node of height l. If the chain stays intact and moves down to a node of height $l-1$ then

$$T(r, l) = T(r, l-1) + 2, \tag{2.1}$$

since at most two comparisons are required to process a chain at a node. If the chain splits, one chain of size $\lceil r/2 \rceil$ stays at height l (in the worst-case) and the other chain of size $\lfloor r/2 \rfloor$ moves down to height $l-1$. Then the worst-case number of comparisons is,

$$T(r, l) = T(\lceil r/2 \rceil, l) + T(\lfloor r/2 \rfloor, l-1) + 4. \tag{2.2}$$

By the hypothesis and Eq. (2.2)

$$T(r, l-1) \leq T(\lceil r/2 \rceil, l-1) + T(\lfloor r/2 \rfloor, l-2) + 4. \tag{2.3}$$

Thus, when the chain stays intact, we can combine Eq.s (2.1) and (2.3) to obtain

$$T(r, l) \leq T(\lceil r/2 \rceil, l-1) + T(\lfloor r/2 \rfloor, l-2) + 6.$$

Again, by hypothesis $T(\lfloor r/2 \rfloor, l-1) \geq T(\lfloor r/2 \rfloor, l-2) + 2$ and we require at least one comparison for a chain to move down a level. Hence $T(\lceil r/2 \rceil, l) \geq T(\lceil r/2 \rceil, l-1) + 1$ and the lemma holds. □

To determine the number of comparisons required in the multiselection algorithm, we consider two cases. In the first case, when $r \leq m$, we have the following.

Lemma 2.4.2 *In the worst case, the total number of comparisons required by the parallel multiselection algorithm for r selections is* $O(r(1 + \log(m/r)))$, *if* $r \leq m$.

Proof: The size of the initial chain is r. Lemma 2.4.1 implies that the chain must split at every opportunity for the worst-case number of comparisons. Thus, at height i the maximum size of a chain is $r/2^i, 0 \leq i \leq \lfloor \log r \rfloor$. The maximum number of chains possible is r (with each containing only one element), which could be spread over the first $\lceil \log r \rceil$ levels. From a node at height i, the maximum number of comparisons a search for an element can take is $(\lceil \log m \rceil - i)$ (for this chain at this node). Hence the number of comparisons after the chains have split is bounded by

$$\sum_{i=0}^{j} 2(2^i)(\lceil \log m \rceil - i), \quad j = \lfloor \log r \rfloor$$

which is $O(r \log m/r)$. We also need to count the number of comparisons required for the initial chain to split up into r chains, and fill up $\lceil \log r \rceil$ levels. Since the size of a chain at a node of height i is at most $r/2^i$, the maximum number of splits possible is $(\lceil \log r \rceil - i)$. Also, recall that a chain requires four comparisons for each split in the worst-case. Thus, the number of comparisons is bounded by:

$$4 \sum_{i=0}^{j} 2(2^i)(\lceil \log r \rceil - i), \quad j = \lfloor \log r \rfloor$$

since there can be at most $2(2^i)$ chains at level i. Thus, the number of comparisons for splitting is $O(r)$. □

Consider the second case, when $r > m$.

Lemma 2.4.3 *If $r > m$, the parallel multiselection algorithm performs r selections in $O(m \log(r/m))$ comparisons.*

Proof: Using Lemma 2.4.1 and arguments similar to the ones in the proof of previous lemma, we know that the maximum size of a chain is $r/2^i$ at height i. This chain can split at most $(\lceil \log r \rceil - i)$ times. Hence the number of comparisons needed for splitting is bounded by

$$4 \sum_{i=0}^{\lfloor \log m \rfloor} 2(2^i)(\lceil \log r \rceil - i)$$

which is $O(m \log(r/m))$. After the chains have split, there may be at most $O(m)$ chains remaining. The number of comparisons required is then bounded by

$$\sum_{i=0}^{\lfloor \log m \rfloor} 2(2^i)(\lceil \log m \rceil - i) \quad = O(m).$$

□

Note that in the preceding analysis, we need not consider the integer sorting used in the post-processing phase of the multiselection algorithm as it does not involve any key comparisons. The sequential complexity of our multiselection algorithm matches the information-theoretic lower bound for the multiselection problem. The number of operations performed by our parallel multiselection algorithm also matches the lower bound if we have an optimal integer sorting algorithm for the EREW PRAM.

2.5 Parallel Merging

2.5.1 Previous Results

Consider the problem of merging two sorted sequences A and B of length m and n, respectively, where $m \leq n$ into one sorted sequence on an EREW PRAM. Merging is a fundamental non-numeric operation which frequently arises in many diverse applications. Our approach to parallel merging consists of identifying elements in A and B which would have appropriate rank in the merged array. These elements partition the arrays A and B into equal-size subproblems which then can be assigned to each processor for sequential merging. We use our multiselection algorithm, presented in Section 2.3, for selecting the desired elements, which leads to a simple and optimal parallel algorithm for merging on the EREW PRAM. Thus, our technique differs from those of other optimal parallel algorithms for merging on the EREW PRAM where the subarrays are defined by elements at fixed positions in A and B.

First, we note that Snir's results [57] imply that regardless of the number of processors available, $\Omega(\log(m+n))$ time is required to merge two sequences on the EREW PRAM. On the CREW PRAM merging can be performed in $O(1)$ time using mn processors [56]. If the number of processors is restricted to $O(n \log^c n)$, for any fixed c, then $\Omega(\log\log(m+n))$ is a lower bound on the time [16]. In the following discussion, we limit ourselves to algorithms for merging on the EREW PRAM.

Optimal parallel algorithms for merging in [7, 13, 38, 61] use different techniques to essentially overcome the difficulty of multiselection. Bilardi and Nicolau [13] gave the first optimal algorithm for merging on the EREW PRAM that runs in $O(\log(m+n))$

time using $((m+n)/\log(m+n))$ processors. The number of comparisons in their algorithm is within a factor of 2 of the minimum that can be achieved by any algorithm, even a sequential one. Hagerup and Rüb [38] (as well as Anderson, Mayr and Warmuth [7]) have later presented another optimal algorithm on the EREW PRAM. Their algorithm is based on the CREW PRAM merging algorithm developed by Shiloach and Vishkin [56]. In order to reduce the number of comparisons as compared to Bilardi and Nicolau's algorithm, their algorithm recursively calls itself once and then uses Batcher's merging [10] for merging two sub-sequences for partitioning. Also, in order to avoid read conflicts, parts of the sequences are copied by some processors.

We show that the number of comparisons in our merging algorithm matches those of Hagerup and Rüb's algorithm [38] and is within lower-order terms of the minimum possible, even by a sequential merging algorithm. Moreover, our merging algorithm uses fewer comparisons when the two arrays differ in size significantly.

Finally, we note that Akl and Santoro [5] and Deo and Sarkar [26] have used selection as a building block in parallel merging algorithms. Their parallel merging algorithms employ either sequential median or sequential selection algorithm. Even though these algorithms are cost-optimal, their time complexity is $O(\log^2(m+n))$ on the EREW PRAM.

2.5.2 Parallel Merging using Multiselection

In this section, we give an formal description of our merging algorithm.

1. Find the $i\lfloor\log(m+n)\rfloor$, $i = 1, 2, \ldots, j-1$(where $j = \lceil(m+n)/\lfloor\log(m+n)\rfloor\rceil$), ranked element using multiselection. Let the output be two arrays $R_A[1..j]$ and $R_B[1..j]$, where $R_A[i] \neq 0$ implies that $A[R_A[i]]$ is the $(i\lfloor\log(m+n)\rfloor)$th element and $R_B[i] \neq 0$ implies that $B[R_B[i]]$ is the $(i\lfloor\log(m+n)\rfloor)$th element.

2. Let $R_A[0] = R_B[0] = 0$, $R_A[j] = m, R_B[j] = n$

for $i = 1, \ldots, j-1$ do

if $R_A[i] = 0$ then $R_A[i] = i * \lfloor\log(m+n)\rfloor - R_B[i]$

else $R_B[i] = i * \lfloor\log(m+n)\rfloor - R_A[i]$

3. for $i = 1, \ldots, j$ do

merge $(A[R_A[i-1]+1]..A[R_A[i]])$ with $(B[R_B[i-1]+1]..B[R_B[i]])$.

The merging algorithm is illustrated with an example in Figure 2.2. Steps 1 and 3 both take $O(\log(m+n))$ time using $(m+n)/\log(m+n)$ processors. Step 2 takes $O(1)$ time using $(m+n)/\log(m+n)$ processors. Thus the entire algorithm takes $O(\log(m+n))$ time using $(m+n)/\log(m+n)$ processors, which is optimal. The total amount of data copied in Step 1 is $O((m+n)/\log(m+n))$, since $r = (m+n)/\log(m+n)$, and compares favorably with $O(m+n)$ data copying required by Hagerup and Rüb's [38] merging algorithm. If fewer processors, say p, are available, the proposed parallel merging algorithm can be adapted to perform $p-1$ multiselections and will require a total of $O((m+n)/p + \log m + \log p))$ time.

A $m = 9$

1	2	3	4	5	6	7	8	9
3	5	7	14	17	19	3	27	29

B $n = 12$

1	2	3	4	5	6	7	8	9	10	11	12
2	4	6	12	15	16	18	20	24	28	30	31

$p = 5 \quad (m+n) = 21 \qquad \lfloor (m+n)/p \rfloor = 4$

	after Step 1		after Step 2
R_A	2 4 0 0 9	$\longrightarrow$	2 4 5 7 9
R_B	0 0 7 9 12	$\longrightarrow$	2 4 7 9 12

Figure 2.2.: An example to illustrate the parallel merging algorithm. In Step 1, the 4th, 8th, 12th, and 16th smallest elements are selected. Step 2 completes the partitioning and the corresponding sections (shown by braces) of the two arrays A and B can then be merged independently.

We are also interested in the number of comparisons required. Step 2 does not require any comparisons and Step 3 requires less than $m + n$ comparisons. The estimation of the number of comparisons in Step 1, the multiselection, is somewhat more involved.

To analyze the number of comparisons in the multiselection algorithm, we consider two cases. In the first case, when $r \leq m$, where $r = (m+n)/\log(m+n)$, we have the following result from Section 2.4.

Lemma 2.5.1 *In the worst case, the total number of comparisons required by the parallel multiselection algorithm for* r *selections is* $O(r(1 + \log(m/r)))$*, if* $r \leq m$*.*

In particular, Lemma 2.5.1 implies that, when $m = \theta(n)$, the total number of comparisons for the merging algorithm is $(m+n)+O(n \log\log n/\log n)$. This matches the number of comparisons in the parallel algorithm by Hagerup and Rüb [38]. When one of the list is smaller than the other, however, we can show better results. Consider the second case when $r > m$, where $r = (m+n)/\log(m+n)$, then we have the following result from Section 2.4.

Lemma 2.5.2 *If* $r > m$*, the parallel multiselection algorithm performs* r *selections in* $O(m \log(r/m))$ *comparisons.*

Hence, if $(m+n)/\log(m+n) > m$, or $m < n/\log n$ approximately, then our merging algorithm requires only

$$(m+n) + O\left(m \log \frac{m+n}{m\log(m+n)}\right)$$

comparisons, which is asymptotically better than that of Hagerup and Rüb's parallel algorithm [38]. Note that in the preceding analysis, we need not consider the integer sorting used in the post-processing phase of the multiselection algorithm as it does not involve any key comparisons.

2.6 Summary

In this chapter, we presented an efficient parallel algorithm for r multiselections in two sorted arrays A and B of sizes m and n respectively. The algorithm requires $O(\log m + \log r)$ time using r processors on the EREW PRAM and is based upon the technique of chaining. Furthermore, we analyzed the number of comparisons required by the multiselection algorithm and found it to be $O(r(1 + \log(m/r)))$ if $r \leq m$ and $O(m(1 + \log(r/m))$ if $r > m$, which matches the information-theoretic lower bound on the multiselection problem. As a result, running the parallel algorithm sequentially provides us with an optimal sequential algorithm for multiselection. An optimal sequential algorithm for multiselection algorithm is not easily obtained using standard techniques for designing sequential algorithms. The number of operations performed by our parallel multiselection algorithm also matches the lower bound if we have an optimal integer sorting algorithm for the EREW PRAM.

As an application of multiselection we consider the problem of merging two sorted arrays A and B on an EREW PRAM. The solution to the multiselection problem provides us with an appropriate partitioning and leads to a simple and optimal algorithm for merging in parallel using a novel approach. Our parallel merging algorithm requires $O(\log(m + n))$ time and $O(m + n)$ cost on the EREW PRAM. We fur-

ther show that the number of comparisons in our merging algorithm matches that of Hagerup and Rüb's parallel merging algorithm [38] and is within lower-order terms of the minimum possible, even by a sequential merging algorithm. Moreover, our merging algorithm uses fewer comparisons when the two given arrays differ in size significantly, that is, when $m < n/\log n$ approximately. Furthermore, our merging algorithm does not use data copying to avoid read conflicts unlike some of the earlier parallel merging algorithms.

Chapter 3

PARALLEL SEARCH AND MULTISEARCH IN SORTED MATRICES

3.1 Introduction

Sets that can be represented as matrices with certain structure arise in many applications. In particular, we are interested in performing search and multisearch in such sets. As instances of these sets, we consider matrices with sorted rows and sorted columns, matrices with sorted columns as well as vector representation of matrices with sorted rows and sorted columns. By vector representation, we imply Cartesian sets of the form $X+Y$ where the elements are $x_i + y_j$, where $x_i \in X, 1 \leq i \leq |X|$ and $y_j \in Y, 1 \leq j \leq |Y|$. If X and Y are sorted, then the resulting Cartesian set is a special case of a matrix with sorted rows and sorted columns. Search and selection in the above mentioned matrices have received considerable attention because of their applications in statistics, operations research and combinatorics, among others [23, 24, 25, 32, 33, 34, 37, 42, 50, 61].

Because of the constraints placed on these sets, search (as well as selection) may be performed in time sublinear in the cardinality of the set. For a sorted matrix, we assume that each element may be computed as needed in constant time or is already

available in the memory. In many applications, this is indeed the case. Hence, it is possible to have sublinear-time algorithms for search and selection in sorted matrices.

Searching in $X + Y$ has many applications ranging from finding equal keys in a file [45], which in turn has many applications, to efficiently solving NP-complete problems (e.g. the knapsack problem [24, 31, 39]). Searching in sorted matrices has applications in computational geometry [2]. Search in matrices with sorted rows and sorted columns also arises in multiplying sparse polynomials, each given by an ordered list of coefficient-exponent pairs [37].

In this chapter, we will present parallel algorithms for search and multisearch in sorted matrices. For matrices with sorted rows and sorted columns the technique of partitioning will play an important role while for matrices with sorted columns the technique of accelerated cascading will be found useful. In both types of matrices we will combine these techniques along with the technique of chaining. Since these matrices are partial orders the technique of chaining alone does not seem to lead to fast and efficient algorithms.

3.2 Search in Matrices with Sorted Rows and Columns

An $n \times n$ matrix $M = \{m_{ij}\}$ is a sorted matrix if each row and each column is in nondecreasing order. The search problem is to determine whether a given element z occurs in M or not. A related problem is of ranking the element z among the elements of M. Sequentially, the problem can be solved in $O(m+n)$ time [37]. Gries [37] refers to this problem as "Saddleback Search". The sequential algorithms, however, do not seem to be easily parallelizable.

3.2.1 Related Results

Sarnath and He [55] have recently proposed a parallel algorithm for the case of $n \times n$ matrix (square) that leads to an optimal algorithm on the CREW PRAM requiring $O(\log \log n)$ time and $O(n)$ work. Let M_1 be the portion of M whose elements are less than z and M_2 be the portion of the sorted matrix whose elements are greater than z. Then, M_1 and M_2 share a staircase shaped boundary, with length at most $2n$, running in the off-diagonal direction. Sarnath and He's algorithm [55] runs in $O(\log \log n)$ iterations successively refining the known approximation to the staircase shaped boundary until at most n submatrices of size 2×2 are left. The original approximation is the whole matrix. They also show that no comparison based parallel algorithm can solve the problem in time faster than $\Omega(\log \log n)$ using at most $n \log^c n$ processors, where c is a constant.

Matrices with sorted rows and sorted columns satisfy the property of being totally monotone. An $n \times m$ matrix $M = \{m_{ij}\}$ is *totally monotone* if for all $i < k$ and $j < l$, $m_{ij} < m_{il} \Rightarrow m_{kj} < m_{kl}$. Aggarwal *et al.* [2] gave a sequential algorithm for finding the row maxima of totally monotone matrices that requires $\Theta(m)$ time when $m \geq n$ and $\Theta(m \log 2n/m)$ when $m < n$. Later, Aggarwal *et al.* [3] presented an $O(\log n \log \log n)$ time algorithm on the CREW PRAM using $O(n \log n)$ work. Atallah and Kosaraju [9] improved the algorithm to run in $O(\log n)$ time, with the same amount of work, on the EREW PRAM model. These algorithms, however, would be inefficient for searching in sorted matrices since they require $O(n \log n)$-work and also are much more complicated than the algorithms we will present.

Finally, we note that the upper triangular or the lower triangular half of a sorted matrix is also known as a *bi-parental* heap or a *beap*. A beap [51] is an implicit data structure for which the cost of search, insert and delete is $O(\sqrt{n})$ for a beap with n elements. In contrast, search in a heap requires $O(n)$ time. A beap resembles a heap, but each node has two parents as well as two children. Thus, in a full beap level i has i nodes. The upper triangular half of a sorted matrix also forms a *Young tableaux* [45] and algorithms for search as well as many properties are described in Knuth's book [45]. Thus our search algorithms can also be used in these data structures.

3.2.2 Parallel Search in Square Sorted Matrices

We can solve the problem of searching in an $n \times n$ matrix in $O(n)$ time sequentially using the simple algorithm shown in Figure 3.1. The algorithm traces a step shaped boundary in the sorted matrix starting from the left-hand bottom corner. The invariant of the algorithm is expressed by the following:

$$((1 \leq i, j \leq n) \wedge (z \in M[1:i, j:n]))$$

For the problem of ranking, we want to find the number of elements in matrix M that are less than or equal to z. The algorithm shown in Figure 3.1 can be modified easily to compute the rank as shown in Figure 3.2.

Cosnard, Duprat and Ferreira [24] have showed that $\Omega(n)$ time is required to solve the search problem on the sorted matrices. The proof is based on the observation that

```
Algorithm 1: search(z,M)
    {M[i,j] ≤ M[i,j+1], 1 ≤ i < n sorted rows
     M[i,j] ≤ M[i+1,j], 1 ≤ j < n sorted columns }

    i ← n, j ← 1
    while (( i ≥ 1) and (j ≤ n)) do
        if (M[i,j] = z) then
            return (i,j)
        if (z < M[i,j]) then
            i ← i − 1
        else
            j ← j + 1
    endwhile
    return unsuccessful search
end
```

Figure 3.1. Sequential algorithm for search in a sorted matrix M.

```
Algorithm 2: rank(z,M)
    {M[i,j] ≤ M[i,j+1], 1 ≤ i < n sorted rows
     M[i,j] ≤ M[i+1,j], 1 ≤ j < n sorted columns}

    i ← n, j ← 1, rank ← 0
    while (( i ≥ 1) and (j ≤ n)) do
        if (z < M[i,j]) then
            i ← i − 1
        else
            j ← j + 1
            rank ← rank + i
    endwhile
end
```

Figure 3.2. Sequential algorithm for ranking in a sorted matrix M.

the n off-diagonal elements $M[n,1], M[n-1,2], \ldots, M[1,n]$ form an unordered list and z could be one of them. This optimal sequential algorithm does not seem to be easily parallelizable since each step depends upon the previous step. Sarnath and He's parallel algorithm [55] is based upon an entirely different approach. Their algorithm can be modified to run on the weaker model EREW PRAM, but a straightforward adaptation would lead to an algorithm that takes $O(\log n \log\log n)$ time, which is not optimal. We will modify their algorithm such that our algorithm runs in $O(\log n)$ time and requires $O(n)$ work, which is optimal. Our algorithm is based upon the technique of *partitioning* the matrix into many small submatrices, eliminating all but $\Theta(n \log n)$ elements of the matrix M in the process. Then we simultaneously apply the optimal sequential algorithm on each of the remaining submatrices. The algorithm is described more formally in Figure 3.3. The correctness of the algorithm is based upon the following lemmas.

Lemma 3.2.1 *Only the elements retained in the submatrices at the end of each step have to be examined in the next step in Algorithm 3.*

Proof: Consider an element x that is above a submatrix defined in Step 1. Then x is less than $M[p_i, c_i]$ and therefore less than z. Similarly, the elements below the submatrix defined in Step 1 are greater than $M[p_{i-1}+1, c_{i-1}]$ and z. □

Lemma 3.2.2 *The submatrices at the end of the Step 2 are of size at most* $\log n \times \log n$ *and there are* $\Theta(n/\log n)$ *such submatrices left.*

Proof: Follows from the structure of the algorithm. □

Algorithm 3: Parallel_Search
begin

Step 1 For each column with index $c_i = i \times \log n$, where $i = 1, 2, \ldots t (= n/\log n)$, search the column c_i for a row index p_i such that $M[p_i, c_i] < z < M[p_i + 1, c_i]$. (If z is equal to either of these elements the algorithm terminates.) For each i, $1 \leq i \leq t$, retain the elements within the submatrix $M[p_i + 1, p_{i-1} : c_{i-1}, c_i]$ and discard the rest.

Step 2 For each row with index $r_i = i \times \log n$, where $i = 1, 2, \ldots t (= n/\log n)$, search the retained elements in the row r_i for a column index q_i such that $M[r_i, q_i] < z < M[r_i, q_i + 1]$. (If z is equal to either of these elements the algorithm terminates.) For each i, $1 \leq i \leq t$, retain the elements within the submatrix $M[r_{i-1}, r_i : q_i + 1, q_{i-1}]$ and discard the rest.

Step 3 Assign one processor to each remaining submatrix and use the optimal sequential algorithm to finish the search.

end

Figure 3.3. Parallel algorithm for search in a sorted matrix.

Step 1 can be performed using $n/\log n$ processors, each doing a binary search in the appropriate column in $O(\log n)$ time. Similarly, Step 2 can be done in $O(\log n)$ time. In Step 3, there are $O(n/\log n)$ submatrices of size $\Theta(\log^2 n)$. Using the optimal sequential algorithm Step 3 will also finish in $O(\log n)$ time. We can modify the sequential algorithm that is run on each submatrix to compute its share of the rank for the original matrix. At the end of Step 3 we just have to sum $O(n/\log n)$ numbers in parallel, which can be done in $O(\log n)$ time using $O(n/\log n)$ work. Thus we have the following result.

Theorem 3.2.3 *Given an $n \times n$ matrix M with sorted rows and columns and an element z, the problem of searching for z in the matrix M as well as ranking can be solved in $O(\log n)$ time using $O(n)$ work on the EREW PRAM.*

3.3 Multisearch in Matrices with Sorted Rows and Columns

We will now consider the problem of performing multiple searches simultaneously without any read conflicts on the EREW PRAM. We are given r elements $z_1 < z_2 < \ldots < z_r$ to search for in an $n \times n$ sorted matrix M. If these elements are not sorted, they can be sorted in $O(\log r)$ time using r processors [21]. First we will consider a preliminary algorithm where there is one processor assigned to search each element. This parallel algorithm is based upon the technique of *pipelining.* Then we will generalize the fast parallel algorithm for a single search (from Section 3.2) and combine it with the simple pipelined algorithm to obtain a fast parallel algorithm for multisearch in a sorted matrix M.

3.3.1 A Simple Pipelined Algorithm for Multisearch

Consider the sequential algorithm shown in Figure 3.1. The search follows a step shaped path in the off-diagonal direction. In each step the search either moves up one row or right one column. Let $P_1, P_2, \ldots, P_r$ be the r processors associated with searching for corresponding elements $z_1 < z_2 < \ldots < z_r$. Then the path traced by processor P_1 while searching for z_1 is bounded below by the paths traced by processors searching for larger elements $z_2 < z_3 < \ldots < z_r$. Recall that the matrix is sorted in nondecreasing order from top to bottom in each row and from left to right in each column. However, segments of the step-shaped boundary may be common to two or more searches as is shown next.

Let $z_1 < z_2$ be two elements for which we have two processors P_1 and P_2 searching. Consider the fragment of the matrix shown in the Figure 3.4 and suppose that

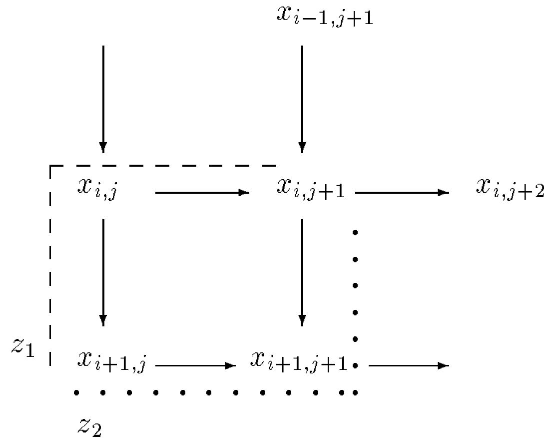

Figure 3.4. Paths followed by two searches in a sorted matrix.

$z_1 < x_{i+1,j}$ and $z_1 > x_{i,j}$. Then the search for z_1 ends up at $x_{i,j+1}$ in two steps starting from $x_{i+1,j}$. Suppose $z_2 > x_{i+1,j}$ and $z_2 < x_{i+1,j+1}$. Then the search for z_2 also ends up at $x_{i,j+1}$ in two steps. From that point the two searches can either stay together or z_1 goes up and z_2 goes to the right, that is, the two searches never cross paths otherwise it implies that $z_1 > z_2$. Thus delaying the search for z_2 by one step ensures that there will be no read conflicts. By extending this argument for r processors, we obtain an $O(n + r)$-time parallel algorithm for r searches on the EREW PRAM.

3.3.2 The Main Parallel Algorithm

The pipelined algorithm presented in previous section makes efficient use of processors but is not fast. In order to obtain a fast parallel algorithm we will generalize the

parallel algorithm for a single search that was presented in Section 3.2. The generalized algorithm will use $rn/\log n$ processors and our goal is to solve all the r searches in $O(\log n + \log r)$ time.

First, let us examine the problem of multiple binary search in a single sorted array of size n since we will be using that as a subroutine. Using the technique of chaining Chen [20] has shown how to perform multiple binary searches efficiently. His result is stated in the following theorem.

Theorem 3.3.1 (Chen [20]) *Given a sorted array of size n, elements $z_1 < z_2 < \ldots < z_r$ can be searched for in $O(\log n + \log r)$ time using r processors on the EREW PRAM.*

Chen also gave another algorithm for multiple search that is more complicated but optimal.

Theorem 3.3.2 (Chen [20]) *Given a sorted array of size n, elements $z_1 < z_2 < \ldots < z_r$ can be searched for in $O(\log n + \log r)$ time using $r \log(n/r)/\log n$ processors on the EREW PRAM.*

Now we are ready to describe the parallel algorithm. In Step 1, for each column with index $c_i = i \times \log n$, where $i = 1, 2, \ldots, , t(= n/\log n)$ use the multiple binary search algorithm by Chen [20] using r processors to find row indices $p_i^{(j)}$, $j = 1, 2, \ldots, r$ such that

$$M[p_i^{(j)}, c_i] < z_j < M[p_{i+1}^{(j)}, c_i] \;\; j = 1, 2, \ldots, r.$$

Since each column is sorted we can use the multiple binary search algorithm. At the end of the search define submatrices as follows.
For each j, where $1 \leq j \leq r$ and for each i, where $1 \leq i \leq t$, retain the elements in the submatrix $M[p_i^{(j)} + 1, p_{i-1}^{(j)} : c_{i-1}, c_i]$ and discard the rest.

In Step 2, for each row with index $r_i = i \times \log n$, where $i = 1, 2, \ldots, , t(= n/\log n)$ use the multiple binary search algorithm by Chen [20] using r processors to search for $z_1 < z_2 < \ldots < z_r$ in the retained elements in row r_i for column indices $q_i^{(j)}$, $j = 1, 2, \ldots, r$ such that

$$M[r_i, q_i^{(j)}] < z_j < M[r_i, q_i^{(j)} + 1] \;\; j = 1, 2, \ldots, r.$$

Since each row is also sorted we can use the multiple binary search algorithm. At the end of the search define submatrices as follows. For each j, where $1 \leq j \leq r$; For each i, where $1 \leq i \leq t$, retain the elements in the submatrix $M[r_{i-1}, r_i : q_i^{(j)} + 1, q_{i-1}^{(j)}]$ and discard the rest.

In Step 3, each search has been narrowed down to $O(n/\log n)$ submatrices of size at most $\log n \times \log n$. However, each submatrix may have up to r searches still left in it. Each such search has a processor associated with it and these processors will have read conflicts if we allow them to search at the same time. Let r' denote the number of searches (processors) sharing a submatrix. Then depending upon the value of r' we have the following three cases.

Case I. $r' = O(\log n)$. Then we can use the simple pipelined algorithm described in Section 3.3.1. Since each submatrix is of size at most $\log n \times \log n$ the time required is $O(\log n + r')$, which is $O(\log n)$.

Case II. $r' = \Omega(\log^2 n)$. Since the number of processors assigned to the submatrix is larger than the number of elements in the submatrix we can afford to sort the $O(\log^2 n)$ elements in the submatrix while retaining the original i, j indices for each element. Sorting the elements of the submatrix gives a sorted chain of $O(\log^2 n)$ elements with at least one processor per element. Using the multiple binary search algorithm the search can be finished in $O(\log r + \log(\log^2 n))$ time.

Case III. $\log n < r' < \log^2 n$. In this case we don't have enough processors to sort all the remaining elements in the submatrix and we have too many searches to be done in $O(\log n + \log r)$ time using the simple pipelined algorithm. The idea is to apply the algorithm recursively to each such submatrix. In Step 1, let $t = \sqrt{\log n}$ and let column indices be $c_i = i \times \sqrt{\log n}$, where $i = 1, 2, \ldots, t$. Similarly take $\sqrt{\log n}$ rows and apply Step 2 appropriately. Now each remaining subsubmatrix has size at most $\sqrt{\log n} \times \sqrt{\log n}$. The number of searches left in each subsubmatrix is either $r' > \log n = (\sqrt{\log n})^2$ and we can apply the solution from Case II or the number of searches left $r' < \log n$ and we can use the pipelined algorithm as in Case I.

Step 1 uses $rn/\log n$ processors and requires $O(\log n + \log r)$ time using the result from Theorem 3.3.1. Step 2 can be performed with the same time and processor requirements. In Step 3, the time required depends upon what case the submatrix falls in. For Case I the time is $O(\log n)$ while for Case II the time required is $O(\log r + \log\log n)$. In Case III the recursive application of Steps 1 and 2 requires $O(\log\log n)$

time using $\sqrt{\log n}$ processors since we are searching in sorted rows and columns of size $\log n$. In the recursive application of Step 3 the time required depends upon whether it is reduced to Case I or Case II. If the number of searches, r', left in the subsubmatrix is less than $\log n$ then the time required is $O(\log n)$ using the simple pipelined algorithm. Otherwise the time required is $O(\log r' + \log\log n)$, based on the analysis for Case II. Thus we have the following result.

Theorem 3.3.3 *Given an $n \times n$ matrix M with sorted rows and columns and r elements $z_1 < z_2 < \ldots < z_r$, the problem of multisearch for these r elements in the matrix M as well as ranking can be solved in $O(\log n + \log r)$ time using $O(rn)$ work on the EREW PRAM.*

Remark 1. The algorithm also requires that $O(n/\log n)$ copies of the r elements be made, which can surely be done in $O(\log n)$ time using $O(rn/\log n)$ work using recursive doubling on the EREW PRAM [41]. Note that the working space used per processor is only $O(1)$.

Remark 2. At the end of Steps 1 and 2 each processor knows which submatrix it needs to search in and the set of processors that are assigned to the same submatrix are consecutive. Thus identifying these processors and combining their searches can be accomplished using the broadcasting algorithm for the EREW PRAM. Furthermore, we can count the number of searches assigned to each submatrix and decide whether Case I, II, or III applies in Step 3. All of these can be done in $O(\log r)$ time and within the work bounds stated in Theorem 3.3.3.

Remark 3. The computation of ranks for the elements $z_1 < z_2 < \ldots z_r$ can be done using the same approach as for a single search, since each group of processors

allocated to a search have the ranks in the corresponding submatrices. In Case II, however, the sorting process while it gives the rank of the element being searched for among the elements of the submatrix but it does not identify which row or column in the upper or right edge of the submatrix the search ends. We need this information in order to compute the rank in the overall matrix. But we can search for the r search-elements in the upper most row in $O(\log r + \log\log m)$ time using r processors. Suppose the search for z_i exits in the jth column. Then its rank is $j-1$ times the row index of the upper most row of the submatrix plus its rank in the submatrix. Some of the searches may end up in the right edge. All of these will be greater than the right most element in the upper most edge. For these elements the rank is $\log m$ times the row index of the upper most row plus their rank in the submatrix.

3.4 Multisearch in X+Y

Let $X = \{x_1, x_2, \ldots, x_n\}$, and $Y = \{y_1, y_2, \ldots, y_n\}$ be the two input vectors. Both X and Y are sorted in increasing order. Given z, the search problem is to find a p and q, if they exist, such that $z = x_p + y_q$. In the multisearch problem, we are given r values, $z_1, \ldots, z_r$, to search for, where $z_1 < z_2 < \ldots < z_r$. A lower bound of $\Omega(n)$ was proved for search in $X+Y$ by Cosnard, Duprat and Ferreira [24]. An $O(n)$-time sequential algorithm has been known under different contexts (e.g. Knuth [45]). The sequential algorithm is almost identical to the one for a matrix with sorted rows and columns, shown in Figure 3.1, except replacing x_{ij} with $x_i + y_j$.

Cosnard and Ferreira [25] have presented parallel algorithms for search in $X+Y$. One of their optimal parallel algorithms is based on merging. Using the relation

that $x_p + y_q = z$, we have $x_p = z - y_q$ and thus, search in $X + Y$ is equivalent to detecting common elements in X and $z - Y$. Their search algorithm uses merging to detect common elements in X and $z - Y$ and requires $O(\log n)$ time using $O(n/\log n)$ processors, if we use an optimal EREW PRAM merging algorithm such as the one presented in Chapter 2. The merging involves unnecessary data movement, instead if we could search for the elements of $z - Y$ in X directly then the problem could be solved without resorting to merging. Using the multiple binary search algorithm from Chen [20] (which is based on the technique of chaining by Paul, Vishkin and Wagner [53]) we can solve the problem of search in $X + Y$ optimally on the EREW PRAM using only $O(m + n)$ space.

The idea of search in $X + Y$ can be extended to multisets of the form $X_1 + X_2 + \ldots + X_m$ where $m > 2$ and all the X_i are sorted vectors of size n. The general problem for any m and n is NP-hard as it contains the knapsack problem as a special case. Cosnard, Duprat and Ferreira [24] give an $O(mn^{\lceil m/2 \rceil} \log n)$-time algorithm with $O(n^{\lceil m/4 \rceil})$ auxiliary space requirement using the basic $X + Y$ algorithm. In particular, they present an $O(n^2)$ algorithm for search in $X + Y + W$ and leave as an open problem if $\Omega(n^2)$ is the lower bound. Parallel algorithms for search in $X + Y$ can be used to solve the knapsack problem [31, 30].

We will consider the multisearch problem in $X + Y$ on the EREW PRAM. On a CREW PRAM, this problem is easy to solve as read conflicts are allowed. But on an EREW PRAM only one processor may access a memory location, which makes the multisearch problem harder. A potential use of an multisearch algorithm is for

solving the $X+Y+W$ search problem since search in $X+Y+W$ can be posed as a multisearch for $z-w_1, z-w_2, \ldots, z-w_n$ in $X+Y$, where $w_i \in W$, $1 \leq i \leq n$.

We can't directly use the multisearch algorithm presented for sorted matrices as there would be many read conflicts. All processors accessing row i, for example, will all be accessing x_i and so on. But if we have $rn/\log n$ processors available we can generate the r vectors $z_1 - y_i, z_2 - y_i, \ldots, z_r - y_i$ where $1 \leq i \leq n$ as well as r copies of the vector X. This requires $O(\log n)$ time using $rn/\log n$ processors on the EREW PRAM. Now we can use the parallel algorithm for search in $X+Y$ on all r pairs simultaneously. The total time required is $O(\log n)$ time and the work done is $O(rn)$. Note that the working space required is $O(\log n)$ per processor.

3.5 Search in Matrices with Sorted Columns

Matrices where only the columns are sorted present another useful structure for investigating search algorithms. An $m \times n$ matrix with sorted columns can be equivalently viewed as m arrays of size n each since there is no order among the columns. We are interested in searching for the given value z as well as computing the rank of the value z. Let this rank be denoted by k. This problem was solved sequentially by Frederickson and Johnson [32], and we will now sketch their optimal sequential algorithm.

The optimal sequential algorithm consists of performing an one-sided binary search, proposed by Bentley and Yao [12], on each column to find the insertion point in each column. If the value z is found during this procedure then the search is done, otherwise the one-sided binary search identifies an interval where the value may lie. This phase

of the one-side binary search is just a linear search over the elements $1, 2, 4, 8, \ldots, 2^i$. Next, ordinary binary search is used on this interval to either find the value or locate the insertion point. Let the insertion point in the jth column be k_j, where $k_j > 0$. In $O(m)$ time, we can easily determine k_js that are non-zero and then carry out the search only in those columns. Of course, if the rank $k < m$ then at most k columns will be selected. The time required for search in each column is $O(\log k_j)$ and, if $p = \min\{k, m\}$, the total time required by the algorithm is

$$\sum_{j=1}^{p} \log k_j = \log(k_1 k_2 \cdots k_p) \leq \log(k/p)^p = O(p \log(k/p)).$$

Frederickson and Johnson [32] also proved a corresponding lower bound. There is no reported parallel algorithm, to the best of our knowledge, for search or multisearch on matrices with sorted columns.

For ease of exposition we consider two cases:(i) the rank $k \leq m$, and (ii) the rank $k > m$. In the first case, the time taken by the optimal sequential algorithm is $O(m)$ and in the second case the time taken is $O(m \log(k/m))$. In the extreme case when $k = \Theta(mn)$, the time taken is $O(m \log n)$. Note that the *partitioning* technique used for designing parallel algorithm for matrices with sorted rows and sorted columns is not applicable here since the rows are not sorted. Straightforward approaches like using m processors and performing a binary search in each row in $O(\log n)$ time is fast but not cost-optimal except in the extreme case of the rank being $\Theta(mn)$. If the rank of the element z is less than n then we could improve the cost of this approach slightly to $O(m \log k)$ by using the one-sided binary search in each row. But that is

still not cost-optimal as many processors will idle in rows where the one-sided search ends much sooner than other rows. We will attempt to develop a cost-optimal parallel algorithm by reducing the number of processors to $m/\log m$.

Consider the first case, when $k \leq m$. Since we have $m/\log m$ processors the natural approach seems to be to divide the columns into blocks of $\log m$ columns and assign one processor to perform the one-sided binary search in each block of rows. Since there is no relative ordering among the rows, all of the search may be concentrated in one or a few blocks leading to little or no improvement. Instead we will use the idea of *accelerated cascading* of two algorithms, one cost-optimal but relatively slow, the other fast but not cost-optimal. The main idea is to proceed with the one-sided binary searches in parallel in each column till the number of unterminated searches is $O(m/\log m)$ and then to switch to the fast algorithm where one processor is assigned to each of the $O(m/\log m)$ remaining columns.

Initially the first element in each column is examined in $O(\log m)$ time using $m/\log m$ processors. This is the first step of the one-sided binary search in each column. Now let r_1 be the number of unterminated searches. The algorithm proceeds in two phases.

Phase I. In each iteration use $m/\log m$ processors to simulate one step each of the linear search part of the one-sided binary search, that is, examine elements at positions $2, 4, 8, \ldots, 2^i$ in each column. This requires $r_i/(m/\log m)$ steps where r_i is the number of unterminated searches before iteration i. Continue this phase until $r_i \leq m/\log m$. At this point the number of unterminated searches is less than or equal to the number

of processors, Thus, each processor can finish the one-sided binary search taking no more than $O(\log m)$ time.

Phase II. Reallocate the processors to the remaining intervals identified in Phase I such that each processor finishes the search in a section of $O(\log m)$ elements, possibly spanning several columns.

At the end of Phase I we have either found the rank k_j of the element z in the jth column or have identified an interval to which the element z belongs. The size of the interval is $< 2k_j$ where k_j is the rank of element z in column j. The total size of these intervals is $O(k)$ and there could be as many as $O(k)$ such intervals. However, the size of any one interval could be as much as $2k$.

Since the rank $k \leq m$ the number of unterminated searches after initialization, r_1, is less than m. In iteration i of Phase I the 2^ith element is examined in each of the r_i columns where the search has not been terminated. Thus the value of $r_i < m/2^i$, otherwise we are examining elements with rank greater than $m \geq k$. Phase I ends when $r_i < m/2^i \leq m/\log m$, which implies that the number of iterations is $O(\log\log m)$. Furthermore the total work done during Phase I is given by

$$\sum_{i=1}^{\log\log m} \frac{m}{2^i} = O(m)$$

in the case when $k = m$ and $O(k)$ if $k < m$. In each iteration, however, the $m/\log m$ processors have to be reassigned to those columns where the one-sided binary search hasn't finished. This processor scheduling and assignment requires a parallel prefix operation to compact the list of columns where the search has not yet finished. The

parallel prefix can be computed in $O(\log m)$ time using $O(m)$ work [46]. Thus the overall time for Phase I is $O(\log m \log\log m)$.

For implementing Phase II we construct two arrays $WT[1..m]$, where $WT[j]$ is the number of elements left in the interval identified in the jth column, and $POS[1..m]$, where $POS[j]$ is a two tuple identifying the start and end of the interval in the jth column that still has to be searched. Note that the sum of the values in the array WT is $O(m)$, since $k \leq m$, and is otherwise $O(k)$. In Phase II we want to divide these $O(m)$ elements equally among the $m/\log m$ processors for finishing the search. The Phase II can be performed as follows.

Phase II.a Compute the parallel prefix of the array $WT[1..m]$.

Phase II.b Perform multisearch for $i \times \log m$, where $1 \leq i \leq m/\log m$ in the array consisting of the partial sums for array $WT[1..m]$. The result of the multisearch are column indices.

Phase II.c Let the ith processor, where $1 \leq i \leq m/\log m$, use simple linear search in its section of size $O(\log m)$ and, if desired, compute the rank.

All of these steps of Phase II can be finished in $O(\log m)$ time using $O(m)$ work. Overall the algorithm requires $O(\log m \log\log m)$ time, but since we are using $m/\log m$ processors the cost is not optimal. Since the total work done is only $O(m)$ we can use Brent's Scheduling and reduce the number of processors to $(m/\log m \log\log m)$ and then the time required for Phase I would be

$$O(\log m \sum_{i=1}^{\log\log m} \frac{\log\log m}{2^i}) = O(\log m \log\log m)$$

Phase II will also require $O(\log m \log\log m)$ time using $m/\log m \log\log m$ processors. Thus we have the following result.

Theorem 3.5.1 *Given an $m \times n$ matrix with sorted columns, a value z, with rank $k \leq m$, can be searched for and its rank k computed in $O(\log m \log\log m)$ time and $O(m)$ work on the EREW PRAM.*

Now we will consider the algorithm for the case when the rank $k > m$. Note that we will detect that this is the case at the end of Phase I by summing up the weight array WT. Consider iteration i in Phase I. If the one-side binary search is active in all columns then we have examined $2^i m$ elements already. This can continue as long as $2^i m < k$, that is, the number of iterations is $O(\log(k/m))$. In each iteration we need $O(\log m)$ time to compact the list of unfinished searches using parallel prefix. Thus the overall time for this part of Phase I is $O(\log m \log(k/m))$ using $m/\log m$ processors. For the rest of Phase I of the algorithm, we need to distinguish between two cases: (i) when $k/m \geq \log m$, and (ii) $k/m < \log m$. If $k/m \geq \log m$ then in the next iteration there are at most $O(m/\log m)$ unfinished one-sided searches. These can be done in $O(\log n)$ time, in the worst-case, using $O(m \log n/\log m)$ work, which in this case is still $O(m \log(k/m))$. In addition, we will also have computed the rank of the element z in these columns. If $k/m < \log m$ then we may need another $O(\log\log m)$ iterations to finish the Phase I. So Phase I requires $O(\log m \log\log m + \log m \log(k/m)))$ time in this case and the work is $O(m \log(k/m) + m)$.

In Phase II we may have one interval each left in some of the columns after the one-sided binary search. The total number of elements in these intervals is given by $\sum_{j=1}^{m} WT[j]$, which is less than $2k$. If $k/m \geq \log m$ then the size of any interval

is $O(k/m)$. Assigning one processor to $\log m$ columns finishes the search and the computation of the rank in $O(\log m \log(k/m))$ time. If, however, $k/m < \log m$ then the size of the intervals in each column could be as much as $O(\log m)$ and the above approach leads to a time of $O(\log m \log\log m)$ with $O(m \log\log m)$ cost, which is not optimal. Cost-optimality can be achieved by the following method. The sum of the entries in the WT array provides us with an estimate on the value of the rank k, let this be k'. Take the first k'/m elements from each interval and use one processor to search $\log m$ columns in $O(\log m \log(k/m))$ time and optimal cost. Now the number of elements remaining are $O(m)$ and we can use the same approach as use in the case where rank $k \leq m$. This can be done in $O(\log m)$ time with $O(m)$ work.

Overall the algorithm runs in $O(\log m \log(k/m) + \log m \log\log m)$ time and $O(m \log(k/m))$ work. Applying Brent's Scheduling we can use $m/\log m \log\log m$ processors and the resulting algorithm is cost-optimal for all values of the rank k.

Theorem 3.5.2 *Given an $m \times n$ matrix with sorted columns, a value z can be searched for and its rank k computed in $O(\log m \log\log m + \log m \log(k/m))$ time and $O(m + m\log(k/m))$ work on the EREW PRAM.*

3.6 Multisearch in Matrices with Sorted Columns

The problem of multisearch consists of ranking the elements $z_1 < z_2 < \ldots < z_r$ in the $m \times n$ matrix M with sorted columns. To avoid confusion we will identify the elements $z_1, \ldots, z_r$ as the search-elements and the elements of the matrix M as merely elements. The number of search-elements, r, is assumed to be between 1 and mn.

First we will consider sequential algorithms for the problem of multisearch that are sensitive to the number of search-elements and to the rank of the search-elements. Then we will present the parallel algorithm for multisearch in matrices with sorted columns.

3.6.1 Sequential Algorithm

Initially we can sort the first row of the matrix M in $O(m \log m)$ time and then merge the elements $z_1, \ldots, z_r$, in $O(m + r)$ time, with the sorted row in order to determine which columns are to be searched. Now we can search for each element separately. The total time required by this simple approach is

$$O(m \log m + r + \sum_{i=1}^{r} m \log(K^{(i)}/m)),$$

where $K^{(i)}$ is the rank of the ith element z_i.

We can improve the above sequential algorithm as follows. First search for the largest element z_r so that we have a bound on how far to search for the rest of the elements in each column. This requires $O(m \log(K^{(r)}/m))$ time, where $K^{(r)}$ is the rank of the element z_r. Let the rank of the element z_r in column j be $K_j^{(r)}$. Note that since $z_1 < z_2 < \ldots < z_r$, we have to only search elements at indices no more than $K_j^{(r)}$ in column j.

In addition we also have to determine in what columns we should search for each of the elements, which is first step of the one-sided search. If the number of elements r is very small then we could just check one at a time in $O(rm)$ time. Otherwise we

can first sort the matrix elements in the first row and then use the multiple binary search algorithm [20] to search for the r elements in the m sorted elements of the first row. If $r \leq m$, then the total time for this initial step is $O(m \log m + r \log(m/r))$, which is $O(m \log m)$. Otherwise, if $r > m$, we can search for the m sorted elements in the r given elements in $O(m \log m + m \log(r/m))$-time.

After the one-sided binary search for z_r ends we have identified a set of elements of size at most $O(m)$ if the rank of z_r is less than or equal to m. In this case we can merge the remaining $r-1$ search-elements with at most $O(m)$ elements identified during the search for z_r and we would have the ranks for all the search-elements.

In all other cases we use the optimal sequential algorithm for multiple binary search presented by Chen [20] on each column where the elements may be found. This algorithm requires $O(r \log(K_j^{(r)}/r))$ time for searching r elements in a sorted column of size $K_j^{(r)}$. Thus the total time for the algorithm is now

$$O(\sum_{j=1}^{m} r \log(K_j^{(r)}/r)),$$

which can be simplified to $O(rm \log(K^{(r)}/mr))$.

In case the number of elements r is greater than n or $K^{(r)} < rm$, we can change the search around such that we are searching for the elements of the column in the r given elements. The time-complexity then becomes

$$O(\sum_{j=1}^{m} K_j^{(r)} \log(r/K_j^{(r)})),$$

which can be simplified to

$$O(K^{(r)} \log(rm/K^{(r)})).$$

Note that the value of $K^{(r)} > m$ for this case and the value of $r > n$. These results are summarized in the next theorem.

Theorem 3.6.1 *Given r search-elements, $z_1 < z_2 < \ldots < z_r$, to search for in an $m \times n$ matrix M with sorted columns, the time-complexity of the sequential algorithm is:*

1. $1 < r \leq \log m$:
   (a) $K^{(r)} = O(m)$: $O(rm)$.
   (b) $K^{(r)} = \Omega(m)$ *and* $K^{(r)} < rm$: $O(rm + K^{(r)} \log(rm/K^{(r)}))$.
   (c) $K^{(r)} = \Omega(m)$ *and* $K^{(r)} > rm$: $O(rm + rm \log(K^{(r)}/rm))$.
2. $\log m < r \leq m$:
   (a) $K^{(r)} = O(m)$: $O(m \log m)$.
   (b) $K^{(r)} = \Omega(m)$ *and* $K^{(r)} < rm$: $O(m \log m + m \log(K^{(r)}/m) + K^{(r)} \log(rm/K^{(r)}))$.
   (c) $K^{(r)} = \Omega(m)$ *and* $K^{(r)} > rm$: $O(m \log m + m \log(K^{(r)}/m) + rm \log(K^{(r)}/rm))$.
3. $m < r \leq n$:
   (a) $K^{(r)} < rm$: $O(m \log r + m \log(K^{(r)}/m) + K^{(r)} \log(rm/K^{(r)}))$.
   (b) $K^{(r)} > rm$: $O(m \log r + m \log(K^{(r)}/m) + rm \log(K^{(r)}/rm))$.
4. $n < r \leq mn$: $O(m \log r + K^{(r)} \log(rm/K^{(r)}))$.

TABLE 3.1

VARIOUS CASES FOR THE SEQUENTIAL TIME-COMPLEXITY OF THE MULTISEARCH ALGORITHM

number of searches	maximum rank	time-complexity
$1 < r \leq \log m$	$K^{(r)} = \Theta(m)$	$O(rm)$
	$K^{(r)} = \Theta(mn)$	$O(rm + rm\log(n/r))$
$\log m < r \leq m$	$K^{(r)} = \Theta(m)$	$O(m\log m + r\log(m/r))$
	$K^{(r)} = \Theta(mn)$	$O(m\log n + rm\log(n/r))$
$m < r \leq n$	$K^{(r)} = \Theta(m)$	not possible since $r > m$
	$K^{(r)} = \Theta(mn)$	$O(m\log n + rm\log(n/r))$
$n < r \leq mn$	$K^{(r)} = \Theta(m)$	not possible since $r > n \geq m$
	$K^{(r)} = \Theta(mn)$	$O(m\log r + mn\log(r/n))$

The time-complexity for the extreme cases of the rank $K^{(r)}$ being $\Theta(m)$ and $\Theta(mn)$ is illustrated in Table 3.1.

3.6.2 Parallel Algorithm

The parallel algorithm for multisearch is also based upon the how many elements we want to search for. Initially we run the parallel algorithm for single search and find the largest search-element z_r and its rank $K^{(r)}$ as well as its rank in each column, $K_j^{(r)}$, $1 \leq j \leq m$. The time taken depends upon the rank of z_r. Let us first consider the case $K^{(r)} \leq m$, which also implies that $r \leq m$ since all the search-elements are distinct. Similar to the sequential algorithm proposed in the previous section, two cases have to be examined separately.

If $1 \leq r \leq \log m$ then after execution of the Phase II.a and II.b of the parallel algorithm for a single search we have identified a total of $O(m)$ elements within sections $[1 \ldots K_j^{(r)}]$, $1 \leq j \leq m$. Phase II.b for a single search also identifies sections of size $O(\log m)$, but now we may have multiple processors searching within these sections. Since $r \leq \log m$, we can simply pipeline the r searches through the section in time $O(r + \log m)$ or equivalently $O(\log m)$ making the total work $O(rm)$. Thus we have $O(\log m \log \log m)$ time algorithm that performs $O(rm)$ work. Similar to the single search, we can apply Brent's Scheduling and reduce the number of processors to $rm/\log m \log \log m$ and keep the same running time.

If $\log m < r \leq m$ then after searching for z_r, we can simply merge the $O(m)$ elements with the r search-elements in $O(\log m)$ time using $m/\log m$ processors and thus compute all the ranks. The total time is then $O(\log m \log \log m)$ and the total work is $O(m \log m)$.

Now we will consider the case when the rank $K^{(r)} > m \log m$, similar to the parallel algorithm for a single search. At the end of the one-sided binary search for the search-element z_r we have narrowed the search to $m/\log m$ columns in $O(\log m \log(K^{(r)}/m))$ time and $O(m \log(K^{(r)}/m))$ work. At this point we will finish the search in these $m/\log m$ columns and know the rank of the search-element z_r in these columns. Thus we can perform multisearch for the rest of $r-1$ search-elements in $O(\log r + \max_j\{\log K_j^{(r)}\})$-time and the work performed is no more than that of the sequential algorithm. After this phase we still have to go back and complete the search in the intervals identified during the initial one-sided search. In this case we will assign r processors to $\log m$ columns to the intervals identified by the search for the element z_r.

From earlier arguments we know that the size of these intervals is $O(K^{(r)}/m)$. We will use the multisearch algorithm by Chen [20], which was summarized in Theorem 3.3.2. The multisearch in one column requires $O(\log(K^{(r)}/m)+\log r)$ time leading to overall time of $O(\log m(\log(K^{(r)}/m) + \log r))$. If $K^{(r)}/m < r$ then we can instead search for the at most $K^{(r)}/m$ sorted elements of a column in the r search-elements and the algorithm has the same cost as the sequential algorithm discussed in the previous subsection.

The other case to be considered is when $n < r \leq mn$. Then we use the multisearch algorithm to search for the elements of the column in the r search-elements. The time is same as the previous case and the work done is the same as in the sequential algorithm. Finally, when the rank $K^{(r)}$ is less than $m \log m$, only in the case $1 \leq r \leq \log m$, then we can split the elements as done in the case for the parallel algorithm for single search and then apply the two methods proposed above on parts of the problem.

3.7 Faster Parallel Search in Matrices with Sorted Columns

The parallel algorithm for searching in a matrix with sorted columns presented earlier can be made to run asymptotically faster while maintaining cost-optimality. The algorithm in this section is inspired by the ideas contained in the paper by Bentley and Yao [12] on unbounded searching. Bentley and Yao present a series of algorithms for searching in an unbounded sorted array for an element with rank n. Their ultimate algorithm requires the following number of comparisons:

$$\log n + \log^{(2)} n + \log^{(3)} n + \ldots + 2\log^{(q)} n + 1,$$

where $\log^{(q)}$ is the log function applied q times to n. The value q is chosen such that $q = \min_q\{\log^{(q)} n \leq 1\}$, that is, $q = O(\log^* n)$. If we use the simple unbounded search then the number of comparisons is $(2\log n + 1)$, which differs from the ultimate algorithm only by a constant factor. Surprisingly, when we apply these ideas to searching in matrices with sorted columns the time will be asymptotically faster.

Recall that we examine elements at index at most $\log m$ before the number of searches is definitely reduced below $m/\log m$. Thus we can take our $n = \log m$. Instead of examining indices $1, 2, \ldots, 2^i$ we will instead examine indices as follows:

$$2, 2^2, 2^{2^2}, \ldots, 2^{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}$$

and thus the number of iterations is $\log^*(\log m)$. After the one-sided search ends we have identified an interval. Within this interval we will now use ordinary binary search but over sparsely situated elements. First time we will examine only $\log^{(q)} m$ elements and then $\log^{(q-1)} m$ elements and so until the last stage when we examine at most $\log m$ elements per column. Thus there are q stages where $q = O(\log^*(\log m))$.

Suppose that the rank of the element being searched for is $k \leq m$. Then in each iteration the number of unfinished searches is much less than half of that in the previous iteration. Therefore the $O(\log^*(\log m))$ iterations of the first stage can be implemented in $O(\log m \log^*(\log m))$ time and $O(m)$ work. In order to finish the search we have to successively refine the interval until we are left with $O(m)$ elements. This is accomplished by the nested binary searches mentioned in the previous paragraph.

The number of such searches is $O(\log^*(\log m))$. We will show that the number of elements examined in the ith nested binary search is no more than $m/2^i$, thus showing that the overall work is $O(m)$ and the time required is $O(\log m \log^*(\log m))$.

Let the rank of the element in column j be k_j. First we observe that

$$\sum_{j=1}^{m} \log k_j = O(m \log(k/m)),$$

which is $O(m)$ (since $k \leq m$). This is the case in the last stage. In the nested search in the stage before that the total number of elements examined is

$$\sum_{j=1}^{m} \log\log k_j \leq \sum_{j=1}^{m} (\log k_j)/2 = O((m \log(k/m)/2)),$$

or $O(m/2)$. Similarly in the qth nested search before the last stage the the total number of elements examined is $O(m/2^q)$. Thus we can search for a element with rank $k \leq m$ in $O(\log m \log^*(\log m))$ time and $O(m)$ work.

When the rank $k > m \log m$, then the number of iterations in the first stage is $O(\log^*(k/m))$ and the overall time is $O(\log m \log^*(\log m) \log(k/m))$ if we are using $m/\log m \log^*(\log m)$ processors. When the rank is intermediate then we can use the strategy used before in Section 3.5 and obtain the time $O(\log m \log^*(\log m) \log(k/m) + \log m \log^*(\log m))$ with $O(m + m \log(k/m))$ work.

3.8 Summary

In this chapter, we have presented parallel algorithms for search and multisearch in sorted matrices. First we presented an $O(\log n)$-time parallel algorithm using $O(n)$ work for searching and ranking in an $n \times n$ matrix with sorted rows and sorted columns. The algorithm required the technique of partitioning. This algorithm together with the technique of chaining then served as a basis for the design of a parallel algorithm for multisearch in the $n \times n$ matrix with sorted rows and sorted columns. The multisearch algorithm for r searches runs in $O(\log n + \log r)$ time and $O(rn)$ work, which matches the best known sequential time. These algorithms are quite different from their sequential counterparts. Then we also briefly considered the use of chaining to design parallel algorithm for search in $X + Y$ where both X and Y are sorted vectors implicitly representing a sorted matrix.

The second part of the chapter dealt with search in $m \times n$ matrices with sorted columns or equivalently search in m sorted arrays of size n each. Since there is no relative order of the elements in the rows of such a matrix the technique of partitioning is not applicable. Furthermore the parallel as well as the sequential algorithm are sensitive to the rank of the search-element. We proposed a cost-optimal algorithm that runs in $O(\log m \log\log m)$ time for small elements (with rank $\leq m$) and in time $O(\log m \;\; \log\log m \;\; \log(k/m))$ for large elements. This algorithm uses the technique of accelerated cascading. Then we presented a sequential algorithm for multisearch in a matrix with sorted columns as a prelude to the parallel algorithm. The sequential algorithm is inspired by the parallel technique of chaining. The parallel algorithm follows this sequential algorithm and has a nontrivial dependence not

only on the ranks of the search-elements but also on the number of search-elements. Finally we show how to adapt ideas from Bentley and Yao's [12] classic paper on sequential unbounded searching and improve the search algorithm for small elements to run in $O(\log m \log^*(\log m))$ time with optimal work and for large elements in $O(\log m \log^*(\log m) \log(k/m))$ time with optimal work.

Chapter 4

PARALLEL MIN-COST FLOW

4.1 Introduction

The max-flow problem is a fundamental problem in combinatorial optimization and tremendous amount of work has been done for obtaining its algorithmic solution. In this chapter, we will design a fast parallel algorithm for the min-cost flow problem on series-parallel networks. Studies on such classes of graphs is well-motivated since often such results shed light on more complex classes of graphs. Furthermore one may encounter such graphs in typical real-life situations and thus expect a faster solution than for general graphs.

The parallel algorithms that we developed in Chapter 1 for multiselection and merging form an important building block for our parallel min-cost flow algorithm. These are combined with the technique of tree contraction to obtain an efficient parallel algorithm. Our parallel solution to the the min-cost flow problem gives us another sequential algorithm quite different than the best known sequential algorithm. The next section contains a description of related previous work. The section following that has the formal definition of the min-cost max-flow problem. Then the parallel algorithm is described and the chapter concludes with a summary.

4.1.1 Previous Work

Many polynomial-time sequential algorithms are known for solving the minimum-cost flow problem (see for example [4, 35]). The best known bound for the running time is $O((m + n \log n) m \log n)$ [52]. Better bounds are known in special situations where the capacities or the costs are integers of moderate size. The min-cost flow problem is considered to be apparently "hard" for parallelization, because it has been shown in [36] that even the max-flow problem is P-complete. This implies that the existence of a polylogarithmic-time parallel algorithm using a polynomial number of processors (NC algorithm) is highly unlikely for this problem. Recently, it has been further proved [58] that approximating the value of the min-cost maximum-flow to within a constant factor is also P-complete.

One way to cope with the apparent inherent sequentiality of these problems is to restrict them to special classes of networks. In [43], parallel algorithms for the maximum flow problem on planar networks have been presented which work either in $O(\log^3 n)$ time using $O(n^4)$ processors or in $O(\log^2 n)$ time using $O(n^6)$ processors. These algorithms have been later improved by [47], to $O(\log^3 n)$ time using $O(n^{1.5})$ processors. Contrasted with this, to the best of our knowledge, there are no known published NC algorithms for the max-flow min-cost problem on restricted classes of networks. Even in the sequential context, there has been at best only sporadic work on this problem. This is perhaps not so surprising, considering the polynomial-time sequential complexity of the problem. Bein, Brucker and Tamir [11] were the first to compute the minimum cost of the maximum flow on trees and series-parallel networks.

For the special case of zero lower bounds, they observe that a simple greedy strategy works, resulting in a time bound of $O(nm + m \log m)$.

Two-Terminal Series-Parallel (TTSP) networks are a subclass of planar networks and can be defined recursively by the following rules.

1. $G = (\{s, t\}, \{(s, t)\})$, is a TTSP network. The vertices s and t are called the terminals.

   Let $G_1 = (V_1, E_1)$, with source s_1 and sink t_1, and $G_2 = (V_2, E_2)$ with source s_2 and sink t_2 be TTSP networks.

2. The network G_p formed from G_1 and G_2 by the *parallel composition operation* which identifies s_1 with s_2 and t_1 with t_2 is TTSP with source $s_1 = s_2$ and sink $t_1 = t_2$.

3. The network G_s formed from G_1 and G_2 by the *series composition operation* which identifies t_1 and s_1, is TTSP with source s_1 and sink t_2.

A network G is said to be *series-parallel* if there exist two vertices s and t such that G is a TTSP network with terminals s and t. In [14, 15], Booth and Tarjan present several improved algorithms for the general min-cost flow problem with lower bounds on the edge flows for series-parallel networks. Their first algorithm runs in $O(m \log m)$ time and $O(m)$ space if only the cost of a minimum cost flow is needed or $O(m \log \log m)$ space if the optimal flow distribution is needed. Her second algorithm for finding the actual flow distribution runs in $O(m \log m \log \log m)$ time but uses only $O(m)$ space. The problems of finding a linear-space algorithm with $O(m \log m)$ running time and obtaining a fast parallel algorithm have been left open in [14, 15].

These algorithms work bottom-up on the *decomposition tree* representation of a series-parallel network. A *decomposition tree* of a series parallel network is a binary tree in which the leaves represent the edges of the network and each internal node represents either a series or a parallel composition of the networks represented by its subtrees. In Figure 4.1, we show a series-parallel network and its decomposition tree.

The decomposition tree of a series-parallel network can be obtained in $O(m+n)$ time sequentially [59]. The best algorithm for recognizing a series-parallel network and obtaining a decomposition tree for it has been developed by Eppstein [29] which runs in $O(\log^2 n)$ time on the EREW PRAM with $C(m,n)$ processors; here $C(m,n)$ is the number of processors required to compute connected components of a graph in logarithmic time. The best known bound for $C(m,n)$ is $O(m\alpha(m,n)/\log n)$ [22].

4.1.2 Definitions of Max-Flow and Min-Cost Max-Flow Problems

Let $G=(V,E)$ be a directed network with two distinguished vertices s and t called the *source* and the *sink* respectively. For each $e=(u,v)\in E$, we define *source* $s(e)=u$ and *sink* $t(e)=v$. We always denote the number of vertices in G by n and the number of edges by m. With every edge $e\in E$ are associated three real-valued functions, a *lower bound* $l(e)$, a *capacity* $u(e)$, and a *cost* $c(e)$. A *flow* on a network is a real-valued function f on the edges satisfying the following constraints:

capacity constraints,

$$l(e)\le f(e)\le u(e) \text{ for all } e\in E, \text{and}$$

conservation constraints,

$$\sum_{e\in E,\ t(e)=v} f(e)=\sum_{e\in E,\ s(e)=v} f(e) \text{ for all } v\in V-\{s,t\}.$$

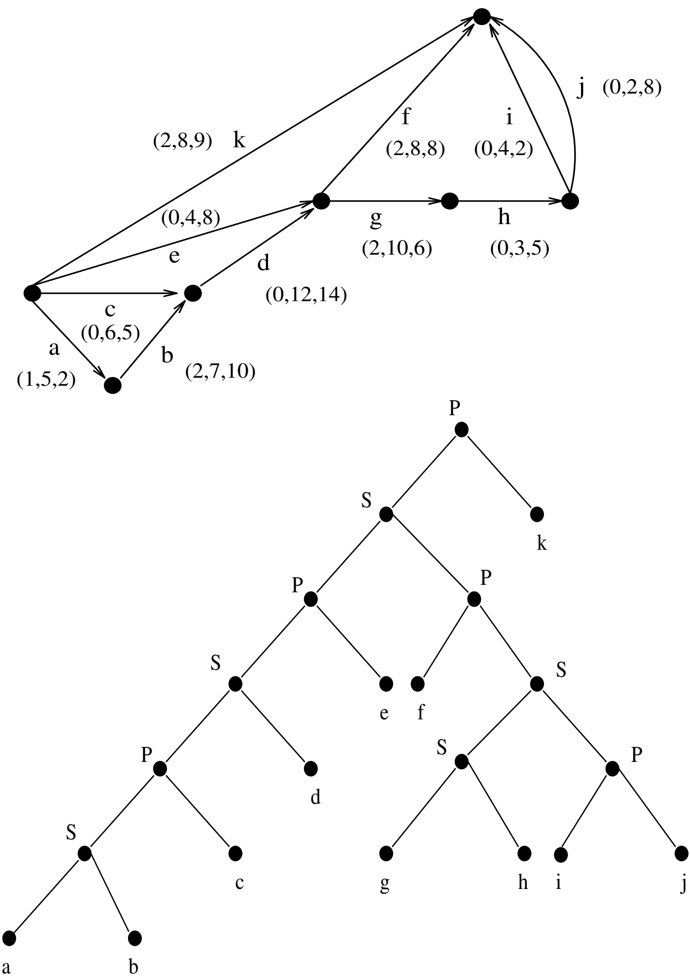

Figure 4.1.: A series-parallel network and its decomposition tree. The edges are labeled by 3-tuples $(l(e), u(e), c(e))$.

The value of the flow f is $\sum_{e \in E,\ t(e)=t} f(e)$, and the cost of f is $\sum_{e \in E} c(e)f(e)$. A flow is *maximum* if it has the maximum possible value. The *max-flow problem* is that of finding the maximum-flow from s to t. The *minimum-cost flow problem* is to find the minimum cost of a given feasible flow, among all flows with the same value. In particular, the *minimum-cost maximum-flow* problem is that of computing the minimum cost of the maximum flow from s to t.

4.2 Min-Cost Flow on Series-Parallel Networks

4.2.1 Flow Feasibility

We can compute the maximum flow value and the minimum flow value on a series-parallel network G in $O(m)$ time sequentially using the decomposition tree. If the minimum flow value is greater than the maximum flow value there is no feasible flow. Let the maximum flow value be *maxval(G)* and the minimum flow value be *minval(G)*. Then the maximum and the minimum flow in G can be computed from the decomposition tree bottom-up, by the following equations.

base network Let $G = (\{s,t\},\{e\})$. Then $minval(G) = l(e)$, $maxval(G) = u(e)$.

Let G_1 and G_2 be series-parallel. Let their parallel composition be G_p and the series composition be G_s.

parallel composition

$$\begin{aligned} minval(G_p) &= minval(G_1) + minval(G_2) \\ maxval(G_p) &= maxval(G_1) + maxval(G_2) \end{aligned}$$

series composition

$$
\begin{aligned}
minval(G_s) &= max\{minval(G_1), minval(G_2)\} \\
maxval(G_s) &= min\{maxval(G_1), maxval(G_2)\} \\
\text{If } minval(G_s) &> maxval(G_s) \text{ then no flow is feasible.}
\end{aligned}
$$

The maximum and the minimum flow can be computed in parallel on an EREW PRAM using the tree contraction technique [1]. Briefly, the tree contraction approach reduces a binary tree by means of repeated *shunt* operations to a single node. The *shunt* operation involves pruning a leaf, eliminating its parent and setting its sibling's new parent to its previous grandparent. Some familiarity with the tree contraction technique is assumed, and the interested reader is directed to many excellent references dealing with tree contraction [1, 41, 48, 49]. We describe below, how to compute the maximum flow in G by providing the transformations associated with the shunt operation in the contexts of series and parallel composition operations.

With each parallel composition node in the decomposition tree, we associate the operation of *addition*, and with each series composition node we associate the operation of *minimum*. Each node (x) has a value $val(x)$ which is the maximum flow in the subgraph corresponding to the decomposition subtree rooted at x and a 2-tuple, say (a, b), associated with it. Then the actual value passed to the parent of x is $min\{val(x) + a, b\}$. Initially, each node is labeled with $(0, \infty)$. Each leaf node of the decomposition tree corresponding to an edge $e \in E$ is assigned $val(e) = u(e)$, the maximum flow value on that edge. The $val(y)$ for each internal node y is undefined at the start of the algorithm.

Figure 4.2 shows a portion of a decomposition tree, illustrating the shunt operation on the leaf u composed in parallel with v at node w (shown as P). The value of the node v is $min\{val(v) + v_p, v_s\}$ where $val(v)$ is unknown. The value of the node u is $val(u)$. The value computed at node w is thus $val(u) + min\{val(v) + v_p, v_s\}$ which can be rewritten as $min\{val(u) + val(v) + v_p, val(u) + v_s\}$. In other words, the 2-tuple associated now with v is $(val(u) + v_p, val(u) + v_s)$. But we have the tuple (w_p, w_s) for w which should be combined with this, to form a new 2-tuple (z_p, z_s) for v such that $min\{val(v) + z_p, z_s\}$ is the same as (w_p, w_s) combined with $min\{val(u) + val(v) + v_p, val(u) + v_s\}$. A little manipulation yields (z_p, z_s) as equal to $(v_p + val(u) + w_p, min\{w_s, v_s + val(u) + w_p\})$. In the special case when $(w_p, w_s) = (0, \infty)$, we have $(z_p, z_s) = (val(u) + v_p, val(u) + v_s)$. If u is the right child, we can derive a similar expression. Also if the node w represents a series operation, then Figure 4.3 shows the result after shunting u. For computing the minimum flow the two operations are *addition* and *maximum*. There we assign $(0, -\infty)$ to each internal node. We can carry out the analysis for the series operation just the same as above. Hence, we have the following theorem using the results from [1].

Theorem 4.2.1 *Let $G = (V, E)$ be a series-parallel network with n vertices and m edges. If the decomposition tree for G is given, then the maximum flow, the minimum flow and feasibility of flow can be determined in $O(\log m)$ time using $O(m/\log m)$ processors on an EREW PRAM.*

4.2.2 Min-Cost Flow Value and Flow List Computations

The value of a maximum flow can be computed knowing only the maximum flow values of the constituent networks. That is not true, however, of the minimum cost

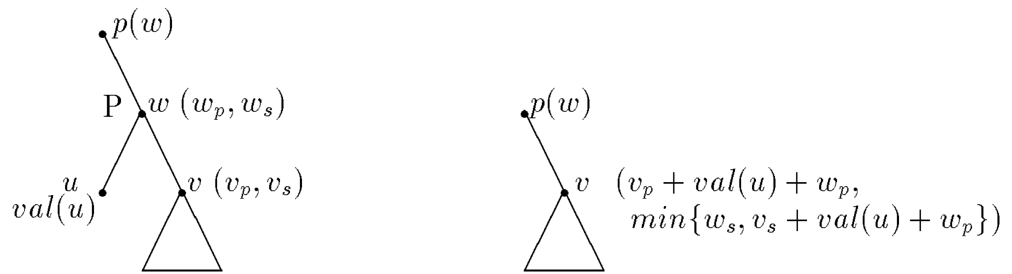

Figure 4.2. SHUNT on a parallel composition node for computing the max-flow.

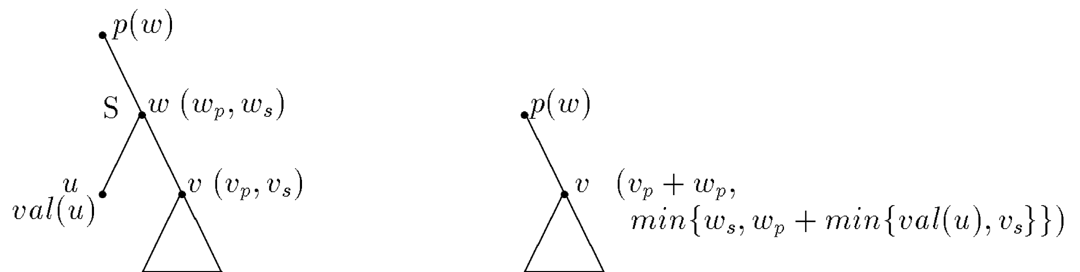

Figure 4.3. SHUNT on a series composition node for computing the max-flow.

of a maximum flow. To compute the minimum cost we need, in general, to know the costs of minimum cost flows of all possible values in the constituent networks. To represent this Booth and Tarjan introduced a *flow list*. A flow list is a list of pairs, sorted with respect to cost, representing the flow that can be pushed on the augmenting paths and the cost associated with it. Formally a flow list is a list of pairs $(l_0, c_0); (u_1, c_1), (u_2, c_2), \ldots, (u_k, c_k)$ with the following properties:

1. $u_i > 0$ for $1 \leq i \leq k$.

2. $c_i \leq c_{i+1}$ for $1 \leq i < k$.

3. l_0 is the value of the minimum flow, c_0 is the minimum cost of l_0 units of flow and $u_0 = l_0 + \sum_{i=1}^{k} u_i$ is the value of a maximum flow.

4. for any flow value $x, l_0 \leq x \leq u_0$, where $x = l_0 + \sum_{i=1}^{j-1} u_i + \alpha u_j$ with $0 < \alpha \leq 1$, the cost of a minimum-cost flow of value x is $c_0 + \sum_{i=1}^{j-1} c_i u_i + \alpha u_j c_j$.

The first pair is a *special* pair and the remaining pairs are *normal* pairs. The first component of a normal pair is the flow capacity and the second component is the cost. These pairs give the flow that can be pushed and the cost per unit flow for that flow, with each pair corresponding to an augmenting path in the network. Typically, the pairs with the same cost in a flow list are combined to reduce the size of the flow list. For general networks the flow list can be exponential in size. For a series-parallel network, however, the flow list has size $O(m)$ [14, 15].

A flow list for an arbitrary network can be constructed by first computing a minimum flow of minimum cost using any minimum-cost flow algorithm. This gives

the special pair (l_0, c_0). Each subsequent pair is computed by finding a minimum-cost augmenting path in the residual network for the current flow. To construct a flow list for a series-parallel network G however, it is more efficient to build it by working on a decomposition tree of G bottom-up instead of using the minimum-cost augmenting path method. The following recursive formulation captures the essence of the algorithm.

base graph $G = (\{s,t\},\{e\})$. The flow list is $(l(e), c(e)l(e)); (u(e) - l(e), c(e))$ if $u(e) \geq l(e)$, else there is no feasible flow.

Let G_1 and G_2 be two series-parallel networks with flow lists $L_1 = (l'_0, c'_0); \bar{L}_1$ and $L_2 = (l''_0, c''_0); \bar{L}_2$ respectively.

parallel composition The flow list is $L_p = (l'_0 + l''_0, c'_0 + c''_0); \bar{L}_p$ where $\bar{L}_p$ is formed by merging lists $\bar{L}_1$ and $\bar{L}_2$, ordering the pairs in nondecreasing order of costs, and combining pairs of equal cost by adding their capacities.

series composition Let the flow list be $L_s = (l^s_0, c^s_0); \bar{L}_s$. First we have to compute the special pair. If $l'_0 = l''_0$ then $l^s_0 = l'_0$ and $c^s_0 = c'_0 + c''_0$. Otherwise we have to increase the minimum flow in the graph with smaller minimum flow to be equal to the minimum flow in the other graph. This may lead to deletion of some pairs in the flow list as they will be used to push the flow. Next we have to compute the remaining pairs of $\bar{L}_s$. Remove the first pair (u', c') from $\bar{L}_1$and the first pair (u'', c'') from $\bar{L}_2$. Add the pair $(min\{u', u''\}, c' + c'')$ to $\bar{L}_s$. If $u' > u''$ add the pair $(u' - u'', c')$ to the front of $\bar{L}_1$; if $u'' > u'$ add the pair $(u'' - u', c'')$ to the front of $\bar{L}_2$. Repeat this step until one of the two lists is empty.

The above description of the algorithm closely follows that of Booth and Tarjan [14, 15]. In Figure 4.4, a decomposition tree for the series-parallel network in Figure 4.1 is shown along with all the intermediate flow lists. A straightforward implementation using linked lists to represent the flow lists, can compute a flow list for an m-edge series-parallel network using $O(m^2)$ amount of work. Booth and Tarjan, however, obtained an $O(m \log m)$ time bound by using a clever representation of the flow lists using finger search trees. But, this is where the implementation of their algorithm becomes involved which in turn makes the analysis complex. Here we present a method which completely avoids the use of finger search trees. Although we describe our method in the parallel setting, it can be used profitably in the sequential setting as well. If we have the flow list for a graph then the minimum cost of a maximum flow (rather, of any flow) can be computed easily. From the flow list and a valid flow value k, we can compute the cost of the min-cost flow of value k as follows: Given flow list $(l_0, c_0); (u_1, c_1), (u_2, c_2), \ldots, (u_l, c_l)$ if $1 \leq j \leq l$ and $0 < \alpha \leq 1$ are such that $l_0 + \sum_{i=1}^{j-1} u_i + \alpha u_j = k$ then the cost of the min-cost flow of capacity k is $c_0 + \sum_{i=1}^{j-1} u_i c_i + \alpha u_j c_j$.

Now we introduce two operators: the parallel flow list operator $\odot_p$ and the series flow list operator $\odot_s$. These binary operators work on two flow lists and produce a new flow list corresponding to parallel or series composition operation. The identity element for the operator $\odot_p$ is the flow list $((0, 0); \phi)$ where ϕ represents an empty list. The identity element for the operator $\odot_s$ is the flow list $((-\infty, 0); (\infty, 0))$. If there are no lower bounds on the flows on the edges, then the identity elements are ϕ and $(\infty, 0)$. In the following lemma we show some of the properties of these operators.

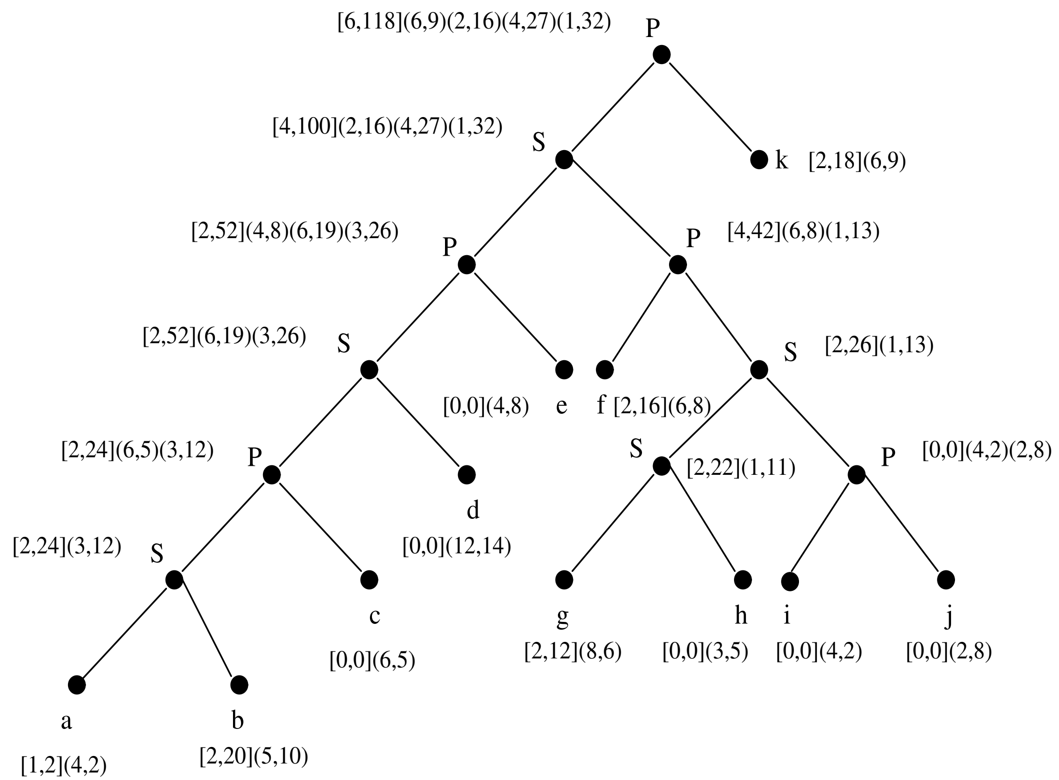

Figure 4.4.: Flow lists for the series-parallel network in Figure 4 (special pairs are shown in square brackets).

Lemma 4.2.2 *Series and parallel flow list operators have the following properties:*

1. *$\odot_p$ is commutative and associative,*

2. *$\odot_s$ is commutative and associative, and*

3. *$\odot_p$ distributes over $\odot_s$ but not vice versa.*

Proof: Both parallel and series flow list operations are commutative by definition. Computation of the special pair for $\odot_p$ operator involves only addition which is associative. Combining the two flow lists uses straightforward merging which is also associative. Hence $\odot_p$ is associative. The series composition results in excess capacity of one of the constituent graphs being removed. Hence if we are combining three graphs in series the result depends only on which graph has the minimum capacity. Similarly computation of the special pair involves increasing the minimum flow to be the same as the largest minimum flow among the three graphs. Hence the $\odot_s$ operator is also associative.

For distributivity, let us consider $(A \odot_s (B \odot_p C))$. If $\odot_s$ distributes over $\odot_p$ then the resulting flow list should be the same for $((A \odot_s B) \odot_p (A \odot_s C))$. Suppose B had more capacity than C. Then we could push flow equal to the smaller of the capacity of A and C. But in the distributed expression we can push more flow. Hence the $\odot_s$ operator does not distribute over $\odot_p$ operator. The other case is similar.

Now, let us consider $(A \odot_p (B \odot_s C))$. If $\odot_p$ distributes over $\odot_s$ then the resulting flow list should be the same for $((A \odot_p B) \odot_s (A \odot_p C))$. The flow that can be pushed is the flow in A plus the minimum of the flow that can be pushed through B and C. If the distributed expression is considered, the flow is the the minimum of

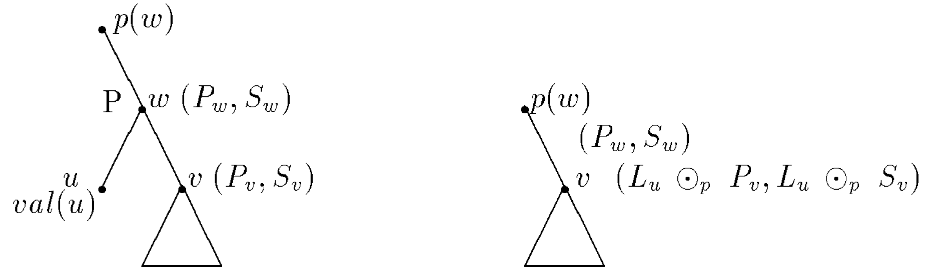

Figure 4.5.: SHUNT on a parallel composition node for computing the min-cost flow.

the flow through A and B and the flow through A and C which is the same as before. Hence the operator $\odot_p$ distributes over $\odot_s$. □

Our parallel algorithm for the min-cost flow problem follows a modified form of the tree contraction technique, where a shunt operation takes more than a constant amount of time because of the presence of flow lists of size $O(m)$. We associate a value with each node of the decomposition tree which is the flow list of the network rooted at that node. For each leaf node, corresponding to an edge e in the original network, the flow list is $((l(e), c(e)l(e)); (u(e) - l(e), c(e)))$, assuming flow is feasible. With each node we also associate a 2-tuple whose components are flow lists. Suppose for a node x, the flow list is L_x and the 2-tuple is (P_x, S_x). Then the flow list passed to the parent is $((L_x \odot_p P_x) \odot_s S_x)$. Initially, label each internal node of the decomposition tree by the 2-tuple of identity elements for the operators $\odot_p$, $\odot_s$, i.e., $(((0,0); \phi), ((-\infty, 0); (\infty, 0)))$.

Now consider the tree in Figure 4.5, where the shunting is being done on node u. Then the value computed at node w is $(L_u \odot_p ((L_v \odot_p P_v) \odot_s S_v))$ which by Lemma 4.2.2 can be rewritten as $((L_v \odot_p (L_u \odot_p P_v)) \odot_s (L_u \odot_p S_v))$. Hence the new 2-tuple should be $(L_u \odot_p P_v, L_u \odot_p S_v)$. This 2-tuple has to be combined

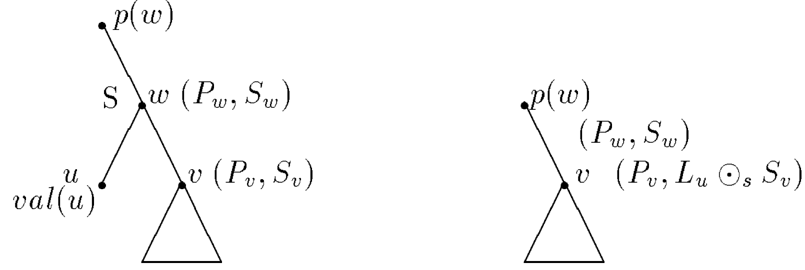

Figure 4.6.: SHUNT on a series composition node for computing the min-cost flow.

with the 2-tuple (P_w, S_w) which is discussed below in general terms. The case when u is the right son can be handled similarly. The case when the node w is a series composition node is also similar and the result is shown in Figure 4.6.

We show here, how to combine two 2-tuples (P_x, S_x) and (P_y, S_y) where y is the parent of x such that L_x when applied to the resulting 2-tuple gives the same result as when applied to the two 2-tuples in succession. Using Lemma 4.2.2 we have:

$((((L_x \odot_p \ P_x) \odot_s \ S_x) \odot_p \ P_y) \odot_s \ S_y) \Rightarrow$

$((((L_x \odot_p \ P_x) \odot_p \ P_y) \odot_s \ (S_x \odot_p \ P_y)) \odot_s \ S_y) \Rightarrow$

$((L_x \odot_p \ (P_x \odot_p \ P_y)) \odot_s \ ((S_x \odot_p \ P_y) \odot_s \ S_y)) \Rightarrow$

L_x applied to $(P_x \odot_p \ P_y, (S_x \odot_p \ P_y) \odot_s \ S_y)$

Hence, we have the following theorem that is analogous to Theorem 4.2.1.

Theorem 4.2.3 *Given a series-parallel network with the decomposition tree the minimum cost maximum flow value and the flow list for the graph can be computed in* $O(\log m)$ *iterations using tree contraction.*

The time needed to perform an iteration of the algorithm is in the order of the time needed to compute the series or parallel composition of all the flow lists processed in any iteration. In the next subsection, we will look at this problem in more detail.

4.2.3 Obtaining Flow Lists using Multiselection and Merging

Suppose L_1 and L_2 are two flow lists of size m_1 and m_2 respectively. Let $m_1 \leq m_2$, and let $L_1 = (l'_0, c'_0); \bar{L}_1$ and $L_2 = (l''_0, c''_0); \bar{L}_2$. Assume that the flow lists are stored in arrays; later we will say how to maintain these flow lists. We prove the following lemmas.

Lemma 4.2.4 *Given two flow lists with m_1 and m_2 pairs, the composition of these two flow lists for the parallel composition operation can be done in $O(\log(m_1 + m_2))$ time using $O((m_1 + m_2)/(\log(m_1 + m_2)))$ processors on the EREW PRAM.*

Proof. The special pair can be computed in $O(1)$ time by a single processor. The rest of the two flow lists can be merged in $O(\log(m_1 + m_2))$ time using $O((m_1 + m_2)/(\log(m_1 + m_2)))$ processors on the EREW PRAM. Any optimal merging algorithm for the EREW PRAM can be used to accomplish this step [13, 27, 38]. After merging there may be up to two pairs with the same cost. These two pairs can be combined easily. We can then use the parallel prefix [46] algorithm to compact the array containing the flow list. □

Lemma 4.2.5 *Given two flow lists with m_1 and m_2 pairs, the series composition of these two flow lists can be done in $O(\log(m_1+m_2))$ time using $O((m_1+m_2)/(\log(m_1+m_2)))$ processors on the EREW PRAM.*

Proof. We provide the proof in two parts: (I) computing the special pair and (II) merging the remaining flow lists.

(I) Computing the special pair: If the minimum flow values are the same then computing the special pair is trivial and can be done in $O(1)$ time. Otherwise, without loss of generality, let $l'_0 > l''_0$ then we have to increase the flow in the second subgraph. This can be done in parallel as follows.

I.1 Compute in parallel, the prefix-sums on capacity u_i and cost $u_i c_i$ for the flow list L_2, i.e., $\sum_{i=1}^{1 \leq k \leq m_2} u''_i$ and $\sum_{i=1}^{1 \leq k \leq m_2} u''_i c''_i$.

I.2 Determine the index j of the pair and a fraction α such that $l'_0 = l''_0 + \sum_{i=1}^{j-1} u''_i + \alpha u''_j$, where $0 < \alpha \leq 1$.

I.3 Compute the cost as $c^s_0 = c'_0 + c''_0 + \sum_{i=1}^{j-1} u''_i c''_i + \alpha u''_j c''_j$.

I.4 Compress $\bar{L}_2$ to $((1-\alpha)u''_j, c''_j) \ldots (u''_k, c''_k)$ using parallel prefix.

All of the above can be done in $O(\log m_2)$ time using $O(m_2/\log m_2)$ processors on the EREW PRAM.

(II) Merging the remaining flow lists: This is a special type of merge operation where the merger of two pairs can result in at most two pairs with one added back to one of the flow lists being merged. The pair added back is the one with excess flow capacity. Sequentially the series operation can be accomplished in $O(m_1 + m_2)$ time by modifying the standard merging algorithm. In the parallel setting, the key issue is how to partition the two flow lists into sublists such that these can be merged independently and are computationally balanced.

The idea is to partition the flow lists using the flow capacities and not just the cost. Note that the flow lists are ordered only by cost and not by capacity. We provide a high-level description of how to perform the series operation in parallel in the following steps.

II.1 Compute the prefix-sums on the flow capacities in the flow lists. (if not computed in Step I.1).

II.2 Partition the flow lists based on the cumulative flow capacity such that the corresponding sublists represent approximately equal flow capacity.

II.3 Make the flow capacity in corresponding sublists equal by splitting pairs in the adjacent sublists.

II.4 Merge the sublists independently using the sequential series operation.

Step 1 can be done in $O(\log(m_1 + m_2))$ time in parallel. Suppose the total flow capacity is larger in one of the flow lists. Then we can discard the excess capacity in that list except one extra pair. The reason to keep one extra pair will become clear as we describe the rest of the steps. The prefix sums of the flow capacities for the two flow lists obviously form two increasing sequences. Let these two sequences be A (for L_1) and B (for L_2). To perform Step 2 we will use these two sequences as follows.

II.2a Find the $i\lfloor\log(m_1+m_2)\rfloor$th ranked element in the union of the two sequences A and B, $i = 1, 2, \ldots, j (= (m_1 + m_2)/(\lfloor\log(m_1 + m_2)\rfloor))$. Let the output be two arrays $R_A[1..j]$ and $R_B[1..j]$, where $R_A[i] \neq 0$ implies that $A[R_A[i]]$ is the $(i\lfloor\log(m_1 + m_2)\rfloor)$th element and $R_B[i] \neq 0$ implies that $B[R_B[i]]$ is the $(i\lfloor\log(m_1 + m_2)\rfloor)$th element.

II.2b Let $R_A[0] = R_B[0] = 0$

for $i = 1, \ldots, j$ do

if $R_A[i] = 0$ then $R_A[i] = i * \lfloor \log(m_1 + m_2) \rfloor - R_B[i]$

else $R_B[i] = i * \lfloor \log(m_1 + m_2) \rfloor - R_A[i]$

Consider the flow sublists corresponding to the subsequences $A[R_A[i-1]+1]..A[R_A[i]]$ and $B[R_B[i-1]+1]..B[R_B[i]]$. The total flow capacity in one of the flow sublists may be smaller (the sequence which had $R_A[i]$ or $R_B[i]$ equal to zero at the end of Step II.2a). Without loss of generality, let this be the sublist of L_1 corresponding to A. Then we can split the $(R_A[i]+1)$th pair in the flow list L_1 into two pairs such that the flow capacity of sublist corresponding to A plus the new pair is exactly equal to that of the sublist corresponding to B. Note that we never need to split more than one pair else the value of $R_B[i]$ is not correct. Further, if we have $O((m_1+m_2)/\log(m_1+m_2))$ partitions of the flow lists L_1 and L_2, the sizes of the flow lists will increase by at most $O((m_1 + m_2)/\log(m_1 + m_2))$. Using parallel prefix, we can perform Step II.3 and form new flow sublists which are exactly balanced in flow capacity. Then we can assign one processor to merge each flow sublist pair independently using the usual sequential algorithm.

The only other step we need to explain is Step II.2a. The ranking in Step II.2a can be done by merging the sequences A and B and then computing the positions. However, we can use a simpler approach by using the multiselection algorithm [27] described in Chapter 2 by which we can avoid merging the two sequences. Given two ordered sets A and B of sizes m and n, where $m \leq n$, let us say we want to perform r selections where the ranks are $\{K_1, \ldots, K_r\}$. Our multiselection algorithm solves

the problem in $O(\log m + \log r)$ time using r processors on the EREW PRAM and is used as a subroutine for developing an optimal merging algorithm on the EREW PRAM. The multiselection algorithm can be used to compute Step II.2a by setting $m = m_1$, $n = m_2$ and $r = (m_1 + m_2)/\log(m_1 + m_2)$. □

Remark 1. The sum of the sizes of all flow lists is initially at most m pairs (not counting the special pairs). Each parallel or series composition operation reduces or maintains the number of pairs. The various flow lists can be kept in an array of size $O(m)$ and in each iteration, we can keep track of the locations of the flow lists with appropriate bookkeeping. One way is to augment the decomposition tree such that each node also has pointers to the location and the size of its associated flow list. Hence each iteration of the tree contraction can be done in $O(\log m)$ time using $O(m)$ processors and $O(m)$ space. Thus the MCMF value can be computed in $O(\log^2 m)$ time using $O(m)$ processors on the EREW PRAM.

Remark 2. Using Brent's scheduling the number of processors can be reduced. The time taken for an iteration of the tree contraction is $max\ O(\log(m_{i_1} + m_{i_2}))$, $1 \leq i \leq k$, where k is the number of nodes in the decomposition tree being shunted in that iteration. We know that $\sum_{i=1}^{k} m_{i_1} + m_{i_2} = O(m)$. A simple strategy is to allocate $\lceil m_i/\log m \rceil$ processors to a flow list of size m_i in each iteration of the tree-contraction, instead of $\lceil m_i/\log m_i \rceil$ processors. Hence the total number of processors is bounded by $O(\sum_{i=1}^{k}(m_{i_1}+m_{i_2})/(\log m))$ and the time by $O(\log m)$ in each iteration. Hence we can reduce the number of processors to $O(m/\log m)$ without increasing the asymptotic running time, giving rise to our main theorem:

Theorem 4.2.6 *Given a series-parallel network G with n vertices and m edges, the minimum cost of any flow can be computed in $O(\log^2 m)$ time using $O(m/\log m)$ processors on the EREW PRAM from the decomposition tree for G.*

4.3 Summary

In this chapter, we have presented an efficient parallel algorithm for finding the cost of the minimum cost flow problem of a series-parallel network. The parallel algorithms that we developed in Chapter 1 for multiselection and merging form an important building block for our parallel min-cost flow algorithm. Given a decomposition tree, our algorithm runs in $O(\log^2 m)$ time using $O(m/\log m)$ processors on the EREW PRAM. The salient features of our algorithm are summarized in the following two paragraphs.

The spirit of our algorithm is based on the sequential algorithm by Booth and Tarjan [14, 15] but the efficiency is due to a careful application of the tree contraction technique [1]. The time-processor product of our algorithm matches the best known sequential time-complexity for min-cost flow on series-parallel networks due to Booth and Tarjan [14, 15].

The algorithm computes a *flow list* at each internal node of the decomposition tree which represents the costs of minimum-cost flows of all possible values for the subtree rooted at that node. The size of the flow list is $O(m)$ for a series-parallel network. The bottom-up procedure for obtaining the flow list for a node (from the flow lists of its subtrees) can take as much as $O(m)$ work resulting in an $O(m^2)$ work algorithm for the problem. To avoid this, Booth and Tarjan use an elaborate finger

search tree representation of the flow lists and achieve an $O(m \log m)$ running time for their algorithm. Our parallel algorithm does not use the finger search tree and it is conceptually simpler. We believe that the sequential simulation of our parallel algorithm may perform better than Booth and Tarjan's in practice.

Chapter 5

CONCLUSION AND OPEN PROBLEMS

5.1 Conclusions

The technique of chaining plays an important role in the design of parallel algorithms for multiselection and multisearch. The technique of chaining has surprisingly been applied to only a few problems [6, 8, 19, 63]. To summarize, a *chain* is an ordered sequence of operations that can be processed together as long as the individual operations are expected to progress identically. This technique is applicable only when the test for identical progress of operations can be determined in constant time. Otherwise, when the individual operations cannot progress together, the chain is split into two and processed, either independently or using pipelining. Another technique that we found useful is partitioning. The technique of *partitioning* consists of breaking up a given problem into independent subproblems of equal sizes and then solving the subproblems in parallel. The main problem is in finding the proper partitioning. We saw that multiselection is an useful technique for partitioning. On many occasions we also made use the technique of pipelining. *Pipelining* is the process of breaking up a operation into a sequence of suboperations such that once the first suboperation is finished, the sequence corresponding to a new operation can begin and proceed at the same rate as the first operation. Pipelining by itself may not lead to fast parallel

algorithms but is combined with the above two techniques. The other two techniques that were found useful are *accelerated cascading* and *tree contraction*.

In Chapter 2, we presented an $O(\log m + \log r)$-time parallel algorithm using r processors to perform r multiselections in two sorted arrays, of sizes m and n with $m \leq n$, on the EREW PRAM. The multiselection algorithm is based on a novel application of the technique of chaining. A careful analysis of our parallel algorithm shows that the total number of comparisons performed is $O(r \log(m/r))$, when $r \leq m$, and $O(m \log(r/m))$, when $m \leq r$, which matches the information-theoretic lower bound on the problem of multiselection.

A natural way to merge two sorted arrays in parallel is to partition the arrays into equal size subarrays such that independent merging of the subarrays solves the original problem. Thus, the problem of merging in parallel can be solved if we can efficiently select the correct elements to partition the arrays. Our multiselection algorithm was used to select the correct elements, leading to a simple and optimal algorithm for merging on the EREW PRAM requiring $O(\log(m+n))$ time and $O(m+n)$ work. The merging algorithm does not move or copy any data during the partitioning phase and the number of comparisons is within lower order terms of the minimum possible, even by a sequential algorithm. Hagerup and Rüb [38] have given a parallel merging algorithm on the EREW PRAM that has the minimum number of comparisons for any parallel merging algorithm for the EREW PRAM. Our merging algorithm has the same number of comparisons when $m = \Theta(n)$. When the size of the two lists differs significantly, however, our merging algorithm performs asymptotically fewer comparisons than Hagerup and Rüb's algorithm.

In Chapter 3, we have presented parallel algorithms for search and multisearch in sorted matrices. First we presented an $O(\log n)$-time parallel algorithm, based on the technique of partitioning, using $O(n)$ work for searching and ranking in an $n \times n$ matrix with sorted rows and sorted columns. This algorithm combined with the technique of chaining then served as a basis for the design of a parallel algorithm for multisearch in the $n \times n$ matrix with sorted rows and sorted columns. The multisearch algorithm for r searches runs in $O(\log n + \log r)$ time and $O(rn)$ work, which matches the best known sequential time. These algorithms are quite different from their sequential counterparts. Then we also considered the use of chaining to design parallel algorithm for search in $X + Y$ where both X and Y are sorted vectors implicitly representing a sorted matrix.

The second part of the Chapter 3 dealt with search and multisearch in $m \times n$ matrices with sorted columns or equivalently m sorted arrays of size n each. Since there is no relative order of the elements in the rows of such a matrix the technique of partitioning is not applicable. Furthermore the parallel as well as the sequential algorithm are sensitive to the rank of the search-element. We proposed a cost-optimal algorithm that runs in $O(\log m \log\log m)$ time for small elements (with rank $\leq m$) and in time $O(\log m \ \log\log m \ \log(k/m)$ for large elements. This algorithm uses the technique of accelerated cascading. Then we presented a sequential algorithm for multisearch in a matrix with sorted columns as a prelude to the parallel algorithm. The sequential algorithm is inspired by the parallel technique of chaining. The parallel multisearch algorithm follows this sequential algorithm and has a nontrivial dependence not only on the ranks of the search-elements but also on the number of

search-elements. Finally we show how to adapt ideas from Bentley and Yao's [12] classic paper on sequential unbounded searching and improve the search algorithm for small elements to run in $O(\log m \log^*(\log m))$ time with optimal work and for large elements in $O(\log m \log^*(\log m) \log(k/m))$ time with optimal work.

In Chapter 4, the problem of finding the minimum cost of a feasible flow in directed series-parallel networks with real-valued lower and upper bounds for the flows on edges is addressed. The best known sequential algorithm for computing the minimum cost of a feasible flow on series-parallel networks, given by Booth and Tarjan [14, 15], takes $O(m \log m)$ time and $O(m)$ space for a network with m edges. Their algorithm works bottom-up on the decomposition tree representation of a series-parallel network. We develop, for the first time in [40], an efficient and fast parallel algorithm to compute the minimum cost of a feasible flow on directed series-parallel networks solving, in part, a problem posed by Booth [14]. Our algorithm takes $O(\log^2 m)$ time using $O(m/\log m)$ processors on an EREW PRAM and it is optimal with respect to Booth and Tarjan's algorithm, if the decomposition tree is given. The algorithm owes its efficiency to the tree contraction technique, the multiselection and merging algorithms, and using simple data structures for flow list manipulations as opposed to finger search trees used by Booth and Tarjan. Indeed, the sequential simulation of our algorithm gives a new sequential algorithm for computing the cost of a min-cost flow on a series-parallel network that is as efficient as Booth and Tarjan's [14, 15] algorithm but uses simpler data structures.

5.2 Open Problems

We will now discuss some open problems and further directions for research that arise from our work.

We have given an optimal merging algorithm on the EREW PRAM. This algorithm operates on two sorted arrays. Another direction for research is to investigate the case when the two lists are given as height-balanced trees. Brown and Tarjan [18] have given an $O(m \log(n/m))$-time sequential algorithm to merge two lists represented as height-balanced trees, which matches the the information-theoretic lower bound. Note that the this lower bound can be matched only if the input sorted lists are represented as height-balanced trees. Paul, Vishkin and Wagner [53] have given a parallel insertion algorithm for 2-3 trees that can be used to merge two lists represented as 2-3 trees in $O(\log m + \log n)$ time using $O(m)$ processors on an EREW PRAM. Obviously, their algorithm is not optimal as the total work done is $O(m \log n + m \log m)$. The parallel insertion algorithm is preceded by a parallel search that can be shown to take only $O(m \log(n/m))$ work by extending the analysis that we have presented for our multiselection algorithm [28]. The open problem is to design a parallel merging algorithm, for the case when the input is represented as height-balanced trees, that runs in $O(\log n)$ time but uses only $O(m \log(n/m))$ work. This would be a parallel analogue to the classic sequential result of Brown and Tarjan.

We considered search and multisearch in square matrices with sorted rows and sorted columns. Our search algorithm are efficient for non-square matrices as long as $m = \Theta(n)$. How about sorted matrices that are very thin and long? Frederickson and Johnson [32] have given an optimal sequential algorithm that requires $O(m \log(n/m))$

time using different ideas from the square case. In parallel, our algorithms for square matrices do not directly lead to optimal algorithms for non-square matrices. For the case of multisearch the best known sequential algorithm is $O(rn)$ for r multisearches in an $n \times n$ sorted matrix. There is no lower bound known for this problem except a trivial lower bound of $O(n)$. A very important case is the search for n elements, that is, $r = n$. Note that using the ideas from our parallel algorithm we can reduce the time for the first two steps to $O(n\sqrt{n})$ but the final step still requires $O(n^2)$ time. To improve the sequential algorithm should be of great interest since it has important ramifications for the knapsack problem [24, 31, 39].

Frederickson and Johnson have also given optimal sequential algorithms for the selection problem on sorted matrices [32, 34] that have the same sequential time-complexity as search. Sarnath and He [55] have presented a parallel algorithm for selection in an $n \times n$ matrix with sorted rows and sorted columns that runs in $O(\log n \log\log n \log^* n)$ time with $O(n \log\log n)$ work on the EREW PRAM. For the case of selection in matrices with sorted columns, Frederickson and Johnson's sequential algorithm can be parallelized easily to run in $O(\log n \log m \log\log m)$ worst-case time. It would be interesting to see if the time can be reduced similar to the Sarnath and He's parallel algorithm for matrices with sorted rows and sorted columns.

For the min-cost flow on series-parallel networks, we have left the computation of actual flow values efficiently as an open problem. It would also be interesting to see if we could extend the technique of tree contraction and the idea of maintaining flow lists to obtain min-cost flow algorithms to more complex classes of graphs. Finally, we would like to investigate if multiselection could also be applied to other problems.

REFERENCES

[1] Abrahamson, K., N. Dadoun, D. G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10:287–302, 1989.

[2] Aggarwal, A., M. M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987.

[3] Aggarwal, A., and J. Park. Notes on searching in multidimensional monotone arrays. In *Proceedings of 28th Conference on Foundations of Computer Science*, pages 497–512, 1988.

[4] Ahuja, R. K., T. L. Magnanti, and J. B. Orlin. Network flows. *SIAM Review*, 33(2):175–219, June 1991.

[5] Akl, S. G., and N. Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Transactions on Computers*, C-36(11):1367–1369, November 1987.

[6] Amir, A., and M. Farach. Adaptive dictionary matching. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 760–766, 1991.

[7] Anderson, R. J., E. W. Mayr, and M. K. Warmuth. Parallel approximation algorithms for bin packing. *Information and Computation*, 82:262–277, September 1989.

[8] Atallah, M. J., D. Z. Chen, and H. Wagner. An optimal parallel algorithm for the visibility of a simple polygon from a point. *Journal of the ACM*, 38(3):516–533, July 1991.

[9] Atallah, M. J., and S. R. Kosaraju. An efficient parallel algorithm for the row minima of a totally monotone matrix. *Journal of Algorithms*, 13:394–413, 1992.

[10] Batcher, K. E. Sorting networks and their applications. In *Proceedings of AFIPS Spring Joint Computer Conf.*, pages 307–314, 1968.

[11] Bein, W. W., P. Brucker, and A. Tamir. Minimum cost flow algorithms for series-parallel networks. *Discrete Applied Mathematics*, 10:117–124, 1985.

[12] Bentley, J. L., and A. C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5:82–87, 1976.

[13] Bilardi, G., and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. *SIAM Journal on Computing*, 18(2):216–228, April 1989.

[14] Booth, H. Fast algorithms on graphs and trees. Technical Report 90-76, Center for DIMACS, Rutgers University, December 1990. (also Ph.D. Thesis, Princeton University, 1990).

[15] Booth, H., and R. E. Tarjan. Finding the minimum-cost maximum flow in a series parallel network. *Journal of Algorithms*, 15:416–446, 1993.

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

[17] Brent, R. P. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[18] Brown, M. R., and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing*, 9(3):594–614, August 1980.

[19] Chen, D. Z. Efficient geometric algorithms in the EREW-PRAM. In *Proceedings of the 28th Annual Allerton Conference on Communication, Control, and Computing*, pages 818–827, 1990.

[20] Chen, D. Z. Efficient parallel binary search on sorted arrays. Technical Report 1009, Purdue University, Department of Computer Science, August 1990.

[21] Cole, R. J. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, August 1988.

[22] Cole, R., and U. Vishkin. Approximate parallel scheduling. II. Applications to logarithmic-time optimal parallel graph algorithms. *Information and Computation*, 92(1):1–47, May 1991.

[23] Cosnard, M., J. Duprat, and A. G. Ferreira. Known and new results in selection, sorting, and searching in sorted matrices and sorted $X+Y$. Technical Report LIP-IMAG 89-10, LIP - CNRS, Ecole Normale Supe′rieure de Lyon, France, 1989.

[24] Cosnard, M., J. Duprat, and A. G. Ferreira. The complexity of searching in $X+Y$ and other multisets. *Information Processing Letters*, 34:103–109, 1990.

[25] Cosnard, M., and A. G. Ferreira. Parallel algorithms for searching in $X+Y$. In *Proceedings of the 18th International Conference on Parallel Processing*, pages III–16–III–19, 1989.

[26] Deo, N., and D. Sarkar. Parallel algorithms for merging and sorting. *Information Sciences*, 51:121–131, 1990. Preliminary version in Proc. Third Intl. Conf. Supercomputing, May 1988, pages 513–521.

[27] Deo, N., A. Jain, and M. Medidi. Parallel multiselection and optimal parallel merging. In *Proceedings of Thirtieth Annual Allerton Conference on Communication, Control, and Computing*, pages 215–224, 1992.

[28] Deo, N., A. Jain, and M. Medidi. An optimal parallel algorithm for merging using multiselection. *Information Processing Letters*, 50(2):81–88, 1994.

[29] Eppstein, D. Parallel recognition of series-parallel graphs. *Information and Computation*, 98:41–55, 1992.

[30] Ferreira, A. G. The knapsack problem on parallel architectures. In Cosnard, M., Y. Robert, P. Quinton, and M. Raynal, editors, *Parallel and Distributed Algorithms*, pages 145–152. North Holland, 1989.

[31] Ferreira, A. G. A parallel time/hardware tradeoff $T.H = O(2^{n/2})$ for the knapsack problem. *IEEE Transactions on Computers*, 40(2):221–225, 1991.

[32] Frederickson, G. N., and D. B. Johnson. The complexity of selection and ranking in $X + Y$ and matrices with sorted columns. *Journal of Computer and System Sciences*, 24:197–208, 1982.

[33] Frederickson, G. N., and D. B. Johnson. Finding kth paths and p-centers by generating and searching good data structures. *Journal of Algorithms*, 4(1):61–80, 1983.

[34] Frederickson, G. N., and D. B. Johnson. Generalized selection and ranking: Sorted matrices. *SIAM Journal on Computing*, 13(1):14–30, 1984.

[35] Goldberg, A. V., Éva. Tardos, and R. E. Tarjan. Network flow algorithms. Technical Report CS-TR-216-89, Princeton University, March 1989.

[36] Goldschlager, L., L. Shaw, and J. Staples. The maximum flow problem is log space complete for P. *Theoretical Computer Science*, 21:105–111, 1982.

[37] Gries, D. *The Science of Programming.* Springer Verlag, 1981.

[38] Hagerup, T., and C. Rüb. Optimal merging and sorting on the EREW PRAM. *Information Processing Letters*, 33:181–185, December 1989.

[39] Horowitz, E., and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21(2):277–292, 1974.

[40] Jain, A., and N. Chandrasekharan. An efficient parallel algorithm for min-cost flow on directed series-parallel networks. In *Proceedings of the 7th International Parallel Processing Symposium*, pages 188–192, 1993.

[41] Ja′Ja′, J. *Introduction to Parallel Algorithms.* Addison Wesley, 1992.

[42] Johnson, D. B., and T. Mizoguchi. Selecting the Kth element in $X + Y$ and $X_1 + X_2 + \ldots + X_m$. *SIAM Journal on Computing*, 7(2):147–153, May 1978.

[43] Johnson, D. Parallel algorithms for minimum cuts and maximum flows in planar networks. *Journal of the ACM*, 34:950–967, 1987.

[44] Karp, R. M., and V. Ramachandran. Parallel algorithms for shared-memory machines. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, chapter 17, pages 869–941. The MIT Press/Elsevier, Cambridge, MA, 1990.

[45] Knuth, D. E. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

[46] Ladner, R. E., and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[47] Miller, G., and J. Naor. Flow in planar graphs with multiple sources and sinks. In *IEEE Foundations of Computer Science*, volume 30, pages 112–117, 1989.

[48] Miller, G. L., and J. H. Reif. Parallel tree contraction part 1: Fundamentals. In Micali, S., editor, *Randomness and Computation*, Advances in Computing Research, Volume 5. Jai Press, Greenwich, CT, 1989.

[49] Miller, G. L., and J. H. Reif. Parallel tree contraction part 2: Further applications. *SIAM Journal on Computing*, 20(6):1128–1147, December 1991.

[50] Mirzaian, A., and E. Arjomandi. Selection in $X+Y$ and matrices with sorted rows and columns. *Information Processing Letters*, 20:13–17, 1985.

[51] Munro, J. I., and H. Suwanda. Implicit data structures for fast search and update. *Journal of Computer and System Sciences*, 21:236–250, 1980.

[52] Orlin, J. B. A faster strongly polynomial minimum cost flow algorithm. In *Proc. 20th ACM Symposium on Theory of Computing*, pages 377–387, 1988.

[53] Paul, W., U. Vishkin, and H. Wagener. Parallel dictionaries on 2-3 trees. In *Proceedings of ICALP, 154*, pages 597–609, July 1983. Also R.A.I.R.O. Informatique Theorique/Theoretical Informatics, 17:397–404, 1983.

[54] Sanz, J. L., editor. *Opportunities and Constraints of Parallel Computing.* Springer Verlag, 1989.

[55] Sarnath, R., and X. He. Efficient parallel algorithms for selection and searching on sorted matrices. In *Proceedings of the International Parallel Processing Symposium*, pages 108–111, 1992.

[56] Shiloach, Y., and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *Journal of Algorithms*, 2:88–102, 1981.

[57] Snir, M. On parallel searching. *SIAM Journal on Computing*, 15:688–708, 1985.

[58] Stein, C., and J. Wein. Approximating the minimum-cost maximum-flow is P-complete. *Information Processing Letters*, 42:315–319, 1992.

[59] Valdes, J., R. E. Tarjan, and E. L. Lawler. The recognition of series parallel digraphs. *SIAM J. Comput.*, 11(2):298–313, May 1982.

[60] Valiant, L. G. General purpose parallel architectures. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, chapter 18, pages 945–971. The MIT Press/Elsevier, Cambridge, MA, 1990.

[61] Varman, P. J., B. R. Iyer, B. J. Haderle, and S. M. Dunn. Parallel merging: Algorithm and implementation results. *Parallel Computing*, 15:165–177, 1990.

[62] Vishkin, U. Structural parallel algorithmics. In *Proceedings of ICALP*, pages 363–380. Springer Verlag, 1991.

[63] Vishkin, U. A parallel blocking flow algorithm for acyclic networks. *Journal of Algorithms*, 13:489–501, September 1992.