Old Dominion University
ODU Digital Commons

Computer Science Theses & Dissertations | Computer Science

Summer 1991

Fast Parallel Algorithms for Basic Problems

Zhaofang Wen
Old Dominion University

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

Recommended Citation
Wen, Zhaofang. "Fast Parallel Algorithms for Basic Problems" (1991). Doctor of Philosophy (PhD), dissertation, Computer Science, Old Dominion University, DOI: 10.25777/9m63-1989
https://digitalcommons.odu.edu/computerscience_etds/120

This Dissertation is brought to you for free and open access by the Computer Science at ODU Digital Commons. It has been accepted for inclusion in Computer Science Theses & Dissertations by an authorized administrator of ODU Digital Commons. For more information, please contact digitalcommons@odu.edu.

FAST PARALLEL ALGORITHMS FOR BASIC PROBLEMS

by

Zhaofang Wen
B.S. July 1982, ZhongShan University, China
M.S July 1985, ZhongShan University, China

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

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

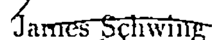
OLD DOMINION UNIVERSITY
August, 1991

Approved by:

C. Michael Overstreet (Advisor)

Stephan Olariu (Advisor)

Chester E. Grosch


James Schwing

Przemysław Bogacki

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABSTRACT

FAST PARALLEL ALGORITHMS FOR BASIC PROBLEMS

Zhaofang Wen

Old Dominion University

Advisors: C. Michael Overstreet and Stephan Olariu

Parallel processing is one of the most active research areas these days. We are interested in one aspect of parallel processing, i.e. the design and analysis of parallel algorithms. Here, we focus on non-numerical parallel algorithms for basic combinatorial problems, such as data structures, selection, searching, merging and sorting. The purposes of studying these types of problems are to obtain basic building blocks which will be useful in solving complex problems, and to develop fundamental algorithmic techniques.

In this thesis, we study the following problems: priority queues, multiple search and multiple selection, and reconstruction of a binary tree from its traversals. The research on priority queue was motivated by its various applications. The purpose of studying multiple search and multiple selection is to explore the relationships between four of the most fundamental problems in algorithm design, that is, selection, searching, merging and sorting; while our parallel solutions can be used as subroutines in algorithms for other problems. The research on the last problem, reconstruction of a binary tree from its traversals, was stimulated by a challenge proposed in a recent paper by Berkman et al. ("Highly Parallelizable Problems," *STOC* 89) to design doubly logarithmic time optimal parallel algorithms because a remarkably small number of such parallel algorithms exist.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGEMENT

I would like to express my deep appreciation to Dr. C. Michael Overstreet, my advisor, for his constant support and kindness during my graduate study. He not only provided many ideas and much guidance in academic research, but also taught me English and many other important skills.

I am very grateful to Dr. Stephan Olariu, also my advisor, for his help, guidance, and friendship over the years. With his art of lecturing, his classes were the most wonderful thing to experience; meanwhile, they gave me the theoretical background for my research. Among them I want to mention one: *Advanced Graph Theory* in Fall 89, from which I learned the interesting research area of design and analysis of parallel algorithms.

I would like to thank the members of my committee: Thanks to Dr. Mason Chew for his help. Dr. Schwing provided many helpful suggestions. I am grateful to Dr. Bogacki for his careful reading on an early draft and many constructive comments which have largely improved the presentation of the thesis. Special acknowledgement goes to Dr. Grosch, from whose expertise in parallel processing I benefited a lot.

Thanks to Dr. Stewart Shen and Dr. Shensheng Zhao for their help during my first year at ODU. Many thanks to Dr. Larry Wilson for his help during these years. To Frank Paterra, my fellow student, I would like to express my appreciation for his friendship. Thanks also to Dr. Kurt Maly, the chairman of this department, for encouraging students to participate actively in research.

I am very grateful to Dr. Yuesheng Xu, my best friend. Without his support and encouragement, I could not have pursued my Ph.D degree.

I would like to thank my parents for teaching me to work hard for fulfilling my dreams. Finally, I am indebted to my wife, Shaofen, for her understanding and encouragement in the whole process of my graduate study. She is the one who has helped me the most.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Contents

1 Introduction 1

1.1 Parallel Computation Models 1

1.2 Analysis of Parallel Algorithm 4

1.3 Problems of Interest 5

2 Priority Queues 8

2.1 Preliminaries 9

2.2 A Meldable Double-ended Priority Queue 11

2.2.1 Basic Operations 13

2.2.2 Melding 18

2.2.3 Discussion 20

2.3 Parallel Implementations of Priority Queue Operations 22

2.3.1 A Parallel Melding Algorithm 23

2.3.2 Parallel Initialization Algorithms for A Class of Priority Queues 28

3 Multiple Search and Multiple Selection 42

3.1 Multiple Search 44

3.1.1 Preliminaries 45

3.1.2 Sequential Complexity Bounds for Multiple Search 46

3.1.3 Parallel Multiple Search Using n Processor 48

3.1.4 Parallel Multiple Search Using Fewer Processors 52

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.1.5 Discussion 56

3.2 Multiple Selection 57

3.2.1 A Sequential Multiple Selection Algorithm 58

3.2.2 Time Bounds for Single Selection on Exclusive Write PRAMs 60

3.2.3 A Parallel Multiple Selection Algorithm 62

3.2.4 Discussion 64

4 Tree Reconstruction 66

4.1 Preliminaries 67

4.2 Sequential Algorithms 71

4.3 A Highly Parallel Algorithm 74

4.4 Discussion 82

5 Concluding Remarks 83

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Figures

1.1 SIMD Computer 2

2.1 A min heap 10

2.2 A min-max heap 11

2.3 A min-max-pair heap 13

2.4 A min-min-pair heap 21

2.5 The parallel initialization scheme for priority queues 30

2.6 A new way to look at the deap 40

4.1 a binary tree, and its various (Euler) paths and traversals 69

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 1

Introduction

The increasing success in development of parallel processing hardware has stimulated the recent developments in the design of parallel algorithms (see [4, 24, 41, 53, 56, 17, 35, 55, 64, 69, 72] for recent results). A parallel algorithm is a solution method for a given problem designed to be performed on a parallel computer. The study of parallel algorithms enables us to understand the inherent parallelism of a problem. It also provides a context in which we may identify difficult computational problems.

1.1 Parallel Computation Models

As is the case for the sequential algorithms which are designed on sequential computation models, parallel algorithms need to be developed on parallel computation models. Any computer, whether sequential or parallel, operates by executing instructions on data. A steam of instructions (the algorithm) tells the computer what to do at each step. A stream of data (the input to the algorithm) is affected by these instructions. Depending on whether one or several of these streams, two types of parallel computers are used (see [4, 71] for a complete survey): SIMD (Single Instruction stream, Multiple Data stream) computers and MIMD (Multiple Instruction stream,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

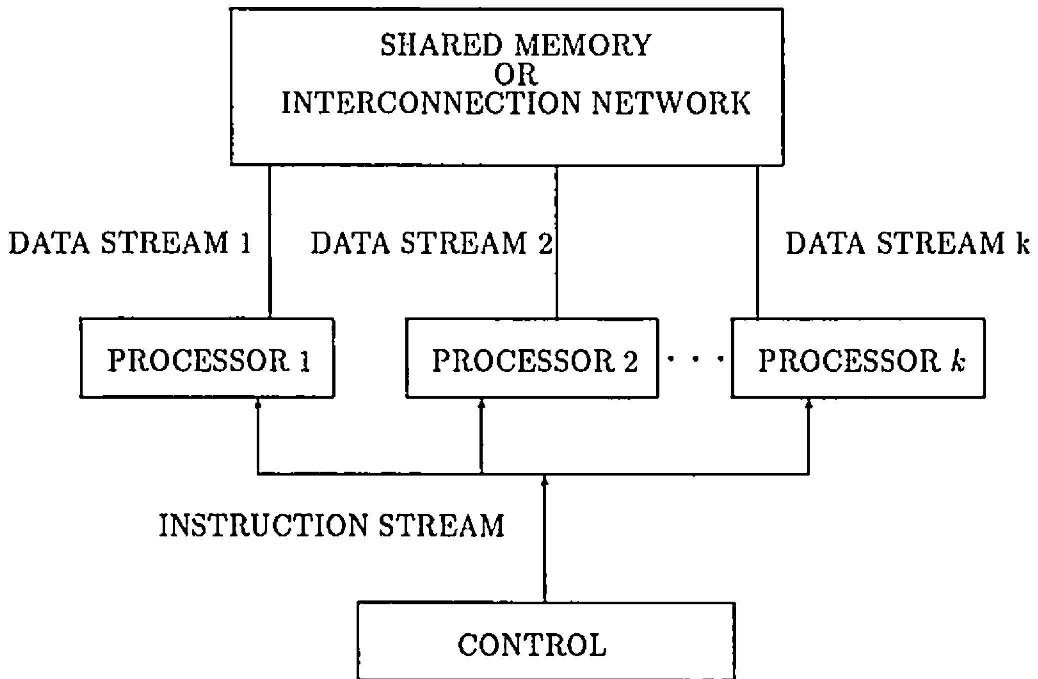


Figure 1.1: SIMD Computer

Multiple Data stream) computers. Processors in these models can be connected in many ways such as mesh, hypercube, or shared memory [68, 58]. Among these models, we give more details about the shared memory SIMD model. A SIMD shared memory computation model consists of k processors [4], as shown in Figure 1.1.

Each of the k processors has its own local memory in which it can store both programs and data. The processors operate synchronously: in every time step (controlled by some mechanism such as a global clock), the central control issues an instruction to each of the processors. All processors execute the same instruction, each on a different datum. Therefore, k data streams exist. Those processors that complete the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

execution of the instruction before others must remain idle until the next instruction is issued. The time interval between two instructions may be fixed or may depend on the instruction being executed. The processors in this model communicate through shared memory (SM). Four sub-models are used according to whether two or more processors can gain access to the same memory location simultaneously:

- Exclusive-Read, Exclusive-Write (EREW) SM SIMD model. Concurrent access to the same memory location in reading or writing is prohibited.
- Concurrent-Read, Exclusive-Write (CREW) SM SIMD model. Simultaneous read from the same memory location is allowed, but simultaneous write into the same memory location is disallowed.
- Exclusive-Read, Concurrent-Write (ERCW) SM SIMD model. Multiple processors are allowed to write into the same memory location but read access remains exclusive.
- Concurrent-Read, Concurrent-Write (CRCW) SM SIMD model. Both simultaneous read and simultaneous write are permitted.

The shared-memory SIMD model is also known in the literature as the *Parallel Random Access Machine* (PRAM) model. Although it ignores constraints in real architecture, the PRAM model has proved a popular model for parallel algorithm design. As Cole put it [39]: "The task of designing efficient, highly parallel algorithms is quite difficult, in general. The PRAM model provides an abstraction that strips away problems of synchronization, reliability and communication delays, thereby permitting algorithm designers to focus first and foremost on the structure of the computational problem at hand, rather than the architecture of a currently available machine." For this reason, we will also use the PRAM model as the computation model for our parallel algorithms in this thesis.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.2 Analysis of Parallel Algorithm

In the design and analysis of parallel algorithms, we need several complexity measures. The most important measure in evaluating a parallel algorithms is its *running time*, since speeding up computations appears to be the main motivation for studying parallel computing. The running time of a parallel algorithm is defined as the number of basic operations, or steps executed by the algorithm in the worst case. Operations such as *comparing, adding*, or *swapping* of two numbers are commonly accepted as basic operations in the PRAM model (in fact, each of these operations requires a constant number of time units on a typical sequential machine). Hence, the running time (or time complexity) of a parallel algorithm is an expression describing the number of such basic steps as a function of the number of processors used and the input size.

In evaluating a parallel algorithm for a given problem, it is natural to compare its time complexity with that of the fastest possible sequential algorithm for the same problem. Thus, a good measure of a parallel algorithm is the *speedup* it produces. The speedup obtained by a parallel algorithm for a problem is defined to be ratio of the worst-case running time of the fastest possible sequential algorithm for the problem to the worst-case running time of the parallel algorithm.

The *cost* of a parallel algorithm is defined as the product of the number of processors used and the parallel running time.

Assume that a lower bound is known on the number of sequential operations required in the worst case to solve a problem. If the cost of a parallel algorithm for that problem matches this lower bound to within a constant multiplicative factor, then the parallel algorithm is said to be *cost optimal.* A cost optimal algorithm is usually said to be *optimal* in literature. We will follow this tradition when no confusion is possible. When no optimal sequential algorithm is known for solving a problem, the *efficiency* of a parallel algorithm for that problem is sometimes used as a measure. In particular, the efficiency of a parallel algorithm for a problem is defined to be the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ratio of the worst-case running time of the fastest known sequential algorithm for the problem to the cost of the parallel algorithm.

Parallel algorithms are often characterized by different complexity classes. The most popular parallel complexity class is *NC* (Nick's Class [23, 24]). In particular, a parallel algorithm is an NC algorithm if it runs in $O((\log n)^c)$ time using $O(n^k)$ processors for some constants $c > 0$ and $k > 0$ (see [23] for more details). Studying membership in the class NC has been the focus of the complexity theory for parallel computation [24, 41].

Recently, several new classifications for parallel algorithms were introduced: *fully parallel* [13], *almost fully parallel* [13], and *highly parallel* [12]. Specifically, a parallel algorithm is *fully parallel* if it is optimal and runs in $O(1)$ time; a parallel algorithm is *almost fully parallel* if it is optimal and runs in $O(\alpha(n))$ time, where $\alpha(n)$ is the inverse-Ackermann function (see [13, 66] for details about the definition of $\alpha(n)$); an optimal parallel algorithm is *highly parallel* if it runs in $O(\log\log n)$ time. The notion of fully parallel algorithms represents an ultimate theoretical goal for parallel algorithm designers. However, research on lower bounds for parallel computation shows that most of the time this goal is unachievable; this is also the case for designing almost fully parallel algorithms for the same problems. For example, any optimal parallel algorithm to find the minimum among n elements requires at least $\Omega(\log\log n)$ time. A remarkably small number of problems are known for which there exist optimal parallel algorithms that run in $O(\log\log n)$ time. The class of highly parallel algorithms and the challenge of designing such algorithms is discussed in [12].

1.3 Problems of Interest

We are interested in non-numerical parallel algorithms for basic problems, such as data structures, selection, searching, merging and sorting, which are also fundamental in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the sequential setting. Studying the parallel algorithms for these problems is of both practical and theoretical importance: the parallel solutions can be used as building blocks to solve complex problems while the techniques developed in the process can also be useful in solving many other problems. Much of the recent advance in non-numerical parallel algorithms is due to several algorithmic techniques and the progress in solving a number of basic problems such as, parallel prefix sum [29], parallel linked list ranking [21], the Euler tour technique [65], the parallel tree contraction technique [44], parallel merge sort [20], parallel merging [43, 7, 33].

In this thesis, we study the following problems: priority queues, multiple search and multiple selection, and reconstruction of a binary tree from its traversals. A priority queue is a data structure (more formally, an abstract data type) which finds many applications in software engineering [45], disk scheduling [18, 36], simulation [32, 30, 45, 14], external sorting [8], operating systems [38], and network optimizations [67, 31]. Due to its far-reaching applications, parallel implementations of priority queue operations have recently received much attention in literature. In Chapter 2, we give a survey of the current research on parallel implementations of priority queue operations and also present our research results. Chapter 3 is devoted to two problems, multiple search and multiple selection. As we will explain later, the purpose of this chapter is to explore the relationships among four fundamental problems in algorithm design, i.e. selection, searching, merging and sorting. It turns out that our parallel solutions can be used as subroutines in algorithms for other problems. For example, our optimal parallel solution for the multiple search problem can be used in Hagerup and Rub's parallel merging algorithm [33] to obtain the optimal implementation of their algorithm on the EREW PRAM. In Chapter 4 we study a classical problem which is to reconstruct a binary tree from its traversals [42]. We present an optimal $O(\log\log n)$ time parallel algorithm (i.e. highly parallel) for this problem. Our solution to this problem is of theoretical importance for the following

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

reasons: (i) An extremely small number of problems are known to have highly parallel solutions, and thus designing such algorithms is proposed as a challenge in [12]. (ii) Due to the research on lower bounds in [25], Berkman et al. point out in [12] that doubly logarithmic time parallel algorithms usually need to run on an CRCW PRAM. (A known exception is Kruskal's $O(\log\log n)$ time parallel merging algorithm which runs on an CREW PRAM.) They also proposed in [12] a highly parallel algorithm for the binary tree reconstruction problem on the CRCW PRAM. Compared to their algorithm, however, our algorithm can be implemented on the CREW PRAM and hence gives one more example in the class of highly parallel algorithms that run on the CREW PRAM. Finally, we conclude the thesis in Chapter 5. The research results in this thesis can also be found in [48, 49, 50, 47, 46, 74].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 2

Priority Queues

In this chapter, we consider priority queues. The research results in this chapter also appear in [46, 48, 49, 50, 51]. A *priority queue* is a data structure each of whose elements is assigned a label representing its priority. In this context, the natural order of the elements in such a structure is dictated by their respective priority. Priority queues are widely used in software engineering [45], disk scheduling [18, 36], simulation [32, 30, 45, 14], external sorting [8], operating systems [38], and network optimizations [67, 31], to name just a few (see [10, 38] for a more competent discussion).

More formally, a priority queue can be viewed as an abstract data type maintaining a set of keys from a totally ordered universe and supporting the following basic operations:

Initialization: initialize the priority queue;

Find-min: find the minimum (find the element with the highest priority);

Delete-min: delete the minimum;

$Insert(x)$: insert key x into the structure.

Of course, instead of finding or deleting the minimum we could just as well insist

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

on maintaining the structure such that the maximum is operated upon. Depending on applications [46], priority queues also support other operations such as *Meld*:

Meld($Q1, Q2$): combine priority queues $Q1$ and $Q2$

The idea of a priority queue can be naturally extended to a *double-ended priority queue* where, in addition to *Find-min*, *Delete-min*, the operations of *Find-max* and *Delete-max* are also of interest. Double-ended priority queues can be used to support order-statistic trees [8] which find applications to signal processing [60].

We give some background about priority queue implementations in Section 2.1. Our research results will be presented in the following sections: a meldable double-ended priority queue in Section 2.2, and parallel algorithms for initialization of a class of priority queues in Section 2.3.

2.1 Preliminaries

Typically, heaps are used to implement priority queues in computer systems. Various heaps have been invented such as: binomial heaps [73], leftist heap [67, 62], Fibonacci heaps [31], and relaxed heaps [27]. Here, we are interested in the one proposed by Williams [75] called the heap. Specifically, a heap is a binary tree with the following properties:

- *heap-shaped property*: all leaves occur on the last two adjacent levels in the structure, with the leaves on the last level being confined to the leftmost position; all other levels are complete.
- *min-ordering*: every element is no larger than the smallest of its children.

Figure 2.1 illustrates the heap concept.

It is well known that in the heap implementation of priority queues *Find-min* takes $O(1)$ time, while both *Delete-min* and *Insert* take $O(\log n)$ time.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

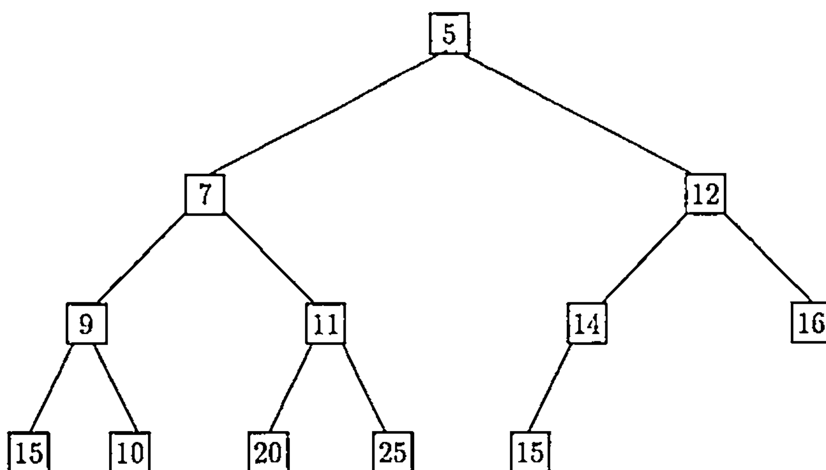


Figure 2.1: A min heap

Due to the heap-shaped property, a nice feature of heaps is that they can be implemented in situ, with no need for additional pointers. As a matter of fact, an n-element heap can be stored in an array of size n [10]: an array $H[1..n]$ can be interpreted as a heap-shaped binary tree if for every i ($1 \leq i \leq \lfloor \frac{n-1}{2} \rfloor$), the children of $H[i]$ are $H[2i]$ and $H[2i+1]$.

To implement a double-ended priority queue, Atkinson et al. [8] have recently proposed an interesting variation on the idea of a heap: they defined the *min-max heap* as a binary tree such that: (i) it has the heap-shaped property; and (ii) it is *min-max ordered*: elements on even levels are less than or equal to their descendants, and elements on odd levels are greater than or equal to their descendants. Figure 2.2 illustrates this min-max heap concept. *Max-min heaps* are defined completely analogously: such a structure begins with the maximum element at the root and then the heap conditions alternate between minima and maxima.

As it turns out [8], when the double-ended priority queue is implemented as min-max heap, *Find-min* and *Find-max* can be performed in $O(1)$ time, while *Delete-min*,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

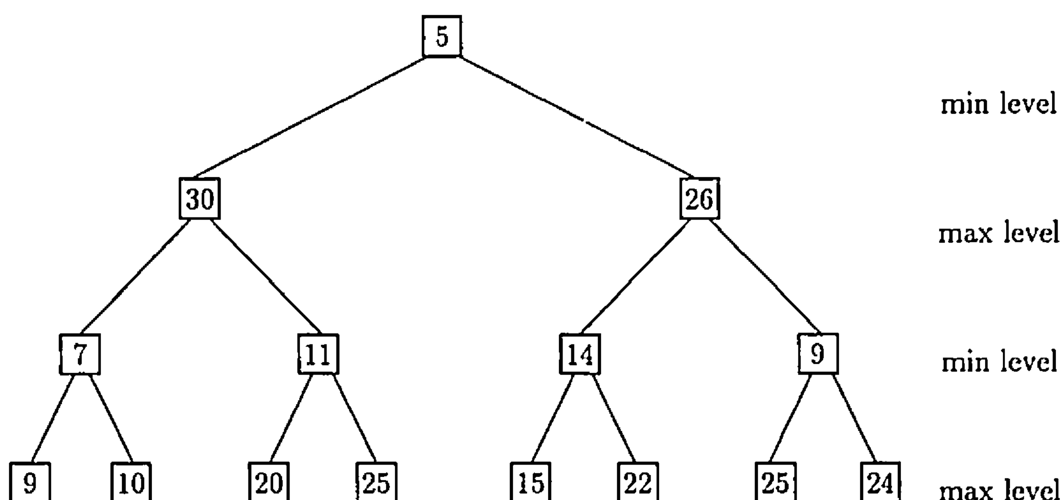


Figure 2.2: A min-max heap

Delete-max, and *Insert* takes $O(\log n)$ time. In addition, *Atkinson et al.* [8] propose an $O(n)$ time, and thus optimal, algorithm to initialize a min-max heap.

As an alternative to min-max heaps introduced in [8], Carlsson [15] propose a new data structure called the *deap* which provides an efficient implementation of a double-ended priority queue. Formally, a deap is a heap-shaped data structure featuring the following properties: the left (right) sub-tree of the non-existing root is a min-heap (max-heap); each leaf in the min-heap is smaller than a corresponding leaf in the max-heap. On an n-element deap, the operations *Find-min* and *Find-max* take $O(1)$ time, *Delete-min, Delete-max*, and *Insert* take $O(\log n)$ time [15, 16]. Moreover, the deap can be implemented in situ and initialized in $O(n)$ sequential time.

2.2 A Meldable Double-ended Priority Queue

An interesting problem arising in fault-tolerant distributed simulation [46] is the following: assume that several (computationally active) sites in a distributed system are simulating a process. It is sometimes desirable to implement the corresponding event

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

lists as double-ended priority queues. Basic fault-tolerant requirements specify that if one of these sites, say S_i, suddenly becomes computationally inactive, another one continue the simulation performed by S_i. For this purpose, we need to elect a site S_j ($i \neq j$) which will then import the event list of S_i and will meld it with its own event list.

It is natural to consider first the meldabilities of the existing double-ended priority queue implementations, i.e. min-max heaps and deaps. To the best of our knowledge, it is still an open question whether deaps are meldable. However, it has recently been proven [34] that min-max heaps are not meldable, that is, melding two min-max heaps of sizes n and k, respectively, cannot be done in less than $\Omega(n+k)$ time. The inherent structure of the min-max heaps causing this negative result motivates us to investigate a different data structure to implement efficiently a double-ended priority queue. This data structure can be defined by modifying slightly the structure in the definition of the min-max heaps. As we are about to show, however, with this modification the resulting data structure is meldable. This data structure was first proposed in a different form by Williams [75], and is herewith referred to as the *min-max-pair heap.* In essence, a min-max-pair heap is a binary tree H featuring the heap-shaped property, such that every node in H has two fields, called the *min field* and the *max field*, and such that H has a *min-max-pair* ordering: for every i ($1 \leq i \leq n$), the value stored in the *min field* of $H[i]$ is the smallest key in the subtree of H rooted at $H[i]$; while the value stored in the *max field* $H[i]$ is the largest key stored in the subtree of H rooted at $H[i]$ (see Figure 2.3). We will show that min-max-pair heaps can be implemented in situ, with no need for additional pointers.

As it turns out, when the double-ended priority queue is implemented as a min-max-pair heap, *Find-min and Find-max* can be performed in $O(1)$ time, while *Delete-min, Delete-max*, and *Insert* take $O(\log n)$ time. However, what really distinguishs min-max-pair heaps from min-max heaps is the fact that min-max-pair heaps can be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

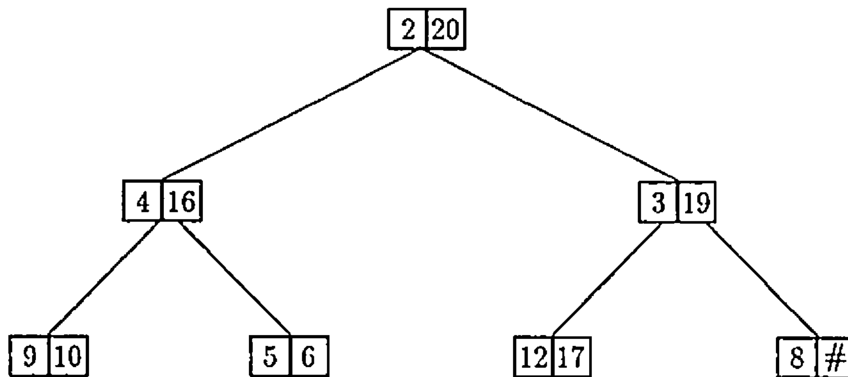


Figure 2.3: A min-max-pair heap

melded efficiently in sublinear time. More precisely, we show that two min-max-pair heaps with n and k nodes can be melded in $O(\log k \log \frac{n}{k})$ time.

2.2.1 Basic Operations

Consider an array $H[1..n]$ as input. For $1 \leq i \leq n$, each element $H[i]$ of H has two fields, $H[i].min$ and $H[i].max$. Therefore, the array H can be viewed as containing $2n-1$ or $2n$ keys altogether; in case H contains $2n-1$ keys, the max field $H[n].max$ contains a special symbol, namely #.

The initialization algorithm for a min-max-pair heap resembles the initialization of the standard heap structure [10]. Let $H[i]$ be an arbitrary node of the array to be made into a min-max-pair heap. We further assume that for all j $(i \leq j)$, the subtrees rooted at the children of $H[j]$, namely $H[2j]$ and $H[2j+1]$, provided they exist, have been made into min-max-pair heaps. First, we restore the min-max-pair heap property along the min fields of the nodes in the subtree rooted at $H[i]$, by trickling down larger keys. We then restore the min-max-pair heap property along the max fields of the nodes in the subtree rooted at $H[i]$, by trickling down smaller

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

keys. The purpose of this is to ensure that the $H[i].min$ and $H[i].max$ contain the smallest and the largest keys in the subtree rooted at $H[i]$, respectively. The details are given below:

Procedure Initialization($H[1..n]$);

  For $i \leftarrow n$ **downto** 1 **do** Siftdown($H[i]$);

end;

Procedure Siftdown($H[i]$);

/* Subtrees rooted at $H[2i]$ and $H[2i+1]$ are already min-max-pair heaps */

  Trickledown-min-field($H[i]$);

  Trickledown-max-field($H[i]$);

end;

Procedure Trickledown-min-field($H[i]$);

  $p \leftarrow H[i]$;

  if $p.max < p.min$ **then** Swap($p.min, p.max$);

  if p is a leaf **then** return;

  $p1 \leftarrow$ child of p with smallest *min* field;

  if $p1.min < p.min$ **then**

   Swap($p1.min, p.min$);

   Trickledown-min-field($p1$);

  endif

end;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Procedure Trickledown-max-field is similar to procedure Trickledown-min-field. The following result establishes the correctness and the time complexity of our procedure.

Theorem 2.2.1 *Procedure Initialization correctly constructs a min-max-pair heap structure over* $2n$ *or* $2n-1$ *keys in* $O(n)$ *time.*

Proof. To settle the correctness we notice the following: For every $H[i]$ ($2 \leq i \leq n$), when Trickledown-min-field($H[i]$) (resp. Trickledown-max-field($H[i]$)) terminates, $H[i].min$ (resp. $H[i].max$) contains the smallest (resp. largest) key in the subtree rooted at $H[i]$, while the subtrees rooted at $H[2i]$ and $H[2i+1]$ (provided they exist) are min-max-pair heap; this is easily seen by induction on the height of node $H[i]$. Therefore, when Initialization($H[1..n]$) terminates, the whole structure is made into a min-max-pair heap.

To address the complexity, consider what happens in procedure Trickledown-min-field when node $H[i]$ is being processed. To ensure that $H[i].min \leq H[i].max$ and to determine the child of $H[i]$ with smallest *min* field three comparisons are required. Consequently, the total number of comparisons to perform initialization is at most:

$\sum 3(\log n - \log i + 1)$

which is $O(n)$. □

Next, we show that performing the standard operation *Insert*(x) and *Delete-min* as well as *Delete-max* can be done in $O(\log n)$ time. Basically, the idea of inserting a new element x into a min-max-pair heap is the same as the insertion of a new element into a standard heap. We first place the new key at the bottom of the structure and then perform the well known bubble-up operation. Just as in the case of heaps, the time complexity of the *Insert*(x) operation for the min-max-pair heap is dominated by the cost of the bubble-up which is easily seen to $O(\log n)$ as shown in the following procedures:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Procedure Bubbleup(H[i]);
        p ← H[i];
        b ← false;
        if p.min > p.max then Swap(p.min,p.max);
        if p is the root then return;
        p1 ← the parent of p;
        if p1.max < p.max then
          Swap(p1.max,p.max);
          b ← true
        endif
        if p1.min > p.min then
          Swap(p1.min,p.min);
          b ← true
        endif
        if b then Bubbleup(p1);
end;
Procedure Insert(x,H[1..n]);
        if H[n].max = '#' then
          H[n].max ← x;
        else
          n ← n + 1;
          H[n].min ← x
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            H[n].max ← '#'
        endif
        Bubbleup(H[n]);
end;
```

Similarly, the idea of *Delete-min* and *Delete-max* resembles the corresponding operations on heaps. The details are spelled out in the following procedures. It is an easy matter to confirm that both these operations can be executed in $O(\log n)$ time, while *Find-min* and *Find-max* take $O(1)$ time.

```
Procedure Delete-min(H[1..n]);
        if H[n].max = '#' then
            H[1].min ← H[n].min;
            n ← n − 1;
        else
            H[1].min ← H[n].max;
            H[n].max ← '#'
        endif
        Trickledown-min-field(H[1]);
end;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2.2 Melding

Recently, Sack and Strothotte [61] proposed an efficient algorithm to meld two heaps in sublinear time. Specifically, melding two heaps of size n and k can be done in $O(\log k \log \frac{n}{k})$ time. The general case of the heap-melding algorithm in [61] reduces, in stages, to that of melding perfect heaps. (A heap H is perfect if the leaves occur at the last level only.) The idea in [61] is very elegant: first, to meld two perfect heaps $H1$ and $H2$ of equal size, make the rightmost leaf of $H2$ into the new root, whose children will be the old roots of $H1$ and $H2$. After this, the new root is sifted down to restore the heap property.

Next, let $H1$ and $H2$ be two perfect heaps of sizes n and k, respectively, with $k < n$. Start at the root of $H1$ and compare it to the root of $H2$; if the root of $H2$ is smaller than the root of $H1$ then exchange the two roots and perform a "sift-down" on $H2$. This operation is repeated along the path (*Walk-down*) in $H1$ from the root down to the leftmost leaf of $H1$ for $\log \frac{n}{k}$ steps.

We show that the heap melding algorithm in [61] can be adapted to meld two min-max-pair heaps in sublinear time. We shall therefore focus on melding perfect min-max-pair heaps, that is, min-max-pair heaps whose leaves occur at the last level only. We refer interested readers to [48] where the tedious details are provided.

Just as in [61], to reduce the amount of data movement during the execution of our melding algorithm, we shall assume a pointer-based implementation. In this context, a min-max-pair heap node v contains the following fields:

- $v.min$ and $v.max$ fields;
- $v.lchild$ contains a pointer to the left child of v in the min-max-pair heap;
- $v.rchild$ contains a pointer to the right child of v in the min-max-pair heap;

It is convenient to assume that $depth(H)$ returns the depth of the min-max-pair heap H in constant time. The details of our melding algorithms are as follows.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Procedure Meld-perfect-equal(H_1, H_2);

/* H_1 and H_2 are two min-max-pair heaps of same size */

$p \leftarrow$ the last node in H_2

remove p from H_2;

$p.lchild \leftarrow H_1$;

$p.rchild \leftarrow H_2$;

Siftdown(p);

$H_1 \leftarrow p$;

end;

Procedure Meld-perfect(H_n, H_k);

$p \leftarrow$ node on the path from the root to the leftmost leaf in H_n,

such that the subtree rooted at p has k nodes

$r \leftarrow$ root of H_n;

Walk-down(H_n, H_k, r, p);

$p1 \leftarrow$ parent of p;

Meld-perfect-equal(p, H_k);

if $p1 \neq nil$ **then** $p1.lchild \leftarrow p$;

else $H_n \leftarrow p$

end;

Procedure Walk-down($H_n, H_k, from, to$);

/* H_n is a min-max-pair heap with n nodes;

H_k is a min-max-pair heap with k nodes;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

'from' is the starting location of current operation

on the path from the root in H_n to the leftmost leaf;

'to' is the ending position of the operation */

if $H_k.min < from.min$ **then** Swap($H_k.min, from.min$);

if $H_k.max > from.max$ **then** Swap($H_k.max, from.max$);

Siftdown(H_k);

if $from = to$ **then** return;

else

$next \leftarrow from.lchild$

Walk-down($H_n, H_k, next, to$);

endif

end;

It is easy to see that the complexity of our algorithm is exactly the same as that of the heap-melding algorithm in [61].

Theorem 2.2.2 *Two min-max-pair heap of n and k elements, respectively, can be melded in* $O(\log k \log \frac{n}{k})$ *time.* □

2.2.3 Discussion

We have shown in this section that min-max-pair heaps are meldable. It is interesting to see that the idea which leads to the min-max-pair heap can be further expanded. As an example, we define a min-min-pair heap as a heap-shaped binary tree with each

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

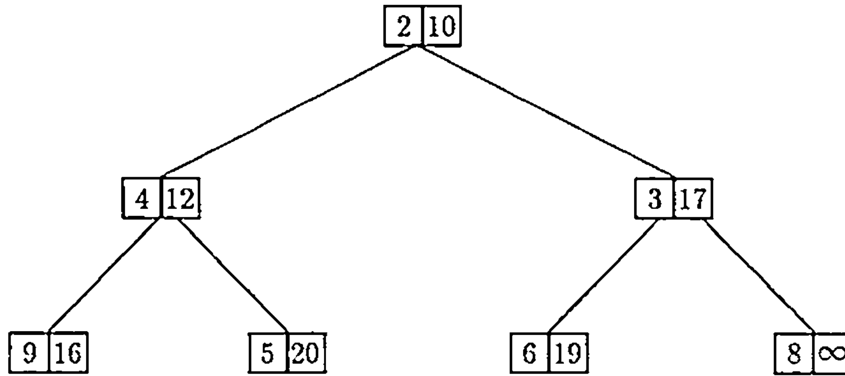


Figure 2.4: A min-min-pair heap

node p, containing two fields called *min1* and *min2*, respectively. The value of *min1* is the smallest of all the values stored in the subtree rooted at p; *min2* contains the smallest of all the values stored in the *min2* fields of all nodes in the subtree rooted at p. Finally, for every node q in the subtree rooted at p, $p.min2 > q.min1$ (see Figure 2.4).

An interesting feature of a min-min-pair heap is that the *min1* field of the root contains the minimum value in the whole structure, while *min2* of the root contains the median of the whole structure. As it turns out [48], a min-min-pair heap containing $2n - 1$ or $2n$ keys can be initialized in $O(n)$ time. Clearly, the operations *Find-min* and *Find-median* can be performed in $O(1)$ time. Similarly, *Insert*(x), *Delete-min* and *Delete-median* can be done in $O(\log n)$ time [48]. Similarly, one can define a max-max-pair heap and a max-min-pair heap [48]. Unfortunately, none of these variations of the min-max-pair heap are meldable in sublinear time.

Finally, an interesting open question is whether or not deaps are meldable in sublinear time. In particular, it would be interesting to see whether the techniques in [61] can be extended to meld deaps.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.3 Parallel Implementations of Priority Queue Operations

In this section, we study the parallel implementation of priority queue operations. Priority queues have been used in a wide variety of parallel algorithms, e.g. multiprocessor scheduling, graph search, and branch-and-bound algorithms [57, 56, 59]. In 1983, Yoo [76, 57] parallelized the *Delete-min* operation on a priority queue implemented by a heap, in order to obtain a parallel version of Kruskal's minimum spanning tree algorithm. In particular, he showed that although a single *Delete-min* operation on an n-element heap required $O(\log n)$ time, by using a software pipelining technique a new *Delete-min* operation can begin after only $O(1)$ time. In 1987, Biswas and Browne studied simultaneous updates of priority queue structures: their scheme allows $O(\log n)$ processors to be active on a heap. In 1988, Rao and Kumar [59] presented an interesting approach to allow concurrent *Insert* and *Delete-min* operations on the heap in the shared memory MIMD computation model. Their main contribution is to have changed the traditional *Insert* from the well known bottom up fashion to a novel top-down approach. In their scheme, several *Insert* and *Delete-min* operations can be active simultaneously without causing deadlocks. In addition, their scheme retains the strict priority ordering of the serial-access heap algorithms; i.e. a *Delete-min* operation returns the smallest key of all the keys in the structure, including those whose insertion is in progress. Concurrent insertions and *Delete-min* operations on a priority queue implemented by *skew heaps* were studied by Jones [40]. he showed that on an MIMD shared memory model both *Insert* and *Delete-min* operations on an n-element skew heap can performed in $O(\log n)$ time but, using piplelining, a new operation can begin after only $O(1)$ time. Quinn [57] reports a parallel initialization algorithm (due to Yoo) of a priority queue implemented by an n-element heap in $O((\log n)^2)$ time using $O(n)$ processors on an MIMD model. It

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is obvious that Yoo's initialization algorithm is not cost-optimal. All the parallel priority queue schemes above use heaps in which internal nodes contain just one key.

Most recently, Deo and Prasad [26] and Pinotti and Pucci [54] proposed similar variations on the traditional heap structure. Their data structure (called *bandwidth heap* in [54] and *parallel heap* in [26]) has the heap-shaped property, with every internal node containing k elements, for some k. In [26] and [54] concurrent insertion and deletion operations on this new data structure are also investigated. Network implementation of simultaneously accessible priority queue is studied in [28].

In Section 2.3.1, we will present a parallel algorithm for melding priority queues which will be followed by optimal parallel initialization algorithms for a class of priority queues in Section 2.3.2.

2.3.1 A Parallel Melding Algorithm

In this section, we consider melding priority queues in parallel. We propose a method to parallelize Sack and Strothottes's heap melding algorithm [61] (they called it heap merging in [61]). As it turns out, our method can also be applied to obtain a parallel melding algorithm for double-ended priority queues implemented by min-max-pair heaps.

To reduce the amount of data movement during the execution of parallel melding algorithm, we shall assume a pointer-based implementation. In this context, a heap node v contains the following fields:

- $v.key$ contains the key stored at node v;
- $v.lchild$ contains a pointer to the left child of v in the heap;
- $v.rchild$ contains a pointer to the right child of v in the heap;

Our parallel algorithm relies, in part, on a new version of the well known "sift-down" procedure used for restoring the heap property (see [10] and [49]). More

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

precisely, several elements of H will be sifted down in parallel. Initially, the root of H is assigned one processor which proceeds to siftdown for two time units; after that, a new processor is assigned to the (new) root, all active processors acting on H proceed to siftdown for two time units, and so on. It is important to note that a processor continues to be active as long as it can sift down. After this, it becomes inactive and will stay inactive until it is reassigned to the root of H at a later moment, in a cyclic way. To justify the idea of the processor assignment we note that if we use at least $t = depth(H)$ processors $P_0, P_1, ..., P_{t-1}$, and if the processors are assigned modulo $depth(H)$ then, we are always guaranteed to assign only inactive processors. It is clear that this processor allocation scheme avoids read and write conflicts in H. As a matter of convenience, we assume that every processor P_i $(0 \leq i \leq t-1)$ stores in its local memory the following information:

- $current(P_i)$, standing for the node in H where P_i is currently at;
- $active(P_i)$, which is either a 1 or a 0 depending on whether or not P_i is active. The initial value is 0.

The details of the processor allocation scheme and siftdown are presented in procedures *Siftdown* and *Parallel-Walkdown.*

Procedure Siftdown(P_i);

$v \leftarrow current(P_i)$,

let w be the child of v with the smallest key;

if $v.key > w.key$ **then**

Swap($v.key, w.key$);

$current(P_i) \leftarrow w$

else

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$active(P_i) \leftarrow 0$; {*deactivated*}

end;

Call a heap H perfect if the leaves occur at the last level only. The general case of the sequential heap-melding algorithm in Sack and Strothotte reduces, in stages, to that of melding perfect heaps. We shall therefore, focus on melding perfect heaps in parallel. First, melding two perfect heaps *heap*1 and *heap*2 of equal size can be easily done sequentially: make the rightmost leaf of *heap*2 into the new root whose children become the old roots of *heap*1 and *heap*2, after which the new root is sifted down to restore the heap property. We shall refer to this simple procedure as *Meld-Equal-Perfect-Heaps.*

Next, we present the procedure *Parallel-Walkdown* which is at the heart of our parallel algorithm.

Procedure Parallel-Walkdown($heap1, from, to, heap2$);

/* $depth(heap1) > depth(heap2)$ */

$t \leftarrow \min\{depth(heap2), depth(heap1) - depth(heap2)\}$;

/* We use t processors, $P_0, P_1, ..., P_{t-1}$, to

restore the heap property on *heap*2 */

$j \leftarrow 0$;

for $i \leftarrow 0$ **to** $t-1$ **do** in parallel

$active(P_i) \leftarrow 0$;

for $i \leftarrow 0$ **to** $depth(heap1) - depth(heap2)$ **do**

if $from.key > root(heap2).key$ **then**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        Swap(from.key, root(heap2).key);
        assign processor P_j to root(heap2);
        j ← (j + 1) mod t;
      endif;
      for c ← 1 to 2 do
        all active processors P_j do in parallel
          Siftdown(P_j);
      from ← from.lchild;
    endfor ;
    /* let all active processors siftdown as far as they can */
    for i ← 1 to depth(heap2) do
      all active processors P_i do in parallel
        Siftdown(P_i);
end;
```

We can now present the details of a parallel procedure to meld two perfect heaps.

```
Procedure Parallel-Meld-Perfect-Heaps(heap1, heap2);
0.        d_1 ← depth(heap1);
1.        d_2 ← depth(heap2);
2.        u ← root(heap1);
3.        for k ← 1 to d_1 − d_2 − 1 do
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. $u \leftarrow u.lchild$;

5. $to \leftarrow u$;

6. Parallel-Walkdown($heap1, root(heap1), to, heap2$);

7. $u \leftarrow to.lchild$;

8. Meld-Equal-Perfect-Heaps($subheap(heap1, u), heap2$);

9. Add the root of the new heap as the leftchild of to;

end;

Theorem 2.3.1 *Procedure Parallel-Meld-Perfect-Heaps correctly melds two perfect heaps* heap1 *and* heap2 *with* n *and* k *nodes, respectively, in* $O(\log n)$ *time on an EREW PRAM with* $\min\{\lfloor \log n \rfloor - \lfloor \log k \rfloor, \lfloor \log k \rfloor\} + 1$ *processors.*

Proof. To begin, we note that $d_1 = \lfloor \log n \rfloor$ and $d_2 = \lfloor \log k \rfloor$. The correctness follows directly from Lemma 2.1 in [61], together with the observation that when all processors become inactive, *heap2* is guaranteed to be a heap. Afterwards we use the sequential algorithm to meld perfect heaps of equal size, as describe above. To argue for the running time, we note that by assumption lines 1-2 take $O(1)$ time. Lines 4-5 and 7 take $O(\lfloor \log n \rfloor - \lfloor \log k \rfloor)$ time. Altogether, the time complexity of the algorithm is $O(\log n)$. From the previous discussion about the parallel-siftdown, we know that no memory conflict is possible. Therefore, the computation can be carried out on an EREW PRAM, using $\min\{\lfloor \log n \rfloor - \lfloor \log k \rfloor, \lfloor \log k \rfloor\} + 1$ processors. □

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.3.2 Parallel Initialization Algorithms for A Class of Priority Queues

In this section, we present a technique for inducing a class of priority queue structures upon an n-element array. As examples, we show that this technique can be applied to initialize a heap, a min-max heap, a min-max-pair heap, and a deap in $O(\frac{n}{p})$ time using p $(1 \leq p \leq \lceil \frac{n}{\log n} \rceil)$ processors on an EREW PRAM.

As it turns out, once these data structures implementing double-ended priority queues have been initialized, the techniques in [59] can be applied directly to obtain efficient concurrent *Insert, Delete-min*, and *Delete-max* operations. As a result of these efficient concurrent operations on double-ended priority queues, efficient concurrent operations on *order statistics trees* [8] can be obtained.

Throughout the rest of the section, we assume that processors $P_1, P_2, ..., P_p$ $(1 \leq p \leq \lceil \frac{n}{\log n} \rceil)$ are available.

Initializing Heaps

Our terminology pertaining to binary trees is borrowed from [10]. Recall that an array $H[1..n]$ can be interpreted as a heap-shaped binary tree if for every i $(1 \leq i \leq \lfloor \frac{n-1}{2} \rfloor)$, the children of $H[i]$ are $H[2i]$ and $H[2i+1]$. When no confusion is possible, we shall refer to the array $H[1..n]$ simply as H.

For further reference we shall review basic properties of heap-shaped binary trees. Let H be a heap-shaped binary tree with n nodes. The following statements are satisfied (see [10] for proofs):

(A.1) The depth of H is exactly $\lfloor \log n \rfloor$.

(A.2) For all $i = 0, 1, ..., \lfloor \log n \rfloor - 1$, there are 2^i nodes at level i.

(A.3) The nodes at level i $(1 \leq i \leq \lfloor \log n \rfloor)$, are $2^i, 2^i + 1, ..., 2^{i+1} - 1$, provided they exist.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(A.4) A binary tree of depth d has at most $2^{d+1} - 1$ nodes.

Writing $k = \lfloor \log p \rfloor$.

(B) There are at most p subtrees of H rooted at nodes of level k.

(To see that this is the case, note that by (A.2), the number of these subtrees is $2^k \leq 2^{\log p} = p$.)

Let H_i $2^k \leq i \leq 2^{k+1} - 1$) be the subtrees of H rooted at nodes of level k. Next, we claim that:

(C) Every H_i $2^k \leq i \leq 2^{k+1} - 1$) contains at most $\frac{4n}{p}$ nodes.

To justify this claim, note that by virtue of (A.1) and by our choice of k, the depth of every such H_i is exactly $\lfloor \log n \rfloor - k = \lfloor \log n \rfloor - \lfloor \log p \rfloor < \log n - \log p + 1 = \log \frac{n}{p} + 1$. Now (A.4) guarantees that the total number of nodes in H_i is bounded above by $2^{\log \frac{n}{p} + 2} - 1 \leq \frac{4n}{p}$. Furthermore, we note that:

(D) $2^k - 1 < p$

(Trivially, $2^k - 1 \leq 2^{\log p} - 1 < p$.)

Perhaps the easiest way to explain our technique is by showing how to induce a heap structure on H. For this purpose, we proceed in the following two stages (see Figure 2.5):

Stage 1. Writing $k = \lfloor \log p \rfloor$, assign one processor to each of the subtrees H_i ($2^k \leq i \leq 2^{k+1} - 1$). By (B), at most p processors are assigned in this way; by (C) each subtree contains at most $\frac{4n}{p}$ nodes of H. The unique processor assigned to H_i makes H_i into a heap in $O(\frac{n}{p})$ sequential time.

Stage 2. We propose to "grow" in parallel the heaps H_i ($2^k \leq i \leq 2^{k+1} - 1$) into H itself by adapting the well known sequential trickle down. For this purpose, the p processors are redistributed to the first $2^k - 1$ nodes of H, one processor per node. Note that by (D) this can be done using at most the p processors at our disposal. At this stage, it is convenient to assume that every processor P_i ($1 \leq i \leq 2^k - 1$) stores

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

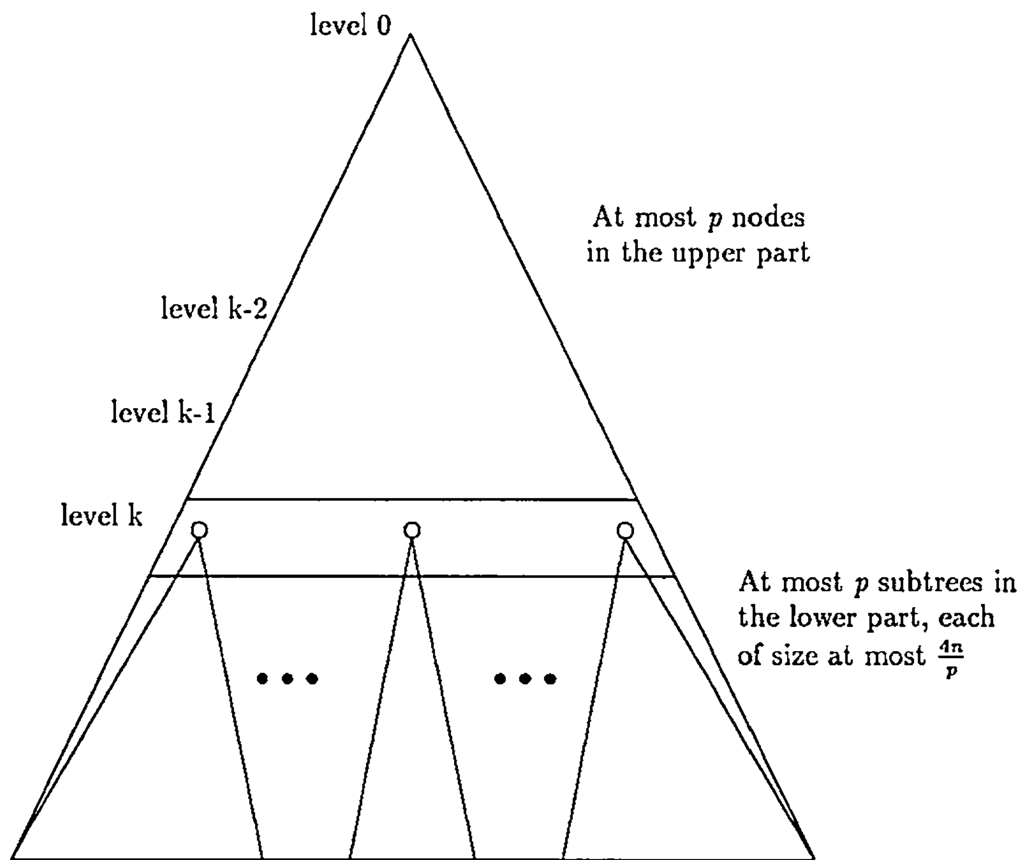


Figure 2.5: The parallel initialization scheme for priority queues

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in its local memory the following information:

- $current(P_i)$, standing for the node in H which P_i is processing. Initially, $current(P_i) = i$;

- $active(P_i)$, which is either a 1 or a 0 depending on whether or not P_i is active. The initial value is 0.

Every element of H at level 0 through $k-1$ will be trickled down in parallel. As it turns out, it is convenient to assume that the processor P_i initially assigned to $H[i]$ ($i = 0, 1, ..., 2^k - 1$) will move along with the key contained in $H[i]$. To avoid read and write conflicts among processors we proceed in a pipelined fashion: we begin by activating the processors at level $k-1$ which will proceed to "trickle down" two levels. After this, the processors at level $k-2$ will begin to trickle down, and so on. Every processor remains active until it reaches a leaf where it will become inactive.

Naturally, in moving down from a node w to one of its children, processor P does the following: let v stand for the child of w with the smallest key; if key(w) is larger than key(v), they are swapped. The details of this procedure are spelled out as follows:

Procedure Parallel-Trickledown(P_i);

1. $j \leftarrow current(P_i)$;
2. $t \leftarrow$ the index of the child of $H[j]$ with the smallest key;
3. **if** $key(H[j]) > key(H[t])$ **then**
4. Swap($key(H[j]), key(H[t])$);

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5. $current(P_i) \leftarrow t$;

6. **if** $H[t]$ is a leaf **then**

7. $active(P_i) \leftarrow 0$

end;

We are now in a position to show that the different pieces of our heap initialization algorithm fit together.

Procedure Parallel-Initialize-Heap($H[1..n]$);

Input: an array $H[1..n]$ containing n key from a totally ordered universe U;

Output: the same array, organized as a heap;

1. $k \leftarrow \lfloor \log p \rfloor$;

2. **for** all i, $(2^k \leq i \leq 2^{k+1} - 1)$ **do** in parallel

3. construct the heap H_i rooted at i;

4. **for** all i, $(1 \leq i \leq 2^k - 1)$ **do** in parallel

5. assign processor P_i to $H[i]$;

6. $current(P_i) \leftarrow i$;

7. $active(P_i) \leftarrow 0$; {all inactive}

8. **endfor**

9. **for** $level \leftarrow k - 1$ **downto** 0 **do**

10. **for** all active processors P_i with $\lfloor \log current(P_i) \rfloor = level$ **do** in parallel

11. $active(P_i) \leftarrow 1$;

12. **for** all active processors P_i **do** in parallel

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
13.                 Parallel-Trickledown(P_i);
14.             for all active processors P_i do in parallel
15.                 Parallel-Trickledown(P_i);
16.         endfor;
17.         /* Let all processors trickle down as far as possible */
18.         for i ← 1 to ⌊log n⌋ do
19.             for all active processors P_j do in parallel
20.                 Parallel-Trickledown(P_j);
21.         return(H)
end;
```

Theorem 2.3.2 *An n element array $H[1..n]$ can be made into a heap in $O(\frac{n}{p})$ time using p $(1 \leq p \leq \lceil\frac{n}{\log n}\rceil)$ processors on an EREW PRAM.*

Proof. To show the correctness of the procedure, we note the loop in lines 18-20 guarantees that, eventually, all processors will become inactive. Therefore, we only need prove that:

when all processor P_i are inactive, H is a heap.

Suppose not; we find an index j $(1 \leq j \leq \lfloor\frac{n-1}{2}\rfloor)$ such that $H[j] > \min\{H[2j], H[2j+1]\}$. This cannot occur as the result of a swap in line 4 of *Parallel-Trickledown*. Hence, no processor P_i has had $current(P_i) = j$. By our processor allocation scheme specified in lines 4-8 of *Parallel-Initialize-Heap* it is impossible that $j \in [1, 2^k - 1]$. On the other hand, if $2^k \leq j \leq \lfloor\frac{n-1}{2}\rfloor$, then $H[j]$ belongs to precisely one of the heaps H_i constructed in lines 2-3 of *Parallel-Initialization-*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Heap, so the violation of the heap property cannot occur at $H[j]$. Therefore, we find a contradiction.

It is easy to see that no active processors can be involved in read or write conflicts in procedure *Parallel-Initialize-Heap* (note that inactive processors cannot create read or write conflicts). It follows that the computation can be performed in the EREW model of computation. To address the complexity, we note that, by our previous discussion, lines 2-3 take $O(\frac{n}{p})$ time using at most p processors. Lines 4-8 take $O(1)$ time and p processors.

Clearly, every invocation of the procedure Parallel-Trickledown(P_i) takes $O(1)$ time. Consequently, lines 9-16 run in $O(k)$ time, while lines 17-20 take $O(\log n)$ time using at most p processors. With this the proof of the theorem is complete. □

Initializing Min-max Heaps

Next, we propose to show that our technique can also be applied to min-max heaps. Consider, again, an array $H[1..n]$ that we want to make into min-max heap.

The first stage of our parallel min-max heap initialization algorithm is almost the same as Stage 1 in the previous section: we assign one processor to each subtree H_i ($2^k \leq i \leq 2^{k+1} - 1$) of H, and let every assigned processor make H_i into a min-max heap or a max-min heap depending on whether $\lfloor \log i \rfloor$ is even or odd. Since every H_i contains at most $\frac{4n}{p}$ keys, this takes $O(\frac{n}{p})$ time using the sequential algorithm in [8].

Once this step is completed, the p processors are reassigned to the first $2^k - 1$ elements of H. The idea of the second stage is to "grow" in parallel the min-max heaps H_i ($2^k \leq i \leq 2^{k+1} - 1$) into H itself, by adapting the Trickledown procedure in [8]. More precisely, every element of H at levels 0 through $k - 1$ will be trickled down two *min* (resp. *max*) levels in parallel in a pipelined fashion: we start with the elements at level $k - 1$ which will proceed to "trickle down" for two *min* (resp. *max*) levels; after this, in parallel, the elements at level $k - 2$ will begin the "trickle down",

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and so on. As for heaps, it is convenient to assume that processors move along with elements; every processor keeps moving down until it reaches the leaf level, where it becomes inactive.

For definiteness, we show the actions taken by a processor P performing a trickle down from a node w situated at *min* level (trickle down from a max level is similar):

---

if w has grandchildren **then**

  $v \leftarrow$ the grandchild with the smallest key field;

  if $key(v) < key(w)$ **then**

    Swap$(key(v), key(w))$;

    if $key(v) > key(parent(v))$ **then**

      Swap$(key(v), key(parent(v)))$;

  processor P moves down to v;

else /* w has no grandchildren */

  $u \leftarrow$ the child with the smallest key;

  if $key(u) < key(w)$ **then**

    Swap$(key(u), key(w))$;

  processor P moves down to u;

Theorem 2.3.3 *An n element array $H[1..n]$ can be made into a min-max heap in $O(\frac{n}{p})$ time using p ($1 \leq p \leq \lceil \frac{n}{\log n} \rceil$) processors on an EREW PRAM.*

Proof. To settle the correctness, we note that, eventually, all processors will reach a leaf node, thus becoming inactive. Therefore, we only need prove that when this happens, H is a min-max heap.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We proceed by contradiction. If the statement is false, then let j $(1 \leq j \leq \lfloor\frac{n-1}{2}\rfloor)$ stand for the subscript at which a violation of the properties of the structure occurs. Symmetry allows us to assume, without loss of generality, that $\lfloor\log j\rfloor$ is even (i.e. j is at a *min* level). Trivially, the following predicate is satisfied:

$$(H[j] > \min\{H[2j], H[2j+1]\}) or (H[j] > \min\{H[4j], H[4j+1], H[4j+2], H[4j+3]\})$$

That is, $H[j]$ is larger than the smallest of its children or grandchildren. Note that obviously, this cannot occur as a result of a swap in a trickle down. Consequently, no processor P has "visited" $H[j]$ during our construction algorithm. By our processor allocation scheme, it is impossible that $j \in [1, 2^k - 1]$. On the other hand, if $2^k \leq j \leq \lfloor\frac{n-1}{2}\rfloor$, then $H[j]$ belongs to precisely one of the min-max heaps H_i, a contradiction.

Furthermore, it is easy to see that our way of organizing computation makes read/write conflicts impossible, and so the computation can be performed on an EREW PRAM. By our allocation scheme, we only use p $(1 \leq p \leq \lceil\frac{n}{\log n}\rceil)$ processors; the running time is clearly bounded by $O(\frac{n}{p})$. □

Initializing Min-max-pair Heaps

Consider an array $H[1..n]$ as input. For $1 \leq i \leq n$, each element $H[i]$ of H has two fields $H[i].min$ and $H[i].max$. Therefore, the array H can be viewed as containing $2n-1$ or $2n$ keys altogether. In case H contains $2n-1$ keys, the *max* field of $H[n]$ contains a special symbol namely #.

To make H into a min-max-pair heap, we use a technique similar to the one developed previously. However, instead of having two stages, our parallel min-max-pair heap initialization algorithm contains three stages. Stage 1 is quite a reminiscent of Stage 1 of the algorithm in the heap initialization algorithm: as a first step, letting k stand for $\lfloor\log p\rfloor$, our algorithm assigns one processor to each of the subtrees H_i of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

H rooted at H_i ($2^k \leq i \leq 2^{k+1} - 1$). The unique processor assigned to H_i makes H_i into a min-max-pair heap in $O(\frac{n}{p})$time, using the sequential algorithm in [46].

After Stage 1, the processors are redistributed to the first $2^k - 1$ elements. Again, to avoid read and write conflicts among processors, we activate the processors in a pipelined fashion. That is, we start with the *min* fields at level $k-1$ which will proceed to trickle down for two levels: after this, in parallel, the *min* fields at level $k-2$ will begin to trickle down, and so on. Just as for the case of heaps, it is convenient to assume that processors move along with the key value in the *min* fields; every processor keeps moving down until it reaches the leaf level, at which point it becomes inactive. Finally, in Stage 3, all p processors are reassigned the first $2^k - 1$ elements to trickle down the *max* fields of these elements in parallel. To give the reader an idea, we show the actions taken by a processor P when performing a trickle down of the *min* field of a node w (trickle down on a *max* field is completely similar):

---

if $w.min > w.max$ **then**

  Swap($w.min, w.max$);

$v \leftarrow$ the child of w with the smallest *min* field (if exists);

if $w.min > v.min$ **then**

  Swap($w.min, v.min$);

Processor P moves down to v.

Theorem 2.3.4 *An n element array $H[1..n]$ can be made into a min-max-pair heap in $O(\frac{n}{p})$ time using p ($1 \leq p \leq \lceil\frac{n}{\log n}\rceil$) processors on an EREW PRAM.*

Proof. To settle the correctness we only need to prove that when all processors are inactive, H is a min-max-pair heap.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To begin, we notice that when all processors are inactive, $H[j].min < H[j].max$, for all subscripts j $(1 \leq j \leq \lfloor\frac{n-1}{2}\rfloor)$. If not, we find a subscript j such that $H[j].min > H[j].max$. Clearly, this situation cannot arise from a swap operation. Consequently, it must be the case that no processor P has "visited" $H[j]$ in Stage 2 or 3. By our processor allocation scheme it is impossible that $1 \leq i \leq 2^k - 1$; if $2^k \leq j \leq \lfloor\frac{n-1}{2}\rfloor$, then $H[j]$ belongs to precisely one of the min-max-pair heaps H_i, a contradiction.

To settle our main claim, we proceed by contradiction. If the statement is false, then let j $(1 \leq j \leq \lfloor\frac{n-1}{2}\rfloor)$ stand for the smallest subscript at which a violation of the properties of the min-max-pair heap occurs. Symmetry, together with our previous observation, allows us to assume that the following predicate is true:

$$(H[j].min > H[2j].min) or (H[j].min > H[2j].max)$$

Again, we note that this cannot occur as the result of a swap operation. It follows that no processor P has "visited" $H[j]$. By our processor allocation scheme, this is impossible for j to be in the range $[1 \leq j \leq 2^k - 1]$; on the other hand, if $2^k \leq j \leq \lfloor\frac{n-1}{2}\rfloor$, then $H[j]$ belongs to precisely one of the min-max-pair heaps H_i, a contradiction. Therefore, the above predicate cannot be true and the conclusion follows.

To see the complexity, we note that since no read or write conflicts can arise (due to our way to perform the trickle down operation), the computation can be performed on an EREW PRAM. The first stage of our algorithm runs in $O(\frac{n}{p})$ time. Stages 2-3 can be performed in $O(\log n)$ time using p processors. Therefore, the algorithm runs in $O(\frac{n}{p} + \log n) = O(\frac{n}{p})$ time using p $(1 \leq p \leq \lceil\frac{n}{\log n}\rceil)$ processors on an EREW PRAM. □

Initializing Deaps

Consider an array $H[1..n+1]$ with $H[1]$ undefined and such that $H[2], .., H[n+1]$ contain n arbitrary keys from a totally ordered universe. To motivate our approach,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

it is useful to note that when H becomes a deap $H[2]$ is the root of the corresponding min heap, while $H[3]$ is the root of the corresponding max-heap (see [15] for more details about the deap properties). Furthermore, it is easy to see that the entries at level i in the min-heap are 2×2^i, $2 \times 2^i + 1$, $2 \times 2^i + 2$, ..., $2 \times 2^i + 2^i - 1$ provided they exist; similarly, the entries at level i in the max-heap (i.e. the right subtree of $H[1]$) are 3×2^i, $3 \times 2^i + 1$, $3 \times 2^i + 2$, ..., $3 \times 2^i + 2^i - 1$.

To construct a deap we mirror the scheme presented at the beginning Section 2.3.2: in the first stage, with $k = \lfloor \log p \rfloor$, we assign one processor to each pair of subtrees $(H_t, H_{t'})$ of H rooted at $H[t]$ and $H[t']$ with $t = 2 \times 2^k + j$, and $t' = 3 \times 2^k + j$ $(1 \leq j \leq 2^k - 1)$. Every assigned processor makes its pair of subtrees into a deap in $O(\frac{n}{p})$ time as in [15]. After this, the p processors are redistributed to the first $2^k - 1$ elements of H. To make our description more transparent, it is helpful to imagine a deap as in Figure 2.6.

The nodes of H of the form $H[s]$ with $s = 2 \times 2^i + j$ such that $0 \leq i \leq k$ and $(1 \leq j \leq 2^i - 1)$ will be called *upper nodes*; and all the nodes of the form $H[s']$ with $s' = 3 \times 2^i + j$ such that such that $0 \leq i \leq k$ and $(1 \leq j \leq 2^i - 1)$ will be called *lower nodes*. We also call a pair subtrees (as shown in Figure 2.6) respectively rooted at $s = 2 \times 2^i + j$ and $s' = 3 \times 2^i + j$ $(0 \leq i \leq k, 1 \leq j \leq 2^i - 1)$ a diamond. It is easy to see that the diamond in Figure 2.6 is bottom-heavy, i.e. the smallest key is at node s and the largest key at node s'. Now the remaining part of our algorithm is divided into two stages.

In Stage 2, the processors are assigned to the upper nodes only. Every upper node is trickled down in parallel in a pipelined fashion. This trickle down differs from the standard one as we are about the explain. In Stage 3, the lower nodes receive processors and they will trickle down (moving up in Figure 2.6) as in Stage 2.

Again, to show the idea, we give the action by a processor P located at an upper node $H[s]$ when it performs a trickle down in Stage 2 (Stage is perfectly symmetric).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

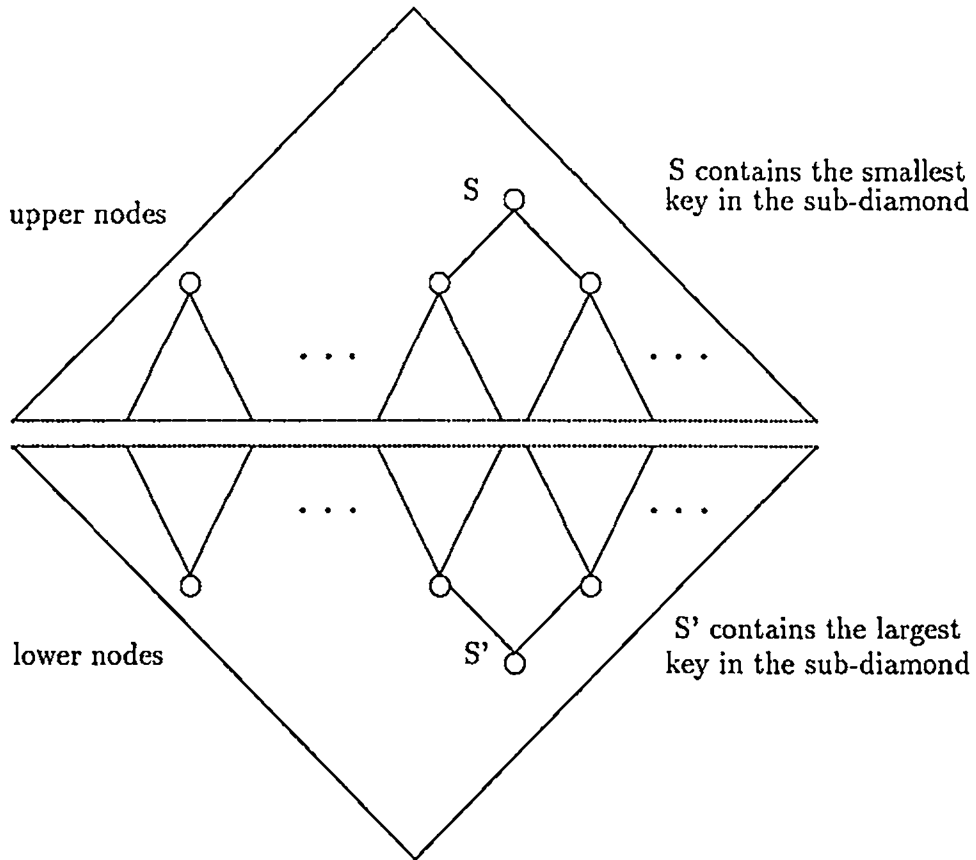


Figure 2.6: A new way to look at the deap

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For simplicity, we let s' stand for $s + 2^i$, s' is the other end of the diamond as shown in Figure 2.6.

---

if $key(H[s]) > key(H[s'])$ **then**

  Swap$(key(H[s]), key(H[s']))$;

$H[v] \leftarrow$ the child of $H[s]$ with the smallest key;

if $key(H[s]) > key(H[v])$ **then**

  Swap$(key(H[s]), key(H[v]))$;

Processor P moves down to $H[v]$.

Theorem 2.3.5 *An array $H[1..n+1]$ with $H[1]$ unused can be made into a deap in $O(\frac{n}{p})$ time using p $(1 \leq p \leq \lceil \frac{n}{\log n} \rceil)$ processors on an EREW PRAM.* □

Discussion

We presented in Section 2.3.2 a technique to develop optimal parallel initialization algorithms for a class of priority queues. As examples, we have applied our technique to initialize priority queues implemented by heaps, min-max heaps, min-max-pair heaps, and deaps. The basic idea is first to partition the original structure into a number of smaller substructures for which existing optimal sequential algorithms are readily applicable. After this first stage, the algorithm proceeds to grow these smaller structures, in parallel, to obtain the final structure. Our point is that this methodology works well for the data structures we discussed in this section. An interesting question is whether this methodology can be applied to other data structures.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 3

Multiple Search and Multiple Selection

Searching, merging, sorting and selection are the most fundamental problems in the design and analysis of algorithm [4, 1, 42]. In this chapter, we study the natural extension of these problems. The research results in this chapter can also be found in [52, 74]. Our research results are of both theoretical and practical interests because, on one hand, they reveal the relationships between these fundamental problems; on the other hand, they can be used as basic building blocks for developing algorithms to solve complex problems. For example, our optimal parallel solution for the multiple search problem can be used in Hagerup and Rub's parallel merging algorithm [33] to obtain the optimal implementation of their algorithm on the EREW PRAM.

The problems discussed in this chapter are the multiple search problem and the multiple selection problem. They are defined as follows:

Multiple Search Problem: Let $A = a_1, a_2, ..., a_n$ and $B = b_1, b_2, ..., b_m$ be two sorted sequences of items. Determine, for each a_i ($1 \leq i \leq n$), the item b_j such that $b_{j-1} \leq a_i \leq b_j$ (if necessary, we let $b_{j-1} = -\infty$ or $b_j = \infty$).

The multiple search problem is important because it generalizes two problems:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

searching and merging. It is easy to see that the multiple search problem is an extension of searching. We will show in Section 3.1 that it is also a generalization of merging. Therefore, this is a unification of searching and merging.

Multiple Selection Problem: Given an unsorted set S of n items from a totally ordered universe and a set Q of m integers $1 \leq q_1 < q_2 < ... < q_m \leq n$, answer the query "find the q_i-th smallest element in S" for $i = 1, 2, ..., m$.

The multiple selection problem is a natural extension of the traditional (single) selection problem. Moreover, if $m = n$ the problem is equivalent to sorting. Hence, the multiple selection problem bridges the gap between selection and sorting.

For simplicity of our presentation, we make the following assumptions: (1) If we say "a sequence is sorted," we mean that "the sequence is sorted in non-decreasing order (or in increasing order, whenever necessary)". (2) We limit our discussion to any set of items (e.g. real numbers) over which there is a natural linear order "$<$".

Before discussing the solutions for the multiple search and the multiple selection problem, we would like to give a brief literature review for the four fundamental problems, selection, searching, merging and sorting. The problem of selection is to find the k-th smallest element in a sequence of n elements (unsorted). It is well known that the sequential complexity of this problem is $\Theta(n)$. On the comparison model [70] (in this model, only the time used for comparisons is counted), the following results have been obtained: upper bounds of $O((\log\log n)^2)$ time using n processor by Cole and Yap [22], and $O(\log\log n)$ time using n processors [3]; a lower bound of $\Omega(\log\log n)$ time using n processors. On the PRAM, the following results have been achieved: upper bounds of $O(\log n \log\log n)$ time using $O(\frac{n}{\log n \log\log n})$ processors on the EREW PRAM by Vishkin (reported in [19]), or in $(\log n \log^* n)$ time using $O(\frac{n}{\log n \log^* n})$ processors on the EREW PRAM [19], or in $O(\frac{\log n \log^* n}{\log\log n})$ time using optimal number of processors on the CRCW PRAM [19]. The problem of searching is to look up an item (e.g. a number) in a sorted sequence of size n. The following

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

results have been achieved: $\Theta(\frac{\log(n+1)}{log(p+1)})$ time using p processors on the CREW PRAM by Kruskal [43]; an upper bound $O(\frac{\log(n+1)}{log(p+1)} + \log p)$ time using p processors on the CREW PRAM by Kruskal [43]. To merge two sorted sequences of size n, optimal parallel algorithms have been proposed: $\Theta(\log n)$ time using $O(\frac{n}{\log n})$ processors on the EREW PRAM [7, 33], or in $\Theta(\log\log n)$ time using $O(\frac{n}{\log\log n})$ processors on the CREW PRAM [43]. For parallel sorting, the following results have achieved: a sorting network of $O(n)$ and depth $O(\log n)$ by Ajtai et al. [2]; $\Theta(\log n)$ time using n processors on both EREW and CREW PRAMs by Cole [20]; and upper bound $O(\frac{\log n}{\log\log(1+\frac{2p}{n})})$ time using $2n \leq p \leq n^2$ processors on the CRCW PRAM [20]; and $\Theta(\frac{\log n}{\log\log(1+\frac{p}{n})})$ processors in a parallel comparison model [9].

In the rest of this chapter, we first discuss the multiple search problem in Section 3.1; we then study the multiple selection problem in Section 3.2.

3.1 Multiple Search

We present parallel solutions to the following problem. The materials in this chapter also appear in [74]. Let $A = a_1, a_2, ..., a_n$ and $B = b_1, b_2, ..., b_m$ be two sorted sequences of items. It is required to determine, for each a_i ($1 \leq i \leq n$), the item b_j such that $b_{j-1} \leq a_i \leq b_j$ (if necessary, we let $b_{j-1} = -\infty$ or $b_j = \infty$). Akl and Meijer [5] first considered this problem under the assumption $m > n$, and named it the multiple search problem. For convenience, we release the restriction, $m > n$, in their definition, and still use the name, multiple search problem.

An easy way to solve the multiple search problem is by merging sequences A and B. Merging two sorted sequences of sized m and n takes $O(\log(m+n))$ sequential time, or $O(\log(m+n))$ time using $O(\frac{m+n}{\log(m+n)})$ processors on an EREW PRAM (using the parallel algorithms in [33]). The cost of this solution is $O(m+n)$, which is far from optimal when $m \gg n$ (e.g. When $n = 1$, binary search takes only $O(\log m)$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sequential time).

Another easy solution is to do binary search in B for each item of A; this takes $O(n \log m)$ sequential time. It can also be implemented in $O(\log m)$ time using n processors on the CREW PRAM (n processors each carrying an item of A to do (simultaneously) binary search in B). By simulating the CREW PRAM algorithm on an EREW PRAM, we can obtain an algorithm for the problem on an EREW PRAM which takes $O(\log m \log n)$ time using n processors.

Akl and Meijer [5] proposed an algorithm for the problem (assuming $m > n$) which takes $O(\frac{\log m \log n}{\log \log m})$ time using n processors on an EREW PRAM. Their solution was then extended to the case where fewer than n processors are available. This yielded an EREW PRAM algorithm whose cost is $O(n \log m)$ using p processors, where $p \le \min\{\frac{n \log \log m}{\log n}, n\}$ [5].

In this section, we first propose a parallel algorithm for the multiple search problem. This algorithm improves those in [5] by achieving larger speed-up without increasing the cost. We then combine the ideas of our first algorithm with those of the optimal parallel merging algorithm in [7, 33], and present a better algorithm for the problem. Our second algorithm improves the first algorithm in the sense that it can run as fast as the first algorithm while using fewer processors. The second algorithm is optimal.

3.1.1 Preliminaries

To simplify our presentation, we borrow some terminology of Cole [20]. Let A and B be two sorted sequences, and let f be an item. We define an item f to be *ranked* in B, if we know the item b_j of B such that $b_{j-1} \le f < b_j$ (if necessary, we let $b_{j-1} = -\infty$ or $b_j = \infty$) We say that f is *straddled* by the b_{j-1} and b_j; and we define the *rank* of f in B to be $j-1$. We define A to be *ranked* in B (denoted $A \to B$) if each item of A is ranked in B. and define A and B are *cross-ranked* if both $A \to B$ and $B \to A$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We use $A \bigcup B$ to denote the sorted merged list of all items in A or B [20]. With the terminologies above, the multiple search problem is actually a matter of computing $A \rightarrow B$.

We would also like to use some terminologies of Hagerup and Rub [33]. Let A and B be two sorted sequences. Define the *partition* of B induced by A to be the $(|A|+1)-tuple$ $(B_0, B_1, ..., B_{|A|})$ where B_i, for $i = 0, 1, ..., |A|$, is the subsequence of B consisting of all items of B with rank i in A. We can represent a partition $(B_0, B_1, ..., B_k)$ in $O(k)$ space by storing for $i = 1, 2, ..., k$ an indication of whether $B_i = \phi$; if $B_i \neq \phi$, the ranks in B of the minimal and maximal items of B_i [33]. We also denote the partition of B induced by A as $(B_0{}^{(A)}, B_1{}^{(A)}, ..., B_{|A|}{}^{(A)})$, where $|A|$ is the length of sequence A.

Lemma 3.1.1 *(a modified version of Proposition 2.1 in [33]). Let A and B be two sorted sequences. Given $A \rightarrow B$, the partition of B induced by A can be computed in constant time using $|A|$ processors on the EREW PRAM.* □

It is pointed out in [20] that cross-ranking and merging are equivalent concepts in the following sense. Let A and B be two sorted sequences. For every item of A or B, its position (rank) in the merged sequence $A \bigcup B$ is the sum of its ranks in A and B. If A and B are cross-ranked, then $A \bigcup B$ can be computed without extra comparison. On the other hand, cross-ranking of A and B can be computed by merging A and B. From this point of view, the multiple search problem is a generalization of merging.

3.1.2 Sequential Complexity Bounds for Multiple Search

For the analysis of our parallel algorithms, we need an optimal sequential solution and the sequential time complexity bound of the problem. Let A of size n and B of size m be the input of the multiple search problem. Consider two case:

(i) If $m \leq n$, by the definition of the problem, any sequential solution requires at

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

least $\Omega(n)$ time. Note that $A \rightarrow B$ can be computed by merging A and B, which requires $O(n)$ time. Therefore, in this case the sequential time complexity of the problem is $\Theta(n)$.

(ii) If $m > n$, the sequential time complexity of the problem was shown to be $\Theta(n \log \frac{2m}{n})$ (see "the generalized binary algorithm g", Theorem 1, and Theorem 2 in [37]). To make our presentation self-contained, we give a simpler proof of this result as the following. First, we show that the $O(n \log \frac{2m}{n})$ sequential time is a lower bound. Then we show that the $O(n \log \frac{2m}{n})$ is also an upper bound. To show the lower bound $\Omega(n \log \frac{2m}{n})$, we use the decision tree model [1]. Given two sorted sequences A of size n and B of size m, there are $\begin{pmatrix} m+n \\ n \end{pmatrix}$ possibilities that the items of A are straddled by the items of B. Therefore, on the decision tree model, any comparison algorithm for the problem requires at least $O(\log \begin{pmatrix} m+n \\ n \end{pmatrix})$ sequential time.

$$\log \begin{pmatrix} m+n \\ n \end{pmatrix} = \log \frac{(m+n)(m+n-1)...(m+1)}{n(n-1)...1} \geq \log \begin{pmatrix} m+n \\ n \end{pmatrix}^n = n \log(1 + \frac{m}{n})$$

$$O(n \log(1 + \frac{m}{n})) = O((n \log \frac{2m}{n})$$

From the discussion above, we know that $O(n \log \frac{2m}{n})$ is a sequential lower bound for the problem. To show $O(n \log \frac{2m}{n})$ is also an upper bound, consider the following algorithm:

Algorithm Sequential-multiple-search;

Step 1. Extract from B a sequence B' of $n-1$ items, which (almost) equally divide B into n subsequences of size $\frac{m}{n}$ each;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Step 2. Merge A and B' so as to determine, for each item of A, the subsequence of B to which the item belongs;

Step 3. For each item of A, do a binary search in the corresponding subsequence of B;

The correctness of this algorithm is easily seen. Steps 1-2 take $O(n)$ time. Step 3 takes $O(n \log \frac{m}{n})$ time. So the algorithm takes $O(n \log \frac{2m}{n})$ time. Hence, $O(n \log \frac{2m}{n})$ is also a sequential time upper bound of the problem.

Summarizing two cases above, we have the following lemma:

Lemma 3.1.2 *Let A and B be two sorted sequences with sizes n and m, respectively. $A \rightarrow B$ can be computed in $\Theta(n)$ sequential time when $m \leq n$, or in $\Theta(n \log \frac{2m}{n})$ sequential time when $m > n$.* □

3.1.3 Parallel Multiple Search Using n Processor

We now present the parallel solutions using n processors. Solving the problem on a CREW PRAM with n processors is straightforward: we can simply implement the sequential algorithm in Section 3.1.2 on an CREW PRAM.

Theorem 3.1.1 *Let A and B be two sorted sequences with sizes n and m, respectively. $A \rightarrow B$ can be solved on a CREW PRAM with n processors in in $O(\log \log n)$ time when $m \leq n$, or in $O(\log \log n + \log \frac{m}{n})$ time when $m > n$.*

Proof. Assume the problem is solved on a CREW PRAM with n processors. When $m \leq n$, we solve the problem by Kruskal's merging algorithm [43] which takes

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$O(\log\log n)$ time. When $m > n$, we parallelize algorithm Sequential-multiple-search in the previous section. Step 1 takes $O(1)$ time; Step 2 takes $O(\log\log n)$ time; and Step 3 needs $O(\log\frac{m}{n})$ time. □

Solving the problem on an EREW PRAM with n processors is complicated because we want to avoid concurrent memory access. We modify the sequential algorithm in Section 3.1.2 so that the new algorithm can be implemented efficiently on an EREW PRAM. Let A and B be two sorted sequences with sizes n and m, respectively. When $m \leq n$ we solve the problem by parallel merging because the cost of merging is $O(m+n) = O(n)$, which is optimal in this case. When $m > n$ merging does not guarantee an efficient solution, so our strategy is to reduce a search in a large range to a search in a small range. For this reason, we first divide sequence B into n subsequences of $\frac{m}{n}$ each and determine, for every item of sequence A, the subsequence of B to which the item belongs. We then group the items of A that belong to the same subsequence of B, and thus divide A into segments, each containing all the items of A that belong to the same subsequence of B. Finally, in parallel, we recursively continue the searches for all the segments of A in their corresponding subsequences of B. More precisely, the algorithm is spelled out as follows:

Algorithm EREW-PRAM-multiple-search. /* n processors are used */

Input: $A = a_1, a_2, ..., a_n$ and $B = b_1, b_2, ..., b_m$; **Output:** $A \rightarrow B$;

Step 1. **if** $n = 1$ **then** compute $A \rightarrow B$ by sequential binary search **return**;

Step 2. **if** $m \leq n$ **then** compute $A \rightarrow B$ by parallel merging **return**;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Step 3. Divide B into n subsequences, $B_1, B_2, ..., B_n$, of size $\frac{m}{n}$ each. Let $B' = b'_1, b'_2, ..., b'_{n-1}$, where b'_i is the last item of B_i, for $i = 1, ..., n-1$; That is, B' is the list of last items of the first $n-1$ subsequences. Compute $A \rightarrow B'$ by parallel merging;

Step 4. Find all items of A, $a_{j_1}, a_{j_2}, ..., a_{j_t}$ with following properties:

(1) $1 \leq j_1 < j_2 < ... < j_t < n$;

(2) for $1 \leq i \leq t$, a_{j_i} and a_{j_i+1} have differnt ranks in B'.

Step 5. Divide A into $t+1$ segments, $A_1, A_2, ..., A_{t+1}$ such that $A_1 = a_1, ..., a_{j_1}$, $A_{i+1} = a_{j_i+1}, ..., a_{j_{i+1}}$, $(i = 1, 2, ..., t-1)$; and $A_{t+1} = a_{j_t+1}, ..., a_n$. Note that by properties (1)-(2) in Step 4, items of the same segment of A have the same rank in B', while items of different segments of A have different ranks in B'.

Step 6. **for** $i = 1, 2, ..., t+1$ **do** in parallel

r_i =(the rank in B' of the items of A_i) +1;

Compute $A_i \rightarrow B_{r_i}$ (recursively, by $| A_i |$ processors);

end.

Theorem 3.1.2 *Let A and B be two sorted sequences with sizes n and m, respectively. $A \rightarrow B$ can be computed in $O(\log m + \log n)$ time using n processors on an EREW PRAM.*

Proof. With the correctness of the algorithm being obvious, we turn to the time complexity. Assume that the algorithm is implemented on an EREW PRAM with

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

n processors. The complexity of the algorithm is analyzed as follows. Step 1 takes $O(\log m)$ time. Steps 2-3 can be performed in $O(\log n)$ time using one of the merging methods in [11]. Step 4 can be implemented like this: for all $i = 1, ..., n-1$, item a_i checks with a_{i+1}, and a_i marks itself if they have different ranks in B'. Collecting the "marked" items in A is an instance of the parallel prefix problem. Using the results in [21], Step 4 can be implemented in $O(\log n)$ time. By Lemma 3.1.1, Step 5 can be performed in $O(1)$ time (with the result from Step 4). Let $T(m,n)$ denote the time complexity of the algorithm implemented on an EREW PRAM with n processors; then the time complexity of Step 6 is

$\max\{T(\frac{m}{n}, j_1), T(\frac{m}{n}, j_2 - j_1), ..., T(\frac{m}{n}, j_t - j_{t-1}), T(\frac{m}{n}, n - j_t)\}$. The time complexity of the algorithm satisfies:

case 1. $m \leq n$: $T(m,n) = O(\log n)$

case 2. $m > n = 1$: $T(m,n) = O(\log m)$

case 3. $m > n > 1$:

$$T(m,n) = O(\log n) + \max\{T(\frac{m}{n}, j_1), T(\frac{m}{n}, j_2 - j_1), ..., T(\frac{m}{n}, j_t - j_{t-1}), T(\frac{m}{n}, n - j_t)\}$$

, where $1 \leq j_1 < j_2 < ... < j_t < n$;

We claim that $T(m,n) = O(\log m + \log n)$. When $m \leq n$, as given in case 1, $T(m,n) = O(\log n) = O(\log m + \log n)$. When $m > n = 1$, by case 2, $T(m,n) = O(\log m) = O(\log m + \log n)$. When $m > n > 1$, we prove the claim by induction on m as follows. (i) When $m = 3$, the claim is obviously true. (ii) Assume that $T(m,n) = O(\log m + \log n)$ for $m < k$. (iii) When $m = k$, we know that $\frac{m}{n} < k$. By case 3 and the induction hypothesis (or case 1 if $(j_i - j_{i-1}) \geq \frac{m}{n}$), we have,

$$T(m,n) = O(\log n) + \max\{\log\frac{m}{n} + \log(j_1), \log\frac{m}{n} + \log(j_2 - j_1), ..., \log\frac{m}{n} + \log(n - j_t)\}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$T(m,n) = O(\log n) + O(\log \frac{m}{n}) + O(\max\{\log(j_1), \log(j_2 - j_1), ..., \log(n - j_t)\})$$

$$\text{where } 1 \leq j_1 < j_2 < ... < j_t < n;$$

Because $\max\{j_1, j_2 - j_1, ..., n - j_t\} \leq n$, $T(m,n) = O(\log m + \log n)$. □

We note that both our EREW PRAM and CREW PRAM solutions in this section are not cost optimal. To see the reason, let us focus on the EREW PRAM solution. As we know, merging can be used to solve the multiple search problem. Using the optimal EREW PRAM merging algorithm in [7, 33], two sorted sequences, A and B with sizes n and m, respectively, can be merged in $O(\log(m+n))$ time using p ($p \leq \frac{m+n}{\log(m+n)}$) processors on an EREW PRAM [33]. It is obvious that, only when $m > n \log n$, Algorithm *EREW-PRAM-multiple-search* leads to a better solution than using the optimal parallel merging algorithms in [7, 33]. Therefore, to improve our algorithms, we will combine our ideas with those in the optimal parallel merging algorithms of [7, 33].

3.1.4 Parallel Multiple Search Using Fewer Processors

In this section, we develop a new algorithm which combines the ideas of our algorithms in Section 3.1.3 and those in the parallel merging algorithms [7, 33]. Let A and B be two sorted sequences with sizes n and m, respectively. Assume that p ($p \leq \min\{m, n\}$) processors are available. The main idea of the algorithm is to divide sequence A into $2p-1$ segments each of size at most $\frac{n}{p}$, and B into $2p-1$ subsequences each of size at most $\frac{m}{p}$; the division of A and B will be made in such a way that, for $i = 0, 1, ..., 2p-2$, the i-th segment of A belongs to the i-th subsequence of B. Based on these divisions,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

searching each segment of A in its corresponding subsequence of B will be solved sequentially (every processor is responsible for at most two segments). The algorithm is spelled out as follows.

Algorithm Adaptive-parallel-multiple-search.

/* $p \leq \min\{m, n\}$ processors are used */

Input: two sorted sequences, $A = a_1, a_2, ..., a_n$ and $B = b_1, b_2, ..., b_m$;

Output: $A \rightarrow B$;

Step 1. Let A' be a list of $p - 1$ items of A which equally split A. That is,

$A' = a_{j_1}, a_{j_2}, ..., a_{j_{p-1}}$, where $j_i = \frac{i \times n}{p}$ for $i = 1, ..., p - 1$.

Let B' be a list of $p - 1$ items of B which equally split B. That is,

$B' = b_{k_1}, b_{k_2}, ..., b_{k_{p-1}}$, where $k_i = \frac{i \times m}{p}$ for $i = 1, ..., p - 1$.

Compute $C = A' \bigcup B'$;

Step 2. Compute $C \rightarrow A$, and $C \rightarrow B$ using our first algorithm.

(Note that $| C | = 2p - 2$)

Step 3. Construct partitions $(A_0{}^{(C)}, A_1{}^{(C)}, ..., A_{|C|}{}^{(C)})$, and

$(B_0{}^{(C)}, B_1{}^{(C)}, ..., B_{|C|}{}^{(C)})$;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Step 4. **for** $i = 0, ..., \mid C \mid$ **do in parallel**

Compute $A_i^{(C)} \rightarrow B_i^{(C)}$;

end.

Theorem 3.1.3 *Let A and B be two sorted sequences with sizes n and m, respectively. $A \rightarrow B$ can be computed on a CREW PRAM with p ($p \leq \min\{m, n\}$) processors, in $O(\log\log p + \frac{n}{p})$ time when $m \leq n$, or in $O(\log\log p + \frac{m}{p} + \frac{n}{p}\log\frac{2m}{n})$ time when $m > n$.*

Proof. The correctness of our second algorithm is obvious. Consider the time complexity of the algorithm implemented on an CREW PRAM with p processors. Step 1 takes $O(\log\log p)$ time using one of the merging algorithms in [43]. By Theorem 3.1.1, Step 2 takes $O(\log\log p + \log\frac{n}{p} + \log\frac{m}{p})$ time. Step 3 takes $O(1)$ time by Lemma 3.1.1. We now consider Step 4. Due to the choices of A' and B' in Step 1, and $C = A' \bigcup B'$, we have $\mid A_i^{(C)} \mid \leq \frac{n}{p} \mid B_i^{(C)} \mid \leq \frac{m}{p}$, for $i = 0, 1, ..., \mid C \mid$. By Lemma 3.1.2, computing $A_i^{(C)} \rightarrow B_i^{(C)}$ ($0 \leq i \leq \mid C \mid$) can be done in $O(\frac{n}{p})$ time when $m \leq n$, or in $O(\frac{n}{p}\log\frac{2m}{n})$ time when $m > n$. Because p processors are available, Step 4 can be performed in $O(\frac{n}{p})$ time when $m \leq n$, or in $O(\frac{n}{p}\log\frac{2m}{n})$ time when $m > n$. Let $T(m, n, p)$ denote the time complexity of the algorithm implemented on a CREW PRAM with p processors. Adding up the time required by all the steps, we have,

$$T(m,n,p) = \begin{cases} O(\log\log p + \log\frac{m}{p} + \log\frac{n}{p} + \frac{n}{p}) & m \leq n \\ O\left(\log\log p + \log\frac{m}{p} + \log\frac{n}{p} + \frac{n}{p}\log\frac{2m}{n}\right) & m > n \end{cases}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$T(m,n,p) = \begin{cases} O(\log\log p + \frac{n}{p}) & p \le m \le n \\ O\left(\log\log p + \log\frac{m}{p} + \frac{n}{p}\log\frac{2m}{n}\right) & p \le n < m \end{cases}$$

Summarizing the discussion above, we have proved the theorem. □

Corollary 3.1.1 *Let A and B be two sorted sequences with sizes n and m, where $n < m$. $A \to B$ can be computed in $O(\log\log n + \log\frac{m}{n})$ time using $O(\frac{n\log\frac{2m}{n}}{\log\frac{m}{n}+\log\log n})$ processors on a CREW PRAM.*

Proof. The result follows when $p = O(\frac{n\log\frac{2m}{n}}{\log\frac{m}{n}+\log\log n})$ in the theorem above. □.

Theorem 3.1.4 *Let A and B be two sorted sequences with sizes n and m, respectively. $A \to B$ can be computed on an EREW PRAM with p ($p \le \min\{m,n\}$) processors, in $O(\log n + \frac{n}{p})$ time when $m \le n$, or in $O(\log m + \frac{n}{p}\log\frac{2m}{n})$ time when $m > n$.*

Proof Consider the time complexity of algorithm *Adaptive-parallel-multiple-search* implemented on an EREW PRAM with p ($p \le \min\{m,n\}$) processors. Using one of the merging methods in [11], computing $C = A'\bigcup B'$ in Step 1 requires $O(\log p)$ time. By Theorem 3.1.2, Step 2 can be implemented in $O(\log m + \log n + \log p)$ using p processors on an EREW PRAM. By Lemma 2.1, Step 3 takes $O(1)$ time. We know from the proof the previous theorem that Step 4 takes $O(\frac{n}{p})$ time when $m \le n$, or in $O(\frac{n}{p}\log\frac{2m}{n})$ time when $m > n$. Let $T(m,n,p)$ denote the time complexity of the algorithm implemented on a CREW PRAM with p processors. Adding up the time required by all the steps, we have,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$T(m,n,p) = \begin{cases} O(\log m + \log n + \log p + \frac{n}{p}) & m \leq n \\ O\left(\log m + \log n + \log p + \frac{n}{p} \log \frac{2m}{n}\right) & m > n \end{cases}$$

$$T(m,n,p) = \begin{cases} O(\log n + \frac{n}{p}) & p \leq m \leq n \\ O\left(\log m + \frac{n}{p} \log \frac{2m}{n}\right) & p \leq n < m \end{cases}$$

Summarizing the discussion above, we have the theorem. □

Corollary 3.1.2 *Let A and B be two sorted sequences with sizes n and m, where $n < m$. $A \rightarrow B$ can be computed in $O(\log m)$ time using $O(\frac{n \log \frac{2m}{n}}{\log m})$ processors on an EREW PRAM.*

Proof. The result follows when $p = O(\frac{n \log \frac{2m}{n}}{\log m})$ in the theorem above. □.

3.1.5 Discussion

We have developed parallel algorithms for the multiple search problem. In Section 3.1.3, we gave two parallel solutions using n processors. The EREW PRAM algorithm improves the algorithms of Akl and Meijer [5]. Our CREW PRAM solution runs faster than our EREW PRAM algorithm using the same number of processors.

In Section 3.1.4, we combined the ideas of our algorithm in Section 3.1.3 with those of the optimal parallel merging algorithms in [7, 33], and presented an adaptive parallel algorithm using p processors. To appreciate the performance of algorithm *Adaptive-parallel-multiple-search* in Section 3.1.4, we notice the following cases:

(a) By Lemma 3.1.2, the algorithm running on an EREW PRAM is cost optimal, if $m > n$ and $p \leq \frac{n \log \frac{2m}{n}}{\log m}$, or if $m \leq n$ and $p \leq \frac{n}{\log n}$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(b) The algorithm running on a CREW PRAM is cost optimal, if $m > n$ and $p \leq O(\frac{n \log \frac{2m}{n}}{\log \frac{m}{n} + \log \log n})$, or if $m \leq n$ and $p \leq \frac{n}{\log \log n}$.

(c) When $p = n < m$ both of our algorithms have the same performance. Therefore, our second algorithm is a generalization of our first algorithm.

(d) Consider the case when $n < m$. On an EREW PRAM, algorithm *Adaptive-parallel-multiple-search* needs only $\frac{n \log \frac{2m}{n}}{\log m}$ processors to run in $O(\log m)$ time, while our first algorithm needs n processors to achieve the same speed. On a CREW PRAM, our first algorithm runs in $O(\log \log n + \log \frac{m}{n})$ using n processors, yet our second algorithm needs only $O(\frac{n \log \frac{2m}{n}}{\log \frac{m}{n} + \log \log n})$ processors to achieve the same speed. Hence, The algorithm in Section 3.1.4 is also an improvement over the algorithms in Section 3.1.3.

3.2 Multiple Selection

In this section, we consider the following problem (the materials in this section also appear in [52]): Given an unsorted set S of n items from a totally ordered universe and a set Q of m integers $1 \leq q_1 < q_2 < ... < q_m \leq n$, answer the query "find the q_i-th smallest element in S" for $i = 1, 2, ..., m$.

For convenience, we assume that Q is given in an array with elements sorted in increasing order. To avoid tedious but inconsequential complications, we further assume that all the elements in S are distinct. It is well known that the sequential complexity of single selection is $O(n)$ [1]. So we assume familiarity with the details of the traditional single selection algorithm. Here, the selection algorithm in [1] will

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

be referred to as $Select(A, k)$, which returns the k-th smallest key in set A.

Before we present our parallel EREW PRAM algorithm for the multiple selection problem in Section 3.2.3, we first give an efficient sequential solution to the problem in Section 3.2.1; we then discuss the parallel complexities of the single selection on PRAMs with Concurrent Write (i.e. the EREW and CREW PRAMs) in Section 3.2.2.

3.2.1 A Sequential Multiple Selection Algorithm

We present an efficient sequential algorithm to the multiple selection problem. The idea is very simple: Let q stand for $Q[\lceil \frac{m}{2} \rceil]$; using procedure *Select*, we find the q-th smallest element z in S, and compute the sets $S_1 = \{x \in S \mid x \leq z\}$ and $S_2 = \{x \in S \mid x > z\}$. At the same time, we partition Q into Q_1 containing the first $\lceil \frac{m}{2} \rceil - 1$ entries in Q, and Q_2 containing the last $\lfloor \frac{m}{2} \rfloor$ entries in Q.

For further reference, we note that all the queries in Q_1 pertain to S_1, while all the queries in Q_2 refer to S_2.

This process is continued recursively until the number of queries that have to be answered on any particular subset of S is 1: at that time, the corresponding query is answered using procedure *Select* [1]. The details are presented in the following procedure.

Procedure Sequential-multiple-selection($S, Q[1, m]$);

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Input: a set S of keys; a global array Q of queries with

elements sorted in increasing order

Output: a global array R with $R[i]$ containing the answer to query $Q[i]$;

1. **if** $m = 0$ **then** return;
2. $t \leftarrow \lceil \frac{m}{2} \rceil$;
3. $q \leftarrow Q[t]$;
4. $z \leftarrow Select(S, q)$;
5. $R[t] \leftarrow z$;
6. $S_1 \leftarrow \{x \in S \mid x < z\}$;
7. $S_2 \leftarrow \{x \in S \mid x > z\}$;
8. **for** $j \leftarrow t+1$ **to** m **do** /* update queries */
9. $\quad Q[j] \leftarrow Q[j] - \mid S_1 \mid -1$;
10. Sequential-multiple-selection($S, Q[1, t-1]$);
11. Sequential-multiple-selection($S, Q[t+1, m]$);
12. return(R);

end;

Theorem 3.2.1 *Given a set S of n elements and a set Q of m queries with $m \leq n$, the multiple selection problem can be solved in $O(n \log 2m)$ sequential time.*

Proof. The correctness being obvious we turn to the complexity. Let $T(n, m)$ stand for the total running time of this procedure. Since lines 4, 6-9 take $O(n)$ time. The recurrence system describing the behavior of $T(n, m)$ is given by

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$T(n,m) \leq c'n + T(| S_1 |, \lceil \frac{m}{2} \rceil - 1) + T(| S_2 |, \lfloor \frac{m}{2} \rfloor)$$

We claim that for some positive constant c,

$$T(n,m) \leq cn \log 2m \tag{3.2.1}$$

The proof of (3.2.1) is by induction. The basis being trivially satisfied, the induction hypothesis allows us to write

$$T(| S_1 |, \lceil \tfrac{m}{2} \rceil - 1) \leq c \mid S_1 \mid \log 2(\lceil \tfrac{m}{2} \rceil - 1) \leq c \mid S_1 \mid \log m \tag{3.2.2}$$

and

$$T(| S_2 |, \lfloor \tfrac{m}{2} \rfloor) \leq c \mid S_2 \mid \log 2 \lfloor \tfrac{m}{2} \rfloor \leq c \mid S_2 \mid \log m \tag{3.2.3}$$

Therefore, by (3.2.2) and (3.2.3) combined,

$$T(n,m) \leq c'n + cn \log m$$

If we write $c = c'$ we have $T(n,m) \leq cn \log 2m$, and (3.2.1) is proved. With this, the proof of the theorem is complete. □

3.2.2 Time Bounds for Single Selection on Exclusive Write PRAMs

In our parallel algorithm for the multiple selection problem in the next section, we need as a subroutine an efficient parallel single selection algorithm on the EREW PRAM. We note that such an algorithm has been proposed by Cole in [19]. In particular, Cole's result can be specified by the following theorem:

Theorem 3.2.2 *(Cole [19]). Given a set S of n items (unsorted), the k-th smallest item in S can be found in* $(\log n \log^* n)$ *time using* $O(\frac{n}{\log n \log^* n})$ *processors on the*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

EREW PRAM. (Here, $\log^{(1)} n = \log n$, ..., $\log^{(i)} n = \log(\log^{(i-1)})n)$, *and* $\log^* n = \min\{i \mid \log^{(i)} n \leq 1\}$ *).* □

Corollary 3.2.1 *Given a set S of n items (unsorted), the k-th smallest item in S can be found in $O(\frac{n}{p})$ time using p ($p \leq \frac{n}{\log n \log^* n}$) processors on an EREW PRAM.* □

To appreciate Cole's result, we need to know the lower time bound of the single selection problem on the PRAMs which do not allow concurrent writes.

Theorem 3.2.3 *(Cook, Dwork, and Reischuk [25]). On an CREW PRAM, every parallel algorithm that computes the logical "or" of n bits requires at least $\Omega(\log n)$ time, no matter how many processors are used.* □

Corollary 3.2.2 *Let S be a set of n items (unsorted) from a totally ordered universe. On a CREW PRAM, or an EREW PRAM, every parallel algorithm that computes the k-th (for any integer k) smallest item in S requires at least $\Omega(\log n)$ time, no matter how many processors are used.*

Proof. Assume the computation model is a CREW PRAM which is stronger than an EREW PRAM. The following three-step procedure reduces the computation of the logical "or" of n bits to the problem of selecting the k-th (for any integer k) smallest item.

Input : $B[1..n]$ of 0/1 bits; *Output*: logical "or" of the bits of $B[1..n]$;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Step 1. Compute array $A[1..n+k-1]$ such that $A[i] = 1 - B[i]$ $(i = 1, ..., n)$, and

$A[i] = 0$ $(i = n+1, ..., n+k-1)$;

Step 2. $R \leftarrow$ the k-th smallest number in array $A[1..n+k-1]$;

Step 3. Return$(1 - R)$;

It is easy to see that this procedure returns "1" if and only if the logical "or" of the bits in $B[1..n]$ is "1". Let $T(n,k)$ be the time complexity of the fastest parallel algorithm to select the k-th smallest item in a set of size n. Obviously, $T(n,k)$ dominates the time complexity of this procedure. By Theorem 3.2.3, $T(n,k)$ is at lest $\Omega(\log n)$, no matter how many processors are used. □

3.2.3 A Parallel Multiple Selection Algorithm

We are now in a position to explain how to solve the multiple selection problem on an EREW PRAM. Our parallel procedure is, in fact, a simple parallelization of the sequential multiple selection procedure presented previously. For convenience, we import the whole context and the notation used in the description of our sequential procedure. For completeness, however, we give the details of the parallel version as well.

Procedure Parallel-multiple-selection$(S, Q[1, m], p)$;

Input: a set S of keys; a global array Q of queries with

elements sorted in increasing order; p is the number of processors used

Output: a global array R with $R[i]$ containing the answer to query $Q[i]$;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1.   if m = 0 then return;
2.   t ← ⌈m/2⌉;
3.   if p = 1 then
4.      Sequential-multiple-selection(S, Q[1, m]);
5.   q ← Q[t];
6.   find in parallel the q-th smallest element z of S;
7.   R[t] ← z;
8.   S1 ← {x ∈ S | x < z};
9.   S2 ← {x ∈ S | x > z};
10.  for j ← t + 1 to m do /* update queries */
11.      Q[j] ← Q[j]− | S1 | −1;
12.  p1 ←| S1 | × p/n;
13.  p2 ←| S2 | × p/n;
14.  do in parallel
15.      Parallel-multiple-selection(S, Q[1, t − 1], p1);
16.      Parallel-multiple-selection(S, Q[t + 1, m], p2);
17.  return(R);
end;
```

Theorem 3.2.4 *Given a set S of n elements and a set Q of m queries where $m \leq n$, the multiple selection problem can be solved in $O(\frac{n}{p}\log 2m)$ time using p ($p \leq \frac{n}{\log n \log^* n}$) processors on an EREW PRAM.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Proof. The correctness being obvious we trun to the complexity. Since no read or write conflicts occur, the computation can be performed in the specified model.

The recursive process in lines 15-16 is continued, in parallel, until the number of queries that have to be answered on any particular subset of S is 1. At that time, the corresponding query is answered using Cole's parallel selection algorithm [19]. Similarly, if p is 1 then we use the sequential procedure for the multiple selection problem.

The processor assignment is as follows: we assign $p_1 = \mid S_1 \mid \times \frac{p}{n}$ of the processors to S_1, and $p_2 = \mid S_2 \mid \times \frac{p}{n}$ of the processors to S_2. It is easy to see that with this assignment,

$$\frac{|S_1|}{p_1} = \frac{|S_2|}{p_2} = \frac{|S|}{p} \qquad (3.2.4)$$

We shall let $T(n, m, p)$ stand for the worst-case running time of our parallel procedure. To get a recurrence describing $T(n, m, p)$, we can see that line 6 takes $(\frac{n}{p} + \log\log^* n)$ time by using Cole's algorithm [19]. Since $p \leq \frac{n}{\log n \log^* n}$, we have $O(\frac{n}{p} + \log\log^* n) = O(\frac{n}{p})$. Lines 8-9 take $O(\frac{n}{p} + \log n) = O(\frac{n}{p})$ time by simple prefix computation; similarly, the for loop in lines 10-11 runs in $O(\frac{n}{p} + \log n) = O(\frac{n}{p})$ time. Finally, the recusive calls in lines 15-16 are done in parallel, taking

$$\max\{T(\mid S_1 \mid, \lceil \tfrac{m}{2} \rceil - 1, p_1), T(\mid S_2 \mid, \lfloor \tfrac{m}{2} \rfloor, p_2)\}$$

Consequently, we can write

$$T(n, m, p) \leq c\frac{n}{p} + \max\{T(\mid S_1 \mid, \lceil \tfrac{m}{2} \rceil - 1, p_1), T(\mid S_2 \mid, \lfloor \tfrac{m}{2} \rfloor, p_2)\}$$

By induction, we can easily show: $T(n, m, p) \leq c\frac{n}{p} \log 2m$. □

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2.4 Discussion

Note that our parallel multiple selection procedure uses Cole's parallel single selection algorithm as subroutine. From the discussion in Section 3.2.2 we know that Cole's algorithm has not met the time lower bound for parallel single selection on an EREW PRAM. Actually, if a faster EREW PRAM parallel algorithm for the single selection is available, our parallel algorithm for multiple selection can be sped up by an $O(\log^* n)$ factor.

When $m = 1$ the complexity of our parallel algorithm matches that of Cole's parallel selection algorithm. However, when $m = n$ our algorithm (being cost optimal) is an $O(\log n)$ factor slower than the fastest sorting algorithm, e.g., Cole's parallel merge sort [20]. An interesting open question is whether or not we can use a different approach to obtain a faster parallel multiple selection algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 4

Tree Reconstruction

In this chapter, we present a parallel algorithm to reconstruct binary trees from their traversals. The materials here can also be found in [46]. Formally, the problem is defined as follows: For a binary tree $T = (V, E)$ where $V = \{1, 2, ..., n\}$, given its inorder traversal and either its preorder traversal or its postorder traversal, reconstruct the binary tree.

It is well known a binary tree can be reconstructed from its inorder traversal along with either its preorder traversal or its postorder traversal [42]. Recently, a sequential solution to this classical problem has been reported in [6]. Specifically, the algorithm in [6] takes $O(n)$ time and space. Parallel solutions to this problem can be found in [12, 63]. In particular, the algorithm in [63] runs in $O(\log n)$ time using $O(n)$ processors on the CREW PRAM; and the solution in [12] takes $O(\log \log n)$ time using $O(\frac{n}{\log \log n})$ processors on the CRCW PRAM.

Here, we present a new algorithm for this problem. Our algorithm requires $O(n)$ space. The main idea of our algorithm is to reduce the reconstruction process to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

parallel merging. With the best results for parallel merging, our algorithm can be implemented in $O(\log n)$ time using $O(\frac{n}{\log n})$ processors on the EREW PRAM, or in $O(\log\log n)$ time using $O(\frac{n}{\log\log n})$ processors on the CREW PRAM. Our algorithm thus improves the results in [12, 63].

Our parallel solution is of theoretical importance for the following reasons: (i) Recently, Berkman et al. defined a new class of problems called highly parallelizable problems [12] which contains problems that can be solved in $O(\log\log n)$ time using optimal number of processors. An extremely small number of problems are known to have optimal doubly logarithmic solutions and thus designing such algorithms was proposed as a challenge in [12]. (ii) Due to the research on lower bounds in [25], Berkman et al. pointed out [12] that doubly logarithmic time parallel algorithms usually need to run on an CRCW PRAM. A known exception is Kruskal's $O(\log\log n)$ time optimal parallel algorithm on a CREW PRAM. Our parallel solution thus finds one more example in the class of problems that can be solved in $O(\log\log n)$ using optimal number of processors on a CREW PRAM.

4.1 Preliminaries

Many methods can be used to generate traversals for a binary tree. Here, we are interested in one of them, known as the Euler tour technique [65]. This technique was proposed by Tarjan and Vishkin for designing efficient parallel algorithms on trees. Specifically, this technique reduces the computation of various kinds of information about the tree structure to the computation on a linked list [65]. To make our

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

presentation self-contained, the technique is described below:

The Euler tour technique: let T be a binary tree rooted at node r, Every node v of T is split into three copies v_1, v_2, v_3, all having the same node label as v. For simplicity, we assume that the nodes of the binary tree are labeled by integers $1, 2, ..., n$. For each of the resulting nodes, we define a *next* field as follows: If v has no left child then $v_1.next = v_2$. If v has no right child then $v_2.next = v_3$. If w is the left child of v then $v_1.next = w_1$, and $w_3.next = v_2$. If w is the right child of v then $v_2.next = w_1$, and $w_3.next = v_3$. What results is a list, called the *Euler path*, which starts at r_1, and ends at r_3 and which traverses each edge of T exactly once in each direction. In other words, let $\psi(T)$ denote the Euler path of a binary tree T. The Euler path of a binary tree with left subtree T_1 and right subtree T_2 can be expressed as $r_1\psi(T_1)r_2\psi(T_2)r_3$.

When no confusion is possible, we let Euler path also stand for the sequence of node labels induced by and Euler path.

Obviously, an Euler path of a tree contains three copies of each node label in the tree. An interesting property of the Euler path of a tree T is that keeping only the first copy of each label results in a preorder traversal of T; keeping only the second copy of each label gives an inorder traversal of T; keeping only the third copy of each label yields a postorder traversal of T [65].

For convenience, we define a *preorder-inorder path* to be a sequence of labels obtained by deleting the third copy of each label in an Euler path. Similarly, an *inorder-postorder path* is a sequence of labels obtained by deleting the first copy of each label in an Euler path. It is known that a binary can be reconstructed from its inorder traversal along with either its preorder traversal or postorder traversal. It

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

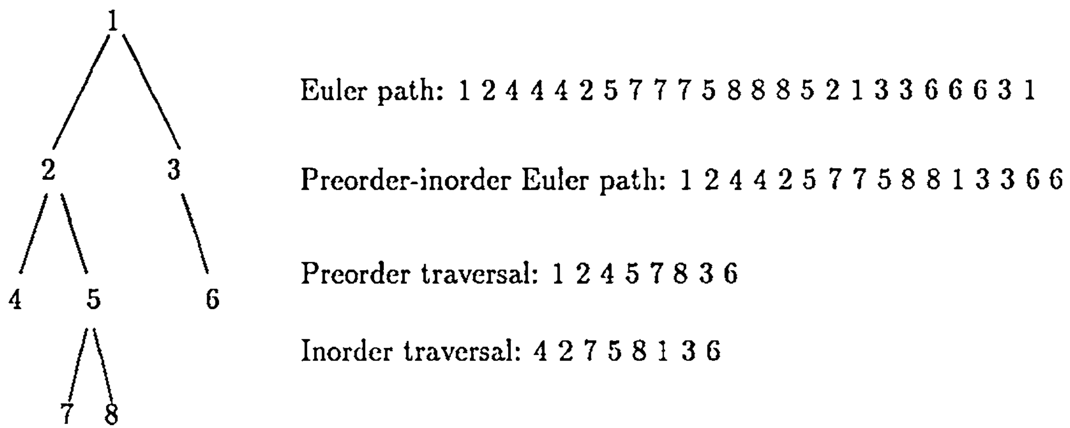


Figure 4.1: a binary tree, and its various (Euler) paths and traversals

follows that a binary tree is completely determined by its preorder-inorder path or its inorder-postorder path.

For example, Figure 4.1 features a binary tree along with the associated Euler path, preorder-inorder path, preorder traversal, and inorder traversal.

Lemma 4.1.1 *A sequence of labels $b_1, b_2, ..., b_{2n}$ represents a preorder-inorder path (respectively, the inorder-postorder path) of an n-node binary tree T if and only if the following conditions hold:*

(1) exactly two copies of each label occur in the sequence; and

(2) there exist no integers i, j, k, m with $1 \leq i < j < k < m \leq 2n$ such that $(b_i = b_k)$ and $(b_j = b_m)$.

Proof. We prove the statement for the case of a preorder-inorder path (the case of an inorder-postorder path follows by a mirror argument).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Let $\psi(T)$ denote the preorder-inorder path of a tree T. By definition, the preorder-inorder path of a tree rooted a node r with left subtree T_1 and right subtree T_2 can be expressed as $r_1\psi(T_1)r_2\psi(T_2)$. Thus, the "only if" part of the lemma if obvious.

The "if" part will be proved by induction on n. When $n = 1$ the lemma is obviously true. Assume the lemma is true for $n < l$. When $n = l$, let b_t be the second copy of b_1, i.e. $b_t = b_1$. By condition (2) and the induction hypothesis, $b_2, ..., b_{t-1}$ and $b_{t+1}, ..., b_{2n}$ can both be seen as preorder-inorder paths. Let T_1 and T_2 be the binary trees induced by $b_2, ..., b_{t-1}$ and $b_{t+1}, ..., b_{2n}$, respectively. The tree rooted at b_1 with left subtree T_1 and right tree T_2 is the tree determined by $b_1, b_2, ..., b_{2n}$. □

Corollary 4.1.1 *Let $c_1, c_2, ..., c_n$ and $d_1, d_2, ..., d_n$ be the preorder and the inorder traversals of a binary tree, respectively. There do not exist integers i_1, i_2, j_1, j_2, k_1, k_2, such that $(1 \leq i_1 < j_1 < k_1 \leq n)$, $(1 \leq k_2 < i_2 < j_2 \leq n)$, and $(c_{i_1} = d_{i_2}) \bigwedge (c_{j_1} = d_{j_2}) \bigwedge (c_{k_1} = d_{k_2})$.*

Proof. (by contradiction) Assume there exist integers i_1, i_2, j_1, j_2, k_1, k_2, such that $(1 \leq i_1 < j_1 < k_1 \leq n)$, $(1 \leq k_2 < i_2 < j_2 \leq n)$, and $(c_{i_1} = d_{i_2}) \bigwedge (c_{j_1} = d_{j_2}) \bigwedge (c_{k_1} = d_{k_2})$. Then, in the corresponding preorder-inorder path, d_{k_2} is the second copy of the c_{k_1}. So we have $k_1 < k_2$, which further implies $i_1 < j_1 < k_1 < k_2 < i_2 < j_2$. The preorder-inorder path must be of this form, $...c_{i_1}...c_{j_1}...c_{k_1}...d_{k_2}...d_{i_2}...d_{j_2}...$ contradicting the condition (2) Lemma 4.1.1. □

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2 Sequential Algorithms

In order to build a background for our parallel algorithm, in this section we present two sequential algorithms. The first algorithm generates the preorder-inorder path from the preorder and inorder traversals of a binary tree. The second algorithm uses the first algorithm as a subroutine to compute preorder-inorder path from the preorder and inorder traversals, and then reconstruct the binary tree using the information stored in the preorder-inorder path. The details of the algorithms are given below:

Procedure Traversal-path;

Input: sequence of labels, $c_1, c_2, ..., c_n$ and $d_1, d_2, ..., d_n$ as the preorder and inorder traversals of a binary tree;

Output: $b_1, b_2, ..., b_{2n}$, the preorder-inorder path of the tree, in which every label remembers the position of its duplicate;

$Stack \leftarrow \Phi$;

$j \leftarrow k \leftarrow 1$;

for $i \leftarrow 1$ **to** $2n$ **do**

if $d_k = top(Stack)$ **then**

$b_i \leftarrow d_k$;

$k \leftarrow k + 1$;

$\alpha \leftarrow$ **pop**$Stack$;

α and d_k remember each other's position in $b_1, b_2, ..., b_i$

else

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            b_i ← c_j;
            push c_j onto Stack;
            j ← j + 1;
        return(b_1, b_2, ..., b_2n);
end;
```

The correctness and the time complexity of this procedure are established by the following result.

Lemma 4.2.1 *Given the preorder and the inorder traversals of an n-node binary tree T, procedure* Traversals-path *computes in $O(n)$ time the preorder-inorder path of T, such that every label remembers the position of its duplicate in the preorder-inorder path.*

Proof. We prove the correctness of the procedure by induction on n. When $n = 1$ the algorithm is obviously correct. Assume that the algorithm is correct for $n < k$. Consider the case when $n = k$. Without loss of generality, assume $d_q = c_1$. By the definition of preorder and inorder traversals, c_1 is the root of T, and the left subtree of T has preorder traversal $c_2, ..., c_q$ and inorder traversal $d_1, ..., d_{q-1}$ while the right subtree of T has preorder traversal $c_{q+1}, ..., c_k$ and inorder traversal $d_{q+1}, ..., d_k$. By the induction hypothesis, consuming subsequences $c_2, ..., c_q$ and $d_1, ..., d_{q-1}$, the algorithm computes $b_1, b_2, ..., b_{2q-2}$ as the preorder-inorder path of the left subtree, with c_1 left on the top of the *Stack*. After matching d_q with top of the *Stack*, the algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

computes $b_{2q}, ..., b_k$ as the preorder-inorder path of the right subtree by consuming subsequences $c_{q+1}, ..., c_k$ and $d_{q+1}, ..., d_k$. From the statements above, we see that the output sequence $b_1, b_2, ..., b_{2n}$ satisfies both conditions of Lemma 4.1.1. Thus, $b_1, b_2, ..., b_{2n}$ represents the preorder-inorder path of some tree. Furthermore, deleting the second copies of the duplicate labels in $b_1, b_2, ..., b_{2n}$ results $c_1, c_2, ..., c_n$, while deleting the first copies gives $d_1, d_2, ..., d_n$. It follows that $b_1, b_2, ..., b_{2n}$ is the preorder-inorder path of T. According to the way the stack is used in the algorithm, we ensure that each label in the output sequence remembers the position of its duplicate. The algorithm runs in $O(n)$ time, since each iteration of the for loop has $O(1)$ time. □

Procedure Traversal-path-tree;

Input: sequence of labels, $c_1, c_2, ..., c_n$ and $d_1, d_2, ..., d_n$ as the preorder and inorder traversals of a binary tree;

Output: A binary tree with root node r, and the node set $S = \{d_1, d_2, ..., d_n\}$, the preorder-inorder path of the tree, in which every label remembers the position of its duplicate;

1. $S \leftarrow \{d_1, d_2, ..., d_n\}$;
2. Compute preorder-inorder path $b_1, b_2, ..., b_{2n}$ of the tree
   such that every label remembers the position of its duplicate;
3. $r \leftarrow$ the second copy of b_1;
4. **for** each label b_i $(2 \leq i \leq 2n)$ **do**
5. **if** (b_i is the second copy of its duplicate) **then**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6. **if** (b_{i-1} is the second copy of its duplicate label) **then**

7. $leftchild(b_i) \leftarrow b_{i-1}$;

8. **if** (b_{i+1} is the first copy of its duplicate label) **then**

9. $a \leftarrow$ the second copy of b_{i+1};

10. $rightchild(b_i) \leftarrow a$;

11. return($b_1, b_2, ..., b_{2n}$);

end;

Lemma 4.2.2 *Given a preorder-inorder path with* $2n$ *labels, procedure* Traversal-path-tree *correctly reconstructs the corresponding binary tree in* $O(n)$ *time.*

Proof. The correctness of the algorithm follows directly from the proof of Lemma 4.1.1. It is also easy to see the time complexity of the algorithm is $O(n)$. □

Combining Lemma 4.2.1 and Lemma 4.2.2, we have,

Theorem 4.2.1 *An n-node binary tree can be reconstructed from its preorder and inorder traversals in* $O(n)$ *time with* $O(n)$ *extra space.* □

4.3 A Highly Parallel Algorithm

We are now in a position to present our parallel solution to the problem of reconstructing an n-node binary tree from its preorder and inorder traversals. Our parallel algorithm is developed by parallelizing our sequential procedure, *Traversal-path-tree.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It is easy to see that, except for Line 2, procedure *Traversal-path-tree* can be implemented in $O(1)$ time using n processors on an EREW PRAM. The difficult part is to parallelize Line 2 of procedure *Traversal-path-tree.* Our idea here is to show that computing the preorder-inorder path from the preorder and inorder traversals can be reduced to parallel merging.

We now discuss how to compute the preorder-inorder path from a preorder traversal $c_1, c_2, ..., c_n$ and inorder traversal $d_1, d_2, ..., d_n$. For simplicity, we assume that $c_1, c_2, ..., c_n$ is $1, 2, ..., n$ (the case where $c_1, c_2, ..., c_n$ is a permutation of $1, 2, ..., n$ can be reduced to this case easily; we discuss this later). We compute the preorder-inorder path from $c_1, c_2, ..., c_n$ and $d_1, d_2, ..., d_n$ by merging according to some linear order as we are about explain. We will define such an order that both sequence $c_1, c_2, ..., c_n$ and $d_1, d_2, ..., d_n$ are already sorted.

Construct two sequences of triples: a sequence $(1, j_1, c_1), (1, j_2, c_2), ..., (1, j_n, c_n)$ such that $d_{j_i} = c_i, (i = 1, 2, ..., n)$ (i.e. j_i is the position of c_i in sequence $d_1, d_2, ..., d_n$); and a sequence $(2, 1, d_1), (2, 2, d_2), ..., (2, n, d_n)$.

Denote $\prod = \{(1, j_1, c_1), (1, j_2, c_2), ..., (1, j_n, c_n), (2, 1, d_1), (2, 2, d_2), ..., (2, n, c_n)\}$ Define a binary relation $\ll$ on $\prod$ as follows: for arbitrary triples (α, β, γ) and $(\alpha', \beta', \gamma')$ in $\prod$ we have:

<u>Rule 1.</u> $((\alpha = 1) \wedge (\alpha' = 1)) \rightarrow (((\alpha, \beta, \gamma) \ll (\alpha', \beta', \gamma')) \leftrightarrow (\gamma < \gamma'))$;

<u>Rule 2.</u> $((\alpha = 2) \wedge (\alpha' = 2)) \rightarrow (((\alpha, \beta, \gamma) \ll (\alpha', \beta', \gamma')) \leftrightarrow (\beta < \beta'))$;

<u>Rule 3.</u> $((\alpha = 1) \wedge (\alpha' = 2)) \rightarrow (((\alpha, \beta, \gamma) \ll (\alpha', \beta', \gamma')) \leftrightarrow ((\beta < \beta') \vee (\gamma \leq \gamma')))$

<u>Rule 4.</u> $((\alpha = 2) \wedge (\alpha' = 1)) \rightarrow (((\alpha, \beta, \gamma) \ll (\alpha', \beta', \gamma')) \leftrightarrow ((\beta < \beta') \wedge (\gamma < \gamma')))$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Theorem 4.3.1 *The binary relation $\ll$ defined above is a linear order on $\prod$.*

Proof. It is easy to see from Rules 1-4 that the binary relation $\ll$ is total on $\prod$. To prove that $\ll$ is a linear order we need to show that it is transitive. We shall present our arguments in the form of a case-by-case analysis. Let (α, β, γ), $(\alpha', \beta', \gamma')$ and $(\alpha", \beta", \gamma")$ be arbitrary triples in $\prod$ satisfying:

$(\alpha, \beta, \gamma) \ll (\alpha', \beta', \gamma')$ and $(\alpha', \beta', \gamma') \ll (\alpha", \beta", \gamma")$

We need $(\alpha, \beta, \gamma) \ll (\alpha", \beta", \gamma")$.

case 1. $\alpha = \alpha"$

subcase 1.1 $\alpha = \alpha' = \alpha" = 1$.

By Rule 1 and the assumption, we have $\gamma < \gamma'$ and $\gamma' < \gamma"$ and therefore $\gamma < \gamma"$. The conclusion follows by Rule 1.

subcase 1.2 $\alpha = \alpha" = 1$ and $\alpha' = 2$

By Rule 3,

(a) $(\gamma \leq \gamma') \bigvee (\beta < \beta')$

By Rule 4,

(b) $(\beta' < \beta") \bigwedge (\gamma' < \gamma")$

If $\gamma < \gamma"$ then conclusion follows immediately from Rule 1. Therefore, we assume

(c) $\gamma" < \gamma$

Note that (a), (b) and (c) combined imply that

$(\gamma' < \gamma" < \gamma) \bigwedge (\beta < \beta' < \beta")$ which is contradicting Corollary 4.1.1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

subcase 1.3 $\alpha = \alpha' = \alpha" = 2$

By Rule 2, we have $\beta < \beta'$ and $\beta' < \beta"$ and therefore $\beta < \beta"$. The conclusion follows by Rule 2.

subcase 1.4 $\alpha = 2$, $\alpha' = 1$ and $\alpha" = 2$

By Rule 4,

(d) $(\gamma < \gamma') \bigwedge (\beta < \beta')$

By Rule 3,

(e) $(\beta' < \beta") \bigvee (\gamma' \leq \gamma")$

If $(\beta < \beta")$ then conclusion follows instantly from Rule 2. We may assume, therefore,

(f) $(\beta" < \beta)$

By now, (d), (e) and (f) combined imply that

$(\gamma < \gamma' < \gamma") \bigwedge (\beta" < \beta < \beta')$ which contradicts Corollary 4.1.1.

case 2. $\alpha \neq \alpha"$

subcase 2.1 $\alpha = \alpha' = 1$ and $\alpha" = 2$.

By Rule 1,

(g) $(\gamma < \gamma')$

By Rule 3,

(h) $(\beta' < \beta") \bigwedge (\gamma' \leq \gamma")$

Note that if $(\beta < \beta") \bigvee (\gamma' \leq \gamma")$ then by Rule 3, we have $(\alpha, \beta, \gamma) \leq (\alpha", \beta", \gamma")$. Therefore, we may assume that

(i) $(\beta" < \beta) \bigwedge (\gamma" \leq \gamma)$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

But now, (g), (h) and (i) combined imply

$(\gamma" < \gamma < \gamma') \bigwedge (\beta' < \beta" < \beta)$ which contradicts Corollary 4.1.1.

subcase 2.2 $\alpha = 1$ and $\alpha' = \alpha" = 2$.

By Rule 3,

(j) $(\beta < \beta') \bigvee (\gamma \leq \gamma")$

By Rule 2,

(k) $(\beta' < \beta")$

Note that if $(\beta < \beta") \bigvee (\gamma \leq \gamma")$ then the conclusion follows by Rule 3,

Therefore, we may assume that

(l) $(\beta" < \beta) \bigwedge (\gamma" < \gamma)$

But now, (j), (k) and (l) combined imply

$(\gamma" < \gamma < \gamma') \bigwedge (\beta' < \beta" < \beta)$ which contradicts Corollary 4.1.1.

subcase 2.3 $\alpha = 2$ and $\alpha' = \alpha" = 1$.

By Rule 4,

(m) $(\beta < \beta') \bigwedge (\gamma < \gamma')$

By Rule 1,

(n) $(\gamma' < \gamma")$

Note that if $(\beta < \beta") \bigwedge (\gamma \leq \gamma")$ then the conclusion follows by Rule 4,

Therefore, we may assume that

(o) $(\beta" < \beta) \bigwedge (\gamma" \leq \gamma)$

But now, (m), (n) and (o) combined imply

$(\gamma < \gamma' < \gamma") \bigwedge (\beta" < \beta < \beta")$ which contradicts Corollary 4.1.1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

subcase 2.4 $\alpha = \alpha' = 2$ and $\alpha" = 1$.

By Rule 2,

(p) $(\beta < \beta')$

By Rule 4,

(q) $(\beta' < \beta") \wedge (\gamma' < \gamma")$

Note that if $(\gamma < \gamma")$ then (p) and (q) combined give

$(\alpha, \beta, \gamma) \ll (\alpha", \beta", \gamma")$ by Rule 4. Thus, we may assume

$(\gamma" < \gamma)$. But now (p) and (q) imply

$(\gamma' < \gamma" < \gamma) \wedge (\beta < \beta' < \beta)$ which contradicts Corollary 4.1.1.

□

By Rules 1-2, we can see that according to linear order $\ll$ both sequence $(1, j_1, c_1)$, $(1, j_2, c_2)$, ..., $(1, j_n, c_n)$ and $(2, 1, d_1)$, $(2, 2, d_2)$, ..., $(2, n, d_n)$ are already sorted. Merging these two sequences according to $\ll$ we obtain a sequence of triples: $(\alpha_1, \beta_1, \gamma_1)$, $(\alpha_2, \beta_2, \gamma_2)$, ..., $(\alpha_n, \beta_n, \gamma_{2n})$. We claim that γ_1 , γ_2, ..., γ_{2n} is the preorder-inorder path determined by the traversals. The correctness of the claim relies on the following facts:

(a) Exactly two copies of each label appear in γ_1, γ_2, ..., γ_{2n} satisfying condition (1) of Lemma 4.1.1;

(b) There do not exist integers $1 \leq i < j < k < l \leq 2n$ such that $\gamma_i = \gamma_k$ and $\gamma_j = \gamma_l$ satisfying condition (2) of Lemma 4.1.1;

(c) Deleting the second copies of the duplicate labels in $\gamma_1, \gamma_2, ..., \gamma_{2n}$ results in $c_1, c_2, ..., c_n$, and deleting the first copies of the duplicate labels in $\gamma_1, \gamma_2, ..., \gamma_{2n}$ gives $d_1, d_2, ..., d_n$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Fact (a) follows directly from Rule 3 and the construction of the triples in $\prod$. By the definition of $\ll$ both sequences $(1, j_1, c_1)$, $(1, j_2, c_2)$, ..., $(1, j_n, c_n)$ and $(2, 1, d_1)$, $(2, 2, d_2)$, ..., $(2, n, d_n)$ are already sorted, so fact (c) is also true. The proof of fact (b) is given below:

Proof of (b). (by contradiction)

(1)$(\gamma_i = \gamma_k) \wedge (\gamma_j = \gamma_m)$ where $1 \leq i < j < k < m \leq 2n$	[Assumption]
(2)$(\alpha_i, \beta_i, \gamma_i) \ll (\alpha_j, \beta_j, \gamma_j) \ll (\alpha_k, \beta_k, \gamma_k) \ll (\alpha_m, \beta_m, \gamma_m)$	[by $1 \leq i < j < k < m \leq 2n$ in (1)]
(3)$(\alpha_i = 1) \wedge (\alpha_j = 1) \wedge (\alpha_k = 2) \wedge (\alpha_m = 2)$	[by (1), (2), Rule 3 and the construction of the triples]
(4)$(\beta_i = \beta_k) \wedge (\beta_j = \beta_m)$	[by (1) and the construction of the triples]
(5)$(\beta_j < \beta_k) \vee (\gamma_j \leq \gamma_k)$	[by $(\alpha_j = 1) \wedge (\alpha_k = 2)$ in (3), $(\alpha_j, \beta_j, \gamma_j) \ll (\alpha_k, \beta_k, \gamma_k)$ in (2) and Rule 3]
(6)$(\beta_m < \beta_k) \vee (\gamma_m \leq \gamma_k)$	[replace β_j, γ_j in (5) with β_m, γ_m respectively, by (1) and (4)]
(7)$(\beta_k < \beta_m)$	[by $(\alpha_k, \beta_k, \gamma_k) \ll (\alpha_m, \beta_m, \gamma_m)$ in (2), $(\alpha_k = 2) \wedge (\alpha_m = 2)$ in (3) and Rule 2]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(8)$(\gamma_m \leq \gamma_k)$ [by (6) and (7)]

(9)$(\gamma_i < \gamma_j)$ [by $(\alpha_i, \beta_i, \gamma_i) \ll (\alpha_j, \beta_j, \gamma_j)$ in (2), $(\alpha_i = 1) \wedge (\alpha_j = 1)$ in (3) and Rule 1]

(10)$(\gamma_k < \gamma_m)$ [by (1) and (9)]

(11)Contradiction [(8) and (10)]

□

Up to this point, we have successfully reduced computing the preorder-inorder path to parallel merging. We now discuss the complexity of this reduction. First, we consider the complexity to construct $\prod$. For this purpose, let us see how to construct from the given traversals, sequences $(1, j_1, c_1)$, $(1, j_2, c_2)$, ..., $(1, j_n, c_n)$ and $(2, 1, d_1)$, $(2, 2, d_2)$, ..., $(2, n, d_n)$ such that $c_i = d_{j_i}$ $(i = 1, 2, ..., n)$. We note that this can be done easily with an auxiliary array $A[1..n]$. Since $c_1, c_2, ..., c_n$ is $1, 2, ..., n$, and $d_1, d_2, ..., d_n$ is a permutation of $1, 2, ..., n$, we can compute an $A[1..n]$ as follow: $A[d_i] = i$ $(1, 2, ..., n)$ in $O(1)$ time on an EREW PRAM with n processors. To determine the subscript j_1 satisfying $c_i = d_{j_i}$ $(1, 2, ..., n)$, we simply take $j_i = A[c_i]$ $(1, 2, ..., n)$. This again can be computed in $O(1)$ time on an EREW PRAM with n processors. Consequently,

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

can be constructed in $O(1)$ time using n processors on an EREW PRAM.

Next, we consider the complexity to merge $(1, j_1, c_1)$, $(1, j_2, c_2)$, ..., $(1, j_n, c_n)$ and $(2, 1, d_1)$, $(2, 2, d_2)$, ..., $(2, n, d_n)$ according $\ll$. Optimal parallel algorithms are proposed in [7, 33, 43]. With their results, we have,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Theorem 4.3.2 *For a binary tree $T = (V, E)$ where $V = \{1, 2, ..., n\}$, given its preorder and inorder traversals, the binary tree can be reconstructed using $O(n)$ extra space, in $O(\log n)$ time using $O(\frac{n}{\log n})$ processors on the EREW PRAM, or in $O(\log\log n)$ time using $O(\frac{n}{\log\log n})$ processors on the CREW PRAM.* □

4.4 Discussion

We have shown how to reconstruct a binary tree from its inorder traversal along with either its preorder traversal or its postorder traversal by reducing the problem to parallel merging. With the best known results for parallel merging, our reconstruction algorithm can be implemented in $O(\log n)$ time using $O(\frac{n}{\log n})$ processors on the EREW PRAM, or in $O(\log\log n)$ time using $O(\frac{n}{\log\log n})$ processors on the CREW PRAM. We have thus found one more example in the class of problems that can be solved in doubly logarithmic time using optimal number of processors on the CREW PRAM.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 5

Concluding Remarks

We have studied several basic problems in the design and analysis of non-numerical parallel algorithms. As we explained in Chapter 1, the purposes of studying this type of problems are to obtain basic building blocks which will be useful in solving complex problems and to develop fundamental algorithmic techniques.

In Chapter 2 we studied priority queues. Priority queues have received a great deal of attention in literature because of its many applications [45, 18, 36, 32, 30, 14, 38, 67, 31]. Our research in this area started at looking for meldable double-ended priority queues [48, 46]. The recent enthusiasm in parallel implementations of priority queue operations [57, 56, 59, 76, 40, 28, 26, 54] also stimulated us to carry on this research in the parallel setting [49]. As results of our research, we have found a meldable double-ended priority queue; we proposed a parallel melding algorithm for priority queues implemented by heaps or min-max-pair heaps; and more importantly, we have presented a technique which can be used to develop optimal parallel initialization

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

algorithms for a class of priority queues.

In Chapter 3 we studied two problems, multiple search and multiple selection. The purpose of this chapter is to explore the relationships among four of the most fundamental problems in algorithm design, i.e selection, searching, merging and sorting. As it turns out, our parallel solutions for the two problems can be used as subroutines in algorithms for other problems. For example, our optimal parallel solution for the multiple search problem can be used in Hagerup and Rub's parallel merging algorithm [33] to obtain the optimal implementation of their algorithm on the EREW PRAM.

In Chapter 4 we studied the classical problem of reconstructing a binary tree from its traversals [42]. We presented a highly parallel algorithm for the problem. This research was motivated by a challenge proposed in [12] to design doubly logarithmic time optimal parallel algorithms (highly parallel), since a remarkably small number of such algorithms are known. Another highly parallel algorithm on the CRCW PRAM was proposed by Berckman et al. for the same problem. Due to the research on lower time bounds in [25], Berckman et al. pointed out in [12] that doubly logarithmic time parallel algorithms usually need to run on an CRCW PRAM. A known exception is Kruskal's doubly logarithmic time parallel merging algorithm which run on an CREW PRAM. Compared to the algorithm in [12], however, our algorithm can be implemented on the CREW PRAM and thus gives one more example in the class of highly parallel problems that run on the CREW PRAM.

Although we have achieved some progress the problems we studied, many questions remain open. First of all, we would like to know whether or not our technique to initialize priority queues can be applied to other data structures. We believe our

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

parallel multiple selection algorithm, though efficient, is not the fastest possible. It could be nice to find a faster parallel algorithm (keeping the same cost) by a different approach. Finally, it will be interesting to know whether our highly parallel algorithm in Chapter 4 can be improved.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[2] M. Ajtai, J. Komlos, W. L. Steiger, and E. Szemeredi. An o(n log n) sorting network. In *Proceedings of the Annual ACM Symposium on the Theory of Computing*, pages 1–9, 1983.

[3] M. Ajtai, J. Komlos, W. L. Steiger, and E. Szemeredi. Deterministic selection in o(log log n) parallel time. In *Proceedings of the Annual ACM Symposium on the Theory of Computing*, pages 188–195, 1986.

[4] S. G. Akl. *The Design and Analysis of Parallel Algorithms*. Pretice Hall, Englewood Cliffs, New Jersey 07632, 1989.

[5] S. G. Akl and J. Meijer. Parallel binary search. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):247–250, April 1990.

[6] A. Anderson and S. Carlsson. Construction of a tree from its traversals in optimal time and space. *Information Processing Letters*, 34:21–25, 1990.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

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

[8] M. D. Atkinson, J. R. Sack, N. Santoro, and T. Strothotte. Min-max heaps and generalized priority queues. *Communications of ACM*, 29:996–100, 1986.

[9] Y. Azar and U. Vishkin. Tight comparison bounds on the complexity of parallel sorting. *SIAM Journal on Computing*, 16(3), June 1987.

[10] S. Baase. *Computer Algorithms-An Introduction to Design and Analysis.* Addison-Wesley, 198.

[11] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference 32*, pages 307–314, 1968.

[12] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin. Highly parallelizable problems. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 11–20, 1989.

[13] O. Berkman and U. Vishkin. Recursive *-tree parallel data-structure. In *Proceedings of the Annual IEEE Symposium on Foundation of Computer Science*, pages 196–202, 1989.

[14] Brown and Randy. Calendar queues: A fast o(1) priority queue implementation for the simulation event set problem. *Communications of ACM*, 31(10):1220–1227, 1988.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[15] S. Carlsson. the deap-a double ended heap to implement double ended priority queues. *Information Processing Letters*, 26:33–36, 1987.

[16] S. Carlsson, J. Chen, and T. Strothotte. A note on the construction of the data structure 'deap'. *Information Processing Letters*, 31:315–317, 1989.

[17] L. Chen and Y. Yesha. Parallel recognition of the consecutive ones property with applications. *Journal of Algorithms*, 12(3):375–393, 1991.

[18] E. G. Coffman and M. Hofri. On scanning disks and the analyst of their steady state behavior. In *Proceedings of the Conference of Measurement, Modeling and Evaluating Computer Systems*, 1982.

[19] R. Cole. An optimally efficient selection algorithm. *Information Processing Letters*, 26(6):295–299, 1988.

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

[21] R. Cole and U. Vishkin. Approximate parallel scheduling. part 1: the basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM Journal on Computing*, pages 128–142, 1988.

[22] R. Cole and C. K. Yap. A parallel median algorithm. *Information Processing Letters*, 20(3):137–139, April 1985.

[23] S. A. Cook. Towards a complexity theory of synchronous parallel computation. *L'Enseignment Mathematique*, 30, 1980.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[24] S. A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[25] S. A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, pages 87–98, 1986.

[26] N. Deo and S. Prasad. Parallel heap. In *Proceedings of the IEEE International Conference on Parallel Processing*, 1990.

[27] J. R. Driscoll, R. Shrairman H. N. Gabow, and R. E. Tarjan. Relaxed heaps: an alternative to fibonacci heaps with applications to parallel computation. *Communication of ACM*, 31(11):1343–1354, 1988.

[28] Z. Fan and K. H. Cheng. A simultaneous access priority queue. In *Proceedings of the IEEE International Conference on Parallel Processing*, pages 195–198, 1989.

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

[30] W. R. Franta and K. Maly. An efficient data structure for the simulation event-set. *Communications of ACM*, 20(8):585–606, 1977.

[31] H. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[32] G. H. Gonnet. Heaps applied to event-driven mechanisms. *Communications of ACM*, 19:417–418, 1976.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[33] T. Hagerup and C. Rub. Optimal merging and sorting on the erew pram. *Information Processing Letters*, 33:181–185, December 1989.

[34] A. Hasham and J. R. Sack. Bounds for min-max heaps. *BIT*, 27:315–323, 1987.

[35] X. He. Efficient parallel algorithms for series parallel graphs. *Journal of Algorithms*, 12(3):409–430, 1991.

[36] M. Hofri. Disk scheduling: Fcfs vs sstf revisited. *Communications of ACM*, 23, 1980.

[37] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal on Computing*, 1:31–39, March 1972.

[38] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1984.

[39] Editor J. L. C. Sanz. *Opportunities and constraints of Parallel Computing.* Springer-Verlag, 1988.

[40] D. W. Jones. Concurrent operations on priority queues. *Communications of the ACM*, 32(1):132–137, Jan. 1989.

[41] R. M. Karp and V. Ramachandran. A survey of parallel algorithm for shared memory machines. *Rep. No. UCB/CSD 88/408*, Computer Science Division, University of California, Berkeley, CA 94720, March 1988.

[42] D. E. Knuth. *The Art of Computer Programming, Vol. 1, Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1973.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[43] C. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Transactions on Computers*, pages 942–946, October 1983.

[44] G. L. Miller and J. Reif. Parallel tree contraction and its application. In *Proceedings of the Annual IEEE Symposium on Foundation of Computer Science*, pages 140–148, 1985.

[45] O. Nevalaineen and J. Teuhola. Priority queue administration by sublist index. *The Computer Journal*, 22:220–224, 1977.

[46] S. Olariu, C. M. Overstreet, and Z. Wen. A mergeable double-ended priority queue. *The Computer Journal-A Special Issue on Data Structures*, Oct. 1991.

[47] S. Olariu, C. M. Overstreet, and Z. Wen. An optimal parallel algorithm to reconstruct a binary tree from its traversals. In *Proceedings of the International Conference on Computing and Information (Carleton University, Ottawa, Canada*, May 27-29 1991. Submitted to Journal of Parallel and Distributed Computing.

[48] S. Olariu and Z. Wen. The min-max-pair heaps and its variations. Tech. Rep. TR-89-33, Department of Computer Science, Old Dominion University, Sep. 1989.

[49] S. Olariu and Z. Wen. Fast parallel algorithms on heaps. Tech. Rep. TR-90-12, Department of Computer Science, Old Dominion University, Feb. 1990.

[50] S. Olariu and Z. Wen. An optimal parallel construction scheme for heap-like structures. In *Proceedings of the 28th Annual Allerton Conference in Control, Communication, and Computing(University of Illinois, Urbana-Champaigh)*, October 1990.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[51] S. Olariu and Z. Wen. Optimal parallel initilization algorithms for a class of priority queues. *IEEE Transactions on Parallel and Distributed Systems*, (in press).

[52] S. Olariu and Z. Wen. An efficient parallel algorithm for multi-selection. *Parallel Computing*, (to appear).

[53] I. Parberry. *Parallel Complexity Theory*. John Wiley and Sons, Inc. New York, Toronto, 1987.

[54] M. C. Pinotti and G. Pucci. Parallel priority queue. In *Proceedings of the 28th Annual Allerton Conference in Control, Communication, and Computing(University of Illinois, Urbana-Champaigh)*, 1990.

[55] T. Przytycka and D. G. Corneil. Parallel algorithms of parity graphs. *Journal of Algorithms*, 12(1):96–109, 1991.

[56] M. J. Quinn. *Designing efficient algorithms for parallel computers*. New York, McGraw-Hill, 1987.

[57] M. J. Quinn and N. Deo. Parallel graph algorithms. *Computing Survey*, 16(3):319–348, Sep. 1984.

[58] A. G. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42:307–326, 1991.

[59] V. N. Rao and V. Kumar. Concurrent access of priority queues. *IEEE Transactions on Computer*, 37(12):1657–1665, Dec. 1988.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[60] D. S. Richards and J. S. Salowe. Stacks, queues, and deques with order-statistic operations. In *Proceedings of the 28th Annual Allerton Conference in Control, Communication, and Computing(University of Illinois, Urbana-Champaigh)*, 1990.

[61] J. R. Sack and T. Strothotte. An algorithms for merging heaps. *Acta Informatica*, 22:171–186, 1985.

[62] D. D. Sleator and R. Tarjan. Self adjusting heaps. *SIAM Journal on Computing*, 1986.

[63] F. Springsteel and I. Stojmenovic. Parallel general prefix computations with geometric, algebraic and other applications. In *Proceedings of International Conference on Fundamentals of Computation Theory*, pages 424–433, 1989.

[64] R. Tamassia and J. S. Vitter. Parallel transitivity closure and point location in planar structures. *SIAM Journal on Computing*, 20(4):708–726, 1991.

[65] R. Tarjan and U. Vishkin. Approximate and exact parallel scheduling with application to list, tree and graph. In *Proceedings of the Annual ACM Symposium on the Theory of Computing*, pages 487–491, 1984.

[66] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.

[67] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAm, Philadelphia, Pa., 1983.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[68] P. C. Treleaven. Parallel architecture overview. *Parallel Computing*, pages 59–70, 1988.

[69] J. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1):100-125, 1991.

[70] L. G. Valiant. Parallelism in comparison problems. *SIAM Journal on Computing*, 4(3):348-355, 1975.

[71] U. Vishkin. Synchronous parallel computation. a Survey, TR71, Department of Computer Science, Courant Institute, NYU, 1983.

[72] U. Vishkin. Deterministic sampling-a new technique for fast pattern matching. *SIAM Journal on Computing*, 20(1):22–40, 1991.

[73] J. A. Viullemin. A data structure for manipulating priority queues. *Communications of ACM*, 21:309–314, 1978.

[74] Z. Wen. Parallel multiple search. *Information Processing Letters*, Feb. 1991.

[75] J. W. J. Williams. Algorithm 232. *Communications of ACM*, 7:347–348, 1964.

[76] Y. B. Yoo. Parallel processing for some network optimization problems. Ph.D dissertation, Computer Science Dept. Washington State University, Pullman, WA, 1983.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Autobiographical Statement

Zhaofang Wen

I was born on September 2nd, 1961 in Canton Province, China. I have two previous degree as follows:

- B.S. Computer Science, ZhongShan University, Canton, China., July 1982.
- M.S. Computer Science, ZhongShan University, Canton, China., July 1985.

The following is a list of my published articles:

- "A mergeable double-ended priority queue," (with *S. Olariu* and *C. M. Overstreet*) *The Computer Journal-A Special Issue on Data Structures*, October, 1991.
- "Merging heaps in parallel," (with *S. Olariu*) *International Journal of Computer Mathematics*, Feb. 1991.
- "Optimal parallel initialization algorithms for a class of priority queues," (with *S. Olariu*) *IEEE Transactions on Parallel and Distributed Systems* (in press).
- "Parallel multiple search," *Information Processing Letters*, 37(4), Feb. 1991.
- "An efficient parallel algorithm for multi-selection," (with *S. Olariu*) *Parallel Computing*, (in press).
- "A faster optimal parallel algorithm for the measure problem," (with *S. Olariu* and *W. Zhang*) *Parallel Computing*, (in press).
- "An optimal parallel algorithm to reconstruct a binary tree from its traversals," (*S. Olariu* and *C. M. Overstreet*) in *Proceedings of the International Conferences on Computing and Information*, May 27-29, 1991, Ottawa, Canada.
- "Efficient parallel algorithms for some integer problems," (with *W. Zhang*) in *Proceedings of the Annual ACM Computer Science Conference*, March 4-7, 1991, San Antonio, TX.
- "An optimal parallel construction scheme for heap-like structures," (with *S. Olariu*) in *Proceedings of the 28th Annual Allerton Conference on Control, Communication and Computing*, University of Illinois, Urbana-Champaign, IL, Oct. 3-5, 1990.
- "Optimal parallel encoding and decoding algorithms for trees," (with *S. Olariu*, *J. Schwing* and *J. Zhang*) in *Proceedings of the Annual ACM Computer Science Conference*, March 4-7, 1991, San Antonio, TX.

Upon receiving my master degree from ZhongShan University, China, I was hired as an Instructor of Computer Science at the same university, where I served for two years. In 1987, I enrolled as a graduate student at Oklahoma State University. In 1988, I joint the Ph.D program of computer science at Old Dominion University. My research interests include *Parallel Processing*, *Software Engineering*, and *Data Structures*. Funding for the research done in this thesis was through a research assistantship from NASA and Navy, and a Special Doctoral Fellowship from the Department of Computer Science, Old Dominion University.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.