EFFICIENT LINKED LIST RANKING ALGORITHMS AND PARENTHESSES MATCHING AS A NEW STRATEGY FOR PARALLEL ALGORITHM DESIGN

DISSERTATION

Presented to the Graduate Council of the University of North Texas in Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

By

Ranette Hudson Halverson, B.S., M.S.

Denton, Texas

December, 1993
EFFICIENT LINKED LIST RANKING ALGORITHMS AND
PARENTHESES MATCHING AS A NEW STRATEGY
FOR PARALLEL ALGORITHM DESIGN

DISSERTATION

Presented to the Graduate Council of the
University of North Texas in Partial
Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

By

Ranette Hudson Halverson, B.S., M.S.
Denton, Texas
December, 1993

The goal of a parallel algorithm is to solve a single problem using multiple processors working together and to do so in an efficient manner. In this regard, there is a need to categorize strategies in order to solve broad classes of problems with similar structures and requirements. In this dissertation, two parallel algorithm design strategies are considered: linked list ranking and parentheses matching.

Deterministic and randomized linked list ranking algorithms are presented for the exclusive-read exclusive-write (EREW) parallel random access machine (PRAM) model. They are based on a technique unlike the traditional reduction method. The randomized algorithm is work-optimal, and, although the deterministic is not, the technique is quite simple in comparison to previously proposed algorithms and has the advantage of small constant factors in terms of time and space requirements.

Another contribution of this dissertation is the establishment of parentheses matching as a general strategy for designing efficient parallel algorithms. This is accomplished through the development of a class of tree related algorithms for the PRAM model which are solved using parentheses matching as a major component. The problems solved include the heights and extreme values of the nodes of a tree, the least common ancestor problem, inorder traversal of a tree, tree contraction, and balancing binary trees. Finally, a hypercube implementation for parentheses matching and its application to the nearest enclosing parentheses problem are presented.
ACKNOWLEDGEMENTS

I wish to thank my advisor, Dr. Sajal K. Das, for the benefit of his expertise and wisdom, which he so generously shared with me. I will be forever grateful for the experience I obtained while working with him.

I want to thank the members of my Ph.D. committee, Dr. Roy T. Jacob, Dr. C. Q. Yang, and Dr. Neal Brand for their time and effort on my behalf. I also thank the faculty of the UNT Department of Computer Science for their teaching, which helped prepare me for this dissertation. Thanks also to my friend and colleague, Richard Simpson, who never failed to encourage me when times were rough.

Finally, I wish to thank my family. Without their continual love and support this dissertation would not have been possible. I am grateful to my parents, Rita and Randall Hudson, who instilled in me the value of an education and a job well-done; to my husband, Patrick, who demonstrated the patience and support that few people possess; and to my sons, Stuart and Andrew, who make the future important.

“I can do all things through Christ Jesus who strengthens me.” Phillipians 4:13.
# TABLE OF CONTENTS

1 INTRODUCTION
   1.1 Motivation .................................................. 1
   1.2 Performance Metrics ....................................... 2
   1.3 Models of Computation ...................................... 3
      1.3.1 PRAM Model ........................................... 4
      1.3.2 Hypercube Model ..................................... 5
   1.4 Parallel Algorithm Design Strategies ................. 6
   1.5 Research Overview ......................................... 10

2 EXISTING WORK ON LINKED LIST RANKING ............... 12
   2.1 Introduction ............................................... 12
   2.2 Problem Definition ....................................... 12
   2.3 Evolution of Parallel List Ranking Algorithms ........ 14
   2.4 Parallel List Ranking Approaches ...................... 17
      2.4.1 Pointer Jumping (Doubling) Approach ............. 18
      2.4.2 Reduce–Rank–Expand Technique ..................... 21
   2.5 Conclusion .................................................. 27

3 LINKED LIST RANKING ALGORITHMS ON EREW PRAM ....... 29
   3.1 Introduction ............................................... 29
   3.2 APRAM Model and Performance Metrics .................. 29
   3.3 APRAM List Ranking Algorithm ........................... 30
   3.4 EREW List Ranking Algorithms ............................ 32
3.4.1 Randomized Algorithm ........................................... 33
3.4.2 Deterministic Algorithm ........................................ 36
3.5 Advantages of New Algorithms .................................... 40
3.6 Conclusion .......................................................... 42

4 PARENTHESES MATCHING AS A DESIGN STRATEGY ............ 43
4.1 Introduction .......................................................... 43
4.2 Existing Work with Parentheses Matching ........................ 45
4.3 Existing Work in the Application of Parentheses Matching .... 48
  4.3.1 Breadth-first Traversal of a Tree ............................... 48
  4.3.2 Sorting Integers in a Restricted Class ........................ 49
  4.3.3 Coloring of an Interval Graph .................................. 49
4.4 Application of Parentheses Matching to Trees .................. 50
  4.4.1 Representations and Traversals of Trees ....................... 50
  4.4.2 Tree Representations Related to Parentheses ................. 53
  4.4.3 Euler Tour Technique vs. Parentheses Matching .............. 54
4.5 Parentheses Matching Applied to Basic Tree Operations ....... 54
  4.5.1 Tree Traversals via PPM ........................................ 55
  4.5.2 Converting Parentheses to Trees ............................... 55
  4.5.3 Constructing a Tree from Traversals ........................... 58
  4.5.4 Tree Contraction by Parentheses Matching .................... 59
4.6 Conclusion .......................................................... 62

5 APPLICATION OF PPM TO TREE RELATED PROBLEMS ............. 64
5.1 Introduction .......................................................... 64
5.2 Bottom-up Tree Computations ...................................... 64
  5.2.1 Heights of all Nodes of a Tree ................................. 66
6.8 Conclusion ...................................................... 101

7 CONCLUSION .................................................. 102

BIBLIOGRAPHY .................................................. 105
LIST OF TABLES

2.1 Applications of Linked List Ranking .............................................. 15
2.2 Comparison of List Ranking Algorithms ........................................... 17
3.1 Demonstration of the EREW Randomized Algorithm .......................... 35
3.2 Demonstration of the EREW Deterministic Algorithm ....................... 38
3.3 Space Requirements of List Ranking Algorithms .............................. 41
4.1 Tree Contraction by Parentheses Matching .................................... 62
5.1 Demonstration of Heights of all Nodes ......................................... 67
5.2 Demonstration of Algorithm NEPA .............................................. 70
5.3 Tree Related Algorithms using PPM ............................................ 83
LIST OF FIGURES

1.1 Shared Memory PRAM Model .......................... 5
1.2 Hypercube Model for n=2,4,8 .......................... 6
2.1 Linked List in Array Form .......................... 13
2.2 Linked List in Logical Form .......................... 13
2.3 Strategy for Work-Optimal List Ranking ................. 18
2.4 Demonstration of Pointer Jumping ....................... 19
2.5 Pointer Jumping Method for List Ranking ............... 21
2.6 Deleting an Element from a Linked List .................. 24
2.7 List Ranking Example ................................ 24
2.8 Linked List Expansion ............................... 26
4.1 Parentheses Matching Strategy ........................ 44
4.2 Encoding of Parentheses ............................. 47
4.3 Sample Tree ........................................ 52
4.4 Parentheses Representations of a Tree .................. 53
4.5 Demonstration of Algorithm INORDER .................... 56
4.6 Demonstration of Algorithm PARENTHESES-TO-TREE .......... 58
5.1 Demonstration of Range Maxima Problem ................. 66
5.2 Complete and Perfect Trees of Twelve Nodes ............ 73
5.3 Balancing a Binary Search Tree ........................ 79
5.4 Embedded Parentheses for Balanced Binary Tree .......... 79
5.5 Generation of Parentheses String ....................... 83
6.1 Divide-and-Conquer Parentheses Matching Technique .... 91
6.2 Algorithm 4-CUBE Distribution of Data ................................. 94
6.3 Marking of Subsequences of Parentheses ................................. 96
CHAPTER 1

INTRODUCTION

1.1 Motivation

Parallel computing has become one of the major areas of research in the computer science field with numerous journals and conferences dedicated to its development. Multiprocessor computers (parallel computers) are becoming more prevalent, with several manufacturers and models readily available, several at quite low prices. However, while the hardware is progressing rapidly, the methodologies and algorithms necessary to exploit their power are not as well developed. Expanding the use of parallel computers is severely handicapped by the lack of available software. Thus, parallel algorithm design is very important to the success of parallel architectures and parallel computing in general.

The goal of a parallel algorithm is to be able to solve a single problem using several processors working together and to do so in an efficient manner. With respect to parallel computing, it is desirable that the parallel algorithm run faster than its sequential counterpart in proportion to the number of processors used. Such efficient algorithms are termed work-optimal. For many problems, efficient parallel algorithms have been developed, and several design techniques have evolved in the process. However, there are still many problems, particularly in the domain of non-numeric computations, for which work-optimal solutions are yet to be devised. And for some problems, existing strategies do not seem to be effective in leading to efficient parallel solutions.

Tied to the development of parallel algorithms is the issue of the associated parallel
data structures. Although many of the general strategies used in sequential processing are easily applied within the parallel environment (e.g. divide-and-conquer), many of the data structures commonly used in sequential computing (queue, stack, linked list) are linear in nature and do not parallelize easily. Therefore, there is a need for new access methods and data structures to be associated with the new algorithm design techniques in order to make the transition to parallel machines.

It is the goal of this dissertation to address these underdeveloped areas. With the expansion of parallel computing into so many areas of application, there is a need to formalize strategies to be used in solving problems. In particular, there is a need to be able to categorize problems and define strategies to more easily solve problems with similar structure and requirements. The intent of this work is to establish a new strategy for designing work-optimal parallel algorithms and to do so for a broad class of problems. The work begins with the study of a well-established strategy for parallel algorithm design, linked list ranking, and then moves to the development of parentheses matching as a new strategy.

1.2 Performance Metrics

To measure the effectiveness of a parallel algorithm, it is necessary to devise a method to measure the performance of that parallel algorithm with respect to a sequential algorithm or another parallel algorithm which solves the same problem. Clearly the goal of a parallel algorithm is to solve a problem faster than its sequential counterpart. Thus, an important measure of the algorithm is the running time. While the actual running time is of concern, this measurement is difficult at best. In addition to the fact that not all models are available to researchers, the dependence of performance upon the actual hardware and the manner in which that hardware is
utilized, require that general and easy to compute measures be utilized.

For a given problem, let $T_1$ and $T_p$ represent the running times of the best known sequential algorithm and the parallel algorithm using $p$ processors, respectively. Then the speedup of the parallel algorithm is given by $S = T_1 / T_p$ and the work is $W = p \cdot T_p$. If $W = O(T_1)$ then the algorithm is said to be work-optimal, and it achieves linear speedup within a constant factor. (The term cost is also used in the literature to mean work.)

Another desirable characteristic of a parallel algorithm is that of scalability. An algorithm is said to be scalable if the performance increases linearly with the number of processors utilized. This implies that the algorithm sustains good (expected) performance for a wide range of processors used. (Obviously, there is a point at which adding more processors will no longer reduce the running time.) Therefore, scalability increases the algorithm's flexibility for use on machines with varying numbers of processors.

1.3 Models of Computation

There are numerous parallel models available, some realizable, some theoretical. In general there are two categories of machines – shared memory and fixed connection topologies. This research incorporates examples of both categories. With respect to the shared memory model, the parallel random access machine (PRAM) is emphasized. With respect to the fixed connection models, the hypercube model is considered.

A survey of the field of parallel algorithms demonstrates that the preferred and widely accepted model for the design of parallel algorithms is the shared memory parallel random access machine (PRAM) model. Although not currently realizable, it provides a well-defined, easy-to-use platform for parallel algorithm design, with
algorithms already available for a broad spectrum of problems [3, 61, 52]. Unfortunately for owners of commercially available, fixed connection parallel computers, the repertoire of algorithms for these machines is not so well developed. However, one popular, commercially available fixed connection computer, the hypercube model has probably generated more activity in this area than any of the others [92, 74].

The initial research presented here begins with the development of algorithms for shared memory computers, but subsequent research reflects the interest in the underdeveloped area of fixed connection models, with particular emphasis on the hypercube. This approach will demonstrate the applicability of the results to both ends of the parallel computer spectrum.

1.3.1 PRAM Model

The parallel random access machine (PRAM) model consists of $p$ processors, each directly connected to a shared global memory. All communication among processors is through this shared memory. Each of the $p$ processors is a general purpose processor having a local memory. In general, all processors may access the shared, global memory simultaneously, with various restrictions applied to different models. Figure 1.1 shows the structure of the PRAM model.

With respect to reading, *concurrent read* indicates that two or more processors may read the same memory location simultaneously; similarly, *concurrent write* indicates that two or more processors may write to the same address simultaneously. At the other extreme, *exclusive read* or *write* implies that simultaneous access to global memory is allowed only if each processor is accessing a unique address. The four models are therefore $EREW$ (exclusive read/exclusive write), $CREW$ (concurrent read/concurrent write), $ERCW$ (exclusive read/concurrent write) and $CRCW$ (concurrent read/concurrent write), the most powerful of the four. In addition, variations
of CRCW exist which determine the method by which concurrent writes are resolved. For more details see [3].

The shared memory PRAM model is a widely accepted fundamental, theoretical model for parallel algorithm development, the use of which allows for the development of algorithms without the problems of communication delays, independent memories and distribution of data. Many researchers begin their quest for a parallel solution to a problem using this model, as it allows the researcher to concentrate on the traits of the problem itself with few architectural restrictions.

![Shared Memory PRAM Model](image)

**Figure 1.1: Shared Memory PRAM Model**

1.3.2 Hypercube Model

The hypercube is a distributed memory, message-passing, parallel computer. An *r*-dimensional hypercube is a fixed connection computer having $N = 2^r$ nodes and $r2^{r-1}$ edges. The nodes are numbered 0 to $N - 1$ in binary form and are connected so that any two nodes whose binary numbers differ in exactly one bit position are connected by an edge. Figure 1.2 shows hypercubes for $N = 2, 4$ and 8.

The hypercube computing model provides a different set of considerations to be addressed from the PRAM models. First, it is a message passing model, with each processor having a local, not shared memory. Second, because of the configuration of the hardware, communication delays between processors must be taken into ac-
Clearly, any parallel algorithm developed for a shared memory computer can be ‘forced’ onto the hypercube model and the algorithm will find a solution to the problem. The drawback is that if the data distribution and communication are not well planned, the time complexity can increase dramatically due to communication overhead. Therefore, issues such as load balancing [62, 111, 96], data distribution [97], and communication [84, 65, 64] must be carefully considered.

Figure 1.2: Hypercube Model for \( n = 2, 4, 8 \)

1.4 Parallel Algorithm Design Strategies

A paradigm is defined as a general strategy used to aid in the development of the solution to a problem. The establishment of a paradigm first involves identifying a technique and applying it to the solution of a problem. The next step is to define the algorithmic technique and demonstrate that it is applicable in the solutions of a set of related problems. The establishment of such paradigms is important in that they provide basic approaches through which new problems may be solved or through which previous problems may be solved more efficiently.
There are numerous paradigms associated with parallel algorithm development. These include divide-and-conquer [9, 36], branch-and-bound [73, 72], and dynamic programming [35, 104, 76], each of which have been extended from their sequential use. Others, such as deterministic coin tossing [25, 26], symmetry breaking [53], accelerating cascades [25], tree contraction [87, 1], Euler tours [102], linked list ranking [112, 24, 28, 6, 54], and all nearest smaller values [63], have been developed specifically for application to the development of parallel algorithms. Although this list is not complete, it serves to provide an overview of some the most widely used techniques in parallel algorithm design. To demonstrate the applicability of these techniques, a brief description is provided for each. A survey of many of these techniques is given in [109].

The divide-and-conquer paradigm [9, 36] is the very basis of parallel algorithm design. The idea is to divide a given problem into several smaller subproblems which can be solved independently. The results from each of the subproblems are then combined into the final solution for the original problem. In the parallel environment, each subproblem is assigned to a separate processor, and all subproblems are solved simultaneously. The results are combined using one or more processors. Clearly, any problem which is to be solved on a parallel computer utilizes some form of the divide-and-conquer strategy.

Dynamic programming [35, 104, 76] is a divide-and-conquer technique which is applied when the subproblems of the original problem are not independent, that is, when the subproblems share common subproblems. As subproblems are solved, the solutions are saved in a table for use as needed, as opposed to recomputing a solution each time an instance of a subproblem arises. Dynamic programming is often applied to a category of problems referred to as optimization problems in which a maximum or minimum value for the solution is desired.
Branch-and-bound [73, 72] is a breadth-first tree processing technique in which the branch of the tree to be traversed next is dependent upon the current value of a bounding function which is computed as the tree is traversed. The value of the bounding function allows some branches of the tree to be pruned (i.e. eliminated from further consideration), thus reducing the search time. The computation of the bounding function is used to lead to the solution of the original problem. The branch-and-bound strategy is often applied to problems in which the solution is determined by searching through a tree, as in a game problem.

Symmetry breaking [53] refers to a technique by which a linked structure such as a linked list is partitioned into several disjoint pieces. One specific symmetry breaking technique, deterministic coin tossing [25, 26], uses the binary representation of the index of each element to select nonadjacent elements for processing. Repeated application of the algorithm allows the user to select elements from the list whose distances between them are within a specified range. This technique is widely used among the various linked list ranking algorithms.

Accelerating cascades [25] involves applying two or more different algorithms to a single problem, changing from one algorithm to another when the ratio of the problem size to the number of processors reaches a certain level, referred to as the threshold. This use of multiple algorithms allows for the fine-tuning of an algorithm in order to obtain better performance than any one algorithm can attain when used alone.

Tree contraction [87, 1] is a tree processing technique in which the nodes of a tree are removed, and the information contained in the node being removed is combined with the information contained in its parent. In a parallel environment, processors are assigned to independent nodes so that multiple nodes are removed concurrently. When the tree has been reduced to a single node, the root, the solution to the problem is found in that node. A common application of tree contraction is to tree problems
in which the value for a given node is dependent upon the values of the nodes in its subtree, such as expression evaluation.

The Euler tour technique [102] has been successfully applied to many tree and graph problems. It involves duplicating each link in a tree (graph) pointing in the opposite direction of the original link, forming an Euler circuit or path through the tree (graph). This essentially allows the tree (graph) to be approached as if it were a linked list and, thus, allows for easier processing. This is the technique most often applied in algorithms to compute the traversals of trees (graphs).

Linked list ranking [112, 24, 28, 6] is a technique whereby the elements of a linked list are assigned a rank which corresponds to the number of elements succeeding (preceding) it in the list. (See chapter 2 for details.) This technique has been applied to a wide range of tree and graph problems. (See table 2.1.)

Given an array of numbers, the all nearest smaller values (ANSV) problem is to determine, for each number \( x \) in the array, the location of the closest number to the left (right) of \( x \) which is smaller than \( x \) [13]. The solution of ANSV has been applied successfully to several problems, including the depth first search of an interval graph [33] and parentheses matching, line packing, and triangulating a monotone polynomial [63].

A study of the application of these strategies to various problems solved in parallel reveals that it is seldom the case that only one technique is applied to a given problem. One finds that many problems of interest require a combination of techniques in order to achieve an efficient parallel algorithm. These strategies, when used in combination, serve to complement each other and allow for a wide variety of approaches in algorithm design.

In the parallel environment, there are many problems which have not yet been solved in an efficient manner. And in some cases, it seems that the currently known
strategies do not provide the necessary functionality to do so. Thus, the search for new and innovative strategies continues. As a result, this dissertation proposes to establish a new strategy - parallel parentheses matching - by demonstrating its applicability to a variety of problems.

1.5 Research Overview

The remainder of this dissertation is organized into six chapters. Chapter 2 defines the related terminology and summarizes the previous work in the area of linked list ranking, including the evolutionary development and applications. This material is taken from “A Comprehensive Survey of Linked List Ranking Algorithms” (125 pages) [54], which has been submitted to ACM Computing Surveys for possible publication. The complete survey provides detailed discussions of all known linked list ranking algorithms for the various models of parallel computation. In addition, examples for each algorithm and comparisons of the algorithms are provided. This survey is the only known complete collection and analysis of existing list ranking algorithms and provides a thorough background study for persons interested in pursuing research in this area.

Chapter 3 presents two new linked list ranking algorithms. This work is presented in “Simple Deterministic and Randomized Algorithms for Linked List Ranking on the EREW PRAM Model” (TR # CRPDC-92-21, UNT, 12 pages) [38], which has been submitted to Parallel Processing Letters for possible publication. The algorithms presented demonstrate an alternative approach to the traditional reduce-rank-expand technique that has been dominant in the past. In addition, they achieve smaller space and time complexity constants than the previously proposed algorithms.

In chapters 4 and 5, parallel parentheses matching is proposed as a general parallel
algorithm design strategy, as presented in the paper “Efficient Parallel Algorithms for Tree-Related Problems Using the Parentheses Matching Strategy” (TR # CRPDC-93-11, UNT, 25 pages) [40]. Chapter 4 describes the general strategy of parentheses matching and reviews the existing work in this area. Discussed are the relationships between parentheses and tree structures, the equivalence of the Euler tour of trees and parentheses matching, and the relationship between tree contraction and parentheses. In addition, a number of algorithms relating to these topics are developed.

In chapter 5, several new algorithms are presented in which the parentheses matching strategy is applied to a variety of tree related problems. These include algorithms for the nearest enclosing parentheses problem, the heights of all nodes in a tree, the extreme values of the nodes of the subtrees of a tree, and the lowest common ancestor problem. Finally, parentheses matching is applied to solve the problem of globally balancing a binary tree.

Chapter 6 presents a new hypercube algorithm for parentheses matching from the paper titled “A Divide-and-Conquer Hypercube Algorithm for Parentheses Matching” [39] and provides a comparison to two recently proposed parentheses matching algorithms for the hypercube model. The distinction of this algorithm is the technique used for ensuring the load balance among the processors of the hypercube. A hypercube solution to the nearest enclosing parentheses problem, which utilizes parentheses matching, is also given. Chapter 7 concludes the dissertation with a summary of the work presented here and with suggestions for future research in this area.
CHAPTER 2

EXISTING WORK ON LINKED LIST RANKING

2.1 Introduction

Linked list ranking is a well-established parallel algorithm design strategy that has been applied to many tree and graph problems. A review of its historical development and its application to numerous problems serves to demonstrate how such strategies become established. Thus, the first line of attack for this project was to survey the existing work in the area of parallel linked list ranking, including not only the applications but also the various implementations of linked list ranking itself. Through this survey, two things have been accomplished. Two new linked list ranking algorithms have been developed and are presented in chapter 3. It also provides the background necessary to propose the establishment of a new algorithm design strategy, parallel parentheses matching, as presented in chapters 4 and 5.

2.2 Problem Definition

The linked list ranking problem (also referred to as list ranking) is defined as follows. A linked list of length \( n \) is stored in an array in which each element (except the last) has a pointer (NEXT) to its successor in the list. The problem is to calculate the distance (or RANK) of each element from the end of the list, i.e., the number of elements succeeding it, including itself, in the list. Analogously, the rank of an element can be defined in terms of elements preceding it in the list. An example list is shown in figures 2.1 and 2.2. Figure 2.1 demonstrates the array implementation
of the linked list, while figure 2.2 shows the logical form of the list. As is shown, the RANK values given in the array correspond to the logical ordering of the list elements.

\[
\begin{array}{c|cccccccccccccccc}
\text{Index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\text{NEXT} & 3 & 2 & 4 & 6 & 11 & 1 & 8 & 10 & 5 & 7 & 14 & 9 & 15 & -1 & 12 & 13 \\
\text{RANK} & 16 & 11 & 10 & 15 & 9 & 12 & 14 & 6 & 13 & 7 & 5 & 8 & 3 & 1 & 4 & 2 \\
\end{array}
\]

Figure 2.1: Linked List in Array Form

Figure 2.2: Linked List in Logical Form

The list ranking problem is solved sequentially by traversing the list and numbering the items. If the number of items is not initially known, the list is first traversed to count the number of items and again for the numbering. In either case, this simple algorithm requires \(O(n)\) time.

Clearly, the nature of this problem is sequential in that for each element in the list, the value of a given element is based completely on the value of its successor (predecessor). At first glance, it appears that a parallel implementation has little to offer. However, this is not the case, and, in fact, work-optimality can be achieved through the use of a parallel algorithm. This fact makes this problem important from two points of view. First, the fact that such a problem is efficiently parallelizable provides strong evidence of the power of parallel computing. It is reasonable to assume that many other seemingly sequential problems can achieve similar results.

Second is the applicability of this algorithm to so many other problems. Linked list ranking has established itself as an important strategy in the solution of many other problems, particularly those related to trees and graphs. To underscore the im-
portance of the list ranking problem, several of these applications are briefly discussed here. A more extensive list of applications of linked list ranking and the corresponding references are shown in table 2.1.

Two common and simple applications of the list ranking algorithm are for list packing and radix (bucket) sorting. By ranking a linked list, it can then be packed into an array in which successive elements are consecutive in the array [70, 69]. After ranking the elements in the buckets, list packing is used in a radix sort to pack the sorted elements into a contiguous array [70, 69].

There are numerous tree applications. At the fundamental level, list ranking is used to construct an Euler tour of a tree. The Euler tour method is then used in solutions to the problems of tree traversals, centroid and accelerated centroid decomposition, and least common ancestors. Another important problem which utilizes list ranking is tree evaluation. Most recently Chen and Das [20] reduce tree traversals to the general list ranking problem and provide solutions to the common preorder, postorder, and inorder traversals, as well as breadth-first and breadth-depth traversals of trees.

List ranking has also proven useful in the solutions to many graph related problems. Some of these problems include Euler tours, connected and biconnected components, forest and ear decompositions, strong orientation, vertex connectivity and st-numberings.

2.3 Evolution of Parallel List Ranking Algorithms

Wyllie gave the first parallel algorithm for linked list ranking which runs in $O(\log n)$ time using $n$ processors [112]. He also conjectured that it would be impossible to produce a logarithmic time algorithm which attains linear speedup. Thus began
### Table 2.1: Applications of Linked List Ranking

<table>
<thead>
<tr>
<th>Application</th>
<th>References</th>
<th>Model</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GENERAL USE</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1) List Packing</td>
<td>KRS90, KRS86</td>
<td>EREW</td>
<td><em>(uses LLR)</em> uses (1)</td>
</tr>
<tr>
<td>(2) Radix Sort</td>
<td>KRS90, KRS86</td>
<td>EREW</td>
<td></td>
</tr>
<tr>
<td><strong>TREES</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3) Euler Tour</td>
<td>EG88, CV88b,88c,86d</td>
<td>EREW</td>
<td>*</td>
</tr>
<tr>
<td>(4a) Traversals</td>
<td>CD92a, CDA91, KRS90</td>
<td>EREW</td>
<td>* (3)</td>
</tr>
<tr>
<td></td>
<td>EG88, KRS86, KB85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4b) Breadth-first</td>
<td>CD92a, CD92b</td>
<td>EREW</td>
<td>* (4a)</td>
</tr>
<tr>
<td></td>
<td>GB84b</td>
<td>CREW</td>
<td>* (4a)</td>
</tr>
<tr>
<td>(4c) Breadth-depth</td>
<td>CD92a</td>
<td>EREW</td>
<td>* (4a)</td>
</tr>
<tr>
<td>(5) Subtree Size</td>
<td>EG88</td>
<td>EREW</td>
<td>* (3)</td>
</tr>
<tr>
<td>(6) CD/ACD</td>
<td>CV88b,c, CV86d</td>
<td>EREW</td>
<td>* (3)</td>
</tr>
<tr>
<td>(7) Tree Evaluation</td>
<td>KRS86</td>
<td>EREW</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>MR85</td>
<td>CRCW</td>
<td>*</td>
</tr>
<tr>
<td>(8) Least Com. Ancestor</td>
<td>SV88, G91, LO91b</td>
<td>EREW</td>
<td>(3,4,5,22)</td>
</tr>
<tr>
<td>(9) Spanning Tree</td>
<td>KRS90, KRS86</td>
<td>EREW</td>
<td>* (16)</td>
</tr>
<tr>
<td>(10) Tree Contraction</td>
<td>ADKP89,GMT88</td>
<td>EREW</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>MR85</td>
<td>CRCW</td>
<td>*</td>
</tr>
<tr>
<td><strong>GRAPHS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(11) Euler Tour, Partition (directed)</td>
<td>EG88</td>
<td>EREW</td>
<td>*</td>
</tr>
<tr>
<td>(12) Euler Tour (undir.)</td>
<td>EG88, AIS84</td>
<td>EREW</td>
<td>* (11)</td>
</tr>
<tr>
<td>(13a) Connected Components</td>
<td>G91, KRS90</td>
<td>EREW</td>
<td>* (4,22)</td>
</tr>
<tr>
<td>(13b) Undirected</td>
<td>KRS86</td>
<td>EREW</td>
<td></td>
</tr>
<tr>
<td>(14a) Biconnected Comp.</td>
<td>CV91</td>
<td>CRCW</td>
<td>*</td>
</tr>
<tr>
<td>(14b) Undirected</td>
<td>KRS86,90a, EG88</td>
<td>EREW</td>
<td>(13,19)</td>
</tr>
<tr>
<td>(15) Forest</td>
<td>TV85</td>
<td>CRCW,CREW</td>
<td>(9,4a)</td>
</tr>
<tr>
<td></td>
<td>KRS90, KRS86</td>
<td>EREW</td>
<td>*</td>
</tr>
<tr>
<td><strong>OTHER</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(23) Parentheses Matching</td>
<td>CD91b, PD91</td>
<td>EREW</td>
<td>*</td>
</tr>
</tbody>
</table>
the attempts by numerous researchers to develop such an algorithm. As a result, numerous algorithms on various parallel models have been added to the repertoire of parallel list ranking algorithms.

Most of the list ranking algorithms proposed since Wyllie's work use his pointer jumping algorithm as a subroutine. This is true of three deterministic, work-optimal, but non-logarithmic algorithms presented for the list ranking problem on the EREW PRAM model. These were by Kruskal, Rudolph and Snir [69], Wagner and Han [110] and Cole and Vishkin [25]. Cole and Vishkin [24, 28] also presented the first logarithmic time deterministic algorithm. This work did disprove Wyllie's earlier conjecture; however, the time-complexity constants for the algorithm were so large that the authors suggested the earlier results were actually stronger. Then, Anderson and Miller [6, 8] presented a much simpler, work-optimal, logarithmic time algorithm.

Several of the early parallel list ranking algorithms were randomized. These include algorithms by Vishkin for both the EREW and CRCW PRAM models [106, 108] and later, a simple algorithm for the EREW PRAM model by Anderson and Miller [7]. Recent interest has been shown in the asynchronous PRAM (APRAM) model, for which Martel and Subramonian proposed a work efficient algorithm in [81, 82].

The list ranking problem has also been solved on several other models of computation. Hillis and Steele [59] applied Wyllie's pointer jumping algorithm to Connection Machine, a commercially produced computer. Algorithms have been proposed for the DCM (direct connect machine) in [68, 56]. Ryu and JáJá [96] proposed an algorithm for two variations of the hypercube model. Then Sanz and Cypher [97] improved the time of that algorithm by applying their data-reduction paradigm as well as the accelerating cascades technique [25]. Olariu, Schwing, and Zhang proposed algorithms for a mesh-connected computer with a reconfigurable bus system (PARB) using several sizes of 2-dimensional meshes [89]. List ranking algorithms for both 2-
and 3-dimensional PARBs were provided by Deo, et al. in [45].

A detailed discussion of each of these algorithms is found in [54]. A summary of the algorithms and their time complexities is given in table 2.2.

Table 2.2: Comparison of List Ranking Algorithms

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Model</th>
<th>Time (order)</th>
<th>Processors</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>W79</td>
<td>EREW</td>
<td>O(log n)</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>V84,87</td>
<td>EREW</td>
<td>O((\frac{k}{p} = \log n \log^* n))</td>
<td>(\frac{n}{\log n \log \log n})</td>
<td>random</td>
</tr>
<tr>
<td></td>
<td>CRCW</td>
<td>O(log n)</td>
<td>(\frac{n}{\log n \log \log n})</td>
<td>random</td>
</tr>
<tr>
<td>MR85</td>
<td>CRCW</td>
<td>O(log n)</td>
<td>(\frac{n}{\log n \log \log n})</td>
<td>random</td>
</tr>
<tr>
<td>KRS86</td>
<td>EREW</td>
<td>O((\frac{n}{p \log(\ln p)}))</td>
<td>(n \geq p^2 + \log p)</td>
<td>div/conq</td>
</tr>
<tr>
<td>CV86b</td>
<td>EREW</td>
<td>O((\frac{n}{p \log(\ln p)}))</td>
<td>(n \geq p^2 + \log p)</td>
<td>2-rule/comp</td>
</tr>
<tr>
<td>WH86</td>
<td>EREW</td>
<td>O((\frac{n}{p \log(\ln p)}))</td>
<td>(n \geq p^2 + \log p)</td>
<td>uses W79</td>
</tr>
<tr>
<td>CV86a,88a</td>
<td>EREW</td>
<td>O(log n)</td>
<td>(\frac{n}{\log n})</td>
<td></td>
</tr>
<tr>
<td>HS86</td>
<td>Conn. Mach.</td>
<td>O(log n)</td>
<td>(\frac{n}{\log n})</td>
<td></td>
</tr>
<tr>
<td>AM88</td>
<td>EREW</td>
<td>O(log n)</td>
<td>(\frac{n}{\log n})</td>
<td></td>
</tr>
<tr>
<td>CV89</td>
<td>CRCW</td>
<td>O(log n)</td>
<td>(\frac{n}{\log n})</td>
<td></td>
</tr>
<tr>
<td>H89a</td>
<td>DCM</td>
<td>O((\frac{n}{p \log p}))</td>
<td>(p)</td>
<td></td>
</tr>
<tr>
<td>H89b.91</td>
<td>CRCW(local)</td>
<td>O((\frac{n}{p \log p}))</td>
<td>(p)</td>
<td></td>
</tr>
<tr>
<td>MS90a,b</td>
<td>Asynch.</td>
<td>O((\frac{n}{p \log(\ln p)}))</td>
<td>(\frac{n}{\log n \log \log n})</td>
<td>random</td>
</tr>
<tr>
<td>AM90</td>
<td>EREW</td>
<td>O(log n)</td>
<td>(\frac{n}{\log n \log \log n})</td>
<td>random</td>
</tr>
<tr>
<td>RJ90</td>
<td>Hypercube</td>
<td>O((\frac{n}{p \log p}))</td>
<td>(n = \Omega(p^{1+e}))</td>
<td>all-port</td>
</tr>
<tr>
<td>RJ90</td>
<td>Hypercube</td>
<td>O((\frac{n}{p \log p}))</td>
<td>(n = \Omega(p^{1+e}))</td>
<td>one port</td>
</tr>
<tr>
<td>OSZ91</td>
<td>PARBS</td>
<td>O((\frac{n}{\log n}))</td>
<td>((m(n + 1) + 2) \times 3n)</td>
<td>2 (\leq m \leq n)</td>
</tr>
<tr>
<td>OSZ91</td>
<td>PARBS</td>
<td>O((\frac{n}{\log n}))</td>
<td>((m(n + 1) + 2) \times 3n)</td>
<td>2 (\leq m \leq n)</td>
</tr>
<tr>
<td>SC92</td>
<td>Hypercube</td>
<td>O((\log^2 n \log \log n))</td>
<td>(n \geq p \geq \log(2) n \log(3) n)</td>
<td></td>
</tr>
<tr>
<td>SC92</td>
<td>Hypercube</td>
<td>O((\frac{n \log^2 n}{\log^3 n}))</td>
<td>(n \geq p \geq \log(2) n \log(3) n)</td>
<td></td>
</tr>
<tr>
<td>DGJM92</td>
<td>PARBS</td>
<td>O((\frac{n}{p \log p}))</td>
<td>(n \geq p \geq \log(2) n \log(3) n)</td>
<td>3-D</td>
</tr>
<tr>
<td>DGJM92</td>
<td>PARBS</td>
<td>O((\frac{n}{p \log p}))</td>
<td>(n \geq p \geq \log(2) n \log(3) n)</td>
<td>3-D</td>
</tr>
</tbody>
</table>

2.4 Parallel List Ranking Approaches

There is a common thread which runs through the list ranking algorithms, with few exceptions. They all tend to follow the basic strategy which is referred to as reduce - rank - expand. This technique involves deleting non-adjacent list elements in parallel until the remaining list is of size O(p) - the reduction step - where p is the number of processors. The pointer-jumping algorithm due to Wyllie [112] is then
invoked to compute the ranks of the reduced list. The deleted elements are reinserted into the list in the reverse order of their deletion, computing the ranks of each element as it is replaced — the expansion step. (See figure 2.3.)

Linked List → Reduce → Compute Ranks → Expand → Ranked List

Figure 2.3: Strategy for Work-Optimal List Ranking

Existing algorithms differ primarily in the manner in which elements are selected for concurrent deletion. The two fundamental approaches, pointer jumping and list reduction, are discussed in the following sections.

2.4.1 Pointer Jumping (Doubling) Approach

The pointer jumping approach is an extension of the prefix sums technique in which an element repeatedly determines the successor of its successor to double the length of its pointer. This is the technique used in the first parallel algorithm proposed for solving the list ranking problem [112].

The term pointer jumping is used to refer to the algorithm for list ranking proposed by Wyllie [112]. In order to maintain the integrity of the original list pointers, NEXT, the algorithm requires each element to have an additional field, JUMP, which is initially set equal to NEXT, but as elements are ‘jumped,’ the value of JUMP changes.

The technique of pointer jumping causes each element to repeatedly change its pointer (JUMP) to the successor of its current successor. Thus, after the first iteration, each element points to an element two links away. In the second iteration, this is repeated, so that each element points to an element four links away. This continues [log n] times until the first element contains a nil pointer, indicating the end of the list. This is shown in figure 2.4.

In order to rank the elements during this process, prior to ‘jumping’ an element,
the RANK of the element to be 'jumped' is added to the RANK of its predecessor element. This accumulation produces the final ranking of the elements of the list. The details are given in Algorithm JUMP, in which it is assumed that there are $n$ processors, with one assigned to each list element.

Algorithm JUMP: Linked List Ranking Using Pointer Jumping

Arrays have locations 0..$n-1$; Processors $P_0..P_{n-1}$

$P_i$ is assigned to array location $i$, $0 \leq i \leq n - 1$

for each processor $P_i$, $0 \leq i \leq n - 1$, pardo

$\text{JUMP}(i) := \text{NEXT}(i)$ \{initialize\}

$\text{RANK}(i) := 1$ \{initialize\}
while JUMP(0) ≠ nil do
  if JUMP(i) ≠ nil then
    RANK(i) := RANK(i) + RANK(JUMP(i))
    JUMP(i) := JUMP(JUMP(i))
  endif
endwhile
endfor

Figure 2.5 provides an example to demonstrate the pointer jumping method. The list used is from figures 2.1 and 2.2. The value of NEXT does not change so it is shown only in the initial list.

Time Complexity

The time required for this algorithm is $O(\log n)$ using $n$ processors, for a total work of $O(n \log n)$. It is clear that by doubling the distance between the first element and its new successor on each iteration, $\lfloor \log n \rfloor$ iterations are required in order for the first element to point to the end of the list (nil). Because the number of iterations is fixed, it is necessary to reduce the number of processors in order to achieve work-optimality. The product of the number of processors and the time per iteration must be no greater than $\frac{n}{\log n}$. Since each element is processed on each iteration, the time per iteration is $\frac{n}{p}$, $p$ being the number of processors. This implies that $p \cdot \frac{n}{p} \leq \frac{n}{\log n}$, which is clearly impossible. Thus, this technique cannot be made work-optimal.

The doubling method, while quite simple and elegant, does not work efficiently, in that each processor continues to work, even though at each iteration more of the elements' ranks have been completed. On the first iteration, one processor does no
productive work; on the second iteration, two processors are essentially idle; and on each successive iteration the number of idle processors doubles. One approach to achieving work-optimality is to reduce the number of processors in order to reduce the amount of idle processor time. Although this reduction is not sufficient for pointer jumping, it is the approach taken by the work-optimal methods which use list reduction.

(a) Original list L

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEXT</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>1</td>
<td>8</td>
<td>10</td>
<td>5</td>
<td>7</td>
<td>14</td>
<td>9</td>
<td>15</td>
<td>-1</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>JUMP</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>1</td>
<td>8</td>
<td>10</td>
<td>5</td>
<td>7</td>
<td>14</td>
<td>9</td>
<td>15</td>
<td>-1</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>RANK</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(b) List after 1 iteration - jumping 1 element

| JUMP | 6 | 4 | 11| 8 | 9 | 2 | 5 | 14| 1 | 10| 12| 7 | 13 | -1 | 15 | -1 |
| RANK | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 2 | 2 | 2 |

(c) List after 2 iterations - jumping 2 elements

| JUMP | 5 | 9 | 7 | 1 | 10| 11| 2 | 15| 4 | 12| 13| 14 | -1 | -1 | -1 | -1 |
| RANK | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 3 | 1 | 4 | 2 | 2 | 2 |

(d) List after 3 iterations - jumping 4 elements

| JUMP | 11| 12| 15| 9 | 13| 14| 7 | -1| -1| -1| -1| -1 | -1 | -1 | -1 | -1 |
| RANK | 8 | 8 | 8 | 8 | 8 | 8 | 6 | 8 | 7 | 5 | 8 | 3 | 1 | 4 | 2 | 2 |

(e) Ranked list after \( \log 16 = 4 \) iterations

| JUMP | -1| -1| -1| -1| -1| -1| -1| -1| -1| -1| -1| -1 | -1 | -1 | -1 | -1 |
| RANK | 16| 11| 10| 15| 9 | 12| 14| 6 | 13| 7 | 5 | 8 | 3 | 1 | 4 | 2 |

Figure 2.5: Pointer Jumping Method for List Ranking

2.4.2 Reduce-Rank-Expand Technique

The algorithmic framework used by most parallel list ranking algorithms is referred to as *accelerating cascades*. It is formalized by Cole and Vishkin in [25] but was used by researchers prior to the formalization. The *accelerating cascades framework* is
defined as follows. Given two algorithms A and B for a single problem such that A is more efficient than B (less operations) but B is faster than A (less time). Suppose A is an algorithm in which each phase is a reduction (smaller instance) of the problem in the previous phase. Use Algorithm A until the size of the problem is reduced to some threshold value, and then use Algorithm B to complete the solution. It is the case that the new algorithm may be faster than B and also more efficient than A. More than two algorithms may be used in this manner.

Under the accelerating cascades framework, three general algorithms are used for list ranking, a reduction algorithm, a ranking algorithm, and an expansion algorithm. The reduction algorithm begins the ranking computation by deleting elements until the list is reduced to an acceptable size, generally \( O(p) \). The ranking algorithm then completes the ranking computations on the elements remaining in the reduced list. The expansion algorithm computes the ranks of the previously deleted elements as it inserts them back into the list. This list ranking strategy, which is referred to as reduce–rank–expand, assumes the number of processors is \( p < n \).

List Reduction Stage

Deletion of an element follows the fundamental technique of deletion from a linked list; the predecessor of the deleted element is modified to point to the successor of the deleted element. In addition, the RANK of the predecessor element is incremented by the value of the RANK of the deleted element. During the reduction process, the rank of an element is one greater than the number of elements that have been deleted between the element and its current successor. The complete list ranking algorithm is shown in Algorithm REDUCE, and examples of reduction are demonstrated in figures 2.6 and 2.7.

The previously proposed algorithms differ in the manner in which the elements
are selected for removal in order to assure no deletion conflicts. The techniques used include selection of a 2-ruling set [25, 6], divide-and-conquer [69], expander graphs [28, 28], and maximal matching of a graph [110].

Algorithm REDUCE: Reduce/Rank/Expand Algorithm for Linked List Ranking

REDUCTION STAGE:

Repeat until \( O(p) \) elements remain

Select set \( S \) of non-adjacent elements from list \( L \)

for each element indexed \( A \) in \( S \) do

{delete \( L(A) \) from the list}

\[
\text{RANK(PREV}(A)) := \text{RANK}(A) + \text{RANK}(\text{PREV}(A))
\]

\[
\text{NEXT(PREV}(A)) := \text{NEXT}(A)
\]

\[
\text{PREV(NEXT}(A)) := \text{PREV}(A)
\]

Push return information on stack

endfor

endrepeat

RANKING STAGE: Use Pointer Jumping Algorithm to compute the ranks of the \( O(p) \) elements in \( O(\log p) \) time using \( p \) processors.

EXPANSION STAGE:

{expand list by reinserting deleted elements}

for each deleted element in reverse order do

\[
\text{RANK}(A) := \text{RANK(NEXT}(A)) + \text{RANK}(A)
\]

\[
\text{NEXT(PREV}(A)) := A
\]

\[
\text{PREV(NEXT}(A)) := A
\]

endfor
Steps (a), (b), and (c) of the example in figure 2.7 demonstrate how parallelism is involved in the reduction stage of the list ranking problem.

(a) Original List L

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>3</th>
<th>6</th>
<th>8</th>
<th>5</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>11</th>
<th>9</th>
<th>7</th>
<th>10</th>
<th>14</th>
<th>12</th>
<th>15</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANK</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(b) List after L(3), L(4), L(5), L(7) are deleted in parallel

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>6</th>
<th>8</th>
<th>1</th>
<th>2</th>
<th>11</th>
<th>9</th>
<th>10</th>
<th>14</th>
<th>12</th>
<th>15</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANK</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(c) After deletion of L(8), L(11), L(14) and L(13)

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>6</th>
<th>1</th>
<th>2</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANK</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

(d) Computed ranks of the elements of the reduced list

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>6</th>
<th>1</th>
<th>2</th>
<th>9</th>
<th>10</th>
<th>12</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANK</td>
<td>16</td>
<td>14</td>
<td>11</td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

(e) After reinsertion of L(8), L(11), L(14), L(13)

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>6</th>
<th>8</th>
<th>1</th>
<th>2</th>
<th>11</th>
<th>9</th>
<th>10</th>
<th>14</th>
<th>12</th>
<th>15</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANK</td>
<td>16</td>
<td>14</td>
<td>13</td>
<td>11</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

(f) After reinsertion of L(3), L(5), L(4), L(7)

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>3</th>
<th>6</th>
<th>8</th>
<th>5</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>11</th>
<th>9</th>
<th>7</th>
<th>10</th>
<th>14</th>
<th>12</th>
<th>15</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANK</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2.7: List Ranking Example

Ranking Stage

Computation of the ranks is performed when the size of the list has been reduced.
to $O(p)$. The pointer jumping algorithm [112], which schedules a constant number of elements per processor, is invoked to perform the ranking. Step (d) of figure 2.7 shows the result of the computation of ranks of the reduced list.

Expansion Stage

Once the reduced list is ranked, it is returned to its original length by reinserting the deleted elements in parallel in the reverse order of their deletion and by computing the ranks of those items. As Algorithm REDUCE shows, the steps are simply reversed, thus insuring the same time complexity for insertion as for deletion.

Recall that when an element is deleted, the contents of its fields are not affected. Therefore, the locations of its predecessor and successor at the time of its deletion are known. By returning the elements to the list in reverse order, it is assured that its predecessor and successor are in the current list.

When the element is returned to the list, the pointers of its predecessor and successor are updated to include the new element in the standard fashion. In order to update an element's rank, the RANK of the element being inserted is incremented by the RANK of its successor. Algorithm REDUCE gives the code for expansion. Figure 2.8 demonstrates this process, and steps (e) and (f) of figure 2.7 show the results of the expansion stage for the example.

It should be noted that the expansion step is not trivial. The difficulty is in how to maintain the ordering of the elements so that the elements are returned in reverse order and elements which are deleted concurrently are returned concurrently. Clearly, a stack is the data structure of choice for reversing the order of a sequence. However, a single, global stack for a parallel algorithm introduces contention issues which must be avoided. In addition, the various techniques are such that on each iteration a single processor may delete zero, one, or two elements, with each processor deleting
(a) Original list L

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>3</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANK</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(b) After deletion of L(3), L(6), sequentially, and computation of the ranks

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANK</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

L(3): PREV(3) = 0, NEXT(3) = 6, RANK(3) = 1
L(6): PREV(6) = 0, NEXT(6) = 8, RANK(6) = 1

(c) Ranked list after insertion of L(6)

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANK</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

(d) Ranked list after insertion of L(3)

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>3</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANK</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2.8: Linked List Expansion

a different number. Just as some processors are idle during the reduction stage, they must also be idle during the corresponding expansion phase.

The two methods proposed here for maintaining this ordering are similar in concept to the method suggested by Cole and Vishkin [25] but differ in the implementation details. Each processor maintains its own local stack. During the reduction stage, when a processor deletes an element, the identity of the element (its index) is pushed onto the stack. During any iteration in which a processor does not locate an element to be deleted, it pushes a ‘dummy’ or null entry onto the stack. Thus, each processor will have a full set of entries, one corresponding to each possible deletion. During the expansion stage, each processor pops an item off its local stack. If an item is a valid element, it is returned to the list; if it is null, the processor sits idle for the remainder of that iteration. Obviously, the space requirements can be
quite high, $O(np)$. However, the algorithms which use this technique have a relatively well-balanced workload per processor, thereby reducing this space requirement significantly.

Another possibility for reducing the space requirement is to introduce a global timing mechanism to be stored with each stack entry. Each processor maintains a 'timer' which is initialized to zero. On each iteration of the reduction stage, each processor increments its local timer, thus, ensuring all the local timers have the same value. If an element is deleted, the timer value is stored with the entry in the stack; otherwise, no entry is placed on the processor's stack. The result is that the stacks contain entries only for actual deletions, with each stack entry time-stamped.

During the expansion stage, each processor decrements its timer on each iteration and then looks at the top stack entry. If the current value of the timer and the time-stamp of the stack entry are the same, the element is popped from the stack and returned to the list. Otherwise, the processor sits idle until the next iteration when it decrements the timer and checks the stack entry again. This implementation ensures the proper ordering of the elements while reducing the actual space requirements.

2.5 Conclusion

This study demonstrates the gradual refinement of linked list ranking as a general design strategy and its increasing application to a variety of problems over time. Clearly, linked list ranking has become well established as a design strategy for parallel algorithms.

The reduce–rank–expand strategy is the technique of choice for all the PRAM linked list ranking algorithms and for the majority of those for other models. The variations among the algorithms are in the manner in which independent elements are
selected for deletion. Presented in the next chapter are two algorithms which follow a different line of approach.
CHAPTER 3

LINKED LIST RANKING ALGORITHMS ON EREW PRAM

3.1 Introduction

An asynchronous, CRCW PRAM (or APRAM) algorithm for linked list ranking, proposed by Martel and Subramonian [MS90a, MS90b], performs $O(n \log \log n)$ expected work employing $\frac{n}{\log n}$ processors. Motivated by their unique approach, two EREW list ranking algorithms are proposed — one deterministic and the other randomized. The deterministic algorithm performs in $O\left(\frac{n}{p} \log (n/p)\right)$ time using $p$ processors, where $n \geq p \log p$. Thus, for $p = O(n/\log n)$, it requires $O(\log n \log \log n)$ time and a work of $O(n \log \log n)$. Although not work-optimal, this algorithm is very simple compared to the known work-optimal (deterministic) EREW algorithms for list ranking and has the added advantage of small constant factors in the time and space requirements. The randomized algorithm follows the same line of approach, but uses randomization in one step to decrease the time complexity. With high probability, it requires $O\left(\frac{n}{p} + \log p\right)$ time and, hence, it is an $O(\log n)$-time, work-optimal algorithm employing $p = O(n/\log n)$ processors. Furthermore, it uses less space than the deterministic algorithm.

3.2 APRAM Model and Performance Metrics

A variation of the PRAM model which has recently drawn interest is the $APRAM$ or $asynchronous\ PRAM$ [CZ89, G89, MS90a, MS90b]. This model has the same features as the CRCW PRAM with the exception that the processors may all have
different clock speeds, causing them to work asynchronously. In addition, each processor has access to an independent random-number generator, which is used to allow processors to randomly select the work to be performed.

An algorithm is said to be randomized if some portion of its outcome is non-deterministic, with its performance stated in terms of the expected time complexity. The notation $EO(f(n))$ is used to represent the expected order of various features of a randomized algorithm, including time, space, work and number of elements. The notation $EO(f(n))$ is used to indicate that, with a high probability, the results are within the specified order of $f(n)$.

In order to accurately describe the performance of an asynchronous PRAM algorithm, a different performance metric, also called work, is used in [MPS89, MS90a, MS90b]. The work of a single execution of an algorithm is the total number of single processor instructions performed by the set of asynchronous parallel processors, including busy wait instructions. Because of the randomization incorporated into the asynchronous algorithms, the performance is expected work, which is comparable to the work of a synchronous algorithm.

3.3 APRAM List Ranking Algorithm

The APRAM algorithm which was proposed by Martel and Subramonian requires $EO(n \log \log n)$ expected work using $p = O(n / \log n)$ processors [MS90a, MS90b]. This result is dependent upon the distance between elements which are selected for processing. The following lemma places a probabilistic bound on this distance and computes an expected value.

**Lemma 1** [MS90a, MS90b]: Consider $n/\log n$ random selections with replacement from an ordered list of $n$ cells. Let $X$ be the maximum number of contiguously unse-
lected cells. Then, \( P[X > 4\log^2 n] < 1/n \) and \( E[X] = O(\log^2 n) \).

The APRAM algorithm assumes that a singly linked list is stored in an array. The processors randomly select a set of elements which, based on Lemma 1, have a high probability of being evenly distributed within the list. Knowing that the selected elements will have their ranks computed first, pointer jumping is used to cause each list element, including the selected elements, to point to its nearest selected successor or to have a pointer of length \( \log n \). In order to be able to compute the ranks, the number of pointers jumped is retained in each element.

The selected elements are then compacted into a smaller array where they are ranked using a pointer jumping algorithm [MPS89, W79]. The ranks are written back into the original array. Each unranked element then follows its pointers to read the rank of its nearest selected successor and compute its own rank. An overview of the APRAM algorithm by Martel and Subramonian [81, 82] is provided for the sake of completeness.

Algorithm Asynchronous List Ranking [81, 82]

Step 1. Select \( m = EO(\frac{n}{\log n}) \) elements at random by generating a random number between 1 and \( n \) for each element in the list and selecting those elements having values between 1 and \( \frac{n}{\log n} \), inclusive.

Step 2. Perform \( \log \log n \) iterations of pointer jumping on the list elements so that each element has a pointer to the end of the list, to a selected element or has a pointer of length \( \log n \). (That is, perform each iteration on all elements before proceeding to the next iteration.)

Step 3. Compact the \( m \) selected elements into a smaller array.

Step 4. Follow the pointer of each selected element to guarantee that it points to the next selected element or the end of the list.
Step 5. Compute the ranks of the elements of the reduced list.

Step 6. Copy these ranks into the complete list (resulting from Step 2). Follow the pointer of each non-selected element $x$ to a successor element $y$, which is ranked, and compute $RANK(x) = RANK(x) + RANK(y)$.

In general, APRAM algorithms require a synchronization mechanism to guarantee that in each step of the algorithm, all data elements are processed before proceeding to the next step. As a processor completes the processing of a data element, a group of flags is set. It may be the case that more than one processor will read or set (write) the same flag concurrently. This requires both the concurrent read and concurrent write (CRCW) capabilities of the model, which is independent of any particular algorithm. In addition, the APRAM list ranking algorithm itself may require concurrent read of other elements in steps 2, 4 and 6, while the processors follow pointers asynchronously through the list.

3.4 EREW List Ranking Algorithms

In order to devise an EREW PRAM algorithm, two modifications must be made to the APRAM algorithm, which inherently uses concurrent read and write capabilities. First is the elimination of the synchronization mechanism. Since the processors of the PRAM model operate synchronously, there is no need for the software synchronization program used by the APRAM.

A modification of steps 2, 4 and 6 in section 3 employs a partitioning of the data to eliminate concurrent reads, as well. This approach requires that a linked list of size $O(p)$, where $p$ is the number of processors, be constructed from the original list so that the elements in the shorter list (referred to as selected elements) are equally spaced in the original list with respect to their pointer distances. The APRAM
algorithm accomplishes this task while at the same time causing the pointer of every
list element to point to its nearest selected successor. The proposed EREW algorithms
construct the shorter linked list of selected elements while leaving the pointers of all
the unselected elements intact. As a result, the last step of the algorithms also
vary from that of the APRAM algorithm. To be more precise, while the APRAM
algorithm causes each element to concurrently read the rank of its successor, the
EREW algorithms cause each processor to scan the list from a selected element,
computing the ranks as it proceeds.

The details of the two EREW algorithms are given in the following subsections.
They vary primarily in the manner in which elements are selected for the shorter list.
The randomized algorithm uses Las Vegas style randomization and closely follows
the APRAM method. The deterministic algorithm uses a totally different approach,
referred to as deterministic coin tossing, due to Cole and Vishkin [CV86b].

Assume a linked list of n elements is stored in a contiguous array. The determin-
istic (randomized) algorithm assumes a doubly-linked (singly-linked) list. In addition
to the list fields, an array, RANK (initialized to 1's), is used to store the computed
rank of each element, and the array, STATUS, stores the flag for selected elements.

3.4.1 Randomized Algorithm

The first step of the randomized algorithm follows very closely the APRAM algo-
rithm by randomly selecting $EO(p)$ elements from the input list. These elements are
then compacted into a smaller array, and each processor is assigned to a partition of
size $O(1)$. Each processor scans the original list starting at its assigned elements to
find the first successor element which has STATUS = selected. The pointers and the
pointer distances between the selected elements are stored in the compacted array,
thus constructing a linked list of the selected elements and the (pointer) distance
between them. The compacted list is ranked by the pointer distances using a pointer jumping algorithm [W79], and the ranks are written back into the original list array. Processors again scan the linked list from their assigned elements to compute the ranks of the remaining elements.

Illustrative Example

Assume that \( p = 4 \). The 16-element list is given in table 3.1 where \( L(0) \) is the head of the list and points to \( L(3) \). The array is divided into \( p \) partitions with a processor assigned to each one. A processor processes its assigned partition by generating a random number between 1 and \( n = 16 \) and marking as selected those elements having values between 1 and 4, inclusive. The selected elements are marked with \( * \) in step 1 of table 3.1.

In step 2 the elements are compacted into a smaller array and linked together based on their locations in the original list and the pointer distances between successive elements computed. (The actual compaction is not shown for brevity.) For example, suppose that processor \( P_0 \) is assigned to \( L(0) \) in the list. It follows the links from \( L(0) \) to \( L(3) \), then to \( L(6) \), and finally to \( L(8) \), which is also selected. Thus, \( P_0 \) sets \( \text{RANK}(0) = 3 \) and changes its pointer to the location of \( L(8) \) in the compacted array. In Step 3, the compacted array is partitioned, each processor is assigned to a constant number of elements, and the ranks are computed. In Step 4 the processors write the ranks of their assigned elements back into the original array. Each processor then scans forward from its assigned element, computing the ranks of the unselected elements until it encounters another selected element. For example, \( P_0 \) writing the rank of 16 into \( L(0) \) again follows the pointers to \( L(3) \) and \( L(6) \) computing their ranks as 15 and 14, respectively.
Table 3.1: Demonstration of the EREW Randomized Algorithm

| List L | 0  | 3  | 6  | 8  | 5  | 1  | 2  | 4  | 11 | 9  | 7  | 10 | 14 | 12 | 15 | 13 |
|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| RANK   | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |

**Step 1**

| Random # | 3  | 8  | 10 | 3  | 9  | 13 | 16 | 11 | 2  | 15 | 7  | 1  | 8  | 6  | 4  | 12 |
|-----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Select    | *  | *  |    | *  |    | *  | *  |    |    |    |    |    |    |    |    |

**Step 2**

| New list | 0  | 8  |    | 11 | 10 | 15 |    |    |    |    |    |    |    |    |    |
| RANK      | 3  | 5  |    | 3  | 3  | 2  |    |    |    |    |    |    |    |    |    |

**Step 3**

| RANK      | 6  | 13 | 8  | 5  |    |    |    |    |    |    |    |    |    |    |    |

**Step 4**

| RANK      | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  |

Complexity Analysis

Before analyzing the performance of the randomized algorithm, a generalization of Lemma 1 (from [81]) is given to bound the distance between selected elements.

**Lemma 2:** Consider \( p \) random selections with replacement from an ordered list of \( n \) elements. Let \( X \) be the maximum number of contiguous unselected elements, and \( k < p \) be a function of \( n \). Then \( P[X \geq 2n/k] \leq ke^{-p/k} \) and \( E[X] \leq 2n/k - n(k - 1)e^{p/k} \).

**Proof:** Assume the list is divided into \( k \) partitions each containing \( n/k \) elements. The probability that any given cell is selected is \( p/n \). Define \( Y_i = 0 \) if no cell in the \( i \)th partition is selected, and \( Y_i = 1 \) otherwise. Then

\[
P[Y_i = 0] = (1 - \frac{p}{n})^{n/k} \rightarrow e^{-p/k}
\]

\[
P[X \geq 2n/k] \leq P[\text{at least one } Y_i = 0] \leq \sum_{i=1}^{k} P[Y_i = 0] = ke^{-p/k}
\]

\[
E[X] \leq nke^{-p/k} + (1 - ke^{-p/k}) * \frac{2n}{k} = \frac{2n}{k} - n(k - 1)e^{-p/k}
\]

By selecting the partition size of \( k = n/(2\log^2 n) \) and \( p = n/\log n \), the results obtained are the same as in [MS90a, MS90b]. That is, \( P[X \leq 4\log^2 n] < 1/n \) and \( E[X] = O(\log^2 n) \). \( \Box \)
The random selection of elements requires $O(n/p)$ time. With high probability, $O(p)$ elements are selected which are $O(n/p)$ distance apart, according to Lemma 2. The compaction requires $O(n/p + \log p)$ time using the parallel prefix sums algorithm. Forming the linked list of selected elements requires $EO(n/p)$ time. Using the pointer jumping algorithm [W79], the reduced list is ranked in $EO(\log p)$ time. Because there are $EO(p)$ elements, each processor is assigned to $EO(1)$ elements. Thus, for each processor, writing requires $EO(1)$ time and the sequential scan of the list requires $EO(n/p)$ time. Therefore, the overall time requirement for the randomized EREW algorithm is $EO(n/p + \log p)$, which provides work-optimal speedup for $p \leq O(n/\log n)$ processors, achieving $EO(\log n)$ time.

3.4.2 Deterministic Algorithm

In order to deterministically accomplish the results of the randomized Step 1 of the previous algorithms, the technique of constructing a 2-ruling set is used. Given an $n$-element linked list, a 2-ruling set is a subset $U$ of elements such that no two elements of $U$ are adjacent and for every element in the list, there is a directed path from it to some element in $U$, having path length at most two. In more practical terms, there are exactly one or two non-ruling set elements between each ruling set element and its nearest ruling set successor. Using the algorithm by Cole and Vishkin [25], the 2-ruling set of a linked list is constructed in $O(n/p)$ time using $p$ processors. The 2-ruling set algorithm is first applied to the original list and then to the newly constructed 2-ruling set to increase the distance between the selected elements. By repeatedly applying this algorithm $O(\log n/p)$ times, elements are eliminated from the set and the distance between ruling set elements is doubled each time, thus ensuring that $O(p)$ elements are selected and the distance between successive elements is $O(n/p)$.

As in the randomized algorithm, the selected list is then compacted into a smaller
array, where the ranking is computed via pointer jumping [W79]. The processors write the ranks back into the original array and scan the list to compute the remaining ranks. The steps of the deterministic EREW PRAM algorithm are formally stated as follows.

Algorithm Deterministic EREW List Ranking

Step 1. Select a ruling set of $O(p)$ elements from the list so that the elements are $O\left(\frac{n}{p}\right)$ distance apart, forming a linked list of the selected elements and computing the actual distance between each pair of elements. This is accomplished by repeatedly constructing a 2-ruling set of the previous ruling set, increasing the distance between selected elements on each iteration until the distance between the elements is $\left(\frac{n}{p}\right)$. Add the head of the list to the ruling set if not already selected.

Step 2. Compact the $O\left(\frac{n}{p}\right)$ selected elements into a separate array.

Step 3. Rank the elements of the selected list.

Step 4. Write the rankings back to the original array. Each processor scans forward in the original list, computing the ranks of its successor elements until another ruling set element is encountered.

Illustrative Example

In the deterministic algorithm, the only significant change from the randomized algorithm is in step 1. Each processor is assigned to a partition of the original array for which a 2-ruling set is computed as the first iteration. These selected elements are linked together to form a linked list with the pointer distances between successive selected elements stored in the RANK field as shown in the first iteration of table 3.2.
Table 3.2: Demonstration of the EREW Deterministic Algorithm

| List L   | 0  | 3  | 6  | 8  | 5  | 1  | 2  | 4  | 11 | 9  | 7  | 10 | 14 | 12 | 15 | 13 |
| RANK     | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
| Step 1   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| (iteration 1) | (a) Ruling | * * * * * * * * * * | (b) List | 0 3 8 1 4 7 12 13 |
| RANK     | 1 2 2 2 3 3 3 2 1 |              |
| (iteration 2) | (a) Ruling | * * * * * * * | (b) List | 0 3 1 7 13 |
| RANK     | 1 4 5 5 1 |              |
| Step 3   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| RANK     | 16 15 11 6 1 |              |
| Step 4   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Final RANK | 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 |              |

On the second iteration, each processor scans its entire partition but computes the ruling set based only on the elements selected in the previous iteration.

In step 2, the elements that have been selected are compacted to a new array. (This is not explicitly shown in the example.) Step 3 causes the ruling set elements to be ranked. In step 4, the ranks are written into the original array. The processors then scan the original pointers to compute the ranks as in the previous example.

Complexity Analysis

In step 1 of the algorithm, the selection of the set of $O(p)$ elements that are $O\left(\frac{n}{p}\right)$ distance apart requires $O\left(\frac{n}{p}\log\frac{n}{p}\right)$ time, as detailed below. Steps 2 and 3 require $O\left(\frac{n}{p} + \log p\right)$ and $O(\log p)$ time, respectively. In Step 4, because there are $O(p)$ elements and each processor is assigned $O(1)$ elements, writing takes $O(1)$ time while the sequential scan of the list requires $O\left(\frac{n}{p}\right)$ time.

The implementation and analysis of Step 1 are now detailed. The list is divided
into $p$ partitions, each being assigned to a processor which repeatedly performs the following steps, (a) and (b), until the size of the selected ruling set is $O(p)$.

(a) Construct a 2-ruling set using deterministic coin tossing [25] in $O(\frac{n}{p})$ time. At most, $\frac{n}{2}$ elements are in the ruling set and as few as $\frac{n}{3}$ elements may be selected. On successive iterations, construct a ruling set from that obtained in the previous iteration.

(b) Follow the links from each ruling set element to the next such element to create a linked list consisting of the ruling set elements only. Count the number of links traversed (which is at most 3 links for each element) by adding the contents of the RANK field of the element 'jumped' to the ruling set element being processed. Since each processor is assigned at most $\frac{n}{p}$ elements, this step requires $O(\frac{n}{p})$ time.

The operations described in (a) and (b) above are repeated to reduce the number of elements in the set to $O(p)$. Let $m = \frac{n}{p}$ for the original values of $n$ and $p$. Since no compaction is used between iterations of step 1, each iteration requires $m = O(\frac{n}{p})$ time. Although the number of elements which are actually being used in the computation of the ruling set is reduced by one-half on each iteration, each processor must scan its entire partition to 'find' these elements.\(^2\) The following recurrence equation describes the time complexity of step 1 of the algorithm.

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + O(m) & \text{for } n > p \\ O(1) & \text{for } n \leq p \end{cases}$$

The recurrence equation has the solution $T(n) = O\left(\frac{n}{p} \log \frac{n}{p}\right)$, making the overall time complexity for the deterministic algorithm $O(\log p + \frac{n}{p} \log \frac{n}{p}) = O\left(\frac{n}{p} \log \frac{n}{p}\right)$ for $n \geq p \log p$. Thus, for $p = O\left(\frac{n}{\log n}\right)$, it attains time complexity $O(\log n \log \log n)$ and work $O(n \log \log n)$.

\(^2\)Analysis of a version which does compact between steps shows that the overall time performance of the algorithm is not affected. However, the incorporation of compaction incurs the penalty of a larger space requirement and causes the algorithm to be less simple.
3.5 Advantages of New Algorithms

The proposed EREW algorithms are simple and use very basic parallel techniques. Therefore, they are easy to implement. They also have advantages over other list ranking algorithms, as outlined below.

1. Small Constants: All of the previously known, work-optimal EREW algorithms use the reduce-rank-expand method [7, 24, 25, 28, 69, 110]. This approach requires the additional $O(\log n)$ phase of rebuilding the reduced list after it has been ranked. Several of these algorithms use time consuming techniques such as recursively compacting the array as the list is reduced [25, 28, 69, 110] or generating expander graphs for balancing the work load among processors [24, 28]. Since these techniques are not used in the algorithms proposed here, the constant factors involved in the time complexity analysis are reduced.

2. Small Space Requirements: Table 3.3 provides a comparison of the required work space of six work-optimal EREW list-ranking algorithms (five deterministic and one randomized) to the proposed algorithms. The space requirements of these algorithms may be due to compaction arrays, deletion stacks and list fields. From this table it is seen that all previous algorithms require stack space for reduction, and all but #5 and #6 use repeated compaction.

It is assumed that the linked list for each algorithm is stored in a contiguous array with the minimal fields of NEXT (a pointer to the successor element), PREV (a pointer to the predecessor element), and RANK. All those using reduction (#1 - #6) also require a STATUS field. All but #1 use deterministic coin tossing or an equivalent technique which requires at least three additional fields, for each element of the list. Algorithm #4 requires a counter field for each element to determine the number of successors that have been deleted. Clearly, the space required for the list
Table 3.3: Space Requirements of List Ranking Algorithms

<table>
<thead>
<tr>
<th>ALGORITHM</th>
<th>FIELDS</th>
<th>OTHERS</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1: Divide &amp; Conquer [KRS86]</td>
<td>4n</td>
<td>recursion</td>
<td>9n</td>
</tr>
<tr>
<td></td>
<td></td>
<td>stack (n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>compaction(4n)</td>
<td></td>
</tr>
<tr>
<td>#2: Ruling Set with Compaction [CV86b]</td>
<td>7n</td>
<td>stack(n)</td>
<td>15n</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compaction(7n)</td>
<td></td>
</tr>
<tr>
<td>#3: Reduction Using Maximal Matching [WH86]</td>
<td>7n</td>
<td>stack(n)</td>
<td>15n</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compaction(7n)</td>
<td></td>
</tr>
<tr>
<td>#4: *Ranking With Expander Graphs [CV86a,CV88a]</td>
<td>8n</td>
<td>expander graph(3n)</td>
<td>26n</td>
</tr>
<tr>
<td></td>
<td></td>
<td>binary trees(4n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>stack(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>compaction(8n)</td>
<td></td>
</tr>
<tr>
<td>#5: *Ruling Set without Compaction [AM88]</td>
<td>7n</td>
<td>stack(n)</td>
<td>8n</td>
</tr>
<tr>
<td>#6: *Coin Tossing without Compaction [AM90]</td>
<td>5n</td>
<td>permutation array(n)</td>
<td>6n</td>
</tr>
<tr>
<td>#7: Pointer Jumping on a Ruling Set [DH92]</td>
<td></td>
<td>a) deterministic</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b) randomized</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7n</td>
<td>one compaction(n)</td>
<td>8n</td>
</tr>
<tr>
<td></td>
<td>3n</td>
<td>one compaction(n)</td>
<td>4n</td>
</tr>
</tbody>
</table>

*O(log n) time complexity

is O(n) for all the algorithms, but the variation in the constant factor is shown in the first column of table 3.3.

In the reduce-rank-expand technique of list ranking, each iteration of the algorithm causes approximately one-half of the elements to be marked as deleted. The unmarked elements are then compacted into a new array, producing a shorter list. The new list is then processed in a similar manner to reduce it by approximately one-half. This reduction continues until the list is of size O(p). Because the information in each reduced list is necessary for the expansion phase, a new array must be used for each iteration. The total space used for compaction arrays is O(n), with the constants being that used for storage of the original array.
The column labeled OTHERS gives the space required by other data structures. The numbers in parentheses there indicate the minimum amount of space required. The last column indicates the minimum total space required for the implementation of the algorithm.

As shown table 3.3, the proposed deterministic algorithm requires significantly less space than all but one algorithm. Also, the randomized implementation requires less space than the well-known randomized algorithm [7].

3.6 Conclusion

Although several work-optimal linked list ranking algorithms are known for the EREW PRAM model, the deterministic and randomized algorithms proposed here provide a simple method of list ranking which is efficient in both time and space requirements. It still remains to be determined if the deterministic algorithm can be modified to achieve work-optimal time.
4.1 Introduction

In the area of parallel algorithm design, there is a need to categorize strategies in order to efficiently solve a broad class of non-numeric problems with similar structures and requirements. Although several strategies or paradigms\(^1\) have been developed for designing parallel algorithms, there are still problems for which existing techniques do not seem to lead to efficient solutions or for which new paradigms are necessary. Therefore, the search for efficient design strategies continues.

In the search for unified strategies for designing efficient parallel algorithms, several problems (particularly, related to graphs and trees) have been encountered which are elegantly solved by applying the solution of the well-known parentheses matching problem as an intermediate step. Other than the natural application of parsing arithmetic expressions and dynamic expression evaluation [52, 12], this approach has been taken for minimum coloring of interval graphs [33], breadth-first traversal of trees [21], sorting a special class of integers [21], approximate bin packing [5], string and dictionary matching [4], and maximal matching of cographs [77]. Thus, there is current interest and promise in this work.

However, for the majority of these seemingly unrelated problems, there is no obvious correlation with parentheses matching, and they come from different domains

---

\(^1\)By a paradigm, we mean a general strategy which can be used to solve a wide class of problems.
or applications. Furthermore, parentheses matching has so far been applied on an ad
hoc basis, and to the best of our knowledge, there has been no effort to formalize this
strategy. Hence, the motivation for this work.

In this dissertation, parallel parentheses matching (PPM) is proposed as
a new strategy for designing efficient parallel algorithms. Influenced by the study of
the development of linked list ranking as a design strategy, four primary objectives
addressed:

1. Identify problems for which parentheses matching has previously been applied
   in the solution.
2. Describe the parentheses matching technique and its implementations.
3. Identify specific problems which can be efficiently solved using the parentheses
   matching strategy and develop the associated parallel algorithms.
4. Identify classes of problems for which parentheses matching is likely to be a
   good design technique.

Given a problem, the approach involves transforming it into an equivalent paren-
theses matching problem (which is most often non-trivial) and then applying an
efficient parentheses matching algorithm in parallel. The final step is to interpret
(crucial) the matching information as it applies to the solution of the original prob-
lem. Figure 4.1 shows a schematic diagram of this new strategy.

Parentheses matching as a design strategy is introduced in this chapter through
a presentation of the existing work. It begins with a discussion of the independent
algorithms which have appeared in the literature. This is followed by a detailed
discussion of its relation to tree related problems. This work clearly demonstrates the applicability of PPM to the class of problems related to trees and provides evidence of its probable application to a variety of other types of problems in the future.

4.2 Existing Work with Parentheses Matching

The *parentheses matching* problem is defined as follows. Given a well-formed sequence of parentheses stored in an array, determine the index of the mate of each parenthesis stored in the array. Generally, it is assumed that the input string of parentheses is stored in an array INPUT[1...n]. (Without loss of generality, assume that $n = 2^k$.) The solution to the problem is an array MATCH[1...n] such that MATCH[i] = j (and, likewise, MATCH[j] = i) if and only if the parentheses in locations i and j of INPUT are mates.

The sequential solution to the parentheses matching problem is quite straightforward using a stack. As the input string is read, left parentheses are pushed onto the stack. When a right parentheses is encountered, the stack is popped. This pair of left and right parentheses is a matched pair. Direct implementation of the sequential solution on common parallel models is not feasible because of the contention problems introduced by the sequential nature of the stack. Therefore, most parallel solutions take a different approach.

Several algorithms have been proposed to solve the parentheses matching problem on the PRAM model of computation. Dekel and Sahni [43] proposed an early algorithm for the EREW model in which they implemented the best known sequential algorithm in parallel. It achieves $O(\log^2 n)$ time when using n processors or $O(\log n)$ time when using $n^2/\log n$ processors. Bar-On and Vishkin [12] proposed an optimal CREW PRAM algorithm based on a tree structure in which the parentheses
are partitioned equally among the processors and each processor determines the local matches within its assigned string. Then each processor determines the match of its leftmost left and rightmost right parentheses and uses this information to match the remaining parentheses in the assigned subsequence. The algorithm achieves $O(\log n)$ time using $\frac{n}{\log n}$ processors. The algorithm can also be implemented on the EREW model in $O(\log^2 n)$ time using $\frac{n}{\log n}$ processors, thus reducing the number of processors used in [43].

The first optimal EREW PRAM algorithm was proposed by Tsang, et al. in [103]. The three phase algorithm logically organizes the processors into a binary tree and partitions the parentheses string among the leaf processors. First the local matches within each partition are computed. Each processor then sends its remaining parentheses information to its parent processor, where the information from the two children are combined. The information consists of a triple $(m, l, r)$ where $m$ is the number of pairs matched at that level and $l$ and $r$ are the number of unmatched left and right parentheses, respectively. The final phase begins when a triple is received by the root processor, at which point a unique identifier may be assigned to each matched pair. The identifier is assigned as the information returns to the leaf processors. By pipelining the operations in the final stage, the algorithm achieves $O(\log n)$ time using $O\left(\frac{n}{\log n}\right)$ processors.

In [34] three parentheses matching algorithms are proposed, two for the EREW model and one for the CREW. One EREW algorithm, although not work-optimal, provides a simple divide-and-conquer approach. It requires $O(\log^2 n)$ time using $\frac{n}{\log n}$ processors. (Details are provided in chapter 6.)

The CREW algorithm [34] is work-optimal, achieving $O(\log n)$ time using $\frac{n}{\log n}$ processors. The input string of parentheses is distributed evenly among the processors, and each processor determines the local matches within its string. Within
the substring of each processor, the leftmost left and rightmost right parentheses are marked as the representatives of the string and their mates are found. The unmatched strings within each processor are then encoded to provide the number of consecutive left (right) parentheses and the ending (beginning) index of the string. Figure 4.2 demonstrates this encoding scheme. The encoded strings are then merged by pairs in a tree-like fashion. The matching information is coded as a superscript \(<i,j>\). On a left parentheses, the superscript indicates that its mate is the \(j^{th}\) unmatched right parentheses to the left of \()\_i\). It is defined similarly for a right parentheses, and each subscript is distinguished to indicate whether the mate is found while traversing up or down the merge tree. At each stage, matched pairs are removed, and the remaining strings are re-encoded. Once the mates for some parentheses are found, the information is provided to the children nodes to continue computing mates of the parentheses in the children substrings.

\[
)7)s)9(10→11→3)s(102
\]

Figure 4.2: Encoding of Parentheses

Concurrent reads are necessary when distributing the subscript information between levels of the tree. However, by duplicating unmatched parentheses information and carrying it through the tree, the algorithm can be implemented on the EREW model in the same time bounds, but at the cost of additional space.

The EREW work-optimal algorithm proposed in [34] is referred to by its authors as a privatized, match-and-copy algorithm. It uses an approach similar to that in the CREW algorithm, as well as matching arrays and a variety of variables, all stored in the local memory of each processor. The algorithm requires \(O(\frac{p}{p} + \log p)\) time and space when using \(p\) processors. Optimal EREW algorithms have also been proposed by several others, including [19, 46, 75, 94].
4.3 Existing Work in the Application of Parentheses Matching

Other than the natural application of parentheses matching to parallel parsing of arithmetic expressions [44, 43, 52], some attempts have recently been made in applying PPM as a subproblem to the solution of other problems in parallel. These include the Euler tour of trees [22], minimum coloring of an interval graph [18], breadth first traversal of general trees and sorting of integers in a restricted class [21], bin packing [5], string and dictionary matching [4], and cograph matching [77].

However, each of these solutions was developed independently of the others. That is, there has been no known effort to characterize problems that lend themselves to the application of this technique. This dissertation presents the first stage of work in this area. This section provides a brief overview of three of these previously solved problems to familiarize the reader with the nature of the technique.

4.3.1 Breadth-first Traversal of a Tree

Chen and Das [21] designed a simple algorithm for the breadth-first traversal of a general tree using PPM. The algorithm determines the arc sequence corresponding to an Euler tour of the tree to compute the level number of each arc. After deleting the leftmost and rightmost arcs on each level, it assigns a left parenthesis to each backward arc and a right parenthesis to each forward arc. Using parentheses matching, it computes the mate (match) of each parentheses (i.e., arc). The match of the rightmost arc in level \(i\) is defined to be the leftmost arc in level \(i + 1\).

Let \(A = < x, y >\) indicate an arc leaving node \(x\) and entering node \(y\). A matched pair of arcs \((A, B)\), where \(A = < x, y >\) and \(B = < w, z >\), is interpreted as \(\text{NEXT}(x) = z\). The solution is completed by defining \(\text{NEXT}(x) = y\) if arc \(A\) is the first arc in the arc sequence. The \(\text{NEXT}\) function converts the tree into a linked
list of tree nodes, and performing parallel linked list ranking enumerates the nodes in breadth-first traversal order. The algorithm runs in $O(\log n)$ time using $\frac{n}{\log n}$ processors on EREW PRAM model.

4.3.2 Sorting Integers in a Restricted Class

Chen and Das [21] also used the PPM strategy to sort a sequence consisting of a restricted subclass of integers (RSCI), satisfying the property that any two consecutive elements in the sequence differ in value by at most one. The underlying idea is to construct a tree such that the level numbers of nodes visited while traversing the tree according to its arc sequence correspond to the given sequence of integers in the subclass.

Two different arc sequences are used on the tree – the Euler arc sequence and the complementary Euler (C-Euler) arc sequence – to determine a sequence of well-formed parentheses. The Euler sequence of parentheses is obtained as in the breadth-first traversal algorithm. The C-Euler sequence is obtained from the original Euler tour (without eliminating any arcs) and then assigning each forward (backward) arc a left (right) parenthesis. Each parenthesis (arc) string is then matched to define the NEXT function, which forms the linked list of integers. Finally, linked list ranking is used to sort the integers in the restricted sequence in $O(\log n)$ time, employing $\frac{n}{\log n}$ processors on the EREW PRAM.

4.3.3 Coloring of an Interval Graph

Given a collection of $n$ intervals $\mathcal{I} = \{I_i = [a_i, b_i] | a_i \leq b_i\}$, there exists an interval graph $G_\mathcal{I} = (V, E)$ with $n$ nodes such that the node set is $V = \{I_i \in \mathcal{I}\}$ and the edge set is $E = \{(I_i, I_j) | I_i \cap I_j \neq \emptyset\}$. The problem is to assign colors to the nodes of the interval graph such that the fewest number of colors are used and no two adjacent
nodes have the same color.

A sequential algorithm can be designed as follows for (minimum) coloring of an interval graph using a stack. First, the endpoints are sorted in increasing order. If a beginning endpoint $a_t$ is encountered, a color is popped off the stack and assigned to the corresponding node. On the other hand, if its endpoint $b_t$ is encountered, this indicates that all remaining nodes are not adjacent to it. So the associated color is pushed back onto the stack for reuse.

The algorithm given in [19] uses the PPM technique to replace the stack. After sorting the endpoints in ascending order, a left (or right) parentheses is assigned to each beginning (or ending) point, $a_t$ (or $b_t$). The levels of the parentheses are computed and sorted using the algorithm for sorting the restricted class of integers given in the previous section. Multiple linked lists are then built, one for each distinct level. The first element of each list is given a color, which is distributed to the others in that list using a linked list ranking method. The interval graph coloring problem is thus solved in $O(\log n)$ time using $n$ processors on the EREW PRAM model.

4.4 Application of Parentheses Matching to Trees

In order to apply PPM to trees and their related problems, let us first define a number of necessary terms and discuss the relationship between a given tree and an equivalent sequence of parentheses. Then several algorithms are discussed which prove the equivalence between the traditional representations of a tree and a parentheses string representation of a tree.

4.4.1 Representations and Traversals of Trees

The term tree is defined to mean a general tree unless otherwise specified. A
rooted general tree is represented by either

(i) leftmost-child\((v) = (u)\) and right-sibling\((v) = (w)\) relation, or

(ii) parent-of\((v) = (u)\) relation.

A rooted binary tree has an additional representation

(iii) right child\((v) = (u)\) and left-child\((v) = (w)\) relation.

It is shown in [22] that any of three representations can be converted to another in \(O(\frac{n}{p} + \log n)\) time using \(p\) processors on the EREW model. Therefore, any tree algorithm having \(\Omega(\frac{n}{p} + \log n)\) time complexity is independent of the input data structure on this model.

Several traversals are described here which are necessary to the understanding of the algorithms to follow. The preorder and postorder traversals of a general tree are defined analogous to those for binary trees. However, an inorder traversal is one in which a parent node is ‘visited’ between each of its children. A combination pre/post order traversal is one in which each parent node is visited exactly twice, once before its children are processed and once after all its children are processed. In addition, each leaf node is processed twice, consecutively. The pre/post node sequence obtained from the tree in figure 4.3 is

\[a \ b \ c \ c' \ b' \ d \ e \ h \ h' \ e' \ f \ f' \ g \ g' \ d' \ a'\]

with the second occurrence of each node distinguished. The inorder node sequence from the same tree is \(c \ b \ a \ h \ e \ d \ f \ d \ g\).

Another traversal commonly defined for a tree is the Euler tour, in which each arc is duplicated, but pointing in the opposite direction. That is, for each arc \(A\) leaving node \(a\) and entering node \(b\) in the tree, arc \(A'\) is added leaving \(b\) and entering \(a\). This addition of arcs forms an Euler circuit within the tree. By starting at the root of the tree and following each arc in sequence, the Euler sequence of the nodes is defined.
For the tree in figure 4.3, the Euler node sequence is

$$abcbadehedfdgd\ a.$$ 

The Euler sequence of arcs for the tree is

$$<a, b>\quad< b, c>\quad< c, b>\quad< b, a>\quad< a, d>\quad< d, e>\quad< e, h>\quad< h, e>\quad< e, d>\quad< d, f>\quad< f, d>\quad< d, g>\quad< g, d>\quad< d, a>.$$ 

A more general representation of an Euler tour is the combination of Euler sequence of nodes and Euler sequence of arcs, where each node is embedded between the arcs.

Chen, Das, and Akl [22] presented a unified tree traversal algorithm which is one method used to produce traversals described above. After computing the node sequence corresponding to an Euler tour of a tree, this algorithm defines a NEXT function which describes how to mark the nodes to be retained for the desired inorder, preorder, or postorder traversal from this sequence. A variation of this algorithm constructs the combination pre/post ordering of the nodes. To do this, both the preorder and postorder nodes are marked and the leaf nodes are duplicated. The unified algorithm requires $O\left(\frac{n}{p} + \log n\right)$ time using $p$ processors on the EREW model.
4.4.2 Tree Representations Related to Parentheses

In addition to the traditional traversals, other representations – a parentheses string and a parentheses string with embedded nodes – have been found to be very useful in the application of the PPM strategy. Depending upon the nature of the problem to be solved, there are two variations of such parentheses string representations of a tree: either associating a pair of parentheses to a node (node-associated parentheses string) or associating a parenthesis to an edge (edge-associated parentheses string). The primary difference between the two is that the node-associated parentheses string has an additional outermost pair of parentheses. By way of comparison to the Euler tour representations [102], the node-associated parentheses string is comparable to Euler sequence of nodes, and the edge-associated parentheses string is comparable to Euler sequence of arcs.

A more general form of parentheses string representation of a tree is a parentheses string with embedded nodes. This corresponds to associating a left (right) parenthesis to a forward (backward) arc in the Euler tour augmented with nodes. When a tree is given as input, the Euler tour can be constructed so, the node identifiers between two consecutive parentheses are immediately available. Figure 4.4 gives the parentheses representations of the tree from figure 4.3. Shown in (α) and (β) are the node-associated and edge-associated parentheses strings. In (γ) is shown the parentheses string with embedded nodes.
4.4.3 Euler Tour Technique vs. Parentheses Matching

Many tree related problems are easily solved through Euler tour techniques. The basic Euler tour technique consists of three phases: the construction of an Euler tour, the assignment of proper weights to a node or an arc, and parallel linked list ranking. When applying PPM to tree related problems in which the input is not given as a tree but as a string of balanced parentheses, the differences and similarities between these two techniques become clearer. A major difference between the two techniques is that the tree structure is explicit in the input of the tree related problems, but in the parentheses related problems the tree structure is underlying, that is, the parent-child relationships in the underlying tree are computed as needed using parentheses matching and possibly linked list ranking. In terms of computation costs for solving tree related problems, the Euler tour technique constructs the Euler tour first and then applies linked list ranking. On the other hand, parentheses matching technique uses parentheses matching and possibly linked list ranking, depending on the problem. However, the ability of both to solve many of the same problems indicates their similarity.

4.5 Parentheses Matching Applied to Basic Tree Operations

In addition to the applications discussed in section 4.3, Das, et al. [40] recently proposed several useful PPM algorithms dealing with the relationships between trees and parentheses. These include an algorithm which accepts as input a general tree and produces the inorder traversal of the tree. Another algorithm accepts as input a well-formed parentheses string and produces the corresponding tree, with the nodes numbered in preorder form. A third algorithm reconstructs a tree from the preorder and postorder traversals. Each of these algorithms utilizes PPM to accomplish the
given task in $O(\log n)$ time utilizing $O(\frac{n}{\log n})$ processors on the EREW PRAM model.

In the following section these algorithms are discussed as they serve to establish the direct relationship between parentheses matching and trees.

4.5.1 Tree Traversals via PPM

Depth first traversals (preorder, postorder, and inorder) and the corresponding ordering of the nodes is applicable to a wide variety of tree (and graph) related problems. Application of parentheses matching allows for the development of a relatively simple algorithm. (Preorder and postorder traversals can be solved similarly.)

The algorithm given in [40] first computes the Euler arc sequence of the tree and assigns a left (right) parentheses to each forward (backward) arc. After matching the parentheses, nodes are marked in inorder fashion. That is, each leaf node is determined by a left parenthesis which is immediately followed by a right parentheses. An inner node is determined by a right parenthesis immediately followed by a left parentheses. Finally, the occurrence of a node having a single child is determined when two consecutive right parentheses also have consecutive mates. Each of these nodes is marked by a 1. Performing the prefix sums operation on the list of ones numbers the nodes according to the inorder sequence. The algorithm performs in $O(\log n)$ time utilizing $O(\frac{n}{\log n})$ processors on the EREW PRAM model.

Figure 4.5 demonstrates Algorithm INORDER, showing the parentheses with embedded nodes for the tree given in Figure 4.3. Performing the prefix sums on the nodes marked with 1's completes the inorder numbering of the nodes, as shown in the third row of the table.

4.5.2 Converting Parentheses to Trees

An algorithm for converting a well-formed sequence of parentheses into the corre-
Parentheses with Embedded Nodes

<table>
<thead>
<tr>
<th>a ( b ( c ) b ) a ( d ( e ( h )</th>
<th>1 1 1 1 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>e ) d ( f ) d ( g ) d ) a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 5 6 7 8 9</td>
</tr>
</tbody>
</table>

Figure 4.5: Demonstration of Algorithm INORDER

sponding tree form in which each node has a pointer to its parent used with the nodes numbered in preorder form is also given in [40]. Because this algorithm is utilized in several application algorithms (in chapter 5) in order to convert the parenthesis string back into the desired tree, it is reproduced here. The idea is to select representative nodes from the dummy nodes embedded in the parentheses string according to the desired node numbering. Broadcasting the computed node numbers to the rest of the dummy nodes and checking the nodes surrounding each left parenthesis suffices to identify parent nodes. The algorithm constructs the tree in $O(\log n)$ time using $O(\frac{n}{\log n})$ processors on the EREW model. Constructing a tree where the nodes are numbered in postorder can be computed similarly.

Algorithm PARENTHESES-TO-TREE [40]

*Input:* a string of parentheses, $A[1..n]$.

*Output:* an ordered, rooted tree with nodes numbered in preorder, $PARENT[1..\frac{n}{2} + 1]$.

2. Identify representative nodes in preorder numbering.
For $1 \leq i \leq 2n$ pardo

if $A[i] = '('$ then $HEAD[i + 1] := 1$

$HEAD[1] := 1$

Compute the prefix-sums of $HEAD$, store in $NUM$.

3. Connect duplicate dummy nodes representing the same tree node.

For $1 \leq i \leq 2n$ pardo

if $A[i] = '('$ then $NEXT[i - 1] := MATE[i + 1]$

4. For $1 \leq i \leq 2n$ pardo

if $HEAD[i] = 1$ then broadcast $NUM[i]$ along $NEXT$ pointers, storing results in $NUM$

(i.e. $NUM[NEXT[i]] := NUM[i]$)

5. Determine parents.

For $1 \leq i \leq 2n$ pardo

if $A[i] = '('$ then $PARENT[i + 1] := NUM[i - 1]$

Example

The example in figure 4.6 demonstrates the algorithm for the tree in figure 4.3. The embedded parentheses string is given in row A. The values in $MATE$ are found using parallel parentheses matching. Each node identifier immediately following a left parenthesis is marked with a one in the $HEAD$ field, as is location one. Then the prefix sums are computed on the $MARK$ values and the results are stored in $NUM$. ($NUM$ also provides the mapping of node numbers to node identifiers as listed in A.) For every node immediately preceding a left parenthesis, its $NEXT$ value is set to be one greater than the $MATE$ value of the succeeding left parenthesis. This forms a linked list of the duplicate instances of each node identifier. For each node identifier
having HEAD = 1, the corresponding NUM value is broadcast along the linked list. This organizes the NUM values so that for each left parenthesis, the NUM value to its left is the parent of the NUM value on its right, thus allowing for the creation of the PARENT array.

4.5.3 Constructing a Tree from Traversals

Given the preorder and postorder node sequences of a general tree the problem is to reconstruct the unique binary tree. The algorithm given in [40] compares the relative ordering of nodes in the two traversals to determine parent-child relationships, linking two nodes if the order of two consecutive elements in the two traversals is reversed, (indicating a parent-child relationship). Linking the parent and child causes the creation of several disjoint linked lists, which, when interconnected form a linked list corresponding to the tree to which the parentheses are assigned. After performing
parentheses matching, the tree is generated using Algorithm PARENTHESES-TO-TREE. The tree is reconstructed from its traversals on the EREW model in $O(\log n)$ time using $\frac{n}{\log n}$ processors. A discussion of the correctness of the algorithm is found in [40].

Using this technique the parent is connected to leftmost child in the preorder sequence, the leftmost child is connected to the parent in the postorder traversal, identical leaves in both traversals are connected, and adjacent siblings are connected. The insertion of two parentheses, i.e., '(' or ')', between siblings, constructs the parentheses version of the Euler tour of the tree. Thus the algorithm constructs the unique tree corresponding to the given preorder and postorder traversals. However, the algorithm works correctly only for a general tree in the sense that when a node has a single child it is always considered the leftmost child.

4.5.4 Tree Contraction by Parentheses Matching

It appears that there may be a direct relationship between tree contraction and parentheses matching in that many tree problems solved using PPM can also be solved using the tree contraction technique. The tree contraction problem is the problem of systematically reducing a rooted tree to its root. Abrahamson et al. [1] give an elegant algorithm for this problem which is based on binarization [30] of the input tree and removing odd numbered leaves and their parents at odd iteration and even numbered leaves and their parents at even iteration. The conflicts at parent nodes are avoided by left-right alternation within each iteration. The key to this algorithm is converting a general tree of $n$ nodes into a binary tree of $n$ leaves and $n-1$ internal nodes, and removing the leaves and corresponding parent nodes by odd-even, left-right alternation. Thus, it reduces the number of leaves and internal nodes by half at each iteration, achieving $O(\log n)$ time and $O(n)$ work.
An algorithm that accomplishes tree contraction using the PPM strategy is given in [40]. This algorithm differs from that in [1] in that (i) it does not require an explicit tree data structure, (ii) the binarization is applied not to a tree but to a string of parentheses, and (iii) it uses only parentheses matching and parallel prefix. This algorithm is useful when the input has a natural representation in a linear array but has an underlying tree structure, as in the evaluation of an arithmetic expression. It also provides insight into the nature of the tree contraction problem and demonstrates the versatility of the proposed parallel parentheses matching strategy. Using the optimal parentheses matching and parallel prefix sums algorithms, Algorithm TCPM is work-optimal, requiring $O\left(\frac{n}{p} + \log n\right)$ time using $p$ processors on the EREW PRAM model. Because the tree contraction algorithm is such a widely used technique, the algorithm from [40] is reproduced here.

Algorithm TCPM [40]

**Input:** a rooted general tree, $T$ ($|V_T| = n$)

**Output:** contracted tree node, i.e., a single node.

1. Construct Euler tour from the tree.
2. Construct a string of parentheses by assigning a left parenthesis to a forward arc and a right parenthesis to a backward arc, stored in array $A[1..2n+1]$. (All even locations are for parentheses and odd locations for nodes.) {If the input is given in the form of a string of parentheses, skip steps 1 and 2.}
3. Match parentheses.
4. Binarize the parentheses.
   a) Compute $NEXT$ pointers, connecting the duplicate nodes in the string.
      
      If $A[i+1] = \ '('$ then $NEXT[i] \leftarrow Mate[i+1] + 1$
   b) Assign weights where new parentheses are to be inserted.
      
      {We assume that there is a dummy ‘)’ after the end of array $A$.}
If \( A[i - 1] = ')' \) and \( A[i + 1] = '(' \) or \( A[i - 1] = ')' \) and \( A[i + 1] = ')' \)
then \( WEIGHT[i] \leftarrow 2 \)
else \( WEIGHT[i] \leftarrow 0. \)

c) Compute prefix-sums of the weights along the Next pointers by using multiple linked list ranking, storing the results in \( WSUM[i] \).

d) Adjust weights to correctly reflect the displacement.
    Reinitialize \( WEIGHT[i] \) to 1 for all \( i \)
    if \( A[i - 1] = ')' \) and \( A[i + 1] = '(' \) then \( WEIGHT[i] \leftarrow 3 \)
    if \( A[i - 1] = ')' \) and \( A[i + 1] = ')' \) then \( WEIGHT[i] \leftarrow 3 \) and \( WEIGHT[i + 1] \leftarrow WSUM[i] + 1 \)

e) Compute prefix-sums of the weights along the consecutive indices of array \( A \), storing the results in \( LOC[i] \).

f) Copy the nodes and parentheses in array \( A \) into array \( A' \) according to the new location, \( LOC[i] \).

g) Insert new parentheses.
    if \( A'[i - 1] = ')' \) then \( A'[i + 1] \leftarrow '(' \)
    if \( i \) is odd and \( A'[i - 1] = '(' \) (empty) then \( A'[i + 1] \leftarrow ')' \)

h) Apply parentheses matching to the binarized parentheses.

5. Tree contraction by parentheses removal.

   a) Number the terminal parentheses pairs from left to right. (Assign number from 0)
      
      {The 0th terminal parentheses pair and the last one are not removed throughout step b) as in typical tree contraction.}

   b) for \( \lfloor \log_2(n - 2) \rfloor + 1 \) times do
      
      (1) Remove odd numbered left terminal parentheses pair.
(2) Remove nearest enclosing parentheses pair of the parentheses pair removed in step (1).

(3) Remove odd numbered right terminal parentheses pair.

(4) Remove nearest enclosing parentheses pair of the parentheses pair removed in step (3).

(5) Renumber the remaining terminal parentheses pairs by simply halving the terminal ordering number.

{When removing a pair of parentheses, we are updating the left and right parent pointer using the result of parentheses matching in step 4.b).}

c) Remove the last two remaining pairs of parentheses.

{The terminal parentheses pair number 0 and the last one are removed.}

Table 4.1 demonstrates Algorithm TCPM for the tree given in Figure 4.3. $A$ is the original parentheses string, $A'$ is the binarized parentheses, row x.ol (or x.or) shows the parentheses after the removal of odd-left (or odd-right) terminal parentheses pair and its nearest enclosing parentheses pair, and row 4 is the (empty) parentheses, equivalent to a single contracted node at the end.

<table>
<thead>
<tr>
<th>A</th>
<th>A'</th>
<th>1 ol</th>
<th>1.or</th>
<th>2 ol</th>
<th>2.or</th>
<th>3 ol</th>
<th>3.or</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
<td>{</td>
<td>{</td>
<td></td>
<td>{</td>
<td></td>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[</td>
<td>[</td>
<td>[</td>
<td></td>
<td>[</td>
<td></td>
<td>[</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>(</td>
<td>(</td>
<td></td>
<td>(</td>
<td></td>
<td>(</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.6 Conclusion

This chapter has presented the initial stages of the development of parentheses matching as a design strategy. An overview of the existing work which uses PPM and
several implementations of PPM have been given. This existing work in parentheses matching, though much of it has been conceived independently of the others, when brought together demonstrates the validity of the development of parentheses matching as a general strategy. The application of parentheses matching to trees in [21] and to other related structures such as interval graphs [33] and cographs [77] supports the validity of that work. The fact that parentheses matching has also been applied to a variety of seemingly unrelated problems such as sorting integers [21], approximate bin packing [5], and string and dictionary matching [4] serves to emphasize its promise for use in the future.

The establishment of the relationships between tree representations and parentheses strings and between Euler tours and parentheses matching is well-developed in [40]. The relationship between parentheses matching and tree contraction is also addressed [40]. This leads to the work presented in the following chapter which proposes several algorithms which apply parentheses matching to a variety of tree related problems, further supporting the establishment of PPM as a general algorithm design strategy.
5.1 Introduction

As was demonstrated in the previous chapter, parentheses string trees are very closely related. Algorithms have been developed which will convert the traditional tree representation into the equivalent parentheses string and vice-versa [40]. For this reason, the attempt to solve tree related problems by applying PPM seems to be a natural next step in the further development of the general PPM strategy.

The algorithms based on parallel parentheses matching in this chapter fall into two categories – bottom-up computations on trees and the application of balancing binary trees. Specifically, solutions are given for the problems of computing the heights of all nodes of a tree, the extreme value in a tree, and the lowest common ancestor. Also included is the related problem of computing the nearest enclosing parentheses in a well-formed string. Finally, two new algorithms are presented which globally balance an unbalanced binary tree.

5.2 Bottom-up Tree Computations

There are numerous problems for which the computed value of a given node is based on the values in its subtree and, in a sequential setting, are typically solved in a bottom-up fashion. Those considered here include computing the heights of all nodes in a tree, determining the extreme values (maximum/minimum) of all subtrees, and finding the lowest common ancestor for all pairs of nodes. In a parallel context, the
typical technique used to solve these problems is tree contraction [1]; therefore, the proposed PPM-based algorithms for these problems provide an alternate approach.

A subproblem encountered in some PPM solutions is the range maxima (minima) problem. Given a sequence of \( n \) integers stored in an array and given the indices, \( u \) and \( v \), of two arbitrary elements in the sequence, the range maxima (minima) problem is to determine the maximum (minimum) integer in the sequence which lies in the closed range \([u, v]\). The solution to this problem requires two stages. In the preprocessing stage, a complete binary tree is constructed having \( n \) leaves, one for each integer. Each internal node of the tree contains both the prefix and suffix maxima (minima) of the subarrays of data corresponding to the leaves of the subtree rooted at the given node. These subarrays are computed using a merging process. This stage requires \( O(\log n) \) time using \( \frac{n}{\log n} \) processors on the EREW PRAM model.

In the second stage, given any two indices \( u \) and \( v \) \((u < v)\), the range maximum (minimum) is determined by locating the least common ancestor of the corresponding leaf nodes. The integer corresponding to \( u \) in the suffix max array of the left child (of the least common ancestor) is selected. The value corresponding to \( v \) in the prefix max array of the right child is similarly selected. The maximum of the two selected values is the desired result. Because the tree is complete, the least common ancestor is located in \( O(1) \) time. Utilizing concurrent reads allows the minimum (maximum) to be found for \( p \) arbitrary ranges in \( O(1) \) time using \( p \) processors [61].

Example

A demonstration of the range maxima problem is given here with the corresponding tree shown in figure 5.1. The input data array is given as the leaf nodes of the tree. Let \( P \) and \( S \) represent prefix maxima and suffix maxima arrays, respectively, for the labeled nodes.
The maximum between the values 2 and 6 (locations 3 and 6), is the maximum of the third element of \( S(#2) \), 9, and the second element of \( P(#3) \), 6, which provides the result of 9.

![Figure 5.1: Demonstration of Range Maxima Problem](image)

5.2.1 Heights of all Nodes of a Tree

The *height* of a node in a tree is defined to be the length of the longest path from it to a leaf. Algorithm HEIGHT uses PPM to compute the height of each node in the tree in \( O(\log n) \) time using \( O(\frac{n}{\log n}) \) processors on the CREW PRAM model. Step 5 of Algorithm HEIGHT requires concurrent reads in order to compute the range maxima for all nodes in \( O(\log n) \) time. For comparison purposes, using the tree contraction technique, this problem can be solved in \( O(\log n) \) time using \( \frac{n}{\log n} \) processors on the EREW model. Table 5.1 demonstrates the algorithm for the tree given in figure 4.3.
Algorithm HEIGHT

Input: a rooted tree, T.
Output: heights for all the nodes in T.

1. Obtain the pre/post ordering of nodes in the tree.
2. Assign '(' to each first occurrence of a variable and ')' to the second.
3. Match the parentheses.
4. Compute the nesting level of each parenthesis (LEVEL).
5. For each left parenthesis (node) compute the maximum level (MAX) contained between it and its mate (MAX). (Computed using the solution to the range maxima (minima) problem.)
6. For each left parenthesis (representing a node), compute the following:
   
   \[ \text{HEIGHT}(\text{node}) \leftarrow \text{MAX}(\text{node}) - \text{LEVEL}(\text{node}) \]

Table 5.1: Demonstration of Heights of all Nodes

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>c'</th>
<th>b'</th>
<th>d</th>
<th>e</th>
<th>h</th>
<th>h'</th>
<th>e'</th>
<th>f</th>
<th>f'</th>
<th>g</th>
<th>g'</th>
<th>d'</th>
<th>a'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parenthesis:</td>
<td>( ( ( ) ) ) ( ( ( ) ) ) ( ( ) ) ) ) ) ) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index:</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>Mate:</td>
<td>16</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>15</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>12</td>
<td>11</td>
<td>14</td>
<td>13</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>LEVEL:</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>MAX:</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>HEIGHT:</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Algorithm HEIGHT can also be applied to solve the problem of determining if an arbitrary binary tree is an AVL tree, in which the difference in the height of the two children of the node is in the set \{-1,0,1\}. Another problem which can be similarly solved within the same complexity bounds but on the EREW model is that of the internal path length of a tree. The internal path length of a tree is computed through the use of the levels of left parentheses.
5.2.2 Extreme Values of the Subtrees in a Tree

The solution to the problem of computing the heights of all nodes in a tree is quite similar to that for the extreme (maximum or minimum) value problem in a tree. Given a rooted tree, the problem is to determine for an arbitrary node the maximum (minimum) value contained in its subtree for which it is the root. The first three steps are the same as in Algorithm HEIGHT. In the fourth and final step, the extreme value in the subtree rooted at node $u$ is found by applying the range maxima (minima) algorithm to find the desired extreme between $u$ and its matched right parentheses $u'$. The algorithm computes the extreme value for all nodes in the tree in $O(\log n)$ time, using $O(\frac{n}{\log n})$ processors on the CREW PRAM model.

Algorithm EXTREME

*Input:* a rooted tree, $T$.

*Output:* extreme values for all subtrees in $T$.

1. Compute the pre/post ordering of nodes of the tree.
2. Assign each first occurrence of a node with '(' and each second with ')'.
3. Match the parentheses.
4. For a given node $u$, the maximum (minimum) value in its subtree is the maximum value in the range $(u, u')$, where $u'$ is the mate of the left parenthesis associated with $u$.

{This value is computed using the range maxima algorithm}.

5.2.3 Nearest Enclosing Parentheses

Another algorithm which has proven itself useful in the application of PPM is that of determining the nearest enclosing pair of parentheses for one or more matched pairs in a sequence. Given a well-formed sequence of parentheses and an arbitrary matched
pair \((u, u')\) of parentheses, the nearest enclosing parentheses (NEP) problem is to determine the nearest parentheses pair which encloses \((u, u')\). This problem can be solved by a simple prefix-max operation. A more general version of the NEP problem, referred to as NEPA (nearest enclosing parentheses for an arbitrary pair), determines the nearest enclosing pair of parentheses for two arbitrary matched pairs, \((u, u')\) and \((v, v')\). Algorithm NEPA computes the solution in \(O(\log n)\) time using \(O\left(\frac{n}{\log n}\right)\) processors on the EREW PRAM model. The algorithm extends to any constant number of pairs of parentheses within the same time bounds.

**Algorithm NEPA** \((u, v)\)

*Input:* A well-formed sequence of parentheses and two left (or right) parentheses, \(u\) and \(v\).

*Output:* The nearest enclosing parentheses.

1. Determine the mate of each parenthesis using PPM.
2. For the given pairs \((u, u')\) and \((v, v')\) compute \(MIN\) as the minimum index of \(u\) and \(v\), and \(MAX\) as the maximum index of \(u'\) and \(v'\).
3. Broadcast \(MIN\) and \(MAX\) to all locations (less than \(MIN\)).
4. For each left parenthesis \(x\) pardo
   
   if \(INDEX(x) < MIN\) and \(INDEX(MATE(x)) > MAX\) then

   \(MARK(x) \leftarrow x\) (a candidate solution).
5. Compute prefix-max on \(MARK\). The prefix-max value at location \(MIN\) is the index of the left parenthesis of the nearest enclosing pair of parentheses.

**Example**

Table 5.2 demonstrates Algorithm NEPA for a parentheses string of length twenty. The '∗' indicates the pairs of parentheses for which the nearest enclosing pair is to be
found, i.e. (10,11) and (13,17). Thus, MIN := 10 and MAX := 17, which are broadcast
to L(1) through L(10). For L(1) and L(2), INDEX and MATE are within the MIN-
MAX range and set MARK := 1. All others assign MARK := 0. After computing the
prefix-max of the MARK values, array location PREFIX-MAX(MIN) provides the
left index of the nearest enclosing pair.

| Parenthesis: | ( | ( | ( | ) | ) | ( | ) | ) | | |
| INDEX: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| MATE: | 20 | 19 | 5 | 5 | 4 | 5 | 12 | 9 | 8 | 11 | 10 | 13 | 13 | 16 | 15 | 15 | 27 | 18 | 19 | 26 |
| MARK: | 1 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| PREFIX-MAX: | 1 | 2 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

5.2.4 Lowest Common Ancestor

Given a rooted tree, T, the lowest common ancestor, LCA(u,v), of two arbitrary
nodes u and v is their common ancestor which is farthest from the root. The solution
to this problem presented here utilizes both PPM and Algorithm NEPA. The
pre/post ordering of the nodes is computed and '(' and ')' are assigned to the first
and second occurrences of each node, respectively, and the parentheses are matched.
The LCA(u,v) is the node associated with the nearest enclosing parentheses of pairs
(u, u') and (v, v') (using Algorithm NEPA). The details are given in Algorithm LCA,
which determines the LCA for a single node pair in O(log n) time employing O(n/\log n)
processors on the EREW PRAM model.

Algorithm LCA(u,v)

*Input:* a rooted tree, T, and two arbitrary nodes, u and v.

*Output:* lowest common ancestor of u and v.

1. Compute a pre/post ordering of the nodes of the tree.
2. Assign '(' to the first occurrence and ')' to the second of each node identifier.
3. Determine the mate of each parenthesis.
4. Let \((u, u')\) and \((v, v')\) correspond to the parentheses pair associated with nodes \(u\) and \(v\). The lowest common ancestor of \(u\) and \(v\), denoted \(\text{LCA}(u, v)\), is the node associated with the nearest enclosing parentheses of pairs \((u, u')\) and \((v, v')\) using Algorithm NEPA.

An LCA algorithm due to Scheiber and Vishkin [98] preprocesses the input tree in \(O(\log n)\) time using \(\frac{n}{\log n}\) processors. It then allows a single query to be answered in \(O(1)\) time using a single processor. Using \(k\) processors, \(k\) parallel queries can be answered in \(O(1)\) time, but this requires the CREW model.

A work-optimal algorithm in [78] computes the LCA of all node pairs in \(O(\log n)\) time using \(O\left(\frac{n^2}{\log n}\right)\) processors on the EREW model. With this method the solutions of the preprocessing stage are stored in a table for fast retrieval. However, it does not use parentheses matching, instead the computation of the LCA for a given pair of nodes is based on the preordering of the nodes, which provides that for any node \(v\), the preorder numbers of nodes in its subtrees are consecutive. Each node contains the range of indices of its subtree. Thus, \(v\) is the LCA for all pairs \((v, w)\), where \(w\) is in the subtree of \(v\), which is easily determined by processing the range within \(v\). Similarly, each node \(v\) is the LCA for all nodes \((w, x)\) such that \(w\) is in the subtree of one child of \(v\) and \(x\) is in the subtree of a different subtree of \(v\). The key to performing this work in \(O(\log n)\) time is the balancing of the workload among the \(\frac{n^2}{\log n}\) processors based on the number of children of each node.

The LCA algorithm proposed here obtains the same performance as [78] by modifying the last step (application of NEPA). The matched pair of parentheses provide equivalent (though not identical) ranges indicating which nodes are in a given subtree. Replacing the last step with the technique used in [78] causes the computation of twice as many table entries (due to each node having an associated left and right
parentheses), but a query is answered by a simple read from the table.

5.3 Balancing Binary Trees

In this section, parentheses matching is applied to a useful and common problem, namely (height) balancing trees. It is often the case that a tree becomes unbalanced and it is desirable to rebalance the tree while preserving the relationships (for example, inorder traversal) in the original tree.

Balanced trees of various types have been proposed and studied in an effort to obtain an efficient data structure for the maintaining of ordered data. The goals are to be able to perform the desired operations such as insert, delete, update and search with the least time and difficulty possible. One major facet of this problem is to keep the tree to the minimum height possible through balancing. Two general approaches are commonly used, incremental balancing and global balancing. With incremental balancing, a tree is returned to a balanced state immediately upon each insertion or deletion, while global balancing is a technique in which a tree is balanced periodically (when it becomes too unbalanced to be acceptable).

5.3.1 Definitions and Notations

A binary search tree is one in which for each node in the tree, the data values in the left subtree are less than (or equal to) that in the node, while all nodes in the right subtree are greater. A complete tree (not necessarily binary) is one in which each level of the tree is filled from left to right before the next level is started. A full (binary) tree is a complete tree having \(2^n - 1\) nodes, implying that all levels contain the maximum number of nodes. A perfectly balanced tree is one in which for any given node, the number of nodes in its left and right subtrees differ by no more than
one. Figure 5.2 gives the configuration of a complete tree of twelve nodes and one of several possible configurations of a perfect tree of the same size.

![Figure 5.2: Complete and Perfect Trees of Twelve Nodes](imageurl)

5.3.2 Existing Work

A variety of balancing algorithms have been proposed in the literature. Adelson-Velskii and Landis [2] proposed the well-known incremental sequential balancing technique of AVL trees. Martin and Ness [83] proposed an $O(n)$ time sequential, recursive, global balancing algorithm in which $n$ is repeatedly divided by two to form a perfect tree while an inorder traversal inputs the data to the algorithm in sorted order. Stout and Warren [101] proposed the first sequential global rebalancing algorithm which is optimal in both space and time. The technique causes the unbalanced tree to be converted to a vine, which is a skewed binary tree in which all children are right children. The vine is then converted back to a perfectly balanced tree using a compress operation in $O(n)$ time and constant space.

Chang and Iyengar [17] explored the parallelism inherent in their sequential algorithm based on folding. The algorithm first performs an inorder traversal on the unbalanced tree to determine the order of the nodes and then recursively divides the sequence in half to create the perfectly balanced tree in $O(n)$ time. Two parallel variations are also discussed. In the first, the two subtrees are processed concurrently. In
the second, which requires a duplicate copy of the tree, the tree is sorted during the
traversal, generating pointers in order to compute the children of each node, again
recursively dividing the tree into halves. No specific model of computation is speci-
fied, and the total work of these algorithms is $O(np)$ where $n$ is the number of nodes
and $p$ is the number of processors.

Moitra and Iyengar [88] propose a work-optimal EREW PRAM algorithm which
converts an arbitrary unbalanced binary tree into a complete binary tree in $O(\log n)$
time using $O(\frac{n}{\log n})$ processors. Given the inorder traversal of a tree the algorithm
creates the corresponding complete tree in $O(\frac{n}{p})$ time using $p$ processors. The al-
gorithm uses a constant time computation, utilizing table look-up for some values,
based on the regular structure of the complete tree to determine the index of the left
and right child of each node. Dekel, Peng and Iyengar [42] propose a global balancing
algorithm for an $m$-way search tree which can be applied for $m = 2$ to obtain the
complete binary tree. The strategy is similar to that in [88] and attains $O(1)$ time
complexity utilizing $n$ processors on an MIMD shared memory model and is scal-
able. Venkatraman, Kime and Srinivas [105] propose two parallel algorithms based
on the vine structure in [101] which globally balances a tree. The technique converts
the unbalanced tree into the vine then ranks the elements using parallel linked list
ranking. Using the ranks the list is transformed into a complete binary tree using a
compress operation. This non-work-optimal algorithm attains $O(\log n)$ time utilizing
$O(n)$ processors on the EREW model. However, a revised scheduling technique for
the processors allows the algorithm to attain work optimality [41].

5.3.3 General Balancing Strategy

Two work-optimal, global tree balancing algorithms are presented in this section.
Each achieves $O(\log n)$ time utilizing $O(\frac{n}{\log n})$ processors on the EREW model. The
general strategy of the proposed algorithms is similar with variations in implementa-
tion. Each accepts as input an unbalanced binary tree. In the first phase, an inorder
traversal of the tree is performed to obtain a (sorted) listing of the nodes. This is
accomplished through an Euler tour or the unified tree traversal [22]. In the second
phase, the traversal is converted into a string of parentheses representing the balanced
tree, which is then converted to the actual tree. Although work-optimal algorithms
have been previously proposed [88, 105], these algorithms provide an alternative ap-
proach to the problem.

5.3.4 Perfect Trees via PPM

Algorithm UNBALANCED-TO-PERFECT-VIA-PPM uses a divide-and-conquer
approach to recursively determine the desired roots of subtrees and to insert the
appropriate parentheses to indicate each level of nesting. In each call, the index of
the middle node is computed, assigned as the root of a subtree, and the sequence
of nodes on either side enclosed within parentheses. Then two recursive calls are
simultaneously made for the left and right subtrees just enclosed in parentheses. As
each node is identified as a root or a leaf, its preorder ranking is stored. The procedure
continues to split the substrings in half, enclose them in parentheses, and compute
the preorder rankings until a node with only one child or a leaf node is found, at
which point the recursion ends.

Parameters $X$ and $Y$ are the indices of the first and last nodes to be processed by
the given recursive call. PRENEXT is initialized to 1 and on each call contains the
preorder rank of the next root identified. The $n$ nodes are stored in the odd locations
of array $A[0..2n]$. The even locations have fields $LP[i]$, $RP[i]$, and $TP[i]$ initialized
to 0's, which hold the number of left and right parentheses to be inserted at that
location and the total, respectively. Once the parentheses string corresponding to the
balanced tree is produced, an algorithm from [40] is used to convert the parentheses into the tree. Finally, the array PRE is used to map the node identifiers from the original inorder traversal into the tree.

Algorithm UNBALANCED-TO-PERFECT-VIA-PPM

**Input:** Unbalanced binary tree

**Output:** Balanced binary tree preserving the inorder relationships.

**Declarations:**

- \( T \): input tree
- \( \text{PRENEXT} \): preorder numbering parameter
- \( I(0...2n) \): inorder traversal array
- \( \text{IN}(0...2n) \): corresponding inorder numbering of nodes
- \( \text{PRE}(0...2n) \): corresponding preorder numbering of nodes
- \( E(0...2n) \): Parentheses string

1. Perform inorder traversal on input tree \( T \) and store traversal in array \( I \), in odd locations.
2. \( X := 1; \ Y := 2n - 1; \ \text{PRENEXT} := 1 \)
3. Call \( \text{INORDER-TO-PARENTHESES}(X,Y,\text{PRENEXT}) \)
4. Call \( \text{LOAD-BALANCE} \) \{to obtain modified partitioning of \( I \}\)
5. Each \( P_j, 1 \leq j \leq p \) pardo

   Write the parentheses and right parentheses from new partition to a new array \( E[0..2n] \) using the prefix sums to compute the appropriate locations.
6. Use Algorithm \( \text{PARENTHESES-TO-TREE} [40] \) to convert \( E \) to the tree. (See chapter 4.)

Use \( \text{PRE} \) to map node identifiers into tree.
Algorithm **INORDER-TO-PARENTHESES** \(X,Y,\text{PRENEXT}\)

*Input:* Inorder traversal of a binary tree in an array; first and last index.

*Output:* A parentheses string corresponding to a perfect tree.

*Declarations:*
- \(I(1\ldots2n):\) Array for inorder traversal
- \(LP(1\ldots2n):\) Counters for left parentheses
- \(RP(1\ldots2n):\) Counters for right parentheses

1. If \((X = Y)\) then \(\text{PRE}(X) := \text{PRENEXT}; \text{RETURN}\)
   
   If \((Y - X = 2)\) then Increment \(LP(X - 1)\) and \(RP(X + 1)\);
   
   \(\text{PRE}(X) := \text{PRENEXT} - 1\); \(\text{PRE}(Y) := \text{PRENEXT}; \text{RETURN}\)

2. \(MID := \left\lfloor \frac{X+Y}{2} \right\rfloor.\) If \(MID\) mod 2 = 0 then increment \(MID\)

3. Increment \(LP(X - 1), LP(MID + 1), RP(Y + 1), RP(MID - 1)\)

4. \(\text{PRE}(MID) := \text{PRENEXT}\)

5. In parallel,
   
   Call **INORDER-TO-PARENTHESES** \((X, MID - 2, \text{PRENEXT} + 1)\)
   
   Call **INORDER-TO-PARENTHESES** \((MID + 2, Y, IN(MID) + 1)\)

Algorithm **LOAD-BALANCE**

*Input:* Counters \(LP, RP,\) and \(I\) from Algorithm **INORDER-TO-PARENTHESES**

*Output:* PART: Balanced partitioning of the arrays

*Declarations:*
- \(I(1\ldots2n):\) Inorder traversal of nodes (stored in odd locations)
- \(LP(1\ldots2n):\) Counters for left parentheses
- \(RP(1\ldots2n):\) Counters for right parentheses
- \(TP(1\ldots2n):\) Total counter for parentheses and nodes
- \(MARK(1\ldots2n):\) Temporary used for partitioning
- \(COUNT(1..p):\) Temporary location for computations
PART(1..p): Final partition information

1. For all \( i, 0 \leq i \leq 2n \) pardo
   
   if \( i \) is odd then \( TP(i) := 0 \)
   
   if \( i \) is even then \( TP(i) := LP(i) + RP(i) \)

2. Perform parallel prefix sums on \( TP \)

3. Each \( P_i \) divide the first \( TP \) in its partition by \( \log n \) and save as \( COUNT(i) \).

4. Each \( P_i \) scan partition, assign \( MARK:=1 \) for first \( TP \) value
   
   \[ \geq COUNT(i) * \frac{n}{p} \]
   
   Increment \( COUNT(i) \)

   and continue through partition. \( MARK:=0 \) for all other entries.

5. Compact the \( MARK \)ed entries into a new array using the prefix sums,

   storing the array index containing the mark.

6. Each \( P_i \) read entry \( PART(i) \) and \( PART(i-1) \) to determine new partition.

As implemented, Algorithm UNBALANCED-TO-PERFECT-VIA-PPM creates an array of counters for the numbers of left and right parentheses embedded between each pair of nodes. Algorithm LOAD BALANCING partitions the counter array among the processors. Then after writing the parentheses and node identifiers into array \( E \), Algorithm PARENTHESES-TO-TREE, which converts a parentheses string to the corresponding tree, completes the algorithm.

Example

Figure 5.3 demonstrates an intuitive view of the embedding technique for a tree of size twelve while the actual implementation is demonstrated in figure 5.4. The inorder sequence is stored in an array in the odd locations with additional fields \( LP \) and \( RP \), the counters for left and right parentheses, respectively. Parameters \( X \) and
Y are the (odd) indices of the first and last nodes in the partition of each recursive call.

<table>
<thead>
<tr>
<th>Inorder</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>l</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call 1</td>
<td>(a, b, c, d, e, f, g, h, i, j, k, l)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call 2</td>
<td>((a, b, c, d, e, f, g, h, i, j, k, l), (a, b, c, d, e, f, g, h, i, j, k, l))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call 3</td>
<td>(((a, b, c, d, e, f, g, h, i, j, k, l), (a, b, c, d, e, f, g, h, i, j, k, l), (a, b, c, d, e, f, g, h, i, j, k, l))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.3: Balancing a Binary Search Tree

In the first call, node f is computed as the midpoint (root). Thus, LP(0) and RP(10) are incremented representing the left subtree, as well as LP(12) and RP(24) representing the right. Recursive calls are then made with parameters (1,9) and (13,23), and the process is repeated. Figure 5.4 shows the counters for the embedded parentheses after the recursion ends. The load balancing and parentheses to tree conversion routines are then invoked to complete the process.

<table>
<thead>
<tr>
<th>Inorder</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>LP</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>RP</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>TP</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>16</td>
<td>17</td>
<td>20</td>
</tr>
</tbody>
</table>

Figure 5.4: Embedded Parentheses for Balanced Binary Tree

Because the string is divided into halves on each call, the depth of the recursive calls is $O(\log n)$. One processor is assigned to each partition of a recursive call, which requires $O(1)$ time to insert each pair of parentheses. The total number of operations required to build the parentheses string is $O(n)$. Thus, applying Brent's Principle, the algorithm achieves $O(\log n)$ time complexity using $O(\frac{n}{\log n})$ processors.
5.3.5 Complete Trees via PPM

The strategy of this algorithm is to convert the unbalanced tree to a complete binary tree, making use of the regular structure inherent in the complete tree. After determining the inorder traversal of the unbalanced tree, the algorithm computes the pre/post order traversal of a complete tree of the desired size, assigns left and right parentheses to the first and second occurrences of each node, respectively, and then computes the mates.

The node identifiers from the unbalanced tree are embedded into the parentheses string in the appropriate locations, and then blanks are embedded between the remaining parentheses. The embedded string is converted to the corresponding tree using the parentheses to tree algorithm from [40]. The balanced tree is constructed in $O(\log n)$ time utilizing $O\left(\frac{n}{\log n}\right)$ processors on the EREW model.

Algorithm UNBALANCED-TO-COMPLETE-VIA-PPM

*Input*: Unbalanced binary tree.

*Output*: Complete binary tree in parent-of form.

*Declarations:*

- $T$: Input tree
- $I(1 \ldots n)$: Inorder traversal array
- $C(1 \ldots n)$: Complete tree array
- $PPT(1 \ldots 2n)$: Pre/Post order traversal of $C$
- $P(1 \ldots 2n)$: Parentheses corresponding to $PPT$
- $MAP(1 \ldots n)$: Array to map input into complete tree

1. Compute the inorder traversal of the input tree $T$.
2. Use the *unified traversal algorithm* to generate the pre/post order traversal
of complete tree C and store results in PPT.

3. Store ‘(’ and ‘)’ in P, for the first and second occurrences of each identifier, respectively. Delete first ‘(’ and last ‘)’.

4. Compute the mates of the parentheses using parallel parentheses matching.

5. Call Algorithm EMBED {to merge the parentheses, P, and nodes, I }.

6. Call Algorithm PARENTHESES-TO-TREE [40]
   {to convert parentheses string to tree }.

7. Scan array NUM to associate preorder number with node id.
   If E(i) = node id then PRE(NUM(i)) := node id.

Algorithm EMBED

Input:   I: Inorder sequence of nodes of tree of size n.
         P: Parentheses string generated from pre/post traversal of complete tree of n nodes.

Output: E: parentheses string with embedded node identifiers.

Declarations:

I(1...n): Inorder traversal array
P(1...2n): Parentheses string
PPT(1...2n): Pre/Post order traversal of T
E(1...n): Parentheses string with embedded nodes.
EXPAND(1...2n): Map parentheses into new expanded array.
MARK(1...2n): Flags for locating embedding locations;
initialilized to 0.

{Mark locations for embedding of identifiers. }

1. For all i, 1 ≤ i ≤ 2n, pardo
If \( P(i) = ( \) and \( P(i + 1) = ) \) then \( \text{MARK}(i) = 1 \)

If \( P(i) = ) \) and \( P(i + 1) = ( \) then \( \text{MARK}(i) = 1 \)

If \( n \) is even then \{ fix single child of node \}

\[
\text{if } P(i) = ) \text{ and } P(i + 1) = ) \\
\text{and } \text{MATE}(i) = \text{MATE}(i + 1) + 1 \\
\text{then } \text{MARK}(i) = 1
\]

\{ Compute new locations for parentheses and nodes. \}

2. \( \text{EXPAND}(i) = \text{MARK}(i) + 1 \). Perform parallel prefix sums on \( \text{EXPAND} \).

\{ Write parentheses into \( E \) leaving spaces for nodes. \}

3. for all \( i, 1 \leq i \leq n \) pardo

\[
\text{if } \text{MARK}(i) = 0 \text{ then } E(\text{EXPAND}(i)) = P(i) \\
\text{else } E(\text{EXPAND}(i) - 1) = P(i)
\]

\{ Compute index locations for nodes and write into \( E \). \}

4. Perform parallel prefix sums on \( \text{MARK} \)

for all \( i, 2 \leq i \leq 2n \) pardo

\[
\text{if } \text{MARK}(i) \neq \text{MARK}(i - 1) \text{ then } E(\text{EXPAND}(i)) = I(\text{MARK}(i))
\]

5. Repeat expansion process as above to embed a blank after each parenthesis which is not followed by a node identifier, and include a blank as the first item in the parentheses string.

Figure 5.5 demonstrates the algorithm for a traversal with twelve items, using the complete tree shown in figure 5.2. A value 1 in \( \text{MARK} \) indicates that a node identifier is to follow the corresponding parenthesis in \( E \). Thus, \( \text{MARK} \) and \( \text{EXPAND} \) are used to copy the parentheses into \( E \). Then, after performing the prefix sum on \( \text{MARK} \), the summed values together with \( \text{EXPAND} \) are used to map the node identifiers into \( E \). Then the tree is generated using \( E \).
Table 5.3: Tree Related Algorithms using PPM

<table>
<thead>
<tr>
<th>Problem</th>
<th>Model</th>
<th>Time Complexity</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inorder, preorder, postorder traversals of a general tree</td>
<td>EREW</td>
<td>$O(\log n)$</td>
<td>$n$</td>
</tr>
<tr>
<td>Tree construction from traversals</td>
<td>EREW</td>
<td>$O(\log n)$</td>
<td>$n$</td>
</tr>
<tr>
<td>Rooted ordered tree construction from a parentheses string</td>
<td>EREW</td>
<td>$O(\log n)$</td>
<td>$n$</td>
</tr>
<tr>
<td>Tree contraction</td>
<td>EREW</td>
<td>$O(\log n)$</td>
<td>$n$</td>
</tr>
<tr>
<td>Heights of all nodes in a tree</td>
<td>CREW</td>
<td>$O(\log n)$</td>
<td>$\frac{n}{\log n}$</td>
</tr>
<tr>
<td>Extreme values in subtrees (all)</td>
<td>CREW</td>
<td>$O(\log n)$</td>
<td>$\frac{n}{\log n}$</td>
</tr>
<tr>
<td>Nearest enclosing parentheses (two arbitrary matching pairs)</td>
<td>EREW</td>
<td>$O(\log n)$</td>
<td>$\frac{n}{\log n}$</td>
</tr>
<tr>
<td>Lowest common ancestors (one pair)</td>
<td>EREW</td>
<td>$O(\log n)$</td>
<td>$\frac{n}{\log n}$</td>
</tr>
<tr>
<td>(all pairs)</td>
<td>EREW</td>
<td>$O(\log n)$</td>
<td>$\frac{n}{\log n}$</td>
</tr>
<tr>
<td>Balancing Binary Trees</td>
<td>EREW</td>
<td>$O(\log n)$</td>
<td>$\frac{n}{\log n}$</td>
</tr>
</tbody>
</table>
5.4 Conclusion

The algorithms developed in this chapter demonstrate the applicability of PPM to a variety of tree related problems. This work complements that presented in chapter 4 and furthers the establishment of PPM as a viable strategy for the class of problems related to trees. Table 5.3 summarizes the tree related algorithms to which PPM has been applied. The variety of methods by which parentheses have been assigned to components of trees allows for a great deal of flexibility in the approaches that may be used in the future to solve problems related to trees and, perhaps, to graphs as well.
CHAPTER 6

PARENTHESIS MATCHING ON A HYPERCUBE

6.1 Introduction

Much research completed in the area of parallel algorithm design is related to the theoretical, shared memory PRAM model. This work must then be converted to some realizable parallel computer for actual application. In order to demonstrate that the proposed parentheses matching strategy is, in fact, a realistic strategy for existing parallel computers, this chapter presents two parentheses related algorithms for the well-known and widely used hypercube computer. Presented here are hypercube implementations for parentheses matching and the nearest enclosing parentheses problem.

The parentheses matching algorithm is motivated by a divide-and-conquer algorithm proposed in [34]. The significance of the work is the development of a load balancing strategy which allows the hypercube algorithm to achieve the same time complexity as the shared memory version and to use optimal $O(n)$ space.

6.2 Existing Work

As discussed in chapter 4, several algorithms have been proposed to solve the parentheses matching problem on the PRAM model of computation. Recently two parentheses matching algorithms have been proposed for the hypercube model. The algorithm proposed by Mayr and Werchner [84] is based on a divide-and-conquer routing scheme in which the original parentheses string is stored one parenthesis
per processor and is then divided into small subproblems through the use of various routing techniques. Eventually, each left parenthesis is in a processor adjacent to its right mate. Using a $d$-dimensional hypercube for a parentheses string of length $2^{d-1}$, the parentheses are matched in $O(d)$ time.

The second algorithm by JáJá and Ryu is based on their solution to the all nearest smaller values (ANSV) problem (defined in section 6.2.2). Application of the ANSV solution allows the parentheses matching algorithm to achieve linear speedup for a string of length $n$ in $O((\log^3 n)(\log \log n)^2)$ time using $p$ processors when $1 < p < n/((\log^3 n)(\log \log n)^2)$.

6.2.1 Parentheses Matching through Routing

Mayr and Werchner propose a parallel parentheses matching algorithm for the hypercube in [84] within the context of solving certain classes of routing problems, in particular, parentheses structured routings. The general strategy of the algorithm is a divide-and-conquer approach, similar to that used in Algorithm HYPERCUBE MATCH. The original parentheses string is divided into several local subproblems, which are solved directly, and into one larger global subproblem, which may be solved directly or divided further.

Given a $d$-dimensional hypercube, assign constants $a = 2^{\frac{3}{2}d}$ and $b = 2^{\frac{3}{2}d}$. Initially, the input string of parentheses is divided into intervals of size $a$ and matches within each interval are determined and removed. The remaining string is again partitioned so that each interval contains exclusively opening or closing parentheses. This ensures that there are no duplicate height (nesting level) values within a given partition. Additionally, the length of each partition is limited in that the height of all parentheses within a given interval is greater than or equal to $jb$ but less than $(j+1)b$ for some constant $j$. The string is again reduced by removing the partitions of length
less than \( b \) and those which contain a parenthesis whose mate is in a partition having length less than \( b \), leaving the remaining string partitioned into intervals of exactly \( b \) identical parentheses. This string is divided into subproblems \( u_i, 0 \leq i \leq b - 1, \) such that each \( u_i \) consists of exactly one parentheses from each interval, those whose height mod \( b \) equals \( i \).

Mayr and Werchner show that this process of constructing the independent, contiguous subproblems, \( u_i \), from a string of length less than or equal to \( 2^d \) is accomplished in \( O(d) \) steps on a \( d \)-dimensional hypercube and that the subproblems are of size \( O(a) \). This is accomplished through the use of parallel-prefix and segmented-parallel-prefix operations, a stable sort, and concentration and bit-permutation routings.

Using their proposed processor allocation method (as discussed in section 6.5), the conclusion is that each pair of parentheses in a string of length \( 2^{d-1} \) can be matched on a \( d \)-dimensional hypercube in \( O(d) \) steps. This corresponds to \( O(p \log p) \) total work where \( p = 2^n \).

Note that in order to solve this problem in logarithmic time, the number of parentheses in the hypercube is one-half the number of processors and that the first \( \frac{n}{2} \) processors initially store one parenthesis each. The authors also note that the algorithm suffers from relatively large complexity constants.

6.2.2 Parentheses Matching Using ANSV

The *all nearest smallest values problem (ANSV)* is defined as follows. Given an array \( A[1..n] \) containing values from a linearly ordered set, the problem is to compute for each \( A[i] \) the index \( j < i \) such that \( j \) is the largest index in \( A \) for which \( A[j] < A[i] \). \( A[j] \) is referred to as the left match for \( A[i] \) and is the nearest value to the left of \( A[i] \) that is less than \( A[i] \). The nearest value to the right is similarly defined.

The strategy presented in [63] for solving the ANSV problem is a divide-and
conquer approach in which the string of parentheses is partitioned equally among the processors. Each processor then uses an optimal sequential algorithm to determine the left and right matches (smaller values) for those items that can be computed locally. The smallest value in each partition is identified and used to form a reduced array \( A' \) for which the left and right matches are computed. Utilizing the results from the reduced array, a merging process is used to complete the computations for the full array. The result is that the ANSV problem is solved for an \( n \)-element array on a \( p \)-processor pipelined hypercube in \( O\left(\frac{n}{p}\right) \) time, utilizing \( p \) processors for \( 1 \leq p \leq n/(\log^3 n)(\log \log n)^2 \).

Parentheses matching is solved by applying the solution to the ANSV problem. Given a string of well-formed parentheses, the nesting level of each is computed. For each left parentheses with nesting level \( i \), its mate is the nearest parentheses to the right which also has nesting level \( i \). In other words, the mate of a given left parentheses with nesting level \( i \) is the parentheses associated with the nearest smaller value to the right, where 'smaller' includes 'equal to'. A string of \( n \) parentheses are therefore matched within the same time and complexity bounds as the ANSV problem.

6.2.3 Divide-and-Conquer Technique

Of particular interest here is an EREW PRAM, divide-and-conquer algorithm proposed in [34]. Although not optimal, this algorithm is very simple; therefore, a hypercube implementation of this parentheses matching algorithm is proposed. The overall time complexity of the hypercube algorithm is \( O\left(\log^2 p + \frac{n}{p} \log p\right) \) and attains the same performance as the EREW PRAM version, \( O(\log^2 n) \) time when \( p = O\left(\frac{n}{\log n}\right) \) processors.

The divide-and-conquer algorithm proposed in [34] is based on the two lemmas, which are reproduced here, and which are proved in the reference.
Lemma 1 [DCLP91]: The mate of a parenthesis at an odd position in a balanced input string lies at an even position (and vice versa).

Lemma 2 [DCLP91]: If a balanced string has no left parenthesis at an even position (or, equivalently, a right parenthesis at an odd position), then the mate of each left parenthesis in the string lies immediately to its right.

It is shown that any string that satisfies Lemma 2 is of the form ()()()...() and is referred to as form F.

The parentheses matching algorithm [34] uses the divide-and-conquer approach. Each parenthesis of the input string is first marked based on Lemma 1. Each left parenthesis at an odd position and each right parenthesis at an even position are marked with 0; all others are marked by 1. The marking separates the input string into two disjoint substrings, those marked by 0 and those marked by 1, each of which is copied into a new array. From Lemma 1, it is known that both parentheses of a mated pair are contained in the same substring. Each new array is then processed in the same manner. The processing terminates when each newly formed substring is in form F. The EREW PRAM algorithm is formally stated in Algorithm MATCH.

Algorithm MATCH [DCLP91]

STEP A: for $i = 1$ to $\log n - 1$ do

1. If a left parenthesis is at an odd position in its substring, then mark it by a 0, else by a 1. Similarly mark a right parenthesis at an even position in its substring by a 0, else by a 1.

2. Use the segmented parallel prefix algorithm to determine the new index of each parenthesis in its partition.

3. Move each parenthesis to its new position. endfor
STEP B:

1. Determine if the input string has been converted to form F. If not, then the input string is unbalanced and EXIT.
2. Match the parentheses and store the results in array MATCH.

In [34] it is shown that \( \log n - 1 \) iterations are sufficient for Step A to convert the input string to substrings of form F and that the total time required is \( O(\log^2 n) \) utilizing \( \frac{n}{\log n} \) processors.

6.2.4 Example

The example in figure 6.1 demonstrates the divide-and-conquer technique for an input string of sixteen parentheses. The original index of each parenthesis is shown as the subscript. Based on the value of MARK, the input string is divided into two subproblems as shown in (b). Each parenthesis is reindexed and a new MARK value is assigned. This causes the input string to be split into four subproblems, each of which happens to be in final form F. That is, for each left parenthesis in the input string, its match is in the location to its immediate right.

Note that the subproblems formed do not necessarily have the same number of entries. It is also the case that the subproblems will reach final form in different steps. Therefore, the hypercube implementation must address the balancing of data among the processors.

6.3 Hypercube Implementation

The hypercube is a fixed-connection multiprocessor computer consisting of \( p = 2^k \) processors, labeled 0..\( p \) and connected as a k-dimensional Boolean cube. Two
processors of the hypercube are directly connected if the binary representation of their labels differs in at most one digit. It is also assumed that the hypercube model has pipelining and all-port communication capability with respect to communication among the processors [96].

For this problem, define an entry of the input string to be a record containing two pieces of information, a designation as a left or right parentheses and its original index, which are constant throughout the execution of the algorithm.

For the hypercube implementation, arrays INPUT[1..n] and MATCH[1..n] are divided into partitions of size \( \frac{n}{p} \). Each processor in the cube is assigned a partition of each array. This is done in an ordered fashion so that \( P_0 \) is assigned partition 1, \( P_1 \) is assigned partition 2, etc.

As the algorithm progresses, each processor recognizes matching pairs of parentheses. A match is referred to as local if the match of the given parenthesis is determined by the processor in which the match information is to be stored. Otherwise the match is said to be non-local. Local match information is stored directly by the processor while the non-local match information is sent to the appropriate processor for the
storage. For a given matched pair of parentheses both may be local or non-local, or there may be one of each.

The action taken by the algorithm depends upon the number of processors in the cube. If there are 2 processors in the hypercube, the routine Algorithm 2-CUBE is used to solve the problem. If the hypercube contains 4 or more processors, the problem is repeatedly partitioned using the Algorithm 4-CUBE until each pair of processors contains an independent subproblem. Then the Algorithm 2-CUBE is used to complete the solution. Algorithm 2-CUBE and Algorithm 4-CUBE are discussed first, followed by Algorithm HYPERCUBE MATCH.

6.3.1 Algorithm 2-CUBE

When utilizing a two processor hypercube, \( P_0 \) and \( P_1 \) scan their assigned partitions of INPUT and mark each entry \( E \) with a 0 or 1 (as previously described). The entries are exchanged with all 0 entries being given to \( P_0 \) and 1 entries given to \( P_1 \). Each processor then completes the solution sequentially. The steps are formally described below.

1. Mark each parenthesis as follows: Left parentheses at odd positions and right parentheses at even positions are marked with 0; all others are marked with 1.
2. \( P_0 \) and \( P_1 \) exchange entries so that all parentheses marked 0 are placed in \( P_0 \) and all marked 1 are placed in \( P_1 \).
3. Each processor uses a stack-based algorithm to match its parentheses and store the local results in MATCH.
4. Send non-local match information to the appropriate processor.

6.3.2 Algorithm 4-CUBE

Algorithm 4-CUBE forms the basis for the parentheses matching algorithm. It
insures the distribution of the near equal-sized subproblems to each of the processors of the hypercube. The task of Algorithm 4-CUBE is twofold. In Phase 1, matches within the partition of a processor are determined sequentially, and the non-local match information is communicated to the appropriate processor. In Phase 2, the unmatched parentheses are marked and redistributed so that all those marked with 0 are contained in $P_0$ and $P_1$, in input order, with each processor containing half the elements. Similarly, $P_2$ and $P_3$ contain the parentheses marked with 1. Therefore, Algorithm 4-CUBE divides a sequence of parentheses into two distinct subproblems and balances these equally among the four processors. This algorithm is the routine on which the general parentheses matching algorithm is based. The detailed steps follow.

Phase 1: Sequential Processing

1. Each processor uses a sequential stack-based routine to match any pairs of parentheses contained within its partition. Store the local match information in array MATCH.
2. Send non-local match information to the appropriate processor.
3. Each processor counts the number of unmatched parentheses remaining. Perform the prefix sum on these numbers to reindex the unmatched parentheses.

Phase 2: Mark and Distribute

1. Mark each parenthesis entry as follows: Left parentheses at odd positions and right parentheses at even positions are marked with 0; all others are marked with 1.
2. $P_0$ and $P_2$ exchange elements so that $P_0$ receives all entries marked 0 and $P_2$ receives all entries marked 1; likewise, for $P_1$ and $P_3$. 

3. $P_0$ and $P_1$ exchange information regarding the number of entries contained in its node. Let $m_0$ be the total number of elements marked with 0; likewise for $P_2$ and $P_3$.

4. $P_0$ and $P_1$ exchange entries such that $P_0$ obtains the first $\frac{m_0}{2}$ elements and $P_1$ obtains the remaining $\frac{m_0}{2}$ elements; likewise, for $P_2$ and $P_3$. (Force $P_0$ and $P_2$ to contain an even number of elements.)

Figure 6.2 demonstrates the distribution of data by Algorithm 4-CUBE for $n = 64$.

6.3.3 Hypercube Parentheses Matching Algorithm

The general parentheses matching algorithm for a cube of size greater than or equal to four, is based on the 2-CUBE and 4-CUBE algorithms. The input is partitioned among the processors as described above. Initially, the subcubes of size 4 are defined naturally as follows: $P_0 \ldots P_3$ make up Subcube 1; $P_4 \ldots P_7$ make up Subcube 2, and so forth through $P_{p-1}$.

Algorithm HYPERCUBE MATCH

1. Each subcube of size 4 executes Algorithm 4-CUBE. This logically partitions the hypercube into 2 subcubes of $\frac{n}{2}$ processors, each containing a distinct subproblem. One subcube contains all entries marked 0; the other contains all entries marked 1.
2. Each newly formed subcube of size \( \frac{p}{2} \) performs the prefix sums on the number of entries contained in each processor to determine the array index of each entry within the subproblem.

3. Each subcube (from previous step) recursively repeats steps 1, 2 and 3 until each subcube is of size 2.

4. Execute 2-CUBE to complete the solution.

6.4 Time Complexity Analysis

Let us first consider a feature of the algorithm that guarantees that the sizes of the subproblems assigned to processors do not become unbalanced, as is often the case with divide-and-conquer problems. The example in figure 6.1 demonstrates that subproblems of differing sizes may result in the PRAM implementation of the parentheses matching algorithm.

Each processor begins with a sequence of well-balanced parentheses of length \( \frac{n}{p} \). The sequence is processed sequentially (in time \( O(\frac{n}{p}) \)) to find any matches contained within the partition. This leaves the remaining unmatched parentheses in one of three forms, a sequence of left parentheses only, a sequence of right parentheses only, or a sequence of right parentheses followed by a sequence of left parentheses [52]. Clearly, the marking of these three sequences always produces approximately one-half zeros and one-half ones. If one element in the sequence of like parentheses is marked 0 then its successor is marked 1 (and vice-versa). Figure 6.3 demonstrates this for a sequence of right and left parentheses. This property ensures that in the steps of the algorithm in which zero and one entries are distributed to different processors the number of parentheses per processor is \( O(\frac{n}{p}) \).

Note that Algorithms 2-CUBE and 4-CUBE are executed within the context of a larger cube. Therefore, the communication of non-local match information (Step 4 of
Figure 6.3: Marking of Subsequences of Parentheses

2-CUBE and Phase 1, Step 2 of 4-CUBE) is to processors within the entire cube.

For Algorithm 2-CUBE, the marking of parentheses in step 1 requires $O\left(\frac{n}{p}\right)$ time. The worst-case in step 2 occurs when each processor sends all its parentheses to the other processor, for a total of $O\left(2^{\frac{n}{p}}\right)$ communication steps. The worst-case scenario for step 3 occurs when all the $2^{\frac{n}{p}}$ entries are marked 0 and, thus, are in $P_0$, requiring $O\left(\frac{n}{p}\right)$ processing time. The worst case scenario of step 4 occurs when all $\frac{n}{p}$ matches are non-local and must be communicated to other processors. Utilizing the pipelining capability, the communication time required is $O\left(\frac{n}{p} + \log p\right)$. Therefore, the overall complexity for Algorithm 2-CUBE is $O\left(\frac{n}{p}\right)$ computation time and $O\left(\frac{n}{p} + \log p\right)$ communication time.

In Algorithm 4-CUBE, each processor contains $\frac{n}{p}$ parentheses for processing. In Phase 1 the sequential match requires $O\left(\frac{n}{p}\right)$ computation time. Step 2 requires at most $O\left(\frac{n}{p} + \log p\right)$ communication time. The prefix sum operation of step 3 requires constant computation and communication time for a 4-cube (since there are exactly four processor nodes).

Step 1 of phase 2 requires $O\left(\frac{n}{p}\right)$ computation time for each processor to mark the parentheses in its partition. The exchange of elements between each pair of processors in steps 2 and 4 also requires $O\left(\frac{n}{p}\right)$ time as the maximum number of elements per exchange is $O\left(\frac{n}{p}\right)$ and the transfers are between adjacent processors. Step 3 requires constant time as each processor simply sends the number of parentheses it contains to its adjacent neighbor. Therefore, the total computation and communication time requirements for the Algorithm 4-CUBE are $O\left(\frac{n}{p}\right)$ and $O\left(\frac{n}{p} + \log p\right)$, respectively.
Steps 1 and 2 of Algorithm HYPERCUBE MATCH iterate \( O(\log p) \) times in order to reduce the subcube size to 2. Step 1 is the call to Algorithm 4-CUBE requiring \( O\left(\frac{n}{p} + \log p\right) \) total time. Step 2, the prefix sums, is be performed in \( O(\log p) \). Therefore, the overall time complexity of the loop is \( O\left(\log^2 p + \frac{n}{p} \log p\right) \). Step 4, the call to Algorithm 2-CUBE, requires an additional \( O\left(\frac{n}{p} + \log p\right) \) time. Thus, the overall time complexity of the algorithm remains \( O\left(\log^2 p + \frac{n}{p} \log p\right) \). For \( p = \frac{n}{\log n} \) the time complexity expression simplifies to \( O(\log^2 n) \).

6.5 Possible Speedups

Mayr and Werchner [85] propose a technique for implementing divide-and-conquer algorithms on the hypercube model with very low overhead. This has the prospect of speeding up a divide-and-conquer algorithm at the cost of additional processors. In this section the technique is evaluated to determine its applicability to Algorithm HYPERCUBE MATCH.

The problems addressed by this technique are those inherent in many divide-and-conquer algorithms. The first is that the subproblems generated through the divide process are not proportional to the subcube size that is available for computing the solution. For example, a problem may be divided into two subproblems with each being assigned to one-half the processors of the cube. But suppose that the two problem sizes are actually one-fourth and three-fourths of the original problem. Clearly, the distribution of the problem is not proportional to the resources.

The second problem is that the subproblem intervals may not be aligned with the subcube intervals. This may occur when an attempt is made to distribute the subproblems proportionally to the processors or when subproblems are allocated to the nearest subcube. Suppose, for example, the original problem on an eight processor
hypercube is divided in such a way that there are two subproblems requiring two processors each and one subproblem requiring four processors. The correct alignment is to assign the larger subproblem a four processor subcube containing processors \( P_0 \) through \( P_3 \) or \( P_4 \) through \( P_7 \). However, the algorithm might cause the alignment to be \( P_2 \) through \( P_5 \), which is not generally acceptable.

There are several constraints on the use of this algorithm. At any level of recursion, the total size of the subproblems must not exceed the size of the original problem. The solution of the problem must be no larger than the problem itself. And each subproblem must be distributed to a subcube proportional to its size.

Given that the divide step and the conquer step of the problem to be solved are in \( \Omega(\log n) \), the overhead incurred through the application of the subcube allocation technique is only a constant factor.

Algorithm HYPERCUBE MATCH satisfies all the constraints of the proposed technique. However, the new technique offers no measurable improvement due to the fact that it (Algorithm HYPERCUBE MATCH) does not suffer from the problems inherent in many divide-and-conquer problems. Each divide step of Algorithm HYPERCUBE MATCH divides the problem into two subproblems of size one-half (or less) of the parent problem and the routing of each subproblem to smaller subcubes falls naturally into the cube in which the problem originated. Thus, the routing of subproblems through the cube is not necessary for the parentheses matching algorithm proposed in this dissertation.

6.6 Comparison of PPM Algorithms

Actually, these three parentheses matching algorithms for the hypercube provide a wide variation of approach and utilization of resources. With respect to the allocation
of parentheses to processors, the algorithms span both extremes. At one end of the spectrum, the ANSV solution utilizes a relatively small number of processors. For example, to remain within the specified processor bounds for a string of $2^{10}$ parentheses, only one processor is utilized. Similarly, for a string of length $2^{20}$, a maximum of eight processors is used. At the other extreme, the routing-based solution utilizes $2n$ processors for a parentheses string of length $n$ in order to attain the desired execution time. Algorithm HYPERCUBE MATCH is the most flexible of the three in that it attains the specified time bounds for $2 \leq p < n$. It is unclear at this point what effect the reduction of the number of processors will have on the time complexity of the routing-based algorithm.

In terms of speed, the routing-based algorithm [84] and Algorithm HYPERCUBE MATCH offer the same time complexity, $O(\log \log n)$; however, the routing-based algorithm has large complexity constants. The ANSV-based algorithm offers a significantly larger time complexity, $O((\log^3 n)(\log \log n)^2)$.

To provide an overall comparison of the algorithms, consider the total work performed by each. The total work performed by the ANSV solution is $O(n)$. Substitution of $p = \frac{n}{\log n}$ and $p = 2n$ into the expressions for Algorithm HYPERCUBE MATCH and the routing solution, respectively, indicates that the total work for each of these is $O(n \log n)$, slightly less than work-optimal. Therefore, in terms of complexity, the ANSV algorithm performs less total work than the other two. It is noted that a more realistic comparison would involve the computation of the complexity constants or actual run times.

6.7 Nearest Enclosing Parentheses Problem

The problem of computing the nearest enclosing parentheses for an arbitrary
pair (NEPA) is one application of parallel parenthese matching for which there is a straightforward hypercube implementation. It follows the same line of approach as the PRAM algorithm. (Given in chapter 5.) Given a well-formed string of parentheses, the problem is to determine the nearest enclosing pair of parentheses for two arbitrary matched pairs, $(u, u')$ and $(v, v')$.

**Algorithm HYPERCUBE NEPA**

1. Determine the mate of each parentheses using a hypercube implementation of the parallel parentheses matching algorithm.
2. The node containing $v$ sends the value to the node containing $u$, and the node containing $v'$ sends the value to the node containing $u'$. Compute $MIN$ as the minimum of $u$ and $v$ and $MAX$ as the maximum of $u'$ and $v'$.
3. Broadcast $MAX$ and $MIN$ to each processor.
4. For each left parentheses in location $x$, if $INDEX(x) < MIN$ and $INDEX(MATE(X)) > MAX$ then $MARK(x) := x$ (a candidate solution).
5. Compute the prefix-max on $MARK$. The prefix-max value at location $MIN$ is the index of the left parenthesis of the nearest enclosing pair of parentheses.

The complexity of the first step is dependent upon the algorithm selected, so let us first consider the remaining steps of the algorithm. The communication of the values of $v$ and $v'$ in step 2 to the desired processors require at most $O(\log p)$ time, while the computation of the maximum and minimum requires $O(1)$ computation time. The broadcast of step 3 requires $O(\log p)$ communication time. Step 4 requires only local computation (no communication) in that each processor scans the parentheses assigned to it and marks them appropriately. This requires $O\left(\frac{n}{p}\right)$ time. $O\left(\frac{n}{p} + \log p\right)$
time is required to perform the prefix-max in step 5. The total time required for steps 2 through 5 is $O\left(\frac{n}{p} + \log p\right)$ for $p \leq n$.

The complexity of step 1 varies by the selection of the parentheses matching algorithm. Algorithm HYPERCUBE MATCH achieves an overall complexity of $O(\log^2 n + \frac{n}{p} \log p)$ for any $p \leq n$. Its use increases the time complexity of the algorithm to that of step 1. The use of the ANSV-based algorithm has no detrimental effect on the overall complexity of the algorithm. The complexity of step 1 is then $O\left(\frac{n}{p}\right)$ with $1 \leq p \leq n/(\log^3 n)(\log \log n)^2)$. Thus, the complexity of the algorithm remains $O\left(\frac{n}{p} + \log p\right)$ with the same restriction placed on $p$. At this point, there seems to be no advantage of utilizing the routing-based algorithm for this problem.

6.8 Conclusion

The development of the load balancing strategy of Algorithm 4-CUBE allows the application of the shared memory divide-and-conquer algorithm to achieve efficient results and to do so in a simple manner. Although not work-optimal, the hypercube divide-and-conquer algorithm is quite straightforward and easy to implement. For $p = \frac{n}{\log n}$, it attains the same time complexity as the EREW PRAM version, $O(\log^2 n)$ while requiring $O(n)$ space. It provides an alternative to the approaches previously proposed and provides for a flexible use of resources. The use of the parentheses matching strategy to solve the NEPA problem demonstrates its applicability and indicates the promise of the use for the solution of other problems on the hypercube.

There is still the need to develop a parentheses matching algorithm for the hypercube which attains work-optimal $O(\log n)$ time complexity and to develop parentheses matching based applications algorithms for the hypercube mode.
CHAPTER 7

CONCLUSION

This dissertation presents a look at two parallel algorithm design strategies – linked list ranking and parentheses matching (PPM) – and their application to various problems. The study of linked list ranking demonstrates that it is a well-established strategy, used by a variety of algorithms, particularly those related to tree and graph problems. It also serves to demonstrate how strategies become established, leading to the proposal to establish parentheses matching as a general algorithm design strategy. In support of this proposal, algorithms and approaches which apply parentheses matching to a variety of tree problems have been developed.

This dissertation provides two new linked list ranking algorithms for the EREW PRAM model. The deterministic algorithm performs in $O(n \times \log(n))$ time utilizing $p$ processors, where $n \geq p \log p$. Although not work-optimal, the algorithm is simple when compared to previously proposed work-optimal algorithms and has the added advantage of having small constant factors in terms of space and time requirements. The randomized algorithm follows the same approach, but the use of randomization in one step decreases the time complexity. With high probability, it requires $O(n/p + \log p)$ time and, thus, achieves work-optimal $O(\log n)$ performance when $p = O(n/\log n)$ processors. Furthermore, it uses less space than the deterministic algorithm.

As part of the effort to establish parallel parentheses matching as a viable strategy for algorithm design, the relationships between parentheses matching and several other known techniques have been established. The direct relationship between parentheses matching and the Euler tour technique has been developed. In addi-
tion, the relationship between tree contraction and parentheses matching has been demonstrated through the development of an algorithm to accomplish tree contraction through parentheses matching [40].

In addition, several algorithms have been developed which apply parentheses matching to tree related computations. These include solutions to the problems of computing the heights of all nodes in a tree, computing the extreme (maximum or minimum) values in the subtrees of a tree, and determining the lowest common ancestor (LCA) of two nodes of a tree. The height of the nodes in a tree and the extreme values in the subtrees are found in $O(\log n)$ time using $O(n \log n)$ processors on the CREW PRAM model. Determining the LCA for a single pair of nodes in a tree is computed in $O(\log n)$ time using $O(n \log n)$ processors on the EREW PRAM model. A variation of the algorithm computes a table of the least common ancestors of all node-pairs in $O(\log n)$ time using $O(n^2 \log n)$. Retrieval from the table requires the CREW model for concurrent access. An algorithm for computing the nearest enclosing parentheses for an arbitrary pair of parentheses is also presented as one of the tools necessary for successful application of parentheses matching. It attains $O(\log n)$ time utilizing $O(n \log n)$ processors on the EREW PRAM model.

As a practical application of the PPM strategy, two algorithms have been developed for globally balancing binary trees. Each accepts as input the unbalanced binary tree and creates either the perfect or complete binary tree that maintains the inorder numbering. The balanced tree is created in work-optimal $O(\log n)$ time utilizing $O(n \log n)$ processors on the EREW PRAM model.

In order for the parentheses matching strategy to be extended to other models of computation, in particular the well-known hypercube model, a PPM algorithm has been developed for the hypercube. The development of a load balancing strategy for the problem allows the simple divide-and-conquer technique to run in $O(\log^2 p + \frac{n}{p})$. 
using \( p \) processors. Although, not work-optimal, it compares favorably with two other existing algorithms in terms of simplicity and flexibility with respect to the number of processors used. A hypercube implementation for the nearest enclosing parentheses problem has also been developed. It achieves \( O(\log n) \) time when using \( O\left(\frac{n}{\log n}\right) \) processors.

This work has shown that parallel parentheses matching is a viable technique for the solutions of numerous tree related problems. The recognition of the relationship between parentheses matching to Euler tour techniques and to tree contraction [40] opens the door to the application of this technique to many other problems to which these techniques have been previously applied. In particular, the application of PPM to special classes of graphs, such as interval graphs, cographs, permutation graphs, etc., appears very promising. Another possibility is the application of PPM to problems which are solved sequentially using stacks and queues, as this is also characteristic of several of the problems for which parentheses matching solutions have been provided. The relationship of PPM to these and other data structures is a possible area for future research.

It is believed that further study will provide solutions to other classes of problems as well and will allow for a more concrete characterization of the problems for which parallel parentheses matching is a viable strategy. In addition, it is likely that some of the algorithms presented here can be improved with respect to the time complexity or with transition to a weaker model.

Finally, the implementation of PPM (and NEPA) on the hypercube allows for the application of this technique to many of the same problems for which PRAM solutions have been provided.
BIBLIOGRAPHY


Center for Research in Parallel and Distributed Computing, Univ. of N. Texas, Denton, June 1993.


