CHAPTER 5

APPLICATION OF PPM TO TREE RELATED PROBLEMS

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 correspond-
ing 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.
\[ P(\#2) = (7,7,9) \text{ and } S(\#2) = (9,9,9,9) \]
\[ P(\#3) = (1,6,8,8) \text{ and } S(\#3) = (3,8,8,8) \]
\[ P(\#1) = (7,7,9,9,9,9,9) \text{ and } S(\#1) = (3,8,8,8,9,9,9,9) \]

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.

![Diagram](image.png)

Figure 5.1: Demonstration of Range Maxima Problem

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>
<thead>
<tr>
<th>Node:</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>3</td>
<td>3</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 \(O(\frac{n}{\log n})\) 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 ‘\(\ast\)’ 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.

Table 5.2: Demonstration of Algorithm NEPA

<table>
<thead>
<tr>
<th>Parenthesis:</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>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<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>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>MATE:</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</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>
<tr>
<td>PREFIX-MAX:</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

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(\(\frac{n}{\log n}\)) 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.

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 specified, 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 algorithm 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 scalable. 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 implementation. 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 approach 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
- PRENEXT: preorder numbering parameter
- I(0...2n): inorder traversal array
- IN(0...2n): corresponding inorder numbering of nodes
- 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. \( X := 1; Y := 2n - 1; \) PRENEXT := 1
3. Call INORDER-TO-PARENTHESES(X,Y,PRENEXT)
4. Call 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 PARENTHESES-TO-TREE [40] to convert E to the tree. (See chapter 4.)
   - Use 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}; 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}; RETURN

2. \(MID := \lceil \frac{X + Y}{2} \rceil\). 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\((X, MID - 2, PRENEXT + 1)\)
   
   Call INORDER-TO-PARENTHESES\((MID + 2, Y, \text{IN}(MID) + 1)\)

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
   
   $\geq COUNT(i) \times \frac{n}{p}$. Increment $COUNT(i)$
   
   and continue through partition. $MARK:=0$ for all other entries.

5. Compact the MARKed 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)))</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))))</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></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RP</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></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 \(O(\frac{n}{\log n})\) processors on the EREW model.

Algorithm UNBALANCED-TO-COMPLETE-VIA-PPM

*Input:* Unbalanced binary tree.

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

*Declarations:*

\[
\begin{align*}
T & \text{: Input tree} \\
I(1 \ldots n) & \text{: Inorder traversal array} \\
C(1 \ldots n) & \text{: Complete tree array} \\
PPT(1 \ldots 2n) & \text{: Pre/Post order traversal of } C \\
P(1 \ldots 2n) & \text{: Parentheses corresponding to } PPT \\
MAP(1 \ldots n) & \text{: Array to map input into complete tree}
\end{align*}
\]

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 PARENTHESIS-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 $P(i) = ( \text{ and } P(i + 1) = )$ then $\text{MARK}(i) = 1$

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

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

\begin{align*}
\text{if } P(i) = ) \text{ and } P(i + 1) = ) \\
\text{and MATE}(i) = \text{MATE}(i + 1) + 1 \\
\text{then } \text{MARK}(i) = 1
\end{align*}

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

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

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

3. for all $i$, $1 \leq i \leq n$ pardo

\begin{align*}
\text{if } \text{MARK}(i) = 0 \text{ then } E(\text{EXPAND}(i)) = P(i) \\
\text{else } E(\text{EXPAND}(i) - 1) = P(i)
\end{align*}

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

4. Perform parallel prefix sums on MARK

\begin{align*}
\text{for all } i, 2 \leq i \leq 2n \text{ pardo} \\
\text{if } \text{MARK}(i) \neq \text{MARK}(i - 1) \text{ then } E(\text{EXPAND}(i)) = \text{I(MARK}(i))
\end{align*}

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 MARK indicates that a node identifier is to follow the corresponding parenthesis in E. Thus, MARK and EXPAND are used to copy the parentheses into E. Then, after performing the prefix sum on MARK, the summed values together with EXPAND are used to map the node identifiers into E. Then the tree is generated using E.
Figure 5.5: Generation of Parentheses String

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>(\frac{n}{\log n})</td>
</tr>
<tr>
<td>Tree construction from traversals</td>
<td>EREW</td>
<td>O(log n)</td>
<td>(\frac{n}{\log n})</td>
</tr>
<tr>
<td>Rooted ordered tree construction from a parentheses string</td>
<td>EREW</td>
<td>O(log n)</td>
<td>(\frac{n}{\log n})</td>
</tr>
<tr>
<td>Tree contraction</td>
<td>EREW</td>
<td>O(log n)</td>
<td>(\frac{n}{\log 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) (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.