CHAPTER 6

PARENTHESESS 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 \(1 \leq p \leq n/((\log^3 n)(\log \log n)^2)\).

### 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 \(a = 2^{\lfloor \frac{3}{4}d \rfloor}\) and \(b = 2^{\lfloor \frac{1}{4}d \rfloor}\). 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 \( v_i \), \( 0 \leq i \leq b - 1 \), such that each \( v_i \) 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, \( v_i \), 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 \( p = 2n \).

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 \( n \) 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 $l$, its mate is the nearest parentheses to the right which also has nesting level $l$. In other words, the mate of a given left parentheses with nesting level $l$ 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 $O(\log^2 p + \frac{n}{p} \log p)$ and attains the same performance as the EREW PRAM version, $O(\log^2 n)$ time when $p = O(\frac{n}{\log n})$ 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
(a) Input String of Parentheses

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MARK</td>
<td>0 1 0 1 1 0 0 0 1 1 0 1 1 0 1 0</td>
</tr>
</tbody>
</table>

(b) Subproblem 1            Subproblem 2

<table>
<thead>
<tr>
<th>INDEX</th>
<th>1 2 3 4 5 6 7 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>(1) (3) (6) (7) (11) (14) (16)</td>
</tr>
<tr>
<td>MARK</td>
<td>0 1 1 1 1 1 0</td>
</tr>
<tr>
<td>INDEX</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>INPUT</td>
<td>(2) (4) (5) (9) (10) (12) (13) (15)</td>
</tr>
<tr>
<td>MARK</td>
<td>0 1 1 0 0 1 1 0</td>
</tr>
</tbody>
</table>

(c) Subproblem 1            Subproblem 2            Subproblem 3            Subproblem 4

<table>
<thead>
<tr>
<th>INDEX</th>
<th>1 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>(1) (18)</td>
</tr>
<tr>
<td>MARK</td>
<td>0 0</td>
</tr>
<tr>
<td>INDEX</td>
<td>1 2</td>
</tr>
<tr>
<td>INPUT</td>
<td>(3) (6) (7) (11) (14)</td>
</tr>
<tr>
<td>MARK</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>INDEX</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>INPUT</td>
<td>(2) (9) (10) (15)</td>
</tr>
<tr>
<td>MARK</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>INDEX</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>INPUT</td>
<td>(4) (5) (12) (13)</td>
</tr>
<tr>
<td>MARK</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

Figure 6.1: Divide-and-Conquer Parentheses Matching Technique

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 Redistribute**

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 $\frac{m}{2}$ elements and $P_1$ obtains the remaining $\frac{m}{2}$ 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 $\frac{p}{2}$ 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
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 \( O(\frac{n}{p} + \log p) \) total time. Step 2, the prefix sums, is be performed in \( O(\log p) \). Therefore, the overall time complexity of the loop is \( O(\log^2 p + \frac{n}{p} \log p) \). Step 4, the call to Algorithm 2-CUBE, requires an additional \( O(\frac{n}{p} + \log p) \) time. Thus, the overall time complexity of the algorithm remains \( O(\log^2 p + \frac{n}{p} \log p) \). 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 \( P_3 \), 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 though 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 parenthése 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 \(O\left(\frac{a}{p}\right)\) time. \(O\left(\frac{a}{p} + \log p\right)\)
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 $1 \leq p \leq n/((\log^2 n)(\log \log n)^2)$. 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.