THE UNIVERSITY OF
WARWICK

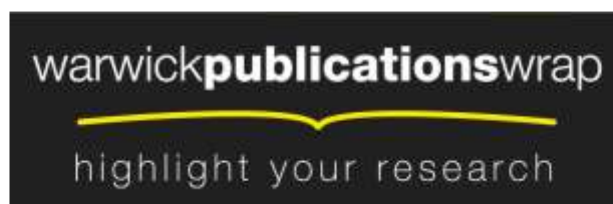
Original citation:
Gibbons, A. M. and Ziani, R. (1990) The balanced binary tree technique on mesh connected computers. University of Warwick. Department of Computer Science. (Department of Computer Science Research Report). (Unpublished) CS-RR-156

Permanent WRAP url:
http://wrap.warwick.ac.uk/60852

Copyright and reuse:
The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

A note on versions:
The version presented in WRAP is the published version or, version of record, and may be cited as it appears here.For more information, please contact the WRAP Team at: publications@warwick.ac.uk



http://wrap.warwick.ac.uk/

Research report 156

THE BALANCED BINARY TREE TECHNIQUE ON MESH CONNECTED COMPUTERS

Alan Gibbons and Ridha Ziani

(RR156)

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

May 1990

The Balanced Binary Tree Technique on Mesh Connected Computers

Alan Gibbons* and Ridha Ziani

Department of Computer Science, University of Warwick, England

Keywords: Efficient parallel algorithms, mesh-connected computer, balanced binary tree technique, bracket matching, expression tree construction, recognition of bracket and input-driven languages, prefix computation

§1 Introduction

For the well-known P-RAM model of computation, the class NC defines the class of efficiently solvable problems (see [3] for example). Such problems can be solved in polylogarithmic time using a polynomial number of processors. Within the constraints of current technology, it is not always possible to achieve such time-complexities because a lower bound for feasible parallel architectures (e.g., SIMD machines such as the mesh-connected computer) is naturally O(r), where r is the maximum path length over which messages have to be passed between co-operating processors. Here we define an **efficiently solvable problem** (of size n) to be a problem which is solvable in O(r) parallel-time using n processors. Such a solution is, of course, (within a constant factor) optimal. The mesh-connected computer as a model for parallel computation is well known [2,4,5,7]. Our presentation will be for two-dimensional mesh-connected computers (MCC^2s), where $r=O(\sqrt{n})$. Generalisation to arbitrary dimension is straightforward.

In [2], a general technique was described which often leads to efficient algorithms on the mesh for a wide class of problems. Here we show that another general technique, called the balanced binary tree technique (commonly employed in optimal P-RAM algorithms), may also be effectively employed on the mesh. It is not the case that established P-RAM algorithms employing this technique can always be directly implemented in efficient manner simply by using some embedding (explicit or implicit) of a tree in the mesh. One reason, for example, is that concurrent reads in the P-RAM model may correspond to routing congestion on the mesh. However, new and efficient algorithms avoiding such problems can often be devised which nevertheless use a related balanced binary tree approach. Such an example is described in §4.

The particular example of §4 is an optimum algorithm for bracket matching on the mesh. That is, given a string of n brackets, the ith bracket (for all i, $0 \leq i \leq n-1$) may learn the position (in the string), called match(i), of its matching bracket in $O(\sqrt{n})$ parallel-time on a $\sqrt{n} \times \sqrt{n}$ mesh. It follows that, given an arithmetic or algebraic expression presented as a string of symbols, the tree form of the expression can be constructed (by an easy extension to the bracket matching algorithm) with similar algorithmic efficiency (see[1,3]). This extends a number of previous results. It was shown in [2] that, if an expression is presented as an expression tree, then the expression can be evaluated in $O(\sqrt{n})$ parallel-time on a $\sqrt{n} \times \sqrt{n}$ mesh. For algebraic expressions such an evaluation requires that the corresponding algebra has a carrier of constant-bounded size. The recognition of bracket and input-driven languages can be reduced to the computation of such algebraic expressions (see [3] for example). It follows that if the input is in the form of a string (of the symbols making up the

*Partially supported by the ESPRIT II BRA Programme of the EC under contract #3075 (ALCOM)

expression) stored in an array, the following problems have efficient solutions on the mesh:

(a) Evaluation of arithmetic expressions

(b) Evaluation of algebraic expressions with a carrier of constant bounded size

(c) Parsing expressions of both bracket and input driven languages

As a by-product, a couple more (but comparatively trivial) problems, partial sums and sub-sequence ranking, are shown to have optimal solutions in §3. Indeed, the general technique is likely to yield optimal solutions for many more problems.

The balanced binary tree method (see [3] for many P-RAM examples) over a string of n characters $(c_0, c_2, \dots, c_{n-1})$ employs a balanced binary tree with the characters placed at the leaves. Figure 1 shows such a tree for n=16. Usually, without loss of generality, we can take n to be a power of 2. Typically, the same operation is performed at each non-leaf node so that a computation might consist of passing computed values up the tree with all operations at the same level being performed in parallel. Thus, for example, evaluation of $(c_0 © c_1 © \dots © c_{n-1})$ for constants $c_0, c_1, \dots, c_{n-1}$ and an arbitrary associative operator © trivially takes O(log n) parallel time on a P-RAM with n/2 processors (reducible to (n/log n) by standard methodology).

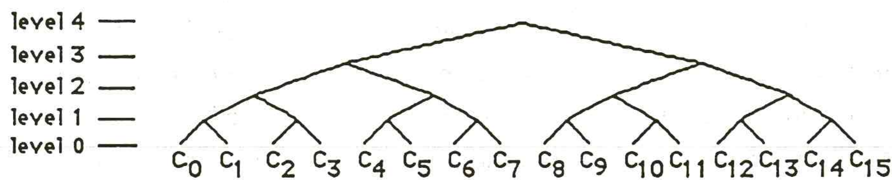
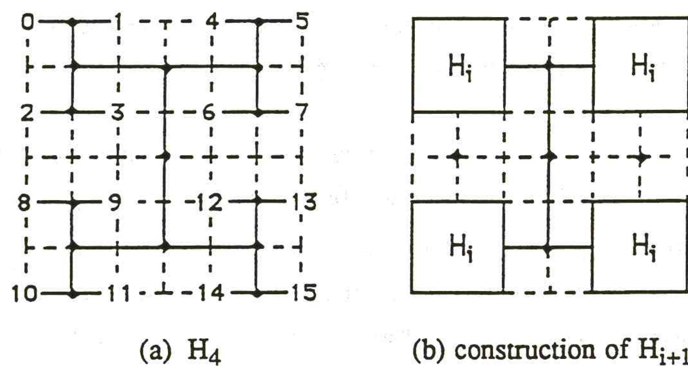


Figure 1

A standard way of embedding a balanced binary tree in the mesh is to use the so-called H-tree representation (see [6] for example). The ith H-tree, H_i, is an embedding of the balanced binary tree with 4^i leaves. Thus figure 2(a) shows H_4 (the embedding of the tree of figure 1, the leaves of H_4 are numbered according to the left to right order of figure 1). Figure 2(b) indicates the inductive construction of H_{i+1} from H_i. A drawback of this construction is that the balanced binary tree with n leaves requires a large mesh of $(2\sqrt{n} - 1) \times (2\sqrt{n} - 1)$ processing elements. However, it is possible to employ a strictly $(\sqrt{n} \times \sqrt{n})$ mesh for our purposes as described in §2.



(a) H_4 (b) construction of H_{i+1}

Figure 2

§2 Implicit Representation of the Balanced Binary Tree

The H-tree embedding of balanced binary trees in the mesh requires an un-satisfactory number of processing elements. Additional elements are used for the disjoint representation of all tree edges (that is, routing paths in the mesh). It is not necessary for all such edges to be disjointly

represented, in fact (for our algorithmic purposes) only those at the same height in the tree have to appear in disjoint regions of the mesh. We therefore adopt the approach of associating binary tree nodes with processing element indices (at this point we do not specify where each such indexed processing element is placed in the mesh) as indicated in figure 3. Two tree nodes (a leaf and an internal node) are associated with (or 'stored at') each processing element. In figure 3, vertical lines (either solid or dashed) connect the two tree nodes which are associated with the processing element whose index appears appears at the foot of each vertical line. Figures similar to figure 3 are inductively constructed as indicated in figure 4. This constuction, which associates tree nodes with processors, has the following properties:

(1) Consecutive tree nodes at level j which are both left (or right) sons are stored at processing elements whose indices differ by 2^{j+1}.
(2) At level j, the first non-leaf node which is a left son occurs at processing element index $(2^{j-1}-1)$ and the first non-leaf node which is a right son occurs at index $(3.2^{j-1}-1)$.
(3) Every processing element with an even (respectively, odd) index stores a leaf which is a left (right) son.

Thus, if every processing element knows its own index i and the number of leaves n, and if every processor executes the following code:

```
k←2, level←'none', typeofson←'none'
for j←1 step 1 until log n do
    begin
    k←2k
    if remainder((i+1)/k) = k/4 then begin level←j, typeofson←left end
    if remainder((i+1)/k) = 3k/4 then begin level←j, typeofson←right end
    end
```

then after O(log n)-time (and because of properties (1) and (2)), every processing element knows the type of non-leaf tree node (left or right) associated with it and at what level in the tree this node is. Additionally each processing element may determine whether it is associated with the root by employing the check j=log n? Similarly (employing property (3)), each processing element may easily determine in O(log n) time if the leaf associated with it is a left or a right son.

Consider now the distribution of processing elements across the two-dimensional mesh. We adopt the shuffled row major indexing scheme (see, for example [4]) which is important to establish the lemma stated at the end of this section. Figure 5(a) indicates where (the first sixteen indexed) processing elements are sited on the mesh. With this indexing scheme the binary tree construction has the property that son to father (and father to son) routing can be performed on the basis of local information only. Figure 5(b) indicates how each processing element (knowing the level=j and type of an associated tree node) knows the route to the next processing element (associated with the father node). For example, a processing element associated with a tree node which is a left son at level j (j>0 and odd) will find the processing element storing the father of this node at a distance of $2^{(j-1)/2}$ mesh steps to the right. Figure 6 shows (through (a) to (b)) routing from the leaves to the root for the balanced binary tree with 16 leaves. Notice that in general all tree nodes at the same level route in disjoint areas of the mesh and can therefore perform in parallel.

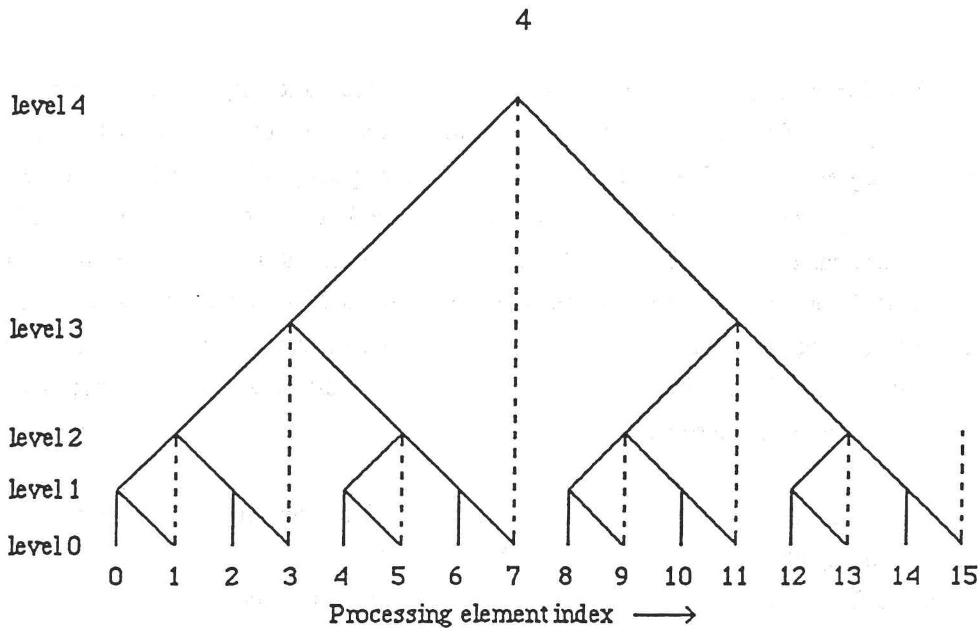


Figure 3

Lemma 1

For a complete binary tree with n leaves, the total time to route (in parallel) messages of constant bounded length from leaves to the root (or from root to leaves) of the tree can be achieved in $O(\sqrt{n})$-time on a square mesh of n processing elements.

Proof

It is sufficient to consider only routing from the leaves to the root. Figure 5(b) shows that the maximum length path (in terms of mesh steps) from a leaf to the root as traced on the mesh is that corresponding to repeated right-son to father routing. A section of such a path, in moving from an odd level j (>0) in the tree to the next odd level j+2, has length:

$$(2^{(j-1)/2} + 2^{(j-1)/2}) + (2^{(j+1)/2-1} + 2^{(j+1)/2}) = 5.\ 2^{(j-1)/2} \text{ mesh steps.}$$

In going from level 0 to level 1, such a path uses one mesh step. Thus, if the root (at level $\log_2 n$) is at an odd level, the path has overall length:

$$(1+ \Sigma_{\text{odd } j=1 \text{ to } (\log n)-2}\ 5.\ 2^{(j-1)/2}) = 5\sqrt{(n/2)} - 4 \text{ mesh steps}$$

On the other hand, if the root is at an even level, then the path has overall length:

$$(1 + (\Sigma_{\text{odd } j=1 \text{ to } (\log n)-3}\ 5.\ 2^{(j-1)/2}) + \sqrt{n}) = 7\sqrt{(n/4)} - 4 \text{ mesh steps}$$

where, on the left hand side of the equation, the term $\sqrt{n}$ is the length of the final section of the path from level (log n) – 1 to the root. □

We immediately have the following corollary.

Corollary 1

Any P-RAM algorithm based on the balanced binary tree and:

(a) whose parallel up/down activity on the binary tree is time-bounded by a constant number of leaf to root (or root to leaf) routings
(b) which performs all operations at nodes in constant bounded time
(c) which sends father to son (and son to father) messages of constant bounded length only

- will have an optimal implementation on the mesh. That is, a parallel computation time of $O(\sqrt{n})$ on n processing elements is possible. □

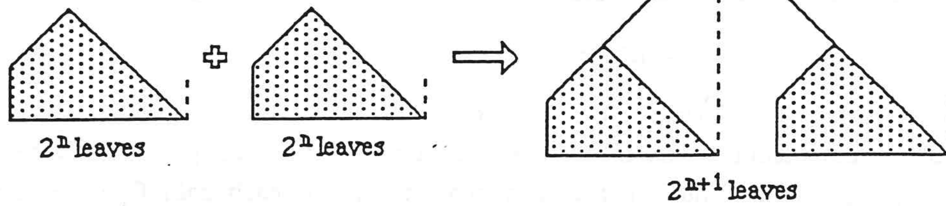


Figure 4

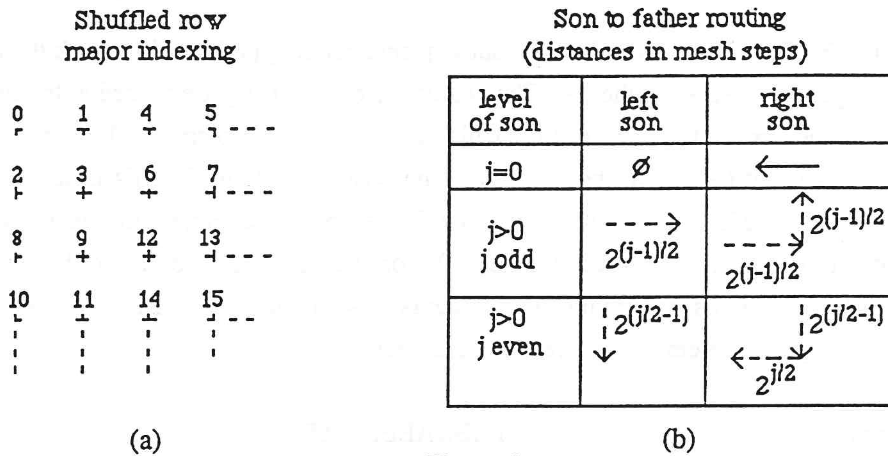


Figure 5

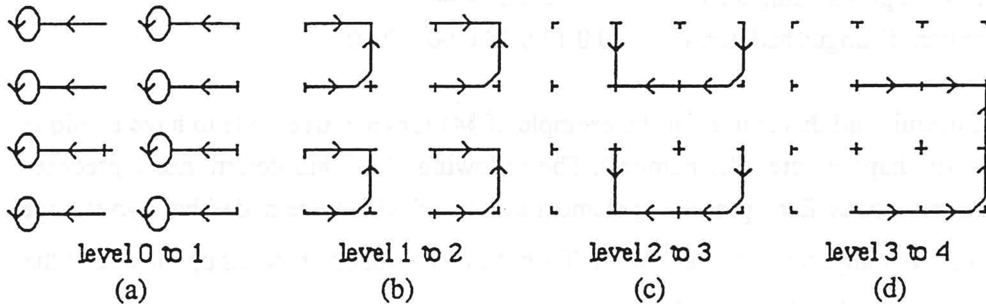


Figure 6

§3 Elementary examples

Here we describe two simple algorithms (partial sums computations and subsequence ranking) which satisfy corollary 1. These algorithms appear as sub-tasks in the example of §4. It will also be convenient at the end of this section to describe, for the purposes of §4, how the tree nodes of the balanced binary tree may be pre-order numbered by an efficient algorithm.

Given n values (v(0), v(1), ... , v(n-1)), a partial sums (or prefix) computation evaluates, for all i ($0 \le i \le n-1$), each of the sums $\Sigma_{j=0 \text{ to } i} v(j)$. Such a computation is a common sutask for many parallel computations (see [3] for many examples). The computation starts with (for all s, $0 \le s \le n-1$) v(s) being stored at the processing element associated with leaf s (the leaves being numbered from left to right). During the computation a processing element associated with tree node i employs three local storage locations A_i, B_i and C_i. At the outset of the computation, (for all s, $0 \le s \le n-1$) v(s) at the sth leaf is assigned to A_j. Then at successively higher levels in the tree, all non-leaf nodes (those at the same level in parallel) compute:

$$A_i \leftarrow A_{leftson(i)} + A_{rightson(i)}$$

$$B_i \leftarrow A_{rightson(i)}$$

After the root has computed these values it sets $C_{root} = A_{root}$ and then non-root nodes at successively lower levels in the tree (those at the same level in parallel) compute:

$$C_{i\ (\text{is a left son})} \leftarrow C_{father(i)} - B_{father(i)}$$

$$C_{i\ (\text{is a right son})} \leftarrow C_{father(i)}$$

An invariant of the computation is that, if tree node i is the root of a subtree spanning leaves r to s, then C_i equals $\Sigma_{j=1 \text{ to } s}$value(j). Thus, when the computation stops, for each leaf i, C_i is the partial sum $\Sigma_{j=1 \text{ to } i}$value(j).

Given a string (eg YABBABBAXABABA) the subsequence ranking problem is to rank the items in a sublist of distinguished items (eg the Bs). The characters of the string are placed at the leaves of the tree. Each leaf has a special storage location which is initially made to contain 1 if the associated character is distinguished (is a B in the example) otherwise it contains 0. This is schematically shown in (i) below. A prefix computation is then performed on these values (the result of such a computation for our example is shown in (ii) below). For each distinguished item, the associated storage location then contains its ranking (the contents of such storage locations can be locally zeroed for non-distinguished items as (iii) below illustrates)

String:	YABBABBAXABABA
(i) assign values	0 0 11 0 11 0 00 1 0 10
(ii) perform a prefix computation	0 0 12 2 34 4 44 5 5 66
(iii) zero non-distinguished items	0 0 12 0 34 0 00 5 0 60

Sometimes it is useful (and this is true for the example of §4) for each tree node to have a unique defining integer (perhaps its preorder number). The following algorithm determines a preorder numbering of the tree nodes. Each processing element associated with a tree node i has two storage locations A_i and B_i. Initially, for all leaves, $A_i \leftarrow 1$. Then the computation proceeds up the tree in the usual way with the non-leaf nodes computing:

$$A_i \leftarrow A_{leftson(i)} + A_{rightson(i)} + 1.$$

In this way, A_i for all tree nodes becomes equal to the number of nodes in the subtree rooted at tree node i. When A_{root} has been computed, the assignment $B_{root} \leftarrow 1$ is made and then the computation proceeds down the tree with non-root nodes computing:

$$B_{i\ (\text{a left son})} \leftarrow 1 + B_{father(i)}$$

$$B_{i\ (\text{a right son})} \leftarrow 1 + B_{father(i)} + A_i.$$

When the leaf nodes have finished their computation, B_i for all nodes is (as the reader will readily verify) the preorder number for that node.

§4 Bracket Matching

Given a sequence of n brackets [we make use of the example: (( ) ((( ))) (( )) ( ))], the bracket matching problem is to compute the function match[i] which, for all $0 \le i \le (n-1)$, is the position (in the string) of the bracket matching that at position i. For example, match[3]=8. The algorithm of Bar-On and Vishkin ([1] also described in [3]), a P-RAM algorithm to construct an expression tree from an expression presented as a string, is essentially an algorithm to compute the function match. As [1] indicates, knowing match, it is an easy extension to compute the tree (in constant time on a

P-RAM and (as the reader will easily find) in $O(\sqrt{n})$ time on the mesh). The algorithm described in [1] to compute the function match is not readily implemented on the mesh in efficient manner. This is because (c) of corollary 1 is not satisfied. Bar-On and Vishkin's algorithm consists (essentially) an upward phase in the tree followed by a downward phase in the course of which each bracket follows its unique path in the tree (of 'partial results' illustrated in figure 7) from the leaf at which it is stored to the leaf storing its matching bracket. In the course of the upward phase many paths coalesce thus violating (c) of corollary 1. In the following algorithm the upward phase of [1] is simulated by a downward (step 2), and the downward phase of [1] is replaced by a new technique (contained in steps 3 and 4). This new algorithm satisfies corollary 1.

Given a sequence S of brackets reduced[S] is the sequence obtained from S by repeatedly deleting adjacent pairs '( )' of brackets until no more deletions are possible, eg reduced[))(())()]=))(. In general any irreducible sequence is of the form $)^i(^j$, thus a pair of ordered integers are sufficent to represent any reduced form. Given any two reduced sequences $S_1=)^i(^j$ and $S_2=)^k(^l$ it is possible to compute reduced[S_1S_2] in constant time:

$$\text{reduced}[\ )^i(^j\ )^k(^l] \leftarrow \textbf{if } k\geq j \textbf{ then })^{i+k-j}(^l \textbf{ else })^i(^{l+j-k}$$

In order to compute the function match, we employ the balanced binary tree with the brackets placed at the leaves of the balanced binary tree. After execution of step 1, A_i will store the reduced form of the sequence of brackets which are stored at the leaves of the subtree rooted at i. Moreover, B_i^R (respectively B_i^L) will store the reduced form of the sequence of brackets stored at the the leaves of the tree rooted at the left (right) son of i. The superscript here refers to the direction in which the contents of the location B_i^R (or B_i^L) will be passed down the tree in step 2 of the algorithm and so is contrary to a seemingly natural superscripting at this point. Initially, for every processor associated with a leaf of the tree, A_i is set equal to the type of bracket stored at that leaf.

Step 1

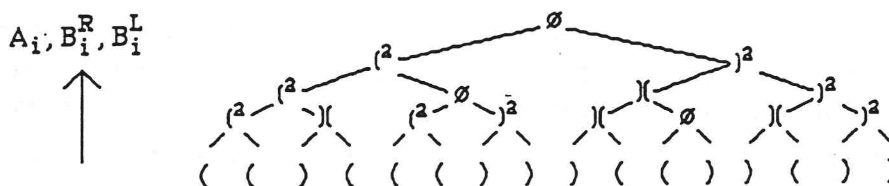
Starts with the input sequence of brackets at the leaves of the balanced binary tree (for each leaf, A_i is the bracket stored at that leaf) and in an upward phase computes, for all nonleaf nodes i:

$$B_i^R \leftarrow A_{\text{leftson}(i)}$$

$$B_i^L \leftarrow A_{\text{rightson}(i)}$$

$$A_i \leftarrow \text{reduced}[\ B_i^R\ B_i^L\ ]$$

Figure 7 illustrates this step for the example bracket string.



(only A_i shown)

Figure 7

Step 2

In this step, each non-leaf node i (in parallel) passes down the tree the value of B_i^R (respectively, B_i^L) to every node of the tree rooted at the right (left) son of i. When each of these values passes through a node it is copied to both sons of that node. Thus, each leaf receives a stream of B values (the first from its father, the next from its grandfather and so on). On receiving the current B value, a constant-time computation is performed at each leaf. Because all internal nodes (not just those at the same level) send values down the tree in parallel, the computation-time is bound by the time taken to send a message from the root to the leaves.

On completion of this step of the computation, the ith leaf (for all i, $0 \le i \le (n-1)$) will know the pre-order number of the least common ancestor of the leaves with indexes i and match(i). The least common ancestor of two leaves is that ancestor with the lowest level number. During the computation each non-root tree node i stores a triple T_i, initialised as follows:

$T_i(1) \leftarrow$ prefix number of tree node i
$T_i(2) \leftarrow$ **if** i is a left son **then** $B_{father(i)}^L$ **else** $B_{father(i)}^R$
$T_i(3) \leftarrow$ **if** i is a left son **then** L **else** R

In addition, if i is a leaf then there is a second triple L_i, initialised as follows:

$L_i(1) \leftarrow 0$
$L_i(2) \leftarrow \emptyset$
$L_i(3) \leftarrow$ **if** i stores a left bracket **then** L **else** R

The stream of B values that each leaf receives is in fact transmitted by sending down the tree the triples T_i. The additional information stored in these triples is employed to guide the computation that takes place at the leaves. If i is a leaf, then the arrival of each new T_i induces computation:

```
if Li(1)=0 then
   if Li(3)=Ti(3) then begin
                    if Li(3)=L then begin
                                   Li(2) ← reduced[Li(2)Ti(2)]
                                   if Li(2) begins with ')' then Li(1)← Ti(1)
                                   end
                              else begin
                                   Li(2) ← reduced[Ti(2)Li(2)]
                                   if Li(2) ends with '(' then Li(1)← Ti(1)
                                   end
                    end
```

After all leaves have performed this computation for the last time, $L_i(1)$ stores the prefix number of the least common ancestor of leaf i and the leaf which stores the bracket matching the one at leaf i. For our sample input the result of applying this step is indicated in figure 8.

In order to understand how this step works, consider the case that a left bracket is stored at a particular leaf. For this leaf $L_i(3)$=L. Each time a value $T_i(2)$ arrives at the leaf, $L_i(2)$ which is initially the empty sequence, is updated as follows: $L_i(2) \leftarrow$ reduced$[L_i(2)T_i(2)]$. When $L_i(2)$ begins with a right bracket, the least common ancestor is given by $L_i(1)$. The computation works (for left brackets at leaves) because of the following invariant. Suppose that i is the root of a subtree spanning leaves p to q and that the leaf is at position r (between p and q), then after the assignment:

$$L_i(2) \leftarrow \text{reduced}[L_i(2)T_i(2)]$$

$L_i(2)$ is the reduced form of the brackets stored from positions (r+1) to q.

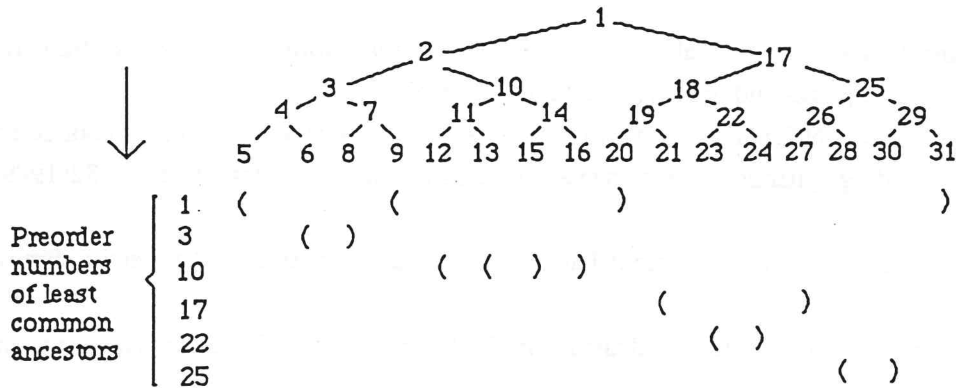


Figure 8

Step 3

Following step 2, the subsequence of brackets all having the same least common ancestor form a string of left brackets followed by the string of their matching right brackets:

$$(, (, \dots ,(,),), \dots ,)$$

For a given least common ancestor, we rank (by the previously described algorithm for subsequence ranking) the brackets so obtaining:

$$(_1, (_2, \dots , (_k,)_{k+1},)_{k+2}, \dots ,)_{2k}.$$

where subscripts denote the ranks of the brackets. However, we wish to induce the following subscripting:

$$(_1, (_2, \dots , (_k,)_k,)_{k-1}, \dots ,)_1$$

This is easily achieved by causing each processor storing a right bracket with rank r (and knowing k which is passed down the tree in the ranking computation) to compute the new subscript (2k-r+1) in constant time. Now such a subscripting has to be obtained for all possible least common ancestors. Every non-leaf tree node is such a possible ancestor. The computations for all non-leaf nodes at the same level can be performed in parallel and if the subtrees rooted at a certain level have k leaves, all computations for this level will take $O(\sqrt{k})$ parallel-time. Summing over all levels gives a computation time for this step of $O(\sqrt{n}+\sqrt{n/2}+\sqrt{n/4}+ \dots)=O(\sqrt{n})$.

Step 4

At this stage every bracket stored at a leaf knows:

(a) The least common ancestor (A) shared with its matching bracket.

(b) Its subscript (S) from step 3.

(c) Whether it is a left (L) or a right (R) bracket. Denote L or R by B.

This step then simply sorts all brackets according to the triple (A, S, B) associated with each bracket. (Permutation sorting on a two-dimensional can be done in $O(\sqrt{n})$ time by several extant algorithms, see [7] for example). Let L<R for sorting purposes. If '(' ends up at processing element i, then its matching bracket will be at processing element (i+1) ... and vice versa.

If in the sorting phase each bracket carries with it its initial address, then matching brackets can exchange addresses and then return to their original positions by re-sorting on their own addresses.

End of the bracket matching algorithm

References

[1] I Bar-On and U Vishkin, Optimal parallel generation of a tree form, ACM Transactions on Programming Languages and Systems 7, 2 (1985), 348-57.

[2] A M Gibbons and Y N Srikant. A class of problems efficiently solvable on mesh-connected computers including dynamic expression evaluation, Information Processing Letters 32(1989) 305-311.

[3] A M Gibbons and W Rytter. Efficient Parallel Algorithms. Cambridge University Press, Cambridge (1988)

[4] D Nassimi and S Sahni, Data broadcasting in SIMD computers, IEEE Transactions on Computing 30 (1981).

[5] D Nassimi and S Sahni, Finding connected components and connected ones on a mesh-connected computer, SIAM J. Comp.(1980) 744-757

[6] J Ullman, Computational Aspects of VLSI, Computer Science Press (1984)

[7] C Thompson and H Kung, Sorting on a mesh-connected parallel computer, Communications of the ACM (1987) 263-71.