Simple Parallel Algorithms for Dynamic Range Products

Christos Zaroliagis[1,2(✉)]

[1] Department of Computer Engineering and Informatics, University of Patras, 26504 Patras, Greece
[2] Computer Technology Institute and Press "Diophantus", N. Kazantzaki Str., Patras University Campus, 26504 Patras, Greece
zaro@ceid.upatras.gr

Abstract. We consider here the problem of answering range product queries on an n-node unrooted tree labelled with elements of a semigroup provided with an associative operator only. We present simple parallel dynamic algorithms for one of the weakest models of parallel computation (EREW PRAM). Our main result is an algorithm which answers a query in $O(\alpha(n))$ time using a single processor after $O(\log n)$-time and $O(n)$-work preprocessing, where $\alpha(n)$ is the inverse of Ackermann's function. The data structures set up during preprocessing are updated in $O(\log n)$ time and $O(n^\beta)$ work, for any (arbitrarily small) constant $0 < \beta < 1$, after a dynamic change in the label of a tree node.

1 Introduction

Developing algorithms for solving fundamental problems in parallel computing, except for being an important challenge by itself, flags the beginning of my collaboration with Paul Spirakis. Paul firmly believes that studying and solving fundamental problems, as well as possessing a solid theoretical background, is the key to the solution of virtually any problem. This belief has been the true motivation for most of my work, including the current one.

1.1 The Problem

We consider the following fundamental problem. Suppose we are given a semigroup $(S, \circ)$, i.e., a set S of elements with an associative operator $\circ$ on them, which is the only available one. Assume also that the product $x \circ y$ between any two elements $x, y \in S$ can be computed in $O(1)$ time. Let $s_1, s_2, ..., s_n$ be elements of S. We want to preprocess the sequence $s_1, s_2, ..., s_n$, such that subsequently *range queries* can be efficiently answered. A range query specifies two indices i, j, where $1 \leq i \leq j \leq n$, and asks for the product $s_i \circ s_{i+1} \circ \cdots \circ s_{j-1} \circ s_j$. This problem is known as the *linear range product* problem.

Similarly, the *tree range product* problem is defined as follows. Let T be an n-node unrooted tree, where every node of T is labelled (or associated) with an element of S. We want to efficiently preprocess T such that, given any two nodes

C. Zaroliagis et al. (Eds.): Spirakis Festschrift, LNCS 9295, pp. 396–407, 2015.
DOI: 10.1007/978-3-319-24024-4_23

u, v of T, the product of the labels associated with the nodes lying on the path from u to v is computed as fast as possible. (Labels can also be associated with edges of T rather than nodes. But it is easy to see that this is a special case of the problem considered here with labels on nodes.) It is clear that the linear range product problem is a special case of the tree range product problem, where the tree is simply a path of n nodes.

In this paper, we investigate the parallel complexity of the dynamic version of the tree range product problem. In this setting, the labels of the nodes may change. After a change in a node label, we would like to update the data structures already computed during preprocessing as efficiently as possible, without recomputing everything from scratch and without sacrificing the query time.

1.2 Applications

The tree range product problem, as well as its dynamic version, has applications to all problems that can be expressed as products of labels along paths in a tree. Along with its special case (the linear range product problem), they appear to be fundamental in many theoretical applications [2] (e.g., addition in unbounded-fan-in circuits, range minima, merging two sorted sequences of elements) and also to applications of a particular practical importance. We mention below some of them.

Network Communication. Consider a network connecting various sites using a spanning tree topology. Assume also that each link of the network has a specified capacity. Each time two sites want to communicate, they have to know the maximum size of a message that can be sent. This maximum message size is equal to the minimum capacity along the tree path connecting the two sites. Moreover, the knowledge of maximum message size is even more important when a communication link between two sites is replaced by another one with a different capacity. (This may happen because of a link failure, or because at a certain time some different link is used.) This problem reduces to the dynamic range product problem on a tree, whose edges are labelled with elements of a semigroup $(S, \circ)$ and $S, \circ$ are the set of reals and the minimum operator, respectively.

Matrix-Chain Multiplication Problem. Given a chain of n matrices $A_1, A_2, ..., A_n$, where matrix A_i, $1 \leq i \leq n$, has dimension $p_{i-1} \times p_i$, we want to fully parenthesize the product $A_1 A_2 ... A_n$ such that the total number of scalar multiplications is minimized. This problem is solved by dynamic programming (see [6], Chap. 15), where the problem reduces to determining the minimum cost of a parenthesization of $A_i A_{i+1} ... A_j$, for any pair $1 \leq i < j \leq n$. If matrices A_k and A_{k+1}, $1 \leq k \leq n$, are also free to change their common dimension p_k, then clearly the dynamic matrix-chain multiplication problem reduces to the dynamic linear range product one.

Information Retrieval in Databases. Usually queries in a relational database correspond to a selection of tuples (satisfying a particular condition) from an existing relation, or from a relation generated by some relational algebra operators most of which are associative [7]. In both cases, the selection of tuples

corresponds to an appropriate linear range product query made to the relation. Since values of fields in tuples change dynamically as data are updated, the same database query may result in a different selection of tuples when it is performed after such an update. Hence, an efficient solution of the dynamic linear range product problem appears to be fundamental here. Note also that the problem of generating a new relation by applying associative algebra operators on existing relations, reduces to the above mentioned matrix-chain multiplication problem, and is of particular importance in database query optimization [7,11].

1.3 Our Contribution and Related Work

In sequential computation, an efficient solution to the static tree range product problem was given independently in [1,5]. Their algorithms perform an $O(n)$ time and space preprocessing of the tree T and then any range product query is answered in $O(\alpha(n))$ time, where $\alpha(n)$ is the inverse of Ackermann's function which is a very slowly growing function with n. The algorithm presented in [1] can be also optimally parallelized on the EREW PRAM model of computation [1,8]. It performs an $O(\log n)$-time and $O(n)$-work preprocessing of T, such that afterwards a range query is answered in $O(\alpha(n))$ time using a single processor.

In this paper, we present simple and efficient parallel algorithms, on the EREW PRAM model, for the dynamic tree range product problem. Our algorithms answer a range product query in $O(\alpha(n))$ time, after an $O(\log n)$-time and $O(n)$-work preprocessing of T. The data structures set up during preprocessing can be updated, after the modification in the value of some node label, in $O(\log n)$ time and $O(n^\beta)$ work, for any constant $0 < \beta < 1$. Furthermore, we give a trade-off between query time and the work required for an update. With $O(n^\beta)$ update work, the query time is $O(\alpha(n))$; decreasing the update work towards $O(\log n)$, degrades the query time to logarithmic. It is worth mentioning here that our parallel algorithms imply a sequential dynamic algorithm for the tree range product problem with $O(n)$ time and space preprocessing, $O(\alpha(n))$ query time and $O(n^\beta)$ update time.

Our method is based on the following result: given an n-node binary tree T, partition T into $\Theta(n/m)$ node-disjoint connected components, $1 \leq m \leq n$, such that each component is of size at most m and is connected to the rest of the tree with at most 3 edges. In [4], a very simple algorithm is given that achieves such a partition in $O(\log n)$ time using $O(n)$ work on an EREW PRAM.

The main idea of our approach is the following. Based on the above tree partitioning result, we partition the tree into a small number of subtrees with disjoint node sets such that each subtree is connected with the rest of the tree with at most 3 edges. Then, we construct another tree, called the *condensed* tree, by shrinking each subtree into a tree of $O(1)$ size. The sizes of the subtrees are chosen so that the subtrees and the condensed tree have size $O(\sqrt{n})$. We then construct data structures for answering range product queries on each subtree and on the condensed tree. This enables us to answer range product queries on the initial tree. Since the node sets are disjoint, a change in a node label affects the data structure of only one subtree. Then, we update the data structures

of this subtree and of the condensed tree, both of which are smaller than the original tree. Applying this idea recursively yields the trade-off between update work and query time.

We note that our approach is a generalization of a method presented in [3,4] for solving dynamic shortest path problems on digraphs of small treewidth. However, the approach in [3,4] introduces large constants and a different analysis is needed in order to reduce them substantially and thus being able to provide the (above mentioned) update vs. query trade-off.

2 Tree Partitioning

For the sake of completeness, we present in this section the tree-partitioning result in [4], which will be used for the solution of the dynamic tree range product problem.

Definition 1. *A* (c, d, m)-partition *of an* n*-node binary tree* T*, where* $1 \leq m \leq n$ *and* c, d *are positive integer constants, is a node-partition of* T *into at least* n/m *and at most* dn/m *connected components such that each component has at most* m *nodes and is connected to the rest of the tree through at most* c *edges, called the* outgoing edges *of the component.*

We give an algorithm which is a variant of the well-known parallel tree contraction algorithm (see e.g., [9]). Assign a weight of 1 to each node in the tree. By adding a leaf (with weight 0) as a child to each node that has one child, we obtain a tree in which each node is a leaf or has two children. Number the leaves of the tree from left to right using the Euler tour technique [9]. From now on assume that we have a tree with weights on the nodes adding up to n, in which each internal node has two children, and in which some of the leaves are numbered from left to right. Our algorithm for obtaining the desired partition performs a number of *rounds*. Each round (consisting of three steps) forms groups of nodes which, at the end, will give the components. The algorithm is as follows:

ALGORITHM Binary-Tree-Partition

Repeat the following steps (round) $\log n$ times.

1. In parallel, for each odd numbered leaf that is a left child, if the sum of the weights of the leaf, its parent and its sibling is at most m, then shrink the edges connecting the leaf and its sibling to their parent. Assign the parent a weight equal to the sum of the weights of the three nodes. If the sibling is a leaf, it is even numbered. Assign this number to the parent (which is now a leaf in the modified tree). If the sum of the weights exceeds m, then delete the numbers (if they exist) from the leaf and the sibling.
2. Repeat step 1 for each odd numbered leaf that is a right child.
3. After these two steps, all the numbered leaves in the tree have an even number. Divide each of these numbers by 2.

END.

In order to implement the above algorithm – as well as the subsequent ones – on an EREW PRAM, we make the following conventions for the *input-output representation*. We assume that algorithm Binary-Tree-Partition has its input tree specified as a linked structure in n contiguous memory cells. The algorithm produces its output in $O(n)$ contiguous memory cells, divided into contiguous blocks, each block containing one of the connected components in the same linked format, and one final block containing the compressed tree (i.e., the tree at the end of the shrinking process) in a linked format.

The aforementioned input-output representation can be easily accomplished using standard EREW PRAM methods [9] using $O(\log n)$ time and $O(n)$ work, which we now describe briefly. Let q be the number of nodes in the compressed tree. By assigning the preorder number to each node in the compressed tree, we can assign a unique number between 1 and q to each connected subtree. Then, by solving a prefix summation problem on q elements, where the i-th element is the number of nodes in subtree i, we can allocate contiguous memory blocks for the various subtrees. It remains to copy the subtrees into the appropriate blocks. Since each node in the compressed tree knows the memory addresses allocated for its subtree, by reversing the shrinking process we can assign a unique memory address in the appropriate block to each node in a subtree. Now it is a simple matter for each node to copy itself into this address, and duplicate its link structure.

Theorem 1. *Given any $1 \leq m \leq n$, a $(3, 8, m)$-partition of an n-node binary tree T can be computed in $O(\log n)$ time using $O(n)$ work on an EREW PRAM.*

Proof. It is not hard to see that after the i-th iteration, at most $l/2^i$ leaves have numbers, where l is the initial number of leaves. Thus, at the end, there are no numbered leaves. Throughout, the following invariant is maintained: if a leaf does not have a number, then the weights of the leaf, its parent and sibling add up to more than m. (Note that such a leaf will not participate in any subsequent iteration.) Call such a triple of leaf, parent and sibling an *overweight* group.

Each non-numbered leaf is contained in some overweight group, and no node can belong to more than two overweight groups. Thus, the sum of the weights of all the overweight groups is at most $2n$, hence the number of overweight groups is at most $2n/m$. Since each overweight group contains at most two non-numbered nodes, the total number of non-numbered leaves at the end is $4n/m$. Since each internal node has two children, the total number of nodes remaining in the tree is at most $8n/m$.

Each node v in the remaining tree is associated with the connected subtree induced by the nodes that were shrunk into v in the above process. These are the required groups. It is easy to see that v has a weight equal to the number of nodes in the associated subtree. Since this weight is at most m, there are at least n/m such connected subtrees. Also, as shown above, there are no more than $8n/m$ connected subtrees. It follows from the construction that each subtree is connected to the rest of the tree through at most 3 edges which are incident on at most 2 nodes of the subtree.

The complexity bounds follow by the aforementioned input-output representation of the input tree and the fact that the algorithm is a variant of the parallel tree contraction algorithm [9]. □

3 Dynamic Range Products

In this section, we shall give our algorithms and data structures for the dynamic version of the tree range product problem. Our solution is based on the tree-partitioning result presented in the previous section.

For a function f let $f^{(1)}(n) = f(n)$; $f^{(i)}(n) = f(f^{(i-1)}(n))$, $i > 1$. Define $I_0(n) = \lceil \frac{n}{2} \rceil$ and $I_k(n) = \min\{j \mid I_{k-1}^{(j)}(n) \leq 1\}$, $k \geq 1$. The functions $I_k(n)$ decrease rapidly as k increases, in particular, $I_1(n) = \lceil \log n \rceil$ and $I_2(n) = \log^* n$. Define $\alpha(n) = \min\{j \mid I_j(n) \leq 1\}$.

Recall the assumptions made for the semigroup $(S, \circ)$ in Sect. 1. As in [1], we also assume that S has a unit element. (Otherwise, we can simply add such an element to S.) The following has been proved in [1,8].

Theorem 2. *Let T be an n-node unrooted tree such that each node is labelled with an element from a semigroup $(S, \circ)$. Then, the following hold on an EREW PRAM: (i) for each $k \geq 1$, after $O(\log n)$-time and $O(nI_k(n))$-work preprocessing, the product of labels along any path in the tree can be computed in $O(k)$ time using a single processor; and (ii) after $O(\log n)$-time and $O(n)$-work preprocessing, the product of labels along any path in the tree can be computed in $O(\alpha(n))$ time using a single processor.*

In the following, we shall denote by $\ell(u)$ the label of a node u. We define $P[\ell(u), \ell(v)]$ to be the product of the elements associated with the nodes lying on the path from u to v in a tree T. If u and v are the same node, then the above product is defined to be equal to $\ell(u)$ or $\ell(v)$. For a node w of T, we shall denote by $lc(w)$ (resp. $rc(w)$) its left (resp. right) child in T.

We shall consider first the case where the given labelled tree T is rooted and binary. At the end of the section we shall discuss how the general case is handled. Note that it suffices to give a data structure for answering range product queries between any two nodes u, v of T when u is a descendant of v (called *upward query*), or vice versa (called *downward query*). Since otherwise, let w be the lowest common ancestor of u and v in T and w.l.o.g. assume that u (resp. v) is a descendant of $lc(w)$ (resp. $rc(w)$). The path, in T, between u and v passes through w. Hence, $P[\ell(u), \ell(v)] = P[\ell(u), \ell(lc(w))] \circ \ell(w) \circ P[\ell(rc(w)), \ell(v)]$, i.e., it reduces to the problem of computing an upward and a downward range query. (Note that we have to consider separately upward and downward range queries, since the operator $\circ$ may not be commutative.)

Assume we are given a $(3, 8, m)$-partition of an n-node binary tree T into connected components T_i, $1 \leq i \leq 8n/m$. From the construction of the partition, it is clear that each component T_i will either have one or three outgoing edges. We use the following notation for certain nodes of the components. If T_i has only one outgoing edge, we shall refer to the single node incident on that edge as x_i.

If T_i has three outgoing edges, we shall refer to the two nodes of T_i incident on these edges as x_i and y_i, where x_i will be the one closest to the root of T. Note that in this latter case: (a) x_i is incident on one outgoing edge and y_i on the other two outgoing edges; (b) y_i is a descendant of $lc(x_i)$ or $rc(x_i)$.

Now, we construct a binary tree T', called *the upward condensed tree* of T, as follows. Replace each T_i in T by a subtree on nodes x_i and y_i, and edge (x_i, y_i) (if y_i exists). Node x_i in T' has label $\ell'(x_i) = \ell(x_i)$. Node y_i (if exists) has label $\ell'(y_i) = P[\ell(y_i), \ell(lc(x_i))]$ or $\ell'(y_i) = P[\ell(y_i), \ell(rc(x_i))]$, depending whether y_i is a descendant of $lc(x_i)$ or $rc(x_i)$.

We can similarly construct a *downward condensed tree* T''. In T'', node x_i will again have label $\ell(x_i)$. Node y_i (if exists) will have label $P[\ell(lc(x_i)), \ell(y_i)]$ (resp. $P[\ell(rc(x_i)), \ell(y_i)]$) if it is a descendant of $lc(x_i)$ (resp. $rc(x_i)$).

The next lemma shows that the above constructions can be done optimally on an EREW PRAM.

Lemma 1. *Both condensed trees of a labelled rooted n-node binary tree can be constructed in $O(\log n)$ time using $O(n)$ work on an EREW PRAM.*

Proof. We shall show how the upward condensed tree T' is constructed. The construction of the downward condensed tree is similar. Having the $(3, 8, m)$-partition of T into q connected components and using the input-output representation described in Sect. 2, we can find in $O(1)$ time the nodes x_i and y_i for each component T_i, by assigning one processor per each node of T. Then, we allocate an array A of $2q$ contiguous memory cells for T'. Each x_i and y_i is copied into the position $2(i-1)+1$ and $2(i-1)+2$ of A, respectively, for $1 \leq i \leq q$. It remains only to establish the parent-child relationships in T'. By the construction, node y_i knows immediately its parent x_i in T'. Hence, it remains only to inform x_i for its parent in T'. But this can be done using the data structure of T. All the above take $O(1)$ time and $O(n)$ work on an EREW PRAM. The bounds now follow from Theorem 1. □

The next lemma shows how a $(3, 8, m)$-partition of T and its upward condensed tree can be used to answer upward range product queries in T.

Lemma 2. *Assume we are given a $(3, 8, m)$-partition of an n-node labelled rooted binary tree T into connected components T_i, $1 \leq i \leq 8n/m$, and let T' be its upward condensed tree. If T' and all T_i's are provided with a data structure for answering upward range product queries, then: (i) We can answer correctly upward range product queries in T. (ii) If the label of a node w in T is changed, and w belongs also to component T_i, then an updating to the data structures of T' and T_i suffices to continue answering correctly upward range product queries in T.*

Proof. (i) Let u be a descendant of v in T. We want to show that $P[\ell(u), \ell(v)]$ is computed correctly using the data structures of T' and T_i's. If both u and v belong to the same component T_i, then clearly the data structure of T_i gives the correct range product. Therefore, assume that $u \in T_i$ and $v \in T_j$, $i \neq j$.

W.l.o.g. assume also that y_j, which is an ancestor of x_i, is a descendant of $lc(v)$. (The case where y_j is a descendant of $rc(v)$ is similar.)

Consider now the components T_p and T_q such that y_p is the parent of x_i and x_q is the child of y_j. From the associativity of $\circ$, we have that $P[\ell(u), \ell(v)] = P[\ell(u), \ell(x_i)] \circ P[\ell(y_p), \ell(x_q)] \circ P[\ell(y_j), \ell(v)]$. Clearly, the range products $P[\ell(u), \ell(x_i)]$ and $P[\ell(y_j), \ell(v)]$ can be obtained from the data structures of T_i and T_j respectively. In the special case where either y_p, x_q, y_j and v coincide, or y_p, x_q, x_i and u coincide, we make the convention that $P[\ell(y_p), \ell(x_q)]$, in the above product, is equal to the unit element of S.

Hence, to complete the proof, it suffices to show that $P[\ell(y_p), \ell(x_q)] = P_{T'}[\ell'(y_p), \ell'(x_q)]$, where the RHS product is taken from T' and $y_p \neq x_q$ (otherwise the proof is trivial). The proof goes by induction on the number t of nodes in the path from y_p to x_q in T'.

Consider first the basis case, $t = 0$. This means that $p = q$, and $P[\ell(y_p), \ell(x_q)] = P[\ell(y_p), \ell(x_p)] = P[\ell(y_p), \ell(lc(x_p))] \circ \ell(x_p) = \ell'(y_p) \circ \ell'(x_p) = P_{T'}[\ell'(y_p), \ell'(x_q)]$, by the construction of T' and the fact that x_p and x_q are the same nodes.

For the induction hypothesis, assume that $P[\ell(y_p), \ell(x_q)] = P_{T'}[\ell'(y_p), \ell'(x_q)]$, if the path contains at most $t - 1$ nodes.

For the induction step, let the path contain t nodes. Then $P[\ell(y_p), \ell(x_q)] = P[\ell(y_p), \ell(x_r)] \circ P[\ell(y_q), \ell(x_q)]$, where x_r is the child of y_q that belongs to component T_r. By the induction hypothesis, $P[\ell(y_p), \ell(x_r)] = P_{T'}[\ell'(y_p), \ell'(x_r)]$. By the associativity of $\circ$, $P[\ell(y_q), \ell(x_q)] = P[\ell(y_q), \ell(lc(x_q))] \circ \ell(x_q)$. But from the construction of T' we have, $P[\ell(y_q), \ell(lc(x_q))] \circ \ell(x_q) = P_{T'}[\ell'(y_q), \ell'(x_q)]$. Therefore, $P[\ell(y_p), \ell(x_q)] = P_{T'}[\ell'(y_p), \ell'(x_r)] \circ P_{T'}[\ell'(y_q), \ell'(x_q)] = P_{T'}[\ell'(y_p), \ell'(x_q)]$, as required.

(ii) We have just shown that if we have built data structures for all of T_i's and for T', we can correctly answer upward queries in T'. Therefore, it is clear that updating these data structures such that upward queries can be answered correctly in all T_i's and in T', is sufficient to answer correctly upward range queries in T. Hence, it remains to argue that we need to update only one of the components, namely the one, say T_i, which contains the node w whose label $\ell(w)$ has changed. But this follows immediately from the fact that all T_i's are node-disjoint, thus changing the label $\ell(w)$ of w in T_i does not affect any range product query in some T_j, $i \neq j$, and therefore its data structure. □

The following definition will facilitate the presentation of our results.

Definition 2. *Let $DS(T, \{P_W, P_T\}, \{U_W, U_T\}, Q)$ be a dynamic data structure for the range product problem on a tree T, where $O(P_W)$ (resp. $O(P_T)$) is the preprocessing work and space (resp. time) to be set up, $O(U_W)$ (resp. $O(U_T)$) is the work (resp. time) to update it after a modification in the value of an element associated with a tree node, and $O(Q)$ is the time to answer a product query using a single processor.*

The next lemma gives the bounds, and its proof explains the construction, of our dynamic data structures for the tree range product problem.

Lemma 3. *Let T be an n-node binary tree. Then, for each $k \geq 1$ and any integer $r \geq 0$, there exist dynamic data structures for the upward range product problem on T, with the following characteristics on an EREW PRAM:*

(i) $DS(T, \{(r+1)n - \sqrt{n}, 2\log n + 8r\}, \{c(r)n^{(1/2)^r}, 2\log n + 8r\}, 3^r\alpha(n))$;
(ii) $DS(T, \{((r+1)n - \sqrt{n})I_k(n), 2\log n + 8r\}, \{c(r)n^{(1/2)^r}, 2\log n + 8r\}, 3^r k)$,
where $c(0) = 1$ *and* $c(r) = c(r-1)[1 + 16^{(1/2)^{r-1}}]$ *for* $r \geq 1$.

Proof. We shall prove part (i). Part (ii) can be proved similarly. The proof proceeds by induction on r. If $r = 0$, then the update time exceeds the preprocessing and hence the static data structure of Theorem 2 suffices. In the following, we shall use the notation $D(T, n, r)$ for $DS(T, \{(r+1)n - \sqrt{n}, 2\log n + 8r\}, \{c(r)n^{(1/2)^r}, 2\log n + 8r\}, 3^r\alpha(n))$. Assume that the theorem holds for any value smaller than r. We shall show how $D(T, n, r)$ is constructed.

We first costruct a $(3, 8, \sqrt{n})$-partition of T and an upward condensed tree T'. Let $T_1, T_2, ..., T_q$, $\sqrt{n} \leq q \leq 8\sqrt{n}$, be the connected components of T, each one of size $n_i = |V(T_i)| = \sqrt{n}$ and connected with the rest of T through at most 3 edges. Also, $\sum_{i=1}^{q} n_i = n$, and T' has $n' = |V(T')| \leq 2q \leq 16\sqrt{n}$ nodes.

The data structure for $D(T, n, r)$ consists of the following: $D(T_i, n_i, r-1)$, for each $1 \leq i \leq q$, which enables us to answer upward range product queries in T_i, and $D(T', n', r-1)$ which enables us to answer upward range product queries in T'. By Lemma 2, maintaining these data structures is sufficient to answer correctly upward range product queries in T. Since in the same Lemma, we also showed how an upward query is answered having these data structures, it remains to argue here only for the resource bounds.

The time and work required for the preprocessing is equal to the time and work required for constructing: (1) the $(3, 8, \sqrt{n})$-partition of T and the upward condensed tree T', and (2) the dynamic data structures of T_i's and T' inductively. By Theorem 1 and Lemma 1, the total preprocessing work $P_W(n, r)$ is bounded by

$$P_W(n, r) \leq n + \sum_{i=1}^{q} P_W(n_i, r-1) + P_W(n', r-1).$$

By the induction hypothesis, we have

$$P_W(n, r) \leq n + \sum_{i=1}^{q} (rn_i - \sqrt{n}) + rn' - \sqrt{n} \leq (r+1)n - \sqrt{n}.$$

Similarly, the preprocessing time $P_T(n, r)$ is

$$\begin{aligned} P_T(n, r) &\leq \log n + \max\{P_T(n_i, r-1), P_T(n', r-1)\} \\ &\leq \log n + \max\{2\log n_i + 8(r-1), 2\log n' + 8(r-1)\} \\ &\leq 2\log n + 8r \end{aligned}$$

The time and work required for the update operation is the time and work to update $D(T_i, n_i, r-1)$ and $D(T', n', r-1)$. Using a similar argument as above,

we can show that the time required for an update is $2\log n + 8r$. The work $U_W(n,r)$ required for an update is

$$U_W(n,r) \leq U_W(n_i, r-1) + U_W(n', r-1)$$

which, by the induction hypothesis, gives:

$$\begin{aligned} U_W(n,r) &\leq c(r-1)n_i^{(1/2)^{r-1}} + c(r-1)(n')^{(1/2)^{r-1}} \\ &= c(r-1)n^{(1/2)^r}[1 + 16^{(1/2)^{r-1}}] \\ &= c(r)n^{(1/2)^r}. \end{aligned}$$

Finally, the query time $Q(n,r)$ is bounded by

$$Q(n,r) \leq 2Q(n_i, r-1) + Q(n', r-1)$$

where the first term corresponds to the time required for querying the data structures of the components in which the two nodes belong to, and the last term corresponds to the time required for querying in T'. By the induction hypothesis, this gives

$$Q(n,r) \leq 2(3^{r-1}\alpha(n_i)) + (3^{r-1}\alpha(n')) \leq 3^r\alpha(n).$$

Thus, we can construct $D(T,n,r)$ in the claimed bounds and hence completing the induction. □

Remark 1. Using symmetric arguments, we can prove similar Lemmata to 2 and 3 for answering downward range product queries in a labelled rooted binary tree, using the downward condensed tree T''.

We are now ready to give our main theorem.

Theorem 3. *Let T be an n-node unrooted tree such that each node is labelled with an element from a semigroup $(S,\circ)$. Then, for each $k \geq 1$ and any integer $r \geq 0$, there exist dynamic data structures for the range product problem on T, with the following characteristics on an EREW PRAM:*

(i) $DS(T, \{(r+1)n - \sqrt{n}, 2\log n + 8r\}, \{c(r)n^{(1/2)^r}, 2\log n + 8r\}, 3^r\alpha(n))$;
(ii) $DS(T, \{((r+1)n - \sqrt{n})I_k(n), 2\log n + 8r\}, \{c(r)n^{(1/2)^r}, 2\log n + 8r\}, 3^r k)$,
where $c(0) = 1$ *and* $c(r) = c(r-1)[1 + 16^{(1/2)^{r-1}}]$ *for* $r \geq 1$.

Proof. If T is unrooted, then we root it arbitrarily. If T is not binary, we convert it into a rooted binary tree, in the standard way, by adding dummy nodes. To these dummy nodes added, we associate the unit element of S. Note that the total number of nodes is at most $2n$. We then preprocess the tree (using e.g., the algorithm of [10]) such that lowest common ancestor queries can be answered in $O(1)$ time. All the above will cost $O(\log n)$ time and $O(n)$ work on an EREW PRAM using standard techniques (see e.g., [9]). After the above preprocessing, it follows by Lemmata 2 and 3 and Remark 1, that we can dynamically answer upward and downward range product queries in T. From the discussion in the beginning of Sect. 3, it follows that if we can answer upward and downward range product queries, then we can answer any range product query in T. The bounds follow easily by the above discussion and Lemma 3. □

An immediate consequence of Theorem 3 is a continuous trade-off between preprocessing, query, and update bounds, depending on the particular choice of r. For instance, choosing $r = -\log\beta$, where $0 < \beta < 1$ is any (arbitrarily small) constant, we get the following.

Corollary 1. *Let $k \geq 1$ be any constant integer and let $0 < \beta < 1$ be any (arbitrarily small) constant. The dynamic range product problem, on an n-node unrooted tree T, can be solved by constructing the following data structures on an EREW PRAM:*

(i) $DS(T, \{n, \log n\}, \{n^{\beta}, \log n\}, \alpha(n))$;
(ii) $DS(T, \{nI_k(n), \log n\}, \{n^{\beta}, \log n\}, k)$.

On the other hand, we may let r vary with n. For instance, choosing $r = \log\log n$, we have the following.

Corollary 2. *Let $k \geq 1$ be any constant integer and let $0 < \beta < 1$ be any arbitrarily small constant. The dynamic range product problem, on an n-node unrooted tree T, can be solved by constructing the following data structures on an EREW PRAM:*

(i) $DS(T, \{n\log\log n, \log n\}, \{\log n, \log n\}, \alpha(n)\log^{1.6} n)$;
(ii) $DS(T, \{nI_k(n)\log\log n, \log n\}, \{\log n, \log n\}, k\log^{1.6} n)$.

4 Conclusions

We have presented here simple and efficient dynamic algorithms for the tree range product problem that run on the weakest PRAM model. The dynamic tree range product problem appears to be a fundamental subproblem in many applications, as discussed in Sect. 1. We believe that our solution to this problem will help in the dynamization of graph problems which make use of tree data structures.

References

1. Alon, N., Schieber, B.: Optimal preprocessing for answering on-line product queries, Technical Report No. 71/87, Tel-Aviv University (1987)
2. Chaudhuri, S., Hagerup, T.: Prefix graphs and their applications. In: Mayr, Ernst W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 206–218. Springer, Heidelberg (1995)
3. Chaudhuri, S., Zaroliagis, C.: Shortest paths in digraphs of small treewidth. Part I: sequential algorithms. Algorithmica **27**(3), 212–226 (2000)
4. Chaudhuri, S., Zaroliagis, C.: Shortest paths in digraphs of small treewidth. Part II: optimal parallel algorithms. Theor. Comput. Sci. **203**(2), 205–223 (1998)
5. Chazelle, B.: Computing on a free tree via complexity-preserving mappings. Algorithmica **2**, 337–361 (1987)
6. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press, Cambridge (2009)

7. Date, C.J.: An Introduction to Database Systems, Vol. I, 5th edn. Addison-Wesley, Reading (1991)
8. Hagerup, T.: Parallel preprocessing for path queries without concurrent reading. Inf. Comput. **158**, 18–28 (2000)
9. JáJá, J.: An Introduction to Parallel Algorithms. Addison-Wesley, Reading (1992)
10. Schieber, B., Vishkin, U.: On finding lowest common ancestors: simplification and parallelization. SIAM J. Comput. **17**(6), 1253–1262 (1988)
11. Ullman, J.D.: Principles of Database and Knowledge-base Systems, Vol. II. Computer Science Press, Rockville (1989)