An Efficient Parallel Algorithm for Building the Separating Tree[1]

Yijie Han[1] *Sanjeev Saxena*[2] *Xiaojun Shen*[1]

[1]School of Computing and Engineering
University of Missouri at Kansas City
5100 Rockhill Road
Kansas City, MO 64110, USA
hanyij,shenx@umkc.edu

[2]Computer Science and Engineering
Indian Institute of Technology
Kanpur-208 016, India
ssax@cse.iitk.ac.in

Abstract

We present an efficient parallel algorithm for building the separating tree for a separable permutation. Our algorithm runs in $O(\log^2 n)$ time using $O(n \log^{1.5} n)$ operations on the CREW PRAM and $O(\log^2 n)$ time using $O(n \log n \log\log n)$ operations on the COMMON CRCW PRAM.
Keywords: Algorithms, parallel algorithms, separable permutation.

1 Introduction

In the pattern matching problem for permutations, a permutation $T = (t_1, t_2, ..., t_n)$ of $1, 2, ..., n$ is given as *text*. Another permutation $P = (p_1, p_2, ..., p_k)$ of $1, 2, ..., k$, $k \leq n$ is given as *pattern*. We want to find whether there is a length k subsequence T' of T, where $T' = (t_{i_1}, t_{i_2}, ..., t_{i_k})$ with $i_1 < i_2 < ... < i_k$, such that the elements of T' are ordered according to the permutation of P, i.e., $t_{i_r} < t_{i_s}$ iff $p_r < p_s$. This problem was proposed by Wilf[4, 10].

If T' exists, then we say that T contains P or P matches into T. When $P = (1, 2, ..., k)$, then we have to find an increasing subsequence of length k in T. The general problem is known to be NP-complete[4]. P is separable if P contains neither the sub-pattern $(3, 1, 4, 2)$ nor its reverse, the sub-pattern $(2, 4, 1, 3)$ [4, 10]. A separable permutation can also be defined by the sorting algorithm defined in the next paragraph. When P is separable, polynomial time algorithms for the decision and counting problems (decide whether T' exists and count the number of matches of P into T) exist[4, 10].

[1]Research supported in part by NSF grant 0310245. Preliminary version of this paper has been published in [6]. The claims made in [6] are incorrect and they are corrected in this journal version.

Separable permutations can be sorted in a particular way. Imagine a straight segment of railway track, with the elements of the permutation lined up on the track. Imagine that a segment in the middle of the track can be rotated 180° and reversing the order of the elements on the middle segment. We may move any consecutive subsequence of the elements onto the middle of the track and reverse their order. There is one restriction: a sequence of elements to be rotated on the middle section of the track must be first coupled together and remain coupled forever afterwards. Separable permutations are exactly the permutations that can be sorted using this method. The linear time algorithm [4] for sorting separable permutations also provides a test if a permutation is separable. It works as follows. Use a stack S whose elements are contiguous subranges $l, l+1, ..., l+m$ of the range $1, ..., k$. To add a contiguous range r to S, check if the top element of S forms a larger contiguous range together with r, i.e. the union of the two sets of numbers is a contiguous range. If so, pop the top element of S, form a combined contiguous range and adding that contiguous range to S, as just described. If at the end of P the stack S contains a single contiguous range then the permutation is separable, otherwise it is not. P's separating tree T can be easily constructed by this algorithm. This sequential sorting algorithm takes $O(k)$ time. A permutation can also be defined to be separable if it can be sorted by this sorting algorithm.

A separating tree for a separable permutation $P = (p_1, p_2, ..., p_k)$ of $1, 2, ..., k$ is a binary tree T with leaves $(p_1, p_2, ..., p_k)$ in that order, such that for each node v, if the leaves of the subtree rooted at v are $p_i, p_{i+1}, ..., p_{i+j}$, then the set of numbers $\{p_i, p_{i+1}, ..., p_{i+j}\}$ is a contiguous subrange of the range $1, 2, ..., k$, i.e. is of the form $\{l, l+1, ..., l+m\}$, for some l, m with $1 \leq l \leq k$, $0 \leq m \leq k-l$. This contiguous subrange is called the range of the node v. If v is a node of the tree with left child v_l and right child v_r then either the range of v_l just precedes the range of v_r or the range of v_r just precedes the range of v_l.

Lemma 1 ([4]): A pattern $P = (p_1, p_2, ..., p_k)$ is separable iff it has a separating tree.

Yugandhar and Saxena gave parallel algorithms for constructing the separating tree[13]. One of their algorithm runs in $O(\log n)$ time but uses $O(n^2)$ operations, where operation is defined to be the time processor product. Another parallel algorithm of theirs could be more efficient. It runs in $O(d \log n)$ time with $O(nd)$ operations, where d is the depth of an optimal tree. Their first algorithm is not efficient since it uses $O(n^2)$ operations compared to the sequential algorithm which has $O(n)$ operations. Their second algorithm is efficient when d is small. However, they show that there exists a separable permutation (i.e. $P = (1, 3, 5, 7, ..., n-1, n, .., 8, 6, 4, 2)$) such that the depth of its optimal separating tree is $\Theta(n)$.

Nevertheless Yugandhar and Saxena also gave parallel algorithms for testing whether a permutation is separable [13]. Their algorithm is very efficient and has time $O(\log n)$ with n processors on the CREW (Concurrent Read Exclusive Write) PRAM (Parallel Random Access Machine) model. See also Han and Saxena [8] for an $O(\log n)$ ($O(\log \log \log n)$) time and $n/\log n$ ($n/\log \log \log n$) processor CREW (CRCW) PRAM algorithm for testing the separability of a permutation.

In this paper we present an efficient parallel algorithm for building the separating tree. Our algorithm runs in $O(\log^2 n)$ time and $O(n \log^{1.5} n)$ operations on the CREW PRAM model and in $O(\log^2 n)$ time and $O(n \log n \log \log n)$ operations on the COMMON CRCW (Concurrent Read Concurrent Write) PRAM model.

CREW PRAM and CRCW PRAM are parallel computation models. On the CREW PRAM model all processors can read the shared memory concurrently, but only one processor can write into a shared memory cell in one step (although different processors can write into different memory cells concurrently). On the CRCW PRAM model more than one processor can concurrently read from or write into a shared memory cell. In the case of current write we use the COMMON model in which all processors writing into a memory concurrently have to write the same value.

The difficulty in building a separating tree efficiently in parallel is to identify a node v of the separating tree such that v has k/c leaves for a constant c. Suppose we identified such a node with leaves in the contiguous range $\{l, l+1, ..., l+m\}$, then we can take these leaves out, relabel each leaf t as $t - l + 1$. This forms one subproblem for building the separating tree. For the leaves remaining in the original tree we relabel each leaf as follows. If the leaf is less than l, we need not to relabel it. Since node v is now a leaf (since we deleted the subtree rooted at v), we label it as l. For each leave t greater than $l + m$ we relabel it as $t - m$. This forms the second subproblem for building the separating tree. These two separating tree building subproblems can be solved recursively in parallel. Since v has k/c leaves we know the recursion has $O(\log k)$ levels. If in each recursion level we spend $O(\log k)$ time and $O(k \log^{0.5} k)$ ($O(k \log \log k)$ operations on CREW PRAM (COMMON CRCW PRAM) then the time complexity of our algorithm will be $O(\log^2 k)$ and the operation complexity will be $O(k \log^{1.5} k)$ ($O(k \log k \log \log k)$) .

2 Identify a Contiguous Subsection

In order to identify a contiguous subsection to be rooted at node v we need to introduce several concepts.

2.1 Auxiliary Concepts

The all nearest smaller values problem is defined as follows[2]. Let $A = (a_1, a_2, ..., a_n)$ be an array of elements from a totally ordered domain. For each a_i, $1 \leq i \leq n$, find the nearest element to the left of a_i and the nearest element to the right of a_i that are less than a_i, if such elements exist. That is, for each $1 \leq i \leq n$, find the maximum $1 \leq j < i$, and the minimum $i < k \leq n$ such that $a_j < a_i$ and $a_k < a_i$. a_j is called the left match and a_k is the right match of a_i. The all nearest smaller values problem can be solved on the CREW PRAM in $O(\log n)$ time and $O(n)$ operations or on the COMMON CRCW PRAM in $O(\log \log n)$ time and $O(n)$ operations[2].

Given an array of n real numbers $A = (a_1, a_2, ..., a_n)$, a range minimum (maximum) query requests the minimum (maximum) elements in a subarray $A_{i,j} = (a_i, ..., a_j)$, for some $1 \leq i \leq j \leq n$. After $O(\log n)$ time and $O(n)$ operation preprocessing on the CREW PRAM or $O(\log \log n)$ time and $O(n)$ operation preprocessing on the COMMON CRCW PRAM, k queries can be processed in constant time using k processors[2].

The Cartesian tree is defined as follows[12]. The Cartesian tree for an array $A = (a_1, a_2, ..., a_n)$ of n distinct real numbers is a binary tree with vertices labeled by the numbers. The root has label a_m, where $a_m = \min\{a_1, a_2, ..., a_n\}$. Its left subtree is a Cartesian tree for $A_{1,m-1} = (a_1, ..., a_{m-1})$, and its right subtree is a Cartesian tree for $A_{m+1,n} = (a_{m+1}, ..., a_n)$. (The tree for an empty subarray is the empty tree.)

By the way the Cartesian tree is built, the inorder traversal of the Cartesian tree represents the order of the elements in array A.

We need the concept of Cartesian tree. For each number a_i in array A we need to know l and m such that $a_l, a_{l+1}, ..., a_m$ are the nodes of the sub-Cartesian tree rooted at a_i. For this purpose we use an observation in [2]. Let a_l and a_r be the left match and right match of a_i in A, if such exist. Then the nodes of the subtree rooted at a_i are $a_{l+1}, a_{l+2}, ..., a_{r-1}$. If $a_l > a_r$ then a_i is the right child of a_l. If $a_l < a_r$ then a_i is the left child of a_r.

2.2 The Partition Lemma

If nodes $v_1, v_2, ..., v_t$ in a tree are contiguous in the inorder traversal of the tree then we say that $v_1, v_2, ..., v_t$ is a contiguous section of the tree ($v_1, v_2, ..., v_t$ need not form a contiguous range). If nodes $v_1, v_2, v_3, ..., v_t$ are descendants of v and are just preceding v in the inorder traversal of the tree, then we say that $v_1, v_2, ..., v_t$ are v's left inorder neighbors. If nodes $v_1, v_2, v_3, ..., v_t$ are descendants of v and are just succeeding v in the inorder traversal of the tree, then we say that $v_1, v_2, ..., v_t$ are v's right inorder neighbors.

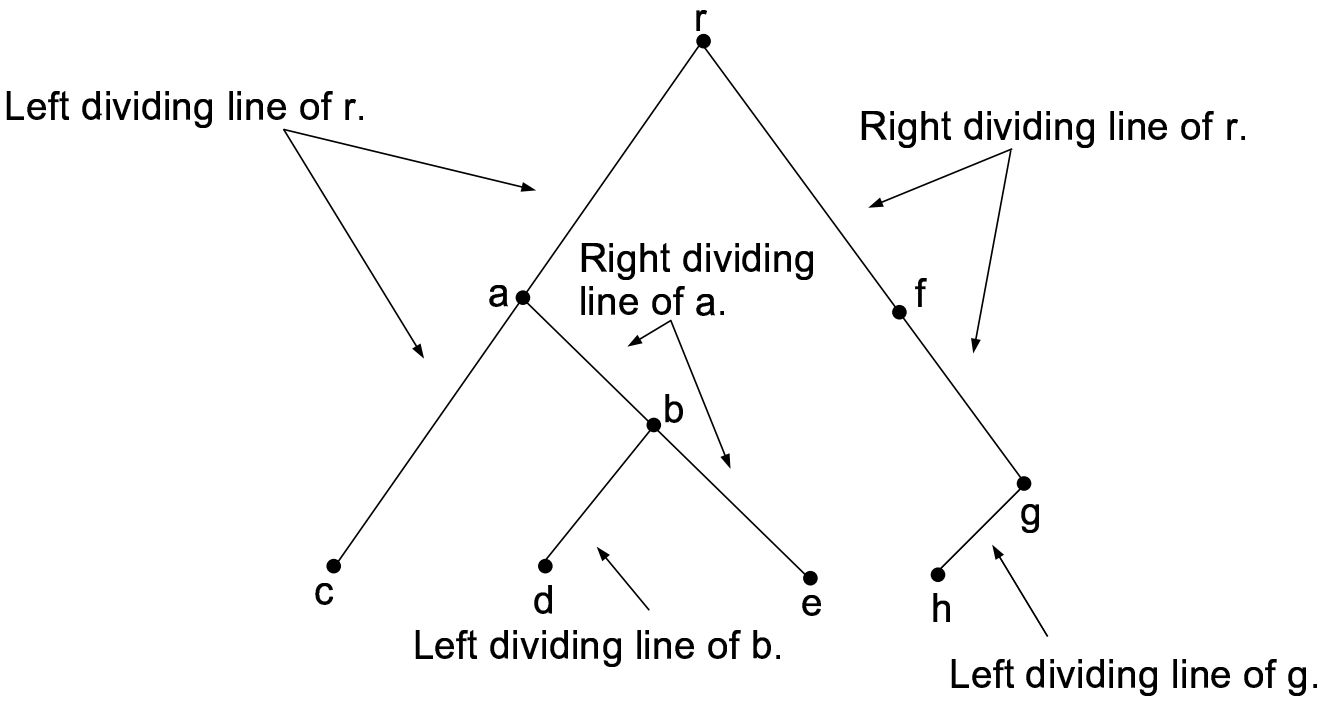

Fig. 1. Dividing lines. Note that f has no dividing lines.

Let r be the root of a tree T. Let a_{i+1} be a_i's left child (right child), $1 \leq i < t$, and a_1 be the left child (right child) of r and a_t has no left child (right child), then we say that $r, a_1, a_2, ..., a_t$ is the left dividing line (right dividing line) of r. Recursively, suppose the dividing line(s) of w has already been specified. Let v be on a left (right) dividing line of w, let b_{i+1} be b_i's right child (left child), $1 \leq i < s$, and b_1 be the right child (left child) of v and b_s has no right child (left child), then we say that $v, b_1, b_2, ..., b_s$ is the right dividing line (left dividing line) of v. Only root can have both left and right dividing line. Internal nodes of the tree can have only either left dividing line or right dividing line but not both.

Readers are referred to Fig. 1 to have a visual conception.

We have the following partition lemma:

Lemma 2 (Partition Lemma): Let v be a node in the Cartesian tree for a separable permutation. Suppose v's dividing line is a right (left) dividing line. Let $c(v)$ be v's right (left) child. Then the subtree rooted at $c(v)$, node v and a set S of v's left (right) inorder neighbors form a contiguous range. Besides, nodes in S partitions nodes in the subtree rooted at $c(v)$ in the following sense, if $a_1 < a_2 < ... < a_t$ are elements in S, then nodes u satisfying $a_i < u < a_{i+1}$ form a contiguous section in the right (left) subtree of v. If u, w are nodes in the right (left) subtree of v and $a_i < u < a_j < w$ then u precedes (succeeds) w in the inorder traversal of the tree and a_j precedes (succeeds) a_i in the inorder traversal of the tree.

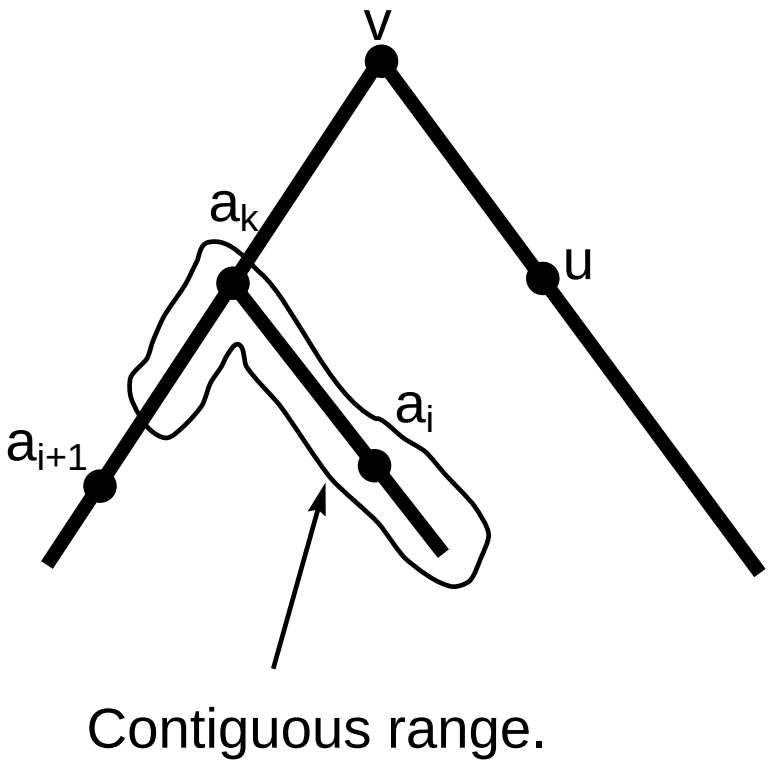

Fig. 2.

Proof: See Fig. 2. Let v have both left and right children. Let $a_1 < a_2 < ...a_t$ be the nodes in the left subtree of v. If there are u, w in the right subtree satisfying $a_i < u < a_{i+1} < w$, let a_k be the lowest common ancestor of a_i and a_{i+1}, then $a_k \neq v$ (this is because the

lowest common ancestor property ensures that a_i and a_{i+1} are not in the same subtree of a_k. Therefore $a_k \neq v$ because both a_i and a_{i+1} are in the left subtree of v.) and a_{i+1} must be in the left subtree rooted at a_k, for otherwise suppose a_{i+1} is in the right subtree of a_k, then we have a_k, a_{i+1}, v, u (this is the order they appear in the inorder traversal of the tree) forms a sequence of 2, 4, 1, 3, (v is the smallest because it is the root of the Cartesian tree. $a_k < a_i$ because a_k is the root of the Cartesian subtree where a_i is in and $a_i < u < a_{i+1}$ is given above.) and therefore the permutation is not separable (because of the sequence of 2, 4, 1, 3 as stated in the Introduction section). Thus whenever $a_i < u < a_j$ then a_j is in the left subtree of v whose root is the lowest common ancestor of a_i and a_j. Let x be the lowest common ancestor of u and w, then $x \neq v$ (same as above because of the lowest common ancestor property) and w must be in the right subtree of x, for otherwise a_{i+1}, v, w, x (this is the order they appear in the inorder traversal of the tree) forms sequence 3, 1, 4, 2 (v is the root and therefore the smallest. $x < u$ because u is in the Cartesian subtree rooted at x and $u < a_{i+1} < w$ is given above.) and therefore the permutation is not separable (because of the sequence 3, 1, 4, 2 as stated in the Introduction section.) .

This basically says that u partitions the nodes in the left subtree of v because nodes larger than u are in the left (subtree) and nodes smaller than u are in the right (subtree). Also a_{i+1} partitions the nodes in the right subtree of v. Therefore the nodes in the right subtree of v partitions the nodes in the left subtree of v and nodes in the left subtree of v partitions the nodes in the right subtree of v.

Now let v be the topmost node in the tree which has both left and right children (v need not be the root because the root may have only one child.). The nodes in the subtree rooted at v form a contiguous range (if there are ancestors of v those ancestors are the smallest ones because they are smaller than any node in the tree rooted at v.). By the proof above we know that nodes in the left subtree are partitioned into contiguous ranges and nodes in the right subtree are also partitioned into contiguous ranges. For each contiguous range, further branches in the tree again follow our proof above and further partitions this contiguous range into smalller contiguous ranges. □

2.3 Identify Subsection

For an input separable permutation $P = (p_1, p_2, ...p_k)$, we first compute, for each p_i the minimum and maximum range queries for the range $(p_l, p_{l+1}, ..., p_r)$, where $p_l, p_{l+1}, ..., p_r$ constitute all the nodes in the sub-Cartesian tree rooted at p_i. That is, p_{l-1} is the left match

of p_i and p_{r+1} is the right match of p_i, if they exist. This can be done for all p_i in constant time and $O(k)$ operations on the CREW PRAM after $O(\log k)$ time and $O(k)$ operation preprocessing on the CREW PRAM or $O(\log\log k)$ time and $O(k)$ operation preprocessing on the COMMON CRCW PRAM. Suppose the range maximum is a_M and range minimum is a_m. If $a_M - a_m = r - l$ then $p_l, p_{l+1}, ..., p_r$ form a contiguous range and the sub-Cartesian tree rooted at p_i can be taken out to form a subproblem for constructing the separating tree. What we wanted is $k/4 \leq r - l \leq k/2$. Suppose there exists a p_i for which $a_M - a_m = r - l$ and $k/4 \leq r - l \leq k/2$. Then we are done at this recursion level since we can take the sub-Cartesian tree rooted at p_i and form a subproblem for constructing the separating tree. The remaining numbers form another subproblem for constructing the separating tree. Thus we are done at this recursion level and we can continue with the recursion at the next level.

If, however, there is no p_i such that $a_M - a_m = r - l$ and $k/4 \leq r - l \leq k/2$, then we need more sophisticated solution. Here we resort to the Partition Lemma.

We obtain a node v of the Cartesian tree such that the sub-Cartesian tree rooted at v has $n(v)$ nodes, where $k/8 \leq n(v) \leq k/4$. We use $p(v)$ to denote v's parent. Let $p(v)$ have a right dividing line, v be $p(v)$'s right child, if such a node v exists. If such a node v does not exist then goto paragraph F. We sort these $n(v)$ nodes. Let $v_1 < v_2 < ... < v_{n(v)}$. We say that (v_i, v_{i+1}) is an interval of size $v_{i+1} - v_i - 1$.

If there is an interval of size s satisfying $s \geq k/8$, then by Partition Lemma these s nodes are in the left subtree of $p(v)$ and form a contiguous section. Since $n(v) \geq k/8$ we have that $s \leq k - n(v) \leq 7k/8$. Therefore we can take these s nodes out to form a subproblem and the recursion can continue.

If there is no interval of size $\geq k/8$ then all intervals are of size $< k/8$. In this case we take $v_1, v_2, ..., v_i$ such that interval (v_i, v_{i+1}) are of size ≥ 1 and $k/4 \leq v_i - v_1 \leq 3k/4$. Since we can add a contiguous section alternatively in the left subtree and right subtree of $p(v)$ and each such contiguous section is of size $< k/4$, therefore such v_i exists. Interval (v_i, v_{i+1}) of size ≥ 1 guarantees that $v_1, v_2, ..., v_i$ are in a contiguous section of the tree (because there is a node (because (v_i, v_{i+1}) of size ≥ 1) on the left subtree of $p(v)$ which groups $v_1, v_2, ..., v_i$ and separates them from v_{i+1} (by the Partition Lemma)). Also all nodes u satisfying $v_i < u < v_{i+1}$, $1 \leq i < n(v)$ are in a contigous section in the sibling tree (by the Partition Lemma). Note that all nodes w satisfying $v_1 \leq w \leq v_{n(v)}$ form a contiguous section in the tree. And therefore these nodes can be taken out to form a subproblem for the recursion.

The situation where $p(v)$ has a left dividing line and v is $p(v)$'s left child can be treated

similarly.

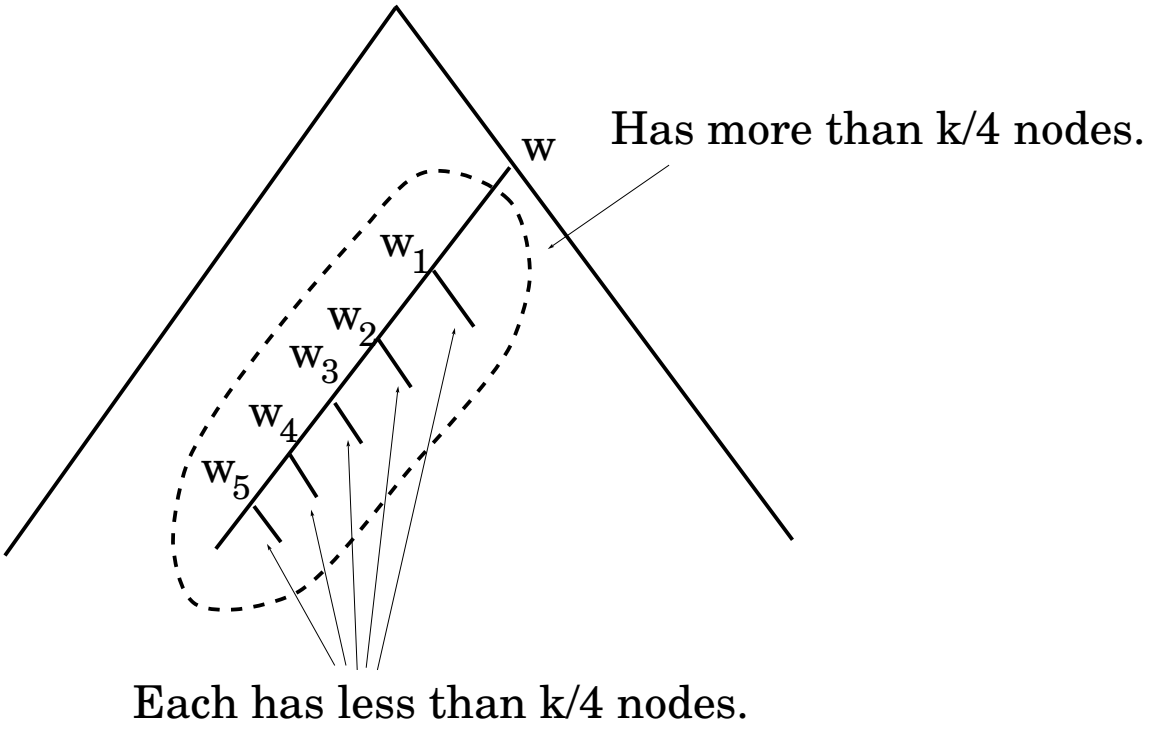

Fig. 3.

(Paragraph F) If such a node v does not exist, then we end up with the situation that there exists a node w such that the total number of descendants of nodes on the w's dividing line is at least $n/4$. Let $w_1, w_2, ..., w_t$ be the nodes on the dividing line of w. The number of nodes on the diving line of each w_i and their descendants is $< n/4$, as shown in Fig. 3. We sort nodes in the subtree T_i on w_i's dividing line (right subtree of w_i in Fig. 3), $1 \leq i \leq t$, separately in parallel. Let $w_{i,1} < w_{i,2} < ... < w_{i,a}$ be the nodes in T_i. If there is an interval $(w_{i,j}, w_{i,j+1})$ of size s satisfying $k/8 \leq s \leq k/4$ then we are done since these s nodes form a contiguous section in T_i's sibling tree. Otherwise there is a contiguous section S contains contiguous range of nodes with size $\geq k/4$ but further partition (with the Partition Lemma) has all intervals of size $< k/4$. Since each partition is of size $< k/4$, by grouping several partitions we can form a contiguous range and section T of size s satisfying $k/4 \leq s \leq k/2$. This grouping can be done by simply checking w_i's. Let $w_i, w_{i+1}, ..., w_a$ be the nodes on w's dividing line and also in T. Let $k/4 \leq w_j - w_i \leq k/2$, then we take the nodes on the

dividing line of $w_i, w_{i+1}, ..., w_{j-1}$ and their descendants and form a set U. We sort set U. Let $u_1 < u_2 < ... < u_b$ be the nodes in U. Let $c(w_i)$ be the child of w_i on the dividing line of w_i. Let $c_s(w_i)$ be the sibling of $c(w_i)$. We also take nodes z in the interval (u_i, u_{i+1}), i.e. $u_i < z < u_{i+1}$, $1 \leq i < b$, from the subtrees rooted at $c_s(w_i)$'s. This gives us a contiguous section and contiguous range of size s with $k/4 \leq s \leq k/2$. This is the section we will take out to form a subproblem.

The number of descendants of nodes in the tree can be computed using Euler tour [11] and linked list prefix ([1] for EREW and CREW PRAM and [7] for CRCW PRAM) in $O(\log n)$ time and linear operations. To check whether there is an interval of size $k/8$ is also done with linked list prefix or linked list contraction [1][7]. The nodes to be taken out can be labeled with 1's and other nodes can be labeled with 0's (this can be done by broadcasting 0's and 1's to consecutive memory addresses ($O(\log n)$ time and linear operations). And then by doing (linked list) prefix twice, once on nodes labeled with 0's and once on nodes labeled with 1's, the nodes labeled with 1's can be taken out. Set U is sorted using integer sorting which has time complexity $O(\log n)$ ($O(\log n / \log\log n)$) and operation complexity $O(n \log^{0.5} n)$ ($O(n \log\log n)$) on the CREW (CRCW) PRAM ([3] for CRCW PRAM and [9] for CREW PRAM).

In the worst case we take out 1/8 of the nodes, and the recurrence relation for time and operation complexity can be represented as:

$T(n) \leq c \log n + T(7n/8)$ for both CRCW and CREW PRAMs.

$Op(n) \leq cn \log^{0.5} n + Op(n/8) + Op(7n/8)$ for CREW PRAM.
$Op(n) \leq cn \log\log n + Op(n/8) + Op(7n/8)$ for CRCW PRAM.

where c is a constant. Thus we have:

Theorem 1: A separating tree for a separable permutation of size n can be computed on the CREW PRAM in $O(\log^2 n)$ time with $O(n \log^{1.5} n)$ operations, or on the COMMON CRCW PRAM in $O(\log^2 n)$ time with $O(n \log n \log\log n)$ operations.

3 Conclusion

Our improvements are build on Partition Lemma which outlines the distribution of continuous subsections on the Cartesian tree. Although we are able to improve previous results for building the separating tree, our results are shy of being optimal. Further improvements to approach optimality are left to the future work.

4 Acknowledgment

We are grateful to the reviewers of this paper. Their dedicated efforts made this paper much better. In particular, a reviewer's insistence on the time and operation complexity analysis made us aware of an analysis error in the conference version of this paper which is corrected here. In the conference version we miscalculated the time complexity for the CRCW PRAM as $O(\log n \log\log n)$ and miscalculated the operation complexity as $O(n \log n)$. The increased complexity comes from the complexity calculation of sorting and linked list prefix computation.

References

[1] R. J. Anderson, G. L. Miller. Deterministic parallel list ranking. *Algorithmica* 6: 859-868(1991).

[2] O. Berkman, B. Schieber, U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms* **14**, 344-370(1993).

[3] P. C. P. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, T. Radzik, S. Saxena. Improved deterministic parallel integer sorting. *Information and Computation* **94**, 29-47(1991).

[4] P. Bose, J.F. Buss, A. Lubiw. Pattern matching for permutations. *Proc. of the Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science*, 1993, 200-209.

[5] H.N. Gabow, J.L. Bentley, R.E. Tarjan. Scaling and related techniques for geometry problems. *Proc. 16th Annual ACM Symp. on Theory of Computing*, 1984, 135-143.

[6] Y. Han. An efficient parallel algorithm for building the separating tree. *Proc. 2006 Int. Conf. on Parallel and Distributed Techniques and Applications (PDPTA'06)* , 369-373(2006).

[7] Y. Han. Parallel algorithms for computing linked list prefix. *Journal of Parallel and Distributed Computing* **6**, 537-557(1989).

[8] Y. Han, S. Saxena. Test separability of permutations in parallel. Manuscript, July 16, 2009.

[9] Y. Han, X. Shen. Parallel integer sorting is more efficient than parallel comparison sorting on exclusive write PRAMs. *SIAM J. Comput.* 31, 6, 1852-1878(2002).

[10] L. Ibarra. Finding pattern matchings for permutations. Inform. Process. Lett. 61 (1997) 293-295.

[11] R. E. Tarjan, U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. *Proc. of 1984 IEEE Symposium on Foundations of Computer Science (FOCS'84)*, 12-20(1984),

[12] J. Vuillemin. A unified look at data structures. *Comm. ACM* **23** (1980) 229-239.

[13] V. Yugandhar, S. Saxena. Parallel algorithms for separable permutations. *Discrete Applied Mathematics*, 146 (2005) 343-364.