A Simple Parallel Cartesian Tree Algorithm and its Application to Parallel Suffix Tree Construction

JULIAN SHUN and GUY E. BLELLOCH, Carnegie Mellon University

We present a simple linear work and space, and polylogarithmic time parallel algorithm for generating multiway Cartesian trees. We show that bottom-up traversals of the multiway Cartesian tree on the interleaved suffix array and longest common prefix array of a string can be used to answer certain string queries. By adding downward pointers in the tree (e.g. using a hash table), we can also generate suffix trees from suffix arrays on arbitrary alphabets in the same bounds. In conjunction with parallel suffix array algorithms, such as the skew algorithm, this gives a rather simple linear work parallel, $O(n^\epsilon)$ time ($0 < \epsilon < 1$), algorithm for generating suffix trees over an integer alphabet $\Sigma \subseteq \{1, \ldots, n\}$, where n is the length of the input string. It also gives a linear work parallel algorithm requiring $O(\log^2 n)$ time with high probability for constant-sized alphabets. More generally, given a sorted sequence of strings and the longest common prefix lengths between adjacent elements, the algorithm will generate a patricia tree (compacted trie) over the strings. Of independent interest, we describe a work-efficient parallel algorithm for solving the all nearest smaller values problem using Cartesian trees, which is much simpler than the work-efficient parallel algorithm described in previous work.

We also present experimental results comparing the performance of the algorithm to existing sequential implementations and a second parallel algorithm that we implement. We present comparisons for the Cartesian tree algorithm on its own and for constructing a suffix tree. The results show that on a variety of strings our algorithm is competitive with the sequential version on a single processor and achieves good speedup on multiple processors. We present experiments for three applications that require only the Cartesian tree, and also for searching using the suffix tree.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Cartesian trees, suffix trees

ACM Reference Format:
Shun, J. and Blelloch, G. E. 2014. A simple parallel Cartesian tree algorithm and its application to parallel suffix tree construction. *ACM Trans. Parallel Comput.* 1, 1, Article 8 (September 2014), 20 pages.
DOI:http://dx.doi.org/10.1145/2661653

1. INTRODUCTION

A Cartesian tree on a sequence of elements taken from a total order is a binary tree that satisfies two properties: (1) heap order on values, *i.e.* a node has an equal or lesser value than any of its descendants, and (2) an in-order traversal of the tree defines the sequence order.

A preliminary version of this work appeared in the Proceedings of the Workshop on Algorithm Engineering and Experiments, ALENEX 2011 [Blelloch and Shun 2011]. This work was supported by generous gifts from Intel, Microsoft, and IBM and by the National Science Foundation under award CCF1314590. J. Shun is supported by a Facebook Graduate Fellowship.
Authors' address: J. Shun and G. E. Blelloch, Computer Science Department, Carnegie Mellon University; email: {jshun, guyb}@cs.cmu.edu.

DOI:http://dx.doi.org/10.1145/2661653

Given the suffix array (an array of pointers to the lexicographically sorted suffixes) of a string and lengths of the longest common prefixes (LCP) between adjacent entries, a Cartesian tree on the interleaved suffix array and LCPs can be used to answer queries related to the string. By adding downward pointers (e.g. using a hash table), this gives a suffix tree for binary alphabets. The approach can be generalized to arbitrary alphabets by using multiway Cartesian trees (Cartesian trees where connected components of equal value are contracted into single nodes) without much difficulty.

For a string s of length n over a character set $\Sigma \subseteq \{1,\ldots,n\}$[1] the suffix tree data structure stores all the suffixes of s in a patricia tree (a trie in which maximal branch-free paths are contracted into a single edge). In addition to supporting searches in s for any string $t \in \Sigma^*$ in $O(|t|)$ expected time,[2] suffix trees efficiently support many other operations on strings, such as finding the longest common substring, maximal repeats, longest repeated substrings, and the longest palindrome, among many others [Gusfield 1997]. As such, it is one of the most important data structures for string processing. For example, it is used in several bioinformatic applications, such as REPuter [Kurtz and Schleiermacher 1999], MUMmer [Delcher et al. 2002], OASIS [Meek et al. 2003] and Trellis+ [Phoophakdee and Zaki 2007, 2008]. Both suffix trees and a linear time algorithm for constructing them were introduced by Weiner [1973] (although he used the term position tree). Since then various similar constructions have been described [McCreight 1976; Ukkonen 1995] and there have been many implementations of these algorithms. Although originally designed for fixed-sized alphabets with deterministic linear time, Weiner's algorithm can work on an alphabet $\{1,\ldots,n\}$, henceforth $[n]$, in linear expected time simply by using hashing to access the children of a node.

The algorithms of Weiner and its derivatives are all incremental and inherently sequential. The first parallel algorithm for suffix trees was given by Apostolico et al. [1988] and was based on a quite different doubling approach. For a parameter $0 < \epsilon \leq 1$ the algorithm runs in $O(\frac{1}{\epsilon}\log n)$ time, $O(\frac{n}{\epsilon}\log n)$ work, and $O(n^{1+\epsilon})$ space on the CRCW PRAM for arbitrary alphabets. Although reasonably simple, this algorithm is likely not practical since it is not work efficient and uses super-linear memory (by a polynomial factor). The parallel construction of suffix trees was later improved to linear work and polynomial space by Sahinalp and Vishkin [1994], with an algorithm taking $O(\log^2 n)$ time on the CRCW PRAM (they note that linear space can be obtained by using hashing and randomization), and linear work and linear space by Hariharan [1994], with an algorithm taking $O(\log^4 n)$ time on the CREW PRAM. Farach and Muthukrishnan [1996] improved the time to $O(\log n)$ with high probability[3] on the CRCW PRAM. These later results are for constant-sized alphabets, are "considerably non-trivial," and do not seem to be amenable to efficient implementations.

As mentioned earlier, one way to construct a suffix tree is to first generate a suffix array and then convert it to a suffix tree using a Cartesian tree algorithm. Using suffix arrays is attractive since in recent years there have been considerable theoretical and practical advances in the generation of suffix arrays (see e.g. Puglisi et al. [2007]). The interest is partly due to their use in the popular Burrows-Wheeler compression algorithm [Burrows and Wheeler 1994], and also as a more space-efficient alternative to suffix trees. As such, there have been dozens of papers on efficient implementations of suffix arrays. Among these, Kärkkäinen, Sanders and Burkhardt have developed a

[1]More general alphabets can be used by first sorting the characters and then labeling them from 1 to n.
[2]Worst case time for constant sized alphabets.
[3]We use "with high probability" to mean with probability at least $1 - 1/n^c$ for any constant $c > 0$.

quite simple and efficient parallel algorithm for suffix arrays [Kärkkäinen et al. 2006; Kulla and Sanders 2007] that can also generate LCPs.

The case of generating Cartesian trees in parallel is less satisfactory. Berkman et al. [1993] describe a parallel algorithm for the all nearest smaller values (ANSV) problem, which can be used to generate a binary Cartesian tree. However, it cannot directly be used to generate a multiway Cartesian tree, and the algorithm is very complicated. Iliopoulos and Rytter [2004] present two much simpler algorithms for generating suffix trees from suffix arrays, one based on merging and one based on a variant of the ANSV problem that allows for multiway Cartesian trees. However they both require $O(n \log n)$ work.

In this article we describe a linear work, linear space, and polylogarithmic time algorithm for generating multiway Cartesian trees. The algorithm is based on divide-and-conquer and we describe two versions that differ in whether the merging step is done sequentially or in parallel. The first, based on a sequential merge, is very simple, and for a tree of height d, it runs in $O(\min\{d \log n, n\})$ time on the EREW PRAM. The second version is only slightly more complicated and runs in $O(\log^2 n)$ time on the EREW PRAM. They both use linear work and space.[4]

Given any linear work and space algorithm for generating a suffix array and corresponding LCPs using $O(S(n))$ time, our results lead to a linear work and space algorithm for generating suffix trees in $O(S(n) + \log^2 n)$ time. For example, using the skew algorithm [Kärkkäinen et al. 2006] on the CRCW PRAM, we have $O(\log^2 n)$ time with high probability for constant-sized alphabets and $O(n^{\epsilon})$ time ($0 < \epsilon < 1$) for the alphabet $[n]$. We note that a polylogarithmic time, linear work, and linear space algorithm for the alphabet $[n]$ would imply stable radix sort on $[n]$ in the same bounds, which is a long-standing open problem [Rajasekaran and Reif 1989].

For comparison, we also present a technique for using the ANSV problem to generate multiway Cartesian trees on arbitrary alphabets in linear work and space. The algorithm runs in $O(I(n) + \log \log n)$ time on the CRCW PRAM, where $I(n)$ is the best time bound for a linear-work stable sorting of integers from $[n]$. Of independent interest, we show that the Cartesian tree can be used to solve the ANSV problem in linear work and $O(\log^2 n)$ time, and the algorithm is much simpler than that of previous work [Berkman et al. 1993].

We have implemented the first version of our Cartesian tree algorithm and present various experimental results analyzing our algorithm on a shared memory multi-core parallel machine on a variety of inputs. First, we compare our Cartesian tree algorithm to a simple stack-based sequential implementation. On a single thread, our algorithm is about 3x slower, but we achieve about 35x speedup (about 12x with respect to the sequential implementation) on 40 cores with hyper-threading (2 threads per core). We show three queries on strings that can be answered with the Cartesian tree on the suffix array and LCP array of the string. First, we compute the number of leaves in the subtree at each internal node in order to support various types of queries relating to counts of repeated substrings in the input. As an example, we use this information to compute the longest substring that occurs at least k times in the input. Finally, we compute the minimum position of a suffix in the subtree of internal nodes, which is useful for computing the Ziv-Lempel decomposition of a string. These computations only require some basic parallel operations and are fast compared to the suffix array and LCP construction times.

[4]Very recently, Poon and Yuan [2013] have improved the time bound by describing a modification to our algorithm that runs in $O(n)$ work and $O(\log n)$ time.

We also analyze the Cartesian tree algorithm when used as part of code to generate a suffix tree from the original string. We compare the code to the ANSV-based algorithm described in the previous paragraph and to the fastest existing sequential implementation of suffix trees. Our algorithm is always faster than the ANSV-based algorithm. The algorithm is competitive with the sequential code on a single core, and achieves good speedup on 40 cores. We present timings for searching multiple strings in the suffix trees that we construct. On one core, our search times are always faster than searching with the sequential suffix tree and are an order of magnitude faster on 40 cores using hyper-threading.

2. PRELIMINARIES

Given a string s of length n over an ordered alphabet Σ, the *suffix array*, SA, represents the n suffixes of s in lexicographically sorted order. To be precise, $SA[i] = j$ if and only if the suffix starting at the j'th position in s appears in the i'th position in the suffix-sorted order. A *patricia tree* [Morrison 1968] (or compacted trie) of a set of strings S is a modified trie in which, (1) edges can be labeled with a sequence of characters instead of a single character, (2) no node has a single child, and (3) every string in S corresponds to concatenation of labels for a path from the root to a leaf. Given a string s of length n, the *suffix tree* for s stores the n suffixes of s in a patricia tree.

In this article, we assume an integer alphabet $\Sigma \subseteq [n]$, where n, is the total number of characters. We require that the patricia tree and suffix tree support the following queries on a node in constant expected time: finding the child edge based on the first character of the edge, finding the first child, finding the next and previous siblings in the character order, and finding the parent. If the alphabet size is constant, then all of these operations can easily be implemented in constant worst-case time.

A *Cartesian tree* [Vuillemin 1980] on a sequence of elements taken from a total order is a binary tree that satisfies two properties: (1) heap order on values, *i.e.* a node has an equal or lesser value than any of its descendants, and (2) an in-order traversal of the tree defines the sequence order. If the elements in the sequence are distinct, then the tree is unique, otherwise it might not be. When elements are not distinct we refer to a connected component of equal value nodes in a Cartesian tree as a *cluster*. A *multiway Cartesian tree* is derived from a Cartesian tree by contracting each cluster into a single node while maintaining the order of the children. A multiway Cartesian tree of a sequence is always unique.

Let $LCP(s_i, s_j)$ be the length of the longest common prefix of s_i and s_j. Given a sorted sequence of strings $S = [s_1, \ldots, s_n]$, if the string lengths are interleaved with the length of their longest common prefixes ($[|s_1|, LCP(s_1, s_2), |s_2|, \ldots, LCP(s_{n-1}, s_n), |s_n|]$) the corresponding multiway Cartesian tree has the structure of the patricia tree for S. The patricia tree can be generated by adding strings to the edges, which is easy to do, *e.g.* for a node with value $v = LCP(s_i, s_{i+1})$ and parent with value v', the edge corresponds to the substring $s_i[v' + 1, \ldots, v]$. As a special case, interleaving a suffix array with its LCPs and generating the multiway Cartesian tree gives the suffix tree structure. Adding the nodes to a hash table to allow for efficient downward traversals completes the suffix tree construction.

In this article, we use the exclusive-read exclusive-write (EREW) parallel random access machine (PRAM) model and the concurrent-read concurrent-write (CRCW) PRAM. For the CRCW PRAM, we use the model where concurrent writes to the same location results in an arbitrary processor succeeding. We analyze the algorithm in the work-time framework where we assume unlimited processors and count the number of time steps (T) and total number of operations (W). Using Brent's Work-Time scheduling principle, this implies an $O(\frac{W}{P} + T)$ time bound using P processors [Jaja 1992].

```
 1  struct node { node* parent; int value; };
 2
 3  void merge(node* left, node* right) {
 4    node* head;
 5    if ( left−>value > right−>value) {
 6      head = left;  left  = left−>parent;}
 7    else {  head = right; right= right−>parent; }
 8
 9    while(1) {
10      if ( left  == NULL) { head−>parent = right; break; }
11      if (right  == NULL) { head−>parent = left; break; }
12      if ( left−>value > right−>value) {
13        head−>parent = left; left = left−>parent; }
14      else {  head−>parent = right; right = right−>parent; }
15      head = head−>parent; }}
16
17  void cartesianTree(node* Nodes, int n) {
18    if (n < 2) return;
19    cilk_spawn cartesianTree(Nodes, n/2);
20    cartesianTree(Nodes+n/2, n−n/2);
21    cilk_sync;
22    merge(Nodes+n/2−1, Nodes+n/2);}
```

Fig. 1. C code for Algorithm 1a. The `cilk_spawn` and `cilk_sync` declarations are part of the CilkPlus parallel extensions [Leiserson 2010] and allow the two recursive calls to run in parallel.

3. PARALLEL CARTESIAN TREES

We describe a work-efficient parallel divide-and-conquer algorithm for constructing a Cartesian tree. The algorithm works recursively by splitting the input array A into two subarrays, generating the Cartesian tree for each subarray, and then merging the results into a Cartesian tree for A. We define the *right-spine* (*left-spine*) of a tree to consist of all nodes on the path from the root to the rightmost (leftmost) node of the tree. The merge works by merging the right-spine of the left tree and the left-spine of the right tree based on the value stored at each node. Our algorithm is similar to the $O(n \log n)$ work divide-and-conquer suffix array to suffix tree algorithm of Iliopoulos and Rytter [2004], but the most important difference is that our algorithm only looks at the nodes on the spines at or deeper than the deeper of the two roots, and our fully parallel version uses trees instead of arrays to represent the spines. This leads to our $O(n)$ work bound. In addition, Iliopoulos and Rytter's algorithm works directly on the suffix array rather than solving the Cartesian tree problem so the specifics are different.

We describe a partially parallel version of this algorithm (Algorithm 1a) and a fully parallel version (Algorithm 1b). Algorithm 1a is very simple, and takes up to $O(\min(d \log n, n))$ time, where d is the depth of the resulting tree, although for most inputs it takes significantly less time (e.g. for the sequence $[1, 2, \ldots, n]$ it takes $O(\log n)$ time even though the resulting tree has depth n). The algorithm only needs to maintain parent pointers for the nodes in the Cartesian tree. The complete `C` code is provided in Figure 1 and line numbers from it will be referenced throughout our description.

The algorithm takes as input an array of n elements (`Nodes`) and recursively splits the array into two halves (lines 19–21), creates a Cartesian tree for each half, and then merges them into a single Cartesian tree (line 22). For the merge (lines 3–15), we combine the right spine of the left subtree with the left spine of the right subtree (see Figure 2). The right (left) spine of the left (right) subtree can be accessed by

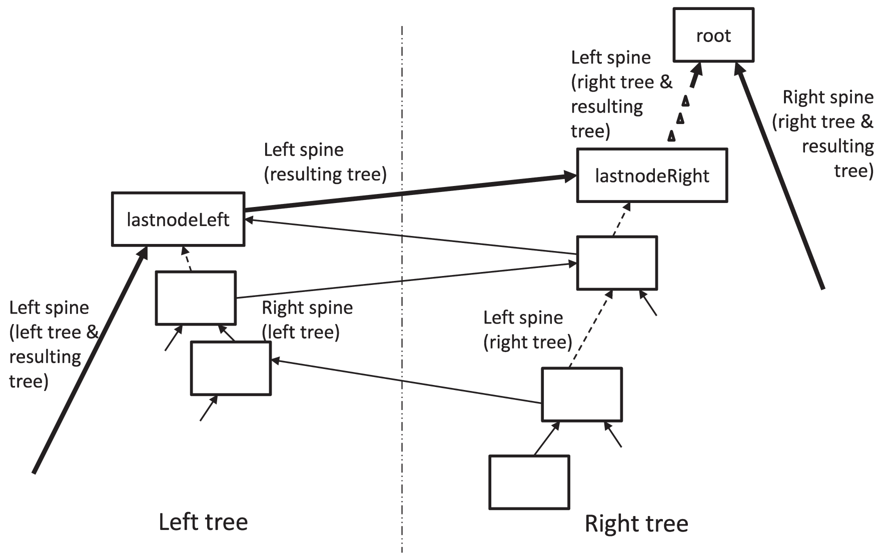
root
Left spine (right tree & resulting tree)
Right spine (right tree & resulting tree)
Left spine (resulting tree)
lastnodeRight
lastnodeLeft
Left spine (left tree & resulting tree)
Right spine (left tree)
Left spine (right tree)
Left tree
Right tree

Fig. 2. Merging two spines. Thick lines represent the *spines* of the resulting tree; dashed lines represent edges that existed before the merge but not after the merge; dotted edges represent an arbitrary number of nodes; all non-dashed lines represent edges in the resulting tree.

following parent pointers from the rightmost (leftmost) node of the left (right) subtree. The leftmost and rightmost nodes of a tree are simply the first and last elements in the input array of nodes. We note that once the merge reaches the deeper of the two roots, it stops and needs not process the remaining nodes on the other spine. The code in Figure 1 does not keep child pointers since we do not need them for our experiments, but it is easy to add a left and right child pointer and maintain them.

THEOREM 3.1. *Algorithm 1a produces a Cartesian tree on its input array.*

PROOF. We show that at every step in our algorithm, both the heap and the in-order properties of the Cartesian trees are maintained. At the base case, a Cartesian tree of one node trivially satisfies the two properties. During a merge, the heap property is maintained because a node's parent pointer only changes to point to a node with equal or lesser value. Consider modifications to the left tree. Only the right children of the right spine can be changed. Any new right children of a node come from the right tree, and hence correspond to elements later in the original sequence. An in-order traversal will correctly traverse these new children of a node after the node itself. A symmetric argument holds for nodes on the left spine.

Furthermore the order within each of the two trees is maintained since any node that is a descendant on the right (left) in the trees before merging remains a descendant on the right (left) after the merge. □

THEOREM 3.2. *Algorithm 1a for constructing a Cartesian tree requires $O(n)$ work, $O(\min(d \log n, n))$ time, and $O(n)$ space on the EREW PRAM.*

PROOF. We use the following definitions to help with proving the complexity bounds of our algorithm. A node in a tree is *left-protected* if it does not appear on the left spine of its tree, and a node is *right-protected* if it does not appear on the right spine of its tree. A node is *protected* if it is both left-protected and right-protected.

In the algorithm, once a node becomes protected, it will always be protected and will never have to be looked at again since the algorithm only ever processes the left and right spines of a tree. We show that during a merge, all but two of the nodes that are looked at become protected, and we charge the cost of processing those two nodes to the merge itself. Call the last node looked at on the right spine of the left tree *lastnodeLeft* and the last node looked at on the left spine of the right tree *lastnodeRight* (see Figure 2).

All nodes that are looked at, except for lastnodeLeft and lastnodeRight will be left-protected by lastnodeLeft. This is because those nodes become either descendants of the right child of lastnodeLeft (when lastnodeLeft is below lastnodeRight) or descendants of lastnodeRight (when lastnodeRight is below lastnodeLeft). A symmetric argument holds for nodes being right-protected. Therefore, all nodes looked at, except for lastnodeLeft and lastnodeRight, become protected after this sequence of operations. We charge the cost for processing lastnodeLeft and lastnodeRight to the merge itself.

Other than when a node appears as lastnodeRight or lastnodeLeft it is only looked at once and then becomes protected. Therefore the total number of nodes looked at is $2n-2$ for lastnodeRight or lastnodeLeft on the $n-1$ merges, and at most n for the nodes that become protected for a total work of $O(n)$.

Although Algorithm 1a makes parallel recursive calls, it uses a sequential merge routine. In the worst case this can take time equal to the depth of the tree per level of recursion. As there are $O(\log n)$ levels of recursion, the time spent is $O(\min(d \log n, n))$. The parallel recursive calls are on different parts of the data, so it runs on the EREW PRAM.

Since each node only maintains a constant amount of data, the space required is $O(n)$. □

We now describe a fully parallel version of the algorithm, which we refer to as Algorithm 1b. The algorithm maintains binary search trees for each spine, and substitutes the sequential merge with a parallel merge. We will use split and join operations on the spines. A split goes down the spine tree and cuts it at a specified value v so that all values less than or equal to v are in one tree and values greater than v are in another tree. A join takes two trees such that all values in the second are greater than or equal to the largest value in the first, and joins them into one. Both operations can run in time proportional to the depth of the spine tree and the join adds at most one to the height of the larger of the two trees.

Without loss of generality, assume that the root of the right Cartesian tree has a smaller value than the root of the left Cartesian tree (as in Figure 2). For the left tree, the end point of the merge will be its root. To find where to stop merging on the right tree, the algorithm searches the left-spine of the right tree for the root value of the left tree and splits the spine at that point. Now it merges the whole right-spine of the left tree and the deeper portion of the left-spine of the right tree. After the merge these two parts of the spine can be discarded, since their nodes have become protected. Finally the algorithm joins the shallower portion of the left spine of the right tree with the left spine of the left tree to form the new left spine. The right-spine of the resulting Cartesian tree is the same as that of the right Cartesian tree before the merge.

THEOREM 3.3. *Algorithm 1b for constructing a Cartesian tree requires $O(n)$ work, $O(\log^2 n)$ time, and $O(n)$ space on the EREW PRAM.*

PROOF. The trees used to represent the spines are never deeper than $O(\log n)$, since each merge does only one join, which adds only one node to the depth. All splits and joins therefore take $O(\log n)$ time. The merge can be done using a parallel merging algorithm that runs in $O(\log n)$ time and $O(n)$ work on the EREW PRAM [Hagerup and Rüb 1989], where n is the number of elements being merged. The depth of Algorithm 1b's recursion is $O(\log n)$, which gives a $O(\log^2 n)$ time bound. The $O(n)$ work bound follows from a similar analysis to that of Algorithm 1a, with the exception that splits and joins in the spine cost an extra $O(\log n)$ per merge, so for the extra cost we have a recurrence $W(n) = 2W(n/2) + O(\log n)$, which solves to $O(n)$. The trees on the spines take linear space so the $O(n)$ space bound still holds. Processor allocation on each level of recursion is straightforward to do within $O(\log n)$ time. □

LEMMA 3.4. *The outputs of Algorithm 1a and Algorithm 1b can be used to construct a multiway Cartesian tree in $O(n)$ work and space. On the EREW PRAM, this requires $O(d)$ time using path compression or $O(\log n)$ time using tree contraction.*

PROOF. We can use path compression to compress all clusters of the same value into the root of the cluster, which can then be used as the representative of the cluster. All parent pointers to nodes in a cluster will now point to the representative of that cluster. This is done sequentially and requires linear work and $O(d)$ time. We can also substitute path compression with a parallel tree contraction algorithm, which requires $O(n)$ work and $O(\log n)$ time on the EREW PRAM [Reid-Miller et al. 1993]. □

For non-constant sized alphabets if we want to search in the tree efficiently ($O(1)$ expected time per edge) the edges need to be inserted into a hash table, which can be done in $O(\log n)$ time and $O(n)$ work (both with high probability) on a CRCW PRAM.

COROLLARY 3.5. *Given a suffix array for a string over the alphabet $[n]$ and the LCPs between adjacent elements, a suffix tree can be generated in hash table format with Algorithm 1b, tree contraction, and hash table insertion using $O(n)$ work and $O(\log^2 n)$ time with high probability, and $O(n)$ space on the CRCW PRAM.*

PROOF. This follows directly from Theorem 3.3, Lemma 3.4, and the bounds for hash table insertion. □

4. CARTESIAN TREES AND THE ANSV PROBLEM

The *all nearest smaller values (ANSV)* problem is defined as follows: for each element in a sequence of elements from a total ordering, find the closest smaller element to the left of it and the closest smaller element to the right of it. Here we augment the ANSV-based Cartesian tree algorithm of Berkman et al. [1993] to support multiway Cartesian trees, and also show how to use Cartesian trees to solve the ANSV problem.

The algorithm of Berkman et al. [1993] solves the ANSV problem in $O(n)$ work and $O(\log \log n)$ time on the CRCW PRAM. The ANSV can then be used to generate a Cartesian tree by noting that the parent of a node has to be the nearest smaller value in one of the two directions (in particular the larger of the two nearest smaller values is the parent). To convert their Cartesian tree to the multiway Cartesian tree, one needs to group all nodes pointing to the same parent and coming from the same direction together. If $I(n)$ is the best time bound for stably sorting integers from $[n]$ using linear space and work, then the grouping can be done in linear work and $O(I(n) + \log \log n)$ time by sorting on the parent ID numbers of the nodes. Stability is important since a suffix tree needs to maintain the relative order among the children of a node.

THEOREM 4.1. *A multiway Cartesian tree on an array of elements can be generated in $O(n)$ work and space and $O(I(n) + \log \log n)$ time on the CRCW PRAM.*

PROOF. This follows from the bounds of the ANSV algorithm and of integer sorting. □

It is not currently known whether $I(n)$ is polylogarithmic so at present this result seems weaker than the result from the previous section. In the experimental section, we compare the algorithms on various inputs. In a related work, Iliopoulos and Rytter

[2004] present an $O(n \log n)$ work polylogarithmic time algorithm based on a variant of ANSV.

4.1. Cartesian Tree to ANSV

We now describe a method for obtaining the ANSVs from a Cartesian tree in parallel using tree contraction. Note that for any node in the Cartesian tree, both of its nearest smaller neighbors (if they exist) must be on the path from the node to the root (one neighbor is trivially the node's parent). We first present a simple linear-work algorithm for the task that takes time equal to the depth of the Cartesian tree. Let d denote the depth of the tree, with the root being at depth 1. The following algorithm returns the left nearest neighbors of all nodes. A symmetric algorithm returns the right nearest neighbors.

(1) For every node, maintain two variables, *node.index* which is set to the node's index in the sequence corresponding to the in-order traversal of the Cartesian tree and never changed, and *node.inherited*, which is initialized to *null*.
(2) For each level i of the tree from 1 to d: In parallel, for all nodes at level i: pass *node.inherited* to its left child and *node.index* to its right child. The child stores the passed value in its *inherited* variable.
(3) For all nodes in parallel: if *node.inherited* $\neq$ *null*, then *node.inherited* denotes the index of the node's left smaller neighbor. Otherwise it does not have a left smaller neighbor.

By using parallel tree contraction [Reid-Miller et al. 1993], we obtain a linear-work and fully parallel algorithm for computing the ANSVs, as described in the following theorem.

THEOREM 4.2. *There is an linear-work algorithm for computing the ANSVs of a sequence using $O(\log^2 n)$ time on the EREW PRAM.*

PROOF. We first compute the binary Cartesian tree of the input sequence. Then we perform tree contraction on the resulting Cartesian tree. We describe tree contraction operations for finding the smaller left neighbors; the procedure for finding the smaller right neighbors is symmetric. To find the left neighbors, we describe how to compress and uncompress the tree for several configurations, and the rest of the configurations have a symmetric argument. For compression, there are the left-left and right-left configurations. The *left-left* configuration consists of three nodes A, B, and C, with B being the left child of A, and C being the left child of B. For this configuration, B is the compressed node, and during uncompression B takes the *inherited* value of A and passes its *inherited* value to C. The *right-left* configuration consists of three nodes A, B, and C, with B being the right child of A, and C being the left child of B. For this configuration, B is again the compressed node, and during uncompression takes the *index* value of A and passes its *inherited* value to C. The *right-right* and *left-right* configurations are defined similarly and have symmetric properties. A raked left leaf takes the *inherited* value of its parent when it is unraked, and a raked right leaf takes the *index* value of its parent when it is unraked. Note that values are only passed during uncompression and unraking, and not during compression and raking. Tree contraction requires $O(n)$ work and $O(\log n)$ time on the EREW PRAM. Combined with the complexity bounds for generating the Cartesian tree of Theorem 3.3, this gives us a $O(n)$ work and $O(\log^2 n)$ time algorithm on the EREW PRAM for computing all nearest smaller values. □

Although the time complexity is higher, our algorithm is much simpler than the linear-work algorithms of Berkman et al. [1993].

5. EXPERIMENTS

The goal of our experiments is to analyze the efficiency of our parallel Cartesian tree algorithm both on its own and also as part of code to generate suffix trees. We show three applications of the Cartesian tree for answering queries on strings. We compare the Cartesian tree-based suffix tree algorithm to the ANSV-based algorithm and to the best available sequential code for suffix trees. We believe that it is important to compare our parallel algorithm with existing sequential implementations to make sure that the algorithm can significantly outperform existing sequential ones even on modest numbers of processors (cores) available on current desktop machines. In our discussion, we refer to the two variants of our main algorithm (Section 3) as Algorithm 1a and Algorithm 1b, and to the ANSV-based algorithm as Algorithm 2. For the experiments, in addition to implementing Algorithm 1a and a variant of Algorithm 2, we implemented parallel code for computing suffix arrays and their corresponding LCPs, and parallel code for inserting the tree nodes into a hash table to allow for efficient search (these codes are now part of the Problem Based Benchmark Suite [Shun et al. 2012]). All of our experiments were performed on a 40-core parallel machine (with hyper-threading) using a variety of real-world and artificial strings. Our suffix tree code is available for download.[5]

5.1. Auxiliary Code

To generate the suffix array and LCPs, we implemented a parallel version of the skew algorithm [Kärkkäinen et al. 2006]. The implementation uses a parallel radix sort, requiring $O(n)$ work and $O(n^{\epsilon})$ time for some constant $0 < \epsilon < 1$. Our LCP code is based on an $O(n \log n)$ work solution for the range-minima problem instead of the optimal $O(n)$. The $O(n \log n)$ work solution creates a table with $\log n$ levels, where the i'th level of the table stores the minimum value of every interval of length 2^i in the sequence (computed in parallel from the $i - 1$'st level). We did implement a parallel version of the linear-time range-minima algorithm by Fischer and Heun [2006], but found that it was slower. Due to better locality in the parallel radix sort than the sequential one, our code on a single core is actually faster than a version of Kärkkäinen et al. [2006] implemented in the paper and available online, even though that version does not compute the LCPs. We get a further 9 to 27 fold speedup on a 40 core machine. Compared to the parallel implementation of suffix arrays by Kulla and Sanders [2007], our times are faster on 40 cores than the 64 core numbers reported by them (10.8 seconds vs 37.8 seconds on 522 million characters), although their clock speed is slower than ours and it is a different system so it is hard to compare directly. Mori provides a parallel suffix array implementation using OpenMP [Mori 2010a], but we found that it is slower than their corresponding sequential implementation. Our parallel implementation significantly outperforms that of Mori. These are the only two existing parallel implementations of suffix arrays that we are aware of.

We note that recent sequential suffix array codes are faster than ours running on one core [Mori 2010a, 2010b; Puglisi et al. 2007], but most of them do not compute the LCPs (though these could be computed sequentially in a post-processing step [Kasai et al. 2001; Kärkkäinen et al. 2009]). For real-world texts, those programs are faster than our code due to many optimizations that these programs make. We expect that many of these optimizations can be parallelized and could significantly improve the performance of parallel suffix array construction, but this was not the purpose of our studies. One advantage of basing suffix tree code on suffix array code, however, is that

[5]http://www.cs.cmu.edu/~jshun/suffixTree.tar.gz

improvements made to parallel suffix arrays would improve the performance of the suffix tree code as well.

We use a parallel hash table [Shun and Blelloch 2014] to allow for fast search in the suffix tree. The hash table uses a compare-and-swap primitive for concurrent insertion. Furthermore we optimized the code so that most entries near leaves of the tree are not inserted into the hash table and a linear search is used instead. In particular, since our Cartesian tree code stores the tree nodes as an in-order traversal of the suffixes of the suffix tree, a child and parent near the leaf are are likely to be near each other in this array. In our code, if the child is within some constant c (16 in the experiments) in the array we do not store it in the hash table, but instead use a linear search.

For Algorithm 2, we use an optimized $O(n \log n)$ work and $O(\log n)$ time ANSV algorithm, which was part of the code of Shun and Zhao [2013], instead of the much more complicated work-optimal version of Berkman et al. [1993].

5.2. Experimental Setup

We performed experiments on a 40-core Intel machine (with hyper-threading) with $4 \times$ 2.4GHz Intel 10-core E7-8870 Xeon processors (1066MHz bus and 30MB L3 cache) and 256GB of main memory. The parallel programs were compiled using the Intel `icpc` compiler (version 12.1.0, which supports CilkPlus [Leiserson 2010]) with the `-O2` flag. The sequential programs were compiled using g++ 4.4.1 with the `-O2` flag.

For comparison to sequential suffix tree code we used the code of Tsadok and Yona [2003] and Kurtz's code from the MUMmer project [Delcher et al. 2002; Kurtz 1999],[6] both of which are publicly available. We only list the results of Kurtz because they are superior to those of Tsadok and Yona [2003] for all of our test files. Kurtz's code is based on McCreight's suffix tree construction algorithm [McCreight 1976]—it is inherently sequential and completely different from our algorithms. Other researchers have experimented with building suffix trees in parallel [Ghoting and Makarychev 2009; Tsirogiannis and Koudas 2010] and our running times appear significantly faster than those reported in the corresponding papers, even after accounting for differences in machine specifications. Iliopoulos and Rytter [2004] describe how to transform a suffix array into a suffix tree in parallel in $O(n \log n)$ work, but they do not have an implementation available.

Independent of our work, Mansour et al. [2011] have developed a disk-based parallel suffix tree algorithm, which works for input strings that do not fit in main memory. Our algorithm is faster than theirs on a per-core basis—on the human genome (3.08 GB), our algorithm takes 168 seconds using 40 cores while the algorithm of Mansour et al. takes 19 minutes on 8 cores and 11.3 minutes on 16 cores. However, their algorithm is disk-based and requires less memory than ours. We note that their algorithm requires super-linear work and time in the worst case. Very recently, Comin and Farreras [2013] described a parallel disk-based algorithm implemented using MPI. For the human genome, they report a running time of 7 minutes using 172 processors, which is slower than our algorithm using 40 cores. However, their algorithm again is disk-based, and their experiments were done on older hardware. A description of other disk-based suffix tree algorithms can be found in the survey by Barsky et al. [2010].

For running the experiments we used a variety of strings available online,[7] a Microsoft Word document (thesaurus.doc), XML code from Wikipedia samples

[6]http://mummer.sourceforge.net

[7]http://people.unipmn.it/manzini/lightweight/corpus/

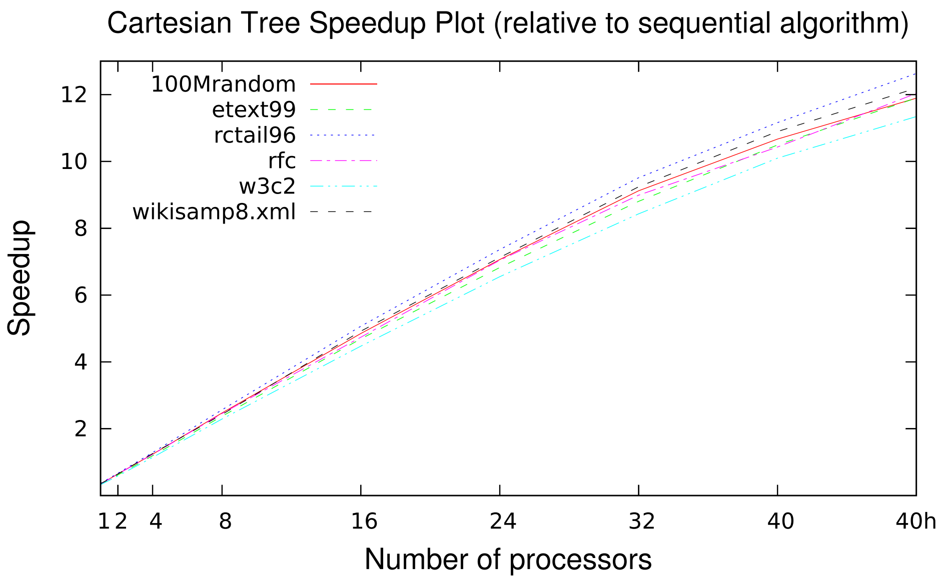

Fig. 3. Speedup of our Cartesian tree algorithm relative to the stack-based sequential algorithm on a 40 core machine. "40h" indicates 80 hyper-threads.

(wikisamp8.xml and wikisamp9.xml), the human genome[8] (HG18.fasta), and artificial inputs. Our artificial inputs are all of size 10^8 and include an-all identical string (100Midentical), random strings with an alphabet size of 10 (100Mrandom), and a string with an alphabet size of 2 where every 10^4th position contains one character and all other positions contain the other character (100Msqrtn). We also included two files of integers, one with random integers ranging from 1 to 10^4 (100MrandomInts10K), and one with random integers ranging from 1 to 2^{31} (100MrandomIntsImax), to show that our algorithms run efficiently on arbitrary integer alphabets. See Table II for all of the input sizes.

5.3. Cartesian Trees

We experimentally compare our Cartesian tree algorithm from Algorithm 1 to the linear-time stack-based sequential algorithm of Gabow et al. [1984]. There is also a linear-time sequential algorithm based on ANSVs, but we verified that the stack-based algorithm outperforms the ANSV one so we only report times for the former. Figure 3 shows the speedup of our parallel Cartesian tree algorithm with respect to the sequential stack-based algorithm on the interleaved SA and LCPs of various inputs. Our parallel algorithm outperforms the sequential algorithm with 4 or more cores, and achieves about 35x speedup (about 12x speedup with respect to the sequential algorithm). The performance is consistent across the different inputs.

5.4. Applications of Cartesian Trees

We are able to answer certain string queries by bottom-up traversals of the Cartesian tree built on the interleaved SA and LCPs of the string, which is essentially a suffix tree without the downward pointers. Abouelhoda et al. [2004] show how to perform certain suffix tree queries using just the SA and LCPs. Their sequential stack-based

[8]http://webhome.cs.uvic.ca/~thomo/HG18.fasta.tar.gz

Table I. Times for Bottom-Up Traversals

Text	Cartesian tree			Leaf counts			Longest substring ($k = 10$)			Leftmost suffix positions		
	T_{40}	T_1	SU	T_{40}	T_1	SU	T_{40}	T_1	SU	T_{40}	T_1	SU
100Midentical	0.23	6.61	28.7	1.7	3.27	1.92	1.69	3.61	2.14	1.73	3.37	1.95
etext99	0.26	9.2	35.4	0.12	4.77	39.8	0.14	5.19	37.1	0.12	4.87	40.6
rctail96	0.28	9.58	34.2	0.15	5.1	34	0.16	5.71	35.7	0.14	5.22	37.3
rfc	0.28	9.78	34.9	0.15	5.16	34.4	0.16	5.66	35.4	0.14	5.27	37.6
w3c2	0.26	9.14	35.2	0.13	4.37	33.6	0.14	4.83	34.4	0.13	4.47	34.4
wikisamp8.xml	0.25	8.52	34.1	0.12	4.48	37.3	0.13	4.99	38.4	0.12	4.58	38.2

Note: Times (seconds) for bottom-up traversals on a 40 core machine with hyper-threading. T_{40} is the time for our parallel algorithm on 40 cores (80 hyper-threads), T_1 is the single-thread time and SU is the speedup computed as T_1/T_{40}.

method essentially computes the ANSVs on the array of LCPs to generate the tree structure, similar to the classic sequential stack-based ANSV algorithm. Berkman et al. [1993] showed how to parallelize the ANSV algorithm, which we generalized in Section 4. At the end of the day, however, we found that building the Cartesian tree directly is more efficient than using the ANSV method, at least in parallel (see the "Cartesian tree" timings in Figure 5 versus the "ANSV", "Compute parents" plus "Create internal nodes" timings in Figure 6). As with the Abouelhoda et al. method, building the Cartesian tree is so fast that it can be recomputed per bottom-up computation, only requiring one to store the SA and LCP arrays between computations. For our experiments, we report the times both for constructing the Cartesian tree and for answering the queries (see Table I). The times include parallel times on 40 cores with hyper-threading (T_{40}), times using a single thread (T_1), and the speedup (SU). The code for the Abouelhoda et al. method is not available online so we were unable to obtain timings.

We use the Cartesian tree on the interleaved SA and LCPs of a string to compute for each internal node, the number of leaf nodes in its subtree. This information can be used to answer queries related to repeated substrings, such as the number of repeated substrings of a given length that appear at least x times, or the number of repeated substrings of length at least y.

To compute the number of leaves contained in the subtree of each internal node, we process the Cartesian tree in a bottom-up manner where initially all of the leaves are *active* and each active node passes the number of leaves in its subtree to its parent, which records the sum of these values it receives. Once a node receives values from all of its children, it becomes active and passes its value to its parent. This process is work-efficient but requires time proportional to the height of the tree. The times for this query are shown in the "Leaf counts" column in Table I.

We also use the Cartesian tree to compute the longest substring that appears at least k times in the text. To answer this query, we modify the previous computation to return the deepest node in the tree that has a subtree of at least size k. The times for this query for $k = 10$ are shown in the "Longest substring ($k = 10$)" column in Table I.

Finally, we use our Cartesian tree on the interleaved SA and LCP array of a string to compute the leftmost starting position of any suffix in the subtree of each node. This is useful for computing the Ziv-Lempel decomposition [Ziv and Lempel 1977] of a string, as described in Abouelhoda et al. [2004]. The code for doing this is very similar to computing the number of leaves per subtree. Instead of summing the children's values, each parent takes the minimum value of its children, and leaves start

Table II. Times for Suffix Tree Construction

Text	Size (MB)	Kurtz	Alg 1a T_{40}	Alg 1a T_1	Alg 1a SU	Alg 2 T_{40}	Alg 2 T_1	Alg 2 SU
100Midentical	100	9.53	2.299	41.7	18.14	2.812	44.75	15.91
100Mrandom	100	168.9	3.352	80.6	24.05	3.971	84.2	21.2
100Msqrtn	100	14.52	3.518	55.2	15.69	4.023	57.97	14.41
100MrandomInts10K	100	–	5.24	81.1	15.48	5.774	84.8	14.69
100MrandomIntsImax	100	–	3.88	61.3	15.8	4.141	64.1	15.48
chr22.dna	34.6	24.5	1.469	32.62	22.21	1.728	34.45	19.94
etext99	105	119	4.977	120.3	24.17	5.75	125	21.74
howto	39.4	27.31	1.785	41.02	22.98	2.062	42.87	20.79
jdk13c	69.7	14.69	3.278	78.22	23.86	3.833	81.73	21.33
rctail96	115	55.13	5.61	133.2	23.74	6.34	138.9	21.91
rfc	116	71.77	5.619	133	23.67	6.476	139.2	21.49
sprot34.dat	110	75.11	5.299	126.2	23.82	6.048	131.6	21.76
thesaurus.doc	11.2	8.61	0.485	7.677	15.83	0.564	8.19	14.52
w3c2	104	28.44	5.24	121.2	23.13	5.913	126.1	21.33
wikisamp8.xml	100	31.48	4.808	117.2	24.37	5.612	124.8	22.24
wikisamp9.xml	1000	–	53	1280	24.15	61.88	1339	21.64
HG18.fasta	3083	–	168	3402	20.25	–[†]	–[†]	–[†]

Note: Comparison of running times (seconds) of Kurtz's sequential algorithm and our algorithms for suffix tree construction on different inputs on a 40 core machine with hyper-threading. T_{40} is the time using 40 cores (80 hyper-threads) and T_1 is the time using a single thread. SU is the speedup computed as T_1/T_{40}. [†]We do not have times for Algorithm 2 on HG18.fasta since for this file, it uses more memory than the machine has available.

with a value equal to the starting position of their corresponding suffix in the original string. The times for this query are shown in the "Leftmost suffix positions" column in Table I.

For most real-world strings, the height of the Cartesian tree of the interleaved SA and LCPs is not very large and these three applications get good speedup. As expected, this process does not get much speedup for the all-identical string, whose tree has linear height (the slight speedup comes from the preprocessing and postprocessing steps). For the real-world strings, the cost of building the Cartesian tree is just about twice the cost of the query, which makes it reasonable to store just the SA and LCP arrays and build the Cartesian tree on-the-fly when performing a query. Other queries, such as finding maximal repeated pairs [Abouelhoda et al. 2004] and finding the longest common substring of two strings [Gusfield 1997] can also be computed by a bottom-up traversal of the Cartesian tree.

5.5. Suffix Trees

We use Algorithms 1a and 2 along with our suffix array code and hash table insertion to generate suffix trees on strings. Table II presents the runtimes for generating the suffix tree based on Algorithm 1a, our variant of Algorithm 2, and Kurtz's code. For the implementations based on Algorithm 1a and Algorithm 2, we give both sequential (single thread) running times (T_1) and parallel running times on 40 cores with hyper-threading (T_{40}). The speedup (SU) is computed as T_1/T_{40}. We note that the speedup ranges from 14 to 24. Compared to Kurtz's code, our code running sequentially is between 2.1x faster and 5.3x slower. Our parallel code, however, is always faster than Kurtz's code and up to 50x faster. Comparatively, Kurtz's code performs best on strings with lots of regularity (e.g. the all-identical string). This is because the incremental

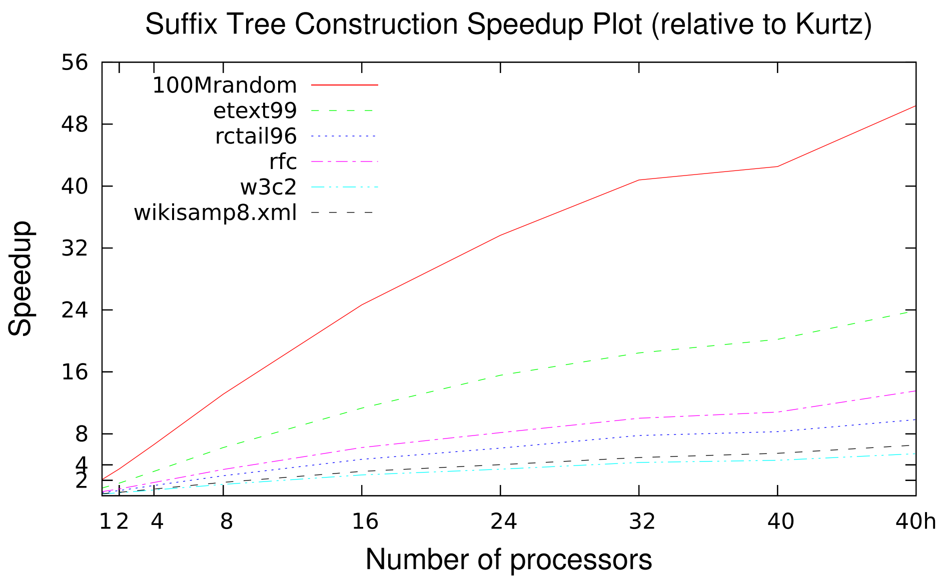

Fig. 4. Speedup of Algorithm 1a relative to Kurtz's sequential algorithm on a 40 core machine. "40h" indicates 80 hyper-threads.

sequential algorithms based on McCreight's algorithm are particularly efficient on these strings. The runtime for our code is affected much less by the type of input string. Kurtz's code only supports reading in text files with fewer than 537 million characters, so we were unable to obtain timings for wikisamp9.xml (1 billion characters) and HG18.fasta (3.08 GB). Also since the code reads the input files as ASCII (alphabet size of 128), we do not have timings for integer files with larger alphabet sizes. We plot the speedup of Algorithm 1a relative to Kurtz's sequential algorithm on various inputs in Figure 4. The speedup varies widely based on the input file, with as much as 50x speedup for 100Mrandom and as little as 5.4x speedup for w3c2.

Figures 5 and 6 show the breakdown of the times for our implementations of Algorithm 1a and Algorithm 2, respectively, when run on 40 cores with hyper-threading. In Figure 5, "Cartesian tree" refers to the time to construct the binary Cartesian tree and "Grouping internal nodes" refers to the time to convert to a multiway Cartesian tree. In Figure 6, "ANSV" is the time to compute the nearest smaller neighbors in the LCP array, and "Compute parents" is the time to select a smaller neighbor to be the parent. "Create internal nodes" does an integer sort to create the internal nodes of the tree. In both figures, "Hash table insertion" is the time to create a hash table for downward traversal, and completes the suffix tree construction. Figure 7 shows the breakdown of the time for generating the suffix array and LCP array. For Algorithm 1a, more than 80% of the total time is spent in generating the suffix array, less than 10% in inserting into the hash table and less than 5% on generating the Cartesian tree from the suffix array (the code shown in Figure 1). For Algorithm 2, we note that the ANSV portion takes less than 2% of the total time even though it is an $O(n \log n)$ work algorithm. Improvements to the suffix array or hash table code will likely lead to an improvement in the overall code performance. Figure 8 shows the performance of Algorithm 1a in terms of characters per second on random character strings of varying sizes. We observe that the ratio remains nearly constant as we increase the input size, indicating good scalability. While our implementation of

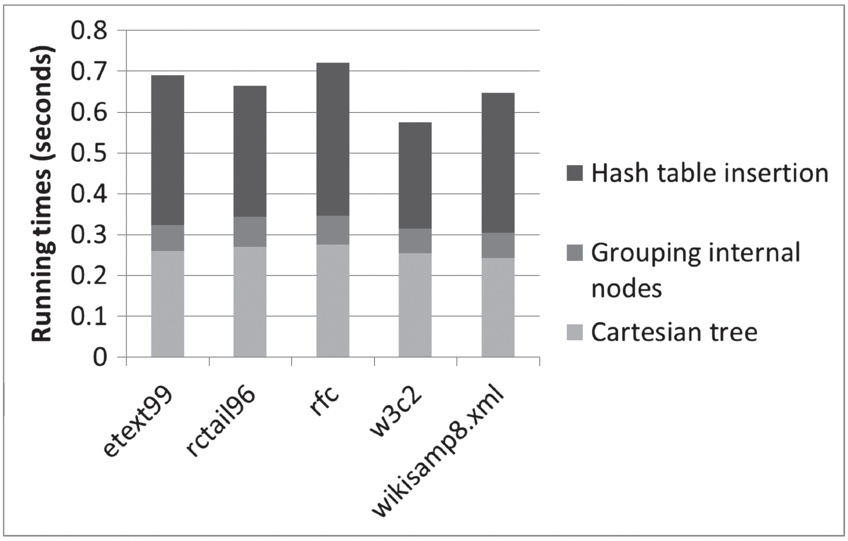

Fig. 5. Breakdown of running times for converting a suffix array to a suffix tree using Algorithm 1a on 40 cores with hyper-threading.

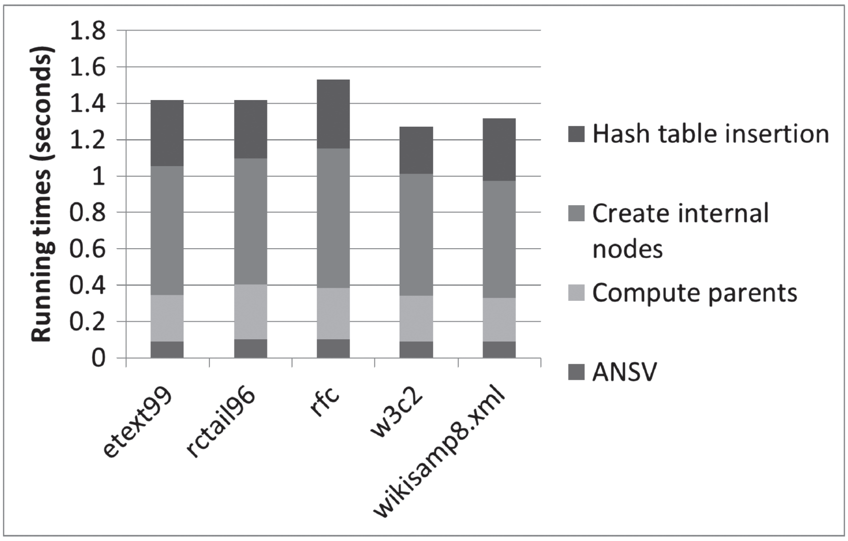

Fig. 6. Breakdown of running times for the suffix tree portion of Algorithm 2 on 40 cores with hyper-threading.

Algorithm 1a is not truly parallel, it is incredibly straightforward and performs better than Algorithm 2.

5.6. Searching the Suffix Tree

We present times for performing existential queries (searching) for random substrings in the suffix trees of several texts constructed using our code and Kurtz's code. We also report times for searches using the suffix array code of Manber and Myers [1993], as Abouelhoda et al. [2004] show that this code (*mamy*) performs searches more quickly

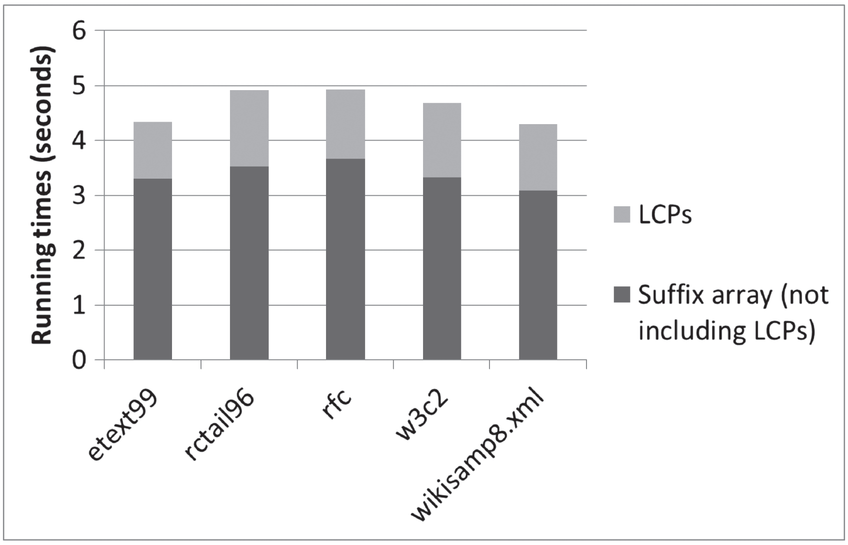

Fig. 7. Breakdown of running times for the suffix array portion of Algorithm 1a and Algorithm 2 on 40 cores with hyper-threading.

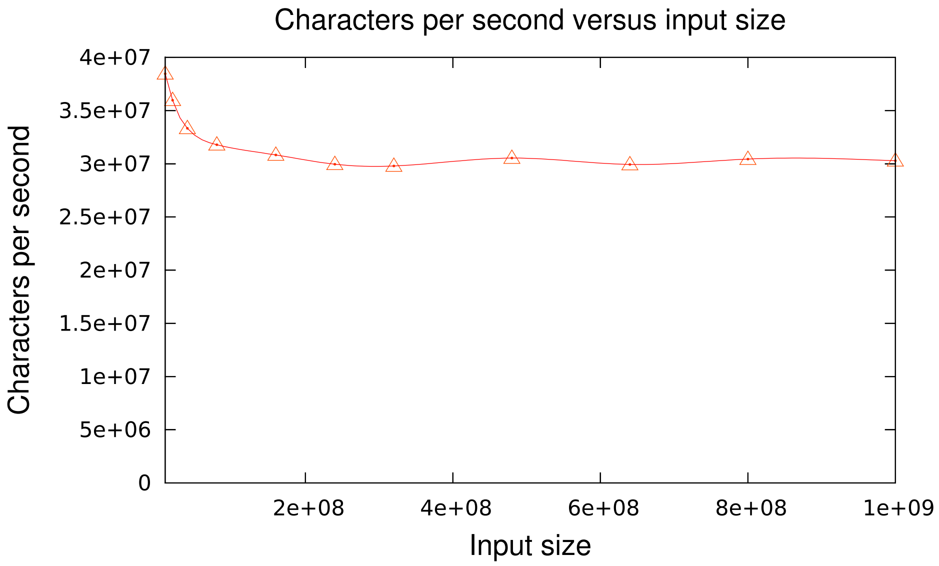

Fig. 8. Performance (characters per second) of Algorithm 1a on random character strings of varying sizes on 40 cores with hyper-threading.

than Kurtz's code does. The suffix array code uses the LCP array and answers queries in $O(m + \log n)$ time, where m is the length of the pattern. For each text, we search 500,000 random substrings of the text (these should all be found) and 500,000 random strings (most of these will not be found) with lengths uniformly distributed between 1 and 50. For all searches, the starting position in the text of the search string is reported if found.

Existential query times are reported in Table III. Searches done in our code are on integers, while those done in Kurtz's code and Myer and Manber's code (*mamy*) are

Table III. Times for String Searching

Text	Alg 1a T_{40}	Alg1a T_1	Alg1a SU	Kurtz T_1	mamy T_1
100Mrandom	0.017	0.78	45.88	1.65	1.05
etext99	0.019	0.9	47.37	6.32	1.38
sprot34.dat	0.014	0.681	48.64	3.29	1.3

Note: Comparison of times (seconds) for searching (existential queries) 1,000,000 strings of lengths 1 to 50 on a 40 core machine with hyper-threading. T_{40} is the time using 40 cores (80 hyper-threads) and T_1 is the time using a single thread. SU is the speedup computed as T_1/T_{40}.

Table IV. Space Requirements

Component	Space (number of bytes)
Computing SA + LCP	$32n$
SA + LCP data structure	$8n$
Node initialization	$24n$
Building Cartesian Tree	$16n$
Cartesian Tree data structure	$16n$
Finding roots	$16n$
Hash table insertion	$31n$
Suffix Tree data structure	$29n$

Note: Space requirements for the different components of Algorithm 1a.

done on characters, which puts us at a slight disadvantage. We report both sequential and parallel search times for our algorithm. Our results show that sequentially, our code performs searches faster than Kurtz's code (2.1–7x) and *mamy* (1.3–1.9x). Abouelhoda et al. [2004] report being 1.2–1.7x faster than *mamy* for searches on strings with small alphabets, but are up to 16x slower than *mamy* on larger alphabets. In contrast, our search performance does not degrade with increasing alphabet size, since we use a hash table to store children of internal nodes.

The layout of our nodes in memory is in suffix array order, so listing occurrences can also be done in a cache-friendly manner by scanning the nearby nodes, similar to *mamy*.

5.7. Space Requirements

Since suffix trees are often constructed on large texts (e.g. the human genome), it is important to keep the space requirements minimal. As such, there has been related work on compactly representing suffix trees [Abouelhoda et al. 2004; Giegerich et al. 2003; Gog and Ohlebusch 2013; Navarro and Mäkinen 2007; Sadakane 2007]. Our suffix tree uses 3 integers per node (leaf and internal) and about $5n$ bytes for the hash table, which totals to about $29n$ bytes. This compares to about $12n$ bytes for Kurtz's code, which has been optimized for space [Delcher et al. 2002; Kurtz 1999]. Table IV shows the space requirements (in bytes) for the different portions and data structures of our implementation. We leave further optimization of the space requirements of our implementation to future work.

6. CONCLUSIONS

We have described a linear work, linear space, and $O(\log^2 n)$ time parallel algorithm for constructing a Cartesian tree and a multiway Cartesian tree. In conjunction with a parallel suffix array algorithm, we can use our algorithm to generate a suffix tree in linear work with $O(\log^2 n)$ time with high probability for constant-sized alphabets and $O(n^{\epsilon})$ time $(0 < \epsilon < 1)$ for the integer alphabet $[n]$. Our approach is much simpler than previous approaches for generating suffix trees in parallel and can handle arbitrary alphabets. We implemented our algorithm, and showed that it achieves good speedup and outperforms existing suffix tree construction implementations on a shared memory multi-core machine. We also describe three applications that can be implemented with just the Cartesian tree, and show that they perform well on real-world inputs.

REFERENCES

M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. 2004. Replacing Suffix Trees with enhanced suffix arrays. *J. Discrete Alg. 2*, 1, 53–86.

A. Apostolico, C. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin. 1988. Parallel construction of a suffix tree with applications. *Algorithmica 3*, 1–4, 347–365.

M. Barsky, U. Stege, and A. Thomo. 2010. A survey of practical algorithms for suffix tree construction in external memory. *Softw. Pract. Exper. 40*, 11, 965–988.

O. Berkman, B. Schieber, and U. Vishkin. 1993. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *J. Algor. 14*, 3, 344–370.

G. E. Blelloch and J. Shun. 2011. A simple parallel cartesian tree algorithm and its application to suffix tree construction. In *Proceedings of the Workshop on Algorithm Engineering and Experiments*. 48–58.

M. Burrows and D. J. Wheeler. 1994. *A Block-Sorting Lossless Data Compression Algorithm*. Tech. rep. HP Labs.

M. Comin and M. Farreras. 2013. Efficient parallel construction of suffix trees for genomes larger than main memory. In *Proceedings of the 20th European MPI Users' Group Meeting*. 211–216.

A. Delcher, A. Phillippy, J. Carlton, and S. Salzberg. 2002. Fast Algorithms for large-scale genome alignment and comparision. *Nucl. Acids Res. 30*, 11, 2478–2483.

M. Farach and S. Muthukrishnan. 1996. Optimal logarithmic time randomized suffix tree construction. In *Proceedings of the International Colloquium on Automata Languages and Programming*. 550–561.

J. Fischer and V. Heun. 2006. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proceedings of the 17th Annual Conference on Combinatorial Pattern Matching*. 36–48.

H. Gabow, J. Bentley, and R. Tarjan. 1984. Scaling and related techniques for geometry problems. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*. 135–143.

A. Ghoting and K. Makarychev. 2009. Indexing genomic sequences on the IBM Blue Gene. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* 1–11.

R. Giegerich, S. Kurtz, and J. Stoye. 2003. Efficient implementation of lazy suffix trees. *Software: Pract. Exper. 33*, 11, 1035–1049.

S. Gog and E. Ohlebusch. 2013. Compressed suffix trees: Efficient computation and storage of LCP-values. *J. Exp. Algorithmics 18*, Article 2.1.

D. Gusfield. 1997. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press.

T. Hagerup and C. Rüb. 1989. Optimal merging and sorting on the EREWPRAM. *Inf. Process. Lett. 33*, 4, 181–185.

R. Hariharan. 1994. Optimal parallel suffix tree construction. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*. 290–299.

C. Iliopoulos and W. Rytter. 2004. On parallel transformations of suffix arrays into suffix trees. In *Proceedings of the 15th Australasian Workshop on Combinatorial Algorithms (AWOCA)*.

J. Jaja. 1992. *Introduction to Parallel Algorithms*. Addison-Wesley Professional.

J. Kärkkäinen, G. Manzini, and S. J. Puglisi. 2009. Permuted longest-common-prefix array. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 181–192.

J. Kärkkäinen, P. Sanders, and S. Burkhardt. 2006. Linear work suffix array construction. *J. ACM 53*, 6, 918–936.

T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. 2001. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 181–192.

F. Kulla and P. Sanders. 2007. Scalable parallel suffix array construction. *Parallel Comput. 33*, 9, 605–612.

S. Kurtz. 1999. Reducing the space requirement of suffix trees. *Softw. Pract. Exper. 29*, 13, 1149–1171.

S. Kurtz and C. Schleiermacher. 1999. REPuter: Fast computation of maximal repeats in complete genomes. *Bioinformatics, 15*, 5, 426–427.

C. E. Leiserson. 2010. The Cilk++ Concurrency Platform. *J. Supercomput. 51*, 3, 244–257.

U. Manber and E. W. Myers. 1993. Suffix Arrays: A new method for on-line string searches. *SIAM J. Comput. 22*, 5, 935–948.

E. Mansour, A. Allam, S. Skiadopoulos, and P. Kalnis. 2011. ERA: Efficient serial and parallel suffix tree construction for very long strings. *Proc. VLDB Endow. 5*, 1, 49–60.

E. M. McCreight. 1976. A space-economical suffix tree construction algorithm. *J. ACM 23*, 2, 262–272.

C. Meek, J. M. Patel, and S. Kasetty. 2003. OASIS: An online and accurate technique for local-alignment searches on biological sequences. In *Proceedings of the 29th International Conference on Very Large Data Bases*. VLDB Endowment, 910–921.

Y. Mori. 2010a. libdivsufsort: A lightweight suffix-sorting library. http://code.google.com/p/libdivsufsort.

Y. Mori. 2010b. SAIS: An implementation of the induced sorting algorithm. http://sites.google.com/site/yuta256/sais.

D. R. Morrison. 1968. PATRICIA - Practical algorithm to retrieve information coded in alphanumeric. *J. ACM 15*, 4, 514–534.

G. Navarro and V. Mäkinen. 2007. Compressed full-text indexes. *ACM Comput. Surv. 39*, 1.

B. Phoophakdee and M. Zaki. 2007. Genome-scale disk-based suffix tree indexing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 833–844.

B. Phoophakdee and M. Zaki. 2008. Trellis+: An effective approach for indexing genome-scale sequences using suffix trees. In *Proceedings of the Pacific Symposium on Biocomputing*. 90–101.

C. K. Poon and H. Yuan. 2013. A faster CREW PRAM algorithm for computing Cartesian trees. In *Proceeding of the International Conference on Algorithms and Complexity*. 336–344.

S. J. Puglisi, W. F. Smyth, and A. H. Turpin. 2007. A taxonomy of suffix array construction algorithms. *Comput. Surveys 39*, 2.

S. Rajasekaran and J. H. Reif. 1989. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput. 18*, 3, 594–607.

M. Reid-Miller, G. L. Miller, and F. Modugno. 1993. List ranking and parallel tree contraction. In *Synthesis of Parallel Algorithms*, Chapter 3, 115–194.

K. Sadakane. 2007. Compressed suffix trees with full functionality. *Theory of Comput. Syst. 41*, 4, 589–607.

S. Sahinalp and U. Vishkin. 1994. Symmetry breaking for suffix tree construction. In *Proceedings of the 26th Annual ACM symposium on Theory of Computing*. 300–309.

J. Shun and G. E. Blelloch. 2014. Phase-concurrent hash tables for determinism. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*. 96–107.

J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. Vardhan Simhadri, and K. Tangwongsan. 2012. Brief announcement: The problem based benchmark suite. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*. 68–70.

J. Shun and F. Zhao. 2013. Practical parallel Lempel-Ziv factorization. In *Proceedings of the 2013 Data Compression Conference*. IEEE Computer Society, 123–132.

D. Tsadok and S. Yona. 2003. ANSI C Implementation of a Suffix Tree. http://mila.cs.technion.ac.il/~yona/suffix_tree/.

D. Tsirogiannis and N. Koudas. 2010. Suffix tree construction algorithms on modern hardware. In *Proceedings of the 13th International Conference on Extending Database Technology*. 263–274.

E. Ukkonen. 1995. On-line Construction of Suffix Trees. *Algorithmica 14*, 3, 249–260.

J. Vuillemin. 1980. A unifying look at data structures. *Commun. ACM 23*, 4, 229–239.

P. Weiner. 1973. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*. 1–11.

J. Ziv and A. Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory 23*, 3, 337–343.

Received April 2013; revised July 2014; accepted August 2014