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

Guy E. Blelloch† Julian Shun‡

Abstract

We present a simple linear work and space, and polylogarithmic time parallel algorithm for generating multiway Cartesian trees. As a special case, the algorithm can be used to 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 algorithm for generating suffix trees over an integer alphabet $\Sigma \subseteq [1, \ldots, n]$, where n is the length of the input string. More generally, given a sorted sequences of strings and the longest common prefix lengths between adjacent elements, the algorithm will generate a pat tree (compacted trie) over the strings.

We also present experimental results comparing the performance of the algorithm to existing sequential implementations and a second parallel algorithm. We present comparisons for the Cartesian tree algorithm on its own and for constructing a suffix tree using our algorithm. 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.

1 Introduction

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 pat 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 longest common substring, maximal repeats, longest repeated substrings, and longest palindrome, among many others [Gus97]. 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 [KS99], MUMmer [DPCS02], OASIS [MPK03] and Trellis+ [PZ08]. Both suffix trees and a linear time algorithm for constructing them were introduced by Weiner [Wei73] (although he used the term position tree). Since then various similar constructions have been described [McC76] 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 algorithm of Weiner and its derivatives are all incremental and inherently sequential. The first parallel algorithm for suffix trees was given by Apostolico et. al. [AIL+88] 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 superlinear memory (by a polynomial factor). The parallel construction of suffix trees was later improved to linear work and space by Hariharan [Har94], with an algorithm taking $O(\log^4 n)$ time on the CREW PRAM, and then by Farach and Muthukrishnan to $O(\log n)$ time using a randomized CRCW PRAM [FM96] (high-probability bounds). These later results are for a constant-sized alphabet, are "considerably non-trivial", and do not seem to be amenable to efficient implementations.

One way to construct a suffix tree is to first generate a suffix array (an array of pointers to the lexicographically sorted suffixes), and then convert it to a suffix tree. For binary alphabets and given the length of the longest common prefix (LCP) between adjacent entries this conversion can be done sequentially by generating a Cartesian tree in linear time and space. The approach can be generalized to arbitrary alphabets using multiway Cartesian trees without much difficulty. Using suffix arrays is attractive since in recent years there has been considerable theoretical and practical advances in the generation of suffix arrays (see e.g. [PST07]). The

*This work was supported by generous gifts from Intel, Microsoft and IBM, and by the National Science Foundation under award CCF1018188.

†Carnegie Mellon University, E-mail: guyb@cs.cmu.edu

‡Carnegie Mellon University, E-mail: jshun@cs.cmu.edu

[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.

interest is partly due to their need in the widely used Burrows-Wheeler compression algorithm, 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 Karkkainen and Sanders have developed a quite simple and efficient parallel algorithm for suffix arrays [KS03, KS07] that can also generate LCPs.

The story with generating Cartesian trees in parallel is less satisfactory. Berkman et. al [BSV93] describe a parallel algorithm for the all nearest smaller values (ANSV) problem, which can be directly used to generate a binary Cartesian tree for fixed sized alphabets. However, it cannot directly be used for non-constant sized alphabets, and the algorithm is very complicated. Iliopoulos and Rytter [IR04] 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 paper 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 CREW PRAM. The second version is only slightly more complicated and runs in $O(\log^2 n)$ time on the CREW PRAM. They both use linear work and space.

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 [KS03] on a CRCW PRAM we have $O(\log^2 n)$ time for constant-sized alphabets and $O(n^\epsilon), 0 < \epsilon \leq 1$ time 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 open problem.

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 $O(I(n) + \log n)$ time on the CRCW PRAM, where $I(n)$ is the best time bound for a linear-work stable sorting of integers from $[n]$. We then show that the Cartesian tree can be used to generate an ANSV in linear work and polylogarithmic time.

We have implemented the first version of our algorithm and present various experimental results analyzing our algorithm on a shared memory multicore parallel machine on a variety of inputs. First, we compare our Cartesian tree algorithm with a simple stack based sequential implementation. On one core our algorithm is about 3x slower, but we achieve about 30x speedup on 32 cores and about 45x speedup with 32 cores using hyperthreading (two threads per core). We also analyze the 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 existing sequential implementations of suffix trees. Our algorithm is always faster than the ANSV algorithm. The algorithm is competitive with the sequential code on a single processor (core), and achieves good speedup on 32 cores. Finally, we present timings for searching multiple strings in the suffix tree we generate. Our times are always faster than the sequential suffix tree on one core and always more than 50x faster on 32 cores using hyperthreading.

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 *pat tree* [Mor68] (or patricia tree, 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 pat tree.

In this paper we assume an integer alphabet $\Sigma \subseteq [n]$ where n is the total number of characters. We require that the pat tree and suffix tree supports 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 sibling in the character order, and finding the parent. If the alphabet is constant sized all these operations can easily be implemented in constant worst-case time.

A *Cartesian tree* [Vui80] 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 inorder traversal of the tree defines the sequence order. If 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_1,s_2) be the length of the longest common prefix of s_1 and s_2. 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 (i.e. $[|s_1|, LCP(s_1, s_2), |s_2|, \ldots, LCP(s_{n-1}, s_n), |s_n|]$) the corresponding multiway Cartesian tree has the structure of the pat tree for S. The pat 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. In summary, beyond some trivial operations, generating a multiway Cartesian tree is sufficient for converting a suffix array and corresponding LCPs to a suffix tree.

In this paper, we use the concurrent-read exclusive-write 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 and total number of operations. Using Brent's WT scheduling principle, this implies an $O(T)$ time bound using W/T processors, as long as processors can be allocated efficiently [Jaj92].

3 Parallel Cartesian Trees

We describe a work-efficient parallel divide-and-conquer algorithm for creating a Cartesian tree. The algorithm works recursively by splitting the input array A into two, 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 [IR04], 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 the $O(n)$ 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 4-15), 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 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 don't 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 inorder 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. Therefore any descendants in the right tree will correctly appear after in the inorder traversal. Similarly in the other direction any left tree descendants of a right tree node will correctly appear before in the inorder traversal. 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 and $O(n)$ space on the RAM.*

Proof. We use the following definitions to help with proving the complexity bounds of our algorithm: A node

```
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);}
```

Figure 1: C code for Algorithm 1a. The `cilk_spawn` and `cilk_sync` declarations are part of the Cilk++ parallel extensions [Int10] and allow the two recursive calls to run in parallel.

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 always only 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 $2n$ for the lastnodeRight or lastnodeLeft on the $n-1$ merges, and at most n for the nodes that become protected for a total work of $O(n)$. Since each node only maintains a constant amount of data, the space required is $O(n)$.

Note that 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. We now describe a fully parallel variant, 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 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 or equal to the largest value in the first and joins them into one. Both 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 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

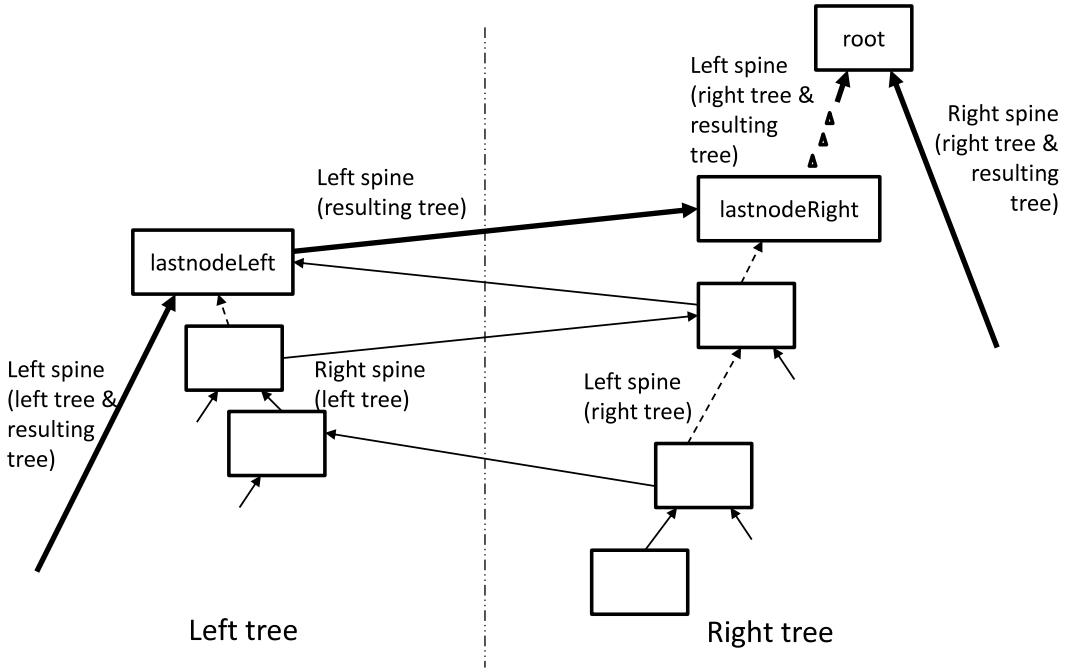

Figure 2: Merging two spines. Thick lines represent the *spines* of the resulting tree; dashed lines represent edges that existed before the merge but not after the merge; dotted edges represent an arbitrary number of nodes; all non-dashed lines represent edges in the resulting tree.

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 thrown out 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 CREW 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 joints 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 CREW PRAM [Jaj92], 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 the sequential version, with the exception that splits and joins in the spine cost an extra $O(\log n)$ per merge, so 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 follows. Processor allocation is straightforward due to the $O(\log n)$ time for the merges.

LEMMA 3.1. *The output of Algorithm 1a can be used*

to construct a multiway Cartesian tree in $O(n)$ work on the RAM. Algorithm 1b can perform the same task with $O(n)$ work and $O(\log n)$ time on the EREW PRAM.

Proof. For Algorithm 1a, this can be done using 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 takes linear work. For Algorithm 1b, substitute path compression with a parallel tree contraction algorithm. Parallel tree contraction can be done in $O(n)$ work and $O(\log n)$ time on the EREW PRAM [Jaj92].

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 w.h.p.) on a randomized CRCW PRAM.

COROLLARY 3.1. *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 using Algorithm 1b, tree contraction, and hash table insertion, in $O(n)$ work, $O(\log^2 n)$ time, and $O(n)$ space on the randomized CRCW PRAM.*

Proof. Follows directly.

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 numbers, 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 [BSV93] to support multiway Cartesian trees, and also show how to use Cartesian trees to solve the ANSV problem.

Berkman et. al.'s algorithm 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, 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 we need 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 know 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 experimentally. In related work Iliopoulos and Rytter [IR04] present an $O(n \log n)$ work polylogarithmic time algorithm based on a variant of 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 it has them) 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 all left nearest neighbors of all nodes. A symmetric algorithm returns the right nearest neighbors.

For every node, maintain two variables, *node.index* which is set to the node's index in the sequence corresponding to the inorder traversal of the Cartesian tree and never changed, and *node.inherited*, which is initialized to *null*.

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.

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.

We now present a linear-work and $O(\log n)$ time parallel algorithm for this task based on tree contraction [Jaj92]. To find the left neighbors, we describe how to decompress the tree several configurations for rake and compress operations, and the rest of the configurations have a symmetric argument. For compress, there is the left-left and right-left configurations. For the left-left configuration, the compressed node takes the *inherited* value of its parent. For the right-left configuration, the compressed node takes the *index* value

of its parent. For raking a left leaf, the raked leaf takes the *inherited* value of its parent, and for raking a right leaf, the raked leaf takes the *index* value of its parent. We can find the right neighbors by symmetry.

5 Experiments

The goal of our experiments is to analyze the effectiveness of our Cartesian tree algorithm both on its own and also as part of code to generate suffix trees. We compare the algorithm to the ANSV-based algorithm and to the best available sequential code for suffix trees. We feel it is important to compare our parallel algorithm with existing sequential implementations to make sure that that the algorithm can significantly outperform existing sequential ones even on a modest number 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 suffix arrays and their corresponding LCPs, and parallel code for inserting the tree nodes into a hash table to allow for efficient search. We ran our experiments on a 32-core parallel machine using a variety of real-world and artificial strings.

5.1 Auxiliary Code. To generate the suffix array and LCP we implemented a parallel version of the skew algorithm [KS03]. The implementation uses a parallel bucket sort [Jaj92], requiring $O(n)$ work and $O(n^{1/c})$ time for some constant $c \geq 1$. Our LCP code is based on an $O(n \log n)$ solution for the range minima problem instead of the optimal $O(n)$. We did implement a parallel version of the linear-time range-minima algorithm by [FH06], 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 [KS03] implemented in the paper and available online, even though that version does not implement the LCP. We get a further 9 to 17 fold speedup on a 32-core machine. Compared to the parallel implementation of suffix arrays by Kulla and Sanders [KS07], our times are faster on 32 cores then the 64 core numbers reported by them (31.6 seconds vs 37.8 seconds on 522 Million characters), although their clock speed is 33% 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 [Mor10a], but we found that it is slower than their corresponding sequential implementation, even when using 32 cores. 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 [PST07, Mor10a, Mor10b], but most of them do not compute the LCPs. For real-world texts, those programs are faster than our code by a factor of between 3 and 6. This is due to many optimizations these codes make. We expect 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 improvements made to parallel suffix arrays would improve the performance of the suffix tree code.

We use a parallel hash table to allow for fast search in the suffix tree. The hash-table uses a compare and swap for concurrent insertion. Furthermore we optimized the code so that most entries near leafs 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 k (16 in the experiments) in the array we do not store it in the hash table but use a linear search.

For Algorithm 2, we implemented the $O(n \log n)$ work and $O(\log n)$ time ANSV algorithm of [BSV93] instead of the much more complicated work optimal version. However, we used an optimization that makes it run significantly faster in practice. In particular, using a doubling search, each left query on location i runs in time $O(\log(k - j + 1))$ where k is the resulting nearest lesser value and $j = \max\{l \in [1, \ldots, i-1] | v[l] < v[l+1]\}$, and symmetrically for right queries.

5.2 Experimental Setup. We performed experiments on a 32-core (with hyper-threading) Dell PowerEdge 910 with 4×2.26GHZ Intel 8-core X7560 Nehalem Processors, a 1066MHz bus, and 64GB of main memory. The parallel programs were compiled using the `cilk++` compiler (build 8503) 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 Tsadok's and Yona's [TY03] code and Kurtz's code from the MUMmer project [DPCS02, Kur99] (`http://mummer.sourceforge.net`),both of which are publicly available. We only list the results of Kurtz because they are superior to those of [TY03] for all of our test files. Kurtz's code is based on McCreight's suffix tree construction algorithm [McC76]—it is inher-

ently sequential and completely different from our algorithms. Other researchers have experimented with building suffix trees in parallel [GM09], but due to difficulty in obtaining the source code, we do not have results for this implementation. However, our running times appear significantly faster than those reported in the corresponding papers, even after accounting for differences in machine specifications.

For running the experiments we used a variety of strings available online (`http://people.unipmn.it/manzini/lightweight/corpus/`), a Microsoft Word document (thesaurus.doc), XML code from wikipedia samples (wikisamp8.xml and wikisamp9.xml), and artificial inputs. We also included two files of integers, 10Mrandom-ints10K, with random integers ranging from 1 to 10^3, and 10Mrandom-intsImax, with random integers ranging from 1 to 2^{31}, to show that our algorithms run efficiently on arbitrary integer alphabets.

We present times for 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 Myer and Manber's suffix array code [MM90] as Abouelhoda et.al. show that this code (*mamy*) performs searches more quickly than Kurtz's code does. For each text, we search 500,000 random substrings of the text (these should all be found) and 500,000 random strings (most will not be found) with lengths uniformly distributed between 1 to 25. The file etext99* consists of the etext99 data with special characters removed (*mamy* does not work with special characters), and its size is 100.8MB. The file 100Mrandom contains 10^8 random characters.

5.3 Cartesian Trees. We experimentally compare our Cartesian tree algorithm from Algorithm 1 to the linear-time stack-based sequential algorithm of [GBT84]. 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 running time vs. number of processors for both our algorithm and the stack-based one in log-log scale. Our parallel algorithm achieves nearly linear speedup and outperforms sequential algorithms with 4 or more cores.

5.4 Suffix Trees. We use Algorithm 1a and 2 along with our suffix array code and hash insertion to generate suffix trees from strings. Table 1 presents the runtimes for generating the suffix tree based on Algorithm 1a, our variant of Algorithm 2, and using Kurtz's code. For the implementations based on algorithm 1a and 2 we give both sequential (single thread) running times and parallel running times on 32 cores with hyper-

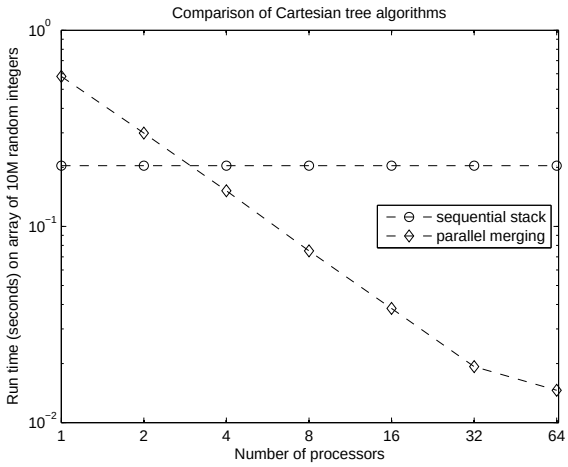

Figure 3: Running times for Cartesian tree algorithms on 10M random characters on 32-core machine in log-log scale. The 64-core time is actually 32-cores using hyper-threading.

threading. We note that the speedup ranges from 9 to 19, with the worst speedup on the string of equal characters. Figure 4 shows runtime as a function of number of cores for 10 million random characters (10Mrandom). Compared to Kurtz's code, our code running sequentially is between 1.4x faster and 5x slower. Our parallel code, however, is always faster than Kurtz's code and up to 27x faster. Comparatively, Kurtz's code performs best on strings with lots of regularity (e.g. the all identical string). This is because the incremental sequential algorithms based on Weiner's algorithm are particularly good on these strings. The runtime for our code is affected much less by the type of input string.

Figures 5 and 6 show the breakdown of the times for our implementations of Algorithm 1a and Algorithm 2 respectively when run on 32 cores with hyper-threading. Figure 7 shows the breakdown of the time for generating the suffix array and LCP array. For Algorithm 1a about 85% of the total time is spent in generating the suffix array, about 10% in inserting into the hash table and less than 5% on generating the Cartesian tree from the suffix array (i.e., the code shown in Figure 1). For Algorithm 2 we note that the ANSV only takes about 10% of the total time even though it is an $O(n \log n)$ algorithm. This is likely due to the optimization discussed above. Since we did not spend a lot of time optimizing the suffix array or hash code, we expect the overall code performance could be improved significantly. Figure 8 shows the running time of Algorithm 1a on random

Text	Size (MB)	Kurtz	Alg 1a 32-core	Alg 1a seq.	Alg 1a Speedup	Alg 2 32-core	Alg 2 seq.	Alg 2 Speedup
10Midentical	10	1.04	0.46	4.34	9.4	1.32	9.64	7.66
10Mrandom	10	10.6	0.39	7.58	19.4	1.01	11.9	11.8
$(a^{\sqrt{10^7}}b)^{\sqrt{10^7}}$	10	1.69	0.54	5.78	10.7	1.36	11.0	8.1
chr22.dna	34.6	31.1	2.23	37.4	16.8	5.22	53.9	10.3
etext99	105	128	8.71	135	15.5	15.7	191	12.2
howto	39.4	33.1	2.90	45.8	15.8	5.88	64.1	11.0
jdk13c	69.7	18.1	5.30	90.2	17.0	11.0	125	11.6
rctail96	115	65.5	9.54	153	16.0	17.1	216	11.4
rfc	116	89.6	8.96	150	16.7	17.4	220	12.6
sprot34.dat	110	89.8	9.42	143	15.2	15.2	202	13.3
thesaurus.doc	11.2	10.7	0.72	9.77	13.6	1.39	14.4	10.4
w3c2	104	33.4	8.87	138	15.6	15.8	201	12.7
wikisamp8.xml	100	35.5	8.42	128	15.2	13.5	173	12.8
wikisamp9.xml	1000	—	118	1840	15.6	—	—	—
10MrandomInts10K	10	—	0.44	7.74	17.6	0.91	10.1	11.1
10MrandomIntsImax	10	—	0.41	5.96	14.5	0.73	8.18	11.2

Table 1: Comparison of running times (seconds) of MUMmer's sequential algorithm and our algorithms for suffix tree construction on different inputs on 32-core (with hyper-threading) Dell PowerEdge 910 with 4×2.26GHZ Intel 8-core X7560 Nehalem Processors and a 1066MHZ bus.

character files of varying sizes. We note that the running time is superlinear in relation to the file size, and hypothesize that this is due to memory effects. While our implementation of Algorithm 1a is not truly parallel, it is incredibly straightforward and performs better than Algorithm 2.

Search times are reported in Table 2. Searches done in our code are on integers, while those done in Kurtz's code and Myer and Manber's code are done on characters. We report both sequential and parallel search times for our algorithm. Our results show that even sequentially, our code performs searches faster than the other two implementations do. Our code is likely to perform even better if we modified it to search on characters instead of integers.

5.5 Space Requirements. Since suffix trees are often constructed on large texts (e.g. human genome), it is important to keep the space requirements minimal. As such, there has been related work on compactly representing suffix trees [AKO03, Sad07]. Our implementations of Algorithm 1a and Algorithm 2 use 3 integers per node (leaf and internal) and about $5n$ bytes for the hash table. This totals to about $29n$ bytes. This compares to about $12n$ bytes for Kurtz's code, which has been optimized for space [DPCS02, Kur99]. We leave further optimization of space of our code to future work.

6 Conclusions

We have described a work-optimal parallel algorithm with $O(\log^2 n)$ time for constructing a Cartesian tree and a multiway Cartesian tree. By using this algorithm, we are able to transform a suffix array into a suffix tree in linear work and $O(\log^2 n)$ time. Our approach is much simpler than previous approaches for generating suffix trees in parallel and can handle arbitrary alphabets. We implemented of our algorithm, and showed that it achieves good speedup and outperforms existing suffix tree construction implementations on small scaled multicore processors.

References

[AIL+88] A. Apostolico, C. Iliopoulos, G. Landau, B. Schieber, and U. Vishkin, *Parallel construction of a suffix tree with applications*, Algomthmica **3** (1988), 347–365.

[AKO03] M. Abouelhoda, S. Kurtz, and E. Ohlebusch, *Replacing suffix trees with enhanced suffix arrays*, Journal of Discrete Algorithms **2** (2003), 53–86.

[BSV93] O. Berkman, B. Schieber, and U. Vishkin, *Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values*, Journal of Algorithms **14** (1993), 371–380.

[DPCS02] A. Delcher, A. Phillippy, J. Carlton, and S. Salzberg, *Fast algorithms for large-scale genome*

Text	Alg 1a seq.	Alg1a 32-core	Kurtz seq.	mamy seq.
100Mrandom	1.08	0.03	1.86	1.78
etext99*	1.24	0.03	7.67	1.73
sprot34.dat	0.96	0.02	3.75	1.73

Table 2: Comparison of times (seconds) for searching 1,000,000 strings of lengths 1 to 25 on 32-core machine with hyper-threading.

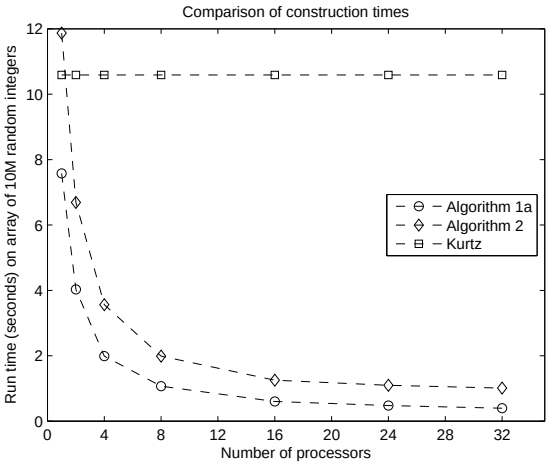

Figure 4: Running times for the different algorithms using varying numbers of cores on 10 million random characters on 32-core machine with hyper-threading.

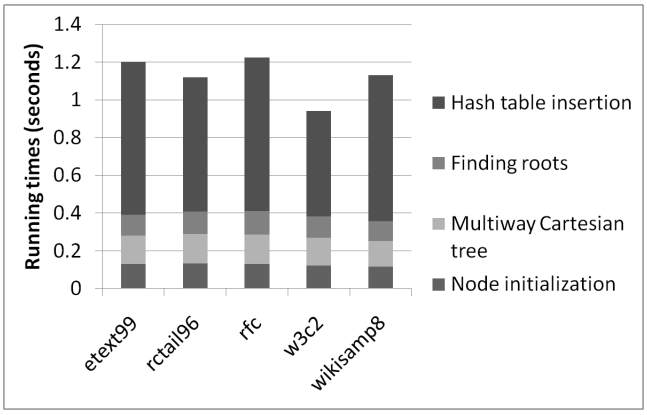

Figure 5: Breakdown of running times for converting a suffix array to suffix tree using Algorithm 1a on 32 cores with hyper-threading.

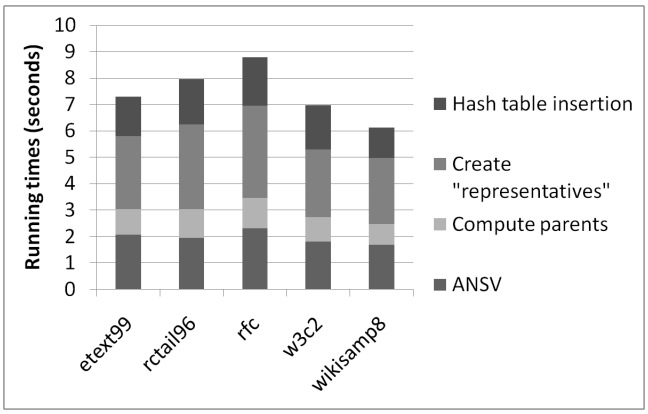

Figure 6: Breakdown of running times for the suffix tree portion of Algorithm 2 on 32 cores with hyper-threading.

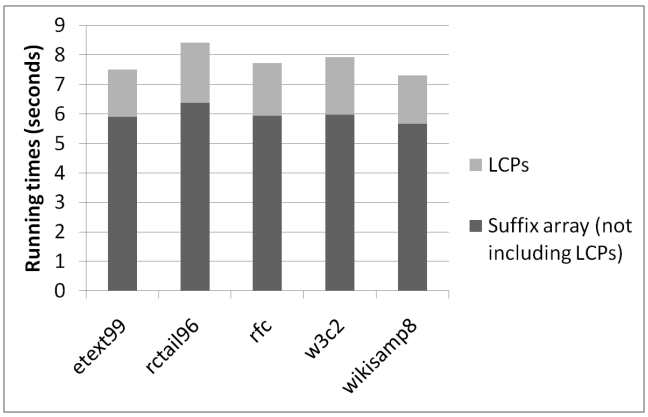

Figure 7: Breakdown of running times for the suffix array portion of Algorithm 1a and Algorithm 2 on 32 cores.

alignment and comparision, Nucleic Acids Research **30** (2002), no. 11, 2478–2483.

[FH06] Johannes Fischer and Volker Heun, *Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE*, CPM, 2006, pp. 36–48.

[FM96] M. Farach and S. Muthukrishnan, *Optimal logarith-*

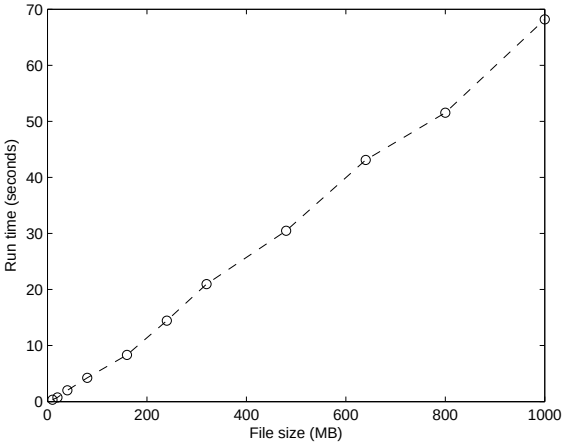

Figure 8: Running times for Algorithm 1a on random character files of varying sizes on 32-core machine with hyper-threading.

mic time randomized suffix tree construction, ICALP, 1996.

[GBT84] H. Gabow, J. Bentley, and R. Tarjan, *Scaling and related techniques for geometry problems*, STOC '84, 1984, pp. 135–143.

[GM09] A. Ghoting and K. Makarychev, *Indexing genomic sequences on the IBM Blue Gene*, Supercomputing 09, 2009.

[Gus97] D. Gusfield, *Algorithms on strings, trees and sequences*, Cambridge University Press, 1997.

[Har94] Ramesh Hariharan, *Optimal parallel suffix tree construction*, STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing, 1994, pp. 290–299.

[Int10] Intel, *Cilk++ programming language*, 2010, `http://software.intel.com/en-us/articles/intel-cilk`.

[IR04] C. Iliopoulos and W. Rytter, *On parallel transformations of suffix arrays into suffix trees*, AWOCA, 2004.

[Jaj92] J. Jaja, *Introduction to parallel algorithms*, Addison-Wesley Professional, 1992.

[KS99] S. Kurtz and C. Schleiermacher, *Reputer: Fast computation of maximal repeats in complete genomes*, Bioinformatics **15** (1999), no. 5, 426–427.

[KS03] J. Karkkainen and P. Sanders, *Simple linear work suffix array construction*, ICALP, 2003.

[KS07] F. Kulla and P. Sanders, *Scalable parallel suffix array construction*, Parallel Computing **33** (2007), no. 9, 605–612.

[Kur99] S. Kurtz, *Reducing the space requirement of suffix trees*, Software Practice Experience **29** (1999), no. 13, 1149–1171.

[McC76] Edward M. McCreight, *A space-economical suffix tree construction algorithm*, Journal of the ACM **23** (1976), no. 2, 262–272.

[MM90] U. Manber and G. Myers, *Suffix arrays: A new method for on-line string searches*, First ACM-SIAM Symposium on Discrete Algorithms, 1990, pp. 319–327.

[Mor68] Donald R. Morrison, *Patricia - practical algorithm to retrieve information coded in alphanumeric*, J. ACM **15** (1968), no. 4, 514–534.

[Mor10a] Yuta Mori, *libdivsufsort: A lightweight suffix-sorting library*, 2010, `http://code.google.com/p/libdivsufsort`.

[Mor10b] ———, *sais: An implementation of the induced sorting algorithm*, 2010, `http://sites.google.com/site/yuta256/sais`.

[MPK03] C. Meek, J. M. Patel, and S. Kasetty, *Oasis: An online and accurate technique for local-alignment searches on biological sequences*, VLDB, 2003.

[PST07] S. Puglisi, W. F. Smyth, and A. H. Turpin, *A taxonomy of suffix array construction algorithms*, ACM Computing Surveys **39** (2007), no. 2.

[PZ08] B. Phoophakdee and M. Zaki, *Trellis+: An effective approach for indexing genome-scale sequences using suffix trees*, Pacific Symposium on Biocomputing, vol. 13, 2008, pp. 90–101.

[Sad07] K. Sadakane, *Compressed suffix trees with full functionality*, Theory of Computing Systems **41** (2007), no. 4.

[TY03] D. Tsadok and S. Yona, *ANSI C implementation of a suffix tree*, 2003, `http://mila.cs.technion.ac.il/~yona/suffix_tree/`.

[Vui80] Jean Vuillemin, *A unifying look at data structures*, Commun. ACM **23** (1980), no. 4, 229–239.

[Wei73] P. Weiner, *Linear pattern matching algorithm*, 14th Annual IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.