Theoretical aspects of ERa, the fastest practical suffix tree construction algorithm

Matevž Jekovec
University of Ljubljana, Faculty of Computer and Information Science
Tržaška 25
1000 Ljubljana, Slovenia
matevz.jekovec@fri.uni-lj.si

ABSTRACT

Efficient construction of the suffix tree given an input text is an active area of research for the past 40 years. Both theoretical computer scientists and engineers trickled the problem. Unfortunately in the last decade not many results were acknowledged between the two communities. In this paper we narrow this gap by providing formal analysis of the practically fastest to date parallel algorithm for suffix tree construction using the well-accepted Parallel External Memory model of computation.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*; H.2.8 [**Database Management**]: Database Applications—*Scientific databases*; J.3 [**Life and Medical Sciences**]: [Biology and Genetics]

General Terms

Algorithms

Keywords

Suffix tree, parallelism, external memory, sequence indexing, genome indexing

1. INTRODUCTION

Suffix tree and suffix array data structures are the most widely used data structure in text indexing applications. They both allow answering three main queries: 1) is the given query string P present in the text, 2) where are located all its' occurrences in the text and 3) finding the longest prefix of the given query string P still present in the text. Suffix trees and suffix arrays allow answering these questions in time $O(|P|)$ and $O(|P| \lg n)$ respectively.

There were many practical suffix tree construction algorithms presented in the past, some of them based on suffix arrays, the others constructing the suffix tree directly. The most recent ones being B^2ST [2], Wavefront [4] and ERa [6]. In this paper we will focus on ERa, the fastest practical algorithm for the suffix tree construction to date. From the theoretical point of view, it was already shown in [3] that the suffix tree and suffix array construction is tight to (parallel) string sorting problem: $\Omega(n \log n)$ time and $\Omega(\frac{n}{pB} \log_{M/B} \frac{n}{M})$ parallel block transfers. In this paper we are interested about the parallel time and I/O complexity of ERa. We show that the algorithm running time and block transfers are not tight thus leaving open question whether it is possible to design an I/O optimal algorithm which is also practical.

The rest of the paper is divided into three parts. In section 2 we introduce our notation, the suffix tree and provide intuitive outline of ERa. The core of the paper is section 3 providing formal time and I/O complexity analysis of the algorithm. We conclude our results in section 4.

2. BACKGROUND

2.1 Suffix tree

Given a text $T[1..n]$, the substring $T[i..n]$ for any $i \in \{1...n\}$ is called a suffix of T. The characters in T are from a finite alphabet Σ except for the last character $T[n] = \$$ which is called the *delimiter character* and is unique in the text. The *suffix tree* (formally introduced and constructed in [7]) is compressed trie storing all suffixes of T. Each edge represents a substring of T of length $1...n$. There are exactly n leafs in the suffix tree where each leaf stores a position of the suffix in the original text. Each path from the root to the leaf defines a unique suffix and the value in the leaf determines its location inside the text T. The children of each node are lexicographically ordered. If we traverse all the leafs from left to right, we obtain lexicographically ordered suffixes of text T.

In order to find all the occurrences of a given pattern P in the text, we scan the pattern character by character and follow the edge corresponding to the character or checking whether the current edge label matches the characters, if the edge's label length is > 1. If there is no corresponding edge in the node or the edge label doesn't match the corresponding character in P, no such pattern exists in the text. Otherwise, we return all the leafs' values (suffix positions in the text) inside the obtained subtree. To avoid $O(P \cdot occ)$ time, we add two additional links to each internal node pointing to the left- and the right-most descendant in its subtree and make a

linked list of all the leafs. Without asymptotically increasing the space complexity, this approach allows us to jump to the leaf in constant time and report the corresponding leafs linear time.

2.2 PEM model

Parallel external memory (PEM) model [1] is of a shared memory kind consisting of p processors and a two-level memory hierarchy. The 2nd level is an external memory and accessible by any processor, whereas the 1st level memories are private caches, each of size M. Processors can only perform operations by accessing their caches. The data is transferred between the external memory and the caches in both directions in blocks of size B. It is assumed there is enough bandwidth between the external memory and caches to support transferring any block to each of the processors in parallel. This is the fundamental difference between the sequential EM model when evaluating the complexity of the algorithm. Figure 1 illustrates the PEM model.

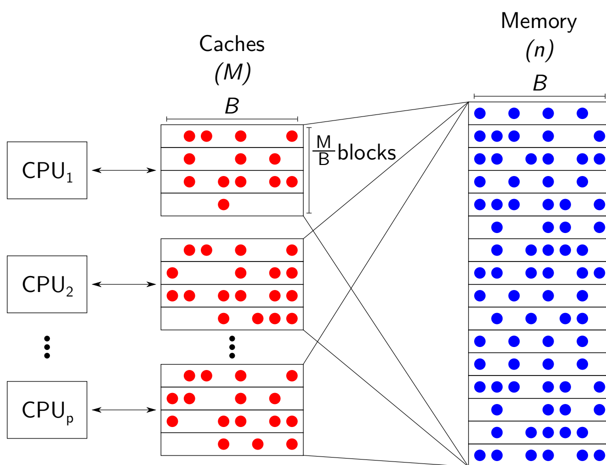

Figure 1: The Parallel External Memory model.

If concurrent writes to the external memory occur, any of the CRCW, CREW or EREW models are allowed. For simulating any CRCW algorithm on EREW, Priority-CRCW can be used to merge contents of the same block first in pairs, fourths and so on in $\lceil \log p \rceil$ rounds. If not stated otherwise, we will use PEM CREW model through the rest of this paper. Beside the parallel running time, we will use the parallel I/O performance metric, i.e. the number of *parallel block transfers* between the external and the main memory.

2.3 ERa

Elastic Range algorithm is the fastest algorithm for the suffix tree construction to date [6]. It runs in two phases: the *vertical partitioning* and the *horizontal partitioning*. The parameters of the algorithm are the number of processors p and the main memory size M. We first outline the two phases in this section intuitively and provide formal algorithms and its formal analysis in the next section.

The vertical partitioning is used to partition the suffix tree into manageable suffix subtrees $\mathcal{T}_{\pi_i}$ fitting into the main memory M. This is done by first scanning the original text and remembering the frequencies of characters f_{π_i} for the character $\pi_i : 1 \leq i \leq |\Sigma|$. Then, obtained characters are appended their succeeding characters of the original text. π_i thus become substrings of the original text of length 2 for $1 \leq i \leq |\Sigma|^2$. Authors of [6] denote these substrings as S-prefixes, because they represent prefixes of the suffixes of the original text. Note that each π_i occurrence in the original text introduces one text suffix starting at that location resulting in one leaf and at most one internal node in the suffix tree. Keeping this in mind and knowing the node space consumption we can calculate the maximum frequency of each S-prefix so its corresponding suffix tree will still fit into the main memory. The goal of the vertical partitioning is to extend S-prefixes until their frequencies are lower than this bound. It returns a set of empty suffix subtrees $\mathcal{T}_{\pi_i}$ consisting only of leafs representing occurrences of π_i. One or more subtrees are packed into *virtual trees*, each fitting into the main memory as tight as possible.

The horizontal partitioning takes the occurrences of each S-prefix π_i as the input and constructs internal nodes of the suffix subtree $\mathcal{T}_{\pi_i}$. The crux of the method is optimizing 1) the original text access and 2) the memory access. Optimizing text access only is relatively easy by building the suffix tree in a breadth-first manner and scanning the whole input text layer by layer in the suffix tree. Readers should consult [6] for details. Combining both the efficient original text access and the memory access is more demanding: by definition, suffix tree contains suffixes ordered lexicographically, whereas the original text can have any character present at arbitrary location. Therefore, one cannot expect to construct the suffix tree using a tree traversal and contiguous text access. How the horizontal partitioning does it is to first read the text from the left to the right and at each S-prefix occurrence store the text following the S-prefix into the buffers of fixed size located in the main memory. Then, the algorithm does an in-memory string sorting of the buffers. This is the only non-sequential memory access step of the whole approach. Finally, by sequentially reading the ordered buffers and knowing their original location in the text, the algorithm constructs a suffix tree in the external memory sequentially by a depth-first tree traversal. Gluing suffix subtrees together to the final suffix tree requires only setting the correct references in constant time per processor.

3. THE ANALYSIS

For the analysis we acknowledge two assumptions:

- The main memory size property: $M^2 > n$ and
- the input text is *not skewed* meaning it has an uniform distribution of substrings. The human genome and music were shown not to be skewed in [5].

3.1 Vertical Partitioning

Algorithm 1 shows the vertical partitioning algorithm from [6] with slightly different notation: We denote the maximum number of leafs in the main memory as M instead of $\mathcal{F}_M$. The algorithm first builds a set P of S-prefixes fitting into the main memory in lines 4-11. The number of iterations depend on the longest path label in the text, $|\Sigma|$ and M. The uniform substring distribution of the input text leads

to $1 \leq k \leq \log_{|\Sigma|} \frac{n}{M}$ expected number of iterations and $|P| = O(\frac{n}{M})$ S-prefixes. The inner loop in lines 7-10 is executed $|\Sigma|^k$ times each iteration k. In line 12 any sorting algorithm of integer numbers is involved. In lines 13-22 a First-Fit Decreasing heuristic for bin packing problem is used ([8]) to construct virtual trees of S-prefixes fitting into the main memory. Assuming a uniform distribution of substrings all frequencies f_{π_i} are equal and expected to be between $\frac{M}{|\Sigma|} < f_{\pi_i} \leq M$, then the loop is iterated between $\frac{n}{|\Sigma| M}$ and $\frac{n}{M}$ times respectively.

Algorithm 1: VerticalPartitioning

Input: String S, alphabet Σ, 1st level memory size M
Output: Set of $VirtualTrees$

```
1  VirtualTrees ← ∅
2  P ← ∅
3  P' ← {∀ symbol s ∈ Σ generate a S-prefix π_i ∈ P'}
4  repeat
5      scan input string S
6      count in S the frequency f_{π_i} of every S-prefix π_i ∈ P'
7      forall the π_i ∈ P' do
8          if 0 < f_{π_i} ≤ M then add π_i to P
9          else forall the symbol s ∈ Σ do add π_i s to P'
10         remove π_i from P'
11 until P' = ∅;
12 sort P in descending f_{π_i} order
13 repeat
14     G ← ∅
15     add P.head to G and remove the item from P
16     curr ← next item in P
17     while NOT end of P do
18         if f_curr + SUM_{γ_i ∈ G}(f_{γ_i}) ≤ M then
19             add curr to G and remove the item from P
20         curr ← next item in P
21     add G to VirtualTrees
22 until P = ∅;
23 return VirtualTrees
```

Time. By exploiting the geometric sum $|\Sigma| + |\Sigma|^2 + ... + |\Sigma|^k = \frac{\Sigma^{k+1}-1}{|\Sigma|-1}$ the first loop in lines 4-11 overall takes the following number of steps:

$$\sum_{k=1}^{k=\log_{|\Sigma|} \frac{n}{M}} (scan(n) + |\Sigma|^k) = \log_{|\Sigma|} \frac{n}{M} \cdot scan(n) + \frac{n/M - 1}{|\Sigma| - 1} \quad (1)$$

We assume comparison-based sorting in line 12 running in $O(\frac{n}{M} \lg \frac{n}{M})$ time. We only provide asymptotic bounds for the last loop in lines 13-22. Overall it requires from $O((\frac{n}{|\Sigma| M})^2)$ to $O((\frac{n}{M})^2)$ time. The whole vertical partitioning phase requires

$$O(n \log_{|\Sigma|} \frac{n}{M} + (\frac{n}{M})^2) \quad (2)$$

time.

I/O. The first loop in lines 4-11 require overall $\log_{|\Sigma|} \frac{n}{M} scan(n)$ block transfers. The internal loop operates in-memory only because $|P|$ and $|P'| < \frac{n}{M}$ and the assumption $M^2 > n$.

The sort in line 12 is an in-memory one because P fits into the main memory and does not require any I/Os. The heuristic for virtual tree construction is executed completely in-memory, because P fits into the main memory. Overall the vertical partitioning requires

$$\frac{n}{B} \log_{|\Sigma|} \frac{n}{M} \quad (3)$$

sequential block transfers.

3.2 Horizontal Partitioning

Algorithm 2 shows the horizontal partitioning algorithm taken from [6]. Relative suffix array $SA[i]$[1] maps the ith rank of the suffix in lexicographically ordered list of all the suffixes of S-prefix p to the position in the original text. SA is originally initialized to suffix locations of S-prefix p in the text and gets the final form when sorting the buffered string. $ISA[i]$[2] denotes the inverse suffix array defined as $ISA[SA[i]] = i$. Relative longest common prefix array $LCP[i]$[3] contains relative branching information between the suffix such that $LCP[i] = lcp(T[i-1], T[i]) - |p|$ where lcp denotes the longest common prefix of two strings.

In line 9 the algorithm determines the string buffer length $range$ depending on the number of still open paths to the leafs and the main memory size M. Then, in lines 10-12 the algorithm fills buffers Buf each of length $range$ in a single scan of the original text. In the analysis we assume $range$ to be a constant factor such that $sizeof(Buf) = O(M)$ and not $O(M^2)$. In lines 13-15 an in-memory string sorting of Buf is involved constructing SA and ISA. Finally in lines 16-23 the in-memory construction of LCP is done by calculating the common prefix cp[4] of $SA[i]$ and $SA[i+1]$ for all i. The number of iterations of the external while loop depends on the similarity of the strings in Buf. Assuming random string distribution, the number of uniquely represented strings in $Buf = O(M)$ is $|\Sigma|^{range}$. $range$ thus bounds the number of iterations to $O(\log_{|\Sigma|} M)$.

Time. We will first calculate the amount of work needed to be done and then provide parallel time execution. Line 9 requires $O(1)$ work to calculate $range$ and lines 10-12 require $O(M)$ work to fill the buffers. String sorting takes $O(M)$ work by using radix sorting. Finally lines 16-23 overall take $O(M)$ work to calculate common prefixes of all the strings. The external loop is executed $O(\log_{|\Sigma|} M)$ times leading to $O(M \log_{|\Sigma|} M)$ overall expected work of the horizontal partitioning for the random input text.

Each processor can run its own horizontal partitioning instance for the assigned suffix subtree. This leads to optimal

[1] SA was denoted as L in the original paper.
[2] ISA was denoted as I in the original paper.
[3] LCP was denoted as B in the original paper.
[4] Common prefix cp was denoted as common S-prefix cs in the original paper, which is, to our understanding, not correct.

Algorithm 2: SubTreePrepare

Input: Input string S, S-prefix p
Output: Arrays SA and LCP corresponding suffix sub-tree $\mathcal{T}_p$

1 SA contains the locations of S-prefix p in string T
2 $LCP \leftarrow \{\}$
3 $ISA \leftarrow \{0, 1, ..., |SA| - 1\}$
4 $A \leftarrow \{0, 0, ..., 0\}$
5 $Buf \leftarrow \{\}$
6 $P \leftarrow \{0, 1, ..., |L| - 1\}$
7 $start \leftarrow |p|$
8 **while** *there exist an undefined* $Buf[i]$, $1 \leq i \leq |SA| - 1$ **do**
9     $range \leftarrow GetRangeOfSymbols$
10   **for** $i \leftarrow 0$ *to* $|SA| - 1$ **do**
11   **if** $ISA[i] \neq done$ **then**
12             $Buf[ISA[i]] \leftarrow ReadRange(T, SA[ISA[i]] + start, range)$ // ReadRange(T,a,b) reads b symbols of T starting at position a
13   **for** *every active area* AA **do**
14   Reorder the elements of Buf, P and SA in AA so that Buf is lexicographically sorted. In the process maintain the index ISA
15   If two or more elements $\{a_1, ..., a_t\} \in AA, 2 \leq t$, exist such that $Buf[a_1] = ... = Buf[a_i]$ introduce for them a new active area
16   **for** *all* i *such that* $Buf[i]$ *is not defined,* $1 \leq i \leq |SA| - 1$ **do**
17   cp is the common prefix of $Buf[i-1]$ and $Buf[i]$
18   **if** $|cp| < range$ **then**
19             $Buf[i] \leftarrow (Buf[i-1][|cp|], Buf[i][|cp|], start + |cp|)$
20   **if** $Buf[i-1]$ *is defined or* $i = 1$ **then**
21   Mark $ISA[P[i-1]]$ and $A[i-1]$ as *done*
22   **if** $Buf[i+1]$ *is defined or* $i = [SA] - 1$ **then**
23   Mark $ISA[P[i]]$ and $A[i]$ as *done* // last element of an active area
24     $start \leftarrow start + range$
25 **return** *(SA,LCP)*

speedup and taking all $O(n/M)$ suffix subtrees into account, the expected parallel execution time to construct all of them in parallel is

$$O(\frac{n}{M}\frac{M\log_{|\Sigma|} M}{p}) = O(\frac{n}{p}\log_{|\Sigma|} M) \quad (4)$$

I/O. Determining $range$ in line 9 requires no I/Os. Lines 10-12 require $O(M/B)$ I/Os to fill the buffers. Lines 13-23 use in-memory sorting and common prefix calculation and do not impose any I/Os. The overall I/O complexity of the horizontal partitioning is $O(M/B\log_{|\Sigma|} M)$.

In parallel environment, scans in lines 10-12 can be shared. To construct all $O(n/M)$ suffix subtrees in parallel, the expected parallel I/O complexity is

$$O(\frac{n}{pM}\frac{M\log_{|\Sigma|} M}{B}) = O(\frac{n}{pB}\log_{|\Sigma|} M) \quad (5)$$

parallel block transfers.

4. CONCLUSIONS

In this paper we provided formal analysis for the ERa algorithm, the fastest practical suffix tree construction algorithm. The algorithm runs in two phases: The first one is executed sequentially and partitions the suffix tree to smaller suffix subtrees fitting into the main memory whereas the second phase fills the suffix subtrees in parallel. We showed the first phase requires $O(n\log_{|\Sigma|}\frac{n}{M} + (\frac{n}{M})^2)$ time and $O(\frac{n}{B}\log_{|\Sigma|}\frac{n}{M})$ sequential block transfers. The second phase requires $O(\frac{n}{p}\log_{|\Sigma|} M)$ parallel time and $O(\frac{n}{pB}\log_{|\Sigma|} M)$ parallel block transfers. The suffix tree construction lower bound is the same as the sorting problem: $\Omega(n\log n)$ time and $\Omega(\frac{n}{pB}\log_{M/B}\frac{n}{M})$ parallel I/Os by [3]. ERa, despite being the fastest practical algorithm to date, is not tight.

5. REFERENCES

[1] Arge, L., Goodrich, M. T., Nelson, M., and Sitchinava, N. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the 20th annual symposium on Parallelism in algorithms and architectures - SPAA '08* (New York, USA, 2008), ACM Press, p. 197.

[2] Barsky, M., Stege, U., Thomo, A., and Upton, C. Suffix trees for very large genomic sequences. In *Proceeding of the 18th ACM conference on Information and knowledge management - CIKM '09* (New York, New York, USA, Nov. 2009), ACM Press, p. 1417.

[3] Farach-Colton, M., Ferragina, P., and Muthukrishnan, S. On the sorting-complexity of suffix tree construction. *Journal of the ACM 47*, 6 (Nov. 2000), 987–1011.

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

[5] Heinz, S., Zobel, J., and Williams, H. E. Burst tries: a fast, efficient data structure for string keys. *ACM Transactions on Information Systems 20*, 2 (Apr. 2002), 192–223.

[6] Mansour, E., Allam, A., Skiadopoulos, S., and Kalnis, P. ERA: efficient serial and parallel suffix tree construction for very long strings. *Proceedings of the VLDB Endowment 5*, 1 (2011), 49–60.

[7] Weiner, P. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on* (1973), IEEE, pp. 1–11.

[8] Yue, M. A simple proof of the inequality FFD (L) \leq 11/9 OPT (L) + 1, \forall L for the FFD bin-packing algorithm. *Acta Mathematicae Applicatae Sinica 7*, 4 (1991), 321–331.