A New Parallel Partition Algorithm for Parallel Suffix Tree Construction

Jun Ho Seo and Joong Chae Na

Department of Computer Science & Engineering, Sejong University
98, Gunja-Dong, Gwangjin-Gu, Seoul, 143-747, Republic of Korea
jhseo@sju.ac.kr, jcna@sejong.ac.kr

1 Introduction

The suffix tree is a compacted trie of all suffixes of a given string. It is a fundamental data structure in a wide range of domains such as text processing, data compression, computer vision, computational biology, and so on [1]. Moreover, it can be used for network researches such as web analysis, which has been studied actively [2], [3]. For example, suffix trees have been utilized to effectively search for genomic DNA data in databases or text in the web.

Recently, as parallel architectures such as distributed systems and CMPs improve, there have been studied developing parallel suffix tree construction algorithms practically. Chen and Schmidt [4] proposed a parallel algorithm for constructing suffix trees on a computational grid. Tsirogiannis and Kouds [5] proposed cache-conscious suffix tree construction algorithms that are tailored to CMP architectures. These algorithms take an approach of dividing suffixes into a number of partitions and constructing parallel the suffix tree for each partition.

In this paper, we propose a new algorithm of partitioning suffixes. Basically, our algorithm is similar to that in [5], which partitions suffixes by prefixes of variable lengths. However, we use a trie instead of a hash table as an auxiliary data structure. Moreover, our algorithm can be easily parallelized to adapt to parallel architectures.

2 Partitioning Algorithm

We present an algorithm for partitioning suffixes using a trie. The partitioning is to divide all suffixes of a given string into a number of partitions such that all suffixes in each partition share a common prefix. Our partitioning algorithm consists of three steps. First, we construct an auxiliary data structure called a partition trie based on sampling. Next, we compute modified suffix link (msl) for each node in the partition trie. At last, suffix indices of the input string populate nodes in the partition trie.

* Corresponding author.

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(2010-0028134).
2.1 Constructing the partition trie

The partition trie is an auxiliary data structure that stores an associative common prefix of suffixes of the input string. Each internal node has $|\Sigma|$ children, where $\Sigma$ is alphabet of the input string and each child represents distinct symbol in $\Sigma$. The concatenation of the symbols on the path from the root to a leaf represents the common prefix of suffixes. We call a leaf a partition node in the partition trie. Fig. 1 shows an example of a partition trie.

![Figure 1](image.png)

**Figure 1.** The partition trie of $S = ABBCABCAAB$ when $k = 1$ and threshold = 2

The partition trie is constructed by using sample-based technique and expanding the partition nodes gradually. We use a number of sample strings from the input string for constructing the partition trie instead of the entire input string. Constructing the partition trie is performed by several passes.

Initially, we construct the trie representing all possible strings of length $k$ with $|\Sigma|^k$ partition nodes, where $k > 0$ is a fixed parameter. Then, expanding partition nodes starts from pass $k$ and ends when no partition node is expanded. At the beginning of pass $p$, all possible substrings of length $p$ are extracted from sample strings and theirs frequencies are computed. Each partition node has a count value to count the frequency of its corresponding string. For each extracted substring, the count value of the corresponding partition node increases by one. If there is no partition node corresponding to the extracted substring, the substring is ignored.

After computing the frequencies, each count value is examined whether it is greater than a given threshold or not. If a count is greater than the threshold, expanding for the partition node is processed and then the node is not a partition node anymore. The expanding of a partition node is defined as attaching $|\Sigma|$ children to the partition node. Each child represents a distinct symbol in $\Sigma$. If a count is less than or equal to the threshold, any processing for the node does not need and it is fixed to the partition node. If an expanded node exists, the next pass continues.
2.2 Computing modified suffix links

The next step is computing modified suffix links (msls) for node in the partition trie. The modified suffix link is defined as follows. Let $v$ be the node $v$ representing a string $x\alpha$ ($x$ is a symbol and $\alpha$ is a string, possibly empty). If there is the node $w$ representing string $\alpha$, then the modified suffix link of node $v$ points to the node $w$. If such node $w$ does not exist, the modified suffix link of node $v$ points to the node representing the maximum length prefix of $\alpha$. To compute msls can be performed by a breath-first-search (BFS) on the partition trie.

2.3 Populating suffix indices

As the last step of this phase, suffix indices of the input string populate nodes in the partition. The populating step performs that it reads each suffix and finds the corresponding partition node in the partition trie. The leftmost suffix is first processed and it proceeds to right. Because we have computed the msls of all nodes in the partition trie, after we completed first suffix, it does not need to start at the root node, but it is possible to start in the internal node in the trie using the msl. That is, when the msl of a node $v$ points to a node $w$, we can start at the node $w$ and compare a suffix with the partition trie.

In the entire partitioning algorithm, the populating step takes the most time because it processes all suffixes in the input string. Except for inserting suffix index, all operations can be executed simultaneously on CMPs. The entire input string is divided into some range, and each range is assigned to each core (thread). Each core (thread) performs populating and uses synchronization mechanism when inserting a suffix index into a partition node.

References