A Parallel Algorithm for Partitioning Strings†

Joong Chae Na[1], Jeong Seop Sim[2]

[1]Sejong University, 98 Gunja-Dong, Gwangjin-Gu, Seoul, KOREA

[2]Inha University, 256 Yonghyun-Dong, Nam-Gu, Inchon, KOREA

Abstract

The string partitioning problem is partitioning given strings according to their prefixes. It is often occurred in many string processing algorithms. In this paper, we consider the parallel version for partitioning strings, especially, partitioning suffixes derived from a string, which is an essential subroutine used in most parallel suffix tree construction algorithms. We propose a new partitioning algorithm using an auxiliary data structure called a trie. Our algorithm is simpler and more efficient than the previous algorithm using a hash table. We present the performance of our algorithm experimentally. The experimental results show that our algorithm is faster than the algorithm using a hash table and achieves good speedup on the CMP architecture.

Keyword : Parallel algorithm, Suffix tree partition, Suffix tree construction, Cilk Plus, TBB

---

† Its original work was presented in IST2012.
This research was supported by the IT R&D program of MSIP/KEIT [10041971, Development of Power Efficient High-Performance Multimedia Contents Service Technology using Context-Adapting Distributed Transcoding].

[1] Joong Chae Na
jcna@sejong.ac.kr
Address. Dept. of Computer Science & Engineering, Sejong Univ. 98 Gunja-Dong, Gwangjin-Gu, Seoul, KOREA
Tel. +82-2-3408-3839

[2] Jeong Seop Sim
jssim@inha.ac.kr
Address. Dept. of Computer Science & Information Engineering, Inha Univ. 253 Yonghyun-Dong, Nam-Gu, Incheon, KOREA
Tel. +82-32-860-7452

1. Introduction

The string partitioning problem is partitioning given strings according to their prefixes. It is often occurred in many string processing algorithms. For example, most parallel suffix tree construction algorithms take the approach of dividing the suffixes into a number of partitions and constructing in parallel the partial suffix tree for each partition. Therefore, the partitioning of the suffixes is a key procedure in parallel suffix tree construction algorithms on disks, distributed systems, CMP architectures, etc. Generally, the suffixes in each partition share a common prefix. Chen and Schmidt [2] partitioned suffixes by prefixes of fixed lengths using a string matching algorithm. Tsirogiannis and Koudas [3] used prefixes of variable lengths so that every partition approximately identical in size and a hash table was used for partitioning.

In this paper we propose a parallel partitioning algorithm for partitioning strings using a trie. Our algorithm partitions the strings by prefixes of variable lengths so that strings are distributed as uniformly as possible. We mainly focus on partitioning suffixes instead of strings. The reason is that our algorithm is especially suitable for suffix tree construction because the suffix tree is basically a trie. Furthermore, our algorithm can be easily applied to the problem of partitioning strings. By using a trie, we improve the performance of the partitioning procedure, compared to an algorithm using a hash table for the auxiliary data structure, as shown experimentally in Section 3.

2. Parallel Partition algorithm

Our partition algorithm consists of three steps. First, we construct an auxiliary data structure called a partition trie. Next, we compute the modified suffix link (MSL) for each node in the partition trie. Finally, suffixes of the input string populate the nodes in the partition trie.

The partition trie is a trie that represents the prefixes of suffixes of the input string. Each internal node has $|\sum|$ children, where $\sum$ is the alphabet of the input string; each child represents a distinct symbol in $\sum$. The concatenation of the symbols on the path from the root to a leaf represents the common prefix of the suffixes. We call a leaf in the partition trie a partition node.

The partition trie is constructed by expanding the nodes gradually according to the frequencies of the corresponding prefixes. Constructing the partition trie consists of several passes. Initially, we construct the trie representing all of the possible strings of length k with $|\sum|k$ leaves, where $k > 0$ is a fixed parameter. Then, the trie is expended starting from pass k until no leaf is expanded.

At the beginning of pass p, we extract all of the possible substrings of length p from the string and compute their frequencies. Each leaf in the trie has a count value used to count the frequency of its corresponding string. For each extracted substring, we increase the count value of the corresponding leaf by one. If there is no leaf corresponding to the extracted substring, the substring is ignored. If the count is greater than t, we expand the leaf by attaching it to |∑| children representing the distinct symbols in ∑. Otherwise, the leaf is not further expanded and is fixed to the partition node. If an expanded node exists, the next pass continues.

The next step is computing the modified suffix links (MSLs) of the nodes in the partition trie. The modified suffix link is defined as follows: Let v be the node representing a string xα, where x is a symbol and α is a (possibly empty) string. If there is a node w representing string α, then the MSL of node v points to node w. If such a node does not exist, the MSL of node v points to the node representing the prefix of α of maximum length. Computing the MSLs can be performed by a breath-first-search (BFS) on the partition trie.

The last step of this phase is that the suffixes of the input string populate partition nodes in the partition trie. For each suffix, we find the corresponding partition node in the partition trie. We process the suffixes from the leftmost to the right. Due to the MSL in the partition trie, to populate the second suffix, we do not need to start at the root node but we can start at an internal node using a MSL. The role of MSLs in the partition trie is similar to the one of suffix links in the suffix tree.

In the entire suffix partitioning algorithm, the populating step takes the most time because it processes all of the suffixes of the input string. Except for inserting the suffix indices, we can execute all of the operations simultaneously on the CMPs. We divide the entire input string into ranges and assign each range to each core (thread). Each core (thread) performs the populating and uses the synchronization mechanism when inserting a suffix index into a partition node.

3. Experimental Results

We show experiment results to compare our algorithm using a trie with the algorithm using a hash table. The experiment environment was as follows: The CPU was an Intel i7-930 which has 4 cores and 8 hyper threads. The system had 24GB of main memory. We used input strings generated randomly with |∑|=4. The parallel algorithm was implemented by Intel's Cilk Plus Platform [4], and the concurrent hash table was implemented by "concurrent_hash_map" in Intel's Threading Building Blocks [5].

Table 1. The execution time (sec) of the partition phase according to the various numbers of threads when the text length is 10^8.

# of threads	**1**	**2**	**3**	**4**
trie	36.2	19.6	13.3	10.0
hash table	69.1	37.2	24.6	17.5
# of threads	**5**	**6**	**7**	**8**
trie	8.7	7.8	7.1	6.1
hash table	14.3	12.2	10.4	9.2

First, we compared the performance of the two partition algorithms. Table 1 shows the execution time of the partition phase for texts of length 108 according to the number of threads. For all ranges of threads examined, the algorithm using the trie was 1.2~2 times faster than the algorithm using the hash table.

Table 2. The execution time (sec) of the partition phase for the various lengths of texts when using eight threads.

length(million)	10	20	30	40	50	60	70	80	90	100
Trie	0.53	1.1	1.5	2.5	3.0	3.7	4.2	4.8	5.5	6.1
hash table	0.6	1.2	1.9	2.8	4.2	5.2	6.0	7.5	8.5	9.2

Table 2 shows the execution time according to the lengths when using eight threads. In both algorithms, the execution times increased almost linearly to the lengths of the strings. Like in Table 1, our algorithm is faster than the algorithm using the hash table for strings of all lengths. The performance improvements between two algorithms were nearly the same regardless of the lengths of input strings.

Reference

[1] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, New York, 1997.

[2] C. Chen and B. Schmidt, “Constructing large suffix trees on a computational grid,” *J. Parallel and Distributed Computing*, vol. 66, pp. 1512-1523, 2006.

[3] D. Tsirogiannis and N. Koudas, “Suffix Tree Construction Algorithms on Modern Hardware,” *Proc. the 13th International Conference on Extending Database Technology*, Lausanne, p. 263-274, 2010.

[4] Intel Cilk Plus SDK Programmer’s Guide,
http://www.clear.rice.edu/comp422/resources/Intel_Cilk++_Programmers_Guide.pdf

[5] Intel Threading Building Blocks , http://threadingbuildingblocks.org/