Contemporary Engineering Sciences, Vol. 7, 2014, no. 15, 743 - 748
HIKARI Ltd, www.m-hikari.com
http://dx.doi.org/10.12988/ces.2014.4684

A Survey on Disk-based Genome Sequence Indexing

Woong-Kee Loh

Department of Software Design and Management, Gachon University
1342 Seongnam-daero, Sujeong-gu, Seongnam-shi,
Gyeonggi-do, 461-701 Republic of Korea

Young-Ho Park

Department of Multimedia Science, Sookmyung Women's University
100 47-gil, Cheongpa-ro, Yongsan-gu,
Seoul, 140-742 Republic of Korea
(Corresponding author)

Abstract

Due to recent remarkable advances in genome sequencing technologies, the DNA sequences of many living organisms including the human being have been collected in databases. To fully harness the genome databases, we should need the technologies for efficient indexing and searching the genome sequences, and the suffix tree has been adopted in many recent research papers. In this paper, we survey the most up-to-date indexing algorithms that construct the disk-based suffix trees for genome sequences and compare their strengths/weaknesses and performance.

Keywords: genome sequences, indexing, suffix tree

1 Introduction

Due to the recent advance in genome sequencing technology, the genome sequences of many living organisms including the human being have been collect-

ed in the genome databases. An essential issue is to index the large-scale genome sequences for efficient search. The suffix tree is most widely adopted for indexing genome sequences [1,3,4,6,7]; it is a data structure built for a given string S to support various search on S such as exact and approximate matching on a substring in S, counting the occurrence of a substring, and finding substrings matching with a specific regular expression.

Ukkonen's algorithm [8] is the most famous one which, given a string of length n, builds the corresponding suffix tree in $O(n)$ time. Its intrinsic assumption is that the entire input string and output suffix tree can be managed in the main memory. However, the genome sequences have more than million or billion times the size of the strings dealt with in the previous traditional suffix tree applications. Moreover, the size of suffix tree itself is two orders of magnitude larger than the original string. Thus, if we apply the algorithm for genome sequences whose size exceeds the main memory, it should cause severe disk swap in and out for virtual memory access. This phenomenon is called *memory bottleneck problem* or *thrashing* and causes severe performance degradation [1,3,7].

To tackle the problem, there have been many algorithms proposed for building the disk-based suffix trees. In this paper, we survey the most recent disk-based genome sequence indexing algorithms. These algorithms are divided into two groups, i.e. sequential and parallel algorithms. The latter make the most of recent multi-core CPUs and/or multi-CPU machines to achieve dramatic performance improvement. Each group of algorithms is described in Sections 2 and 3, respectively.

2 Sequential Algorithms

The memory bottleneck problem arises when the memory access pattern has poor referential locality. For better referential locality, the memory should be accessed sequentially or in a readily-estimated pattern. The poor locality causes frequent memory faults which increase the access of storage in the lower hierarchy level such as hard disk drives (HDD). Since HDD has much higher access costs than main memory, frequent access of HDD incurs severe performance degradation.

Hunt et al. [3] proposed the first disk-based suffix tree construction algorithm. The algorithm partitions the given genome sequence and then constructs a suffix subtree for each partition. Although Hunt's algorithm has the complexity of $O(n^2)$, it has significantly improved the indexing performance than Ukkonen's algorithm by reducing the number of disk accesses. However, the algorithm incurs heavy random disk accesses caused by the persistent Java object storage interface called PJama.

Tian et al. [7] presented the Top-Down Disk-based (TDD) approach for constructing the disk-based suffix tree. The TDD approach consists of (a) Partition and Write Only Top Down (PWOTD) algorithm based on Wotd-eager algorithm and (b) memory buffer management algorithm to improve the performance of PWOTD. It was shown [7] that TDD incurred only one sixth of disk accesses than Hunt's algorithm. TDD constructed the suffix tree for the entire human genome sequence in 30 hours. However, since TDD assigns the largest portion of main memory to the input genome sequence, it suffers from more severe performance degradation due to random disk accesses when the size of genome sequence gets larger.

Phoophakdee and Zaki [6] proposed an algorithm called TRELLIS, which overcame the skewness among suffix subtrees by partitioning with variable-length prefixes. TRELLIS consists of three phases: prefix creation, partitioning, and merging phases. In the partitioning phase, a suffix subtree is generated for each partition using a simple suffix tree build algorithm. In the merging phase, for each prefix *p*, the corresponding suffix subsubtrees are extracted from all the suffix subtrees and then merged. The prefix *p* is obtained in the first prefix creation phase so that each suffix subtree for *p* fits in main memory. TRELLIS outperformed TDD by up to 4 times and constructed the suffix tree for the entire human genome sequence in 4.2 hours.

Barsky et al. [1] proposed an algorithm called DIGEST which consists of two phases similar to the merge-sort algorithm. Figure 1 shows the phases of DIGEST. In the first phase, the entire genome sequence is divided into multiple partitions of the same length. For each partition *P*, the suffixes in *P* are sorted in main memory and then stored in the disk. In the second phase, the partitions are merged to have the suffixes in all partitions sorted. A partition is made of many sequential blocks of the fixed size. The blocks in each partition are read sequentially one by one from the disk, and the suffixes in the blocks from every partition are compared with each other. The smallest one is extracted to save in the merge block. This procedure is repeated until all the suffixes from every partition are extracted and saved in merge blocks. DIGEST outperformed TRELLIS [6] by up to 40% and completed indexing the entire human genome sequence in about 85 minutes. However, DIGEST incurs random disk accesses to read suffix blocks from different partitions in the merging phase and hence suffers from performance degradation.

Barsky et al. [2] proposed B^2ST algorithm, an extension of DIGEST, for handling multi-species genome sequences. An important assumption by B^2ST different from all the previous algorithms is that the resulting suffix tree should not have to refer to the original genome sequence to find the edge label in the tree, since the genome sequence cannot be loaded in main memory due to its enormous size. In the partitioning phase, the entire genome sequence is divided into multiple parti-

tions of the same length as in DIGEST. In the sorting phase, the algorithm performs suffix sorting for every possible combination of partitions. Although the sorting is done in main memory, it takes a lot of time and memory space, since the number of combination is in the order of $O(p^2)$ and it should process two partitions for each sorting, where p is the number of partitions. The sort phase can be hardly parallelized due to very large amount of memory usage. Actually, more than 90% of whole execution time of B^{2}ST is taken by partitioning and sorting phases [2]. In the merging phase, which is performed similarly to DIGEST, the access order of blocks across the partitions can be hardly estimated. This incurs random disk accesses which also causes performance degradation.

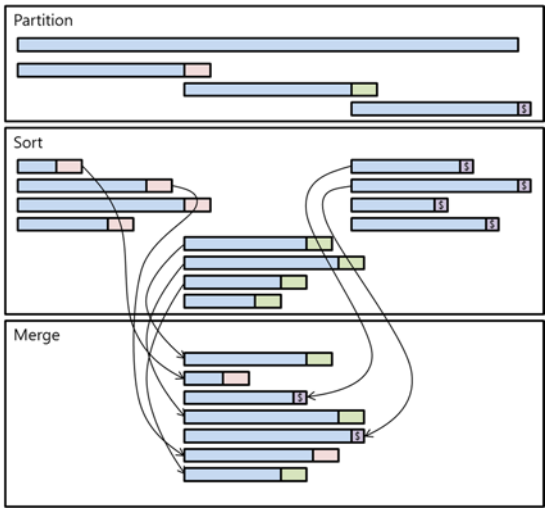

Figure 1. Sort and merge phases of DIGEST.

3 Parallel Algorithms

The major drawback of the algorithms in Section 2 is that they suffer from performance degradation due to random disk accesses and do not fully utilize the multi-core CPUs. To alleviate the drawback, FAST [4] performed disk accesses only in the serial manner and constructed the suffix subtrees concurrently for each partition. FAST divides the genome sequence according to the prefix of suffixes; the suffixes with the same prefix form a partition, and a suffix subtree is built for each partition. The suffix subtrees with different prefixes need not to be merged and thus can be built independently in parallel. FAST uses the naïve method for building a suffix subtree for each partition. The suffix subtree is empty initially and added the suffixes in the partition one by one. The subtree is built in a contiguous memory chunk, and the addition of a new suffix can be easily done by allocating a new area in the chunk and modifying a few pointers. When the suffix subtree building is complete, the chunk is stored in the disk without any further adjustments. Since the chunk is written sequentially, it is done very efficiently and

helps performance improvement of the overall indexing algorithm. When building suffix subtrees for multiple partitions in parallel, each subtree is built in a separate chunk. Since multiple suffix subtrees are built simultaneously, FAST achieves even higher performance improvement. FAST outperformed DIGEST [1] by up to 3.5 times and completed the indexing of entire human genome sequence in 36 minutes and 30 seconds.

Loh and Han [5] constructed a suffix array for selected suffixes in the human genome sequence and then converted it very efficiently into a suffix tree. Given a prefix length p, their algorithm divides all suffixes in the genome sequence into two groups; the first group consists of suffixes whose prefix of length p appears only once, and the remaining suffixes make up the second group. The prefix of length p is called a *window*, and the frequency of a window W is denoted as $Occ(W)$. A genome sequence X of length n contains $(n - p)$ windows, and the window at offset i is denoted as W_i. The algorithm builds two separate suffix subtrees for the windows W with $Occ(W) = 1$ and those with $Occ(W) > 1$ and merges them at the end. Each window W can be represented as an integer. At first, while scanning the genome sequence X, for each window W in X, $count[W]$ is incremented, where the array $count[]$ stores $Occ[W]$ and is set all 0 initially. After scanning the genome sequence, $count[]$ is scanned from the beginning to extract all the windows W with $count[W] = 1$. The extracted windows are all sorted; for any index i, j $(i < j)$ in the list, it holds that $W_i < W_j$, and any window W never appears again in the list. Thus, the list of suffixes S_i corresponding to the extracted windows W_i is a suffix array without any further processing. The suffix array can be converted into a suffix tree in $O(n)$ time. For the second group of the suffixes whose prefixes appear more than once, the suffix subtree is built using the naïve method as in FAST. Table 1 shows the number of windows with different frequencies for a few p values. The algorithm completed the construction of suffix tree for the entire human genome sequence in 17 minutes and 29 seconds and outperformed the previous FAST [4] and B^{2}ST [2] algorithms by up to 2.09 and 3.04 times, respectively.

Table 1. Number of windows with $Occ(W) = 1$ and $Occ(W) > 1$ [5].

Window size (p)	Windows with $Occ(W) = 1$	Windows with $Occ(W) > 1$
15	39,798,435 (1.4%)	2,792,561,477 (98.6%)
16	633,963,006 (22.4%)	2,198,396,906 (77.6%)
17	1,293,025,143 (45.7%)	1,539,334,769 (54.3%)
18	1,772,599,846 (62.6%)	1,059,760,066 (37.4%)

Acknowledgements. 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 (grant number: 2010-0025001).

References

[1] M. Barsky, U. Stege, A. Thomo, and C. Upton, "A new method for indexing genomes using on-disk suffix trees," In *Proc. ACM Conference on Information and knowledge Management (CIKM)*, Napa Valley, California, pp. 649-658, Oct. 2008.

[2] M. Barsky, U. Stege, and A. Thomo, "Suffix trees for inputs larger than main memory," *Information Systems*, Vol. 36, No. 3, pp. 644-654, May 2011.

[3] E. Hunt, M. P. Atkinson, and R. W. Irving, "Database Indexing for Large DNA and Protein Sequence Collections," *The VLDB Journal*, Vol. 11, No. 3, pp. 256-271, 2002.

[4] W.-K. Loh, Y.-S. Moon, and W. Lee, "A Fast Divide-and-Conquer Algorithm for Indexing Human Genome Sequences," *IEICE Trans. Information and Systems*, Vol. E94-D, No. 7, pp. 1369-1377, July 2011.

[5] W.-K. Loh and K.-S. Han, "A Fast Parallel Algorithm for Indexing Human Genome Sequences," *IEICE Trans. Information and Systems*, Vol. E97-D, No. 5, pp. 1345-1348, May 2014.

[6] B. Phoophakdee and M. J. Zaki, "Genome-scale Disk-based Suffix Tree Indexing," In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, pp. 833-844, Beijing, China, June 2007.

[7] Y. Tian, S. Tata, R. A. Hankins, and J. M. Patel, "Practical Methods for Constructing Suffix Trees," *The VLDB Journal*, Vol. 14, No. 3, pp. 281-299, 2005.

[8] E. Ukkonen, "On-line Construction of Suffix Trees," *Algorithmica*, Vol. 14, No. 3, pp. 249-260, Sept. 1995.

Received: May 1, 2014