PAPER

A Fast Divide-and-Conquer Algorithm for Indexing Human Genome Sequences*

Woong-Kee LOH[†a)], **Yang-Sae MOON**[††], ***Members***, ***and*** **Wookey LEE**[†††], ***Nonmember***

SUMMARY Since the release of human genome sequences, one of the most important research issues is about indexing the genome sequences, and the suffix tree is most widely adopted for that purpose. The traditional suffix tree construction algorithms suffer from severe performance degradation due to the memory bottleneck problem. The recent disk-based algorithms also provide limited performance improvement due to random disk accesses. Moreover, they do not fully utilize the recent CPUs with multiple cores. In this paper, we propose a fast algorithm based on 'divide-and-conquer' strategy for indexing the human genome sequences. Our algorithm nearly eliminates random disk accesses by accessing the disk in the unit of contiguous chunks. In addition, our algorithm fully utilizes the multi-core CPUs by dividing the genome sequences into multiple partitions and then assigning each partition to a different core for parallel processing. Experimental results show that our algorithm outperforms the previous fastest DIGEST algorithm by up to 10.5 times.

key words: *human genome sequences, indexing, suffix tree, memory bottleneck problem, divide-and-conquer, parallel processing*

1. Introduction

Due to recent advances in bio technology, genome sequences of diverse organisms including human beings have been collected into databases. The Human Genome Project (HGP), which were initiated in 1990, released the human DNA sequences of approximately 3 Gbp** size in 2003. Since the release, a lot of researches are under their way for harnessing the genome sequences. An essential research issue is about indexing large-scale genome sequences for efficient retrieving of genome subsequences of interest [1], [3], [7], [11], [18], [19], [22]. The suffix tree is most widely adopted for indexing genome sequences [3]–[5], [11], [18], [22]. In general, a suffix tree is created for a given string (or sequence) X and enables efficient exact matching and approximate matching on substrings of X [9]. We explain the suffix tree in more detail in Sect. 2.

Manuscript received December 27, 2010.

Manuscript revised March 10, 2011.

[†]The author is with the Department of Multimedia, Sungkyul University, Korea.

[††]The author is with the Department of Computer Science, Kangwon National University, Korea.

[†††]The author is with the Department of Industrial Engineering, Inha University, Korea.

*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).

a) E-mail: woong@sungkyul.ac.kr (Corresponding author)

DOI: 10.1587/transinf.E94.D.1369

**bp stands for 'base pair.' There are four bases, namely adenine (A), cytosine (C), guanine (G), and thymine (T).

A lot of algorithms have been proposed for efficient construction of the suffix tree. Ukkonen's algorithm [23] is the most famous one which, given a string of length n, constructs the corresponding suffix tree in $O(n)$ time. The algorithm implicitly assumes that n is small enough so that the input string and the output suffix tree can be loaded in the main memory as a whole. However, genome sequences could be several million or billion times larger than the strings dealt with by the traditional suffix tree construction algorithms such as Ukkonen's algorithm. Moreover, the suffix tree is about 10 ∼ 60 times larger than the input sequence [3], [18], [22]. Hence, the application of Ukkonen's algorithm for large-scale genome sequences should cause severe disk swap in and out, which is generally called *memory bottleneck problem* or *thrashing* [3]–[5], [11], [18], [22]. Actually, TOP-Q algorithm [4], an extension of Ukkonen's algorithm, took seven hours for constructing the suffix tree for genome sequences of 40 Mbp, which is much smaller than the human genome sequences, and it could not finish for genome sequences of 60 Mbp [18].

For coping with the memory bottleneck problem, a few disk-based algorithms have been proposed for constructing the suffix tree [3], [5], [11], [18], [22]. Disks have much larger size than main memory at the lower cost; however, they require much longer access time up to several hundred times. Hence, the disk-based algorithms are designed mainly to maximize the main memory utilization and the disk access efficiency. However, these algorithms have a common drawback that they incur random disk accesses. The disk access performance is dependent more on access patterns than access amount; even for accessing the same amount, the random disk access requires much more time than the sequential disk access. Thus, the disk-based algorithms have been improved to decrease the rate of random disk accesses.

Another problem of the previous disk-based algorithms is that they do not fully utilize the most up-to-date CPU technologies. Instead of raising the clock speed, recent CPUs are designed to have multiple, simultaneously running cores that enable intra-CPU parallel processing. However, some previous algorithms run mostly on a single core, and the others suffer from severe interference among the threads and hence have little gain by parallel processing. We explain the problems of the previous algorithms in more detail in Sect. 3.

In this paper, we propose a fast algorithm based on 'divide-and-conquer' strategy for constructing the suffix tree

Copyright © 2011 The Institute of Electronics, Information and Communication Engineers

for large-scale human genome sequences. The most significant difference from the previous algorithms is that the proposed algorithm nearly eliminates random disk accesses by accessing the disk in the unit of contiguous chunks each of which stores an entire suffix subtree. In addition, our algorithm fully utilizes multi-core CPUs by dividing the genome sequences into multiple, independent partitions and then assigning each partition to a different core for parallel construction of suffix subtrees. As an experimental result, our algorithm finished construction of the suffix tree for the entire human genome sequences in 64 minutes and outperformed DIGEST algorithm [3], which had previously been the fastest disk-based algorithm, by up to 10.5 times.

This paper is organized as the following. In Sect. 2, we briefly explain on the suffix tree. In Sect. 3, we explain on the previous disk-based suffix tree construction algorithms. In Sect. 4, we propose a new disk-based suffix tree construction algorithm, and then in Sect. 5, we discuss a few issues of our algorithm. In Sect. 6, we evaluate the performance of our algorithm through a series of experiments. Finally, we conclude this paper in Sect. 7.

2. Suffix Tree

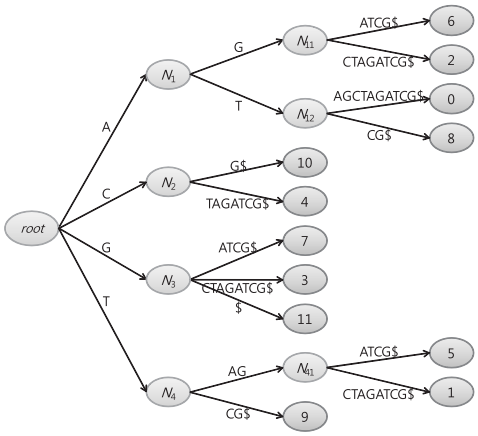
Figure 1 shows the suffix tree for a short DNA sequence X = ATAGCTAGATCG\$. The symbol '\$' is appended at the end of X so as to prohibit any suffix in X from being the prefix of any other suffix. Given a query sequence S, the search begins from the root node of the suffix tree. From the outbound edges of the root node, an edge e is chosen such that the label of e is the prefix of S. If no such edge is found, the search ends; if found, the child node N_e is visited by following the edge e, i.e., e is the inbound edge of N_e. Let l be the label length of e, $p_l(S)$ be the prefix of S of length l, and $s_l(S)$ be the suffix of S of length $Len(S) - l$. Then, it holds that $S = p_l(S) \oplus s_l(S)$, where $\oplus$ is the sequence concatenation operator. The search for query subsequence $s_l(S)$ begins recursively at the node N_e in the same manner as the root node. The search goes on until a terminal node is reached in the suffix tree or there is no query (sub)sequence to be searched for.

Let us take a query sequence S = AGATCG for example. In Fig. 1 (a), from the outbound edges of the root node, the edge with label 'A' is followed and then the node N_1 is visited. The search for query subsequence $s_l(S)$ = GATCG is performed recursively at the node N_1. The search continues until the terminal node with position 6 is reached; it indicates that query sequence S is found at position 6 in the sequence X. Figure 1 (b) shows the suffix tree whose edge labels are represented with (start, end) positions in X. While the labels' representation sizes in Fig. 1 (a) are arbitrary, those in Fig. 1 (b) are all identical.

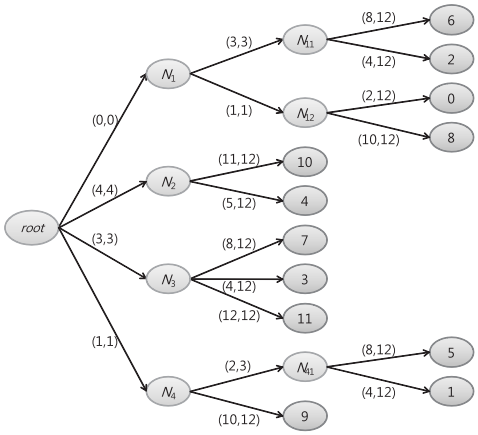
3. Related Work

Hunt et al. [11] proposed the first disk-based suffix tree construction algorithm. Hunt's algorithm excludes construction of suffix links, which caused severe memory bottleneck problem in Ukkonen's algorithm [23]. Hunt's algorithm divides the given genome sequences into partitions and then constructs a separate suffix subtree for each partition. Although Hunt's algorithm has $O(n^2)$ complexity, it shows better indexing performance than Ukkonen's algorithm by reducing disk accesses. However, Hunt's algorithm incurs heavy random disk accesses since it stores each node in the suffix tree as a separate object using the persistent Java object storage interface called PJama [2]. Actually, the algorithm was successful in indexing genome sequences of up to 286 Mbp size, but it could not be used for indexing the human genome sequences [11].

Tian et al. [22] presented the Top-Down Disk-based (TDD) approach for constructing disk-based suffix trees. TDD consists of two algorithms: Partition and Write Only Top Down (PWOTD) algorithm based on Wotd-eager algorithm [8] for constructing suffix trees and a memory buffer management algorithm for maximizing the performance of PWOTD algorithm. The performance of PWOTD algorithm highly depend on the settings of the memory buffer management algorithm [22]. Tian et al. [22] showed that TDD incurred only one sixth of disk accesses than DynaCluster algorithm [5], an extension of Hunt's algorithm, and that TDD



(a) Edge labels are represented with subsequences.



(b) Edge labels are represented with (start, end) positions in X.

Fig. 1 Suffix tree for a sequence X = ATAGCTAGATCG\$.

constructed the suffix tree for the entire human genome sequences in 30 hours. However, the memory buffer management algorithm in TDD assigns only a small portion of memory for keeping the suffix tree in main memory, while it assigns the largest portion to input genome sequences. TDD uses Least Recently Used (LRU) policy for swapping out the memory buffers into disk while constructing the suffix tree. Whenever PWOTD algorithm creates a new node N, it needs to access N's parent node P that could be previously stored far away from N. This causes random disk accesses, and the larger genome sequences should cause more random accesses.

Phoophakdee and Zaki [18] proposed an algorithm called TRELLIS, which eliminated data skewness among suffix subtrees by dividing genome sequences according to variable-length prefixes. Unlike Hunt's algorithm [11] and TDD [22], TRELLIS can create suffix links optionally after the suffix tree is constructed. TRELLIS consists of three phases: prefix creation, partitioning, and merging phases. In the prefix creation phase, variable-length prefixes are created so that, for each prefix P_j, the suffix subtree T_j corresponding to the suffixes having the prefix P_j can be loaded into main memory as a whole. In the partitioning phase, the entire genome sequences are divided into partitions so that each partition R_i and its corresponding suffix tree T_i can be loaded into main memory as a whole. Then, a suffix tree T_i is constructed for each partition in this phase. In the merging phase, for each prefix P_j created in the prefix creation phase, the suffix subtrees $T_{i,j}$ are extracted from the suffix trees T_i and then merged into a single suffix subtree T_j. Phoophakdee and Zaki [18] showed that TRELLIS outperformed TDD by up to 4 times and that it constructed the suffix tree for the entire human genome sequences in 4.2 hours. However, since TRELLIS extracts the suffix subtrees $T_{i,j}$ stored at random positions in the suffix trees T_i in the merging phase, it incurs severe random disk accesses. Actually, the merging phase requires the longest execution time [18].

Ghoting and Makarychev [7] proposed an algorithm called WAVEFRONT based on 'partition-and-merge' strategy as TRELLIS [18]. WAVEFRONT divides the entire data into I/O-efficient partitions and processes each partition independently. In [7], WAVEFRONT was extended to be executed on a massively parallel system. The algorithm completed indexing the entire human genome sequences in 15 minutes on IBM Blue Gene/L system composed of 1024 processors [7]. However, WAVEFRONT executed on a single processor showed no noticeable performance improvement compared with TRELLIS [18].

Barsky et al. [3] proposed an algorithm called DIGEST which consists of two phases similar to the merge-sort algorithm. In the first phase, the entire genome sequence is divided into partitions of the same length so that each partition can be loaded into main memory. For each partition, the suffixes contained therein are sorted in main memory and then are stored in disk. In the second phase, the suffixes sorted separately in each partition are merge-sorted. Suffix blocks from each partition are read sequentially one by one into main memory. The suffixes in different blocks are compared with each other, and the smallest one is extracted and then saved in the output block. When the output block becomes full, it is stored in disk. This continues until all the input blocks are empty. The sorted suffixes is called a *suffix array*, and it is known that a suffix array can be easily converted into a suffix tree [3], [21]. Barsky et al. [3] showed that DIGEST outperformed TRELLIS+ [19], an extension of TRELLIS [18], by up to 40% and that the algorithm completed indexing the entire human genome sequences in about 85 minutes. However, DIGEST should read suffix blocks from each partition stored at random positions in the second phase and hence suffers from severe random disk accesses. Moreover, since the merging phases of TRELLIS and DIGEST cannot be parallelized, they have little performance gain even by using recent multi-core CPUs.

4. Proposed Indexing Algorithm

In this section, we propose a new algorithm for indexing human genome sequences. The human genome is composed of 46 chromosomes: 22 chromosome pairs numbered $1 \sim 22$ and x/y (sex) chromosomes. In this paper, we concatenate the entire genome sequences into a single long sequence and use this sequence as the input of our algorithm. This helps simplify indexing and searching algorithms.

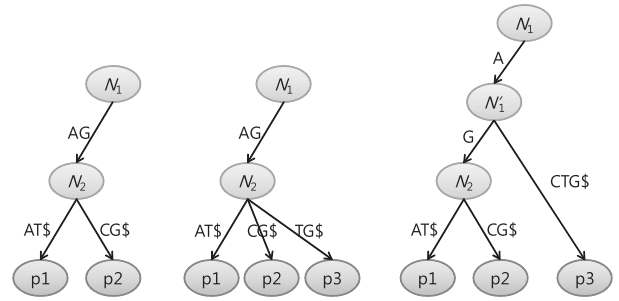
Our algorithm is designed based on divide-and-conquer strategy: it divides the entire human genome sequence into multiple independent partitions and then constructs the suffix tree separately for each partition. The suffix tree for each partition is constructed in a contiguous chunk in main memory. When the construction is completed, the chunk image is stored sequentially into disk as it is. Hence, unlike TRELLIS and DIGEST [3], [18], our algorithm has no performance degradation due to random disk accesses. Moreover, since the suffix trees for different partitions are constructed independently and are not merged thereafter, their construction can be done in parallel by fully utilizing the most up-to-date multi-core CPUs. According to these features, our algorithm achieves dramatic performance improvement compared with the previous algorithms.

Our algorithm represents each base as a 2-bit code as in [3], [18], [19], [24]; A, C, G, and T are represented as 00, 01, 10, and 11, respectively. Since the human genome sequence has the size of approximately 3 Gbp, the 2-bit coded sequence has the size of about 3 Gbp / 4 = 750 MB. Actually, after removing unidentified base pairs, the 2-bit coded sequence has the size of about 700 MB and can be fully loaded in main memory. Our algorithm assigns memory region for the full 2-bit coded genome sequence at the beginning and retains it to the end.

Our algorithm divides the human genome sequence into partitions according to prefixes, i.e., the suffixes having the common prefix belong to the same partition. We explain how to determine the prefixes for partitioning at the end of this section. The partitions are not necessarily cre-

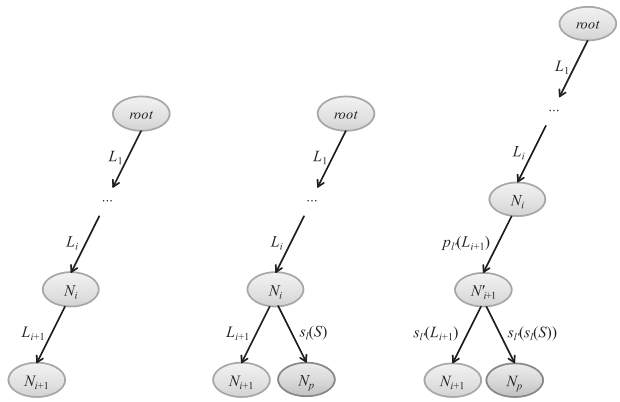
ated by physically dividing the genome sequence, but only the suffix positions are managed for each partition. The detailed procedure for constructing the lists of suffix positions is as follows. First, our algorithm counts the occurrence O_j ($0 \leq j < m$) of each prefix P_j, where m is the number of prefixes, while sequentially scanning the human genome sequence. For a suffix at position i ($0 \leq i < n$), if the suffix has prefix P_j, O_j is incremented by 1. Since the entire human genome sequence is loaded in main memory, this procedure is completed quickly. Then, our algorithm allocates m empty lists L_j for each prefix P_j. The size of L_j is $O_j \times 2$, and the positions of L_j are managed in an array of size m so that each L_j could be easily located. Each L_j is allocated in a contiguous memory region to read/write the list in a single operation and hence to eliminate random disk accesses. We never use hashing throughout our algorithm. Since we prefer smaller m, there exists no complicated memory management problem. Next, our algorithm scans the human genome sequence again and fills L_j with the positions of suffixes having prefix P_j; if the suffix at position i has prefix P_j, our algorithm appends the position into L_j. Instead of appending i directly, our algorithm appends the difference (> 0) of i from the last position in L_j for saving memory. The last positions in L_j are also managed in an array of size m, and hence it is easy to append a new value in L_j. We never perform any additional operation such as sorting on the suffixes in each partition. Since, for each suffix in the human genome sequence, its position appears exactly once in the lists, the size of the entire lists is roughly 3 Gbp × 2 bytes = 6 GB. In our experiment, we scanned the human genome sequence twice and used 3 GB of memory in each scan. Hence, we had no problem in managing main memory. This whole procedure is performed in main memory except saving the final lists of suffix positions into disk and hence is completed without any considerable burden.

When the creation of partitions (i.e., the lists of suffix positions in the human genome sequence) is completed, our algorithm constructs the suffix tree separately for each partition. At first, our algorithm creates an empty suffix tree without any node and then adds suffixes one by one into the suffix tree while scanning the corresponding list of suffix positions. Figure 2 shows an example of adding suffixes into a suffix tree. Figure 2 (a) shows a suffix tree before addition. Figure 2 (b) shows the result of adding a suffix S_1 = AGTG\$ into the suffix tree in Fig. 2(a). S_1 has the prefix $p_2(S_1)$ = AG of length 2 which matches the label of the outbound edge of N_1 and then $s_2(S_1)$ = TG\$ does not have common prefix with any label of the outbound edges of N_2. In this case, our algorithm creates a new outbound edge e of N_2 and labels it with $s_2(S_1)$ = TG\$. The edge e is connected to a new terminal node p_3, i.e., e becomes the inbound edge of p_3. Figure 2 (c) shows the result of adding a suffix S_2 = ACTG\$ into the suffix tree in Fig. 2 (a). The label of the outbound edge of N_1 partially matches the prefix $p_1(S_2)$ = A of S_2. In this case, our algorithm cuts the outbound edge of N_1 and adds a new internal node N'_1; the inbound edge of N'_1 has the label $p_1(S_2)$ = A. A new outbound edge e is added to node N'_1 and is labeled with $s_1(S_2)$ = CTG\$. The edge e is connected to a new terminal node p_3, i.e., e becomes the inbound edge of p_3.



(a) A suffix tree. (b) Adding a suffix S_1 = AGTG\$. (c) Adding a suffix S_2 = ACTG\$.

Fig. 2 Example of adding suffixes into a suffix tree.



(a) Before adding a suffix. (b) The case of adding a terminal node. (c) The case of adding an internal node and a terminal node.

Fig. 3 Generalization of adding suffixes into a suffix tree.

Each time a suffix is added into the suffix tree, a new terminal node is created in the tree. Since every suffix ends with the symbol \$, the suffix cannot be a prefix of any other suffixes and has a unique position in the human genome sequence. Hence, a terminal node should exist in the suffix tree for representing the unique position of each suffix. The terminal node should have an inbound edge in the tree. The edge is an outbound edge of either (1) an existing node (Fig. 2 (b) case) or (2) a new node added between the cut edges (Fig. 2 (c) case). There exist no other cases.

Figure 3 shows the generalization of adding suffixes into the suffix tree by our algorithm. Let us assume that we have visited the node N_i in the course of searching for a suffix S in Fig. 3 (a). The concatenation $L = L_1 \oplus \cdots \oplus L_i$ of edge labels from the root node to N_i should be the same as the prefix $p_l(S)$ of length $l = Len(L)$, i.e., $L = p_l(S)$. In case $L_{i+1} \cap s_l(S) = \emptyset$, an edge e labeled with $s_l(S)$ and a new terminal node N_p with the inbound edge e are added as in Fig. 3 (b). In case $L_{i+1} \cap s_l(S) = L'$ ($\neq \emptyset$), a new internal node N'_{i+1} and a new terminal node N_p are added as in Fig. 3 (c), where $l' = Len(L')$ and $p_{l'}(L_{i+1}) = p_{l'}(s_l(S)) =$

Fig. 4 Data structure of our algorithm: the information on a node and its inbound edge is contained together.

L'. Since the suffix always ends with \$, we cannot have the case $S = L$ in Fig. 3.

Figure 4 shows the data structure of our algorithm. As shown in the figure, the information on a node and its inbound edge is contained together in a single data structure. The fields a and b represent the start and end positions of the inbound edge in the human genome sequence as shown in Fig. 1 (b). The field *right* contains the pointer to the next sibling node, and *foo* represents either (1) a pointer to the leftmost child node in case of an internal node or (2) the suffix position in the genome sequence in case of a terminal node. The field *misc* contains miscellaneous information on the node. The fields a, b, *right*, and *foo* are 4-byte unsigned integers, while the field *misc* is a 2-byte unsigned integer. Hence, the data structure has the fixed length of 18 bytes. For distinguishing between the internal and terminal nodes, the field b is investigated. If $b = n$, where n is the length of genome sequence, it is a terminal node; if $b < n$, it is an internal node (refer to Figure 1 (b)). The fields in the node structure in Fig. 4 have the primitive data types provided by C/C++ standards. The data types are machine/compiler-independent, i.e., their sizes never change according to machine and compiler.

We can efficiently construct the suffix trees using the data structure in Fig. 4. We explain this using Fig. 5, which shows the representation of suffix trees in Fig. 3 using the data structure; Figs. 5 (a) ∼ 5 (c) correspond to Figs. 3 (a) ∼ 3 (c), respectively. In Fig. 5 (a), the fields (a_i, b_i) and (a_{i+1}, b_{i+1}) represent the start and end positions of labels L_i and L_{i+1}, respectively. The fields with X stand for "don't care" fields, which are not used nor updated here. The arrow indicates a pointer to a possibly distant node. The nodes N_i and N_{i+1} may not be adjacent as shown in the figure, though N_{i+1} is easily accessed by following the pointer. Figure 5 (b) shows the case a new terminal node N_p is added. The node N_{i+1} can be either an internal or a terminal node and is a sibling node of N_p. In the figure, the leftmost child node of N_i has been changed from N_{i+1} to N_p. This is because we can efficiently add N_p as a new child node of N_i without accessing N_{i+1} and all its sibling nodes. Figure 5 (c) shows the case a new internal node N'_{i+1} and a new terminal node N_p are added. The field values of the N_{i+1} are copied to the newly allocated node region, and then the field a_{i+1} is adjusted (b_{i+1} is not changed). The field values of N'_{i+1} are set in the region previously used by N_{i+1} as shown in the figure. The node N_p is a sibling node of N_{i+1} and is added as the leftmost child node of N'_{i+1} as in Fig. 5 (b). The key idea we would like to show in Fig. 5 is that, when a suffix is added, there is only slight modification in the suffix tree constructed so far; it can be done only by allocating new

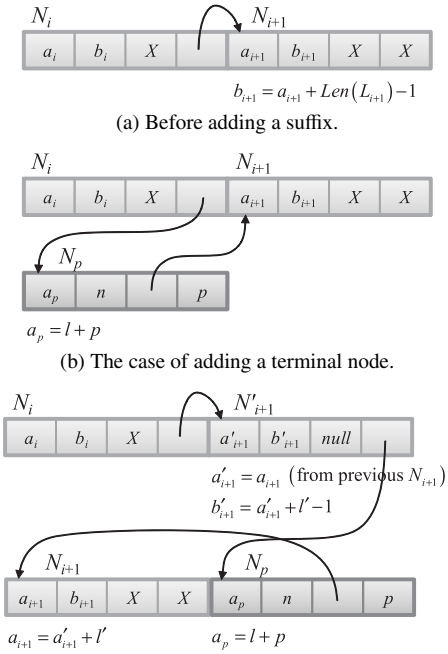


Fig. 5 Data structures corresponding to the suffix trees in Fig. 3.

memory region(s) for one or two nodes and then setting a few appropriate field values therein. This is one of the features providing the efficiency of our algorithm.

Our algorithm constructs a suffix tree in a main memory chunk. Allocations of memory regions for new nodes (and their inbound edges) are made sequentially in the chunk. The pointers in Figs. 4 and 5 are relative offset values from the beginning of the chunk. Once the construction of a suffix tree is completed, our algorithm stores the chunk image into disk without any modification. When the chunk image is reloaded into main memory, the pointers are still valid regardless of where it is reloaded. Since the chunk image is stored in and read from the disk sequentially, there is no performance degradation due to random disk accesses, and thus we have significantly improved performance. When multiple suffix trees are constructed in parallel, our algorithm allocates a separate memory chunk for each suffix tree. Even in this case, the human genome sequence is loaded only once into the memory region shared by the simultaneous processes of our algorithm. This parallel processing enables more significant performance improvement.

We now explain how to determine the prefixes for dividing the human genome sequence into partitions. Each suffix in the genome sequence is assigned to a partition according to its prefix; every suffix in a partition has a common prefix. Given a prefix length p, our algorithm creates a partition for each possible prefix of length p. The number of partitions is 4^p. A weakness of this scheme is that it causes data skewness among the partitions [18]; there may be big differences among the sizes of partitions and hence the corresponding suffix trees. We tackle this weakness as follows. As p increases, the number of suffixes in each par-

tition decreases, and the size of corresponding suffix tree also decreases. We set p to be large enough to make the suffix tree sizes smaller than the size M of available main memory. Then, the simultaneous processes of our algorithm choose the partitions so that the estimated sizes of their corresponding suffix trees sum up very close to M. This can be done with simple computations. By fully utilizing main memory in this way, our algorithm achieves better indexing performance.

The minimum length of prefixes is computed approximately using the following Eq. (1):

$$p_{\min} = \left\lceil \log_4 \frac{n \cdot f}{M} \right\rceil , \quad (1)$$

where n is the length of human genome sequence and f is a multiplication factor to estimate the suffix tree size. M represents the size of remaining main memory after loading the entire 2-bit coded human genome sequence. f is defined as the maximum of $\frac{T}{s}$, where s is the length of a genome sequence and T is the size of the corresponding suffix tree. We estimate the size of a big suffix tree by test construction of small suffix trees. The f value greatly differs according to suffix tree construction algorithms and is about $30 \sim 32$ in our algorithm.

5. Discussion

The index proposed in this paper is identical to the traditional suffix tree except that the proposed index consists of multiple suffix subtrees that are created and saved in disk separately from each other. Each suffix subtree corresponds to a partition of the human genome sequence, which is composed of the suffixes having the same prefix of length p. In general, any two suffixes are contained in the same suffix subtree if and only if they have a common prefix. The suffixes with different prefixes are contained in different suffix subtrees and hence can be processed separately. Figure 6 shows an example of the proposed index for $p = 2$. The leftmost suffix subtree T_0 corresponds to prefix P_0 = 'AA', and the entire suffixes starting with 'AA' are contained in T_0. Given a query subsequence S, if S has the prefix P_0, the search starts by accessing T_0 (and optionally a few more subtrees corresponding to approximate prefixes such as 'AC'). Since the proposed index is designed to eliminate random disk accesses, there should be minimal performance degradation due to disk accesses.

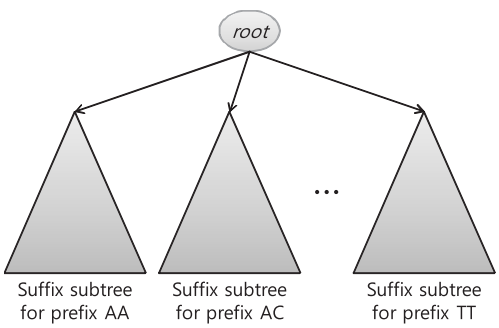


Fig. 6 Architecture of the proposed index.

Since the proposed index is identical to the traditional suffix tree, it can be used in all kinds of suffix tree applications such as exact matching, approximate matching (allowing a pre-specified number of mismatches), and finding frequent substrings, common substrings, and maximal palindromes [9], [11]. Many of these applications traverse the suffix tree in depth-first manner. While running the applications, the suffix subtrees are accessed sequentially and no subtree is accessed more than once. An important feature of the suffix tree is that the subsequences appearing frequently in the target sequence are mapped only to a single path in the corresponding suffix tree [16], [17], and hence we don't have any problem dealing with such subsequences.

The prefix length p determines the number of partitions and has no effect on search accuracy. We prefer smaller p close to $p_{\min}$ given in Eq. (1) and set $p = 4$ in our experiments in Sect. 6. Given p, our algorithm creates 4^p partitions, each of which is processed in a separate process. The number of parallel processes effective for improving indexing performance is bound to the number of cores in CPU, which is 4 in our experiments. We could obtain no noticeable performance improvement by running more processes. Hence, p needs not to be large on the assumption of sufficient main memory.

The proposed index has the following strengths over the existing compressed suffix array [6], [14], [15]. First, since the proposed index is identical to the traditional suffix tree, it can be used in diverse applications including exact matching, approximate matching, and finding frequent substrings, common substrings, and maximal palindromes [9], [11]. Indeed, Rocke [20] and Kurtz and Schleiermacher [13] used the suffix tree for retrieving gapped motifs and maximal repeats, respectively. Second, the proposed algorithm supports faster search than the suffix array. For exact matching, the suffix tree has $O(q + x)$ complexity, where q is the length of query subsequence and x is the number of matching subsequences, while the suffix array has $O(q + \log n + x)$ complexity [6], [10]. Although the compressed suffix array has smaller size of memory, it requires additional processing due to the compression. The FM-index, which is proposed by Ferragina and Manzini [6] based on Burrows-Wheeler Transform (BWT), has $O(q + x \log^e n)$ complexity for exact matching. Since e is a positive number, the complexity of FM-index is higher than the suffix tree. Third, it requires a lot of time to construct the compressed suffix array. Indeed, the compressed suffix array for Bowtie algorithm [14] took 4 hours and 36 minutes for its construction even on a high-end server with 16 GB main memory (there exists a trade-off between construction time and main memory size [14].), and the performance improvement rate is sub-linear to main memory size. The weakness of the suffix tree is that it occupies a large size of volume. This is an inherent problem; however, in this paper, we coped with that by accessing the suffix subtrees in the unit of chunks. We dramatically re-

duced disk access and CPU processing time, and hence only minimal performance degradation was incurred due to disk accesses.

Our algorithm for constructing the suffix subtree for each partition has worst-case complexity $O(s^2)$, where s is the number of suffixes in the partition. However, the suffix subtrees for the human genome sequence are close to balanced trees, whose depths are $\log s$, rather than skewed ones. Hence, we claim that the average-case complexity $O(s \log s)$ describes the performance of our algorithm more accurately. The average-case complexity is obtained as follows. When adding a new suffix into the suffix subtree, our algorithm searches for the suffix in the subtree to find the place to add the suffix therein. This search is performed in the same manner as exact matching, and there is no need to traverse the whole subtree. This search can proceed up to the depth of the subtree, which is $\log s$ on the average. Hence, the average-case search complexity for a suffix is $O(\log s)$, and that for the entire suffixes is $O(s \log s)$. Actually, at most stages of construction, the suffix subtree should have the depth less than $\log s$, and therefore it is expected that the search performance should show almost linear trend to the number of suffixes s.

Although a process accesses the disk for a very short time while constructing the suffix subtree, as the number of such processes increases, the probability of disk access contention among them should also increase. The probability as a function of the number of processes is obtained as follows. Let r ($0 < r \leq 1$) be the ratio of disk access time divided by the entire suffix subtree construction time by a single process. Then, r can also be considered as the probability of disk accesses by the process. Let t be the number of such processes running in parallel. In this circumstances, the probability of simultaneous disk accesses by i ($1 \leq i \leq t$) of t processes is $\binom{t}{i} r^i$. Hence, the probability R of disk access contention by t processes is computed as follows:

$$R = \sum_{1 \leq i \leq t} \binom{t}{i} r^i \ . \tag{2}$$

We call R as *contention rate* in this paper. The value r is dependent on target data and system environment; r was less than 0.01 in our experiments. Figure 7 shows a graph of contention rate R against the number of processes t by setting $r = 0.01$. The maximum t was 4 in our experiments and should also be small in different environments. If the disk access contention becomes the bottleneck on the indexing performance, we can work around by using multiple disks. Therefore, disk access contention can never be an issue with our algorithm.

We do not use Ukkonen's algorithm [23] for the following reason. Ukkonen's algorithm uses *suffix links* to construct a suffix tree in $O(n)$ time. A suffix link is attached to every node in a suffix tree; the suffix link of a node corresponding to a string $\chi\alpha$, where χ is a single character and α is a string (possibly empty), leads to a node corresponding to a string α. The algorithm constructs the suffix tree while following the suffix links. However, it is highly probable that the strings $\chi\alpha$ and α have different prefixes and hence are contained in different partitions. In this case, by following the suffix links, we should access the suffix subtrees corresponding different partitions, which incurs severe random disk accesses. Moreover, we cannot construct the suffix subtrees separately in parallel. Therefore, Ukkonen's algorithm is not adequate for our divide-and-conquer approach.

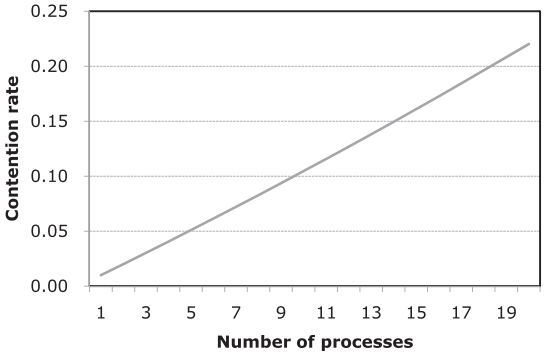


Fig. 7 Contention rate against the number of parallel processes.

6. Performance Evaluation

In this section, we show the superiority of our algorithm through a series of experiments. We use the same data sets as those in [3]. The first set is a short genome sequence of 110 Mbp size obtained from 6643 organisms. The second set is the entire human genome sequence of about 3 Gbp size. These data sets are denoted as VDB and HG18, respectively.

The hardware platform is a PC equipped with Intel Core2Quad Q9550 2.83 GHz CPU, Samsung DDR3 8 GB main memory, and a 500 GB 7200 rpm hard disk. The software platforms are Ubuntu 10.10 32 bit Linux and Windows 7 64 bit Edition. The first experiment was performed on Ubuntu as in [3], and the second and third experiments were performed on Windows 7. The latter two experiments were also performed on Ubuntu, though we had $10 \sim 15\%$ better performance on Windows 7. As C/C++ compilers, we used GNU C++ 4.4.5 on Ubuntu and Visual C++ 2010 Express Edition on Windows 7.

In the first experiment, we compared the performance of our algorithm with DIGEST [3], which had been the fastest disk-based suffix tree construction algorithm. We downloaded the source code of DIGEST from the author's web site [†]. In this experiment, we ran our algorithm and DIGEST on VDB data set and compared their elapsed time for constructing the suffix trees [††]. The time for constructing the lists of suffix positions is also included in the elapsed time. Figure 8 shows the result of experiment; our algorithm

[†]http://webhome.cs.uvic.ca/ mgbarsky/

[††]We also tried the experiment on HG18 data set; however, DIGEST always terminated abnormally with the segmentation fault error. We discussed on this with the author of DIGEST, but we could not solve the problem to the end.

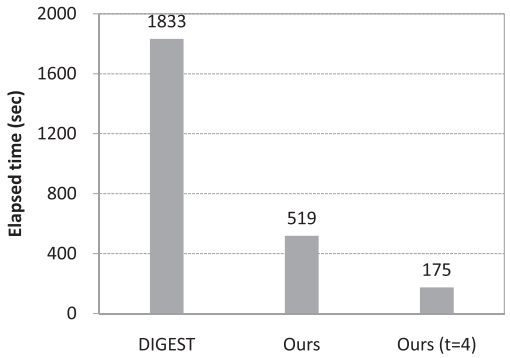
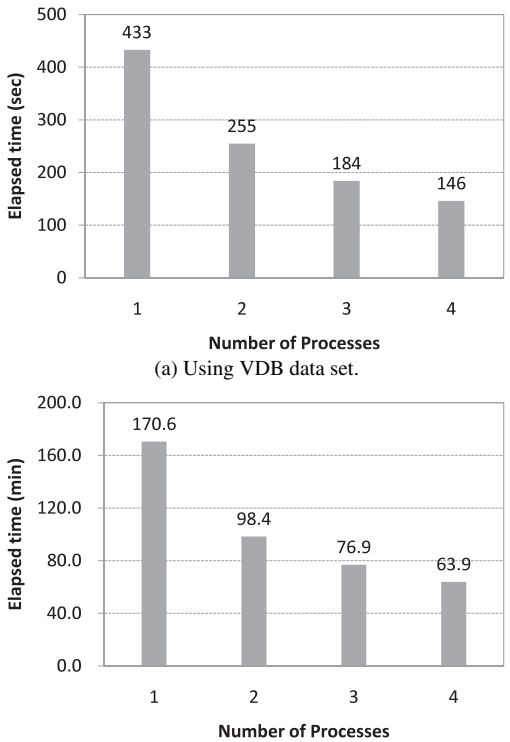


Fig. 8 Result of first experiment: our algorithm outperformed DIGEST by up to 10.5 times.



(a) Using VDB data set.

(b) Using HG18 data set.

Fig. 9 Result of second experiment: we could obtain performance improvement by up to 3.0 times by running four parallel processes.

with four parallel processes outperformed DIGEST by up to 10.5 times.

In the second experiment, we ran our algorithm on both VDB and HG18 data sets and compared the elapsed time for various numbers of parallel processes of our algorithm. Figure 9 shows the experimental result. Since the hardware platform has a four-core CPU, we increased the number of parallel processes up to four. Actually, we obtained almost no performance improvement by running more than four parallel processes on the same platform. Note that the units of vertical axes are seconds and minutes in Figs. 9 (a) and 9 (b), respectively. As shown in the figures, we obtained performance improvement by up to 3.0 times by running four parallel processes compared with a single process. We could not obtain four times performance improvement mostly due to inter-process communication and synchronization. Since our algorithm is designed to minimize the effect of disk accesses, it has the potential of greater performance improvement by using advanced CPUs with more cores and faster clock speeds.

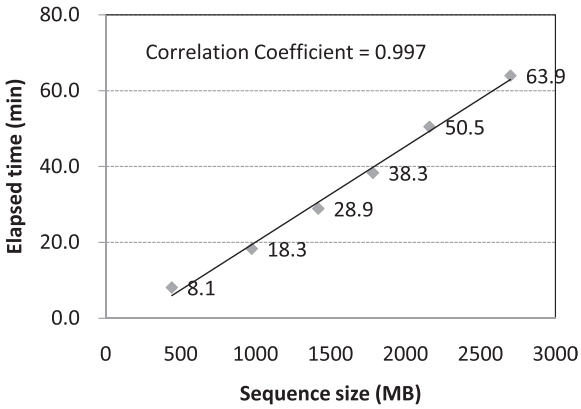


Fig. 10 Result of third experiment: elapsed time has almost linear correlation with the size of genome sequences.

In the third experiment, we measured the elapsed time of our algorithm for various sizes of genome sequences. We ran four processes on the genome sequences consisting of the first 2, 5, 8, 11, 15, and 24 chromosomes in the human genome sequence. Figure 10 shows the result. As the result of regression analysis on the experimental result, we could find that the elapsed time is almost linearly correlated with the size of genome sequences.

7. Conclusions

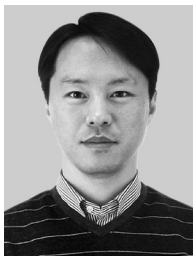
In this paper, we proposed a divide-and-conquer algorithm for constructing the suffix tree for human genome sequences. The most significant difference from the previous algorithms is that our algorithm nearly eliminates random disk accesses by accessing the disk in the unit of contiguous chunks. In addition, our algorithm fully utilizes multi-core CPUs by dividing the genome sequences into separate partitions and then assigning each of them to a parallel process for construction of the corresponding suffix subtree. As an experimental result, our algorithm finished construction of the suffix tree for the entire human genome sequence in 64 minutes and outperformed the previously fastest DIGEST algorithm by up to 10.5 times. We believe that our algorithm should achieve higher performance improvement than the others with the advance of hardware technology.

References

[1] S. Altschul, T. Madden, A. Schaffer, J. Zhang, W. Miller, and D. Lipman, "Gapped BLAST and PSI-BLAST: A new generation of protein database search programs," Nucleic Acids Research, vol.25, no.17, pp.3389–3402, 1997.

[2] M. Atkinson and M. Jordan, "Providing orthogonal persistence

for Java," Proc. European Conf. on Object-Oriented Programming (ECOOP), pp.383–395, Brussels, Belgium, July 1998.
[3] M. Barsky, U. Stege, A. Thomo, and C. Upton, "A new method for indexing genomes using on-disk suffix trees," Proc. ACM Conference on Information and Knowledge Management (CIKM), pp.649–658, Napa Valley, California, Oct. 2008.
[4] S.J. Bedathur and J.R. Haritsa, "Engineering a fast online persistent suffix tree construction," Proc. Int'l Conf. on Data Engineering (ICDE), IEEE, pp.720–731, Boston, Massachusetts, March 2004.
[5] C.-F. Cheung, J. Yu, and H. Lu, "Constructing suffix tree for gigabyte sequences with megabyte memory," IEEE Trans. Knowl. Data Eng., vol.17, no.1, pp.90–105, Jan. 2005.
[6] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," Proc. Annual Symp. Foundations of Computer Science (FOCS), pp.390–398, Redondo Beach, California, Nov. 2000.
[7] A. Ghoting and K. Makarychev, "Serial and parallel methods for i/o efficient suffix tree construction," Proc. Int'l Conf. Management of Data, ACM SIGMOD, pp.827–840, Providence, Rhode Island, June 2009.
[8] R. Giegerich, S. Kurtz, and J. Stoye, "Efficient implementation of lazy suffix trees," Software: Practice and Experience (SPE), vol.33, no.11, pp.1035–1049, 2003.
[9] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, 1997.
[10] W.-K. Hon, R. Shah, S.V. Thankachan, and J.S. Vitter, "On entropy-compressed text indexing in external memory," Proc. String Processing and Information Retrieval Symposium (SPIRE), pp.75–89, Saariselka, Finland, Aug. 2009.
[11] E. Hunt, M.P. Atkinson, and R.W. Irving, "Database indexing for large DNA and protein sequence collections," VLDB Journal, vol.11, no.3, pp.256–271, 2002.
[12] S. Kurtz, "Reducing the space requirement of suffix trees," Software: Practice and Experience (SPE), vol.29, no.13, pp.1149–1171, Nov. 1999.
[13] S. Kurtz and C. Schleiermacher, "REPuter: Fast computation of maximal repeats in complete genomes," Bioinformatics, vol.15, no.5, pp.426–427, May 1999.
[14] B. Langmead, C. Trapnell, M. Pop, and S.L. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," Genome Biology, vol.10, no.3, pp.R25.1–R25.10, March 2009.
[15] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform," Bioinformatics, vol.25, no.14, pp.1754–1760, July 2009.
[16] H. Li and R. Durbin, "Fast and accurate long-read alignment with Burrows-Wheeler transform," Bioinformatics, vol.26, no.5, pp.589–595, March 2010.
[17] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang, "SOAP2: An improved ultrafast tool for short read alignment," Bioinformatics, vol.25, no.15, pp.1966–1967, Aug. 2009.
[18] B. Phoophakdee and M.J. Zaki, "Genome-scale disk-based suffix tree indexing," Proc. Int'l Conf. on Management of Data, pp.833–844, ACM SIGMOD, Beijing, China, June 2007.
[19] B. Phoophakdee and M.J. Zaki, "TRELLIS+: An effective approach for indexing genome-scale sequences using suffix trees," Proc. Pacific Symp. on Biocomputing, pp.90–101, Kohala Coast, Hawaii, Jan. 2008.
[20] E. Rocke, "Using Suffix trees for gapped motif discovery," Proc. Annual Symp. on Combinatorial Pattern Matching (CPM), pp.335–349, Montreal, Canada, June 2000.
[21] R. Sinha, S. Puglisi, A. Moffat, and A. Turpin, "Improving suffix array locality for fast pattern matching on disk," Proc. Int'l Conf. on Management of Data, ACM SIGMOD, pp.661–672, Vancouver, Canada, June 2008.
[22] Y. Tian, S. Tata, R.A. Hankins, and J.M. Patel, "Practical methods for constructing suffix trees," VLDB Journal, vol.14, no.3, pp.281–299, 2005.
[23] E. Ukkonen, "On-line construction of suffix trees," Algorithmica, vol.14, no.3, pp.249–260, Sept. 1995.
[24] J.-I. Won, S.-K. Hong, J.-H. Yoon, S. Park, and S.-W. Kim, "A practical method for approximate subsequence search in DNA databases," Proc. Pacific-Asia Conf. on Advances in Knowledge Discovery and Data Mining (PAKDD), pp.921–931, Nanjing, China, May 2007.



Woong-Kee Loh received his B.S. (1991), M.S. (1993), and Ph.D. (2001) degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), South Korea. From April 2005 through May 2006, he was a visiting professor in Dept. of Computer Science, KAIST. From June 2006 through July 2007, he was a visiting scholar in Dept. of Computer Science & Engineering, University of Minnesota, USA. Currently, he is an assistant professor at Sungkyul University, South Korea. He has served as a PC member for DaWaK, PAKDD, and CIKM. He is a member of the ACM and IEICE. His research interests include large-scale data mining, knowledge discovery, data warehousing, genome database indexing and retrieval, and fault detection and analysis (FDA) in semiconductor manufacturing processes.

Yang-Sae Moon received his B.S. (1991), M.S. (1993), and Ph.D. (2001) degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST). From 1993 to 1997, he was a research engineer in Hyundai Syscomm, Inc., where he participated in developing 2 G and 3 G mobile communication systems. From 2002 to 2005, he was a technical director in Infravalley, Inc., where he participated in planning, designing, and developing CDMA and W-CDMA mobile network services and systems. He is currently an associate professor at Kangwon National University. He was a visiting scholar at Purdue University from 2008 to 2009. His research interests include data mining, knowledge discovery, storage systems, access methods, multimedia information retrieval, mobile/wireless communication systems, and network communication systems. He is a member of the IEEE, ACM.

Wookey Lee received his B.S., M.S., and Ph.D. degrees from Seoul National University, South Korea, and his M.S.E. degree from Carnegie Mellon University, USA. He has been a visiting professor in Dept. of Computer Science, University of British Columbia, Canada. He won the best paper awards in KORMS '04, '10, and KIISE '09. Currently, he is a Professor at Inha University, South Korea, and the editor-in-chief of Journal of Information Technology and Architecture. He has served as a PC member for APWeb, C^3S^2E, CIKM, IEEE DEST, iiWAS, ITA, and WAIM, and an Organizing Committee member for DASFAA, ICUIMC, ITA/EA, SRDS, and VLDB, etc. His research interests are Web information retrieval, mobile & multimedia databases, Web structuring, and Data Warehousing, etc.