A Fast Parallel Algorithm for Indexing Human Genome Sequences*

**SUMMARY** A suffix tree is widely adopted for indexing genome sequences. While supporting highly efficient search, the suffix tree has a few shortcomings such as very large size and very long construction time. In this paper, we propose a very fast parallel algorithm to construct a disk-based suffix tree for human genome sequences. Our algorithm constructs a suffix array for part of the suffixes in the human genome sequence and then converts it into a suffix tree very quickly. It outperformed the previous algorithms by Loh et al. and Barsky et al. by up to 2.09 and 3.04 times, respectively.

**key words:** human genome sequences, suffix tree, parallel algorithm, suffix array, disk-based index

### 1. Introduction

Due to the recent remarkable advances in DNA sequencing technology, the time and cost to determine the sequences of base pairs (bp) in DNA is rapidly decreasing. Sequencing results are used in many areas such as molecular biology and genetics. To make the most of such sequencing results, we provide various kinds of search algorithms on the sequences, and thus we also need an efficient indexing scheme. The suffix tree is widely adopted for indexing genome sequences [8], [9] and actually used in many applications such as sequence alignment and retrieving gapped motifs and maximal repeats.

While supporting highly efficient search, the suffix tree also has a few shortcomings such as very large size and very long construction time. For a sequence $X$, the corresponding suffix tree is 10-60 times the size of $X$. Hence, for the entire human genome sequence of 3 Gbp size, the suffix tree should have a few hundred GB size. Since such a big suffix tree cannot be handled only in main memory, several algorithms have been proposed for the construction of disk-based suffix trees [1], [2], [7]–[9].

In this paper, we propose a very fast parallel algorithm to construct a disk-based suffix tree for human genome sequences. Our algorithm is an extension of that by Loh et al. [7], which is called FAST in this paper. It constructs a suffix array for part of the suffixes in the human genome sequence and then converts it into a suffix tree very quickly. Our algorithm completed the construction of suffix tree for the entire human genome sequence in 17 minutes and 29 seconds, and outperformed the previous FAST and B²ST [2] algorithms by up to 2.09 and 3.04 times, respectively.

### 2. Related Work

A genome sequence $X = x_0 \ldots x_{n-1}$ of length $n$ contains $n$ suffixes $S_i = x_i \ldots x_{n-1}$ ($0 \leq i < n$). The last symbol ‘$’ is appended to prohibit any suffix in $X$ from being a prefix of the other suffix and is needed for efficiency of indexing and searching [5]. Figure 1 shows the suffix tree for a short genome sequence $X = ATAGCTAGATCG$. Every node except the root has two or more outbound edges and only one inbound edge, and every edge is associated with a label. For a node $N$, the concatenation of labels associated to the edges along the path from the root to $N$ is defined as path label of $N$, and the length of path label is defined as string depth of $N$ [5]. For example, the path label of a node $N_{41}$ in Fig. 1 is ‘TAG’ and the string depth is 3.

Searching with a suffix tree starts from the root and branches to the outbound edge whose label has the common prefix with the query (sub)sequence. For example, in search of a query sequence $Q = AGATCG$ with the suffix tree in Fig. 1, the tree is traversed in the order of internal nodes $N_1$, $N_{11}$ and a leaf node 6. In general, the suffix tree for a sequence $X$ of length $n$ has $n$ leaf nodes; each leaf node $i$ has one-to-one correspondence with a suffix $S_i$ in $X$. In Fig. 1,
we can see that $Q$ is located at position 6 in $X$. In actual implementation of suffix trees, edge labels are represented with their start and end positions in $X$ for consistent storage utilization. For example, the inbound edge of $N_{41}$ is represented as $(2, 3)$.

Recently, there have been proposed several algorithms to construct disk-based suffix trees [1], [2], [8], [9]. They consist of three phases; they divide the human genome sequence into multiple partitions, then construct a suffix subtree independently for each partition, and finally merge the subtrees into a single suffix tree. Although they have different partitioning and merging methods, the partitions are determined so that the suffix subtree for a partition could be constructed only in main memory. Their drawback is that they suffer from performance degradation due to random disk accesses and do not fully utilize recent multi-core CPUs. Their merging phases can hardly be parallelized and hence causes severe overhead on the performance. To tackle these problems, FAST [7] performed only sequential disk accesses in the unit of chunk and constructed the suffix subtrees in parallel for each partition that needs not to be merged. FAST completed the indexing of entire human genome sequence in 36 minutes and 30 seconds.

3. Proposed Algorithm

In this section, we propose a very fast algorithm to construct the suffix tree for human genome sequences. Our algorithm is a parallel algorithm that makes the most of the recent multi-core CPUs. We first present single-thread version and then extend it to multi-thread version.

Our algorithm is given an argument $p$ and divides the suffixes in the genome sequence according to the occurrence count of their prefixes of length $p$. In this paper, the prefix of length $p$ is called a window, and the occurrence count of a window $W$ is denoted as $Occ(W)$. A genome sequence $X$ of length $n$ contains $(n - p)$ windows, and the window at position $i$ is denoted as $W_i = [x_i, \ldots, x_{i+p-1}]$. Our algorithm divides the windows with $Occ(W) = 1$ and those with $Occ(W) > 1$ and then constructs suffix trees for each partition of windows in different ways.

Our algorithm consists of two phases. In the first phase, it constructs the suffix tree for the suffixes containing single occurrence windows. For the purpose, all the windows $W$ with $Occ(W) = 1$ should be found in the genome sequence, which is performed as follows. Our algorithm first allocates a one-dimensional array $\text{count}$ of $4^p$ elements in main memory and then initializes all the elements to zero. Then, while scanning the entire genome sequence, it increments $\text{count}[W_i]$ by 1 for each window $W_i$ ($0 \leq i < n - p$).

As in previous indexing algorithms [1], [2], [7]–[9], our algorithm represents genome bases A, C, G, and T as two-bit codes 00, 01, 10, and 11, respectively. Hence, a window of given length $p$ can be represented as an integer of $2p$ bits or $[2p/8]$ bytes. In our implementation, a window is represented as a value of unsigned int type for operational efficiency under the assumption of $[2p/8] \leq 4$ or $p \leq 16$.

Actually, since the elements in $\text{count}$ has unsigned int type, the whole memory space occupied by $\text{count}$ is $4^p \times 4$ bytes. However, the size of an array cannot exceed 2 GB, and hence it should hold that $p < 15$. We explain how to handle the case of $p \geq 15$ later in this section.

After scanning the genome sequence, $\text{count}$ array is then scanned from the beginning, and all the windows $W$ with $\text{count}[W] = 1$ are extracted in the order of appearance. Since the extracted windows are fully sorted, the list of suffixes $S$ corresponding to extracted windows $W$ composes a suffix array without any further operations. A suffix array can be converted into a suffix tree very quickly in $O(n)$ time [2].

Our algorithm appends the suffixes $S$ in the suffix array $\mathcal{A}$ into a suffix tree $T$ one by one in lexicographical order. Here, for any $S$, it holds that $\text{Lcp}(S) < p$, where $\text{Lcp}(S)$ is the length of longest common prefix between $S$ and its right previous suffix $S'$ in $\mathcal{A}$. For example, for two adjacent suffixes $S' = \text{AGCG}$ and $S = \text{AGTG}$ in $\mathcal{A}$, the longest common prefix is ‘AG’ and $\text{Lcp}(S) = 2$. Our algorithm manages an array right of size $p$ for efficient conversion into the suffix tree $T$. An element right[$d$] ($0 \leq d < p$) is a node $N$ in $T$, if $N$ is located on the path from the root to a leaf node corresponding to the lexicographically largest (most lastly appended) suffix and $N$ has string depth $d$. If there is no such node, right[$d$] = NULL. Figure 3 shows a suffix tree and its associated right array; right[5] points at a leaf node $L_2$.

At first, the suffix tree $T$ consists of only a root node; right[0] points at the root and right[d] ($d \geq 0$) = NULL. $\text{Lcp}(S)$ is computed for each suffix $S$ in $\mathcal{A}$ to append $S$ in $T$. Let windows $W$ and $W'$ correspond to the suffix $S$ and its right previous suffix $S'$ in $\mathcal{A}$, respectively. As explained above, $W$ and $W'$ are represented as unsigned int values. Let $b$ ($\geq 0$) be the position of rightmost non-zero bit in $B = W \text{XOR} W'$, then $\text{Lcp}(S) = p - \lceil(b + 1)/2\rceil$. For the first suffix $S$ in $\mathcal{A}$, $\text{Lcp}(S) = 0$.

For a suffix $S$ to be appended, if right[$\text{Lcp}(S)$] is a node $N$ and $N$ has string depth $d$, the path label of $N$ matches the prefix of length $d$ of $S$. Hence, a new outbound edge $e$ is added to $N$ and connected to a new leaf node corresponding to $S$. The label of $e$ is the suffix of length $(\text{Len}(S) - d)$ of $S$. After appending $S$ in $T$, our algorithm sets right[$d$] ($\text{Lcp}(S) < d < p$) = NULL. Figure 3 (a) shows an example of appending a new suffix $S = \text{AGCG}$. Since $S' = \text{AGCG}$, $\text{Lcp}(S) = 2$, and right[2] = $N_2$, a new edge
is added to \( N_2 \) and connected to a new leaf node \( L_3 \) corresponding to \( S \).

In case of \( \text{right}[\text{Lcp}(S)] = \text{NULL} \), a new node \( N' \) whose string depth is \( \text{Lcp}(S) \) should be added. Our algorithm finds largest \( d_1 \) (\(< \text{Lcp}(S)\)) and smallest \( d_2 \) (\(>\text{Lcp}(S)\)) such that \( \text{right}[d_1] = N_1 \) and \( \text{right}[d_2] = N_2 \) are not NULL. It splits the edge connecting two nodes \( N_1 \) and \( N_2 \), inserts a new node \( N'' \) between them, and sets \( \text{right}[\text{Lcp}(S)] = N' \). The operations afterwards are the same as those in the case of \( \text{right}[\text{Lcp}(S)] = N' \) explained above. Figure 3(b) shows an example, where \( S' = \text{ACTG}$ \) and \( S = \text{AGTG}$\). Since \( \text{Lcp}(S) = 1 \) and \( \text{right}[1] = \text{NULL} \), a new node \( N'_1 \) is added between two nodes \( N_1 \) and \( N_2 \). A new outbound edge is added to \( N'_1 \) and connected to a new leaf node \( L_3 \) corresponding to \( S \).

In the second phase, our algorithm constructs the suffix tree for the suffixes containing multiple occurrence windows using FAST [7]. FAST divides the suffixes in the human genome sequence into partitions so that a partition contains the suffixes of the same prefix of length \( p \) and then constructs the suffix subtree independently for each partition. FAST scans the genome sequence to find all the positions of suffixes in each partition, then extracts suffixes at the positions one by one, and inserts them into a suffix subtree using naive algorithm [5].

All the windows \( W \) with \( \text{Occ}(W) > 1 \) should be found in the genome sequence, which is performed in the same way as the first phase. While scanning \( \text{count} \) array, all the windows \( W \) with \( \text{Occ}(W) > 1 \) are extracted. Then, for each window \( W \), the positions of the suffixes containing \( W \) are found by scanning the genome sequence. The positions are given to FAST to construct a suffix subtree \( S_W \), which is then merged to the suffix tree \( T \) constructed in the first phase.

A suffix subtree \( S_W \) is merged to \( T \) as follows. Since all the suffixes added in \( S_W \) have a common prefix \( W \), the root node in \( S_W \) has only one outbound edge \( e \) whose label is \( W \). Let \( R \) be the node connected from \( e \), i.e., \( e \) is the inbound edge of \( R \). Since \( T \) is constructed only for single occurrence windows, no suffix having the prefix \( W \) is found in \( T \). Hence, \( W \) is added in \( T \) without duplication using naive algorithm, and the new leaf node is replaced with the suffix subtree rooted by \( R \).

The case of \( p \geq 15 \) is handled as follows. As explained above, since the size of \( \text{count} \) array cannot exceed 2 GB in actual implementation, maximum value of \( p \) is 14. When \( p \) is greater than 14, the window \( W \) is split into two sub-windows as \( W = YW' \). The length of \( W' \) is set to 14, and the length of \( Y \) is \( \text{Len}(Y) = \text{Len}(W) - 14 \). For every possible sub-window \( Y \), the first and second phases explained above should be repeated to construct a suffix subtree \( S_Y \), and the number of repeat is \( 4^\text{Len}(Y) \). The suffix subtrees \( S_Y \) for different sub-windows \( Y \) have nothing in common, and hence they can be constructed independently.

Our algorithm is parallelized as follows. Since the suffix subtrees \( S_Y \) for different sub-windows \( Y \) can be constructed independently as explained above, they can be constructed in different threads in parallel. Our algorithm creates multiple threads on a computer with multi-core CPUs and constructs different suffix subtrees \( S_Y \) in different threads. In case of \( p \leq 14 \), the length of sub-window \( W' \) in \( W \) is set to be less than 14 to make \( \text{Len}(Y) \geq 1 \). For best performance, \( \text{Len}(Y) \) should be as small as \( \text{Len}(Y) = [\log_4 t] \), where \( t \) is the number of concurrent threads.

We merge suffix subtrees \( S_Y \) for different sub-windows \( Y \) in a suffix subtree \( S_W \) in almost the same way as merging suffix subtrees \( S_W \) in \( T \) explained above. First, a suffix subtree \( S_W \) is constructed by adding all possible sub-windows \( Y \) of length \( \text{Len}(Y) \). The height of \( S_W \) is \( \text{Len}(Y) \) and the number of terminal nodes is \( 4^\text{Len}(Y) \). Then, for each sub-window \( Y \), the corresponding terminal node in \( S_W \) is replaced with the root node of corresponding suffix subtree \( S_Y \).

### 4. Evaluation

In this section, we perform experiments to evaluate our algorithm and to compare it with the previous FAST [7] and B-FAST [2] algorithms. The hardware platform is a PC equipped with Intel Core i7-2600K 3.4 GHz CPU, 16 GB DDR3 main memory, and 256 GB solid-state drive (SSD), and the software platform is Microsoft Windows 7 64-bit Edition and Visual Studio 2010. We set the number of concurrent threads as 4.

In our experiment, we measured the elapsed time of our algorithm for a few window lengths \( p \). Since the entire human genome sequence has the size of 3 Gbp and \( [\log_4 3G] = 16 \), we set \( p \) close to 16. We also measured the elapsed time of FAST under the same settings. Figure 4 shows the result of experiments; it shows the elapsed time and performance improvement ratio of our algorithm. Our algorithm showed best performance with \( p = 17 \) and completed indexing in 1,049 seconds (17 minutes 29 seconds). Our algorithm outperformed FAST by up to 2.09 times. Since FAST was shown in [7] to outperform DiGeST [1] by up to 10.5 times, we can infer that our algorithm transitively outperformed DiGeST, too.

Table 1 shows the occurrence counts of single and multiple occurrence windows. The values in parentheses indicate the percentage of corresponding windows among all
possible windows in the human genome sequence; as \( p \) increases, the ratio of single occurrence windows also increases. Since the performance of the first phase is much better than the second phase in our algorithm, it is advantageous to maximize the ratio of single occurrence windows. However, as explained in Sect. 3, as \( p \) increases, the number of repeat also increases, which causes performance degradation of our algorithm.

We implemented the most recent \( B^2ST \) [2] and compared its performance with our algorithm. \( B^2ST \) consists of three steps, namely input partitioning, suffix sorting, and merging steps. We implemented only the first two steps, since those take more than 90\% of the whole execution time [2]. For the suffix sorting step, we used an algorithm proposed by Bentley and Sedgewick [3], since the algorithm showed the fastest suffix sorting performance among many string sorting algorithms when the average LCP was small as in genome sequences [6]. In our experiment, we divided the whole human genome sequence into \( k = 3 \) partitions as in [2], and thus we performed the suffix sorting for each of \( k(k - 1)/2 = 3 \) partition pairs. As a result, \( B^2ST \) completed the indexing in 3189 seconds (53 minutes 9 seconds). Therefore, our algorithm outperformed \( B^2ST \) by 3.04 times, and the improvement ratio should be higher when compared with the full implementation of \( B^2ST \).

Ghoting and Makarychev [4] claimed that their algorithm completed indexing the entire human genome sequence under 15 minutes. However, the experiment was performed on a supercomputer called IBM Blue Gene/L equipped with 1024 processors and 512MB memory for each processor (512 GB as a whole). Hence, their algorithm itself is not for direct comparison with ours, yet the performance of ours is comparable to theirs, since ours completed indexing the entire human genome sequence in 1,049 seconds (17 minutes 29 seconds) on a machine with much lower specification.

**References**

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

[2] M. Barsky, U. Stege, and A. Thomo, “Suffix trees for inputs larger than main memory,” Inf. Syst., vol.36, no.3, pp.644–654, May 2011.

[3] J.L. Bentley and R. Sedgewick, “Fast algorithms for sorting and searching strings,” Proc. ACM-SIAM Symp. on Discrete Algorithms, pp.360–369, New Orleans, Louisiana, Jan. 1997.

[4] A. Ghoting and K. Makarychev, “I/O efficient algorithms for serial and parallel suffix tree construction,” ACM Trans. Database Systems (TODS), vol.35, no.4, pp.25:1–25:37, Nov. 2010.

[5] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, 1997.

[6] N.J. Larsson and K. Sadakane, “Faster suffix sorting,” Technical Report, LUCS-TR:99-214, Dept. of Computer Science, Lund University, Sweden, 1999.

[7] W.-K. Loh, Y.-S. Moon, and W. Lee, “A fast divide-and-conquer algorithm for indexing human genome sequences,” IEICE Trans. Inf. & Syst., vol.E94-D, no.7, pp.1369–1377, July 2011.

[8] B. Phoophakdee and M.J. Zaki, “TRELLIS+: An effective approach for indexing genome-scale sequences using suffix trees,” Proc. Pacific Symp. on Biocomputing, Kohala Coast, Hawaii, pp.90–101, Jan. 2008.

[9] 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.