Indexing Genomic Sequences on the IBM Blue Gene

Amol Ghoting
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598, USA
aghoting@us.ibm.com

Konstantin Makarychev
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598, USA
konstantin@us.ibm.com

ABSTRACT

With advances in sequencing technology and through aggressive sequencing efforts, DNA sequence data sets have been growing at a rapid pace. To gain from these advances, it is important to provide life science researchers with the ability to process and query large sequence data sets. For the past three decades, the suffix tree has served as a fundamental data structure in processing sequential data sets. However, tree construction times on large data sets have been excessive. While parallel suffix tree construction is an obvious solution to reduce execution times, poor locality of reference has limited parallel performance. In this paper, we show that through careful parallel algorithm design, this limitation can be removed, allowing tree construction to scale to massively parallel systems like the IBM Blue Gene. We demonstrate that the entire Human genome can be indexed on 1024 processors in under 15 minutes.

1. INTRODUCTION

Over the past few years, with advances in sequencing technology and through aggressive sequencing efforts, DNA sequence databases have reached gigantic proportions. The GenBank sequence database recently surpassed the 100Gbp[1] mark [20] and conservative estimates suggest that its size is doubling every six months. Furthermore, individual genomes in this database can also be very large – for example, the Human and Triticum Aestivum genomes span 3Gbp and 16Gbp, respectively. To further from these advances, it is imperative that life science researchers have the ability to efficiently process and query such large sequence data sets.

For the past three decades, the suffix tree has served as a fundamental data structure in string processing. As pointed out by many, it exposes the internal structure of a string in a way that facilitates the efficient implementation of a myriad of string operations. Examples of these operations include string matching (both exact and approximate), exact set matching, all-pairs suffix-prefix matching, finding repetitive structures, and finding the longest common sub-string across multiple strings [11]. Many recent research efforts in the bioinformatics domain [3, 8, 9, 11, 16, 17, 19] have advocated the use suffix trees in evaluating queries on biological sequence data sets and the past few years have seen researchers devise several disk-based suffix tree construction algorithms [2, 4, 7, 14, 15, 21, 22, 24, 25] that maintain very large suffix trees on disk. However, tree construction times continue to be daunting – for example, indexing the Human genome requires in excess of 30 hours on a single processor system with 2 gigabytes of physical memory [25].

Further complicating the issue is the fact that genome indexing is not a one-time problem. For example, consider the area of comparative genomics [23] where one is interested in comparing different genomes, be they from the same or different species. Here researchers may be interested in comparing the genomes of individuals that are prone to a specific type of cancer to those that are not susceptible. In this case, we need to efficiently build a suffix tree for each genome/group of genomes on an on-demand basis.

In this article, we consider the problem of building a suffix tree using a massively parallel system with the intent of reducing index construction time. Specifically, we consider the problem of building a suffix tree for genomic sequences as large as the Human genome (3Gbp) using the IBM Blue Gene/L[2] system. This is a challenging problem for the following reasons:

- Disk-based suffix tree construction algorithms exhibit poor I/O efficiency when accessing the partially constructed suffix tree during construction.
- Disk-based suffix tree construction algorithms are limited in that to garner reasonable I/O efficiency the input string being indexed must fit in main memory.

These two limitations of suffix tree construction algorithms have been well documented in the literature [10, 21, 25]. If one wishes to realize a scalable parallel suffix tree construction, these limitations mean two things:

- Each processor should be capable of housing the entire input string in its local memory.
- Due to poor I/O efficiency, each processor should be provided with I/O bandwidth that is comparable to that of a serial system.

[1] One bp (base pair) is one character in the sequence

[2] `http://www-03.ibm.com/systems/deepcomputing/bluegene`

SC09 November 14-20, 2009, Portland, Oregon, USA.

The first condition is hard to realize as most state-of-the-art massively parallel systems have a small amount of memory (for example, 512 megabytes) per processing element[3]. In the future, the available main memory per processing element will only reduce, further aggravating the situation. Furthermore, state-of-the-art parallel systems do not provide I/O bandwidth that is comparable to that of a serial system, making it hard to satisfy the second condition.

To address the aforementioned challenges, we show that through architecture-conscious algorithm re-design, one of the simplest suffix tree construction algorithms can be *re-architected* to efficiently build very large suffix trees on a parallel system like the IBM Blue Gene/L, even when the input string is significantly larger than the size of main memory on a single processor. Specifically, we make the following contributions:

- We present a parallel suffix tree construction algorithm to build disk-based trees. The algorithm supports a tunable memory footprint, employs in-network caching (for the string and tree), and leverages effective collective communication to realize an efficient parallelization.
- We experimentally evaluate the proposed approach on the IBM Blue Gene/L system and show that the proposed algorithm is scalable and can index the entire human genome (3Gbp) in *under 15 minutes on 1024 processors.*

The remainder of this article is organized as follows. Section 2 presents a background on suffix trees. Related efforts and their drawbacks are discussed in Section 3. We present our parallel suffix tree construction algorithm in Section 4. An experimental evaluation is presented in Section 5 followed by our conclusions in Section 6.

2. BACKGROUND

Let A denote a set of characters. Let $S = s_0, s_1, \ldots, s_{n-1}, \$$, where $s_i \in A$ and $\$ \notin A$, denote a \$ terminated input string of length $n+1$. The i^{th} suffix of S, denoted by S_i, is the substring $s_i, s_{i+1}, \ldots, s_{n-1}, \$$. The suffix tree for S, denoted as T, stores all the suffixes of S in a tree structure. This tree has the following properties:

- Paths from the root node to the leaf nodes have a one-to-one relationship with the suffixes of S. The terminal character \$ is unique and ensures that no suffix is a proper prefix of any other suffix. Therefore, there are as many leaf nodes as there are suffixes.
- Edges spell non-empty strings.
- All internal nodes, except the root node, have at least 2 children. The edge for each child node begins with a character that is different from the starting character of its sibling nodes.
- For an internal node v, let $l(v)$ denote the substring obtained by traversing the path from the root node to v. For every internal node v, with $l(v) = x\alpha$, where $x \in A$ and $\alpha \in A^*$, we have a pointer known as a suffix link to an internal node u such that $l(u) = \alpha$.

[3] `http://www.top500.org`

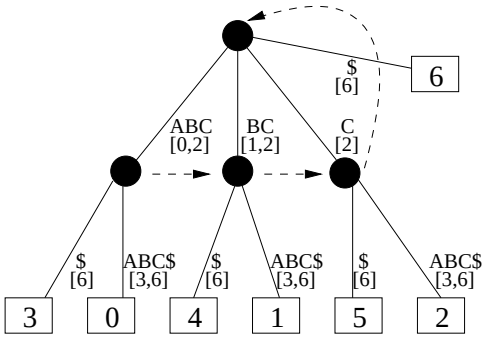

Figure 1: Suffix tree for $S = ABCABC\$$ [0123456]. Internal nodes are represented using circles and leaf nodes are represented using rectangles. Each leaf node is labeled with the index of the suffix it represents. The dashed arrows represent the suffix links. Each edge is labeled with the substring it represents and its corresponding edge encoding.

An instance of a suffix tree for a string $S = ABCABC\$$ is presented in Figure 1. Each edge in a suffix tree is represented using the start and end index of the corresponding substring in S. Therefore, even though a suffix tree represents n suffixes (each with at most n characters) for a total of $\Omega(n^2)$ characters, it only requires $O(n)$ space [26].

3. RELATED WORK

Algorithms due to Weiner [27], McCreight [18], and Ukkonen [26] have shown that suffix trees can be built in linear space and time. These algorithms afford a linear time construction by employing suffix links. Researchers have also studied the parallelization of suffix tree construction algorithms [1, 13] on various parallel abstract machines. Although these algorithms provide theoretically optimal performance, their memory accesses exhibit poor locality of reference. As a consequence, these algorithms are grossly inefficient when either the tree or the string does not fit in main memory.

To tackle this problem, this past decade has seen several research efforts that target large disk-based suffix tree construction. Most disk-based suffix tree construction algorithms partition the suffix tree into smaller sub-trees (suffix sub-trees) and build these sub-trees in main memory. They then merge these sub-trees to realize the final suffix tree. In theory, many these algorithms can be parallelized by having each processor build a sub-tree of the entire suffix tree. However, these algorithms are also I/O inefficient and hence exhibit poor parallel scalability [10]. As a result, very little work has been done in the area of parallel disk-based suffix tree construction. We list these efforts next for the sake of completeness.

3.1 Disk-based Suffix Tree Construction

Hunt *et al.* [14] presented the very first approach to efficiently build suffix trees that do not fit in main memory. The approach drops the use of suffix links in favor of better locality of reference. The method first finds a set of fixed length prefixes such that the corresponding sub-trees of the suffix tree (or suffix sub-trees) will fit in main memory. Next, for each of these prefixes, the approach builds

the associated suffix sub-tree using a scan of the data set. This involves inserting each suffix S_i with the prefix into the tree, starting at the root node. Suffix insertion finds a path in the tree that shares the longest common prefix with S_i and branches from this path when no more matching characters are found. In other words, this method matches S_i with a suffix $S_j : 0 \leq j < i$ that shares the longest common prefix with S_i. As each suffix ends with the unique terminal character (\$), no suffix can be a proper prefix of any other suffix. Hence, every suffix insertion will result in the splitting of an edge and lead to the creation of a leaf node in the partially constructed suffix sub-tree. Although, by design, the suffix sub-tree will fit in main memory, this algorithm exhibits poor reference locality as the input string need not fit in main memory. In fact, every suffix insertion can result in $\Omega(\log(n/B))$ (where B is the size of main memory available to the string) near-random seeks to the input string [10]. The worst case complexity of the approach is $O(n^2)$, but it exhibits $O(n \log n)$ average case complexity. All disk-based suffix tree construction algorithms (that will be presented below) also exhibit this same complexity.

Cheung *et al.* proposed DYNACLUSTER [7], an algorithm that employs dynamic clustering to identify tree nodes that are frequently accessed together. The suffix tree is then built a cluster at a time to reduce random tree accesses during tree construction. While the tree construction phase drops the use of suffix links, the authors demonstrate that one can recover suffix links using a post-construction step.

Tian *et al.* proposed ST-MERGE [25], an algorithm that partitions the input string and constructs a suffix tree for each of these partitions in main memory. These suffix trees are then merged to create the final suffix tree. This approach drops the use of suffix links entirely. The merge phase in this algorithm is known to exhibit poor I/O efficiency when either the suffix-tree or the input string does not fit in main memory [25]. Phoophakdee and Zaki proposed TRELLIS [21], that is similar in flavor to ST-MERGE but differs in the following regards. First, the approach finds a set of variable length prefixes such that the corresponding suffix sub-trees will fit in main memory. Second, it partitions the input string and constructs a suffix tree for each partition in main memory (like ST-MERGE) and stores the sub-trees for each prefix determined in the first step, separately, on disk. Finally, it merges all the sub-trees associated with each prefix to realize the final set of suffix sub-trees. TRELLIS also performs post-construction link recovery. By design, TRELLIS ensures that each of the suffix sub-trees (built using the merge operation) will fit in main memory during the merge phase. However, like ST-MERGE, it exhibits poor reference locality when accessing the input string [21]. Furthermore, the link recovery phase is I/O inefficient as it needs to simultaneously access several suffix sub-trees.

4. PARALLEL CONSTRUCTION

Modern parallel supercomputers are often disk-less, have limited main memory per processing element (for example, 512 megabytes) and do not offer support for virtual memory. Such systems provide file I/O over the network with reasonable I/O bandwidth through parallel I/O, but suffer from high file I/O latency. Existing suffix tree construction algorithms cannot be trivially parallelized on such systems for the following reasons.

- Due to limited main memory per processor, the input string being indexed cannot always be maintained in-core, and needs to be maintained and read off the network file system. Accessing the suffix tree during the tree construction and link recovery processes requires accessing the input string (using start and end indices). These accesses are near random and hence said processes are extremely I/O inefficient when the input string does not fit in main memory [10]. Parallel executions become latency bound.
- The link recovery task requires all processors to simultaneously have both read and write access to the suffix sub-trees. On massively parallel systems, this quickly leads to I/O contention and limits scalability.
- Naive parallelization results in significant amount of redundant work being performed, which also limits scalability.

While modern parallel systems do not offer high (out-of-network) disk I/O bandwidth (per processing element), they do offer low in-network communication latency, and high in-network communication bandwidth. Coupled with the fact that such systems have a significant amount of aggregate main memory, disk I/O-intensive algorithms can continue to deliver high parallel performance as long as the processing elements can effectively utilize their collective main memories for data storage. Furthermore, effective collective communication can also aid in the management of their aggregate main memories and minimize redundant work.

Our parallel suffix tree construction algorithm is specifically designed to index out-of-core input strings and maintain a constant working set size and a fixed memory footprint at all times. This is accomplished by tiling accesses to the input string and the partially constructed suffix tree during the construction and recovery processes. The algorithm only needs to access a fixed portion of the input string at any point during its execution. By caching this input string in the collective main memory of a parallel system, the approach can index large strings while constraining most data accesses to within the network. Moreover, by ensuring that the input string is accessed in a blocked fashion, once a block of the string is fetched from a remote processor, all processing can continue on the local copy of the block of the input string. Furthermore, the approach eliminates I/O contention problems faced by the link recovery tasks by restructuring computation to maximally re-use the suffix sub-trees once they are read into the network. Fast collective communication is leveraged to eliminate redundant work and manage All-to-All in-network data movement.

The algorithm takes the memory budget (per processor) (M), input string (S), and number of processors (C) as input. The overall control flow for our algorithm is presented in Algorithm 1 and is executed by each processor in the system – collective procedures are noted in the pseudo code. The algorithm has the following main steps:

1. In-Network String Caching: This step uses the collective main memories of all the processors on the system to build a cache (with redundant copies) for the input string. This allows one to handle all string accesses within the network.
2. Task Generation: This step finds a set of prefixes P such that the sub-tree of the suffix tree associated with

each prefix $p \in P$ can be built within the memory budget M. Furthermore, this step ensures that the size of this set P is greater than the number of processors C.

3. Prefix Location Discovery: This step finds the location of a prefix $p \in P$ in the input string.

4. Sub-tree Construction: This step builds the sub-tree (T_p) of the suffix tree for each prefix $p \in P$, within the memory budget M.

5. Suffix Link Recovery: This optional stage recovers the complete set of suffix links, should they be needed.

These steps are detailed next.

Input: Input string S, Memory Budget M, Number of Processors C
Output: Suffix Tree T

```
BuildInNetworkStringCache(S, M, C) (Collective);
P = GenerateTasks(S, M, C) (Collective);
while P not empty do
    p = GetNextAvailableTask(P) (Collective);
    L = LocatePrefix(p, S) (Collective);
    T_p = BuildSubTree(p, L, S) ;
end
T = ⋃_{p∈P} T_p
RecoverSuffixLinks(T) (Collective);
```

Algorithm 1: Control Flow

4.1 In-Network String Caching

First, we build an in-network cache for the input string. This is accomplished by having the processors reserve a fixed portion of their main memories for a string cache. The processors then collectively read the input string into their individual memories. We use MPI's collective file I/O primitives to perform these operations. Collective I/O ensures that the same copy of the string is not read multiple times off disk. Once a piece of the string in read into the network, it is efficiently distributed across the network without repeated I/O. The string is replicated as many times as possible in a round robin fashion. *All string accesses in the implementation are forwarded to the closest copy of the string in the network*. Modern architectures support one-sided communication, where a processor can access the content of a remote processor's memory without interrupting the remote processor and several toolkits (such as global arrays[4]) make it possible to implement such caching infrastructures efficiently. Since our algorithm accesses the input string in a tiled fashion, our proposed approach to caching the input string allows us to leverage the high point-to-point network bandwidth on such systems. Note that supporting random string I/O is hard as in-network latency is still significant.

4.2 Task Generation

Typically, the suffix tree is an order of magnitude larger than the string being indexed. As a result, for large input strings, the suffix tree cannot even be accommodated in virtual memory, let alone main memory. The goal of this step is to find a set of prefixes so as to partition the suffix tree into sub-trees (each prefix corresponds to a sub-tree) that can be built in main memory. This approach to partitioning a suffix tree into manageable pieces has been proposed previously [14, 21] for the case of serial tree construction. We extend it to make use of the fast collective communication capabilities on modern parallel systems.

Let $f(p)$ denote the number of times prefix p occurs in S. Let MTS (**M**aximum **T**ree **S**ize) denote the maximum amount of memory space in bytes that can be allotted to the sub-tree of the suffix tree during tree construction (Note: we will explain how MTS is determined at a later stage). Let NS denote the size of a suffix tree node in bytes. The goal of this step is to find a set of prefixes P such that:

1. $\forall p \in P : 2 \times f(p) < \frac{MTS}{NS}$
2. $T = \bigcup_{p \in P} T_p$
3. $|P| \geq C$

In other words, we want to find a set of prefixes P such that each $p \in P$ occurs no more than $\frac{MTS}{NS \times 2}$ times in S (Condition 1). This guarantees that the sub-tree associated with each p will not occupy more than MTS bytes of space. Furthermore, conditions 2 and 3 ensure that the union of these sub-trees will cover the entire suffix tree and that we have sufficient tasks to keep all the processors busy, respectively.

There are various ways to find the set P. One approach is to compose P using fixed-length prefixes of each suffix. This approach works well provided the data set is not skewed. However, many real string data sets are skewed (the Human genome, for example) [21]. As a result, using a fixed prefix length can result in several partitions that are smaller than necessary, resulting in poor memory usage – ideally you want each sub-tree to have a size as close to MTS as possible. Observing that once the sub-tree associated with a prefix fits in main memory, it need not be extended, a second approach is to compose P using variable-length prefixes. Using variable-length prefixes allows one to gracefully handle skewed data by allowing for the construction of sub-trees that are roughly of the same size.

We employ variable-length prefixes due to the aforementioned advantages. We use a simple multiple scan approach to find the set of variable-length prefixes P. Each processor is responsible for processing a partition of the input string. During each scan of the input string, each processor iteratively reads the input string at B byte intervals, in blocks of size $B + sc - 1$ (sc is the scan number starting at 1) in its partition (Note: we will explain how B is determined at a later stage), considering prefixes of length sc during each scan (to limit memory consumption). At the end of the scan, we collectively aggregate the counts for the various prefixes of length sc discovered during the scan using a parallel merge in $\log C$ time. The master node then adds those prefixes that occur fewer than $\frac{MTS}{NS \times 2}$ to the task queue P – each such prefix corresponds to a sub-tree of the suffix tree and can be built independently, and hence constitutes a task. Furthermore, during each scan, if a prefix of size sc has a proper prefix in the task queue determined up to the previous iteration, we ignore it as we no longer need to extend it. For this purpose, before each scan, the master node broadcasts the task queue to all the slave nodes. This process continues until all potential prefixes are covered in the task queue. It is easy to see that this procedure will give us the desired set of variable-length prefixes. At the end of

[4] `http://www.emsl.pnl.gov/docs/global`

this process if $|P| < C$, we reduce MTS as per a geometric schedule and repeat the process. We found that reducing MTS by half works very well in practice.

4.3 Prefix Location Discovery

Tasks discovered in the previous step are distributed across the processors in a round robin fashion. Before suffix sub-tree construction proceeds, one needs to get the list of locations for each prefix being processed. If each processor were to scan the entire string to discover the location for its prefix p, we would have a significant wastage of computation and limited scale-up as most string accesses to find a matching prefix p would be wasteful. To improve performance, this step is performed collectively as there is significant overlap of computation across processors. This step proceeds as follows. First, the processors collectively exchange the subset of P that is to be processed in that iteration – let us call this set of prefixes Q. Second, each processor finds the locations for all prefixes in Q in a partition of the input string. The processors read the input string in blocks of size $B +$ *MaxLengthOfPrefix*, where *MaxLengthOfPrefix* is the length of the longest prefix in Q. Finally, the processors perform an All-To-All collective exchange using the MPI_Alltoallv primitive, at the end of which, each processor has a list of locations for the prefix it is processing in that iteration.

Input: Input string S, Prefix p, Prefix Locations *pLocs*
Output: Suffix sub-tree T_p

```
1 T_p = NULL;
  /* Insert suffix S_i into T_p, reusing the path
  for suffix S_j : 0 ≤ j < i that shares the longest
  common prefix with S_i                          */ ;
2 for each i ∈ pLocs do
3 |  addSuffix(S_i, T_p);
4 end
```

Algorithm 2: Single-loop Suffix Sub-tree Construction

4.4 Suffix Sub-tree Construction

The goal of this step is to build a suffix sub-tree for the prefix p that is assigned to the processor during that iteration. This is the most time consuming step relative to the steps we have already presented. A simple way to build each sub-tree would be to use the "single-loop" approach proposed by Hunt *et al.* [14]. The pseudo code for this approach is presented in Algorithm 2. This approach inserts each suffix S_i with the prefix p into the tree, starting at the root node. The *addSuffix* method finds a path in the tree that shares the longest common prefix with S_i and branches from this path when no more matching characters are found. In other words, this method matches S_i with a suffix $S_j : 0 \leq j < i$ that shares the longest common prefix with S_i. As each suffix ends with the unique terminal character (\$), no suffix can be a proper prefix of any other suffix. Hence, every suffix insertion will result in the splitting of an edge and the creation of a leaf node in the partially constructed suffix sub-tree.

This algorithm works very well in practice provided both the input string and the suffix sub-tree fit in main memory. However, when the input string does not fit in main memory, random string accesses significantly degrade the performance of this algorithm. This degradation can be explained as follows. When inserting S_i, the *addSuffix* method

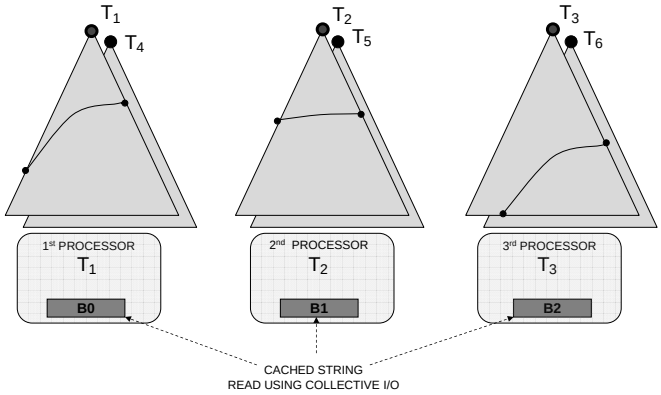

Figure 2: Parallel Suffix Sub-tree Construction

accesses the suffix $S_j : 0 \leq j < i$ that shares the longest common prefix with S_i. While the portion of the input string being referenced by S_i is contiguous, that being referenced by S_j need not be contiguous. In fact, for a random input string from a symmetric Bernoulli distribution, S_j will most likely be distributed across $\Omega(\log(n/B))$ different locations [10]. This results in $\Omega(n \log(n/B))$ random string accesses when constructing the suffix sub-tree. Independent of whether the string is cached in the network or read off the file system, the execution becomes latency bound as most of the execution time is spent on retrieving a substring from a remote processor's memory or the file system.

We will now show that through careful design, the "single-loop" algorithm can be restructured to build a suffix tree within a fixed memory budget with excellent processor utilization, for strings of any length. There are three entities that need to be accessed during suffix sub-tree construction: 1) the partially constructed suffix sub-tree, 2) the set of substrings being referenced by the suffix sub-tree, and 3) the set of suffixes being inserted into the tree. By construction, the task generation step ensures that the suffix sub-tree will fit in main memory. We will now show how computation can be restructured so that both: the set of strings being referenced by the partially constructed suffix sub-tree, and the set of suffixes being inserted into the tree can be maintained in a fixed memory budget.

4.4.1 Tiling Suffix Sub-tree Edge References

Suffix (sub-) trees have the following property: The start index of each edge in the suffix tree is greater than the start index of its parent edge. Thus, for a fully constructed suffix sub-tree, it is possible to partition its edges into disjoint partitions $E_0, E_1, \ldots, E_k$ based on their start indices such that the parent edge for each edge in partition E_i is always located in a partition $E_j : j \leq i$. For an input string from a symmetric Bernoulli distribution, if the string were partitioned into blocks of size B, and the edges were partitioned into n/B partitions based on the block in which their start indices lie, then the tree edges would be partitioned into n/B partitions, where each partition has $O(B)$ edges. This property serves as the basis for our suffix sub-tree construction algorithm.

The pseudo-code for a tiled suffix sub-tree constructor is presented in Algorithm 3. Figures 2 and 3 depict how the tree is constructed. For the time being, we will ignore all *InsertBlock* and *EOB* references in the figure and the algo-

Input: Input string S, Prefix p, Block Size B, Prefix Locations $pLocs$
Output: Suffix sub-tree T_p

```
1  T_p = {} ;
2  AE[ ] = {} /* Active Edge array                          */;
3  TI[ ] = {} /* Tree Index array                          */;
4  SI[ ] = pLocs /* String Index array                     */;
5  EOB = {} /* EOB suffix list                            */ ;
6  Front = {0, ..., sizeof(pLocs) − 1} ;
   /* List of active indices in AE, TI, and SI           */ ;
7  for (i = 0, i < n/B, i++) do
       /* Read i^th Tree Block from string cache         */ ;
8      TreeBlock= S[i × B, i × B + B − 1] ;
       /* Start with the first element in the front
       */ ;
9      cnt = firstElement(Front) ;
10     for (j = i, j < n/B, j++) do
           /* Skip unneeded Insert Blocks                */ ;
11         if SI[cnt] > j × B + B − 1 and EOB = {} then
           continue ;
           /* Read Insert Block from string cache        */ ;
12         InsertBlock= S[j × B, j × B + B − 1] ;
           /* Process end-of-block suffixes              */ ;
13         for each k ∈ EOB do
14             addSuffix(SI[k], AE[k], TI[k]) ;
15             if Incomplete insertion then
16                 if end of InsertBlock not reached during
                   insertion then remove(EOB, k);
17             end
18             else
                   /* Suffix has been completely
                   inserted, hence remove from front
                   and EOB                                */ ;
19                 remove(EOB, k) ;
20                 remove(Front, k) ;
21             end
22         end
           /* Process suffixes that have a String
           Index in InsertBlock                           */ ;
23         while j × B ≤ SI[cnt] ≤ j × B + B − 1 do
24             addSuffix(SI[cnt], AE[cnt], TI[cnt]) ;
25             if Incomplete insertion then
26                 if end of InsertBlock reached during
                   insertion then add(EOB, cnt)
27             end
28             else
                   /* Suffix has been completely
                   inserted, hence remove from front
                   */ ;
29                 remove(Front, cnt) ;
30             end
               /* Pick next element in front             */ ;
31             cnt = nextElement(Front) ;
32         end
33     end
34 end
```

Algorithm 3: Tiled Suffix Sub-tree Construction

rithm. Let us assume that the input string is broken into blocks of size B, giving us a total of n/B blocks (let us call these *TreeBlocks*). The suffix sub-tree is built in n/B steps. During the i^{th} step, each suffix starting with the prefix p is inserted into the suffix tree such that the input string references (from the edges in the partially constructed suffix sub-tree) lie in the i^{th} *TreeBlock*. At the end of the i^{th} step, all leaf nodes with parent edges that have a start index in the i^{th} *TreeBlock* will have been created – the corresponding suffixes have been completely inserted into the suffix sub-tree and no more work needs to be done for these suffixes. Furthermore, all internal edge (those edges that are not connected to a leaf node) accesses that lie in the i^{th} *TreeBlock* will be complete. In summary, at the end of the i^{th} step, each suffix with prefix p is either completely inserted or inserted to a point where all input string references (due to tree edges) up to the i^{th} *TreeBlock* are complete. For suffixes that fit the latter condition, more work needs to be done in the following steps. For these suffixes, we save a *front* of the computation performed up to the i^{th} step and resume from this *front* in the $(i+1)^{th}$ step. For each suffix being inserted that needs further work, we need to maintain state information that tells us the point up to which a suffix was inserted into the suffix sub-tree. Hence, for each suffix, we maintain the Active Edge (AE) that tells us which tree edge was processed last, the Tree Index (TI) that gives us the index (between start and end index) up to which this edge was processed, and the String Index (SI) that gives us the number of characters of this suffix that have been completely inserted. Note that the signature for the *addSuffix* method has been updated to reflect the stateful insertion of suffixes into the suffix sub-tree. This *front* is carried from one step to the next until it is empty – this is guaranteed to happen when the last *TreeBlock* is processed.

4.4.2 Tiling Suffix Accesses

The above mentioned approach assumes that the suffixes being inserted into the suffix sub-tree are always available in main memory. Obviously one cannot make this assumption when processing large strings. We remove this restriction by restructuring computation such that we tile accesses to the suffixes as they are inserted into the tree. Again, let us assume that the input string is broken into blocks of size B (let us call these *InsertBlocks*). The input string can be processed one *InsertBlock* at a time, as is explained in Algorithm 3 and depicted in Figure 3. For every i^{th} *TreeBlock*, only the $j \geq i$ *InsertBlocks* are needed. Introducing *InsertBlocks* raises the issue that it is now possible for a suffix insertion to be incomplete because the suffix crosses an *InsertBlock* boundary. These suffixes are saved as end-of-block (EOB) suffixes and are processed in the following iteration. After all *InsertBlocks* are processed for a certain *TreeBlock*, EOB is guaranteed to be empty. While the *front* has $O(n)$ entries, for a random input string (from a symmetric Bernoulli distribution), only $O(B)$ of these entries are processed with each *InsertBlock*[5]. This property allows one to maintain a constant working set size during execution.

4.5 Suffix Link Recovery

Many important string processing applications require suffix links [5, 6, 12]. For such applications, we invoke the op-

[5]This is possible as the contents of the String Index array are always in a sorted order.

Input: Input string S, Set of suffix sub-trees T, Block Size B
Output: Set of suffix sub-trees T with suffix links
/* Phase 1 */ ;
$L = \{\}$;
for *each* $t \in T$ **do**
  for *each block* b *of size* B *in* S **do**
    Traverse t such that accesses lie in b and find all internal nodes v that have a suffix link to a root node of a sub-tree in T;
    For each internal node v that satisfies the above condition, insert address of v and size of the sub-tree below v into L ;
  end
end
Exchange tasks with other processors using an *AlltoAll* collective exchange ;
/* Phase 2 */ ;
for *each* $t \in T$ **do**
  Find set of link recovery tasks L_t that point to t ;
  for $l \subset L_t$ *such that* $l + t$ *fit in memory* **do**
    Collectively read l into the processor's memory ;
    for *each block* b *of size* B *in* S **do**
      Traverse each task in l such that accesses lie in b and set suffix links to the correct position in sub-tree t;
    end
    $L_t = L_t - l$;
    Collectively write l to the file system ;
  end
end

Algorithm 4: Parallel Link Recovery

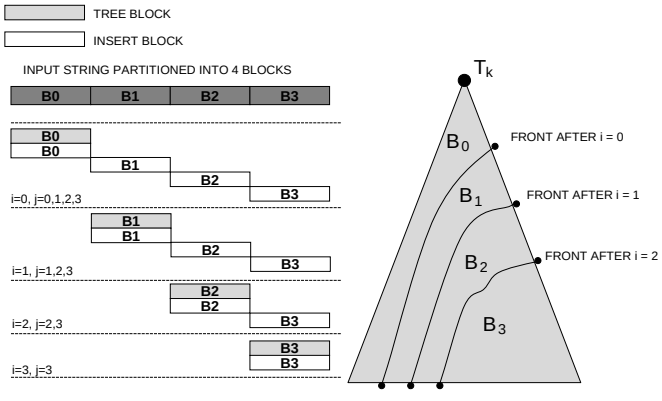

Figure 3: Tiled Suffix Sub-tree Construction

tional step of suffix link recovery. Like others [7, 21], our approach recovers suffix links after the suffix tree construction process. Parallel suffix link recovery is complicated by the fact that it is extremely I/O intensive – the processors simultaneously need both read and write access to multiple suffix sub-trees. We improve parallel suffix link recovery by improving the I/O efficiency of the process (through improved temporal locality) and minimizing I/O contention during execution (leveraging effective collective communication).

Suffix link recovery is performed in two phases and is depicted in Figures 4 and 5. The pseudo code for the two phases of the algorithm is presented in Algorithm 4. In the first phase, the sub-trees are distributed across the processors in a round robin fashion. Each processor then traverses each suffix sub-tree T_p that is assigned to it and finds all internal nodes (for example, the root node of T_k^a in Figure 4) that point to the root nodes of other suffix sub-trees (for example, the root node of T_k^a in Figure 4). We know that the suffix links of all the child nodes of such an internal node will always point to the child nodes of the node that is the pointed to by the suffix link of the said internal node (for example, suffix links for all nodes in T_k^a point to nodes in T_a). Note that finding these internal nodes does not require one to traverse the entire tree – the procedure only needs to access the sub-tree up to a depth where it can find nodes with suffix links that point to root nodes of other suffix sub-trees. We use a methodology similar to tiled suffix sub-tree construction in that we restructure accesses to the input string such that at all times a string of size B is maintained in memory. At the end of this phase, for each suffix sub-tree, we have the addresses of the sub-trees within it (for example, T_k^a) that can be processed independently with one other suffix sub-tree (for example, T_a) – these addresses constitute tasks that will be processed in the second phase. Once each processor has the list of tasks for all the suffix sub-trees assigned to it, we perform an All-To-All exchange, at the end of which, each processor has a complete list of tasks that point to one of the suffix sub-trees that is assigned to it.

In the second phase, each processor iteratively processes each suffix sub-tree. For each suffix sub-tree, first, we get a list of all tasks that point to it. Next, we load the suffix sub-tree into main memory (for example, T_a) and use the remainder of the memory budget to load as many tasks (for example, T_k^a) into main memory as possible, without exceeding the memory budget. Retrieving a task requires reading a sub-tree of the entire suffix sub-tree. If each processor were to do so independently on a massively parallel system, we will suffer from significant I/O contention as each processor may have to read a sub-set of every tree in the worse case. To do so efficiently, first, each processor reads all the suffix sub-trees assigned to it iteratively and then exchanges portions of this tree that are requested by other processors in a collective fashion – this can be done very efficiently within the network and improves temporal reuse. Essentially, to maximize I/O efficiency, once a tree is read into the network, we attempt to use it to the maximum extent possible before purging it from main memory. Finally, we propagate suffix links for all these tasks concurrently, tiling accesses to the input string. After this batch of tasks is processed, we update the sub-trees associated with these tasks on disk. In order to update the suffix links on disk, each processor needs to write the sub-trees associated with the processed

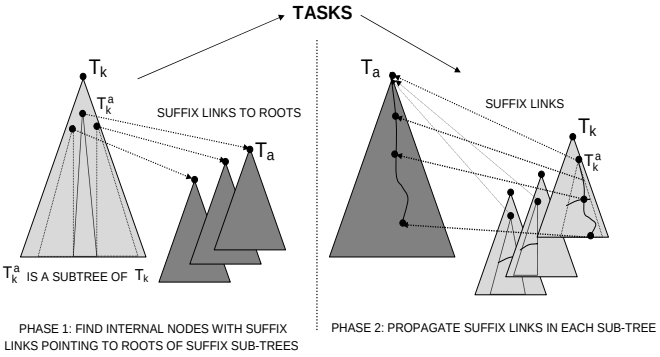

Figure 4: Tiled Suffix Link Recovery

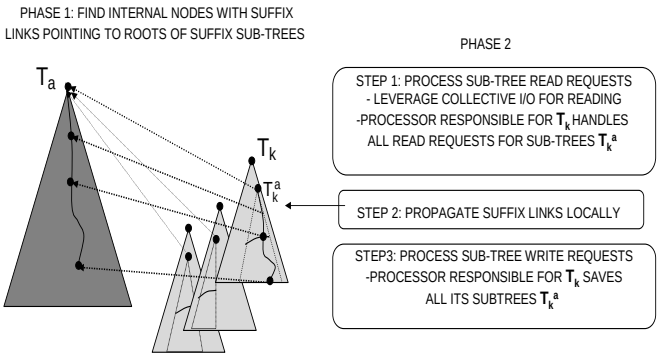

Figure 5: Parallel Suffix Link Recovery

tasks (that span multiple suffix sub-trees) to disk. To do so efficiently, each tree is written to by a single processor. All the writes associated with this tree are exchanged between the processors using collective communication. We repeat the above procedure iteratively until all suffix sub-trees and their associated tasks are processed. The second phase is the more time consuming phase in the link recovery process – by accessing the input string in a tiled fashion and maximizing tree reuse, the process is more I/O efficient when compared with existing techniques. We would like to point out that since suffix links are always guaranteed to exist, during this phase, we only need to access the string referenced in the link recovery tasks (for example, T_k^a), and not the string referenced by the suffix sub-tree being processed in the iteration (for example, T_a).

4.6 Complexity Analysis

In the worst case, the running time of the algorithm is $O(n^2/C)$. Sub-tree construction is the most expensive stage. Each processor inserts $\Theta(n/C)$ suffixes into the tree; and it takes at most $O(n)$ time to insert one suffix. If the string comes from a symmetric Bernoulli distribution, the running time of the algorithm is $O(n \log n/C + n^2/(BC))$ as it takes only $O(\log n)$ time to insert one suffix. The input string is accessed $O((n/B)^2)$ times, each time reading B consecutive bytes. Typically, n/B is a constant and hence the algorithm scales as $O(n \log n/C)$.

4.7 Memory Budget Allotment

To afford an in-memory execution, we need to maintain two entities in main memory – the suffix sub-tree and the input string blocks. Hence, we need to pick MTS and B such that $MTS + 2B < M$. When one increases MTS and correspondingly decreases B, tiling overheads reduce, but the I/O cost increases quadratically in the worst case. On the other hand, when one decreases MTS and correspondingly increases B, tiling overheads increase, but I/O costs decrease. Hence, one needs to pick these two parameters while being cognizant of their tradeoffs. We will evaluate this tradeoff in our experiments.

5. EXPERIMENTAL EVALUATION

In this section, we present results of our performance evaluation. We do not evaluate suffix tree query processing performance as we believe that the task of query processing is orthogonal to that of tree construction – one can always employ a suitable layout for query processing when writing out the final suffix tree to the file system. We would like the reader to note that others have made such evaluations [21, 25] and shown that large disk-based suffix trees afford very reasonable query processing times. We use the distributed memory IBM Blue Gene/L system with 1024 PowerPC 440 processors at 700 MHz and 512 MB of main memory per processor for our evaluation. This system has a three-dimensional torus network for point-to-point communication and a global tree network for collective communication. The implementations were compiled using g++ version 3.4.3 and MPI[6] is used for message passing. We are not aware of any existing parallel suffix tree construction algorithm that can handle out-of-core input strings and hence do not compare with any existing algorithm.

5.1 Parallel Scalability

We measured execution time as we varied the number of processors from 16 to 1024, over different data sets. All the data sets were drawn from the Human genome (that has an alphabet size of 4). Execution times include the time for both suffix tree construction and link recovery and are presented in Figures 6-7. We measure execution times separately for different allocations of the memory budget – 50S-50T indicates that 50% of the memory budget was allocated to the string blocks and string cache, and the remaining 50% was allocated to the suffix sub-tree.

We make the following observations:

- On the smallest data set (125 MB), we obtain a maximum cumulative speedup of only 6.72 – the maximum possible speedup being 64. In fact, our gains are minimal beyond 64 processors on this data set. This is attributed to the fact that as we increase the number of processors on this data set, we need to build smaller suffix sub-trees to keep all the processors busy. As a result, tiling and synchronization overheads account for a significant fraction of the execution time, limiting scalability.

- As we increase the size of the data set from 250 MB to 3000 MB, maximum cumulative speedup increases from 16.1 to 34.7 (the maximum possible speedup is 64). For a fixed memory budget, increasing the number of processors does not require building smaller suffix sub-trees, as we increase the length of the string. Hence, scalability improves as we increase the size of the data set. Furthermore, this suggests that as we

[6] `http://www.mpi-forum.org/`

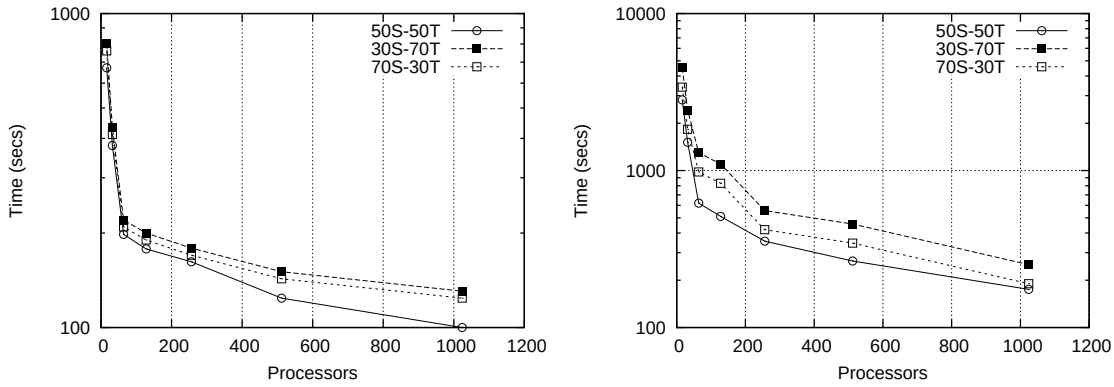

Figure 6: Parallel execution times as we increase the number of processors – data set sizes 125 MB (left) and 250 MB (right).

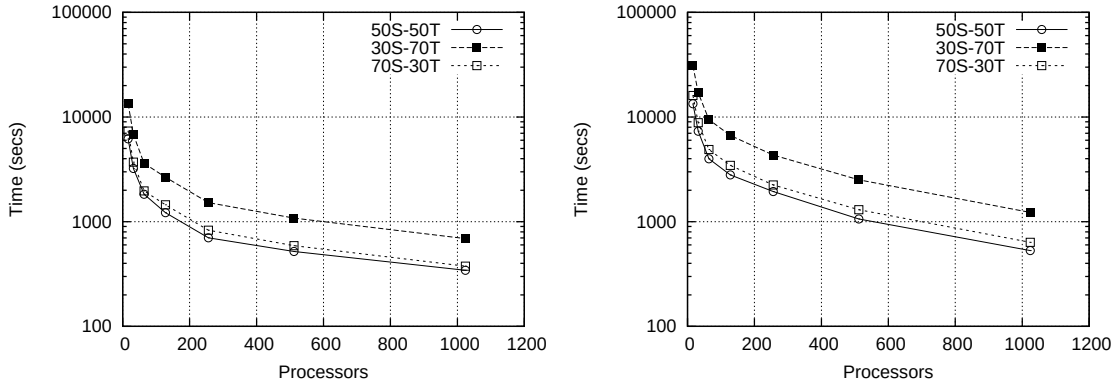

Figure 7: Parallel execution times as we increase the number of processors – data set sizes 500 MB (left) and 1000 MB (right).

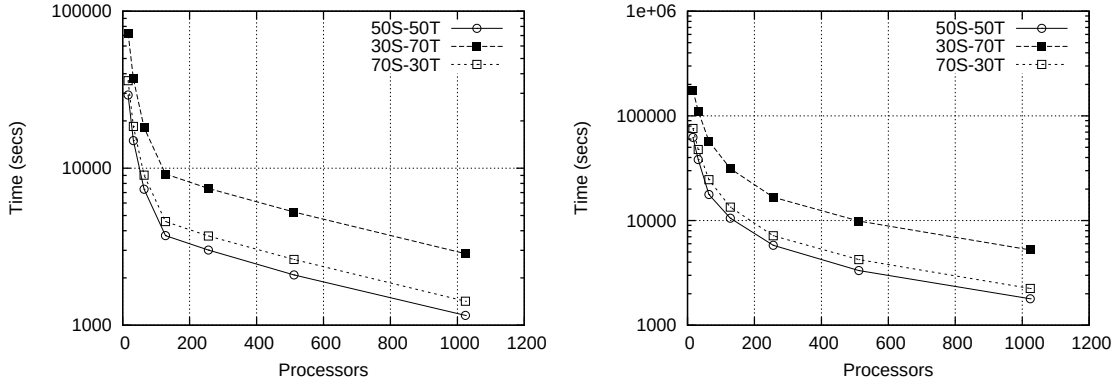

Figure 8: Parallel execution times as we increase the number of processors – data set sizes 2000 MB (left) and 3000 MB (right).

increase the size of the data set beyond 3000 MB, we should continue to deliver high performance provided we have sufficient network I/O bandwidth for reading and writing the suffix sub-trees.

- A 50S-50T allocation of the memory budget consistently provides the best performance. The 30S-70T allocation provides the worst performance; reducing the space available for the string blocks and string cache results in smaller string blocks (increasing the total number of string blocks) – execution time scales quadratically with the number of string blocks.

- We operate at an efficiency of over 50% when indexing the Human genome on 1024 processors. We argue that this number is relatively high given the I/O-intensive nature of the suffix tree construction and link recovery processes. On 1024 processors, suffix tree construction for the entire Human genome (without suffix links) takes *880 seconds*, and suffix link recovery takes an additional *910 seconds.* It may appear that these execution times are rather large considering that we are using 1024 processors. This is attributed to the fact that each processor on the IBM Blue Gene is approximately ten times slower than the Intel/AMD processors used for serial performance evaluation by us and others [21, 25]. The next generation of the IBM Blue Gene family will have a much faster processor and should reduce execution times further.

6. CONCLUSIONS

Aggressive DNA sequencing efforts have resulted in genomic sequence data sets growing at an ever increasing pace. To gain from these advances, it is imperative that we have efficient methods to index and query genomic sequences. Suffix trees have often been used for indexing such data, but large tree construction times and difficulty in parallelizing extant algorithms have limited their usability. To address this challenge, in this paper, we presented an algorithm to index genomic sequences of any length on massively parallel systems like the IBM Blue Gene/L. The algorithm builds a suffix tree by simultaneously tiling accesses to both the input string as well as the partially constructed suffix tree. Together with effective collective communication and in-network caching, the approach allows for scalable parallel suffix tree construction. We empirically evaluated our algorithm and showed that in a parallel setting with 1024 processors and very limited main memory per processor the algorithm provides excellent scale-up and is capable of indexing the entire Human genome in under 15 minutes at an efficiency in excess of 50%.

7. ACKNOWLEDGMENTS

We would like to thank all the anonymous reviewers for their valuable suggestions.

8. REFERENCES

[1] A. Apostolico, C. Iliopoulos, G. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3(1-4), 1988.

[2] S. Bedathur and J. Haritsa. Engineering a fast online persistent suffix tree construction. In *Proceedings of the IEEE International Conference on Data Engineering*, 2004.

[3] N. Bray, I. Dubchak, and L. Pachter. AVID: A global alignment program. *Genome Research*, 13(1), 2003.

[4] A. Brown. Constructing genome scale suffix trees. In *Proceedings of the Asia-Pacific Bioinformatics Conference*, 2004.

[5] A. Carvalho, A. Freitas, A. Oliveira, and M. Sagot. Efficient extraction of structured motifs using box links. In *Proceedings of the 11th Conference on String Processing and Information Retrieval*, 2004.

[6] W. Chang and E. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5), 1994.

[7] C. Cheung, J. Yu, and H. Lu. Constructing suffix trees for gigabyte sequences with megabyte memory. *IEEE Transactions on Knowledge and Data Engineering*, 17(1), 2005.

[8] A. Delcher, S. Kasif, R. Fleischmann, J. Peterson, O. White, and S. Salzberg. Alignment of whole genomes. *Nucleic Acids Res.*, 27(11), 1999.

[9] A. Delcher, A. Phillippy, J. Carlton, and S. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Res.*, 30(1), 2002.

[10] A. Ghoting and K. Makarychev. Serial and parallel methods for i/o efficient suffix tree construction. In *Proceedings of the ACM International Conference on Management of Data*, 2009.

[11] D. Gusfield. *Algorithms on strings, trees, and sequences: Computer science and computational biology.* Cambridge University Press, Cambridge, 1997.

[12] D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *Journal of Computer and System Sciences*, 69(4), 2004.

[13] R. Hariharan. Optimal parallel suffix tree construction. In *Proceedings of the Symposium on Theory of Computing*, 1994.

[14] E. Hunt, M. Atkinson, and R. Irving. A database index to large biological sequences. In *Proceedings of 27th International Conference on Very Large Databases*, 2001.

[15] R. Japp. The top-compressed suffix tree: A disk resident index for large sequences. In *Proceedings of the Bioinformatics Workshop at the 21st Annual British National Conference on Databases*, 2004.

[16] S. Kurtz, J. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich. Reputer: The manifold applications of repeat analysis on a genome scale. *Nucleic Acids Res.*, 29, 2001.

[17] S. Kurtz, A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. Salzberg. Versatile and open software for comparing large genomes. *Genome Bio.*, 5(R12), 2004.

[18] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2), 1976.

[19] C. Meek, J. Patel, and S. Kasetty. Oasis: An online and accurate technique for local-alignment searches on biological sequences. In *Proceedings of 29th International Conference on Very Large Databases*,

2003.

[20] NCBI. Public collections of dna and rna sequence reach 100 gigabases. `http://www.nlm.nih.gov/news/press_releases/dna_rna_100_gig.html`.

[21] B. Phoophakdee and M. Zaki. Genome-scale disk-based suffix tree indexing. In *Proceedings of the ACM International Conference on Management of Data*, 2007.

[22] B. Phoophakdee and M. Zaki. Trellis+: An effective approach for indexing massive sequences. In *Proceedings of the Pacific Symposium on Biocomputing*, 2008.

[23] G. M. Rubin, M. D. Yandell, J. R. Wortman, G. L. Gabor Miklos, C. R. Nelson, I. K. Hariharan, M. E. Fortini, P. W. Li, R. Apweiler, W. Fleischmann, J. M. Cherry, S. Henikoff, M. P. Skupski, S. Misra, M. Ashburner, E. Birney, M. S. Boguski, T. Brody, P. Brokstein, S. E. Celniker, S. A. Chervitz, D. Coates, A. Cravchik, A. Gabrielian, R. F. Galle, W. M. Gelbart, R. A. George, L. S. Goldstein, F. Gong, P. Guan, N. L. Harris, B. A. Hay, R. A. Hoskins, J. Li, Z. Li, R. O. Hynes, S. J. Jones, P. M. Kuehl, B. Lemaitre, J. T. Littleton, D. K. Morrison, C. Mungall, P. H. O'Farrell, O. K. Pickeral, C. Shue, L. B. Vosshall, J. Zhang, Q. Zhao, X. H. Zheng, F. Zhong, W. Zhong, R. Gibbs, J. C. Venter, M. D. Adams, and S. Lewis. Comparative genomics of the eukaryotes. *Science*, 287(5461), 2000.

[24] K. Schurmann and J. Stoye. Suffix tree construction and storage with limited main memory. Technical report, Universitat Bielefeld, 2003.

[25] Y. Tian, S. Tata, R. Hankins, and J. Patel. Practical methods for constructing suffix trees. *VLDB Journal*, 14(3), 2005.

[26] E. Ukkonen. Constructing suffix trees on-line in linear time. In *Proceedings of the IFIP 12th Work Computer Congress on Algorithms, Software, Architecture: Information Processing*, 1992.

[27] P. Weiner. Linear pattern matching algorithms. In *Proceedings of 14th Annual Symposium on Switch and Automata Theory*, 1973.