TECHNICAL UNIVERSITY OF CRETE

SCHOOL OF ELECTRONIC AND COMPUTER ENGINEERING

MASTER THESIS

Suffix Trees Construction Algorithms

Ioannis Flouris

Supervisor: Antonios Deligiannakis, Assistant Professor

CHANIA

DECEMBER 2013

MASTER THESIS

Suffix Trees Construction Algorithms

Ioannis Flouris

Mail: giannisflouris@hotmail.com

SID: 2010039003

Supervisor: Antonios Deligiannakis, Assistant Professor

Abstract

Suffix trees are widely used for indexing biological data. Their use is crucial for search algorithms applied in biology. In the past years there has been an explosive growth in biological data. The aim of this thesis is to study the main algorithms proposed over the years for constructing suffix trees and evaluate both their advantages and bottlenecks.

Through the aforementioned study and with regard to the inherit parallelism of Hadoop Map-Reduce an algorithm is proposed that balances the requirements of an out-of-core suffix tree construction algorithm in a parallel fashion. The benefits of parallelism as well as serial I/O and a cache policy for reading the input string resulted in an efficient algorithm, in both time and space, for constructing a suffix tree for large DNA sequences.

In the experiments conducted it is shown that genome-scale input can be indexed in less than 40 minutes proving that Hadoop Map-Reduce can be used for efficient suffix tree construction.

Table of contents

Figures

Tables

Acknowledgements

Primarily, I would like to thank my family and friends for their unwavering support throughout the course of my master.

I would also like to make a special reference to my professor Antoni Deligiannaki for his help, guidance and understanding and all professors of ECE that I was taught from.

I would also like to thank my fellow students and friends Vaggeli Vazaio, Gianni Christodoulou and Giorgo Ktistaki for their selfless and generous help as well as Xenia Arapi for her endless technical support.

Finally I would like to thank my fellow students Katerina Asimoglou, Julie Vlachou and Stelio Mamma who made my stay in TUC much more pleasant.

Chapter 1 - Introduction

1.1 DNA sequences

Nowadays, textual databases are among the most rapidly growing collections of data. One of these collections, the collection of sequenced genomes, is a textual database where the genomes of different organisms are represented as strings of characters over the 4-letter alphabet *{A, C, G,T}* of DNA bases. The reason life-encoding sequences are put into a digital form is to enable computer programs to extract useful information from this raw data; something human, unassisted by software, could not do, no matter the amount of training.

Even for computer programs, both the size of the genomic sequences and their total number are large. From 1982 to the present, the size of the publicly available GenBank sequence database is doubling approximately every 18 months. The Human genome project, officially completed in 2000, produced, in addition to the draft sequence of Human genome, a massive database of complete reference genomes of different species. As a result, the total size of the GenBank has reached 142 GB. The size of data in the Whole Genome Shotgun (WGS) sequencing project stands currently at about 422 GB [20].

In addition, an unprecedented growth rate of genomic data is expected in the near future. Starting in 2008, the "1000 Genomes Project" has been launched [19] with the ultimate goal of collecting sequences of additional 1,500 Human genomes, 500 each of European, African, and East Asian origin. This will produce an extensive catalog of human genetic variations. The size of just the raw sequences in this catalog would be about 5 TB.

The search and comparison of sequences in such massive collections requires efficient and highly scalable algorithms. However, it is hard to develop such efficient solutions using raw sequences alone. Due to the massive and relatively static nature of the sequence data, it would be very useful to preprocess it into a comprehensive index where all the required information could be efficiently located.

1.2 DNA as digital data

The sequenced genomes represent a new type of digital data that differs from numerical or textual data existed before. Though these sequences can be regarded as strings, one cannot blindly adopt techniques used for indexing natural language texts, since there are fundamental differences between these two kinds of data. The main four features that distinguish DNA data from natural language texts are outlined below.

First of all, the total size of inter-related information is several orders of magnitude larger in DNA than in typical natural language texts. Even an entire book has a moderate size compared to the inter-related information in one "volume" of a genome - one chromosome. For example, the size of middle sized book does not exceed two million characters, while the Human chromosome I is encoded in a sequence of 247 million characters.

The second major difference is the size of tokens. Here, a token is thought as a separate meaning-bearing entity. The size of tokens in natural languages is small, and is equal to the length of separate words or phrases. In contrast, even if we consider genes – the protein-coding units of genome – as DNA "tokens", these units are several orders of magnitude larger than a typical word or phrase.

While we do not have a clear division of DNA sequences into tokens (yet), we may consider to index each different substring that occurs in genomic databases. However, the total number of different substrings to index (2×10^{17} different substrings only in sequences of the Human genome) is significantly larger than the total number of all possible words in one natural language (about 171, 476 words in the Oxford English dictionary).

The last distinctive feature of DNA sequences is the number of different strings with the same meaning. Many "misspelled" tokens of DNA encode for the same output. For example, the *G*ATATATA and TATATAT*T* sequences bear the same meaning in a sense that they initiate the production of the same amount of the same protein.

Due to all these differences, the indexes that work well for natural language texts (such as *inverted indexes* or *B-trees*) cannot be efficiently used for indexing DNA sequences.

1.3 Indexing DNA sequences

We want to create an index that will help locating all the positions of substrings that are equal to the query pattern *q* (*an exact pattern matching problem*). An index that indexes all different substrings of a given text is called *a full-text index.* Examples of such indexes are *suffix trees* [1], *suffix arrays* [4], and *string B-trees* [5].

We have chosen to work towards the more efficient construction of **suffix trees**. A suffix tree not only indexes all distinct substrings of a given text (as a suffix array or string B-tree does), but also "exposes the internal structure of a string in a deeper way [6] (p. 89)", and as such provides efficient solutions to many string problems more complex than exact pattern matching. This includes the approximate pattern matching – the fundamental task in the biological sequence analysis. It was Weiner [2] who initially proposed to use a suffix tree as an explicit index.

Informally, a suffix tree is a digital tree of all suffixes of a given string. Figure 1 depicts the suffix tree for string *ababc*. Each suffix is inserted into the tree: the suffix starting at position 0 – *ababc*, at position 1 – *babc*, at position 2 – *abc*, an so on.

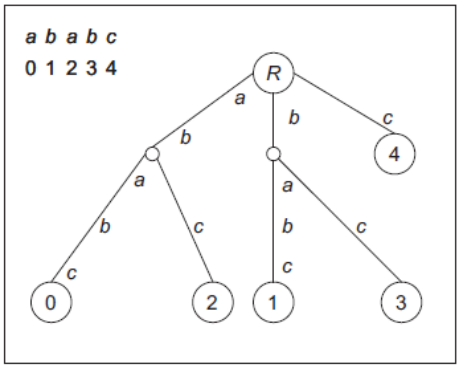


Figure 1: The suffix tree for string "ababc"

Once the suffix tree is built, we can answer multiple combinatorial questions about the input string. For example, with the help of suffix tree we can count the total number of distinct substrings of the input string. This application is based on a fundamental property of a suffix tree: every distinct substring of the input string *X* is spelled out exactly once along a path from the root of the suffix tree. Thus an inventory of all the distinct substrings of *X* can be produced by listing all the strings along each such path. The total number of distinct substrings may be quadratic in the input length. We can count the total number of all distinct substrings by simply adding up the lengths of the edges of the suffix tree in time linear in *N*. For example, there are 12 different substrings in the string *ababc*, as can be calculated from the suffix tree in Figure 1. Thus, the suffix tree represents a compact index of all distinct substrings of a given input string: it represents a quadratic number of substrings in a linear space.

The exact pattern matching using suffix trees is very efficient. In fact, each query pattern *q* can be located in *X* by following the path of symbols from the root of the suffix tree for *X*. The substring containment problem, namely whether *q* is a substring of *X*, can be solved in time proportional to the length of the query *q* and *independent* of the size of the pre-processed input. In order to report all occurrences *occ* of *q*, the subtree induced by the end of the corresponding path is traversed, which results in a search with an optimal performance of $O(|q| + occ)$.

These are only some examples of the potential use of the suffix trees. In fact, theoretically optimal bounds were obtained for many non-trivial tasks such as computing matching statistics, locating all repetitive substrings, extracting palindromes and so on [6]. As elegantly put by Gusfield "the best way to appreciate the power of suffix trees is to spend some time trying to solve these problems without using suffix trees. Without this effort and without some historical perspective, the availability of suffix trees may make certain problems appear trivial, even though linear-time algorithms for those problems were unknown before the advent of suffix trees ".

1.4 Thesis objective and organization

However, all the benefits described in his book [6] appear to be illusory, since, the restraints of both time and space has made the suffix tree difficult to handle. Thus, this work is dedicated to the engineering of practical algorithms for the construction of the suffix trees for genome-scale inputs by taking advantage the inherit parallelism of Hadoop Map-Reduce.

The rest of the thesis is organized as follows. In Chapter 2 we give a formal definition of the suffix tree data structure and present the main challenges that we faced on the way to efficient suffix tree construction. Then, in Chapter 3 we describe the main algorithms that addressed the problem of suffix tree construction in the past years. In chapter 4 we present the Hadoop Map-Reduce platform on which we worked. Then in chapter 5 we introduce our solution for building suffix trees for large inputs and in Chapter 6 we present our experimental results. Finally in chapter 7 we point out the challenges we met, the conclusions we drew and future work that can be done.

Chapter 2 - Suffix tree basics

In this chapter we describe the problems that we solve in this work. First, we give an introduction to suffix trees in Section 2.1. Next, in Section 2.2 we describe its space requirements, which cause problems in the construction and storage of these full-text indexes. In Section 2.3 we outline two possible solutions of the suffix tree memory problem: index compression and use of external storage (disk).

2.1 The suffix tree

The suffix tree was introduced by Weiner in 1973 [2]. We start out by taking a closer look at the suffix tree data structure and its representation. First, we equip ourselves with some useful definitions.

We consider a *string* $X = x_0x_1 \ldots x_{N-1}$ to be a sequence of N symbols. $N - 1$ symbols are over an alphabet Σ. The last symbol x_{N-1} = $, which we attach to the end of X is unique and not in Σ (a so-called *sentinel*). Depending on the application, we regard $ as having a lexicographical value lower or greater than any other symbol.

By $Si = X[i, N - 1]$ we denote a *suffix* of X beginning at position i, $0 \leq i < N$. Thus $S_0 = X$ and S_{N-1} = $. Note that we can uniquely identify each suffix by its starting position.

Prefix P_i is a substring $[0, i]$ of X. The *longest common prefix* LCP_{ij} of two suffixes S_i and S_j is a substring $X[i, i + k]$ such that $X[i, i + k] = X[j, j + k]$, and $X[i, i + k + 1] \_= X[j, j + k + 1]$. For example, if $X = ababc$, then $LCP_{0,2} = ab$, and $|LCP_{0,2}| = 2$.

If we sort all the suffixes of string X in lexicographical order and record this order into an array SA of integers, then we obtain the *suffix array* of X [4]. SA holds all integers i in the range $[0,N]$, where i represents Si. In more practical terms, suffix array SA is an array of positions sorted according to the lexicographic order of their corresponding suffixes. Note that the suffixes themselves are not stored in this array but are rather represented by their start positions. For example, for $X = ababc\$$ $SA = [5, 0, 2, 1, 3, 4]$. The suffix array can be augmented with the information about the longest common prefixes for each pair of suffixes represented as consecutive numbers in SA, as shown in the example of Figure 2.

A *trie* is a type of digital search tree. In a trie, each edge represents a character from the alphabet Σ. The maximum number of children for each trie node is /Σ/, and sibling edges must represent distinct symbols. A *suffix trie* is a trie for all the suffixes of *X*. As an example, the suffix trie for *X* = *ababc* is shown in Figure 2 [Left]. Beginning at the root node, each of the suffixes of *X* can be found in the trie: starting with *ababc*, *babc*, *abc*, *bc* and finishing with a *c*. Because of this organization, the occurrence of any query substring of *X* can be found by starting at the root and following matches down the trie edges until the query is exhausted. In the worst case, the total number of nodes in the trie is quadratic in *N*. This situation arises, for example, if all the paths in the trie are disjoint, as for the input string *abcde*.

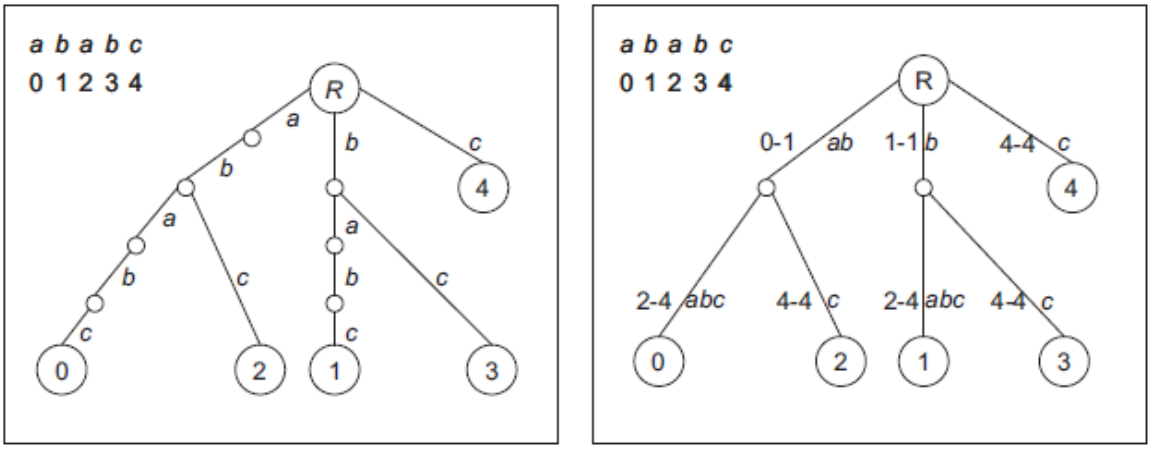


Figure 2 : [left] The suffix trie for "ababc" [right] the suffix tree for input "ababc"

The number of edges in the suffix trie can be reduced by collapsing paths containing unary nodes into a single edge. This process yields a structure called the *suffix tree*. Figure 2 [Right] shows what the suffix trie for *X* looks like when converted to a suffix tree. The tree still has the same general shape, but far fewer nodes. The leaves are labeled with the start position in *X* of corresponding suffixes, and each suffix can be found in the tree by concatenating substrings associated with edge labels. In practice, these substrings are not stored explicitly but are represented as an ordered pair of integers indexing its start and end position in *X*. The total number of nodes in the suffix tree is constrained due to two facts: (1) there are exactly *N* leaves and (2) the degree of any external node is at least 2. There are therefore at most $N - 1$ internal nodes in the tree. Hence, the maximum number of nodes (and edges) is linear in *N*. The tree's total space is linear in *N* for the case that each edge label can be stored in constant space.

Fortunately, this is the case for an implicit representation of substrings by their positions.

Overall, a *suffix tree* is a digital tree of symbols for the suffixes of *X*, where edges are labeled with the start and end positions in *X* of the substrings they represent. Note also that each internal node in the suffix tree represents the end of the longest common prefix for some pair of suffixes.

2.2 Storage requirements

In this section we will discuss the suffix tree representation in memory in order to estimate the space requirements for suffix trees.

The main elements of a suffix tree are its nodes and its edges. Nodes can be either internal or leaves with the later having no outgoing edges. Since we use a four letter alphabet (DNA alphabet) there can be at most 4 edges per internal node and each edge has to store two integers the start and the end position of the corresponding substring of *X.* In the case of leaves the end position is N (the size of the input string). Since, only internal nodes have edges and thus need the start and end position instead of using 4 pointers in each node for its edges we can use 2-d arrays and consequently need 1 pointer per internal node for all its edges.

For purposes of faster tree traversal each edge is best to store the first character also represented by its start position. This very useful feature is easily implemented, since by definition two edges of the same node cannot start with the same character, by storing edges in specific places of the 2-d array (edge starting with A in the first position, with C in the second and so on).

There are many ways proposed to implement the suffix tree including hash tables, linked lists and arrays. In the current implementation the arrays are selected since arrays are the cheapest data structure with the less memory overhead by the JVM in Java.

An array of integers for the nodes is necessary of size 2*N since it is proven that in a suffix tree there can be at most 2*leaves total nodes. In this array we store either a pointer for the edges, in the case of internal nodes, or the suffix position in the input

string, in the case of leaves. A 2-d array represents the edges, the first dimension is 4 corresponding to all possible edges and the second dimension has size of N. Each cell is a pointer to the end node of this edge and stores the offset index of the nodes array whether the end node is an internal node or a leaf node. The third array required to complete the representation is an array with the start and end positions. An array of integers is used, with size of 8*N, 8 integers for the 4 edges of each node of start and end positions.

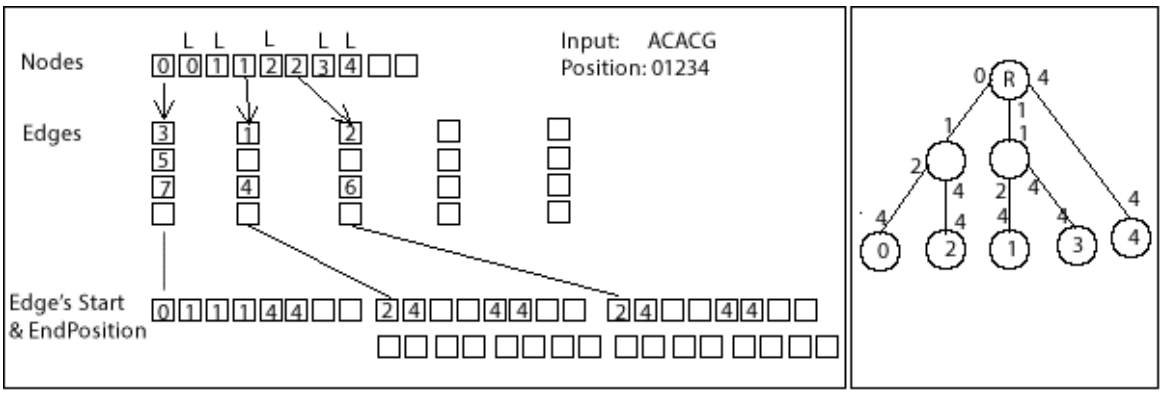


Figure 3: Input string "ACACG" [Left] Suffix Tree memory representation [Right] Conceptual representation

In figure 3 we can see an example of a very small suffix tree for input string ACACG. Since we have 5 suffixes (N=5) we will create an integer array for the nodes with 2*N places, an edges integer 2-d array with 4*N places and an edges start and end position integer array with 8*N places. Hence, the suffix tree needs 14*N integers and with 4 bytes per integer, the total amount of memory required for the above representation is 280 bytes for N=5 suffixes which means 56 bytes are needed per suffix or that the final suffix tree will be 56 times larger than the input string's length. The "L" notation placed above some cells in the nodes array in figure 3, indicate that these are leaves although such information is not stored explicitly but rather deduced from the fact that the edge has end index equal to the string's length. The information stored in the nodes array in the case of a leaf is the suffix start number in the input string.

Another representation, that has also been tested, is without the end position of edge. An additional array is necessary to distinct the nodes from the leaves and to store the information of which is the first edge added in a node. This array can be a byte array of size equal to the nodes' array (that is 2*inserted suffixes). The end position can be deduced from the end node's first edge start position and the resulting tree can be represented consuming 42 times the number of the inserted suffixes.

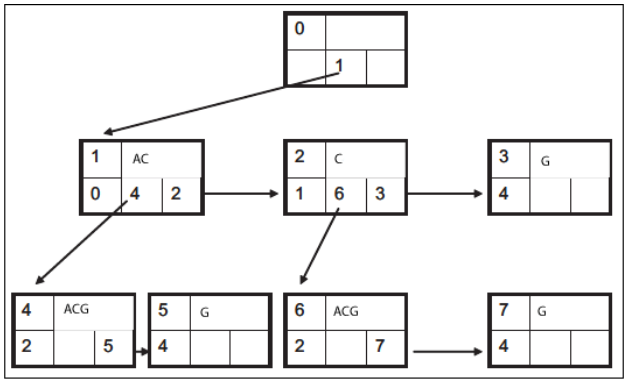


Figure 4: McCreight linked-list representation

The aforementioned representations are not optimal. A linked-list representation has been proposed by McCreight in [1]. In this implementation, the suffix tree is represented as a set of node structures, each consisting of the start position of the substring labeling the incoming edge, together with two pointers – one pointing to the node's first child and the other one to its next sibling as illustrated in figure 4. The end position of the edge is not explicitly stored since for an internal node it can be deduced from the start position of its first child and for a leaf node the end position is N as mentioned before. The McCreight representation is at most 25N. In figure 4 the McCreight representation is illustrated for input ACACG.

Other representations and optimizations have also been proposed achieving down to 18N by Griegerich et al's [16] but have not been chosen in the current implementation due to the overhead added by the JVM for objects (8 byte header for each instance and tail big enough that the total memory of the object is multiple of 8) which eliminates all gain. Another reason for choosing arrays arrives from the fact that when searching a node for an edge starting from a character is accomplished by a single check whereas in the linked list implementation there can be as many as 4 checks adding complexity to both creation and query phase.

Since genome-scale inputs are prohibitive for constructing in memory suffix trees the disk-based concept for building the suffix tree is inevitable. The main problem of the disk-based concept since the tree does not fit the main memory is to break the tree in smaller trees easily fitting the main memory. Finding prefixes that represent all suffixes for a given input becomes necessary. Prefix creation will be discussed more thoroughly in the next chapter. Another subject to be addressed, in the next chapters, is which of the two proposed representations of arrays (the one with the end positions and the one without) is better in terms of both time and space efficiency.

2.3 Suffix Tree partitioning

Creating suffix trees for very large inputs entirely in main memory is proven to be an impossible task since even with the best representation requiring 18 times the input length. For inputs as large as the human genome at least 50 GB of memory are required a number that is multiplied if we surpass the storage capabilities of a 4 byte integer. With the aforementioned observations we can conclude that we must use a partitioning scheme for the suffix tree in order to facilitate the high demands of its construction.

The first partitioning scheme that doesn't affect the suffix tree's consistency was proposed by E. Hunt in [7] made with the simple observation that you can divide a suffix tree based on fixed length suffixes. The main goal is to create suffix trees that can be kept throughout their construction in main memory. If every sub-tree of the original suffix tree that starts with a fixed length prefix fits entirely in main memory then the original suffix tree can be perceived as a forest of fixed length prefixed sub-trees.

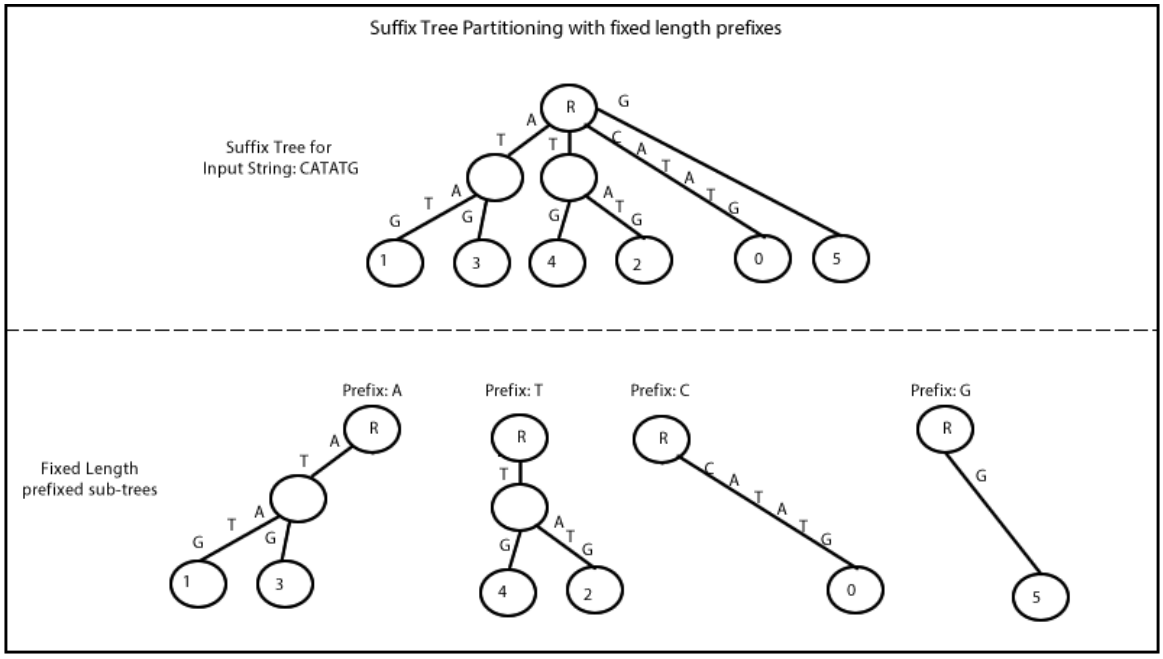


Figure 5: Suffix Tree partitioning with fixed length prefixes

The basic idea of the fixed length partitioning is illustrated in figure 5 where an example of a simple suffix tree is partitioned with prefixes of 1 character resulting in a forest of 4 sub-trees and without the loss of information. The resulting sub-trees are ensured to fit the main memory for both construction and querying. This means that the total number of suffixes that can be inserted in a sub-tree is bound by the available main

memory. If the number of suffixes exceeds this bound then we will use 2 character prefixes resulting in 8 prefixes and sub-trees and so on. The total amount of trees is found by the general type: where Σ is the number of symbols in our alphabet and length is the number of our characters in our fixed length prefixes.

Although it was a rather straightforward approach in real DNA sets the resulting suffix trees were uneven in size leading the construction algorithms to have unused memory during of construction of small trees and full memory for larger trees. This observation led Phoophakdee and Zaki in [11] to propose a variable-length partitioning scheme.

The variable-length partitioning scheme starts with a fixed length set for prefixes and increases the length by appending all characters of the alphabet to those prefixes that don't comply with a given threshold. All those prefixes that complied with the threshold are kept and based on those prefixes the construction algorithms will create the sub-trees.

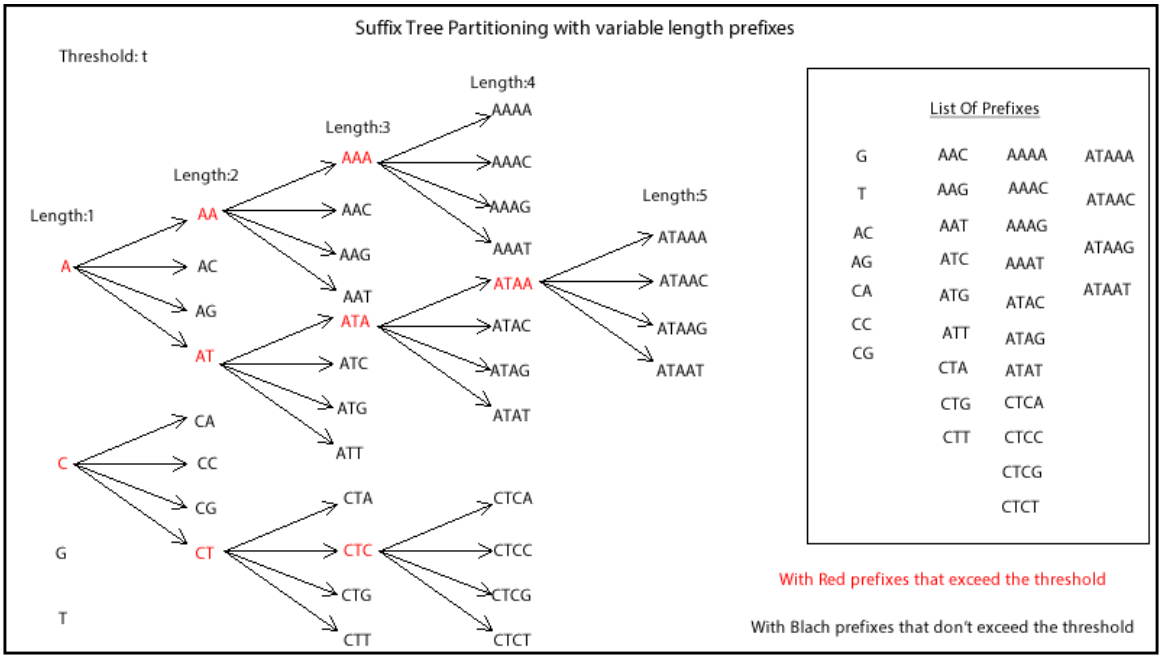


Figure 6: Suffix Tree partitioning with variable length prefixes

In figure 6 an example is illustrated for variable-length prefixes. With the variable length partitioning we've managed to efficiently split the suffix tree without any loss of information. Further discussion for the partitioning of the suffix trees will be made in the following chapters both from the perspective of the related work and the actual implemented in our algorithm.

Chapter 3 - Related work

In this chapter we will discuss the main algorithms that set the course of solving the problem of suffix tree creation in an efficient way based on Barsky's survey [18] and the authors' papers.

3.1 Linear time, in-memory algorithms

The first in memory algorithm for suffix tree construction came from Weiner in 1973 [2]. Although it was linear in time, Weiner's algorithm was space expensive. The second algorithm with O(n) time complexity was from McGreight [1] in 1976 but his algorithm had $O(n^2)$ space requirements. The first O(n) in both time and space algorithm was brought by Ukkonen in 1995 [3].

Ukkonen's main idea was to insert each character at a time and gradually build the whole suffix tree. Ukkonen's algorithm starts with the empty tree (the start node) and then progressively builds an intermediate suffix tree for each prefix. In order to convert that intermediate suffix tree into a true suffix tree all suffixes are extended with the next character until all input is read. The suffixes inserted in all intermediate steps may end in three types of nodes: leaf nodes, internal nodes or in the middle of an edge. Leaf nodes will not change in the course of the algorithm.

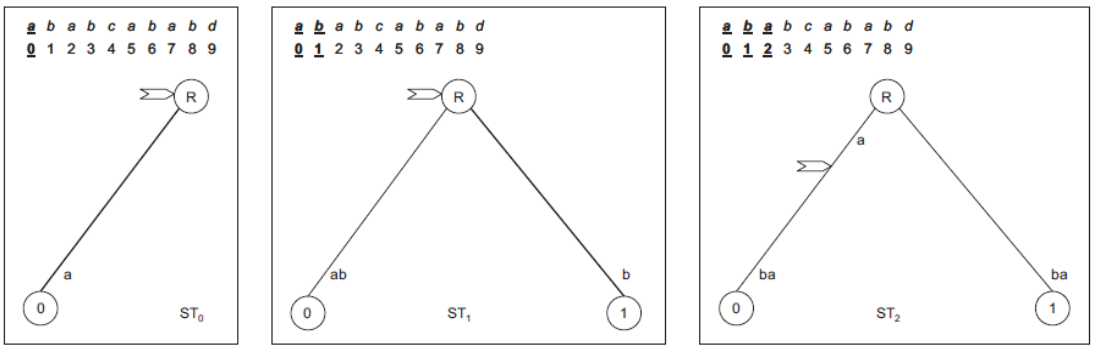


Figure 7: Ukkonen's Algorithm – The first 3 steps

There are several heuristics used by Ukkonen to make the aforementioned algorithm in linear time and space. A heuristic worth mentioning is that apart from leaf nodes all other types of nodes are still active until a leaf is created. This allows extending or splitting edges and nodes more efficiently and less time consuming. Also in order to

be space efficient Ukkonen's algorithm doesn't use actual string to label edges the but rather the start and the end position of the substring in the input string.

Ukkonen's algorithm is the optimal algorithm in both time and space for constructing suffix trees when both the input string and the suffix tree fits in main memory. If the suffix tree exceeds the main memory the algorithm degrades rapidly through excessive I/O.

The main reasons for Ukkonen's degradation when we exceed main memory availability are that at first a single node access requires an entire random I/O. This access time depends on the disk place of the next access point. Moreover, since the edges of the tree are not labeled with actual characters, it is important that we access randomly the input string in order to compare the incoming character with the characters of the input string encoded as positions in the suffix tree edges. Unfortunately, this leads to a very impractical performance, since the algorithm spends all its time moving the disc head from one disc allocation to another.

As a result the following algorithms tried mainly to handle the memory bottleneck in suffix tree construction, focusing mainly in eliminating random I/O over the input string or splitting the tree appropriately that would enable us to fit the sub trees entirely in main memory. The first attempts to address the problem created algorithms that consider that the input string fits entirely in main memory.

3.2 Semi-disk based algorithms

The semi-disk based algorithms is a category of algorithms that consider that the input string fits entirely in main memory but the suffix trees that are created are much larger than main memory. These implementations have the limit that the input string must be less than the available main memory in our machine. These algorithms take the suffix tree creation to the next level making the indexing of real data a feasible and less time consuming task.

3.2.1 Hunt's Algorithm (2001)

An intuitive method of constructing the suffix tree ST is the following: for a given string X we start with a tree consisting of only a root node. We then successively add paths corresponding to each suffix of X from the longest to the shortest.

Based on this brute-force approach, the first practical external memory suffix tree construction algorithm was developed in Hunt [7] incremental construction trades an ideal O(N) performance for locality of access to the tree during its construction. The output tree is in fact represented as a forest of several suffix trees.

The suffixes in each such tree share a common prefix. Each tree is built independently and requires scanning of the entire input string for each such prefix. The idea is that the suffixes that have prefix, say, aa fall into a different subtree than those starting with ab, ac and ad. Hence, once the tree for all suffixes starting with *aa* is built, it is never accessed again. The tree for each prefix is constructed independently in main memory, and then is written to disk.

We remark that we iterate through the input string as many times as the total number of partitions. The construction of a tree for each partition is performed in main memory. At the end, the suffix tree for each partition is written to disk. Note also that in order to perform the brute-force insertion of each suffix into the tree we need to randomly access the input string X, which therefore has to reside in memory. Since the input string is at least an order of magnitude smaller than the tree, this method efficiently addresses the problem of random accesses to the tree in secondary storage, but cannot be extended to inputs which are larger than the main-memory instantiation for holding X.

Hunt's algorithm is the first to introduce splitting the suffix tree in a forest of suffix trees each of which has a common prefix. The prefixes are chosen in such manner that no resulting ST would exceed the main memory. All prefixes have the same number of characters which can result in uneven in size suffix sub trees for skewed data which is often the case in DNA sequences.

Hunt proposed that a quadratic in time $O(n^2)$ brute force algorithm for constructing the suffix trees can be a better solution for constructing out of memory

suffix trees since it offers a better locality of reference during the construction since both the input string and the prefix sub-tree fit in main memory.

The performance of Hunt's algorithm degrades drastically if the input string does not fit the main memory and should be kept on disk. In this case we have O(pN) random accesses, this time to the input string. Time complexity for this algorithm is worst case $O(n^2)$ but it is shown that the average complexity is O(nlogn). For the Human DNA of size up to 247 MB (Human chromosome I) input, the suffix tree with Hunt et al.'s algorithm can be constructed in 97 minutes.

3.2.2 Distributed and Paged Suffix Trees (2003)

A similar idea of processing suffixes of X separately for each prefix was developed by Clifford and Sergot [9]. The distributed and paged suffix tree (DPST) by Clifford and Sergot, which was proposed first in context of distributed computation, has all the properties to be efficiently implemented to run using external memory.

As before, the suffixes of X are grouped by their common prefix whose length depends on the size N of X and the amount of the available main memory. The number of suffixes in each sub tree is small enough for the tree to be entirely built in main memory. Therefore, random disk access to the sub-tree during its construction is avoided. The main difference from Hunt's algorithm of the previous section is that the sub-tree for each particular prefix is built in an asymptotic time linear in N and not quadratic.

In order to do so, the DPST algorithm uses the idea to build the suffix tree on words. The main ideas are similar to the Ukkonen algorithm described in Section 1. However, the Ukkonen algorithm relies heavily on the fact that all suffixes of X are inserted, whereas the suffix tree on words is built only for some suffixes of X, namely the ones starting at positions marked by delimiters.

The DPST runs in time O(NP) where P is the total number of different prefixes. Despite the superior asymptotic internal running time compared to the previous algorithm, the practical performance and the scalability of the DPST as implemented were inferior to the program by Hunt for the real DNA data used in the experiments.

3.2.3 TOP-Q (2004)

Another attempt to address the memory bottleneck was in 2004 when Bedathur and Haritsa proposed a buffer management policy [10] to resolve Ukkonen's memory degradation for large suffix tree that could not fit in main memory.

In TOP-Q, Bedathur and Haritsa studied the patterns of node accesses during the suffix tree construction based on Ukkonen's algorithm. They found that the higher tree nodes are accessed much more frequently than the deeper ones. This gave rise to the buffer management method known as TOP-Q.

In this on-disk version of Ukkonen's algorithm, the nodes which are accessed often have a priority of staying in the memory buffer, and the other nodes are eventually read from disk. This significantly improves the hit rate for accessed nodes when compared to rather straightforward implementations.

However, in practical terms, in order to build the suffix tree for the sequence of the Human chromosome I (approximately 247 MB), the TOP-Q runs for 96 hours, as was recently evaluated using a modern machine, and cannot be considered a practical method for indexing large inputs and proves that Hunt's algorithm which dismisses Ukkonen's linear time construction algorithm is more practical for suffix trees that don't fit in the main memory.

3.2.4 Top-Down Disk-Based Suffix Tree Construction (2004)

Quadratic in the worst case, but a more elaborated approach of the Top Down Disk based suffix tree construction algorithm (TDD) [8] takes the performance of the on-disk suffix tree construction to the next level. The base of the method is the combination of the wotdeager algorithm of Giegerich and Hunt's prefix partitioning described above. Being still an $O(N^2)$ brute-force approach, TDD manages more efficiently the memory buffers and is a cache-conscious method which performs very well for many practical inputs.

The first step of TDD is the partitioning of the input string in a way similar to that of the algorithm by Hunt. Now, the tree for each partition is built as follows. The suffixes

of each partition are first collected into an array where they are represented by their start positions. Next, the suffixes are grouped by their first character into character groups. The number of different character groups gives the number of children for the current tree node. If for some character there is a group consisting only of one suffix, then this is a leaf node and is immediately written to the tree. If there is more than one suffix in the group, the LCP of all the suffixes is computed by sequential scans of X from different random positions, and an internal node at the corresponding depth is written to the tree. After advancing the position of each suffix by LCP's length, the same procedure as before is repeated recursively.

The main distinctive feature of the TDD construction is the order in which the tree nodes are added to the output tree. The tree is written in a top-down fashion, and the nodes which were expanded in the current iteration are not accessed anymore. This reduces the number of random accesses to the partially built tree and the new nodes can be written directly to the disk. The number of random disk accesses is O(P) as in Hunt's algorithm.

However, the size of each partition may be much bigger than before since now the main memory buffer for the suffix tree data structure does not have to hold an entire sub-tree. This pattern of accessing the tree was shown to be very efficient for cached architectures of the modern computer. It was even shown that the TDD algorithm outperforms the linear time algorithm by Ukkonen for some inputs in case when all the data structures fit the main memory. For the same input of 247 millions of symbols (Human chromosome I), which took about 97 minutes with the suffix-by-suffix insertion of Hunt, TDD builds the tree in 18 minutes.

As before, the algorithm performs massive random accesses to the input string when it does the character-by-character comparisons starting at different random positions. The input string for the TDD algorithm cannot be larger than the main memory.

Another problem of TDD is the suffix tree on-disk layout. The trees for different partitions are of different sizes, and some of them can be significantly bigger than the main memory. This poses some problems when loading the sub-tree into main memory for querying. If the entire sub-tree cannot be loaded into and traversed in the main

memory, the depth first traversal of such a tree requires multiple random accesses to different levels of on-disk nodes.

3.2.5 The partition-and-merge strategy of Trellis (2008)

The oversized sub trees caused by data skew can be eliminated by using set of different-length prefixes. In practice, the initial prefix size is chosen so that the total number of prefixes P will allow the process each of the P sub-trees in main memory. For example, we can hold in our main memory in total Tmax suffix tree nodes. The counts in each group of suffixes sharing the same prefix are computed by a sequential scan of input string X. If a count exceeds Tmax, then we re-scan the input string from the beginning collecting counters for an increased prefix length. Based on the final counts, none of which exceeds Tmax, the suffixes are combined into approximately even-sized groups.

As an example consider the case when suffixes starting with prefix ab occur twice more often than the suffixes starting with ba and bb. We can combine suffixes in partitions ba and bb into a single partition b with approximately the same number of suffixes as contained in partition ab. The maximum number of suffixes in each prefix partition is chosen to ensure that the size of the tree for suffixes which share the same prefix will never exceed the main memory. This is in order to ensure that each such subtree can be built and queried in main memory.

Based on this new partitioning scheme, Phoophakdee and Zaki proposed another method for creating suffix trees on disk; the Trellis algorithm [11]. The main innovative idea of this method is the combination of the prefix partitioning and the horizontal partitioning of the input into consecutive substrings, or chunks. In theory, the substring partitioning does not work for any input, since the suffixes in each substring partition do not run till the end of the entire input string. However, this horizontal partitioning works for most practical inputs.

Consider for example the Human genome sequence of about 3 GB in length. In fact, there is not a single string representing Human genome, but rather 23 sequences of DNA in 23 different Human chromosomes, with the largest sequence being only about 247 MB in size. Those chromosome sequences represent natural partitions of the entire

genome. If the size of each natural chunk of the input does not allow us to build the suffix tree for it entirely in main-memory, the chunk can be split into several slightly overlapping substrings. We append to the end of each such substring except the last one, a small “tail”, the prefix of the next partition. The tail of the partition must never occur as a substring of this partition. It serves as a sentinel for the suffixes of the partition, and its positions are not included into the suffix tree of the partition. In practice, for real-life DNA sequences, the length of such a tail is negligibly small compared to the size of the partition itself.

After partitioning the input into chunks of appropriate size, Trellis builds an independent suffix tree for each chunk. It does not output the entire suffix tree to disk, but rather writes to disk the different sub-trees of the in-memory tree. These sub-trees correspond to the different variable-length prefixes. Once trees for each chunk are built and written to disk, Trellis loads into memory the sub-trees for all the chunks which share the same prefix. Then it merges these sub-trees into the shared-prefix-based sub-tree for an entire input string.

The merge of sub-trees for different chunks is performed by a straightforward character-by character comparison, which leads to the same $O(N^2)$ worst-case internal time as the brute force algorithms described above. Trellis was shown to perform at speed comparable to TDD. Further, Trellis does not fail due to insufficient main memory.

If we have K chunks and P prefixes in the variable-length prefixes collection, the number of random disk accesses is O(KP). Since both K and P depend on the length of the input string N, the execution time of Trellis grows quadratically with the increase of N, and is therefore not scalable for larger inputs.

During the character-by-character comparison in the merge step, the input string is randomly accessed at different positions all over the input string. Therefore, the scalability of Trellis does not go beyond the size of the main memory designated for the input.

Trellis was the first algorithm to efficiently address the memory bottleneck and to give a solution using smart techniques inspired by Ukkonen’s linear construction times and E. Hunt partitioning in prefixed suffix trees that brought suffix tree construction to the next level.

3.2.6 DiGeST and an external memory multi-way merge sort (2008)

A simple approach to construct suffix trees is based on an external memory multi-way merge sort. The DiGeST algorithm [12] performs at a speed comparable with TDD and Trellis, and scales for larger inputs since it does not use prefix-based partitioning, but rather outputs a collection of small suffix trees for the different sorted lexicographic intervals.

As in Trellis, DiGeST first partitions the input string into k chunks. The suffixes in each chunk are sorted using any in-memory suffix sorting algorithm. The suffix array for each chunk is written to disk. To each position in this suffix array a short prefix of the suffix is attached. These prefixes significantly improve the performance of the merging phase.

After sorting the suffixes in each chunk, consecutive pieces of each of the k suffix arrays are read from the disk into input buffers. As in the regular multi-way merge sort, a "competition" is run among the top elements of each buffer and the "winning" suffix migrates to an output buffer organized as a suffix tree. When the output buffer is full, it is emptied to disk. In order to determine the order of suffixes from different input chunks, we first compare the prefixes attached to each suffix start position. Only if these prefixes are equal, we compare the rest of the suffixes character-by-character. This comparison requires that the input string be kept in main memory.

Due to the character-by-character comparison of the suffixes, DiGeST runs in $O(N^2)$ internal time. Recall that on average the performance is O(NlogN). The same comparison is performed in order to calculate the longest common prefix of the current suffix with the last suffix previously inserted into the tree. The calculated |LCP| determines the place where the internal node is created, and a new leaf for each suffix is added as a child of this internal node. In this way we build the suffix tree in the output buffer. Before writing the output buffer to disk, the lexicographically largest suffix in this tree is added to a collection of "dividers" which serve locating multiple trees on disk. Since the output buffer is of a pre-calculated size, all trees are of equal size, and thus, the problem of data skew is completely avoided. Further, each tree is small enough to be quickly loaded into the main memory to perform a search or comparative analysis.

While DiGeST still requires the input string to be in main memory, from an external memory point of view, it is very efficient: the algorithm performs only two scans over the disk data and furthermore accesses the disk mainly sequentially. From an internal running time point of view, this algorithm still belongs to the group of brute-force algorithms with a quadratic running time.

3.3 Out-of-core algorithms

Out-of-core category contains three recent methods that support strings larger than the main memory with reasonable efficiency by avoiding random I/Os. These algorithms break the memory barrier and some of them present with parallel implementations that make genome scale input feasible for modern computers to construct the suffix tree.

3.3.1 Big String Big Suffix Trees - BBST (2009)

The main contribution of this paper, presented by Barsky [13], is the first practical algorithm for constructing suffix trees for inputs larger than the size of the main memory.

Barsky's algorithm is an efficient external-memory suffix tree construction algorithm for very large inputs. The algorithm is based on partitioning the input, sorting the suffixes in each partition pair, and efficiently merging the sorted suffixes into a suffix tree. BBST algorithm minimizes random access to the input string, and accesses the disk based data structures sequentially.

BBST scales to much larger inputs than the previous algorithms. This algorithm is able to build a disk-based suffix tree for virtually unlimited size of input strings, thus filling the ever growing gap between the increase of main memory in modern computers and the much faster increase in the size of genomic databases. For example, using the implementation of BBST we build the suffix tree for a DNA sequence of total size of about 12GB in 26 hours on a single machine using only 2GB of main memory.

It is shown that BBST is several times more efficient than the previously proposed algorithms TDD and Trellis with string buffer optimization designed for input strings larger than the main memory. This is because BBST performs sequential access to both the input string and tree being built, whereas the other algorithms cannot avoid a large enough number of random accesses to the input string.

BBST is based on suffix arrays. A suffix array is a vector that contains all suffixes of the input string S sorted in lexicographical order. A longest common prefix array is a vector that stores the length of the common prefix between each two consecutive entries in the suffix array. BBST divides the input string S into several partitions and builds the corresponding suffix array and longest common prefix array for each partition. Then, it merges the suffix arrays of all partitions and generates suffix sub-trees. Note that the tree is constructed in batch at the final phase.

This is an advantage of the algorithm because by avoiding the tree traversal for the insertion of each new node, it is more cache friendly. The time complexity is O(cn), where c = (2n/M) and M is the size of the main memory. If M is comparable to n then c is considered constant and the algorithm performs very well. However, in practice n is expected to be much larger than M; in such a case the complexity becomes $O(n^2)$.

A drawback of BBST is the large size of temporary results. The human genome for example is roughly 2.6G symbols whereas the temporary results are around 343GB. Furthermore, a parallel version of the algorithm would incur high communication cost among the processors during the merging phase.

3.3.2 Wavefront (2009)

The presented algorithm, by Ghoting, called Wavefront [14] is a tree construction algorithm that diverges from the “partition-and-merge” approach to building disk-based suffix trees. Wavefront directly builds the suffix tree in a tiled fashion, allowing us to maintain both: a constant working set size and a fixed memory footprint during tree construction. This property permits large string indexing with high disk I/O efficiency.

It is shown how Wavefront can be extended to build suffix trees on massively parallel systems. The tiled nature of the algorithm, aggressive in-network caching, and effective collective communication are key to realizing an efficient parallelization.

The experimental evaluation of this approach showed that it outperforms the state-of-the-art for disk-based suffix tree construction when the input string does not fit in memory by several orders of magnitude.

WaveFront is the second out-of-core algorithm. In contrast to BBST, which partitions the input string S, WaveFront works with the entire S on independent partitions of the resulting tree T. Tree partitioning is done using variable length S-prefixes, making sure that each sub-tree fits in main memory. Since S may not fit in memory the algorithm may need to read S multiple times. To minimize the I/O cost, WaveFront accesses S strictly in sequential order. Each sub-tree is processed independently without a merging phase, so the algorithm is easily parallelizable. The parallel version has been implemented on an IBM BlueGene/L supercomputer; in absolute time it is the fastest existing method (it can index the human genome in 15 minutes).

Nevertheless, the algorithm cannot scale indefinitely, because more sub-trees increase the so-called tiling overhead. Internally, WaveFront resembles the block nested loop join algorithm and requires two buffers. For optimum performance, these buffers occupy roughly 50% of the available memory, leaving the rest for the sub-tree. This is a drawback of the algorithm, because less memory leads to smaller and more trees that increase the tiling overhead. Moreover, even though the algorithm expands the sub-tree in layers, it needs to traverse the tree top-down for every new node, increasing the CPU cost.

3.3.3 Elastic Range (2012)

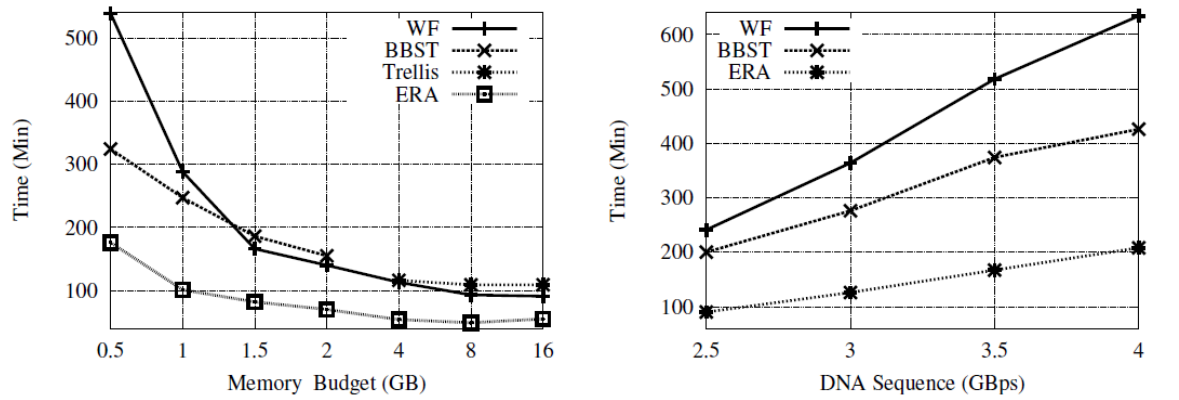
Elastic Range (ERa) [15] is a novel approach that divides the problem vertically and horizontally, presented by E. Mansour. Vertical partitioning splits the tree into sub-trees Tp_1 …Tp_n that fit into the available memory using variable length S-prefixes similarly to Hunt's algorithm and Wavefront.

ERa goes a step further by grouping together sub-trees to share the I/O cost of accessing the input string S. Horizontal partitioning is applied independently in each sub-tree in a top-down fashion. The width of the horizontal partitions is adjusted dynamically (hence the name elastic range) based on how many paths in the sub-tree are still being processed. This allows ERa to use only a small part of the memory for buffers, rendering the algorithm cache-friendly and minimizing the tiling overhead.

Each group represents an independent unit; groups can be processed serially or in parallel. The resulting sub-trees are assembled in the final suffix tree by a trie on the top. The trie is constructed with the S-prefixes used for vertical partitioning and is very small (e.g., the trie for the human genome is in the order of KB).

The construction time for the Human Genome dataset was tested with memory size ranging from 0.5 to 16GB. ERa is consistently twice as fast compared to the best competitor as shown in the plots below, where string size is larger than the memory budget (out-of-core construction). It is worth noting that, while Wavefront is slightly faster than BBST for large memory size, it is dramatically slower when the memory is limited. Note that, the available implementation of BBST does not support large memory.

Since the volume of biologic data increases rapidly; therefore it is essential to have fast suffix tree construction methods. The proposed ERa, is a method that supports very long strings, large alphabets, works efficiently even if memory is very limited and is easily parallelizable. The parallel evaluation was made compared to Wavefront which was the only known algorithm that had a parallel version.



(a) Variable memory; Human Genome (b) Variable string size; DNA; 1GB RAM

Figure 8: Comaparison of the main out-of-core algorithms

Extensive experimental evaluation with very large real datasets revealed that this method is much more efficient than existing ones in terms of speed and computational resources. ERa indexes the entire human genome in 19 minutes on an ordinary 8-core desktop computer with 16GB RAM; and in 8.3min on a low-end cluster with 16 commodity computers with 4GB RAM each. Era is the fastest existing method (i.e., PWaveFront) needs 15min on an IBM Blue-Gene/L supercomputer with 1024 CPUs and 512GB RAM.

3.4. Conclusions

In this chapter we've examined and compared the best algorithms for Suffix Tree Construction throughout the last 20 years. From the first linear, in both time and space, Ukkonen's algorithm to the faster and parallelizable for massive input ERa.

We've examined different techniques applied to address the memory bottleneck that larger input created concerning input partitioning, prefix creation and cache techniques to minimize the overwhelming I/O cost when the resulting tree could not fit the main memory. The latest algorithms (out-of-core) are the only ones that assume input larger than main memory thus creating the need for sequential scanning of the input string.

In table 1 we gathered all common characteristics that we detected among the algorithms in order to present them for deeper understanding of each algorithm's contribution.

Although creating a suffix tree for genome scale input is far from time consuming, like it used to be, with existing algorithms there are still certain aspects that haven't yet been addressed.

A parallel algorithm in a distributed system that scales has not yet been proposed. Even the extremely fast ERa doesn't scale for many processors. Since it doesn't use input partitioning but rather tree partitioning every node must have access to the entire string in a shared-nothing architecture. This is the main drawback for this implementation.

The main challenge yet to overcome is a parallel algorithm for a shared nothing architecture that scales for increasing nodes and would make Suffix Tree construction for huge amount of data an easy process and would enable all those who benefit from Suffix Trees to take their work to the next level.

	In-memory	**Semi-Disk-Based**					**Out-of-core**	
Criteria	**Ukkonen**	**Hunt**	**TDD**	**Trellis**	**Digest**	**BBST**	**Wavefront**	**ERa**
Complexity	O(n)	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Construction Algorithm	Ukkonen	Brute force	Wotdeiger	Ukkonen	Suffix Arrays	Suffix Arrays	Brute Force	Brute Force
Memory locality	Poor	Good	Good	Good	Good	Good	Good	Good
String Access	Random	Rand	Rand	Rand	Rand	Seq	Seq	Seq
Input Partitioning	No	No	Yes	Yes	Yes	Yes	No	No
Prefix Creation	No	Fixed-Length	Fixed-Length	Variable-Length	Per Partition	Per Partition	Variable - Length	Variable - Length
Parallel	No	No	No	No	No	No	Yes	Yes
Speed for Large Input	Very Slow	Med	Fast	Fast	Fast	Very Fast	Very Fast	Very Fast

Table 1 : Comparison of the main algorithms

Chapter 4 - Hadoop Map-Reduce

In this chapter we will present the Map-Reduce platform [16] which allows us to implement parallel algorithms easily and efficiently.

4.1 The Hadoop platform

The Apache Hadoop Map-Reduce is a framework written in Java which supports applications of parallel processing of large amount of data. The idea was proposed originally by Google in 2004, but its implementation was not available. The Hadoop is an aggregate of applications which are executed in inter-connected computers through a cluster. Hadoop offers a level of abstraction in the process of developments parallel programs, by assuming to manage data sharing, result concentration, possible node failures and other issues that usually the program would have to solve. A simple implementation in Java, using Hadoop libraries according to the map-reduce concept, suffices to fully exploit the parallel execution that Hadoop offers.

Hadoop internal structure consists of the following parts: 1) Hadoop Common: The service that gives access to the filesystems supported by Hadoop; 2) Hadoop HDFS: The Hadoop filesystem; 3) Hadoop Map-Reduce: Hadoop engine that assumes to execute the program in a parallel fashion.

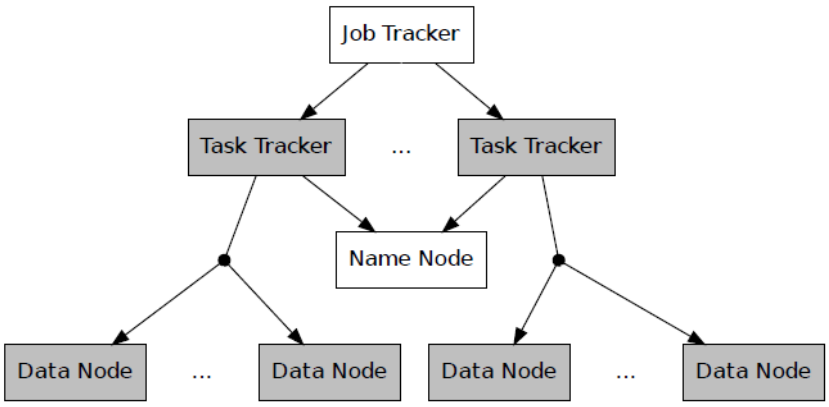


Figure 9: Hadoop architecture, With white are highlighted the master processes and with gray the slave processes.

4.2 Hadoop HDFS

The Hadoop filesystem is HDFS. It runs over the operational system's filesystem and is designed according to the master/slave architecture. The purpose of HDFS is to store large amount of data in a large scale cluster. Further it is tolerant to hardware failures. Other characteristics that make HDFS appealing are the preservation of file metadata (user ownership, permissions), the recording of adjacent nodes, in order to accelerate data traffic, and the rebalancer mechanism that distributes equally data between Datanodes in the cluster.

Hadoop Process Layer: Data are stored in Datanodes (slaves), divided in chunks called blocks. Blocks exist in multiple copies, the number of which is determined by the data replication factor. The Hadoop project suggest for the existence of at least three copies for each block, allowing the system to be extra tolerant to failures either hardware (broken hard-drives) or network (node disconnection). This factor allows the system to detach from solutions such as RAID for data storage, making the cluster construction a cheap solution. HDFS demands the existence of one Namenode (master). The Namenode administrates the filesystem. It keeps record for the availability of blocks in the Datanodes, as well as file metadata. If a Namenode fails, the whole cluster will fail and will stop execution but the last valid state is stored by the Secondary Namenode mechanism. During restart, the Namenode will read the log stored in the Secondary Namenode and will regain the last valid state without loss of information.

4.3 Hadoop Map-Reduce

Map-Reduce is the system that controls the program's parallel execution that is submitted by the programmer to the cluster. It is also designed in master/slave architecture. It is charged with the responsibility of data sharing from the HDFS to the Datanodes for processing , grouping and redistribution of intermediate data and the collection of the end results. It also manages Datanode failures for the reassurance of the success of the whole program.

Hadoop process layer: The Jobtracker (master) is responsible for the organization of job execution submitted by the programmer. Its purpose is to break the whole task in many tasks and assign them to the Tasktrackers (slaves). In this process it takes into account network topology, in order to minimize data traffic, by forwarding data to more adjacent nodes. For the task assignment, the Jobtracker communicates with the Namenodes to find the input data exact location of the task. This technique is usually called as “it is cheaper to assign the process to where the data really are, instead of transferring them”. In order to determine the progress of data processing the Jobtracker receives pulses from the Tasktrackers regularly. In case a pulse is not received by a Tasktracker Jobtracker marks the task as failed and assigns it to another node. The failures are treated as normal facts and the only impact to the while process is the delay caused by the need of reassigning the data. Further, if a task is progressing slower than the other tasks the same task is assigned to a second Tasktracker. The end result comes from the first task that finishes and the other execution is killed and the results are deleted. The aforementioned mechanism is called speculative execution and is actually a method of balancing different performance caused either by differences in the hardware or by input complexity and is so avoided the scenario of a single task stalling the whole process where all other task have successfully finished. It becomes obvious that the Tasktrackers undertake to run the tasks in which the job has been split, they inform the Jobtracker for the success or failure of the task execution and regularly send pulses claiming their proper function. The process level is illustrated in figure 7.

4.4 Computational model

As mentioned above, the programmer is called to implement the algorithm to be parallelized with the concept of two functions, the map and reduce. The general concept is the map processes (key, value) pairs and generates a similar output. The results are grouped with the key and are given as input to the reduce function, in a way that in each call of the reduce function the input will be of the form (key, list of values). The reduce output is the end result of the whole process.

As aforementioned map and reduce functions which the programmer is called to implement are both structured to receive input and send output in the form of key and values. The map function receives pairs of data and outputs a list of pairs.

Map :: (key1, value1) -> list (key2, value2)

It is implemented in a parallel fashion for every chunk the input string has been divided, producing a list of pairs in every call of the map function. Then, the Map-Reduce collects all pairs from all the lists, groups and sorts them in order for a group of keys to be send to the same reducer. This group is input for the reduce function, which will run in different nodes. As we can see from the type annotation, there is no restriction for the key to be the same type as the value neither in the input nor in the output. The reduce function receives a pair which is consisted of the key and a list of values and returns a list of values.

Reduce :: (key2, list(vlaue2)) -> list (value3)

The output of all the reduce calls are grouped as the end result of the Map-Reduce. Finally the map-Reduce has received a list (key, value) and outputs the result as a list of values. The reduce function, like the map, puts no restrictions concerning the types of input or output. Optionally, can use a third function called the combine, which is a reduce like function that runs locally in the node that finished the map task. Its call proceeds the reduce call and data delivery through the network while the results are still in memory. With the combine function a portion of the reduce call has been done before without the data to be send through the network to the nodes that will execute the reduce function, achieving in that manner faster processing of the intermediate data. The computational model is proven to be Turing Complete and every problem that has a computation solution can be adjusted to the Map-Reduce model.

4.5 Data flow layer

We can now examine the total data flow during a Map-Reduce program, by grouping the nodes of our network with map and reduce functions. The main stages of the data flow layer are described above and illustrated in figure 10.

Input Reader assumes to divide the input in appropriate sized chunks called splits which are typically between 16 and 128 MB. Every split is assigned to one and only Mapper. The Input Reader has input data stored in the HDFS and produces output (key, value) pairs. Internally this stage is implemented in two steps, the InputFormat that divides the input into splits and the RecordReader that divides the split in (key, value) pairs.

Map function. Every node assumes a number of Mappers, typically as many as the node's cores. Each Mapper has a unique split as input, which is divided in (key, value) pairs. Each pair is processed with one call of the map function in the same Mapper. The results are stored sorted according to the key value.

Partition function. The output of each map function is assigned to the appropriate Reducer through the partition function. The partition function receives as input the key value along with the number of the available Reducers and returns the id of the Reducer this key will be assigned to.

Sorting Comparison function. The input for every reduce function is transferred from the node that executed the map function and sorted according to the sorting comparison function.

Grouping Comparison function. This function decides the input of a single call of the reduce function. The three aforementioned functions all represent the Shuffle phase, and is responsible to deliver appropriately the output of the Map function as input of the Reduce function.

Reduce function. Each node of the cluster assumes a number of Reducers, typically as many as the nodes cores. Each Reducer has input a list of (key, list(values)). The reduce function is called once for every distinct key resulting from the Shuffle phase, accesses all values related with this key and returns a list of values.

Output Writer. The output writer is responsible to store the output of the reduce phase in HDFS.

Additionally to the aforementioned main functions there are two additional functions complementary to both Map and Reduce phase. The **Setup function** which can be executed before the Map or Reduce function and is used optionally if preparation is necessary before the arrival of the input to the Map or Reduce function and is executed locally once per Mapper or Reducer just before the map or reduce function respectively.

The **Cleanup function** has the same functionality but is executed after the map or reduce function is completed for all their respective input and is useful if processing, of the output of the map or reduce function, is needed.

Explanation is needed when referred to Map phase or Mapper or Map function. The Map phase is the stage of the Map-Reduce in which all nodes execute in parallel their map functions. Each node may execute a number of different Mappers, with each Mapper assuming a unique split of input, dividing the input in (key, value) pairs and calling the map function for every such pair. The same occurs for the Reduce phase as well. These two phases are executed serially among each other. The Map phase is executed in parallel and when it ends the Reduce phase is executed in parallel as well. After the Map phase and before the Reduce phase the intermediate Shuffle phase is executed responsible for data manipulation and distribution from the output of the Map phase to the input of the Reduce. The data flow of a Map-Reduce task is illustrated in figure 10.

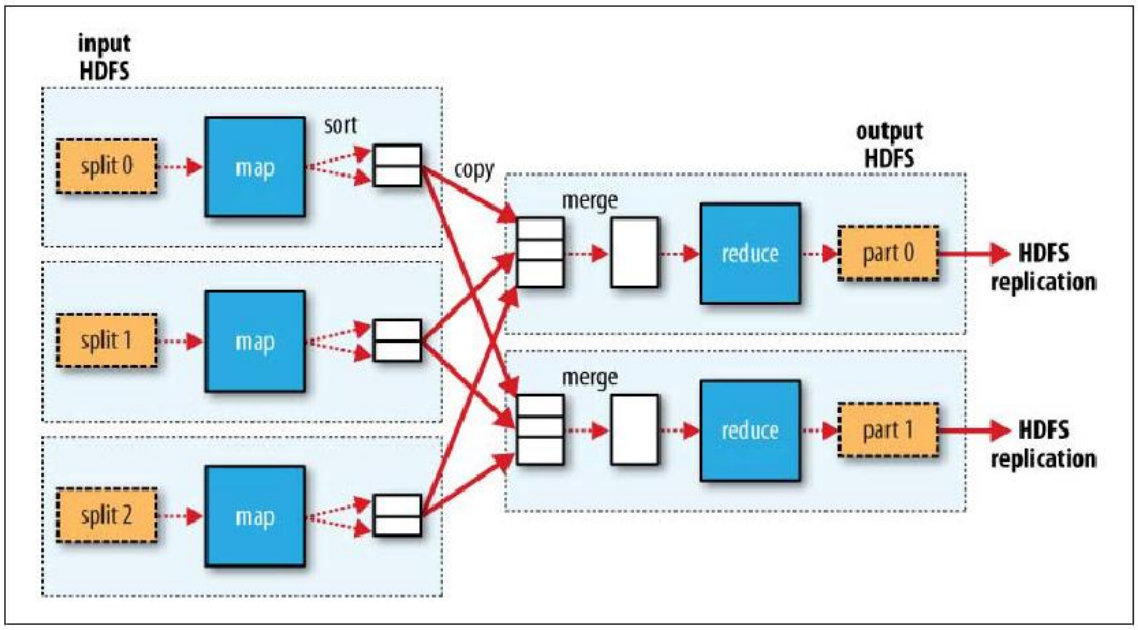


Figure 10: Map-Reduce Data Flow for multiple maps and multiple reducers.

Chapter 5 - Map Reduce ST implementation

In this chapter we will discuss thoroughly the algorithms implemented for all the phases of the suffix tree creation and how they were adjusted in the Hadoop Map-Reduce environment to benefit from its parallel features.

5.1 Layout

Intuitively we can split the problem in 2 main phases; the prefix creation phase and the suffix tree creation phase.

The prefix creation phase originally proposed by E. Hunt in [7] is a method of dividing the suffix tree in a forest of suffix trees each representing a specific prefix. Since each tree derives from all prefixes in a way that all possible suffixes are included and the suffix tree maintains its powers. Hunt in [7] proposed fixed length prefixes which resulted in many uneven trees in a way that the system resources were underused in some cases and used fully in others as discussed in Chapter 3. It was Trellis in [11] the first algorithm that introduced variable-length prefixes as a way to divide the suffix tree to more even prefixed suffix trees and it was this idea that we applied in our algorithm.

The suffix tree creation phase is a mix of many ideas. This phase can be subcategorized in the suffix acquisition phase where all possible suffixes are found for each predetermined prefix and the prefixed suffix tree creation phase where all suffixes are inserted for every prefix separately. The prefixed suffix tree creation phase uses a brute force algorithm of $O(n^2)$ complexity proposed also in E. Hunt in [7] which will be described thoroughly below.

These two phases can also be described as two separate Map-Reduce programs where the output of the one gives necessary information to the other. The one Map-Reduce job finds all the variable-length prefixes that meet a user-defined threshold, that has to do with the maximum available memory assigned for the tree, as well as the frequency of each prefix and then creates a suffix over-tree for all the prefixes. The second Map-Reduce job finds all occurrences of every prefix, forwards all the positions of the input string and then creates all possible prefixed suffix trees.

5.1.1 Suffix tree observations

In this section we will discuss some heuristics and observations that have been proposed in previous work and apply for every suffix tree regardless of data or construction algorithm since they are inherent to its capabilities.

Once a leaf always a leaf [6]. Once a leaf is created in a suffix tree and attached at the end of an edge that leaf will remain on that edge and the path leading to it will not change no matter what suffix will be added in later stages of the construction algorithm. The crucial part is the correct insertion of leaves in a way that the path leading to that leaf is not affected.

The start position of an edge doesn't change [15]. Creating a new edge means adding a suffix in the suffix tree since edges are mainly created to add leaf nodes (suffixes) in the tree. Edges that exist internally that connect internal nodes of the tree are edges originally created for entering leaf nodes and got split along the way. The start position though of an edge cannot change since it marks a specific path towards the original leaf and thus remains the same until the end.

The insertion sequence of suffixes doesn't affect the resulting suffix tree. As a consequence of the aforementioned observations the sequence in which suffixes will be added in the suffix tree cannot change the tree. Edges might have different start positions and end position in different inserting scenario but they will definitely mark the same path.

The division of the suffix tree in prefixed suffix sub-trees can be done without any loss of information [7]. Whether you create a suffix tree and then split it in prefixed suffix trees or whether you create a suffix tree based on prefixes the resulting tree will be the same. Since the insertion sequence doesn't affect the resulting suffix tree the insertion of suffixes that start with the same prefix will be the same suffix tree as if created as a whole and then split in prefixed suffix trees.

With those observations and heuristics we are equipped to use and handle a suffix tree creation algorithm that can successfully build a compact suffix tree that holds the information of all suffixes for a given input string. The use of the above observations and heuristics will become clearer in the following sections in the description of the used algorithms.

5.2 Prefix creation phase

As mentioned before the idea of variable-length prefixes was introduced by B. Phoophakdee and M. Zaki in [11] and proposed a simple way to find all possible prefixes. They proposed to examine all prefixes with length 4-8 and secondly, if needed, with length 9-16. All prefixes that meet the threshold are kept and those surpassing the threshold are searched again (in the next string searching, through the entire input string) by creating four new prefixes by appending all four letters of DNA alphabet in the end of the original nonconforming prefix. If there is a prefix exceeding the threshold with length of 8 the second phase is necessary otherwise for big thresholds is not used. The drawback of this implementation is that they have to scan the entire input string for every different length of the prefixes. Starting from length 4 they will need 5 scans of the input string if the maximum prefix length is 8.

We applied a smarter way to find all possible variable-length prefixes that requires a single run on the input string and can be applied in a parallel fashion. The idea is quite simple; we scan the entire input string reading maxLength characters at a time and store in a HashMap all different prefixes from minLength size up to maxLength size. If they are not already in our HashMap we insert the prefixes with total occurrences of one, otherwise we add one to the prefix's occurrences. This process moves one character at a time on the input string in order to find all occurrences and all prefixes possible as depicted in algorithm PrefixSearching in 5.2.1. MaxLength is set to 8 at first and minLength is set to 4. In the end we look all prefixes starting from minLength and those not exceeding the threshold are kept and those exceeding the threshold are appended with all 4 letters of our alphabet and checked in the HashMap for their occurrences that have already been found, as depicted in algorithm PrefixDetermining in 5.2.2. If there is a prefix of length 8 surpassing the threshold another scan is required with minLength 9 and maxLength.

This simple algorithm is easily parallelized since it can be applied separately to different chunks of the input string as long as there is a tail of maxLength-1 in the end of each chunk overlapping to the start of the next chunk. We will now present in pseudo code the algorithms described above as well as its Map-Reduce implementation.

5.2.1 Algorithm PrefixSearching

Input:: String input, inputLength, prefix minLength, prefix maxLength

Output:: HashMap with all existing prefixes from size minLength to maxLength and each prefix's occurrence

```
1. for i=0 to (inputLength – maxLength) with step 1 character
2.       string = input.readCharacters(i, i+maxLength)
3.       for length = minLength to maxLength with step 1 character
4.                prefix = string.substring(0, length)
5.                if (prefix exists in HashMap)
6.                         HashMap.get(prefix)+=1
7.                else
8.                         HashMap.add(prefix, 1)
9.                end if
10.      end for
11. end for
```

With one simple pass over the input string we are able to find all existent prefixes in the input string and calculate the number of occurrences of each prefix. The next step is to determine which of the existing prefixes regardless of their length meet the threshold the user has defined. The output of this algorithm is a HashMap which contains all existing prefixes with length from minLength up to maxLength and the occurrence number of each prefix. This HashMap is the input for our next algorithm.

5.2.2 Algorithm PrefixDetermining

This algorithm takes the output of the PrefixSearching algorithm along with the threshold and the minLength and maxLength variables and produces 2 new HashMaps, one with the conforming prefixes and one with the nonconforming prefixes. If the the HashMap with the nonconforming suffixes is empty no other action is necessary since the HashMap of the conforming prefixes includes all the prefixes needed to depict all the suffixes in the input string.

Input:: HashMap(prefix, occurrence) of all prefixes, minLength, maxLength, threshold

Output:: HashMap(prefix, occurrence) of conforming prefixes, HashMap(prefix, occurence) of nonconforming prefixes

```
1. for all prefixes of length=minLength
2.      occ = allHashMap.get(prefix)
3.      if(occ<=threshold)
4.          confHashMap.add(prefix, occ)
5.      else if( occ>threshold)
6.          while(length<maxLenght)
7.              for letter = {A,C,G,T}
8.                  newprefix = prefix.append(letter)
9.                  occ = allHashMap.get(newprefix)
10.                 if(occ<=threshold)
11.                     confHashMap.add(newprefix, occ)
12.                     break
13.                 else
14.                     if length<maxLength
15.                         length+=1
16.                     else if(length = maxLength)
17.                         nonConfHashMap.add(newprefix, occ)
18.                     end if
19.                 end if
20.             end for
21.         end while
22.     end if
23. end for
```

If though the nonconforming HashMap has at least one prefix we must execute from the beginning both the PrefixSearching algorithm to find the occurrences of all prefixes excceding threshold and maxLength and passed again through the algorithm PrefixDetermining to find the remaining prefixes that exceed the original maxLength and add them to the conforming HashMap from the previous phase.

5.2.3 Map-Reduce Implementation of the Prefix Creation Phase

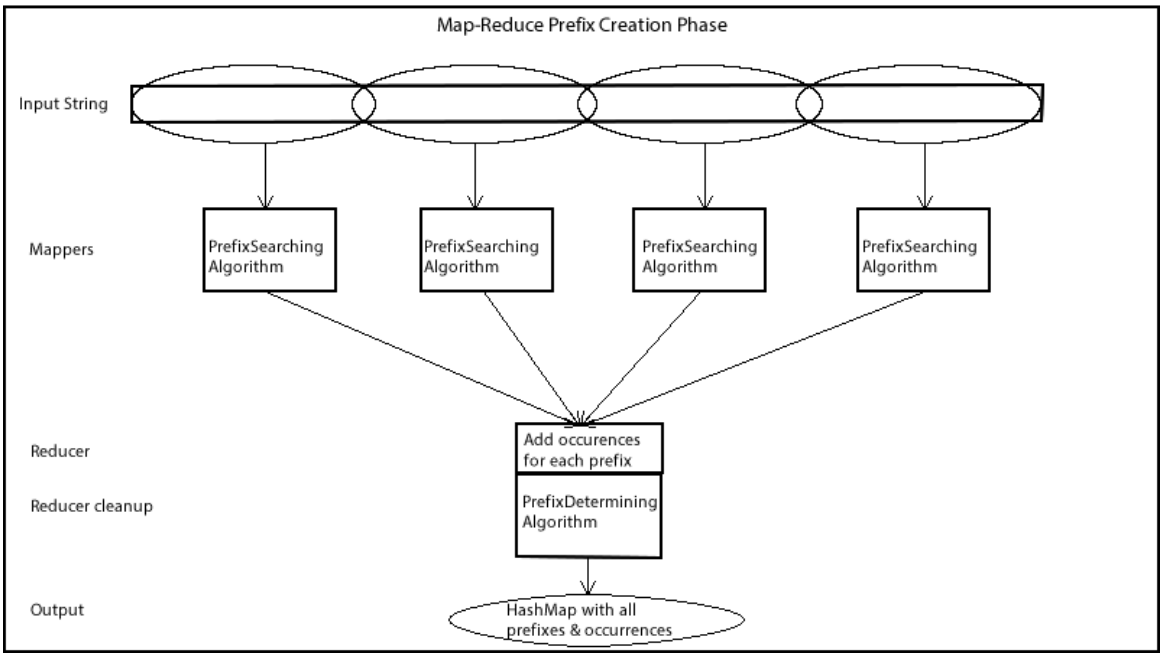


Figure 11: Map-Reduce Prefix Creation Phase

Putting together all the pieces and implementing the necessary map and reduce functions, as discussed in chapter 4, we created a parallel program able to find all variable-length prefixes bounded by user defined threshold as illustrated in figure 9.

To gain a better perspective of how the program really works several issues must be clarified. Starting from the top down in figure 11, it must be clear that the input string must be divided into overlapping chunks in order to find all possible occurrences of every prefix. The overlap is maxLength – 1 characters and ensures that the last substring will check for all respective prefixes from minLength up to maxLength and the next will be checked by the next mapper who will receive the next split.

Each mapper is assigned with a single split and by executing the PrefixSearching algorithms finds all occurrences for every prefix. Consequently it outputs the prefixes with their occurrence number in the form of (prefix, occurrence) pairs. The number of the Mappers is determined by dividing the input length with 64MB which is the block size in Hadoop HDFS. For example a 2Gbps input will be divided in 32 chunks and has as many mappers.

The output of the Mappers is sorted in the shuffle phase and gives the Reducer as input (prefix, list(occurrence)) where prefix is the key and the number of occurrences

is the value. The first task for the reducer is for every call of the reduce function to add the occurrences found by different Mappers for the same prefix and create a single HashMap with prefixes and their occurrences. After the completion of all reduce calls, the cleanup function of the reducer is called and uses the HashMap as input for the PrefixDetermining algorithm to determine which prefixes comply with the threshold.

The output is written to HDFS for the next Map-Reduce job to read and process in a simple file which contains all prefixes and their occurrence in the input string. If there are nonconforming prefixes they are written to a different file and the same Map-Reduce is launched to find the remaining prefixes that exceed MaxLength and threshold and finally appended to the same file that holds the rest of the conforming prefixes.

Another significant contribution of this Map-Reduce job is that after determining the prefixes that are able to depict every suffix in the input string, we are able to build a suffix tree with those prefixes which will be used as the root tree on top of the forest of prefixed suffix trees. In the query phase, where a suffix is searched, the traversal will start from this root suffix tree, and the end will mark the prefix that this suffix bears and consequently continue the search in the file containing the suffix tree for that prefix. The algorithm for the construction of the root suffix tree for the prefixes will not be presented in this section since the suffix tree creation algorithm is the same and will be discussed thoroughly in the following section. The root suffix tree will be written also in the HDFS but its use is limited to the query phase alone.

Overall this job is similar to the famous word count problem solved easily by Map-Reduce, from which this job was actually inspired, but instead of words there are prefixes and instead of documents there is a single file of DNA sequence. This phase has O(n) complexity since a single run over the input string is sufficient for the determination of all the prefixes in most cases and in the worst case, for very small threshold, a second run is necessary. For 2GB of input and threshold higher than $3*10^6$ all prefixes are found between minLength=4 and maxLength=8, whereas for smaller input less minLength might be necessary.

The results of this Map-Reduce job will be presented extensively in Chapter 6 where the performance of this job will be inspected and conclusions will be drawn concerning the threshold and the number of resulting prefixes for various lengths of DNA input.

5.3 Suffix tree creation phase

This phase can be divided into two main basic algorithms. The first is responsible for retrieving all occurrences of all prefixes in the input string and the second for inserting these occurrences into a prefixed suffix tree. In the end of the suffix tree creation phase a forest of prefixed suffix trees will have been produced. The main idea for this phase was originally proposed be E. Hunt in [7], but in this algorithm a prefixed suffix tree was created in every scan of the input string needing at least that many scans, of the input string, as the number of distinct prefixes found in a serial fashion. It was the possibilities of Hadoop Map-Reduce that inspired the division of this phase into two distinct parallel sub-phases where the output of the first is the input of the second.

The first sub-phase is not very different than the prefix creation phase. The main difference derives from the fact that the set of prefixes is already found in the previous phase and we scan the input string to find the exact positions of every suffix starting with a specific prefix. The outcome of this sub-phase is all suffixes for every prefix that appear in the input string. It is easy to deduce that this phase is parallelizable in the same manner as the previous phase.

The second sub-phase is responsible for creating the prefixed suffix trees. Having found all occurrences of every suffix a simple insertion in the suffix tree is needed by traversing the tree from the root to a splitting point. Another aspect that must be mentioned is the fact that all the suffixes for a specific prefix are pre-sorted by suffix position in the input string in ascending order, in an effort to achieve sequential read both in memory and disk, rather than random access.

The construction algorithm we used is a simple $O(n^2)$ brute force algorithm used in [7], [14] and [15]. The main idea is that for every new suffix a traversal of the tree begins from the root until there is a mismatch in the character comparison. In that exact place we will insert the new suffix by creating an internal node with a single edge to the new leaf (suffix) in the case the mismatch is on an edge, or by creating a new edge leading to a leaf (suffix) in the case the mismatch is on a node. The main challenge in each algorithm is how it allocates the input string in terms of memory management and what complementary data structures are used to facilitate a faster insertion.

5.3.1 Algorithm SuffixAcquisition

Input:: HashMap(prefixes, occurences), String input, inputLength, minPref, maxPref

Output:: list of(prefix, list of(positions))

```
1. for i=0 to (inputLength – maxLength) with step 1 character
2.      string = input.readCharacters(i, i+maxLength)
3.      for length = minLength to maxLength with step 1 character
4.              prefix = string.substring(0, length)
5.              if (prefix exists in HashMap)
6.                      list(prefix).add(i)
7.                      break
8.              end if
9.      end for
10. end for
```

The similarities of this algorithm to the PrefixSearching algorithm are obvious with the only basic difference that the prefixes have been obtained from the previous phase and we are interested in finding the starting prefix of our examining suffix. Once it's found we proceed to the next suffix. A simple heuristic to minimize the comparisons is to keep from the previous phase the minimum length of all found prefixes. In that way if, for example, prefixes range from 5-7 characters in length we will make two less comparison per suffix in this phase.

The complexity of this algorithm is O(N) since a single scan on the input string is sufficient for the extraction of all the suffix positions for every prefix. The resulting number of suffixes must be equal to the length of the input string with the loss of a few (less than maxLength) suffixes in the end of the input string that is not possible to put in any prefix since their length will be less than any existing prefix. The inability of inserting the last suffixes can be only be overridden if there are additional characters at the end of the input string enough to form the last suffixes' prefixes. This loss is very small and is inherent to the prefixed suffix tree (typically for 2Gbps is $1,4*10^{-7}$%) and is not a threat to the suffix tree's validity.

5.3.2 Algorithm SuffixTreeCreate

Input:: listof(prefix, listof(suffixPosition)), String Input

Output:: prefixed suffix-trees

1. for every prefix
2. for every suffixPosition of prefix
3. SuffixTreeAddSuffix (suffixPosition)
4. end for
5. end for

The above algorithm takes the suffixes' positions and adds them to the tree using the algorithm SuffixTreeAddSuffix explained in 5.3.3.

5.3.3 Algorithm SuffixTreeAddSuffix

This simple algorithm originally proposed be E. Hunt in [7] is a simple brute-force algorithm of $O(n^2)$ complexity. This algorithm simply takes the suffix position of a suffix for a certain prefix and inserts it in the prefixed suffix tree. The main operations for adding the suffix are addNode, addLeaf and splitEdge as illustrated in figure 12. The algorithm for the suffix insertion is given in pseudo-code below.

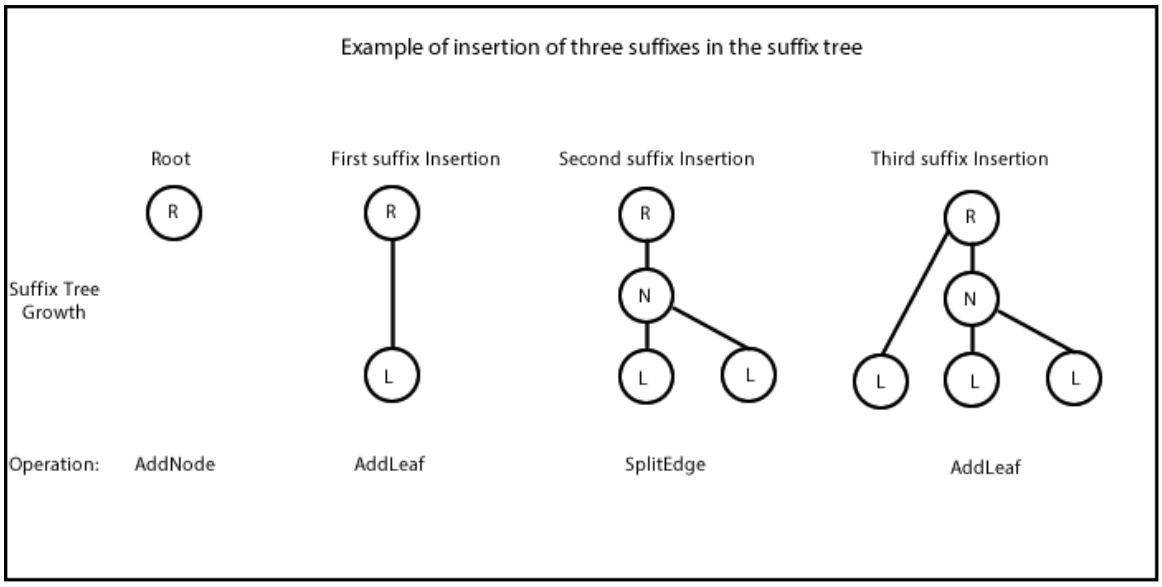


Figure 12: Example of main operations during suffix tree construction

Input:: suffixPosition

Output:: suffix-tree with suffixPosition

```
1. i=suffixPosition + prefix.length
2. node=root
3. edge = edgeStartingFrom(input.charAt(i))
4. while(true)
5.      i++
6.      treeIndex++
7.      if(input.charAt(i) == input.charAt(treeIndex) )
8.              if(treeIndex==edge.endIndex)
9.                      node = edge.endNode
10.                     edge = edgeStartingFrom(i+1)
11.                     if(!edge.exists)
12.                             AddLeaf(node, prefixPosition);
13.                             break
14.                     else
15.                             treeIndex++
16.                             i++
17.                     end if
18.             end if
19.     else
20.             SplitEdge(node, edge, treeIndex, prefixPosition)
21.             break
22.     end if
23. end while
```

The algorithm traverses the tree comparing the tree edges with the suffix until a mismatch occurs and then either an edge is spliited or a leaf added before terminatiion. Its mean complexity is O(nlogn) and the reason is that suffix trees, for real DNA data, is very dense and the maximum common path is several orders of magnitude smaller than the length of the input string. For a 2Gbp input the maximum common path for a prefixed suffix tree (aka maximum tree depth) is about 65000 characters.

5.3.4 AddLeaf and SplitEdge operations

AddLeaf operation

Input:: node, inputIndex, suffixPosition

1. node.addEdge(inputIndex)
2. leaf=newNode(prefixPosition)
3. edge.setEndNode(leaf)
4. edge.setEndIndex(inputLength)

SplitEdge operation

Input:: node, edge, treeIndex,inputIndex, prefixPosition

1. internalNode = newNode()
2. extendedEdge = internalNode.addEdge(treeIndex+1)
3. extendedEdge.setEndNode(edge.EndNode)
4. extendedEdge.setEndIndex(edge.EndIndex)
5. edge.changeEndIndex(treeIndex)
6. edge.changeEndNode(internalNode)
7. addLeaf(internalNode, inputIndex, suffixPosition)

The addLeaf operation is just creating a new edge and a new leaf node. The leaf node must bear the suffixPosition. The edge leading to the leaf must have the startIndex which is the inputIndex, the position in the input string that differs from the tree traversal point the comparisons led, the endIndex which is the size of the input string for edges leading to leaves and the endNode which is the leafNode.

The splitEdge operation must split an edge by inserting an internal node. The old edge must change endIndex and endNode. The new internal node must add two edges the extension of the old edge and must copy its endIndex and endNode and the new edge that will lead to the leaf by executing the addLeaf operation.

Several other operations such as setEndNode, setEndIndex , changeEndNode, changeEndIndex emitted from further explanation since their implementation is rather straightforward and they just change information concerning pre-existing nodes and edges.

5.3.5 Map-Reduce implementation of the suffix tree creation phase

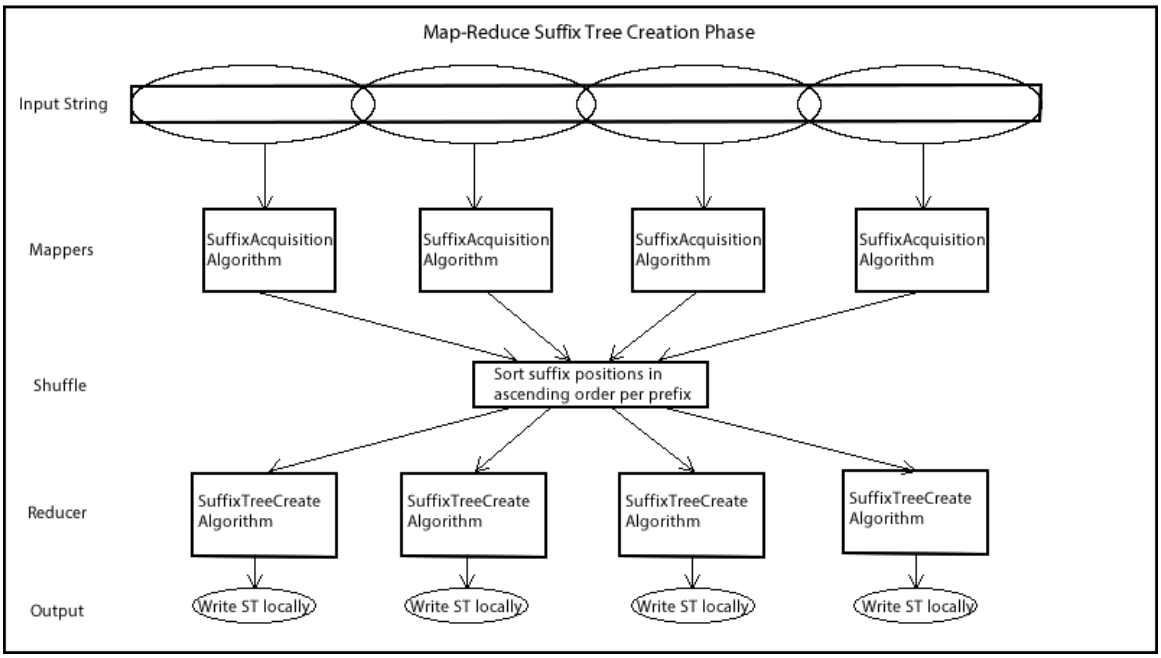


Figure 13: Map-Reduce Suffix Tree Creation Phase

Adding all the aforementioned pieces in a map-reduce fashion we created a parallel program that can successfully create suffix trees for given prefixes and input string. The suffix tree creation phase adjusted to the Map-Reduce requirements is illustrated in figure 13.

As in the prefix creation phase the input string is divided in chunks with overlapping maxLength-1 characters in order to find every suffix. The Mappers receive a split from the input string and the HashMap of selected prefixes and scan their split to find all suffixes for all the different prefixes. At the end of the map function pairs in the form of (prefix, listof(suffixPositions)) are outputted in the HDFS. The Mappers output is sorted in the shuffle phase and for every prefix sorted lists are created containing suffix positions from all the splits in ascending order.

The reducers receive pairs in the form of (prefix, sorted listof(suffixPositions)) and use the SuffixTreeCreate algorithm, as explained in the previous sections, and create a suffix tree for every prefix and store them in the hard disk locally. The reason that the trees are not written in the HDFS is that, as explained before, the Suffix Trees are up to 56 times the size of the input string. For example for 2GB of input string 112GB will be stored in the HDFS.

During the experimental phase it was concluded that flushing the trees in the HDFS created a time overhead that made the suffix tree construction inefficient because of multiple collisions and slow transfer speeds. Another fact that must be clarified is that when the reducers used the input string, to insert suffixes, the read was also done locally having placed the input string in each node's hard drive for the same reasons. The way that the reducers read the input string will be explained separately in the input cache policy section.

After the completion of this phase we can transfer all the resulting suffix trees in the master node with a simple script in the same way that we transfer from the master node to all slave nodes the input string to be available for the reduce function and the suffixTreeCreate algorithm. The time consumption for the transfer of these data is not measured and is not taken into consideration in the experimental study.

The number of the Mappers used in this phase was typically one for every 64MB of input string since Hadoop's block size is 64MB. For 2GB of input string 32 Mappers would be created regardless of the size of the cluster and the available tasks. The number of the reducers is a different story. After extensive experiments it was made clear that if the number of tasks exceeded the number of the available nodes the deterioration of the program's efficiency was dramatic due to hard drive collisions and multiple and simultaneous read and write requests. We could generally say that the number of reducers in our program is bound by the cluster's machines.

We can see the resemblance between the map phase of this map-reduce job with the prefix creation job. This implementation was also word-count like and was inspired by Hadoop Map-Reduces possibilities. The complexity of the map phase is O(n) since a single scan is exactly needed for all suffix positions to be found. The sorting in the shuffle phase is done automatically by Hadoop. The reduce phase has complexity $O(n^2)$ since the brute force algorithm, proposed in [7], was used. The efficiency and scalability of this Map-Reduce job will be thoroughly discussed in Chapter 6 with the experimental results.

What still remains unanswered is the way the program reads the input string and the tree's edges (which are start and end positions in the input string of a substring). This subject will be thoroughly discussed in the next section.

5.3.6 Input cache policy

In order to facilitate the insertion of suffixes and keep in memory chunks of the input string needed for the comparison of suffix characters with the already inserted suffixes in the tree a three way input cache policy is applied. A 2-d array of user-defined cache buffers are kept in memory. Each buffer has 64 MB capacity and the minimum number for our policy to work is 3 buffers.

The first group of buffers, also referred to as stable buffers, contains buffers that once a split from the input string is inserted are kept in memory until the completion of our program. This group of buffers keeps the first k splits of the input string, since those splits would have first inserted the suffixes in the tree, thus most start and end positions of edges higher in the tree would be from those splits. The idea was first introduced in TOP-Q algorithm in [10] and is proven both in their experiments and in ours that the higher the nodes in the tree the bigger the possibility to be requested during an insertion.

The second group of buffers, also referred to as mobile buffers, contains buffers that are used during a complementary insertion phase of suffixes unable to be inserted with the existing in memory buffers. This complementary insertion phase is needed since during an insertion all previous splits that have been inserted might be requested while traversing the tree to find the insertion point. If the split is not available, instead of requesting the split and paying an extra I/O, we store all information at the time of the failed split request in arrays. Typically the information is the node and the edge we are on, the offset of the suffix in the input string and the suffix's starting position. With four 2-d arrays of integers we may group all insertions needing a specific split and at the end of the insertion of all suffixes belonging to the inserting split we can use the mobile buffers to load in memory the requested splits sequentially.

The third group of buffers, also referred to as input split buffers, contains buffers that are used for accessing a split when inserting suffixes that belong to that particular split and is also mobile but changes only when the inserting suffix belongs to the next split and remains in memory until all suffixes from the particular split have been added to the tree.

Typically, both the second and the third group of buffers need a single split buffer in order to function and by increasing the number of buffers in these groups no improvement will be seen. The number of buffers that makes a big difference in the execution of our program is in the first group, the stable buffers. The more the stable buffers are the faster the execution will be and the less I/O required by the entire suffix tree creation phase. Although the policy will work with 3 buffers, one for each group, the I/O cost will decrease linearly while increasing the buffers.

The total number of I/O's can be calculated with the above mathematical type where maxBuffers is the maximum number of buffers used for input cache policy and totalSplits is the number of splits that our input string is divided.

For reasons of clarification the split and the buffer is of the same size (typically 64MB) since the buffers store splits of the input string. Also the mobile buffer, after the completion of the complementary insertion phase, of unsuccessful suffixes, is set to the next uninserted stable split posing as stable buffer until the next complementary phase. An example of the buffers' state is illustrated in figure 14.

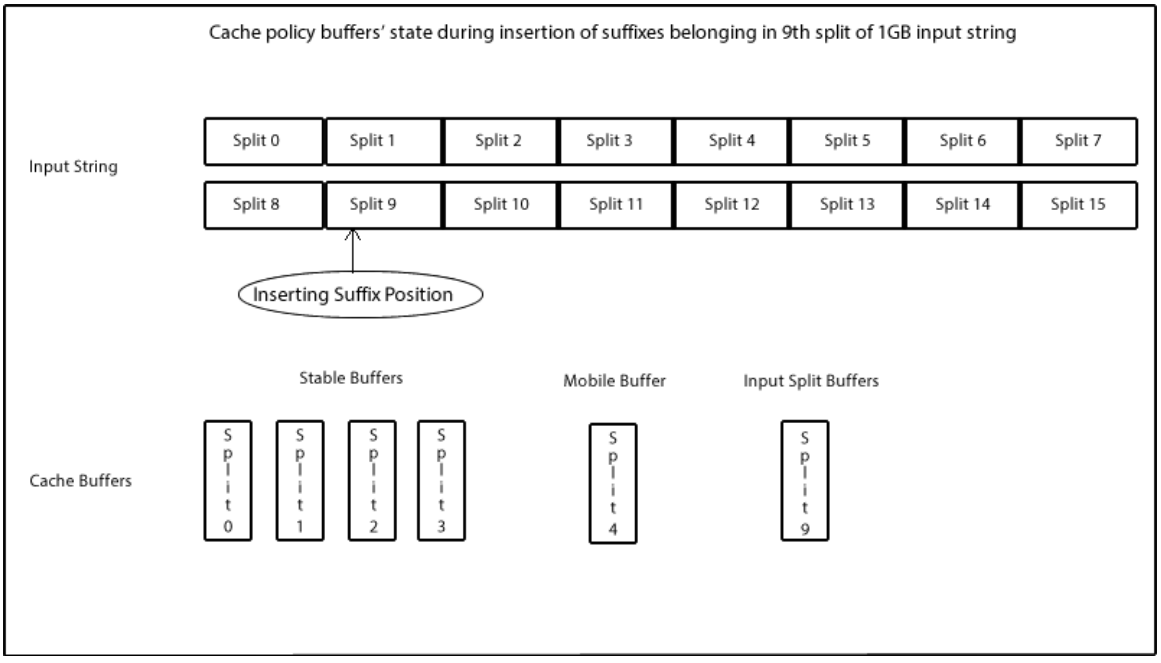


Figure 14: Cache policy buffers' state during insertion of suffixes belonging in 9th split of 1GB of input string and 6 buffers

5.3.7 Complementary insertion phase

The existence of this phase, as mentioned briefly in the previous section, is due to the inability of some suffixes to be inserted with the existing in-memory split buffers. While adding a suffix belonging in a particular split, this split is stored in the Input Split Buffer and remains there until all suffixes from that split are inserted. If the maximum number of splits consisting the input string are more than the maximum split buffers some suffixes would request a split (while traversing the tree in search for the insertion point) that is not available in the main memory. Furthermore, some suffixes might surpass the current Input Split Buffer size while traversing the tree.

In order to insert those suffixes and avoid excessive I/O by granting their requests immediately, we created 4 integer arrays to store their traversal information and create a complementary insertion phase for all these suffixes. The information needed is the node and the edge they are on, and the offset already compared from the suffix as well as the suffix position. These arrays are 2-d and the first dimension has to do with the unavailable split at the time of the request.

Until the start of the complementary insertion phase the tree might have changed but all the unsuccessful requests occur at the start of the edge where according to the start position a split is requested to compare characters with the suffix. This insures that even though further down or up the tree changes have been made the node and edge's start position will for sure be the same as well as the path leading to this node and edge.

This phase will start exactly after the completion of insertion of all suffixes belonging in the working split. Then starting from the lowest to the higher, in terms of split number, we start to insert the unsuccessful suffixes by fetching their requested split in the Mobile Buffer in memory, after all suffixes in each split have been processed successfully or not.

Even in this phase it is possible that an unavailable spit will be requested that is not currently in main memory. The requested split in that case is for sure higher than the working Mobile Buffer, and the suffix's insertion information will be stored in the arrays of the requested split and processed when that split arrives in memory in the same way as in the main insertion phase.

The size of the arrays storing the insertion information for unsuccessful suffixes depends on many factors. The threshold is significant variable since the higher the threshold the more are the suffixes per tree and per split and consequently the higher the possibility that some won't be inserted. Another factor is the input length since regardless of threshold the larger the input length the lesser the suffixes per split. The third factor is the number of available buffers in the cache policy since the more the available buffers are the less the unsuccessful split requests will be made.

All the above have meaning only for input lengths that surpass the capacity of the cache buffers. The exact size of the arrays have not been modeled mathematically since there are too many variables involved including randomness in the suffix insertion but rather empirically calculated. In the experimental phase it was observed that the maximum size of these arrays is about 60.000 and the minimum about 15.000. The size of these arrays plays an important role in the configuration of the suffix tree's variables (threshold, cache buffers and input length) since four 60.000 long arrays for 2GB of input data takes up to 30MB of memory which is half an input buffer.

In figure 15 an example is illustrated for a better understanding of the way the complementary insertion phase with the arrays holding the insertion information of each split that unsuccessfully had been requested in the main insertion phase. The snapshot is at the start of the complementary phase where the arrays will be processed from left to right by fetching the needed split in the Mobile Buffer.

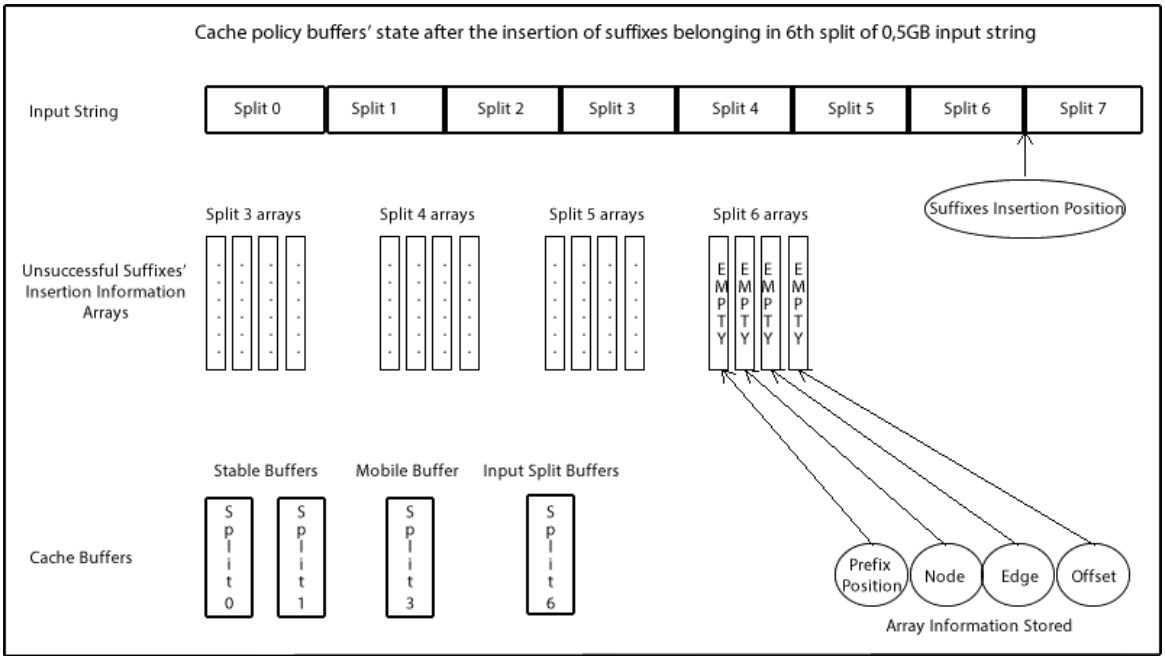


Figure 15: Memory state at the start of execution of complementary phase just after the completion of insertion of all suffixes in 6th split in a 0,5 GB input string and 4 cache buffers example

5.3.8 Observations and optimizations

A significant optimization that facilitates the suffix insertion was made in the tree traversal that reduced significantly the I/O cost and suffix insertion failures due to split buffer unavailability.

As discussed in chapter 2, one additional useful piece of information was each edge's first character. That information was extracted by the position the edge was stored in the array; the 1st edge always starts with 'A', the 2nd with 'C' and so on. The reason that this facilitates the traversal is the fact that due to the Suffix Tree's dense nature in real DNA data many edges have exactly one character. These edges can be compared with no string access and no I/O cost. Adding this useful feature in the tree traversal we are able to go down the tree more easily and for larger (in number) splits access the edge that will eventually be split. This feature is also appealing during the query phase since for small queries the I/O cost is diminished.

Another observation was made concerning the size of the tree. In our original implementation the suffix tree occupies memory equal to 56 times the number of inserted suffixes. As discussed in chapter 2 the end position of an edge can be deduced from the start position of the first edge of the end node. If we remove from our arrays the end index information and instead create an array of bytes that has the size of our nodes array storing the byte 'L' in the case of leaves and the index of the first inserted edge in the internal nodes we can reduce the memory requirements by 14 bytes.

Although this observation sounded appealing to have 42 times the size of the suffixes (consequently 42 times the input string for all the prefixes suffix trees) instead of 56 the experiments have shown a significant delay in the execution of our program. Note that the end index information of an edge is needed in every step of our program whether to check if the edge has one character or to realize when the traversal has reached the end of the edge. The reason for the delay is that in the 56 times implementation the information was accessed directly from the array whereas in the 42 times implementation the end node must be accessed, the byte array must be checked (in order to determine whether the end node is a leaf or an internal node) and finally the start index of the first inserted edge in the node must be accessed to deduce the end index increasing the array accesses (from 1 to 3) and thus the computational overhead.

5.4 Query phase

The query phase didn't have to be done in parallel since after the completion of the creation of our suffix tree we are able to query our results with a simple java application after having all our prefixed suffix trees moved in our hard drive.

The query phase is quite straightforward since the representation of our suffixes already exists for the creation phase. The root suffix tree needs to be loaded in memory and the traversal of this tree will result in the prefix sub-tree that needs to be loaded for the rest of our query.

After loading the suffix tree a simple function is needed in our implementation that traverses the tree from the root down until the substring given for query either ends or a failed comparison is made, in the same way as in the suffixTreeAddSuffix algorithm. In the first case we use a stack structure to store the nodes while traversing all the remaining sub-tree looking for leaves in a pre-order depth first manner. In the second case there is no result since this substring doesn't exist in our suffix tree and consequently neither does it exists in our input string. The leaves are put in an array list which is sorted after the completion of the tree traversal.

In figure 16 a simple query is illustrated. The query is written to the left window of our program and after submission the prefixed suffix tree is loaded (GTAC in the example), the leaves are obtained (which are namely all the leaves that have as common prefix the query sub-string) then sorted and outputted to the right window.

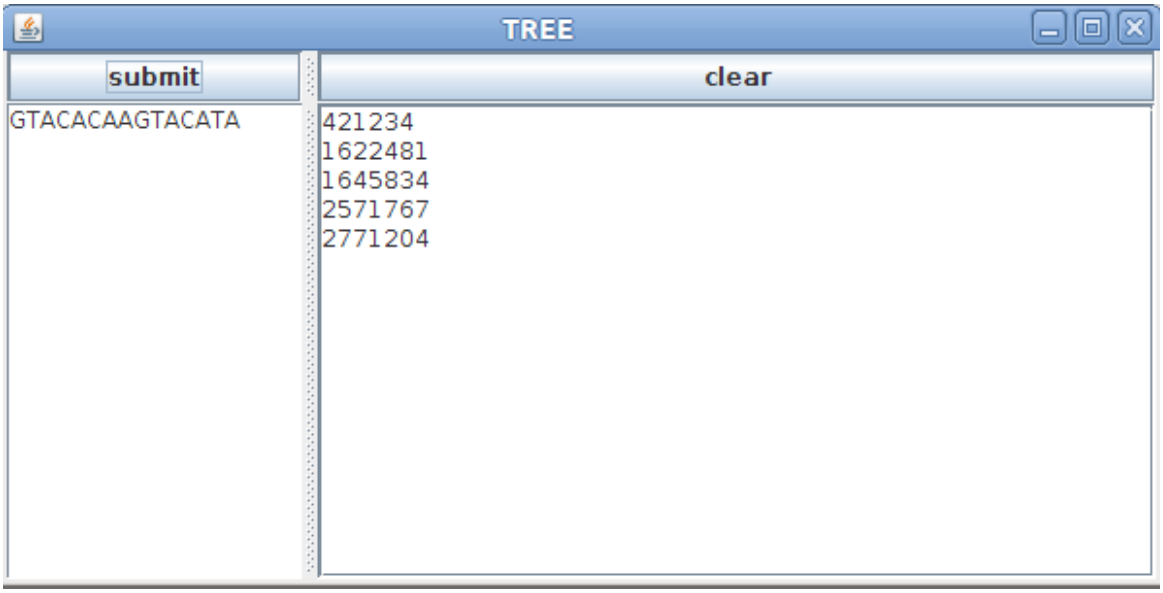


Figure 16: Query snapshot

Chapter 6 - Experimental evaluation

6.1 Introduction

All experiments were conducted in a Hadoop cluster release 1.0.3 with one master node and 16 slave nodes. The cluster is consisted of two types of machines. The one type, master node and 12 slave nodes, runs Ubuntu Linux operation system release 9.0.4 with Intel Xeon 4 core CPU, model X3323, at 2.50GHz and 4 GB of RAM. The other type, 4 slave nodes, runs Ubuntu Linux operation system release 10.04.4 with Intel Xeon 8 core CPU, model X3440, at 2.53GHz and 4 GB of RAM. All machines' hard drives have 500 GB capacity with read and write speed at 7200 rpm.

The Hadoop configuration allows each task (map or reduce) to occupy one core of CPU from the machine and 1 GB of RAM. The aforementioned configuration allows a machine to run 4 parallel tasks at any given time allowing a Map-Reduce job to have up to 64 parallel tasks for all 16 nodes. The HDFS has capacity of 7,43 TB.

All the code for the Map-Reduce jobs were written in JAVA SE 6 in NetBeans IDE 7.0 platform. The library used for accessing Hadoop Map-Reduce capabilities in the code was Hadoop-0-20-2-core.jar. The main files in the implementation were the MapReduce.java which had all the configuration parameters and the main function, the SuffixTree.java which was implemented for two representation (42*N and 56*N), the PrefixFinder.java which had the main functions for the prefix creation phase, the PrefixTree.java which was had the implementation for the root tree of prefixes, the Cache.java which was the implementation of the input cache policy. There were other auxiliary java classes for reading and writing in the HDFS for either a map or reduce task.

In the following sections we will evaluate which suffix tree representation was more efficient, the impact that the threshold had on execution times, the possibilities of the input cache policy with the increase of buffers. Finally after finding the best scenario in terms of threshold, buffers and representation we will evaluate the scalability for increasing input of our implementation. Note that mean times were recorded for Map-Reduce tasks since the cluster nodes' diversity in terms of CPU capabilities would make absolute times misleading for our algorithm's behavior. All DNA input came from [20].

6.2 Prefix creation phase evaluation

The prefix creation phase as explained thoroughly in the previous chapter is a Map-Reduce task that discovers the variable-length prefixes that the suffix tree can be divided into given an input string and user defined threshold.

The first experiment concerned the execution times for this Map-Reduce task for increasing input to test its scalability. As explained in previous Chapters each Mapper has a fixed-length input of 64 MB equal to the Hadoop configuration for the HDFS block size. That means that the number of Mappers was equal to the input length divided with 64 MB. The experiment is illustrated in figure 17 and the scalability for this Map-Reduce job is obvious.

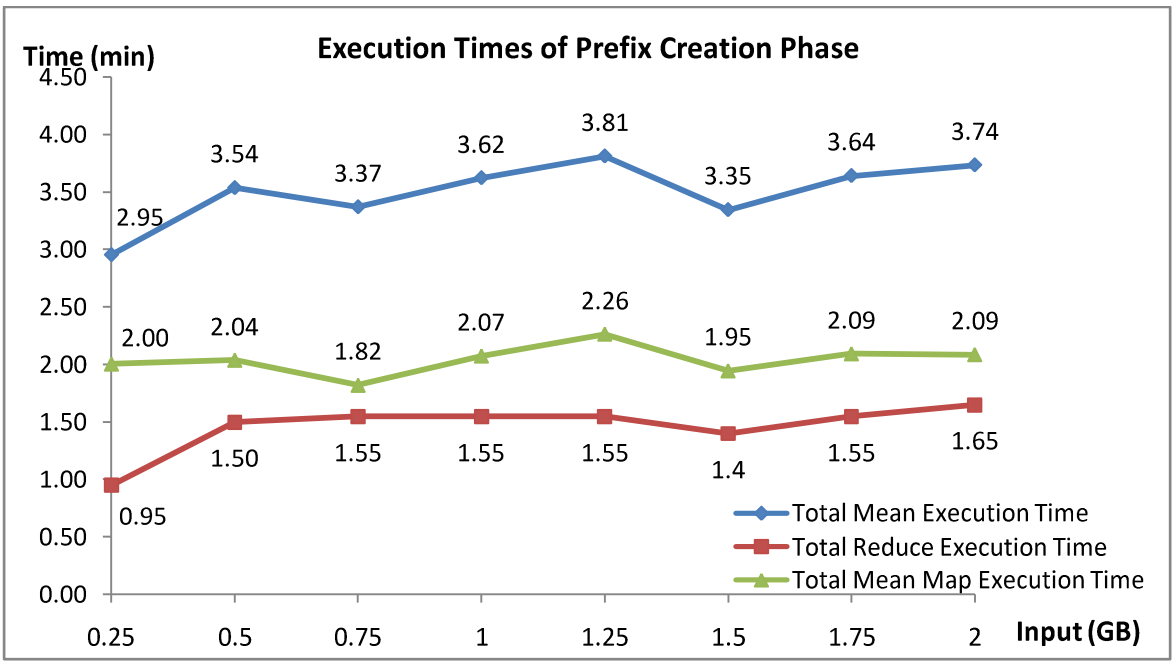


Figure 17: Prefix Creation Phase Execution Times

It takes an average of 3,5 minutes to successfully find all prefixes and create the root prefix suffix-tree. Note that different threshold values didn't affect the execution time for this phase since the main time consuming task was the searching of the prefixes throughout the entire input string. Evaluation shows that this phase scales even though more than 1 task was assigned to each node for inputs greater than 1 GB.

The low peak in 1,5 GB, in figure 17, can be explained since the best ratio discovered for Hadoop Map-Reduce jobs is to have a number of map tasks equivalent to 1,5 times the number of the cluster's nodes.

The second experiment was about the impact of increasing threshold to the number of resulting prefixes. As illustrated in figure 18 the number of resulting prefixes is proportional to the user-defined threshold and the correlation is linear. The larger the threshold the less prefixes would be found.

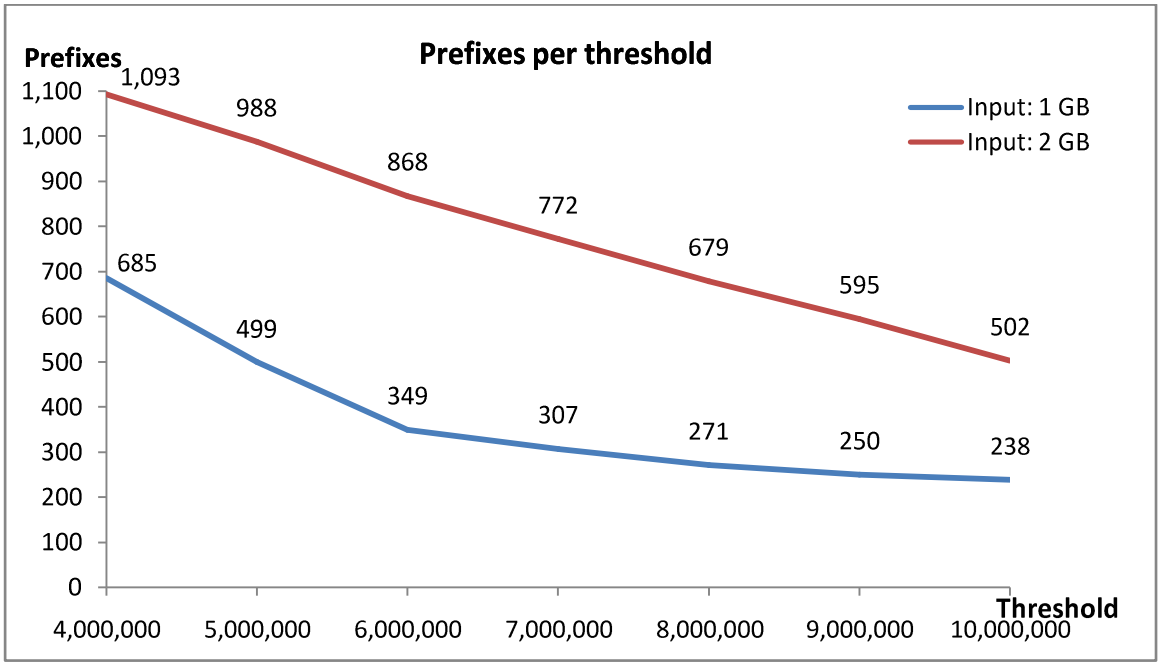


Figure 18: Number of prefixes per threshold for different inputs

The last experiment we used the same threshold (9000000) and we increased the input size as illustrated in figure 19. The increase of resulting prefixes is linear to the increase of the input string's length.

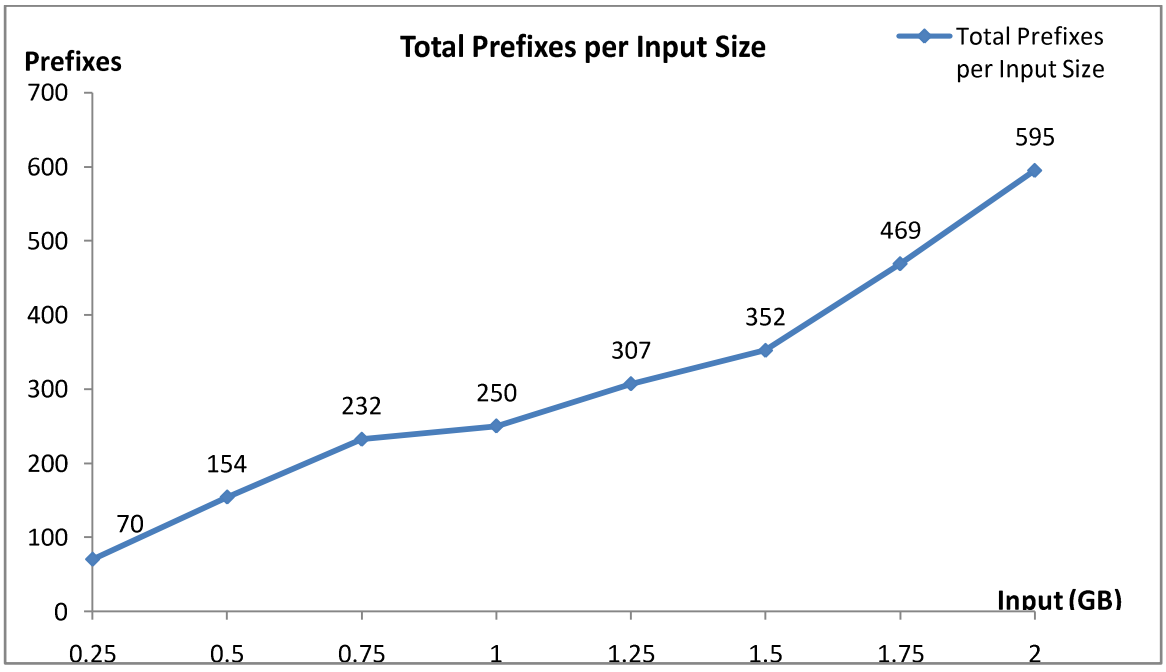


Figure 19: Total Prefixes per input size

6.3 Suffix Tree representation and threshold evaluation

The first step of our experimental evaluation for our suffix tree creation phase was to determine which representation of the suffix tree would provide better execution times for the creation of our suffix tree. Two representation with arrays were implemented, as discussed in previous sections; 56*N and 42*N representation with the difference that in the latter implementation the end positions of edges were deduced by the end node's first edge start position.

Threshold	56*N Max Tree Size (MB)	42*N Max Tree Size (MB)	Gain (MB)
4.000.000	213,6	160,2	53,4
5.000.000	267,0	200,3	66,8
6.000.000	320,4	240,3	80,1
7.000.000	373,8	280,4	93,5
8.000.000	427,2	320,4	106,8
9.000.000	480,7	360,5	120,2
10.000.000	534,1	400,5	133,5

Table 2: Two representations' memory requirements

The two representations' memory requirements are depicted in Table 2 for increasing threshold and the gain for the same threshold between the two is shown in the last column. As expected the gain for increasing threshold reaches up to two buffers size which means that with the 42*N representation for the same threshold we can use up to 2 additional buffers in our input cache policy.

Although one would expect that adding two additional buffers would provide better completion times and for very thresholds lower than 6000000 that was indeed the case but as the threshold increased the 56*N was a definite winner. The reason for this unexpected result was mainly because the computational overhead added by deducing the end position of an edge, an attribute needed in every comparison while traversing down the tree, instead of explicitly storing it in an array was rising drastically as more suffixes needed to be inserted in the suffix tree.

As discussed in Chapter 2 the 56*N representation needs a single check in the positions array to determine the end position of an edge whereas the 42*N representation needs three checks in three arrays; we must bring the next node from the nodes array, we must check whether it is a leaf or internal node from the notations array and finally we must check the first edge's start position. The computational

overhead rises as the threshold increases since the bigger the threshold the more suffixes need to be inserted per prefixed suffix tree.

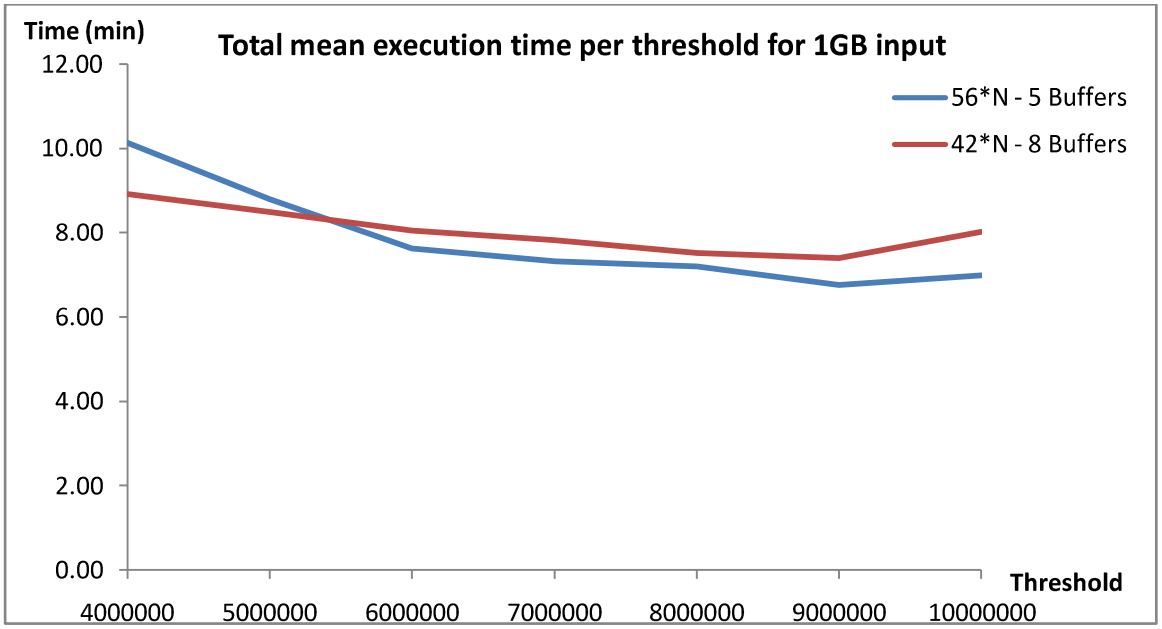


Figure 20: Total mean execution times for the two suffix tree representation

In figure 20 the total mean execution times are illustrated for the two representations. Note that we used 3 more buffers in the 42*N representation which is one buffer more than the actual gain in memory for the biggest threshold and the results are still clearly in favor of the 56*N representation. All the experiments were done for the same input of 1 GB of real DNA data.

Another unexpected discovery was that when the threshold overcame 9000000 the execution times didn't improve but instead worsened. The reason for this unexpected behavior is again computational overhead created by the increasing number of inserted suffixes. Since the behavior is clear in both representations we can deduce that for threshold beyond 9000000, where the prefixed trees will be less for each reducer, no gain is earned.

With the experiments conducted and illustrated in figure 20 we can safely conclude that the 56*N representation is faster for greater thresholds and that the best times for suffix tree creation are met with threshold equal to 9000000 suffixes. With the aforementioned representation and threshold, the memory that will be needed for keeping our biggest prefixed suffix tree in memory will be 480,7 MB which means that almost half of our available memory will be dedicated for suffix tree storage.

6.3 Input cache policy and I/O cost evaluation

After establishing the threshold values and the suitable suffix tree representation the next step is to evaluate our input cache policy and discover the correlation between I/O cost and execution times.

The first experiment to discover the impact of increasing buffers in our input cache policy was made for two different inputs (1 GB and 2 GB) with the same threshold of 6000000 and the same representation (56*N). The reducers were always 16, one for every node of our cluster.

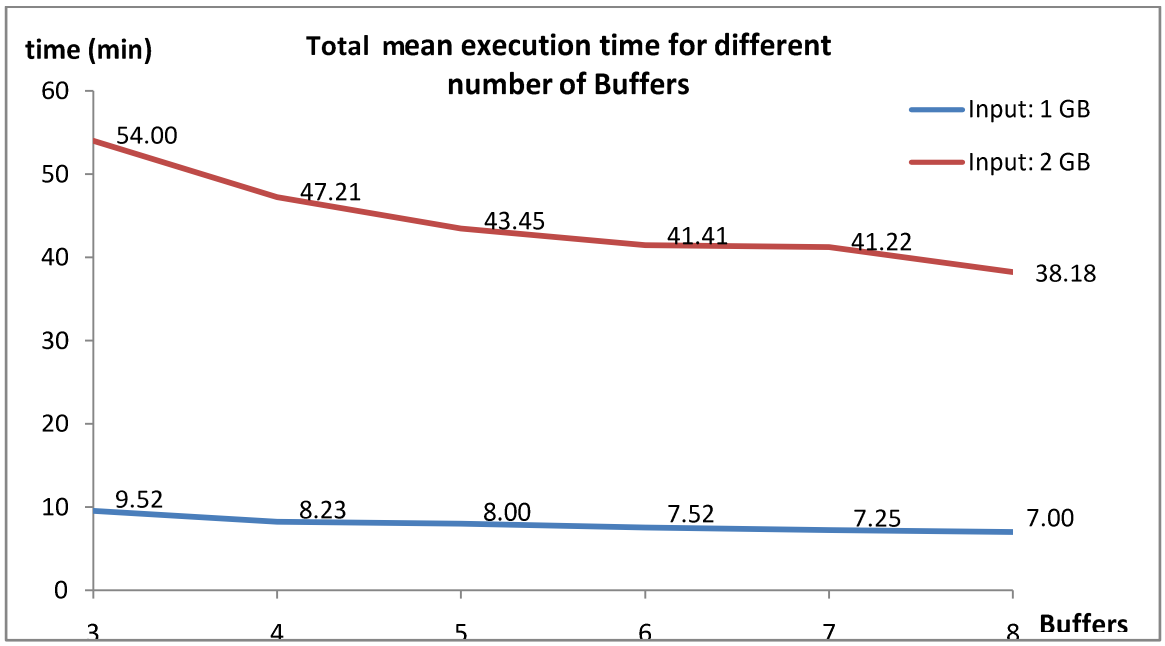


Figure 21: Total mean execution times for increasing buffers and different inputs

In figure 21, the impact of increasing buffers is illustrated and the major difference can be located for larger inputs and the explanation is given in figure 22 where we counted the total I/O made per reducer for the same experiment. The resulting figure informs us that better execution times come with more buffers a conclusion that was expected.

The conclusion that wasn't expected was the high degree of correlation between the reduction of the total number of I/O for increasing buffers and the decrease of execution time for both inputs. The aforementioned observation allows us to reach to the conclusion that the execution time of our program depends heavily on the I/O cost dealt by each reduce task.

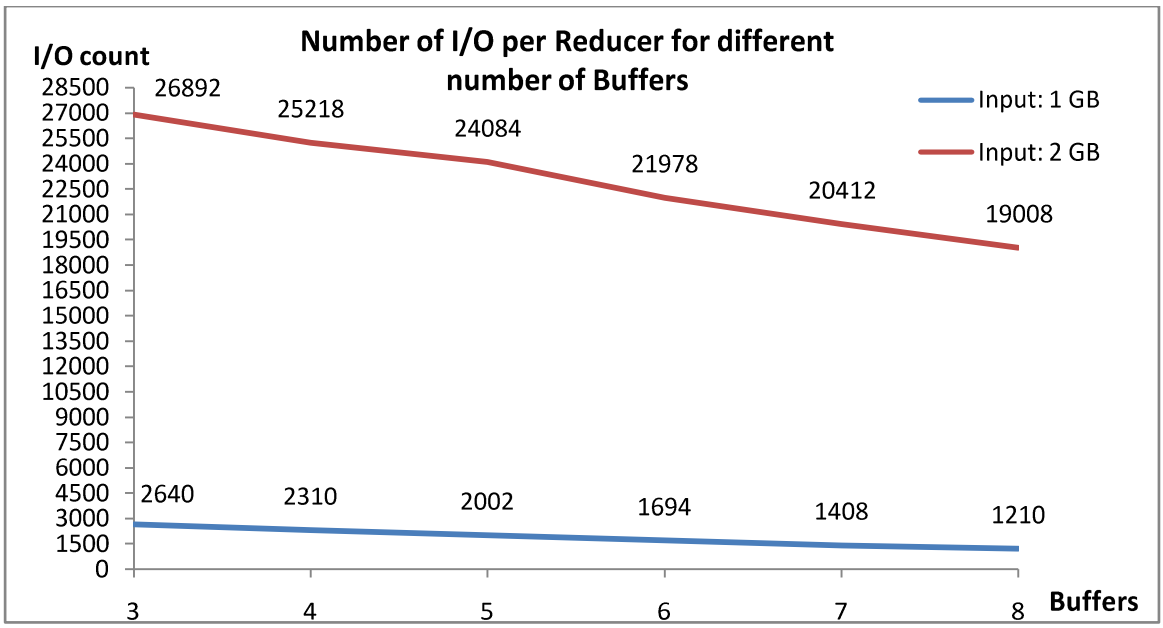


Figure 22: Number of I/O per Reducer for increasing number of Buffers and different inputs

The second experiment in order to evaluate our algorithm's input cache policy involved increasing the input size with fixed threshold (9000000) and fixed number of buffers (6 buffers). The increase of the I/O cost per reducer (fixed total number of 16 reducers) was incrementing quadratically to the increase of the input size (figure 23 blue line).

Note that increasing the input means more prefixes per reducer and more I/O per prefixed suffix tree. Since we concluded that the main factor for execution times was the I/O cost we estimated and projected the I/O cost if we had the same prefixes per reducer (figure 23, red line) and if we had buffers of the same percentage of the input string and the same prefixes per reducer (figure 23, green line).

These estimations were calculated for input larger than 1GB where the buffers occupied almost the 40% of main memory. Using 40% of the input string for buffers means more buffers than the available main memory set by our cluster configuration (1 GB per task). If we had 2 GB available memory per task the green line could be easily result from an experiment.

As far as the red line in figure 23 is concerned we would need more nodes in our cluster in order to have the same prefixes for each reducer since one reducer is given for every node. In table 3 we can see the requirements for the projections we made based entirely on I/O cost which was calculated by the type given in section 5.3.6 of our input cache policy.

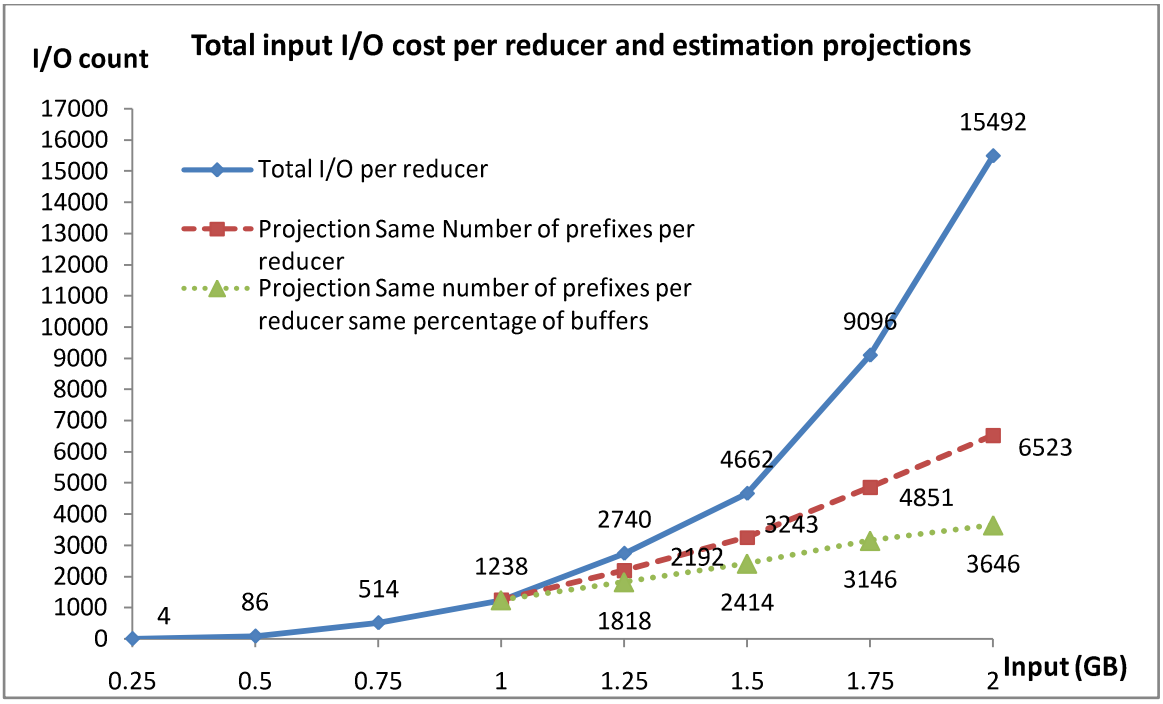


Figure 23: Total I/O cost for increasing input and estimation projections

In either admission for reducing the I/O cost, whether we increase the nodes in the cluster or we additionally increase the available memory for our reduce tasks we can observe from figure 23 that the I/O cost would lose its quadratic increase nature and will result in a simple linear increase. While using both the same number of reducers and the same percentage of the input string as buffers the results are much better.

Input Size (GB)	1	1,25	1,5	1,75	2
Number of buffers	6	8	9	11	12
Buffers size (MB)	384	512	576	704	768
Total reducers	16	20	23	30	38

Table 3: Projections requirements in buffers and reducers

Both the admissions made for the projections in figure 23 are not arbitrary since the increase of the input results not only in increase of the input that must be scanned but also the prefixed suffix trees that must be created. In order to evaluate our algorithm's scalability we have to offer fixed parameters for all the other aspects of our program while increasing the input. That means that the percentage of the input string that the buffers occupy in our input cache policy must be fixed as well as the number of prefixed sub-trees that is assigned to each reducer to create.

In the next section we will evaluate our algorithm in terms of execution times for the aforementioned admissions.

6.4 Scalability evaluation

In order to check our implementation's scalability to increasing input we had to observe both the Map phase of our suffix tree creation job as well as its overall performance in terms of total mean times.

In figure 24 is illustrated the Map's phase mean execution times. For this experiment we used fixed threshold (=9000000), fixed buffers (=6) and fixed split size (=64 MB). The Mappers used was one for each 64 MB of available input.

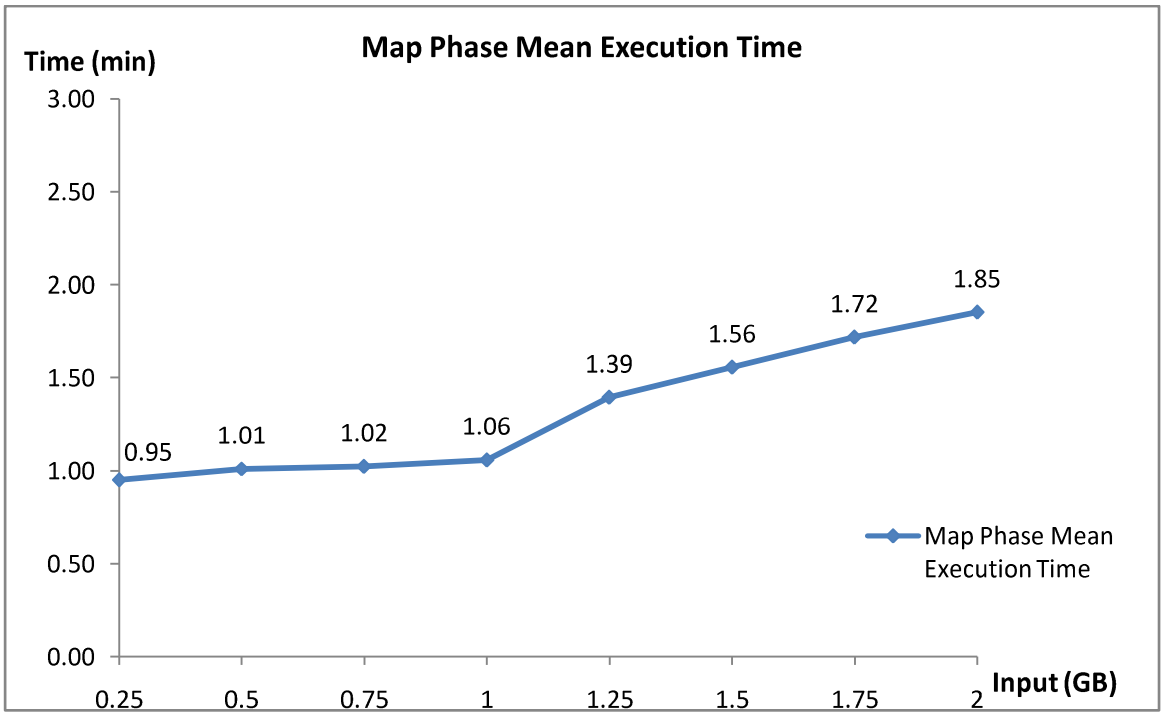


Figure 24: Map Phase mean execution times for increasing input

The main aspect of the map phase that can be addressed is the algorithm's behavior when the Mappers exceed the available nodes. With 1 GB of input we have 16 splits that will be assigned to 16 nodes, consequently each node will be assigned with a single map task. For inputs larger than 1GB more than one map tasks will be assigned to each node and that is the reason for the linear degradation of our map phase's performance.

The time differences between the 4 Mappers (0,25 GB) and the 16 Mappers (1 GB) were marginal which can lead us to the conclusion that if we had one node for each 64 MB of input in our cluster the map phase (of suffix acquisition) would scale for increasing input.

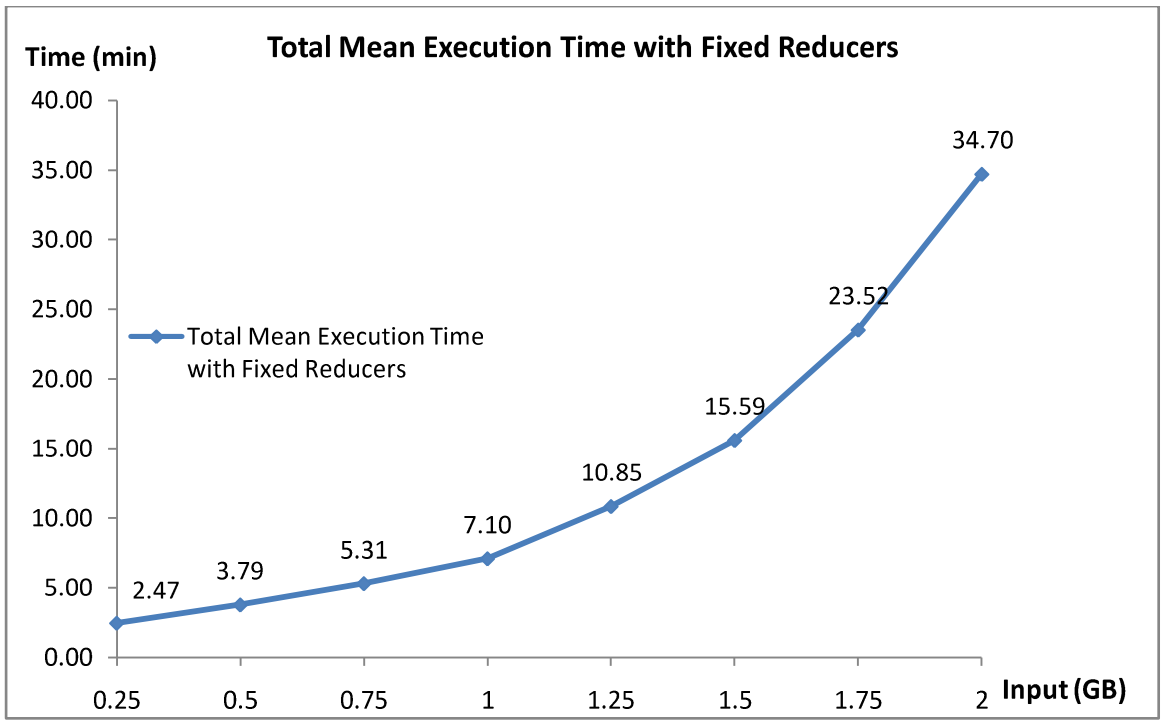


Figure 25: Total Mean execution times with Fixed Reducers

The total execution times for the same experiment are shown in figure 25 with the blue line and we can see that the increase of execution times is quadratic to the increase of the input string. Although as discussed in the previous section and as illustrated in figure 23 we can calculate an estimation of the I/O cost for the reduce phase (which creates the suffix trees) if we has the same number of prefixes for each reducer and additionally the same percentage of input buffers .

Despite its quadratic behavior the observation of an almost linear behavior up until the 1 GB of input data made us conduct another experiment. It is important in order to prove the scalability of our algorithm to provide the same resources as the input rises. A way to prove our expectations in that last experiment we increased the number of Reducers according to the increase of the input string.

In reality we used as many Reducers as the required Mappers to process the input string (4 reducers for 0,25 GB of data, 8 reducers for 0,5GB and so on). The experiment could not exceed 1GB of data since the available machines were 16 and we didn't want to exceed the 1 reducer per machine ratio. The experiment is depicted in figure 26 along with the projections of the expected behavior if we could further increase the number of Reducers and if could further increase the allocated memory for our cache management policy.

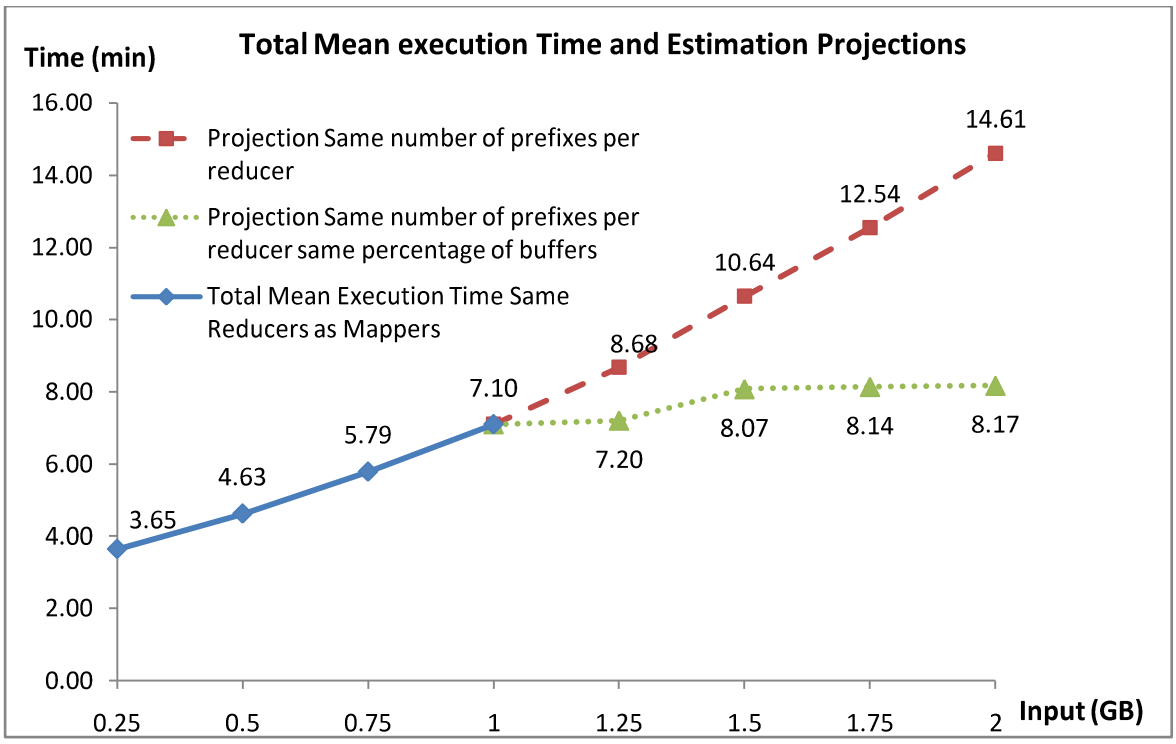


Figure 26: Total Mean execution times and Estimation projections

The blue line illustrates the behavior of our algorithm while increasing the number of reducers along with increase of the input string. It becomes obvious that our implementation behaves in a linear fashion proving a medium scalability if we could increase the number of machines used along with the increase of the input string. In other words the increase of reducers is almost equivalent to keep the prefixes per reducer fixed as proven in the experiment.

The estimation of the execution times for the first admission of having as many reducers needed in order to have the same number of prefixes per reducer (which mean the same number of trees created by each reducer) reveals a linear behavior (red dashed line). The second estimation of having both as many reducers as prefixes and the same buffer percentage of the input string reveals the conditions under which our implementation would scale.

In order to check if our implementation scales it is not arbitrary to assume that while increasing the input all the other parameters of our implementation must remain fixed, since the increase of the input affects both the working set of buffers and the trees that need to be created for the same threshold parameters. For the estimation of the execution times only the I/O cost was taken into consideration as discussed more thoroughly in the previous section.

Chapter 7 – Conclusions

7.1 Challenges

The main challenge derived from the fact that Hadoop Map-Reduce is in fact a Java platform and the program needs to be written in Java. Although there are other alternatives (like C++) only Java was used within this master thesis which was the programming language recommended by the platform.

Although one can use Java for any existing problem and take advantage of all its ready libraries and many conveniences, Java is not an efficient language when it comes to memory allocation and management. Starting from the fact that for every object instantiated in a Java application the Java Virtual Machine allocates additional header and tail bytes for housekeeping purposes, it also frees unused objects and structures whenever the Garbage Collector sees fit making the manipulation and full exploitation of the available main memory a far from easy task.

Another basic problem that we faced was the fact that in Java, unlike C or C++, doesn't have unsigned Integer as primitive type. The problem with that fact is that we couldn't use all 4 bytes for the representation of Suffix positions and that if we wanted to experiment on data larger than 2GB we would have to use long integers that occupy 8 bytes. Our 56*N representation of suffix tree would immediately skyrocket to 88*N and our 42*N representation would reach up to 58*N. Other solutions had been considered like creating a data type with a simple class but the instances overhead made it prohibitive.

Another issue that must be addressed is the fact that for the suffix tree creation phase we used one reducer for every available node. The reason for that admission was the fact that suffix trees need the entire input string to be created. In experiments not included in the experimental evaluation chapter it became obvious that when using more than one reducer per node the degradation of time execution was drastic due to the fact that multiple processes read and wrote to the hard disk simultaneously resulting in I/O collisions, competition and task failures.

A similar issue was dealt with when trying to decide the better way to access the input string by the reducers. The first admission was to access it through the HDFS in the same way all Map tasks in our implementation did. After extensive experimenting we realized that because of the demanding nature of input I/O the HDFS was not the right way for the reducers to access the input string since both the transfer time and collisions between tasks resulted in severe time degradation and task failure.

The second admission was to use Hadoop distributed cache libraries in order to distribute the input string to all the nodes. This solution was better than using the HDFS but it was still too slow since distributed cache was designed for caching small files and not GB of data and additionally transfer time was needed to distribute the input file to all nodes local hard disks, also resulting in time degradation.

The final and winning admission was to keep the input string locally in the hard disk of every participating node. Through the master node a simple script using the "scp –r" command suffices in order to move the input file in all nodes to the /tmp folder which is available for I/O by all reduce tasks.

For the aforementioned reasons and using the same experiments to verify we also concluded that the resulting prefixed suffix trees should be written in the /tmp folder locally in every node and at completion extracted with a simple script to the master node in our account's /tmp folder.

7.2 Conclusions

In this master thesis we made an effort to create efficiently suffix trees using Hadoop Map-Reduce platform. After the study of the main algorithms presented over the years we proposed a parallel implementation using simple techniques and algorithms adapted to the requirements of Hadoop Map-Reduce.

During this work we faced many challenges concerning the programming language, the Hadoop framework and the problem's high complexity as described in previous sections. We've managed to present an efficient algorithm that can index genome-scale DNA sequences in less than 40 minutes, the fastest Map-Reduce implementation to the best of our knowledge.

A Map-Reduce implementation [21] that tried to adjust Trellis in Hadoop Map-Reduce environment managed to index genome-scale DNA sequences into suffix trees in about 3 hours. The main reason was that the intermediate data generated by Trellis were overwhelming for the efficiency of the algorithm a conclusion also reached by the authors.

As a general conclusion about the functionality of a Map-Reduce job was that efficient algorithms cannot flood the network with data since the degradation is inevitable. Clever ways must be applied to minimize intermediate data that have to be sent over the network.

For our algorithms, as a general principle for large inputs the suffix tree creation phase and more specifically the suffix insertion – reduce phase cannot perform efficiently using Hadoop's HDFS for reading the input string and for writing the prefixed suffix trees. The high requirements of this phase in terms of I/O made the overhead of transfer times and requests' collisions prohibitive for our problem and forced us to read and write locally in every node's hard disks.

We also proved that the prefix creation phase scales for increasing data regardless of node availability. Also, as far as the map phase of the suffix creation phase is concerned the scalability is not high when we surpass the 1 Mapper per node rule. With more nodes this phase would scale well. Overall for the suffix tree creation phase we proved that keeping the buffer size fixed as well as the number of reducers makes our algorithms behavior quadratic to the size of the increasing input. We also suggested that keeping the number of prefixes fixed per reducer (by increasing the nodes) would result in a linear behavior and much better scalability and would reach high scalability if we additionally kept the percentage of our buffers to the input string fixed which would require more available memory than 1 GB set by Hadoop configuration.

Another significant contribution of our work is that we managed in a parallel environment to avoid excessive random access I/O using a complementary insertion phase that grouped all suffixes which requested chunks of the input string not available in main memory and the input cache policy that made it possible. The input cache policy was embedded in our program with the sole requirement that all I/O must be done locally to ensure the algorithms efficiency as explained above.

Finally we conclude that Hadoop Map-Reduce provides us with the tools of implementing efficient algorithms for suffix tree creation and also that Java is not the appropriate programming language to implement them with. Our implementation in terms of completion times achieved is slightly inferior to state of the art implementations and would definitely be competitive if we could experimentally reach the high scalability scenario using the two aforementioned admissions concerning fixed prefixes per reducer and fixed percentage of buffers to the input string.

7.3 Future work

A more efficient implementation could be written as far as the programming language is concerned. Hadoop Map-Reduce gives as the capability to run programs written in C++ and we could avoid all the memory allocation and freeing problems generated by the Java Virtual Machine and the Garbage Collector. It is not a random fact that most implementations in related work were written in C and few in C++. That would also allow us to reach up to 4 GB of input DNA using the same representation with unsigned integers and to handle more efficiently the limits of our available main memory.

The second most important thing that could be done, as an extension to the current work, is to experimentally prove that this implementation actually scales if we could keep fixed the number of prefixes assigned to each reducer by increasing the number of nodes (so as to have 1 reducer per node) and the percentage of the buffers to the input string's size, used in our input cache policy. In order to achieve the aforementioned optimizations we would need more machines in our cluster and more memory per task given by Hadoop's configuration.

Finally by implementing the aforementioned optimizations we could experiment on sizes larger than 4 GB by using different representations without increasing our machines CPU and memory capabilities. We could also experiment on using different input cache policy and even different insertion algorithms that wouldn't have to traverse from the root down the tree for every insertion, a problem located in the related work but not dealt with.

References

[1] E. M. McCreight, *A space-economical suffix tree construction algorithm*, J. of the ACM 23 (1976), no. 2, 262–272.

[2] P. Weiner, *Linear pattern matching algorithm*, IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.

[3] E. Ukkonen, "On-line construction of suffix trees." Algorithmica, 14(3):249-260, 1995.

[4] U.Manber and E. Myers, *Suffix arrays: A new method for on-line string searches*, SIAM J. of Computing 22 (1993), no. 5, 935–948.

[5] P. Ferragina and R. Grossi, *The string B-tree: a new data structure for string search in external memory and its applications*, J. of the ACM 46 (1999), no. 2, 236–280.

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

[7] E. Hunt, M.P. Atkinson, R.W. Irving "A database index to large biological sequences." The VLDB Journal, 7(3): 139–148, 2001.

[8] Y. Tian, S. Tata, R. Hankins, J. Patel " Practical methods for constructing suffix trees." The VLDB Journal, 14(3) : 281–299, 2005.

[9] R. Clifford, M. J. Sergot "Distributed and Paged Suffix Trees for Large Genetic Databases." Proceedings of the CPM-2003: 70–82, 2003.

[10] S.J. Bedathur and J.R. Haritsa "Engineering a fast online persistent suffix tree construction." Proceedings of the 20th International Conference on Data Engineering: 720, 2004.

[11] B. Phoophakdee and M. J. Zaki. "Genome-scale disk-based suffix tree indexing." In Proc. of ACM SIGMOD, pages 833-844, 2007.

[12] M. Barsky, U. Stege, A. Thomo, C. Upton "A new method for indexing genomes using on-disk suffix trees." Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM: 649–658, 2008.

[13] M. Barsky, U. Stege, A. Thomo, and C. Upton. "Suffix trees for very large genomic sequences." In Proc. Of ACM CIKM, pages 1417-1420, 2009.

[14] A. Ghoting and K. Makarychev "Serial and parallel methods for I/O efficient suffix tree construction." In Proc. of ACM SIGMOD, pages 827-840, 2009.

[15] E. Mansour, A. Allam, S. Skiadopoulos, P. Kalnis “ERA: Efficient Serial and Parallel Suffix Tree Construction for Very Long Strings” In Proc. of VLDB, Vol 5, pages 49-60, 2012.

[16] R. Giegerich, S. Kurtz, and J. Stoye, *Efficient implementation of lazy suffix trees*, Software—Practice and Experience **33** (2003), no. 11, 1035–1049.

[17] T. White, *Hadoop - The Definitive Guide - MapReduce for the cloud*, O’Reilly Media Inc, June 2009.

[18] M. Barsky, U. Stege and A. Thomo, *A survey of practical algorithms for suffix tree construction in external memory,* John Wiley and Sons ltd, 2010

[19] J. Karow, *Group unveils “1,000 genomes” study to map genetic variants using new sequencing tools*, http://www.genomeweb.com/sequencing/.

[20] NCBI g*enebank overview*, http://www.ncbi.nlm.nih.gov/Genbank/, June 2013.

[21] A. Karmis, T. Sellis, “Implementation of Suffix Tree construction with Hadoop Map-Reduce”, graduate thesis, NTUA, Athens 2010