CS-301
HIGH PERFORMANCE COMPUTING

PROJECT ON
PARALLEL SUFFIX TREE CONSTRUCTION

COURSE INSTRUCTOR: Prof. Bhaskar Chaudhury

SUBMITTED BY:

1. Anusha Phadnis (201401098)
2. Malvika Singh (201401428)

INTRODUCTION:

In Biology, DNA is of prime importance in any kind of genomic research. DNA, when broken down, consists of nucleotides having nitrogen bases, which may be of the type A, G, C or T. The specific arrangement of these nucleotides and its study opens up an entirely new horizon for biological research as this DNA is unique for every organism and is responsible for the characteristics shown by it. Due to this exclusiveness that a DNA offers, it occupies a central role in bioinformatics domain.

Many processes take place in these DNA and protein sequences which change the composition of the sequence, for example, translation, mutation etc. So there arises a need to compute various kinds of matches between different sequences to see the amount of similarity that they have. The results of such computations are then used for many further applications.
Another important aspect in order to develop a data structure to hold such sequences is the size of these strings.The GenBank database contains more than 100Gbp, and it is generally believed that its size will double every 6 months (GeneBank, 2005). The size of the entire human genome is in the order of 3 billion DNA base pairs, whereas other genome scan be as long as 16Gbp.
In this scenario one of the most important needs is the design of efficient techniques to store and query biological data. We, therefore, need mechanisms to handle millions and billions of such strings. Due to this, the notion of suffix trees and suffix arrays evolved. The suffix array of a string is lexicographically sorted list of all its suffixes. Suffix tree is a trie which contains all the suffixes of the string with its edges representing suffixes and nodes representing positions in the text. These are important fundamental data structures useful in many string matching applications in computational biology.

PROBLEM STATEMENT:

We have tried to develop a parallel code for the construction of suffix trees. This needs to be done because suffix trees, when used in protein sequence alignment, contain a very large amount of data, which needs to be processed as fast as we can.

HARDWARE DETAILS:

CPU Model: Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz (Model no.: 60)

Memory information:

a) Cache size: 6144 KB

b) Cache alignment: 64

c) Address sizes: 39 bits physical, 48 bits virtual

Number of cores: 4

Compiler: gcc 4.8.2

Optimization flags: Not used

Precision: 64-bit precision supported.

L1d cache: 32K

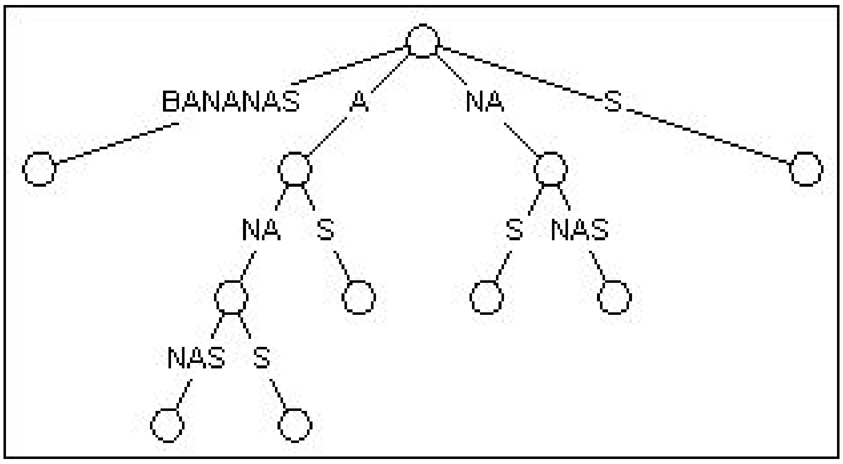
L1i cache: 32K

L2 cache: 256K

L3 cache: 3072K

THEORY:

Suffix trees are compressed tries of all suffixes of a sequence. They also store common prefixes of two suffixes as a common parent. Here, every internal node can have two or more children and every outgoing edge of a node begins with a different character. For example, the suffix tree for the string “BANANAS” looks like this:



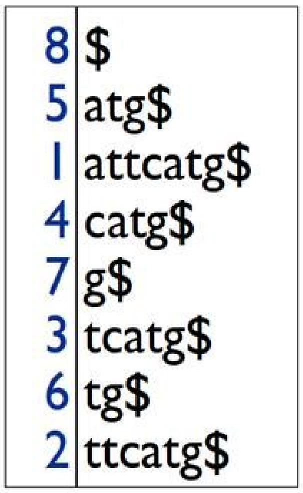
So, now if we are looking for a string “ANA”, we need to go from the root to the edge that starts with the character A, then go to the edge that starts with N, and so on. This is possible because of the rule followed in suffix tree construction, which says that no two children of a node can have edges that begin with the same character. Hence, this is how the data structure called suffix tree helps in sequence matching in proteins, in a time (O(m+n)) which is much smaller than that taken by the naive algorithm (O(m*n^2)) for substring matching.

<u>SERIAL APPROACH TO THE PROBLEM:</u>

Our approach to the problem consists of three steps:

<u>STEP 1: Construction of a suffix array:</u>

Keeping in mind that we need to parallelize the code, we decided to construct the suffix tree using a **suffix array**. A suffix array is an even more compressed version of a suffix tree that stores all the suffixes of a string in a lexicographic order, by storing the starting indices of the suffix in the array.
Example: The suffix array of the string "attcatg$" would be:



8	$
5	atg$
1	attcatg$
4	catg$
7	g$
3	tcatg$
6	tg$
2	ttcatg$

Here is how we implemented the serial construction of the suffix array from a string: Each index of the string is a beginning of a suffix of the string. We implemented a simple bucket sort for these suffixes using an array storing the frequency of appearance of each character in a lexicographic order. So bucket **i** contains suffixes starting with the **i**$^{\text{th}}$ **character** in the string. These buckets are already sorted by first character. Later, they are sent to be sorted individually, where they undergo a 3-way partition quicksort (a modified version of quicksort which implements three partitions, the elements less than, equal to and greater than the chosen pivot). This is a recursive algorithm. Finally, we get a sorted suffix array. *This takes O(nlogn) time.*

<u>STEP 2: Obtaining the LCP array from the suffix array:</u>

The LCP array (Longest Common Prefix) array is an array which stores the maximum common number of initial characters between two consecutive suffixes of the suffix array. We need to obtain the LCP array in order to build a suffix tree (described in later steps).
To build the LCP array, we use Kasai's algorithm. This algorithm uses the ranks of elements in the suffix array, to compare the characters of the string and find out how many maximum common characters those suffixes have. *This takes O(n) time.*

STEP 3: Building a suffix tree from suffix array and LCP array:

For this, we used an algorithm which uses Cartesian trees as an intermediate in building suffix trees, given by Guy E. Blelloch and Julian Shun in their paper[1].
A Cartesian tree is a multiway tree which clusters all the children into one single node and behaves like a binary tree, also satisfying the heap property. The Cartesian Tree is built from the suffix array and LCP array in linear time by a divide and conquer algorithm, which constructs the tree recursively for half the size and then merges it. In our algorithm, to avoid recursion (because it would create a problem in parallelization), we have modified the function to build half Cartesian trees of sizes 2 through n and then to merge them.
From this, the suffix tree is again constructed in linear time, by linking the nodes which have been put into order by the Cartesian tree, with their parents.

SAMPLE INPUT:

Let us take the sample input string as “xabxac”, then,

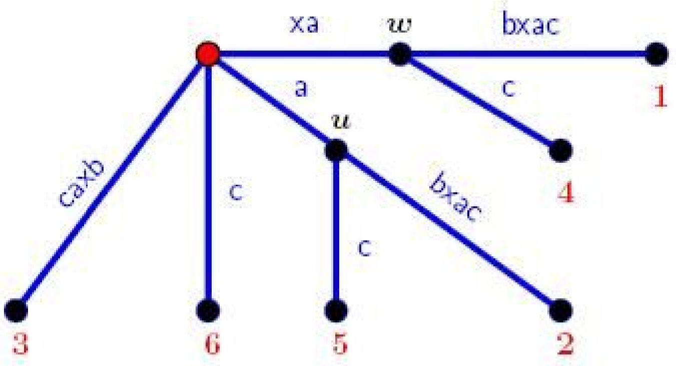
1. Suffix Array formed:

1	4	2	5	0	3

2. LCP Array formed:

1	0	0	0	2	0

3. Suffix tree formed will look like this (our code will return a list of nodes with a root and each node will be connected to its parent and children):



PROFILING:

The result obtained from profiling the serial code is as follows:

```
Each sample counts as 0.01 seconds.
no time accumulated
 %   cumulative   self              self     total
 time   seconds   seconds   calls  Ts/call  Ts/call  name
 0.00          0.00   0.00    19      0.00     0.00  getRoot
 0.00          0.00   0.00           9       0.00     0.00  sub_sort
 0.00          0.00   0.00           1       0.00     0.00  arrayToLCP
 0.00          0.00   0.00           1       0.00     0.00  cartesianTree
 0.00          0.00   0.00           1       0.00     0.00  ss_simple_sort
 0.00          0.00   0.00           1       0.00     0.00  sub_sort_fast
 0.00          0.00   0.00           1       0.00     0.00  suffixArrayToTree

                  Call graph (explanation follows)
granularity: each sample hit covers 2 byte(s) no time propagated
index % time self  children called  name
                  0.00   0.00   19/19          suffixArrayToTree [7]
[1]       0.0     0.00   0.00   19       getRoot [1]
-----------------------------------------------
                  0.00   0.00   9/9            ss_simple_sort [5]
[2]       0.0     0.00   0.00   9        sub_sort [2]
                  0.00   0.00   1/1            sub_sort_fast [6]
-----------------------------------------------
                  0.00   0.00   1/1            main [15]
[3]       0.0     0.00   0.00   1        arrayToLCP [3]
-----------------------------------------------
                  0.00   0.00   1/1            suffixArrayToTree [7]
[4]       0.0     0.00   0.00   1        cartesianTree [4]
-----------------------------------------------
                  0.00   0.00   1/1            main [15]
[5]       0.0     0.00   0.00   1        ss_simple_sort [5]
                  0.00   0.00   9/9            sub_sort [2]
-----------------------------------------------
                  0.00   0.00   1/1            sub_sort [2]
[6]       0.0     0.00   0.00   1        sub_sort_fast [6]
-----------------------------------------------
                  0.00   0.00   1/1            main [15]
[7]       0.0     0.00   0.00   1        suffixArrayToTree [7]
                  0.00   0.00   19/19          getRoot [1]
                  0.00   0.00   1/1            cartesianTree [4]
```

Index by function name

[3] arrayToLCP	[5] ss_simple_sort	[7] suffixArrayToTree
[4] cartesianTree	[2] sub_sort	
[1] getRoot	[6] sub_sort_fast	

Explanation:

The profiling information of the call graph indicates the time spent by each function in execution. As observed in the table, maximum time is spent by the function suffixArrayToTree. This is because, creating a suffix tree involves creating a node (a memory block) using struct and then allocating suffixes and computing from the longest common prefix the nodes of the tree, which further involves 'n' iterations. So, creating a tree with large string sequences, involving corresponding large number of node creation and computational manipulation increases the time spent in that function of conversion from suffix array to suffix tree.

APPROACH USED FOR PARALLELIZATION AND OPTIMIZATION:

We have made various changes in the algorithms used for various parts of the code, to make them more suitable for parallelization:

1. Suffix tree construction is most commonly done by *Ukkonen's algorithm*[2]. But it uses a lot of dependencies. It can construct the next step of the suffix tree **only** based on the previous step, which cannot be parallelized. So we chose to build the tree using a suffix array, which uses less memory and gets constructed in O(n) time.
2. While building a Cartesian tree, the algorithm proposed in the referenced paper[1] uses a recursion based function. Recursion in itself contains inherent dependencies which would cause a problem in efficiently parallelizing the code. So our serial and parallel algorithms do not use recursion but instead, use two loops in place of it.

DESIGN FRAMEWORK AND POINTS OF PARALLELIZATION:

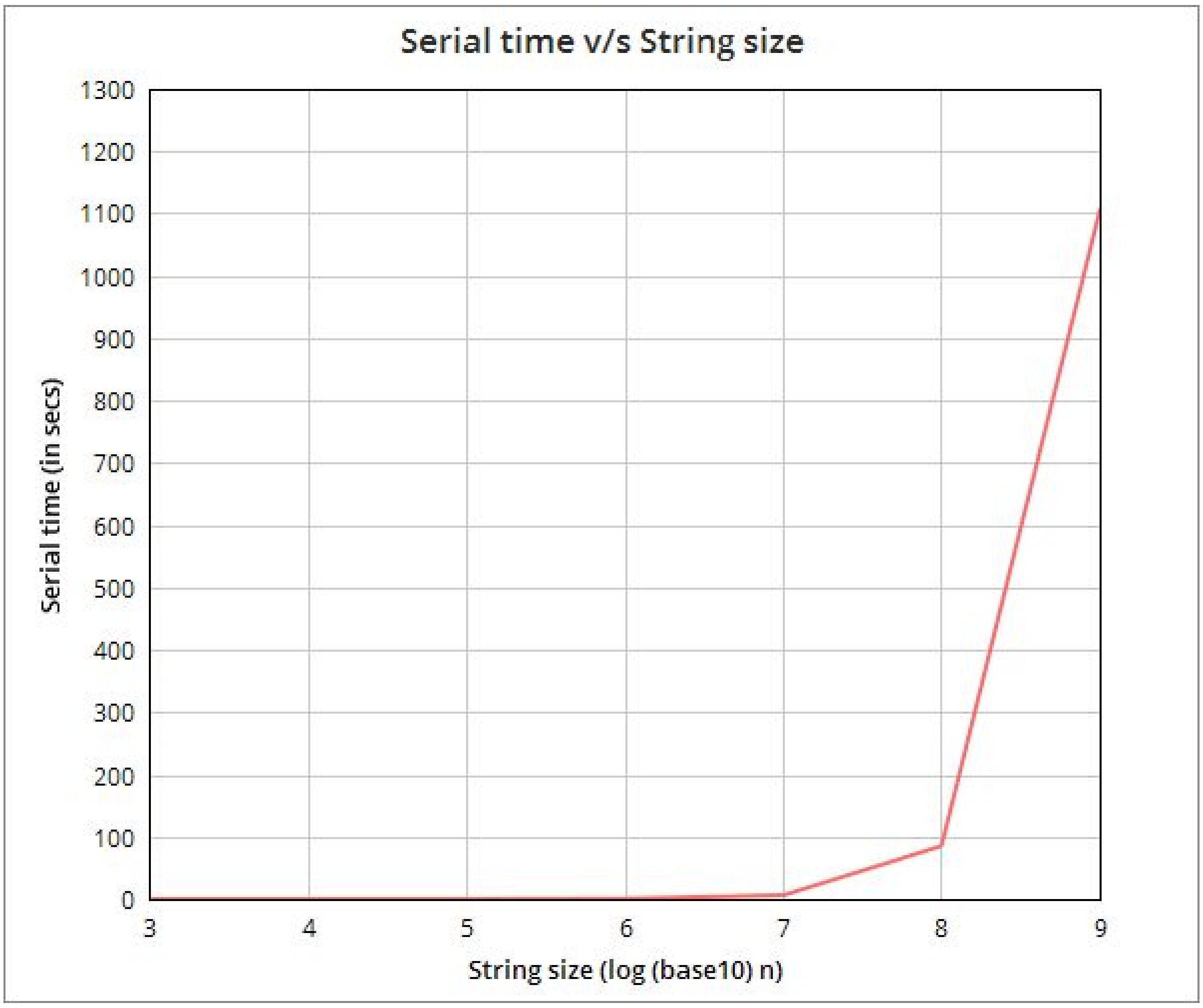
The proposed algorithm of generating suffix trees from suffix arrays exploits the fork and join model effectively.

1. Suffix sorting: The bucket sorting method used in suffix sorting, creates buckets for suffixes starting with the same character. In our parallel code, the optimal number of operations that should be assigned to a particular thread, is calculated, and accordingly, **integral number of buckets** are assigned to

threads keeping in mind whether they have more operations left to be optimal or not. This ensures good **load distribution** between the threads. Also, since the suffix array initially stores together all the suffixes which should be in the same bucket, so when a thread sorts a bucket of a character, for example ‘a’, then all the suffixes to be sorted by it are stored consecutively in the suffix array. This ensures efficient **cache management**, because once the thread accesses the first suffix, it gets the next few suffixes that it needs, in its cache. Thus, temporal locality is improved and **memory access** is minimized as much as possible. *(The threads fork after the buckets are created for suffixes starting with the same character, and they join after their respective buckets have been sorted.)*

2. <u>LCP Construction</u>: The ranks of various suffixes need to be calculated which can be distributed among different threads. *(The threads fork when ranks of various suffixes have to be calculated and join when the entire rank array has been calculated.)*
3. <u>Cartesian Tree Construction</u>: In Cartesian Tree construction, the trees are constructed for step sizes of the order of 2^x. For each step size, the tree construction is divided among the threads. Each thread constructs a Cartesian tree for some starting point and then merges it with a tree which was built in one of the previous stages (lesser step size). So all threads **compulsorily** have to get synchronized before the next stage starts, which is why the inner loop is parallelized with an omp for directive, which ends before the next iteration of step size. *(Forking happens inside a loop that creates various step sizes for construction of tree from the array. The threads join after one particular phase for a step size is done.)*
4. <u>Suffix Tree Construction</u>: In the suffix tree construction, the loop which assigns values and parents to all the nodes based on the suffix array and LCP array, is distributed among the threads. *(When values and links need to be assigned to nodes according to the suffix array and LCP array, the threads fork. They join after each node has these values and links.)*
5. Random allocation of characters to form the string can be done parallelly. *(The threads fork to assign randomized characters to the string. They join after this has been done.)*

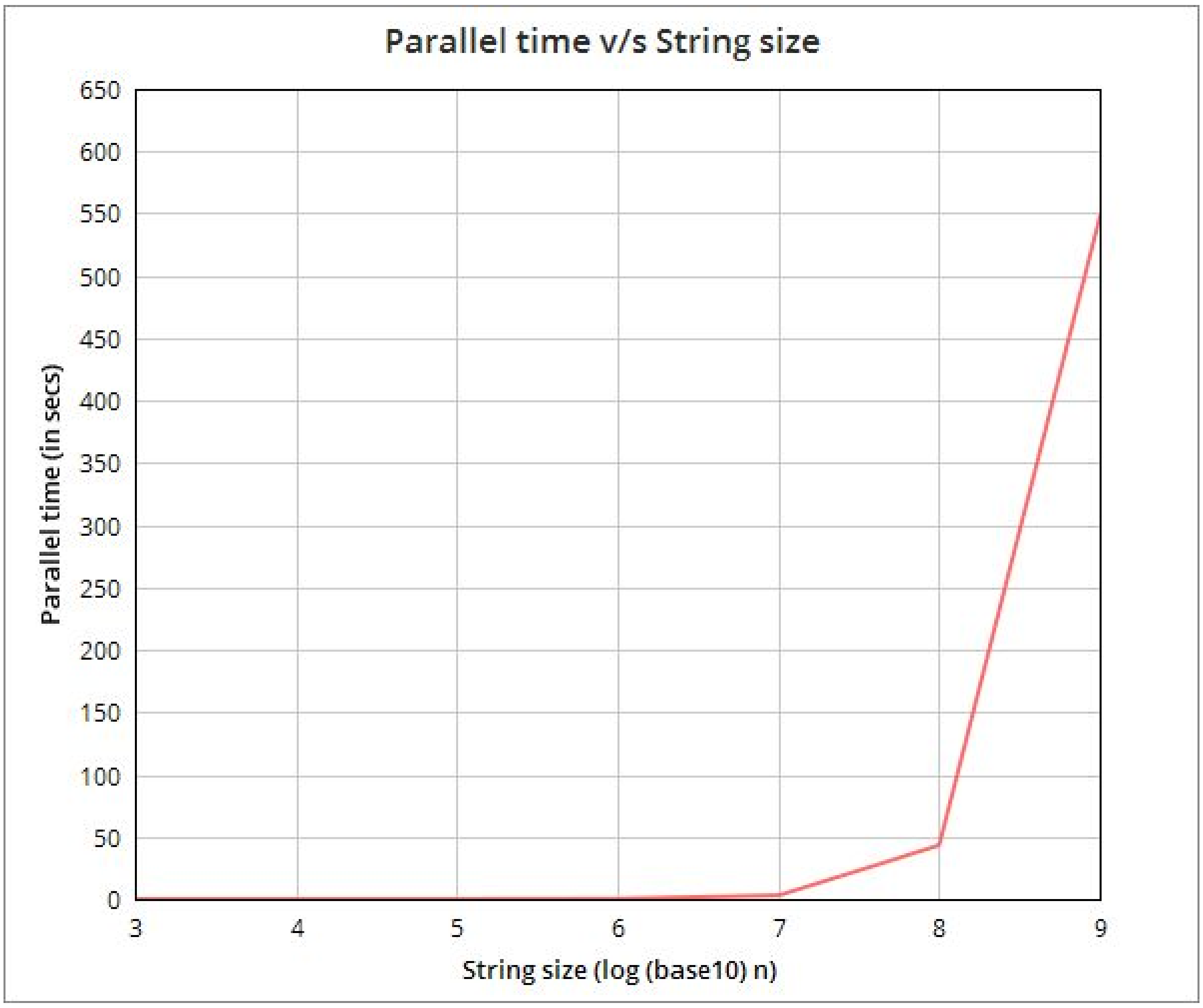
SERIAL TIME V/S SIZE OF THE PROBLEM:



Explanation:

The algorithmic complexity of the serial code is O(n*logn), as shown above, where n is the problem size. Therefore, as the problem size increases, the number of characters in the string increases and the number of nodes in the tree increases. Thus, the serial time for execution also increases by a factor which is more than linear.

PARALLEL TIME V/S SIZE OF THE PROBLEM (FOR 4 CORES):



Explanation:

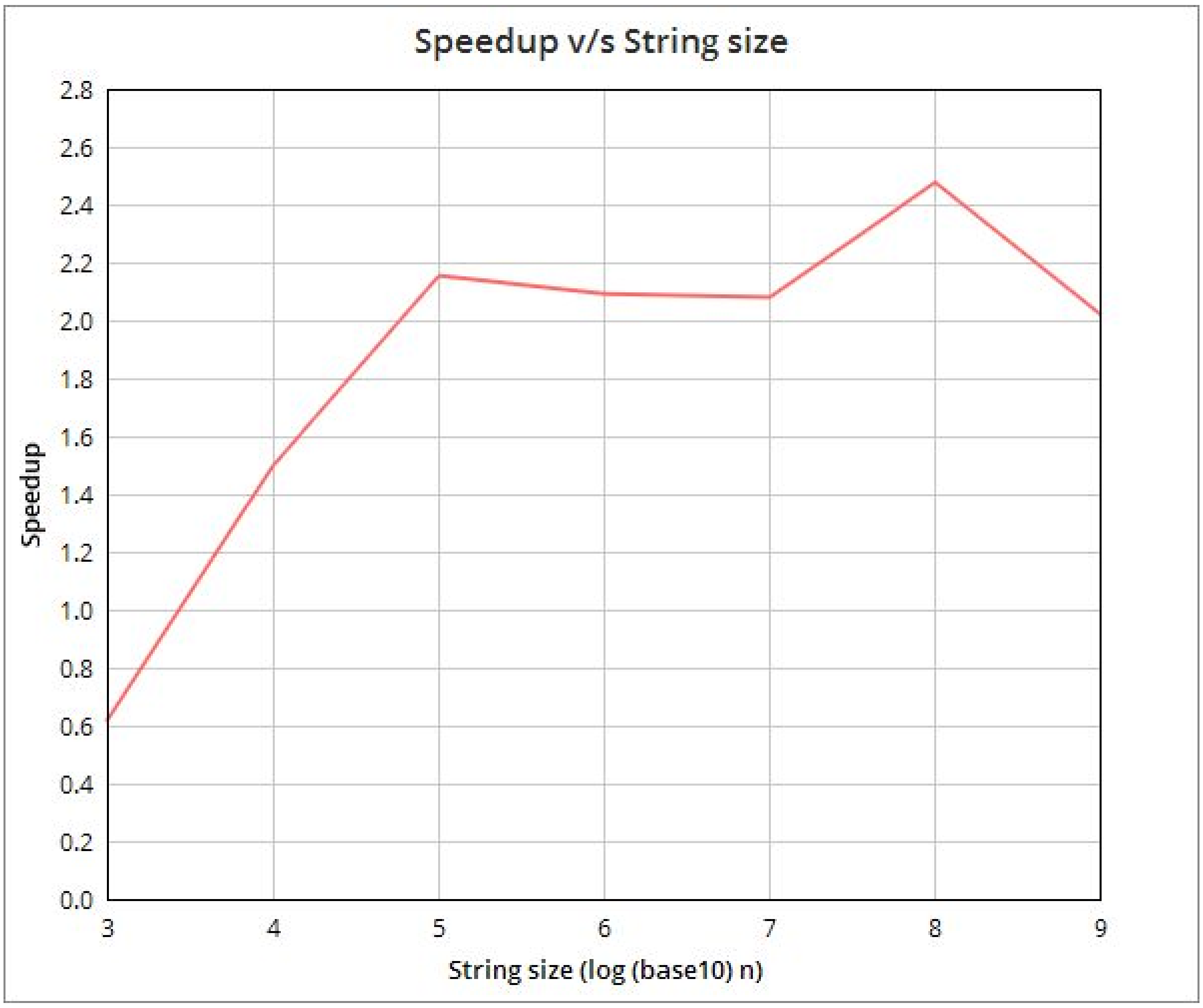
The execution time increases with the increase in problem size, but it is significantly lower than that of its corresponding serial code graph. The optimization and parallelization techniques that we have used in LCP construction and cartesian tree construction reduce the parallel overhead for large problem sizes and give an optimised result.

Parallel overhead time:

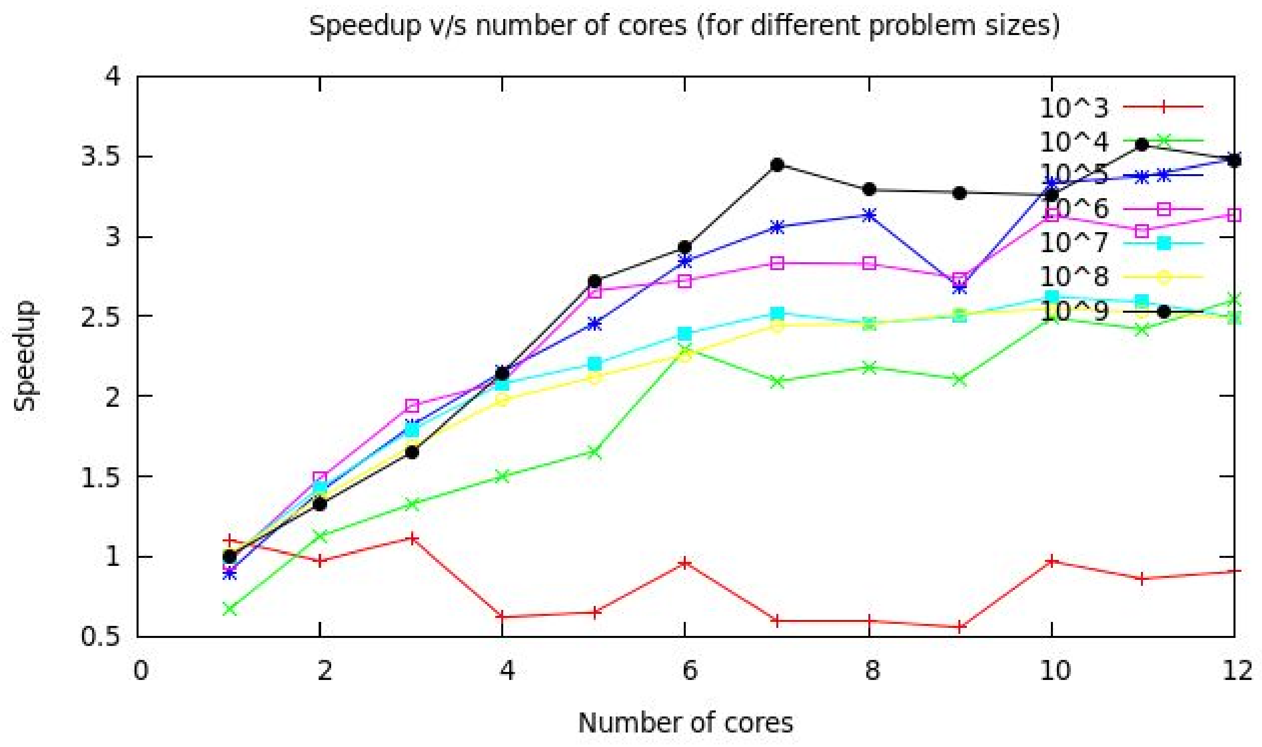
The parallel overhead time is defined as (time taken by parallel code for one core-time taken by serial code).

Here the parallel overhead is 1177.698855-1176.287346=1.411509 seconds for problem size = 10^9.

SPEEDUP V/S SIZE OF PROBLEM (FOR 4 CORES):



SPEEDUP V/S NUMBER OF CORES:



SPEEDUP ANALYSIS SUMMARY:

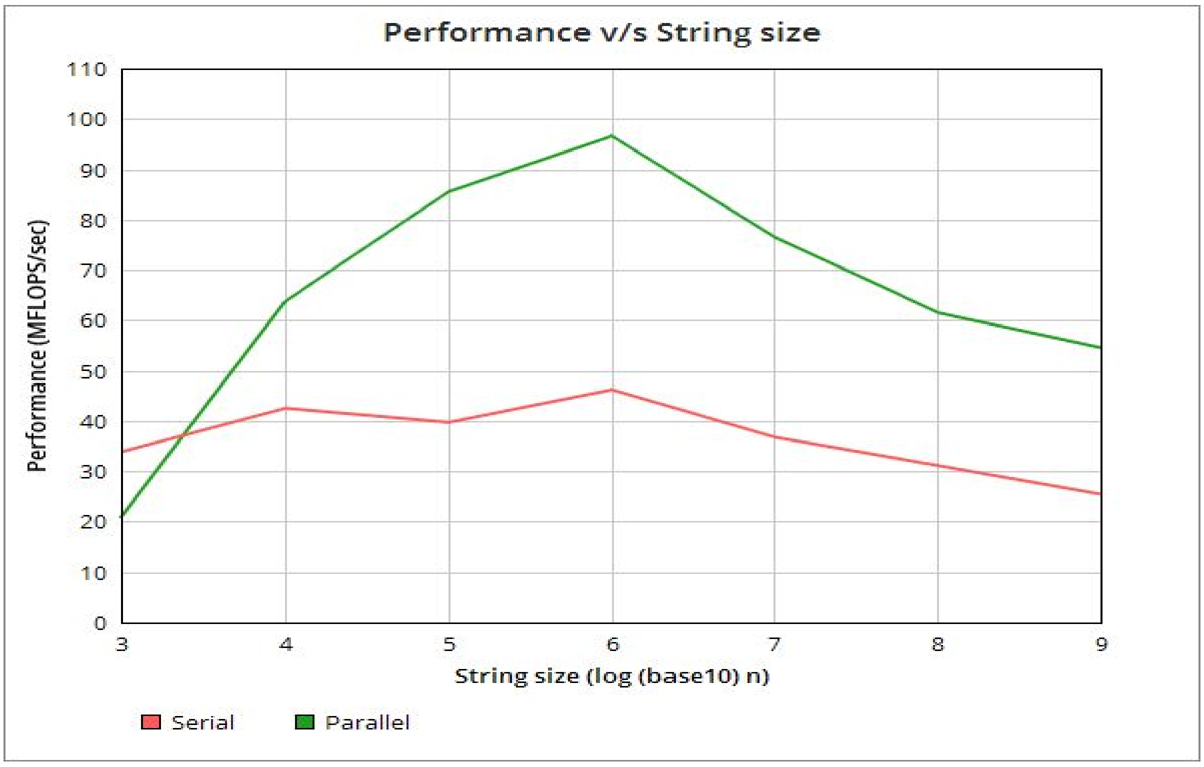
The speedup of the construction of the suffix trees using suffix arrays though is significantly better than its corresponding serial implementation, yet it is not very impressive due to the following reasons:

1. *Gustafson Barsis Law:* This law takes into account the communication overhead between the threads caused by increasing problem sizes. Due to this increase in communication overhead, the speedup decreases.
2. *Amdahl's Law:* This law considers the fraction of a code that has been parallelized. In our code, the fraction of parallelization is less than 75%. This is due to many computations in the code which have a lot of dependencies in them and cannot be parallelized (for example, 3 way quicksort, merging of two parts of a Cartesian tree etc). So according to Amdahl's law, the ideal speedup that we can get is itself lower than the number of processors, which is another reason for less speedup.
3. *Fork join model* uses heavy weight threads due to large string sequences and large number of computations in terms of suffix manipulation and so the communication overhead between these threads increases and hence, the speedup decreases according to Gustafson Barsis law.
4. The problem is *coarsely granular* for large problem sizes as the individual processing units have a lot of load in terms of problem size which is large in case of genetic sequences as well as computation also. So, there is load

imbalance as some processor will tend to do more work than others. This also results in poor speedup.

5. If the number of cores (p) is appropriately chosen, the algorithm exhibits good scaling with number of processors. The size of the bucket array B grows exponentially with the problem size. The poor performance of the program when p=8, particularly for problem sizes 10^4, 10^5 and 10^6 is because optimal data proportion assigned to each processor is too large when compared to the data size. In fact, for this choice of parameters, the bucket size on a processor is not in proportion, despite of greater capacity than the size of the suffix array allocated to it. While the choice of p dictates the exact number of buckets, the number of suffixes falling into respective buckets is completely dependent upon the input data. So, increasing number of processors after an optimum value gives poor speedup.
6. A suffix tree is fundamentally a compact index of all substrings of a given text. While being asymptotically linear in the size of the input, in reality, suffix trees can easily be 50 times larger than the input. As such, suffix trees often exceed typical main memory sizes, even when the input does not. As most existing algorithms are designed for RAM, their performance severely degrades when the tree and/or input do not fit in the main memory. Hence, we get a poor speedup.

<u>PERFORMANCE V/S PROBLEM SIZE:</u>



Explanation:
With increasing problem size, the performance initially increases as the problem exhibits optimal load balance and is finely granular which results in efficient parallelism. But with increasing problem sizes, the communication overhead occurs and so the granularity increases resulting in poor performance. Moreover, since the algorithm uses fork and join model, communication overhead between threads plays an important role in the performance of the code. Since, in the fork join model in heavy-weight threads (heavy due to large problem sizes and computation) communication overhead increases, the performance declines.

GRANULARITY:

The construction of suffix arrays and suffix trees in the proposed project is a coarsely granular problem for large datasets, which is actually the case for biotic protein and genome sequences. It is so because granularity considers the communication overhead between multiple processors or processing elements. Since each of the processing elements in the case of creation of suffix arrays as well as suffix trees requires large number of computations owing not just due to massive datasets but also due to manipulation of suffixes in LCP, particularly in the Cartesian Tree generation. The program is split into large tasks and because of this large amount of computation takes place in the processors. This might result in slight load imbalance as some processors tend to do bulk of work while some remain idle. The advantage that we get is low synchronization overhead and low communication bottleneck. A way in which this limitation can be overcome is the use of MIMD machines which increases communication between the processing elements and hence reduces granularity.

FUTURE SCOPE OF IMPROVEMENT

The possible future improvements in this application are:

- Using some form of compressed structure which would not occupy as much memory and space as the suffix tree.
- Decreasing the dependencies in the code so that more fraction of the code can be parallelized.

REFERENCES:

[1] “A Simple Parallel Cartesian Tree Algorithm and its Application to Suffix Tree Construction” by Guy E. Blelloch and Julian Shun (https://www.cs.cmu.edu/~guyb/papers/BS11.pdf)
[2] “On-Line construction of suffix Trees” by Ukkonen (https://www.cs.helsinki.fi/u/ukkonen/SuffixT1withFigs.pdf)
[3] “Suffix trees for very large strings” by Marina Barsky, Ph.D. University of Illinois at Urbana-Champaign.
[4] “Parallel Suffix Sorting” by Natsuhiko Futamura, Srinivas Aluru, Stefan Kurtz School of EECS Dept. of ECpE Technische Fakultat Syracuse University Iowa State University Universitat Bielefeld Syracuse, NY, USA Ames, IA, USA Bielefeld, Germany respectively.