A DYNAMIC APPROACH TO WEIGHTED SUFFIX TREE CONSTRUCTION ALGORITHM

Binay Kumar Pandey[1], Rajdeep Niyogi[1], and Ankush Mittal[2]
[1]Electronics and Computer Engineering Department,
Indian Institute of Technology
Roorkee, India
(binaydec, rajdpfec)@iitr.ernet.in
[2]Computer Science and Engineering Department
College of Engineering Roorkee
Roorkee, India
dr.ankush.mittal@gmail.com

ABSTRACT

In present time weighted suffix tree is consider as a one of the most important existing data structure used for analyzing molecular weighted sequence. Although a static partitioning based parallel algorithm existed for the construction of weighted suffix tree, but for very long weighted DNA sequences it takes significant amount of time. However, in our implementation of dynamic partition based parallel weighted suffix tree construction algorithm on cluster computing makes it possible to significantly accelerate the construction of weighted suffix tree.

KEYWORDS

Weighted Suffix Tree, Cluster, CUDA, MPI

1. INTRODUCTION

The molecular weighted sequence shown in (Figure. 1) is found in the numerous applications of computational molecular biology [1] and it is defined as sequence of either nucleotides or amino acids, where each character in every position is assigned a certain weight. In computational biology few important biological processes such as DNA assembly process [12], pattern matching, and identification of repeated patterns in biological weighted sequences are modeled by molecular weighted sequences and also very help full in the translation of gene expression and regulation. Therefore, a requirement of efficient algorithm is arises, in order to analyze molecular weighted sequences.

In our study we found that, a suffix tree [2] is play an important in sequence analysis problems but in case to handle weighted sequences a related data structure weighted suffix tree has been recently proposed in [3]. The weighted suffix tree can be deliberated as a general form of common suffix tree to handle weighted sequences. This constructed data structure holds all the sequence manipulation properties of the normal suffix tree. Although weighted suffix tree are more suitable for molecular weighted sequences analysis and an efficient algorithm are exists for their construction, but it takes a lot of processing time in case of long weighted sequences. As result of this, it required to conceive some new techniques which help to reduce the execution time of this algorithm. In the construction of weighted suffix tree algorithms using an efficient algorithm mentioned in [4], we are parallelizing only the construction part of this algorithm. There are several sequential linear-time algorithms are

DOI : 10.5121/ijdps.2011.2103

exists [5-6], which may apply to make the construction part of weighted suffix tree algorithm. Among them, McCreight's algorithm is widely used; it constructs a suffix tree by making use of suffix links. Unfortunately, the parallel implementation of this algorithm is become impractical for constructing large-size suffix trees because of high memory overheads due to suffix links. So in this paper, we parallelize the construction part of weighted suffix tree algorithm without using suffix links on CUDA [7] and Cluster computing. In parallelizing the algorithm, we apply static partition based approach in CUDA and static as well as dynamic partitioning based approach in Cluster computing. Our whole paper is organized into the following sections. Section 2 gives brief explanations of weighted suffix tree. Section 3 gives a study on mccreight's: a suffix tree construction algorithm. Section 4 consists of brief description of modified mccreight's suffix tree algorithm. Section 5 gives detail of parallel architecture used for implementation. Section 6 is dedicated to parallelization approach for weighted suffix tree construction. Section 7 gives detailed of implementation details. Section 8 dictates result and discussion. In section 9 final conclusion of our experiment is given.

Positions	1	2	3	4	5	6
	A(1)	G(1)	T(1)	T(1)	A (0.3) C(0.2) G(0.3) T(0.2)	C(1)

Figure 1. Example of a weighted sequence**.**

2. THE WEIGHTED SUFFIX TREE

The weighted suffix tree may be termed as successor data structure of suffix tree, because it can be seen as suffix tree with incorporating the notion of probability of appearance for every suffix stored in a leaf. An informal definition of weighted suffix tree is given in [4].

2.1. Construction of Weighted Suffix Tree

The weighted suffix tree for a given weighted $X=X[1]X[2]...X[n]$, of length n can be built by following the steps given below.

Step 1: For each position weighted sequence denoted by i ($1 \leq i \leq n$), all possible weighted suffixes of the weighted sequence with probability of appearance greater than 1/k are generated.

Step 2: Now construction of generalized suffix tree, for the set of all possible weighted suffixes are performed. An example of weighted suffix tree is shown in (Figure. 2) for weighted sequence is shown in (Figure. 1).

The steps mentioned above can be broken into three phases that are **Coloring**, **Generation** and **Construction** for the development of efficient weighted suffix tree construction algorithm [4].

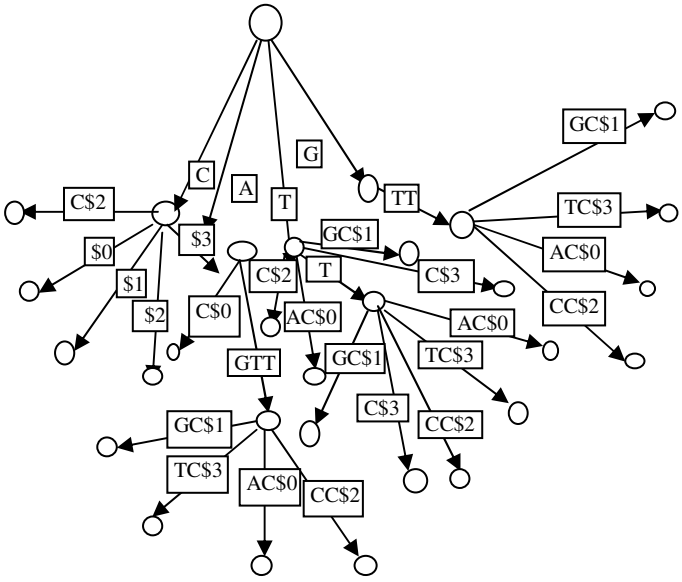


Figure 2. Example of a weighted suffix tree for weighted sequence, shown in (Figure 1) with thresold value θ = 0.1

Coloring: Scan all the positions *i* (*1*≤ *i*≤ *n*) of the weighted sequence and mark each one according to the following criteria:

- Mark position *i* black, if none of the possible characters, listed at position *i*, has probability of appearance greater than *1-1/k*.
- Mark position *i* gray, if one of the possible characters listed at position, has probability of appearance greater than *1-1/k*.
- Mark position i white, if one of the possible characters has probability of appearance equal to *1-1/k*.

Generation: After the coloring phase, the generation phase begins having as input the weighted sequence, the colors' array and the list B. At this phase all the substrings that satisfy the probability of appearance constraint are generated.

In the current phase all the positions in list B are scanned from left to right. At each black position i, a list of possible sub-words starting from this position is created.

The production of the possible sub-words is done as follows: moving rightwards, we extend the current sub-words by adding the same single character whenever we encounter a white or gray position, only one possible choice, and creating new sub-words at black positions, where potentially many choices are provided.

Construction: Having produced all the subsequences from every black position, we insert the subsequence in the generalized suffix tree. This is the part of WST construction which we have parallelized using our modified McCreight's suffix tree construction algorithm. The detailed description of McCrieght's algorithm for suffix tree construction is given in next section

3. MCCREIGHT'S: A SUFFIX TREE CONSTRUCTION ALGORITHM

McCreight's Suffix Tree Construction algorithm is at the core of WST construction as it's a generalized suffix tree of all the solid patterns of a molecular weighted sequence. Though

McCreight's algorithm is meant for constructing suffix Tree, it can be easily modified to work for generalized Suffix Tree also. McCreight's is an *O(n)*, in-memory construction algorithm based on the clever observation that constructing the suffix tree can be performed by iteratively expanding the leaves of a partially constructed suffix tree.

```
Input: String S = α$, where α∈∑, ∑is a finite alphabet.
Output: Suffix Tree(S)

Construct tree for x[1..n]

for i = 1 to n do
if head(i)=ε then
head(i+1) = scan(ε, s(tail(i)))
add i+1 and head(i+1) as node if necessary
continue
u = parent(head(i)) ; v = label(u, head(i))
if u≠ε then w = scan(s(u), v)
else          w = scan(ε, v[2..|v|])
calculate_head(w, head(i+1), tail(i));      //jump to function
                                                    process_w
s(head(i)) = w
add leaf i+1 and edge between head(i+1) and i+1
end_for

calculate_head(w, head(i+1), tail)
if w is an edge then
add a node for w
head(i+1) = w
else if w is a node then
head(i+1) = scan(w, tail)
add head(i+1) as node if necessary
```

Figure 3. Pseudocode of serial McCreight's algorithm

Through the use of suffix links, which provide a mechanism for quickly traversing across sub-trees, the suffix tree can be expanded by simply adding the i+1 character to the leaves of the suffix tree built on the previous i characters.

The algorithm thus relies on suffix links to traverse through all of the sub-trees in the main tree, expanding the outer edges for each input character. We will first understand how McCreight's algorithm works then will see the modification proposed for parallelizing the algorithm. The algorithm can be formally summed up as follows:

- Let head(i) denote the Longest Common Prefix (LCP) of x[i..n]$ and x[j..n]$ for all j < i.
- Let tail(i) be the string such that x[i..n]$=head(i)tail(i).
- Let s(suff) donate the suffix link of a suffix suff.
- Function scan(x,suff) scans the suffix suff in the tree starting from node x.
- Iteration i in McCreight's algorithm consist of finding (or inserting) the node for head(i) and appending tail(i).

The detailed psuedocode for serial McCreight's algorithm is given in (Figure 3), in order to develop better understanding an example of suffix tree construction for possible sequences of weighted sequence is given in (Figure 4).

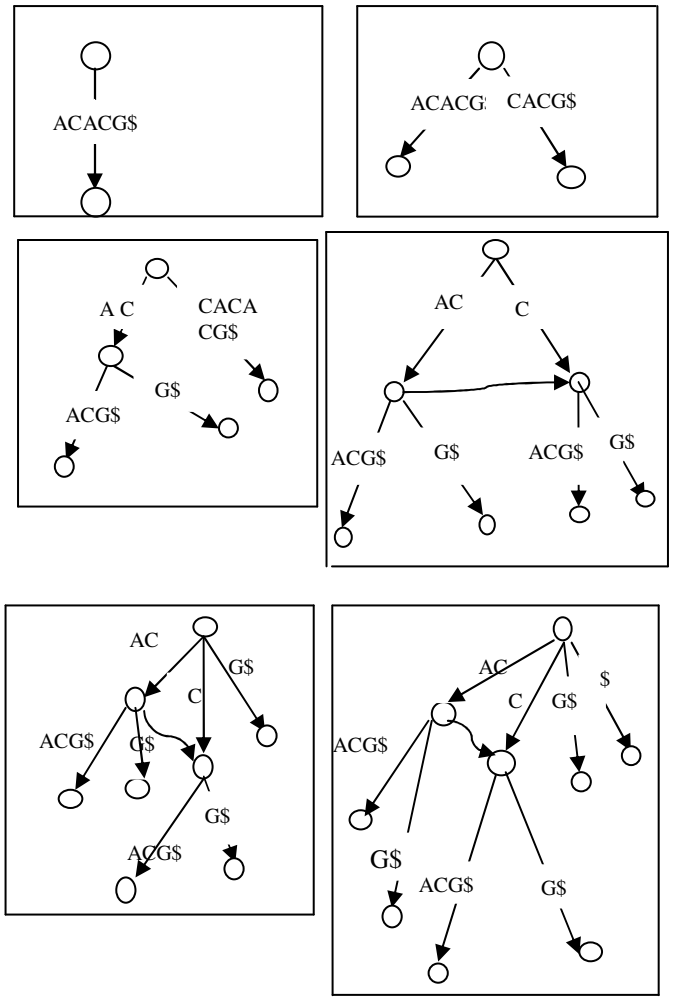


Figure 4. An example of suffix tree construction for possible sequence ACACG$

4. MODIFIED MCCREIGHT'S SUFFIX TREE ALGORITHM

McCreight's Suffix Tree construction algorithm has been modified for parallelization during the work. We are incorporating independent subtree construction strategy. In which four subtrees will be created independently as shown in (Figure 6). There was a problem in applying the McCreight's algorithm directly both in case of CUDA and clusters. As CUDA is shared memory architecture it's theoretically possible to apply McCreight's algorithm as such. But in that case also we can't be sure that a suffix link exists and we may have to wait for other thread to create the suffix link. This waiting will affect the parallelized construction of the tree. In case of clusters it's actually not possible to have suffix links as all subtrees will be created by different nodes in their local memory.

So there were two ways of constructing the tree. First using the naive method without any suffix link with a worst case time complexity of $O(n^2)$. Second is to modifying the suffix link strategy so that it can be incorporated in sub-tree construction. Authors [19] have used a similar approach for subtree construction by modifying Ukonnen approach. McCreight's

algorithm has been modified during this work to accomplish the same goal. To understand the modification we will first understand how McCreight's algorithm uses the suffix link strategy for fast suffix tree construction. McCreight's algorithm does the following tasks during i^{th} iteration:

- Find head (i+1).
- Find and set suffix link for head(i).
- Insert the $(i+1)^{th}$ suffix in the tree using suffix link.

Using these suffix links only McCreight's algorithm facilitates direct jumping to required node without the need of traversing from root each time to insert a new suffix.The problem in incorporating the same strategy with subtree construction is that the next suffix doesn't necessarily start from the next index in case of subtree suffices. McCreight's take advantage of the property of string that head(i)[*2...n*] will prefix head(i+1) in all cases, which might not be true during subtree construction. So, we have modified the strategy by incorporating some checks during an insertion. The pseudocode for modified McCreight's algorithm is shown in (Figure 5).

```
Input: String S = α$, where α∈∑, ∑ is a finite alphabet.

Output: Sub_Tree(S, c)

 n=number_valid_suffix(S, c)

    for i=1 to n do
     pos[i]=position of ith suffix starting with char c
    end_for

      pos[1]= position of first suffix starting with char c
       Construct tree for x[pos[1]...n]

        for i=1 to n-1 do
        valid_index=pos[i]
        next_valid_index=pos[i+1];

       if head(i)=ε then
        head(next_valid_index) = scan(ε,    next_suffix)
add next_valid_index and head(next_valid_index) as node if necessary
continue
u = parent(head(i)) ; v = label(u, head(i))
w=first;
if u≠ε then
if next_valid_index==(valid_index+1) then
w = scan(s(u), v)
calculate_head(w, head(next_valid_index),
 next_valid_suffix- w.path_label)
else if Matches(u.suffix_link.path_label,
next_valid_suffix)
then
w=scan(u.suffix_link, next_valid_suffix- u.suffix_link.pah_label)
head(next_valid_index)=w;
else  temp_str= comman_prefix(v(2..n), next_suffix)
w=scan(ε, temp_str)
head(next_valid_index)=scan(w, next_suffix-temp_str)
s(head(i)) = w
add leaf next_valid_index
end_for
```

Figure 5. Modified algorithm of McCriet algorithm for parallel construction of sub-suffix tree of possible sequence suffixes started from character "A" ,
"C", "G", and "T".

Whenever we will jump to a node using suffix link, we will check the path label of the node. We will compare this path label with next suffix, only in those cases when path label matches the next suffix we will insert the next suffix using McCreight's strategy, otherwise we will travel from root. Similar modifications has been done to find head(next_suffix). Our algorithm does following tasks during i^{th} iteration:

- Find head(next_index). Which is not necessarily the $(i+1)^{th}$ index.
- find and set suffix link for head(i)
- Insert the $(next\_index)^{th}$ suffix in the tree using suffix link.

Algorithmic complexity of out modified algorithm is same as McCreight's algorithm i.e. $O(n)$ if we will take string comparison as a single unit operation. As these are the only extra calculation we are doing in comparison to McCreight's during some of the iterations. But even if we don't take this assumption, in which case the complexity is $O(n^2)$, our algorithm will run faster than naive method with same complexity.

The reason is that we are replacing naive algorithm's tree traversal with string comparison, which is a very fast cache efficient operation on any modern processor than traversing the tree. The cases when next suffix start with next index are treated as special cases and simple McCreight's algorithm is used for further speedup.

The authors [20] have also reported that a cache efficient algorithm with $O(n^2)$ worst case complexity can outperform the popular O(n) Ukkonen Suffix Tree construction algorithm in some cases. The modified algorithm can be formally summed up as follows; the function and variable name are self explaining**:**

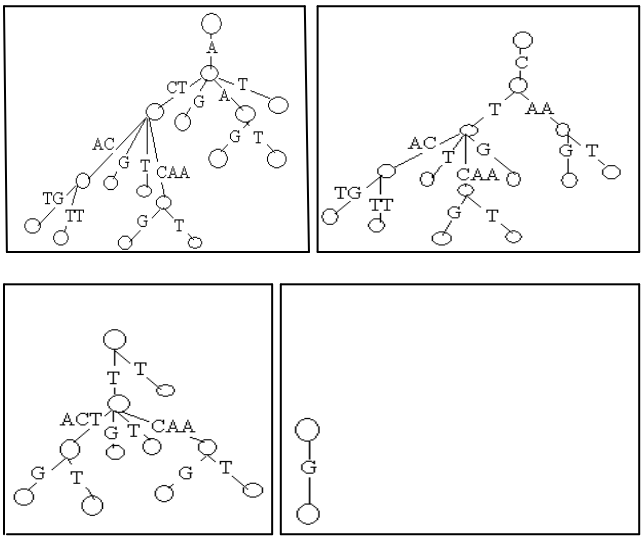


Figure 6. A parallel construction of weighted suffix trees for suffixes of valid sequences starting from symbol A, C, G and T.

5. PARALLEL ARCHITECTURE USED FOR IMPLEMENTATION

In our implementation of weighted suffix tree construction algorithms on parallel architectures, we used nvidia CUDA, and Cluster computing. In CUDA only static partitioning approach is applied for the construction of weighted suffix tree, while in Cluster computing both static and dynamic partitioning approach is applied for the construction of weighted suffix tree. The detailed descriptions of these parallel architectures are given below:-

5.1. Compute Unified Device Architecture (CUDA)

CUDA also known as compute unified device architecture is a new multi-core architecture used for performing computations on the GPU as a data-parallel computing device [7]. It is acted as a middle-ware to application software for transparently scales up its parallelism on GPU. The main concepts of CUDA are basically correlated to hierarchy of thread groups, shared memories, and barrier synchronization. The hierarchy of thread present in the architecture of CUDA allows application developer to assign his work in a analogous hierarchy, where coarse sub-program can be executed independently without using share memory and finer sub-programs can be executed in parallel using shared memory. When user develops application on CUDA, the GPU portion of CUDA is seen as a compute device that is capable of executing a large number of threads in parallel. It acts as a coprocessor to the main CPU [7]. The GPU and CPU present in CUDA are also known by other names that are device and host.

In CUDA there is a function named kernel is executed on the device and this function is capable to accesses only the memory present on the device. Moreover, this kernel is executed in the manner of single program multiple data (SPMD) model, in other words it is also called as a single program multiple thread (SPMT) model, which means that the number of thread configured by a user execute the same application. These user-defined numbers of threads ($\leq$512) are grouped together and formed thread blocks. In addition to this, all threads present within the same block can be synchronized by a barrier-like construct, called __syncthreads().

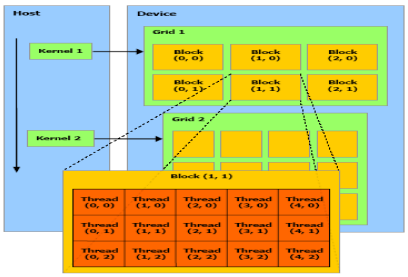


Figure 7. Thread batching in CUDA

The different number of blocks are grouped together to form a grid. Normally, a grid is made-up of 2^{32} blocks, so the total number of thread present in grid is 2^{41}. It is not possible to synchronize blocks within a grid [10]. In the (Figure 7) the thread hierarchy is shown. In first level a grid is present and inside it a block containing two dimensional parameters resides, and each block contains, a thread block which holds a three dimensional parameters that defines the number of threads. The main reason behind to arrange a thread block in a three dimensional way is that, the first implementation of the SDK was thought to be running two or three dimensional application.

5.2. Cluster computing using Message Passing Interface (MPI)

Cluster computing play important role to satisfy the continual demand of high computational speed required for some specific highly data intensive applications like weather forecasting, manufacturing applications, engineering calculations and simulations. In Cluster computing, to gain high computational speed, a large number of multiple processors are used to execute the application.

The application is split into parts and each part is executed by a separate processor in parallel. When the multiple processors work in parallel, it requires an interface to communicate with each other. Message passing interface also known as MPI is a programming paradigm that are widely used on certain classes of parallel and distributed computing environments [11]. Although there are a number of variations exist in message passing, but the basic concept by which processes are communicating through messages is same. The message passing interface is a library used by multiple processors to send messages back and forth using send and receive application processing interface. This approach gives a significant improvement in performance.

In the present time, performance, scalability and portability are the main goal of message passing interface. These features build message passing interface as the most dominant model used in high performance computing and also considered as a standard for communication between different processors both for shared memory and distributed memory. The message passing interface MPI-1 is the standard for traditional message-passing using shared memory between different processors, whereas MPI-2 is standard for remote memory, with parallel input/output and dynamic processing using distributed memory for different processors. The standardization of message passing interface helps users by providing portability and ease-of-use, as well as, it also helps vendors and developers by clearly defined core set of routines. So that, they are able to efficiently implement parallel program.

6. PARALLELIZATION APPROACH FOR WEIGHTED SUFFIX TREE CONSTRUCTION

The parallelizing strategy describe in [9] is a static partitioning approach for weighted suffix tree construction. According to this strategy the number of thread used in per processes is equal to four and remain fixed on it, independent of the input sequence size and properties. This is the main reason behind the inefficient use of available resources as well as un-even division of work among the processes.

Therefore a dynamic partitioning mechanism has been designed to fully exploit the available resources as per requirement. This dynamic approach is basically divides the work between processes depending upon the size and shape of the input, where shape indicated the percentage of appearances of different character in the input sequence. The dynamic partition is further classified into two types.

- Deciding depth of parallelization
- Deciding width of parallelization

6.1. Depth of parallelization

The meaning of depth of parallelization is the tree depth level from which the construction of sub-tree begins. For instance in (Figure 8) a depth level 0 reveals only single sub-tree will be constructed at that depth, and once depth is increase from 0 to 1. The construction of single

sub-tree problem is divided into the construction of four sub-trees. But one flaw that we are encountered in this our strategy is that it is not possible to determine the number of processes that will be required to build a particular sub-tree.

The depth level is determined on the basis of input sequence size and number of available resources. Such as for a larger input weighted sequence. The construction of weighted suffix tree is break into the construction of many sub-tree, so that a large number of threads can be used for construction and to do so depth level is increased. On the other side, in our observation we found that, in the case of very high depth level for a relatively smaller input weighted sequence. The best possible resources utilization and reduction of construction is not overcome overhead time. Therefore, we adopt a try and error based approach to determine the optimum depth of parallelization, and for this we took a file containing the range of sequence length and the record the construction time of weighted suffix tree for various depth levels.

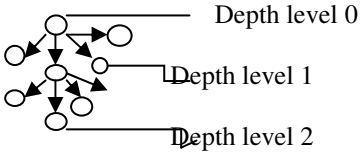


Figure 8 Depth of parallelization for construction of weighted suffix tree**.**

6.2. Width of parallelization

The meaning of width parallelization is to determine the number of processes required to make a particular weighted sub-tree. This number of assigned processes can be varies for each weighted sub-tree and it also depend upon input weighted sequence. Each particular weighted sub-tree will have different width of parallelization. This width of parallelization is being decided by the number of common prefix of the entire suffixes that are used to makes that particular weighted sub-tree.

For better understanding let us consider an example to determine the width of each sub-tree made for sequence T, and let suppose the level of depth is 2. In that case, to find the width of parallelization we compute the all occurrence of prefixes having length 2 such as AA, AC.....,TT, and assumed that

The maximum limit for the number of processes which can be assigned to a particular weighted sub-tree is M. Now a single process is allocated to the weighted suffix sub-tree with minimum number of suffices and M number of process is allocated to the weighted sub-tree with maximum number of suffices. All other remaining weighted sub-trees are allocated to the number of processes exists between 1 and M. The approximate number of process for the construction of each weighted suffix tree is mapped linearly using the following formula:

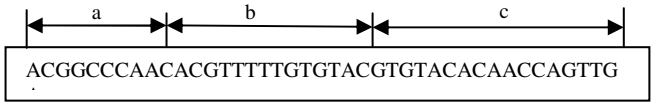
Number_of_ Process= (int) (((N - minV) / (maxV - minV)* (M-1)) + 1)

where Number_of_Process is the number of processes that will be use for the construction of particular weighted suffix sub-tree, N is the number of suffix in that particular weighted suffix sub-tree, minV, and maxV are respectively the minimum and maximum number of suffices in any weighted suffix sub-tree where minV $\neq$ maxV. Where, M is the maximum number of processes that can be assigned to any particular sub-tree construction.

Once the number of processes for a particular sub-tree is decided, we have to decide how the work will be divided among these processes. One of the very basic way is to assign equal number of suffices to each process. This strategy seems to be perfectly normal for implementation performs on a homogeneous cluster, where all nodes have same computational

speed. But normally all suffices are not same length. So a different strategy is designed for the uniform allocation of work among the process.

This strategy is elaborated by considering an example in which we consider one of the possible sequence T of length n generated from weighted sequence, and also suppose total three processes are used to make sub-tree for the sequence. In this approach, rather than giving equal suffices, we will assign first "a" suffices to process 1, and then next "b" , "c" suffices to process 2 and 3 respectively. One important fact to be consider here is that the insertion of a suffices in a tree is proportional to its length, using this fact we calculate the total time taken to insert first "c" number of suffices which is denoted by T_1. Similarly, the insertion time for next "b" and "a" number of suffices is denoted by T_2 and T_3. The generalized calculations of these variables are given below.



$T1 = c.(c + 1)/2;$ (1)

$T2 = b.(2c + b + 1)/2;$ (2)

$T3 = a.(2c + 2b + a)/2;$ (3)

$a + b + c = 1$ (4)

For above four equations it is assumed that insertion time taken by unit length suffix is unit. In order to equally divide the load on all processors, we equate $T1 = T2 = T3$, and this helps us to calculated the value of a, b and c. In this case, the Approximate calculated value of a, b and c are (3/19), (4/19) and (12/19) respectively.

7. IMPLEMENTATION DETAILS

The weighted suffix tree construction algorithm is implemented on two different parallel architectures that are CUDA and Cluster computing. In CUDA, we apply static partitioning based approach for weighted suffix tree construction, whereas in Cluster computing we apply both static and dynamic partitioning based approach for weighted suffix tree construction. The exact details of implementation on both architectures are given in next subsections.

7.1. Implementation of static weighted suffix tree construction algorithm on CUDA

The implementation of the static weighted suffix tree construction algorithm on CUDA mainly consists of construction of generalized suffix tree for suffices of all possible sequences generated from weighted sequence. In order to speed up the whole execution of static weighted suffix tree construction algorithm, the construction part of this algorithm is ported to device. As there is no need for much co-operation amongst the parallel instances, it is prefer to sparsely distribute their corresponding threads over different blocks, with assumption that each blocks containing only few threads, rather than a single block with multiple threads.

The other major reason for such a division of threads is that the logic for the simulation involved so much complication in the flow of control, which depended on randomized inputs. As results of this, it is not possible for the threads of different parallel simulations to follow similar code paths, consequently, we negating the use of threads from the same block and we construct four sub-trees using four threads present in different blocks.

7.2. Implementation of static weighted suffix tree construction algorithm on Cluster computing using MPI

The parallelization of weighted suffix tree construction algorithm on Cluster computing is a bit different to that on CUDA, as there is no shared memory present in Cluster computing. So, the initial steps of the weighted suffix tree construction algorithm, for instances converting weighted sequence into solid pattern are performed individually by all processors. But as this work has done in parallel, thus it is not going to affect the overall construction as far as we are having sufficient number of processors. Implementing the algorithm on computer cluster is more straightforward than CUDA as it never required much communication between different processing units, and also the individual nodes of the cluster provide same architecture of an individual CPU. So, there are only few changes in sequential code is sufficient to port it on Cluster computing.

7.3. Implementation dynamic weighted suffix tree construction algorithm on Cluster computing using MPI

The dynamic partitioning approach was implemented on Cluster computing using MPI-API. As 20 nodes were available to us the maximum depth of parallelization was set to 2 and maximum number of process per sub-tree to 4. There was no need of much corporation between these processes in our strategy. The authors [13] have designed a different strategy for sub-tree construction on computer cluster in which master node assigned more work to a process as soon as it completes the assigned work. Their strategy requires a lot more communication between master process and other processes.

In our strategy once it is decided which process will construct which part of the tree, no further communication is required between processes till they finish their job and return their final results to the master processor.

8. RESULT AND DISCUSSION

As per best of our knowledge, the weighted suffix tree construction is not implemented even on uniprocessor. So we first implemented it on uni-processor and record its execution time while it was executed on a Dell precision 5400 with 8 core Intel Xeon Processor (@2.50 Ghz, 16 GB RAM, 64 bit, X2 MB L2 Cache) for varies length of weighted sequence. All though, the above mentioned system is contains more than one core, but due to the absence of any application level parallel techniques. Our program only runs on one core. So it can be taken as a uni-processor implementation.

The parallel algorithm was implemented on Nvidia Quadro FX 3700 graphics card having configuration (512 MB shared memory, 14 multiprocessors) with nvcc 1.0 compiler is used for CUDA program compilation, and also on Cluster computing with configuration (20 running nodes and each node is Sun fire x2200 Quad core server @2.3 Ghz, 512 KB L2 cache per core). The DNA sequence was obtained from [21] and joined together to make a weighted sequence. The value of threshold is taken as 0.1. It is evident that both parallel

implementations of weighted suffix tree construction algorithms are faster than the serial implementation. Maximum speedup achieved in CUDA implementation over uni-processor implementation is around 2.45X times using 4 threads as shown in Table 1.

Table 1. Result for static approach implementation on Uni-processor, CUDA, and static as well as dynamic approach on Cluster Computing

Weighted Sequence Length	Generated sequence length	Number of nodes	Uniprocessor Construction Time in (ms)	CUDA Construction Time in (ms)	Cluster Construction Time in (ms)	
					Static Approach	Dynamic Approach
20	80	73	0.59	1.62	169.0	288.0
40	160	114	1.5	3.29	169.3	288.2
60	465	233	12.5	7.8	169.9	289.8
80	625	273	23.5	9.8	171.1	291
100	2439	693	123.6	40.5	177.4	291.3
120	3079	738	146.0	60.2	182.0	295.3
140	5262	651	250.0	87.8	189.3	306.2
160	6542	673	344.0	133.0	199.6	309.3
180	19792	763	1010.5	332.8	246.1	340.0
200	24912	808	1209.5	484.5	299.3	353.3
220	42096	651	1828.1	687.1	333.2	426.7
240	52336	673	1984.3	1048.7	413.2	464.2
260	96384	559	3812.7	1691.8	577.8	562.6
280	116864	603	4312.5	2530.0	731.7	593.6
300	137344	633	5247.2	2965.1	898.4	648.9

The result obtained doesn't match the capabilities of GPU as in image processing task. The reason for not getting very high speed up in case of CUDA can be slow global memory access which is not cached and the weighted suffix tree cannot be created in shared memory which is faster but has very small size. So it is to be created in global memory which is bottleneck in the performance. A related limitation of global memory access is that read is faster in case of continuous memory location and tree access doesn't provide very good locality property. On the other hand, while comparing the result shown in "Table 1" we found that the performance on Cluster computing is much faster than uni-processor and CUDA. Maximum speedup achieved by Cluster computing implementation over uni-processor implementation is around 6X times, and we also found that implementation of static approach on Cluster computing is outperforms the CUDA implementation by a factor of 3X times.

The main reason for outperforming CUDA, while using same number of processes/threads is the availability of better processor and faster memory access. In addition to this, an implementation of dynamic approach for weighted suffix tree construction on cluster computing provides more speedup than static approach implementation on cluster computing. Maximum speedup achieved in using dynamic partitioning is around 27% over static partitioning in cluster computing. Moreover, the pictorial representation of our results is also shown in (Figure 9-10).

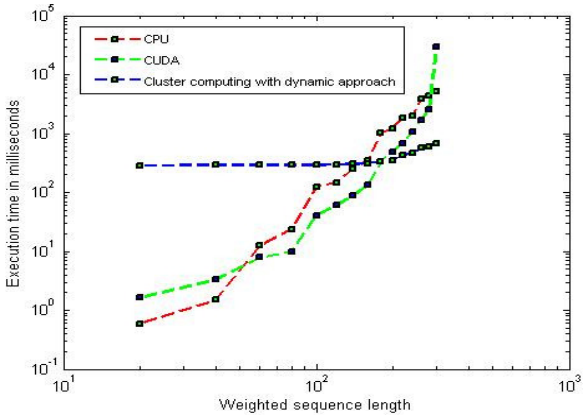


Figure 9. Execution time comparison of weighted suffix tree construction algorithm on parallel architectures.

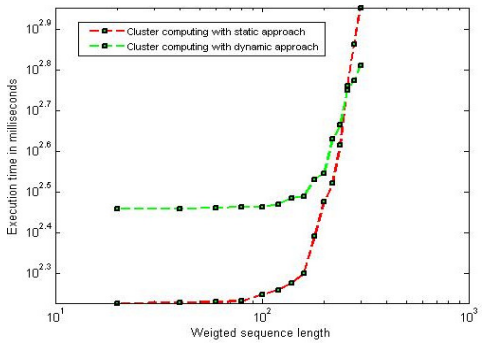


Figure. 10 execution time comparison of static and dynamic partition based weighted suffix tree construction algorithm on cluster computing

9. CONCLUSION

In the field of computational biology, the weighted suffix tree is consider as a one of very important data structure used to perform sequence analysis for molecular weighted sequences. However, static parallel construction of a weighted suffix tree for long weighted sequences is made critical by poor memory locality and high memory overheads**,** which can overcome by applying dynamic approach for weighted suffix tree construction.

In this paper, we have demonstrated how a dynamic approach can be used to speed-up the construction of large-scale weighted suffix trees. In order to achieve a meaning full parallelization, a partitioning of the suffixes of all valid sequences into a number of smaller subgroup of suffixes has been done.

Based on this partitioning we have implemented the static weighted suffix tree construction algorithm on CPU, CUDA and dynamic approach based weighted suffix tree is constructed on cluster computing. Our experiment shows that for molecular nucleotide weighted DNA sequences (up to4^{300} characters long) the performance of dynamic approach on Cluster Computing is significantly better than static approach on CUDA and cluster computing.

REFERENCES

[1]. C. S. Iliopoulos, K. Perdikuri, E. Theodoridis, A. Tsakalidis, and K. Tsichlas, "Algorithms for Extracting Motifs from Biological Weighted Sequences", Journal of Discrete Algorithms, vol.5, pp. 259-277, 2007.

[2]. N. J. Larsson, "Extended Application of Suffix Trees to Data Compression, In the Proceedings of IEEE Conference on Data Compression, Snowbird, USA, pp.190-199, 1996.

[3]. C. S. Iliopoulos, C. Makris, I. Panagis, K. Perdikuri, E. Theodoridis, and A. Tsakalidis, "Computing the repetitions in a weighted sequence using weighted suffix trees, In European Conference on Computational Biology (ECCB), Paris, France, pp. 539-540, 2003.

[4]. C. S. Iliopoulos, C. Makris, Y. Panagis, K. Perdikuri, E. Theodoridis, and A. Tsakalidis, "The Weighted Suffix Tree: An Efficient Data Structure for Handling Molecular Weighted Sequences and its Applications", Fundamenta Informaticae, vol.71, pp. 259–277, 2006.

[5] E. Ukkonen, Online Construction of Suffix Tree, Journal of Algorithmica, vol.14(3), pp. 249–260, 1995.

[6]. E. M. McCreight, "A Space-Economical Suffix Tree Construction Agorithm", Journal of the ACM, vol.23 (2), pp. 262-272, 1976.

[7]. CUDA Programming Guide, Version 2.3, NVIDIA, January 2009

[8]. A. Amir, C. Iliopoulos, O. Kapah, and E. Porat, "Approximate Matching in Weighted Sequences", In the Proceedings of 17th Annual Symposium Combinatorial Pattern Matching, Lecture Notes in Computer Science, Barcelona, Spain, vol. 4009, pp. 365-376, 2006.

[9]. B. K. Pandey, R. Niyogi, and A. Mittal, "Implementation of Weighted Suffix Tree Construction Algorithm on Parallel Architectures, First International Federation of Information Processing (IFIP) International Conference on Bioinformatics, Surat, 25-28 March 2010.

[10]. J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA", ACM Siggraph, New York, pp. 1-14, 2008.

[11] B. Wilkinson and M. Allen, Parallel Programming- Techniques, and Applications using Networked Workstations and Parallel Computers, Prentice Hall , 1999.

[12]. C. Dwyer, "Computer-aided Design for DNA Self-Assembly Process and Applications", IEEE/ACM International Conference on Computer Aided Design, San Jose CA USA, Page(s). 662-667, 6-10 Nov 2005.

[13]. C. Chen and B. Schmidt, "Constructing Large Suffix Trees on a Computational Grid", Journal of Parallel and Distributed Computing, vol.66, pp.1512-1523, 2006.

Binay Kumar Pandey received the B. Tech. (Information Technology) and M. Tech in (Bio-Informatics) degrees from the Institute of Engineering and Technology Lucknow, Maulana Azad National Institute of Technology Bhopal in 2005 and 2008 respectively. He was third topper during graduate studies and was awarded Prime Minister scholarship for meritorious ward of defence personnel for his excellent performance. After completing his M. Tech programme,
He proceeds towards to complete his PhD degree from Electronics and Computer Engg, Indian Institute of Technology Roorkee.