Parallel Generalized Suffix Tree Construction for Genomic Data

Md Momin Al Aziz(✉), Parimala Thulasiraman, and Noman Mohammed

Computer Science, University of Manitoba, Winnipeg, Canada
{azizmma,thulasir,noman}@cs.umanitoba.ca

Abstract. After a decade of digitization and technological advancements, we have an abundance of usable genomic data, which provide unique insights into our well-being. However, such datasets are large in volume, and retrieving meaningful information from them is often challenging. Hence, different indexing techniques and data structures have been proposed to handle such a massive scale of data. We utilize one such technique: Generalized Suffix Tree (GST). In this paper, we introduce an efficient parallel generalized suffix tree construction algorithm that is scalable for arbitrary genomic datasets. Our construction mechanism employs shared and distributed memory architecture collectively while not posing any fixed, prior memory requirement as it uses external memory (disks). Our experimental results show that our proposed architecture offers around 4-times speedup with respect to the sequential algorithm with only 16 parallel processors. The experiments on different datasets and parameters also exhibit the scalability of the execution time. In addition, we utilize different string queries and demonstrate their execution time on such tree structure, illustrating the efficacy and usability of GST for genomic data.

Keywords: Parallel generalized suffix tree · Genomic data indexing · Parallel computation on genomic data · GST construction on disks

1 Introduction

The achievements in human genomics have been remarkable during the last decade. Concepts like genomic or personalized medicine and genetic engineering are slowly becoming reality which seemed impossible a few years ago. We are now capable of storing thousands of genome sequences from patients along with their medical records. Today, medical professionals utilize this large-scale data to study associations or susceptibility to certain diseases.

The recruitment for different genomic research is also increasing as the genome sequencing cost is ever-reducing through technological breakthroughs in the last few years. This growth in genomic data has resulted in consumer products where companies offer healthcare solutions and ancestry search based on human genomic data (*e.g.*, Ancestry.com, 23AndMe). Interestingly, all these

C. Martín-Vide et al. (Eds.): AlCoB 2020, LNBI 12099, pp. 3–15, 2020.
https://doi.org/10.1007/978-3-030-42266-0_1

applications share one major operation: *String Search*. Informally, the string search denotes the presence (and locations) of an arbitrary query nucleotide sequence in a larger dataset. The search results comprise the individuals who carry the same nucleotides in the corresponding positions. Thus, we can perceive the relation between an unknown sequence to pre-existing sequences with such search queries.

On the other hand, suffix tree is proven useful for searching different patterns or arbitrary queries on genomic data [2]. However, their construction suffers from the locality of reference as reported in the initial works [7]. This problem refers to the memory accesses in the same locations within a short period while building the suffix tree. Moreover, it gets severe as suffix trees perform best when the tree (vertices and edges) completely fits in the main memory. Unfortunately, this is quite impossible with off-the-shelf implementations and large scale genomic data.

In this work, we construct generalized suffix trees (GSTs) in parallel. There has been several attempts in efficient, parallel *suffix tree* building which considers only one sequence whereas GSTs represent multiple sequences [5]. We employed two different memory architectures for our parallel GST construction: (a) distributed and (b) shared memory. In a distributed architecture, we utilized multiple machines with completely *separate main memory* system interconnected within a network. On the contrary, these processors have several cores, which share the *same main memory*. These cores are employed in our shared memory model. Furthermore, we employ a data specific parallelism based on the fixed nucleotide set in this construction for the shared memory architecture. Finally, our GSTs are built on file system to remove the dependency for a sizeable memory requirement. We can summarize our contributions below:

- The primary contribution of this paper is a parallel framework using the distributed and shared memory models to construct GST for a genomic dataset.
- We also utilize the external memory (or disks) since GSTs for large-scale genomic data require notable memory size, which is usually not available in a single machine.
- We test the efficiency of our GST with multiple string search queries. Furthermore, we analyze the parallel speedup in terms of dataset size, number of processors, and components of the hybrid memory architecture.
- Experimental results show that we can achieve around *4.7 times* speedup compared to the sequential algorithm with 16 processors to construct the GST for a dataset with $n = 1000$ sequences and 1000 nucleotides each.

Notably, Ukkonen's algorithm [14] went out of memory for $n, m = 10,000$ dataset whereas our proposed approach takes 77.3 s with 16 processors.

2 Preliminaries

2.1 Haplotype Data

In this paper, we utilize the bi-allelic genomic data where the ratio of different nucleotides in a specific position of a chromosome is known beforehand. It is

also called haplotype data, where each allele (or position) on the chromosome is inherited from a single parent. In other words, in one specific location, we can only perceive two variations for such a dataset; therefore, we utilize a binary representation. However, our proposed method is not limited to such binary representation and generalizable over any dataset with a fixed character set.

2.2 Generalized Suffix Tree

Suffix Trie and Tree: Trie (from re**trie**val) is a data structure where each element of the data are placed in the vertex of a tree. Here, the edges represent the relation of one data to the other. In our problem scenario, each nucleotide of the sequences can be seen as the other data points or vertices of a Trie.

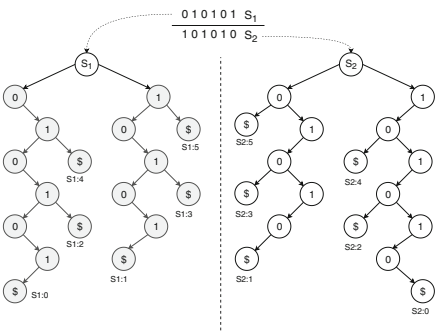

Fig. 1. Uncompressed Suffix Tree (Trie) construction

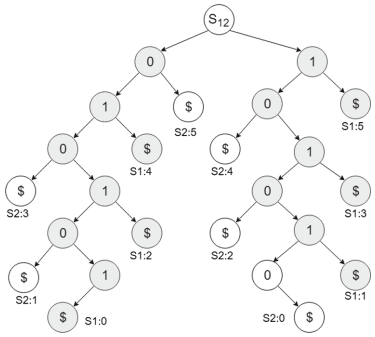

Fig. 2. GST from Fig. 1 where gray and white vertices are from `S1` and `S2`, respectively

Suffix Trees are a compressed version of their Trie counterpart. For example, if a single vertex has only one child on a suffix trie, they are joined and denoted as a single vertex on the Suffix Tree. In Fig. 1, we show two suffix tries `S1` and `S2` from sequences 010101 and 101010 respectively. For any sequence `S1` = 010101, we consider all possible suffixes such as $\{1, 01, 101, 01010, 10101, 010101\}$ and construct a trie. The simple approach to construct such a tree will require an iteration over all suffixes and add/merge new vertices if required.

The suffix tree will also represent the end of the sequence with a special end character (`$`). For example, the suffix 01 (left Fig. 1-`S1`) has an end character with label `S1:4` which denotes the sequence number and the start position of the suffix. Formally, we use vertex label `Sx:y` *s.t.* $x \in \{1, n\}$ and $y \in \{0, m-1\}$ for n sequences of m length.

Generalized Suffix Tree (GST): Generalized Suffix Tree is a collection of suffix trees for multiple sequences. Here, we merge two suffix trees `S1` and `S2` from Fig. 1 and construct `S12` in Fig. 2. Fundamentally, there are no difference in constructing GST as we need to build individual suffix tree per sequence and merge

them afterwards. Thus, the runtime for one GST construction depends on these suffix tree construction and size. For example, the traditional algorithm (Ukkonen) to build the suffix tree has a linear runtime $O(m)$ for m length sequences [14]. Therefore, n sequences with m characters will be at least $O(nm)$ along with the additional linear tree merging cost of $O(n)$.

2.3 Utility Measure

We considered different queries to test the utility of a GST. However, for brevity, we only report three problem instances which are related and incrementally challenging. The input will be a genomic dataset D consisting n individuals with m nucleotides. Since we are considering $n \times m$ haplotypes, D will have $\{s_1, \ldots s_n\}$ records with $s_i \in [0,1]^m$. The query can be of arbitrary length ($1 \leq |q| \leq m$).

Query 1 (Exact Match-EM). For a genomic dataset D and an arbitrary query q, an exact match will only return the records x_i which observe $q[0, |q| - 1] = x_i[j_1, j_2]$ where $q[0, |q|-1]$ denotes the full query and $x_i[j_1, j_2]$ is a substring of the record x_i given $j_2 \geq j_1$ and $j_2 - j_1 = |q| - 1$.

Query 2 (Set Maximal Match-SMM). Similarly, for the same inputs, a set maximal match will return the record x_i which have the following conditions:

1. $q[j_1, j_2] = x_i[j_1, j_2]$ where $j_2 > j_1$ (same length and positions),
2. $q[j_1 - 1, j_2] \neq x_i[j_1 - 1, j_2]$ or $q[j_1, j_2 + 1] \neq x_i[j_1, j_2 + 1]$, and
3. No other records x_i' with substring $[j_1', j_2']$ where $j_2' - j_1' > j_2 - j_1$.

Query 3 (Position-variant Set Maximal Match-PVSMM). Finally, for a threshold t, the PVSMM will report all records where which follow:

1. $q[j_1, j_2] = x_i[j_1', j_2']$ where $j_2 - j_1 = j_2' - j_1' \geq t$,
2. $q[j_1 - 1, j_2] \neq x_i[j_1' - 1, j_2']$ or $q[j_1, j_2 + 1] \neq x_i[j_1', j_2' + 1]$, and
3. No other records x_i'' with substring $[j_1'', j_2'']$ where $j_2'' - j_1'' > j_2' - j_1'$.

3 Methodology

3.1 Architecture and Design Goals

The outline of the parallel generalized suffix tree construction and corresponding computation is depicted in Fig. 3. Here, the genomic dataset $D_{|n \times m|}$) is operated by the data owner and the researchers have q queries on D. The researcher does not have any substantial processing power compared to the data owner since s/he is only interested in a minuscule portion of D.

The high-level design of our architecture is illustrated in Fig. 3, where the data is evenly partitioned between different computing nodes (in one/multiple clusters). Here, we consider and utilize two type of memory environment: distributed and shared. In the distributed memory, the machines are connected via network as they have *mutli-core* processors and their own physical memory (RAM). The mutli-core processors on these machines collectively use the physical memory which is called as shared memory. Hence, we have $|p|$ computing nodes which construct our desired GST jointly.

Our memory dispersion tackles one of the major disadvantages of the GST construction: the sizeable memory requirement for longer sequences. For example, a thousand length sequence can create a maximum of a thousand vertices, and n sequences can lead to an order of nm. Thus, for an arbitrary genomic dataset, it often outruns the memory. Hence, this motivates us to construct our targeted GST in a *distributed memory setting*.

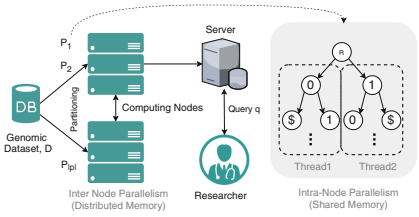

Fig. 3. Computational architecture where data owner has the dataset D while a researcher submits query q

This leads to our proposed design where we distribute the data (partition) and build the suffix tree separately in different computing nodes. These nodes can construct each subtree which is later shared to the other nodes. These shared subtrees are then merged, and the final tree includes all suffix subtrees combining the outputs from all computing nodes (Sect. 3.2.4). The multiple processors in each node will also use shared memory model while constructing and merging their individual GST in parallel (Sects. 3.2.2 and 3.2.3). Therefore our three design goals can be summarized as follows:

1. Partition the dataset for different nodes in a *distributed memory architecture* where individual computing nodes receive a part of the data and only constructs a subtree of the final GST (Inter-node Parallelism)
2. As these nodes are equipped with multiple cores, they will build the individual GSTs in parallel using *shared memory architecture* (Intra-node Parallelism)
3. Use external memory to store and share the resulting GSTs to reduce sizeable main memory requirement.

3.2 Parallel GST Construction

3.2.1 Data Partitioning

We utilize different data partitioning scheme based on the memory locality, availability and the number of computing nodes:

Horizontal partitioning groups a number of sequences for the existing computing nodes. Each node will receive one such group and construct the corresponding GST afterwards. For example, if we have $n = 100$ sequences and $p = 4$ nodes, then we will split the data into 4 groups where each group will contain $|n_i| = 25$ records or genomic sequences. Each node, p_i will build their GST on $|n_i|$ sequences of m length in parallel without any communication. Figure 1 depicts a simple case of this partition scheme for $n = p = 2$.

Vertical partitioning divides the data across the columns and distributes it following the aforementioned mechanism. However, this scheme will have some additional implications while merging the resulting subtrees (Sect. 3.2.2). For example, if we have genomic data of length $m = 100$ and $p = 4$, we will have $n \times m_i$ partitions where each dataset will have $|m_i| = 25$ columns.

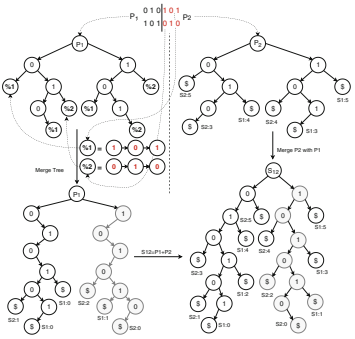

Fig. 4. Vertical partitioning with path graphs (%1, %2) merging

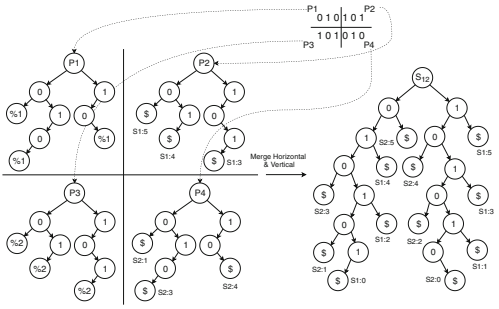

Fig. 5. Example of Bi-Directional partitioning scheme

Bi-directional data partitioning combines both the horizontal and vertical approach as it divides the data into both directions. Notably, it can only operate for $p \geq 4$ cases. Given $n = 100$, $m = 100$ and $p = 4$, each node will receive a $n_i \times m_i = 50 \times 50$ sized data for their computations.

3.2.2 Distributed Memory

We use several machines (or nodes) to build the final GST in parallel (*Inter*-node Parallelism), each with their individual global memory and connected via network. After receiving the partitioned data, these computing nodes are required to build their own GSTs. For example, if there are $p_0, \ldots, p_{|p|}$ nodes then we will have $GST_0, \ldots, GST_{|p|}$ trees. Regardless of the partitioning mechanism, we use the same linear time method to construct the suffix trees using Ukkonen's algorithm [14]. After these individual nodes build their GSTs, they need to share them for the merging operation described next.

In Fig. 1, we see a horizontally partitioned GST construction. Here, two suffix trees are merged where the grey and white colored nodes belonged to different trees. Notably, the merge operation did not duplicate any node at a particular depth. For example, if there was already a node with the value 0 is present, then it will not create another node and simply merge onto its branches. This condition is applied to all merge operations to avoid duplicate branches.

However, for vertical and bi-directional partitioning, the merging requires an additional step for datasets where $m_i < m$. We illustrate this in Fig. 4 where $n = 2, p = 2, m = 6$ and we are creating GST for the sequences $\texttt{S1,S2} = \{010101, 101010\}$. Here, $p1$ operates on $\{010, 101\}$ partitions whereas $p2$ generates the tree for $\{101, 010\}$. Here, the GST from $p1$ needs to have different end characters compared to $p2$ as each end points in a suffix tree needs to represent that the suffix has ended there. However, since we are splitting the data on columns, it needs to address the missing suffices.

Therefore, we perform a simple merge for all cases with $m_i < m$ as we add the *Path Graphs* with $m - m_i$ characters on the resulting GST. For example, in

Fig. 4, we add the path graph of 101 (represented as %1) in all end characters of `S1` in $p1$ (after 010). Similarly, we need to add 010 for `S2` represented as %2. During this merge, we also do not create any duplicate nodes. The addition of these paths will require the merge operation described next.

3.2.3 Shared Memory

In the distributed memory environment, the individual machines get a partition of the genomic data to build the corresponding GSTs. However, these machines or nodes also have multiple cores available in their respective processor which share the fixed global memory. Therefore, we also employ these cores to build and merge the GST in parallel.

We utilize the fixed alphabet size property of genomic data in our shared memory model (*Intra*-node Parallelism). Since there can be only two possible children of the root (0/1), we can initiate two processes where one process will handle the 0 leading suffixes whereas the other process will operate on 1's. For example, in Fig. 1 two processes $p1$ and $p2$ will generate the suffix tree of $\{01, 0101, 010101\}$ and $\{1, 101, 10101\}$ respectively. The output will be two suffix trees, one from each process which can be joined with the root for the final tree.

It is noteworthy that the GSTs on the partitions can also be build with this shared environment. Here, we will partition the data into the cores and they will build, merge the GST in parallel. However, the number of cores and memory is limited in a non-distributed setting which will restrict larger GSTs.

3.2.4 Merging GSTs

As mentioned earlier, the merge operation takes two different GSTs and adds all their vertices. Hence, all $|p|$ GSTs are merged into the final GST where $GST = GST_0 + \ldots + GST_{|p|}$. Here, we employed the shared memory parallelism as the children of the root (0/1) are totally separate and do not have any common edges. In other words, we can treat the root's 0 branch separately from child 1. This allows us to perform the merge operation in parallel and utilize the Intra-node parallelism in each computing nodes.

Notably, merging one branch of a tree is a serial operation as multiple threads cannot add/update branches simultaneously. This creates a bottleneck as we need to perform merge operation in all GST_i's and add the path graphs mentioned in Sect. 3.2.2. However, we can use multiple cores for different branches as mentioned in Sect. 3.2.3. For example, we can create two processes for handling the 0 and 1 branch from the root. This can be extended for the suffixes starting with $00, 01, 10, 11$ as well. Notably, this parallel operation can be followed for any dataset with fixed character set.

The full merge operation is depicted in Fig. 5 where we perform the bi-directional partition and merge accordingly. Inherently, the bidirectional strategy employs both vertical and horizontal merging strategies as the end columns do not include the $m - m_i$ characters.

Table 1. Horizontal and Vertical partition scheme execution time (in minutes) to build GSTs with number of processors $p = \{1, 2, 4, 8, 16\}$

Data	Serial	Distributed				Shared				Hybrid			
	1	2	4	8	16	2	4	8	16	2	4	8	16
Horizontal partitioning													
200	0.08	0.23	0.09	0.09	0.10	0.14	0.05	0.04	0.03	0.14	0.07	0.05	0.05
300	0.27	1.04	0.23	0.2	0.23	0.38	0.15	0.11	0.08	0.37	0.16	0.12	0.12
400	0.59	2.03	0.55	0.38	0.38	1.18	0.35	0.21	0.2	1.12	0.31	0.23	0.25
500	1.53	3.14	1.32	1.06	1.01	2.27	0.57	0.36	0.28	2.09	0.52	0.38	0.41
1000	**14.55**	16.23	8.34	6.31	**6.09**	17.38	5.56	3.27	**2.28**	17.14	4.18	3.12	**3.08**
Vertical partitioning													
200	0.08	0.19	0.08	0.05	0.03	0.16	0.07	0.04	0.02	0.14	0.05	0.03	0.02
300	0.27	0.56	0.28	0.17	0.09	0.48	0.22	0.16	0.08	0.39	0.13	0.10	0.06
400	0.59	1.41	1.05	0.36	0.16	1.44	1.01	0.34	0.19	1.21	0.32	0.21	0.13
500	1.53	3.07	1.49	1.08	0.37	3.18	1.49	1.08	0.36	2.35	0.58	0.40	0.24
1000	14.55	**25.24**	12.25	9.06	5.20	22.56	13.11	7.2	4.37	18.22	6.31	4.49	3.10

3.2.5 Communication and Mapping

We use a sequential distribution of work where incremental computing nodes receive contiguous segments of the data. For example, with horizontal and vertical partitioning, each node p_i will receive $\lceil n/p \rfloor \times m$ and $n \times \lceil m/p \rfloor$ records, respectively.

As p_i constructs its GST_i, it stores it in the file system for further processing. Upon completion, all GSTs are sent via network to the nearest processor based on latency. For example, in Fig. 5, P3 and P4 will send their GST to P1 and P2 respectively and P1, P2 will merge these trees in parallel. We utilize external memory as the suffix tree are arbitrarily large for a genomic dataset and can overflow the main memory of a single computing node.

4 Experimental Results and Analysis

4.1 Evaluation Datasets and Implementation

We evaluate our framework on uniformly distributed synthetic datasets as it allows us to perturb the dimensions and check the performance of the underlying methods. Hence, we generate different datasets with $n, m \in \{200, 300, 400, 500, 1000\}$ and name them accordingly. Notably, the sequential algorithm could not finish for larger dataset ($D_{10,000}$) due to its memory requirement. We did not consider n, m in millions due to our computational restrictions as we were only able to access a small computing cluster [1]. Our implementations along with the data are available in https://github.com/mominbuet/ParallelGST.

4.2 Performance Analysis

We analyze our proposed approach in terms of n, m, p and all three (distributed, shared and hybrid) memory models. Here, the distributed memory model will

Table 2. Execution time (in seconds) of bi-directional partitioning to build GST on different datasets with number of processors $p = \{1, 4, 8, 16\}$

Data	Serial	Distributed			Shared			Hybrid			
	1	4	8	16	4	8	16	4	8	16	32
200	4.8	94.2	90	87	43.8	42.6	38.4	70.8	73.2	75.6	1.51
300	16.2	121.8	107.4	106.2	72	48.6	43.2	88.8	75	75.6	1.37
400	35.4	168.6	148.2	124.8	102.6	54	57.6	114	87	96	1.36
500	91.8	231.6	151.8	154.8	145.2	76.2	62.4	146.4	103.8	105	1.36
1000	873	1135.2	428.4	291.6	856.8	202.2	154.8	635.4	312	214.2	1.36
10,000	–	3191.7	428.5	79.6	–	–	–	2878	418	77.3	18.78

not incorporate any intra-node parallelism instructions as discussed in Sect. 3.2.3 whereas the hybrid method will utilize both.

The shared memory architecture distributes the work into different co-located processors (cores) on one single node. Notably, in this model, we do not require any communication between two processes whereas the distributed model will incur communicating the *GST*s. However, the number of processors and memory available in shared model is fixed and limited as we can add new machines in the distributed model. Nevertheless, this comparison will denote the difference in the two memory architecture.

In Tables 1 and 2 we show the execution time of horizontal, vertical and bi-directional partitioning, respectively. Each method is executed on $p = \{2, 4, 8, 16\}$ processors whereas $p = 1$ denotes the serial or sequential execution. The sequential method is plaintext Ukkonen's algorithm [14]. Furthermore, the proposed hybrid approach uses both distributed and shared memory model with two cores on each processor of distributed machines for the 0 and 1 branches of GSTs.

In Table 1, The GST building time for smaller datasets ($n, m \leq 200$) are almost same for all settings. However, as the dataset size increases, the difference in execution time starts to diverge. For example, the sequential execution of D_{200} takes 0.08 min whereas D_{1000} requires 14.55 min. The same operation takes 3.08 min on the hybrid approach with $p = 16$. Similarly, the distributed model takes 6.09 min which shows the impact of intra-node parallelism.

However, one interesting outcome is the shared model's performance. It takes the minimum time of 2.28 min with $p = 16$ which is the lowest in all three experimental settings. However, it is noteworthy that it ran out of memory for datasets $n, m > 1000$. This depicts the necessity of the distributed or hybrid model as shared memory model are more suitable for datasets which only fits the main memory.

Table 1 also shows the impact of vertical partitioning where we need to add the path graphs. This addition is the only difference from the horizontal approach as all the nodes working on data $m_i < m$, needs to merge $m - m_i$ characters to the underlying GSTs. For example, with vertical method it takes 25.24 min

Table 3. Maximum Execution time (seconds) of Tree Building (TB), Add Path (AP) and Tree Merge (TM) for D_{1000}

p	Horizontal			Vertical			Bi-directional		
	TB	AP	TM	TB	AP	TM	TB	AP	TM
4	113.35	–	70.02	292.97	2.7	66.8	4.01	0.37	3.85
8	47.38	–	85.4	138.87	2.9	61.1	0.62	0.16	1.8
16	15.6	–	98	64.4	3.2	57.6	0.12	0.07	1.2

Table 4. Speedup analysis on D_{1000} for all methods with $p = \{2, 4, 8, 16\}$

Method	Distributed				Shared				Hybrid			
	2	4	8	16	2	4	8	16	2	4	8	16
Horizontal	0.58	1.19	1.61	2.80	0.64	1.11	2.02	3.33	0.80	2.31	3.24	4.69
Vertical	0.90	1.74	2.31	2.39	0.84	2.62	4.45	6.38	0.85	3.48	4.66	4.72
Bi-directional	–	0.77	2.04	2.99	–	1.02	4.32	5.64	–	1.37	2.80	**4.08**

to process D_{1000} whereas it took only 16.23 on horizontal approach. The rest of the execution time also follows the same trend as more processor leads to faster executions overall. The performance gain with shared model compared to hybrid is also lost due to the thread synchronization as the threads operate on $m_i < m$ requires more time for sequential path graph addition.

In Table 2, we show our best results where the data is partitioned into both directions. Here, the tree building cost is reduced compared to the prior two approaches as it resulted in smaller sub-trees. For example, with $n = m = 100$ and $p = 4$, each processor p_i will work on 25×25 sized matrix whereas it will lead to 25×100 and 100×25 partitions for horizontal and vertical, respectively.

Table 3 demonstrates the granular execution time for tree building, path graph addition and the merge operation. We took the maximum time from each run as these functions were executed in parallel. Notably, these values are the building blocks for Tables 1 and 2. For example, the tree building time decreases with the increment of processors p. Furthermore, the bi-directional tree build cost decrements with the increment in processors as it divides the data by half.

In Table 4 we summarize the speedup ($= T_{par}/T_{seq}$) results for D_{1000}. Here, the shared model performs well compared to distributed model due to its zero communication cost. Notably, the distributed one is competitive for all $p > 2$ cases. Nevertheless, the shared model could not finish the $D_{10,000}$ as it ran out of the shared memory. On the contrary, both distributed and our hybrid model constructed the targeted GST as it did not depend on the limited, fixed main memory of one machine.

4.3 Utility Measure

We show the execution time of the targeted queries in Table 5. It denotes the efficiency of GSTs performing arbitrary string search as the time only increases

Table 5. Query 1, 2 and 3 execution time in seconds

Query length $\|q\|$	D_{1000}			D_{500}		
	EM	SMM	PVSMM	EM	SMM	PVSMM
400	5e4	0.14	0.15	4e4	0.13	0.12
500	6e4	0.21	0.22	5e4	0.19	0.18
1000	1e3	0.68	0.72	1e3	0.63	0.59

Table 6. Design-level comparison of previous works and ours in GST construction

Work	Parallelism model		Disk-based	GST
	Distributed	Shared		
TDD [13]	✗	✗	✓	✗
TRELLIS [10]	✗	✗	✓	✗
Wavefront [6]	✓	✗	✓	✗
ER_A [8]	✓	✗	✓	✗
PCF [4]	✓	✗	✗	✗
Shun and Blelloch [11]	✗	✓	✗	✗
DGST [15]	✓	✗	✓	✓
Our work	✓	✓	✓	✓

with the query length $|q|$. Notably, the execution time varies slightly for different datasets as we only show the time for D_{1000} and D_{500} for space limitations.

5 Related Works

There has been multiple attempts in our targeted problem as shown in Table 6. Since GST of a large genomic dataset does not fit a sizeable memory, there have been several works to construct the tree in a file system [5]. These disk-based suffix trees usually store the individual subtree (s) on file similar to our approach [12]. For example, Tian *et al.* [13] showed a different suffix tree merging method *ST-Merge* using the Top-Down Disk (TDD) Algorithm.

Wavefront [6] and its successor ER_A (Elastic Range) [8] both targeted disk-based and parallel approach to construct suffix trees. However, these works only considered a suffix tree and distributed memory model, whereas, in this work, we propose a hybrid method and GST. Comin and Farreras [4] proposed Parallel Continuous Flow (PCF) which efficiently distributes the lexical sorting process into multiple processors. Analogous to this work, Shun and Blelloch [11] also proposed a parallel construction scheme utilizing *cilk* (shared memory) in 2014. However, both works target suffix trees whereas GSTs contain a large number of sequences which is more complicated and at the same time more useful.

Finally, in a very recent work in 2019, DGST [15] offered a 3× speed up with data-parallel platform Spark and performed better than the state-of-the-art ER_A [8]. Nevertheless, it did not employ the shared or hybrid model as we performed better with 4× speedup. One work in 2016 did report speedup upto 6× utilizing parallelism from Graphics Processing Units [9]. However, we do not use such H/W and could not benchmark as their implementations is unavailable.

6 Conclusion

In this paper, we constructed GSTs for genomic data in parallel using external memory. We also analyzed its performance using different datasets and queries. In future, we would like to investigate parallel and private query execution on the suffix/prefix tree structure [3]. Moreover, our methods can be utilized for constructing suffix arrays and benchmarked accordingly. Nevertheless, the proposed parallel constructions can be generalized for other tree-based data structures (*e.g.*, prefix) which can be useful for different genomic data computations.

Acknowledgments. The research is supported in part by the CS UManitoba Computing Clusters, Amazon Research Grant and NSERC Discovery Grants (RGPIN-2015-04147).

References

1. Computing Resources. www.cs.umanitoba.ca/computing. Accessed 4 Dec 2019
2. Bieganski, P., Riedl, J., Carlis, J.V., Retzel, E.F.: Generalized suffix trees for biological sequence data: applications and implementation. In: HICSS (5), pp. 35–44 (1994)
3. Chen, L., Aziz, M.M., Mohammed, N., Jiang, X.: Secure large-scale genome data storage and query. Comput. Methods Programs Biomed. **165**, 129–137 (2018)
4. Comin, M., Farreras, M.: Parallel continuous flow: a parallel suffix tree construction tool for whole genomes. J. Comput. Biol. **21**(4), 330–344 (2014)
5. Farach, M., Ferragina, P., Muthukrishnan, S.: Overcoming the memory bottleneck in suffix tree construction. In: Proceedings 39th FOCS, pp. 174–183. IEEE (1998)
6. Ghoting, A., Makarychev, K.: Serial and parallel methods for i/o efficient suffix tree construction. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, pp. 827–840. ACM (2009)
7. Hariharan, R.: Optimal parallel suffix tree construction. J. Comput. Syst. Sci. **55**(1), 44–69 (1997)
8. Mansour, E., Allam, A., Skiadopoulos, S., Kalnis, P.: ERA: efficient serial and parallel suffix tree construction for very long strings. Proc. VLDB **5**(1), 49–60 (2011)
9. Mišić, M.J., et al.: Parallelization of GST algorithm for source code similarity detection. In: 24th TELFOR, pp. 1–4. IEEE (2016)
10. Phoophakdee, B., Zaki, M.J.: Genome-scale disk-based suffix tree indexing. In: SIGMOD International Conference on Management of Data, pp. 833–844. ACM (2007)

11. Shun, J., Blelloch, G.E.: A simple parallel cartesian tree algorithm and its application to parallel suffix tree construction. ACM TOPC **1**(1), 8 (2014)
12. Tata, S., Hankins, R.A., Patel, J.M.: Practical suffix tree construction. In: Proceedings of the 13th International Conference VLDB, pp. 36–47 (2004)
13. Tian, Y., Tata, S., Hankins, R.A., Patel, J.M.: Practical methods for constructing suffix trees. VLDB J. **14**(3), 281–299 (2005)
14. Ukkonen, E.: Online construction of suffix trees. Algorithmica **14**(3), 249–260 (1995)
15. Zhu, G., et al.: DGST: efficient and scalable suffix tree construction on distributed data-parallel platforms. Parallel Comput. **87**, 87–102 (2019)