Efficient Parallel Construction of Suffix Trees for Genomes Larger than Main Memory

Matteo Comin *
Department of Information Engineering
University of Padova, Italy
comin@dei.unipd.it

Montse Farreras
Barcelona Supercomputing Center
UPC BarcelonaTech
Barcelona, Spain
mfarrera@ac.upc.edu

ABSTRACT

The construction of suffix tree for very long sequences is essential for many applications, and it plays a central role in the bioinformatic domain. With the advent of modern sequencing technologies, biological sequence databases have grown dramatically. Also the methodologies required to analyze these data have become everyday more complex, requiring fast queries to multiple genomes. In this paper we presented Parallel Continuous Flow PCF, a parallel suffix tree construction method that is suitable for very long strings. We tested our method on the construction of suffix tree of the entire human genome, about 3GB. We showed that PCF can scale gracefully as the size of the input string grows. Our method can work with an efficiency of 90% with 36 processors and 55% with 172 processors. We can index the Human genome in 7 minutes using 172 nodes.

Keywords

Suffix Tree; Parallel Algorithms; Whole Genome Indexing

1. INTRODUCTION

The increasing availability of biological sequences, from proteins to entire genomes, poses the need for the automatic analysis and classification of such a huge collection of data. The size of the entire Human genome is in the order of 3 Billion DNA base pairs, whereas other genomes can be as long as 16 Gbp. In this scenario one of the most important needs is the design of efficient techniques to store and query biological data. The most common full-text indexes are suffix trees [21], suffix arrays [19].

Traditionally the suffix tree has been used in very different fields, spanning from data compression [3, 4] to clustering and classification [10, 12, 11]. The use of suffix tree has become very popular in the field of bioinformatics allowing a number of string operations, like detection of repeats [17], local alignment [23], the discovery of regulatory elements [8, 9] and extensible patterns [5]. The optimal construction of suffix tree has already been addressed by [27, 22], that provided algorithms in linear time and space. The main issue is that the suffix tree can easily exceeds the memory available, as the input sequence grows.

In recent years researchers have tried to remove this bottleneck by proposing disk-based suffix tree construction algorithms [16], TDD [25], ST-Merge [26] and Trellis [24]. However only recently two methods to construct a suffix tree for input larger than the main memory have been proposed: wavefront [14] and B^2ST [7]. In particular the latter is the first method that is not based on the construction of subtrees, but it uses suffix arrays. The suffix array of a string S is tightly related to the suffix tree of S. It has been proved that it can be translated into a suffix tree in optimal linear time [13]. The main advantage of using suffix arrays is that the memory occupancy is much smaller than that of suffix trees.

The current massively parallel systems have a small amount of memory per processing element, and in the future, with clusters of GPUs, the available main memory per processing element will become even smaller. Our algorithm, *Parallel Continuous Flow*, is conceived such that every node can process more data than its available memory, and this will comply with the massively parallel system used in this study, MareNostrum [1]. The rest of the paper is organized as follows. In the next section we review the other suffix tree construction methods, we discuss our parallel method in section 2 and we conclude with an experimental evaluation of the proposed algorithm in section 3.

1.1 Preliminaries on Suffix tree

Let denote Σ as a set of characters. Let $S = s_0, s_1, ..., s_{N-1}\$$, with $s_i \in \Sigma$, denote an input string of length N, where $\$ \notin \Sigma$. By S_i we denoted the i suffix of S, that is the substring $s_i, s_{i+1}, ..., s_{N-1}, \$$. The suffix tree for S is a data structure organized as a tree that stores all the suffixes of S. In a suffix tree all paths going from the root node to the leaf nodes spell a suffix of S. The terminal character \$ is unique and ensures that no suffix is a proper prefix of any other suffix. Therefore, the number of leaves is exactly the number of suffixes. All edges spell non-empty strings and all internal nodes, except the root node, have at least 2 children. Moreover there are at most $N-1$ internal nodes in the tree. These ensures that the total number of nodes in the suffix tree is linear in the length N. Hence, the maximum number of nodes and edges are linear in N.

*Corresponding Author

EUROMPI '13, September 15 - 18 2013, Madrid, Spain

The optimal construction of suffix tree has been addressed by Ukkonen[27] and McCreight [22]. These methods perform very well as long as the input string and the resulting suffix tree fit in main memory. For a string S of size N, the time and space complexity are $O(N)$, which is optimal. However, these optimal algorithms suffer from poor locality of reference. Once the suffix tree cannot fit in the main memory, these algorithms require expensive random disk I/Os. If we consider that the suffix tree is an order of magnitude larger than the input string, these methods become immediately unpractical.

1.2 Construction in External memory

To address this issue several methods have been proposed over the years. In general they all solve this problem by decomposing the suffix tree into smaller sub-trees stored on the disk. Among the most important work we can cite [16], TDD [25], ST-Merge [26] and Trellis [24]. TDD and ST-merge partition the input string into k blocks of size N/k. For all partitions a sub-tree is built. A final stage merges the sub-trees into a suffix tree. This phase needs to access the input string in a nearly-random fashion. ST-Merge improved the merge phase of TDD by computing the LCP (Longest Common Prefix) between two consecutive suffixes. In Trellis the authors proposed an alternative way to merge the data. Sub-trees that share a common prefix are merged together. However this requires the input string to fits the main memory otherwise the performance drops dramatically. Only recently two methods solved this issue, Wavefront[14] and B^2ST [7]. The first will be reviewed in the next subsection. The latter is one of the best performing methods and it will be described in Section 2.

1.3 Parallel methods

The parallel construction of suffix tree on various abstract machines is a well studied topic [6, 15]. Similarly to the optimal serial algorithm, these methods exhibit poor locality of references. This problem has been addressed by Farach-Colton et al. [13] that provide a theoretically optimal algorithm. However, due to the intricacy of the method, it does not exist a practical implementation of this algorithm. The only practical parallel implementations of suffix tree construction are Wavefront[14] and ERa[20]. Wavefront splits the sequence S into independent partitions of the resulting tree. The partition phase is done using variable length prefixes, so that every partition starts with the same prefix. This ensures the independence of the sub-trees that can be easily merged. To reduce the cost of expensive I/O, Wavefront reads the string sequentially. However the cost of partitioning the string into optimal sub-trees can be very high. Wavefront runs on IBM BlueGene/L supercomputer and it can index the entire human genome in 15 minutes using 1024 processors. So far this is the best time for a large parallel system. Unfortunately due to IBM policy the code is not available. Similarly to wavefront, ERa [20] divides the string first vertically to construct independent sub-trees. Sub-trees are then divided horizontally into partitions, such that each partition can be processed in memory. In the result section we will review the performance of these two methods.

2. OUR METHOD: PARALLEL CONTINUOUS FLOW

In this section we present our parallel suffix tree construction algorithm called Parallel Continuous Flow (PCF). We describe how we redesigned the best serial method to construct suffix tree for very large input in external memory B^2ST [7], to achieve better performance on a massively parallel system like MareNostrum.

2.1 Description of B2ST

The basic idea is that the suffix tree of S can be constructed from the suffix array of S and an additional structure that stores the LCP (Longest Common Prefix). It has been shown by [13] that the conversion from Suffix Array to Suffix Tree can be performed in linear time. This translation can be implemented efficiently also for large strings because it exhibits good locality of references. B^2ST divides the input string into k partitions of equal size N/k. The size of the partitions depends on the memory available and for whole genomes in a serial environment the number of partitions is typically in the range $[3, 6]$. At the end of each partition, except the last one, it is attached a tail that is the prefix of the next partition. This tail must never occur within the partition and it is required to compute the order of the suffixes within a partition. For example in Table 1 the string $S = ababaaabbabbbab$ is divided into three partitions of size 5. The first partition is composed by the string $ababa$ followed by the tail aa. The substring aa is the shortest prefix of the second partition that does not occur in the first one. This will allow to order each suffix for each partition. The suffix arrays, SA_i, of each partitions i are shown in the last row of Table 1.

	Partition A					Partition B					Partition C				
S	a	b	a	b	a	a	a	b	b	a	b	b	b	a	b
pos	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
SA	5	3	1	4	2	1	2	5	4	3	4	5	3	2	1

Table 1: An example of partitions of size 5 for the input string $S = ababaaabbabbbab$. The first partition is $ababa$ followed by the tail aa that does not occur within the partition. In the last row there are the suffix arrays, SA, for each partition.

Next, we need to establish the relative order of all suffixes. Instead of using the input string we can achieve the same result by ordering the suffix arrays. Thus in this second step we generate the suffix arrays SA_{ij} for each pairs of partitions, along with the LCP length for each suffix. The LCP is the length of the longest common prefix of each suffix in SA_{ij} with its predecessor and it will be used in the final merge phase. In Table 2 the first two partitions, A and B, from the previous example are combined. This generates the suffix array SA_{AB} of size $|SA_A| + |SA_B| = 10$, and two other data structures: the LCP array and the partition array. The output of this step is the *order array* OA_{AB} that is composed by the arrays LCP and partition.

After the second step we have produced k suffix arrays SA_i, one per partition, of total size N and $k * (k-1)/2$ order arrays OA_{ij} of total size kN. This is the data that will be used for the final merge. To establish the relative order of two suffixes it is enough to reuse the information stored into the order array OA_{ij}. At the general step we need to find the suffix that is lexicographically smaller among the suffixes stored into the SA_i, but since the suffix arrays SA_i are already ordered we need to compare only the top elements for each SA_i.

Suffix Start	5	1	3	1	2	5	4	2	4	3
Partition	A	B	A	A	B	B	A	A	B	B
LCP	0	2	1	3	2	3	0	2	3	1

Table 2: An example of suffix arrays of the pair of partitions A and B. The first two rows represent the suffix array SA_{AB}, where the first row is the position of the suffix within the partition and the second row identify the partition. The last row is the length of the LCP between two consecutive suffixes.

2.2 Parallel Continuous Flow: design

In order to parallelize the above process we need to carefully analyze the data dependencies. A summary of the data flow can be found in Figure 1. As already observed the number of partitions pairs grows like $k(k-1)/2$ whereas the overall data grows like kN. If we analyze the performance reported in [7] we found that the suffix tree construction for the human genome requires about 3 hours on a modern workstation. The most demanding task is the construction of the ordered arrays OA_{ij} that accounts for 95% of the time.

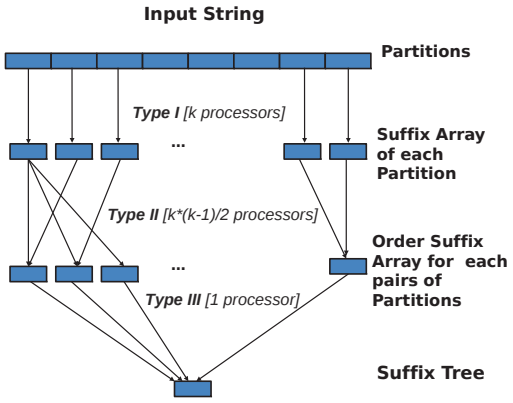

Figure 1: A workflow of the data dependencies of the approach B^2ST.

Another aspect that we need to take into consideration is the order by which the data is needed at each phase. As already discussed all steps will process the data sequentially, in a predetermined, ordered sequence. To distribute the workload evenly, we reserve one processor for each of the most demanding tasks. We choose to allocate $k*(k-1)/2$ processors to construct the ordered array OA_{ij}. Thus each of this processors will compute one order array. We decide to allocate an additional k processors that are dedicated to the construction of suffix arrays SA_i. Similarly one more processor will collect the ordered array and merge the partial results into the final suffix tree. To summarize if k is the number of partitions our algorithm will use $k*(k-1)/2+k+1$ processors. We will call the processors that compute the suffix arrays SA_i, *type I*; those that construct the ordered arrays OA_{ij}, *type II*; and the processor that merges the partial results into the suffix tree, *type III*.

To achieve better scalability we need to make sure that every processor receives a continuous flow of data. This flow should be adequate to maintain active the computation of all processors. Moreover, we implement all communications with asynchronous non-blocking primitives, like *MPI_iSEND*, to overlap communication and computation, so that every processors can continue the computation while the network is taking care of the data transfer. This requires also a series of buffers to store the in-flight temporary data, data which has already been sent but it has not arrived at destination.

2.3 Parallel Continuous Flow: implementation

Next we describe in more details the main algorithm and the different types of processors.

Main algorithm: The main algorithm divides the processors in different types and calls the appropriate procedures. It also prepares the partitions of S for *type I* processors. The string S is read collectively by all *type I* processors. The use of MPI's collective I/O primitives allows to achieve better performance, especially when the input string is large, while ensuring that the same copy of the string is not read multiple times.

Type I Processor: This processors constructs the suffix array SA_i for the partition i. The input is the i^{th} partition of S. The output SA_i will be computed and stored incrementally into the sub-suffix array $SubSA_i$. There are several tools available to construct suffix arrays, we choose one of the best performing methods developed by Larsson and Sadakane [18]. When a new suffix is ordered it is inserted into $SubSA_i$, until it reaches *buffSize*. The sub-suffix array $SubSA_i$ will contain a portion of SA_i of fixed size *buffSize*. Every time $SubSA_i$ is full it will be sent to type II and III processors using non-blocking *MPI_iSEND*, this will allow the type I processor to continue the computation while the network will take care of the message. In order to implement this paradigm we need to store the in-flight messages in a temporary buffer. The number of in-flight messages may affect the overall performance. For this reason we allocate *MemForComm* bytes for the communication buffer. This temporary buffer will contain several messages of size *buffSize*.

Type II Processor: The type II processors will take as input the sub-suffix arrays $SubSA_i$ and $SubSA_j$ and build the ordered array OA_{ij}. Consistently with the type I they will receive the sub-suffix arrays into messages of size *buffSize*. Once both sub-suffix arrays are received it is possible to compute the first portion (of *buffSize* bytes) of the ordered array called $subOA_{ij}$. Similarly with the type I processors the data is kept into a temporary buffer and sent to type III using non-blocking *MPI_iSEND* primitives. In this case we don't need to store several messages because the only node that will need this data is type III. The construction of the ordered arrays consumes the input sub-suffix arrays. Every time one of the two input arrays is empty we will require more data from the corresponding type I processor.

Type III Processor: The type III processor will merge the data and produce the suffix tree. The main task is to order all suffixes so that we can incrementally construct the suffix tree from the ordered list of suffixes. In order to do that we need to compare the suffixes in all arrays SA_i. Since these arrays are already ordered this processor can start as soon as the first $subSA_i$ are produced by all type I processors. For this reason we allocate the space to receive k $subSA_i$ of dimension *buffSize*. Similarly we do the same for all $k*(k-1)$ sub-ordered arrays $subOA_{ij}$. Now given all

these data we can efficiently search for the smallest suffix among the top elements in $subSA_i$. Every time we compare two suffixes, if they are within the same partition since they are already ordered we can output the relative order. On the other hand say one is from $SubSA_i$ and the other from $SubSA_j$, it is enough to check the top element in the order array $subOA_{ij}$ and output the suffix encoded in the partition vector. Every time a new smallest suffix is discovered it is inserted into the final output buffer $subST$. This buffer of size $OutputBuffSize$ contains the partial suffix tree constructed so far, when it is full it is empty to the disk. Once a new suffix, from partition $partId$, is discovered we need to advance a pointer in the corresponding $subSA_{partId}$. Similarly also for all $k-1$ $subOA_{j,partId}$ that involve the partition $partId$. If one of these pointers reach the end of the corresponding array we refill it with a new MPI_Recv, until no more data is available.

2.4 Workflow Analysis and Complexity

To better highlight the properties of our parallel algorithm PCF in Figure 2 a workflow is presented. In this diagram we can see the different types of processors along with the data they produce. For type I and II processors every chunk of data represents $buffSize$ bytes, whereas the data chunks of type III processor are of size $OutputBuffSize$. In this figure we can appreciate how the different processors cooperate to the suffix tree construction.

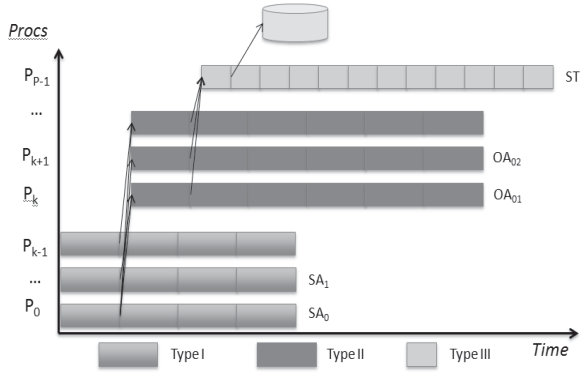

Figure 2: A workflow of our parallel algorithm PCF.

The construction of suffix array takes $O(N)$ time in total for the k partitions. In our parallel algorithm this phase is computed by k type I processors, thus the complexity is $O(N/k)$. Similarly the $k*(k-1)/2$ order array can be built in total time of $O(kN)$, again linearly in the input data. This is in practice the most demanding task that accounts for 95% of the total serial time. The $k*(k-1)/2$ type II processors in $O(N/k)$ time construct all order arrays. The worst case complexity of these two phases is $O(N/k)$, but the number of processors $P = k*(k-1)/2 + k + 1$ scales like $O(k^2)$, thus overall the complexity is $O(N/\sqrt{P})$. This worst case complexity is similar to that of other algorithms like [14] and [20]. It is worth to note that although the worst case complexity does not scales linearly with P, in real applications the construction time scales almost linearly with the number of processors. This relates to the fact that the depth of a suffix tree is in the worst case $O(N)$, whereas, for real data like genomes it can be only $O(\log N)$.

3. EXPERIMENTAL EVALUATION

This section presents the results of our performance evaluation. All the experiments were conducted in the MareNostrum supercomputer [1]. MareNostrum is a cluster of 2560 JS21 blades, each of them equipped with two dual-core IBM PPC 970-MP processors which share 8 GBytes of main memory. Our method called *Parallel Continuous Flow PCF* is implemented in $C++$ using MPI primitives for communication.

3.1 Fine tuning/Memory

As detailed in the previous section our parallel algorithm includes as parameters the sizes of two buffers. The first one, $BuffSize$, controls the size of messages that are being exchanged, the second, $OutputBufferSize$, determines the output file size. We analyzed the best configuration while varying these parameters. In Figure 3 we report the times to construct the suffix tree of a string of 500Mb, drawn from the Human Genome, using 37 processors (8 partitions). In general if the size of messages is small the communication overhead degradates the performance. Similarly if the output buffer is too small it increases the frequency of disk writings and reduces the efficiency. However, bigger buffer sizes results in less frequent communication which may limit the continuous flow of data and hinder load balancing, therefore reducing the overall parallelism of the application. We tested several configurations of these parameters and after an extensive comparison we found that messages of size 5MB to 10MB and an output buffer of size 75MB to 100MB are in general the best choice. These settings are the best performing also for longer inputs (data not shown). Based on these observations we set $BuffSize$ to 10MB and $OutputBufferSize$ of size 100MB.

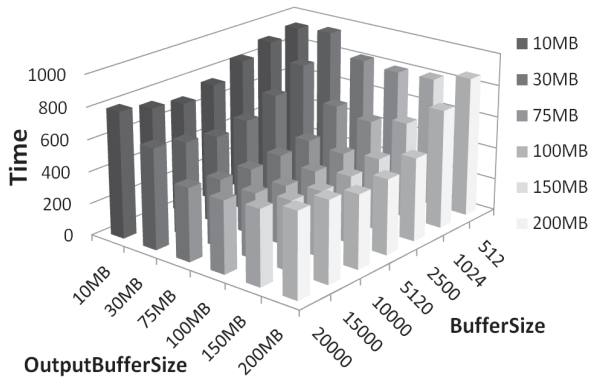

Figure 3: Construction times for a 500MB input with 37 processors (8 partitions) while varying the parameters $BuffSize$ and $OutputBufferSize$.

3.2 Results

We measure execution time for three different data sets of different sizes, all data sets are drawn from the Human genome [2]. We used 43 nodes (172 cores) for our experiments. Each experiment was run 6 times, and the average value was taken. Unfortunately MareNostrum has been recently disassembled and it is no longer available. Moreover the code of Wavefront was not available due to IBM policy and the method ERa was only recently published. To this

end we compare our methods using only the information available from other papers.

For each different data set, we study different allocations for the memory budget - $60D/40C$ indicates that 60% of the memory budget is reserved for the *application data*, while the remaining 40% is used for internal *communication buffers*. Figures 4 and 5 illustrates that the optimal combination is $60D/40C$. A similar results is obtained also for the other dataset (figure not shown). Less memory used in communication buffers means less exploitation of the application parallelism and the application becomes communication bounded, however if more memory is used in communication buffers the application becomes computation bounded. The measurements show that our algorithm scales up to a certain threshold for every size of the data set. The small size data set (500Mb) scales up to 46 threads, up to 106 threads for medium size (1000Mb) (Figure not shown) and up to 172 for the Whole Human genome (3000Mb)(Figure 5). We attribute this scalability threshold to the fact of reaching the inherent parallelism limit of the application.

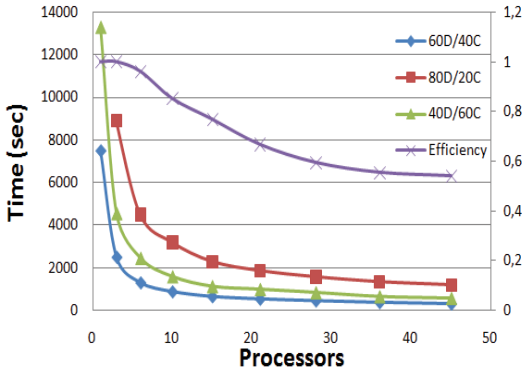

Figure 4: Execution times for different memory allocations and efficiency for the best memory allocation ($60D/40C$) as we increase the number of processors - dataset Human genome of size 500MB.

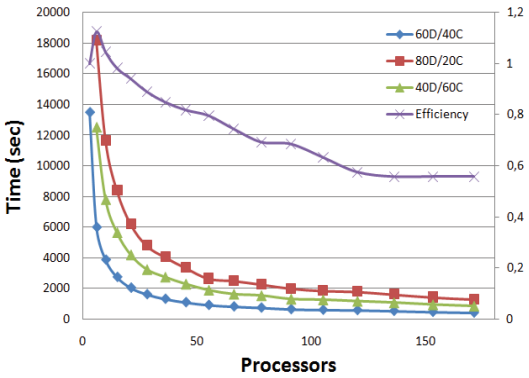

Figure 5: Execution times for different memory allocations and efficiency for the best memory allocation ($60D/40C$) as we increase the number of processors - dataset whole Human genome 3GB.

We define the efficiency as: $Efficiency = SU/MAXSU$ where SU is the actual speedup and $MAXSU$ is the maximum speedup. Since big data sets do not run on a single thread due to memory limitations, we take into account the smallest configuration that is possible to run to compute SU and $MAXS$. This is the run with 3 processes for the Whole Human Genome. Thus in this case the efficiency with p processors is $Efficiency = \frac{PCF(3)/PCF(p)}{p/3}$, where $PCF(p)$ is the construction time with p nodes. The efficiency is close to 90% for small configurations (up to 36 processes for the Whole Human Genome) and it is slightly decreasing as we scale up, due to the parallelization overhead, until it reaches the point where it starts degrading due to the inherent limit of parallelization in the application. For 172 threads we get 55% efficiency (see Figure 5). Similar results are obtained also for the other dataset (Figure 4). We argue that the efficiency is relatively good, given the I/O intensive nature of the suffix tree construction. For example the efficiency of ERa [20] drops very quickly and the best performance reported an efficiency of 53%, but with just 16 processors.

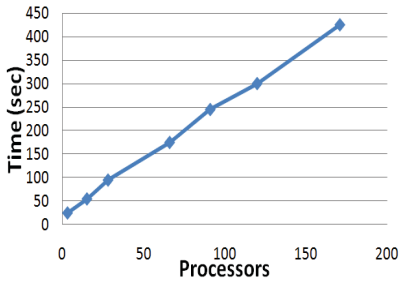

Figure 6: Weak scalability as we increase the input size from 50MB for 3 threads to 3GB for 172 threads.

In the last experiment we test the weak scalability, where the ratio between the size of the input and the number of nodes is constant. We vary the input from 50MB for 3 threads to 3GB for 172 threads. In the ideal case the construction time should remain constant, however no algorithm for this problem reaches this theoretical limit. Figure 6 show that the construction time of PCF increases linearly with the number of processors. In the same test the performance of WaveFront grows more than linearly. Only ERa has a similar behavior, but with a steeper inclination (see figure 13 of [20]). This is very important for real applications, when the input strings can be very long.

PCF constructs the whole Human genome suffix tree in 425 seconds, about 7 minutes, and uses 172 processes in our testing platform. We argue that this is a very good performance comparing with the bibliography, in [14] a time of 880 seconds, 15 minutes, is reported but it requires Blue Gene/L and 1024 processes, similarly in [20] a time of 11,3 minutes is reported with a much recent hardware. However these are absolute times on different platform and thus can not be directly compared.

4. CONCLUSION

The construction of suffix tree for very long sequences is essential for many applications, and it plays a central role in the bioinformatic domain. With the advancing of sequencing technologies the amount of biological sequences available in genome and proteome databases has grown dramatically. Therefore it is essential to have fast method to build suffix

trees. In this paper we presented PCF, parallel continuous flow a parallel suffix tree construction method that is suitable for very long strings. We tested our method for the suffix tree construction of the entire human genome, about 3GB. We showed that PCF can scale gracefully as the size of the input string grows. Our method can work with an efficiency of 90% with 36 processors and 55% with 172 processors. We can index the entire Human genome in 7 minutes using 172 nodes. To be best of our knowledge this is the fastest existing method in terms of absolute time.

5. ACKNOWLEDGMENTS

Many thanks to the BSC for the generous availability of MareNostrum. Work done by M.C. thanks to the HPC-Europa program and the Ateneo Project of the University of Padova. The researchers at BSC-UPC are supported by the Spanish Ministry of Science and Innovation (contract no. TIN2007-60625 and CSD2007-00050). We'd like to thank also Gabor Dozsa for helpful discussions.

6. REFERENCES

[1] Bsc: Barcelona supercomputing center, marenostrum system architecture. `http://www.bsc.es`.

[2] Complete human genome from ncbi public collections of dna and rna sequences. `ftp://ftp.ncbi.nlm.nih.gov/genomes/H_sapiens/Assembled_chromosomes/`.

[3] A. Apostolico, M. Comin, and L. Parida. Bridging lossy and lossless compression by motif pattern discovery. In *General Theory of Information Transfer and Combinatorics, Lecture Notes in Computer Science*, volume 4123, pages 793–813, 2006.

[4] A. Apostolico, M. Comin, and L. Parida. Mining, compressing and classifying with extensible motifs. *Algorithms for Molecular Biology*, 1(4), 2006.

[5] A. Apostolico, M. Comin, and L. Parida. Varun: Discovering extensible motifs under saturation constraints. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 7(4):752–762, October-December 2010.

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

[7] M. Barsky, U. Stege, and A. Thomo. Suffix trees for inputs larger than main memory. *Information Systems*, 36(3):644 – 654, 2011.

[8] M. Comin and L. Parida. Subtle motif discovery for the detection of dna regulatory sites. In *Proceeding of Asia-Pacific Bioinformatics Conference*, pages 27–36, 2007.

[9] M. Comin and L. Parida. Detection of subtle variations as consensus motifs. *Theoretical Computer Science*, 395(2-3):158–170, 2008.

[10] M. Comin and D. Verzotto. The irredundant class method for remote homology detection of protein sequences. *Journal of Computational Biology*, 18(12):1819–1829, December 2011.

[11] M. Comin and D. Verzotto. Alignment-free phylogeny of whole genomes using underlying subwords. *Algorithms for Molecular Biology*, 7(34), 2012.

[12] M. Comin and D. Verzotto. Whole-genome phylogeny by virtue of unic subwords. In *Proceedings of 23rd International Workshop on Database and Expert Systems Applications, BIOKDD*, pages 190–194, 2012.

[13] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM 2000*, 47(6):987–1011, 2000.

[14] A. Ghoting and K. Makarychev. Indexing genomic sequences on the ibm blue gene. In *Proceedings of Conference on High Performance Computing Networking, Storage and Analysis (SC)*, pages 1–11, 2009.

[15] R. Hariharan. Optimal parallel suffix tree construction. In *Proceedings of the Symposium on Theory of Computing*, pages 290–299, 1994.

[16] E. Hunt, M. P. Atkinson, and R. W. Irving. Database indexing for large dna and protein sequence collections. *The VLDB Journal*, 11:256–271, 2002.

[17] S. Kurtz, J. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich. Reputer: The manifold applications of repeat analysis on a genome scale. *Nucleic Acids Res.*, 29(22):4633–4642, 2001.

[18] N. J. Larsson and K. Sadakane. Faster suffix sorting. *Theor. Comput. Sci.*, 387(3):258–272, 2007.

[19] U. Manber and E. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948, 1993.

[20] E. Mansour, A. Allam, S. Skiadopoulos, and P. Kalnis. Era: Efficient serial and parallel suffix tree construction for very long strings. *Proceedings of the VLDB Endowment*, 5(1):49–60, September 2011.

[21] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(1):262–272, 1976.

[22] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of ACM*, 23:262–272, 1976.

[23] C. Meek, J. Patel, and S. Kasetty. Oasis: An online and accurate technique for local-alignment searches on biological sequences. In *Proceedings of 29th International Conference on Very Large Databases*, pages 910–921, 2003.

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

[25] S. Tata, R. A. Hankins, and J. M. Patel. Practical suffix tree construction. In *Proc. of VLDB*, pages 36–47, 2004.

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

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