Parallel Algorithm for Suffix Array Construction

Juan David Quintero Gómez

Universidad de Antioquia
Facultad de Ingeniería
Departamento de Ingeniería Electrónica y de Telecomunicaciones.
Medellín, Colombia

2019

Abstract

Due to the advances of the so-called Next Generation Sequencing technologies (NGS), the amount of genetic information has been significantly increased and it is expected to continue growing, so there is a need to efficiently storing this type of information and an alternative to solve it, is the compression. In many phases of this process, suffix array is a fundamental part and their construction needs a lot of time.

In this degree thesis, an algorithm was designed for the construction of suffix arrays on DNA sequences that combine different techniques and levels of parallelization and contribute to improving the performance in the compression process of this type of data.

1. Introduction

DNA sequencing is the process of determining the nucleotide sequence (Adenine, Cytosine, Guanine, and Thymine) of a DNA fragment. It has had a revolution as a result of the so-called Next Generation Sequencing technologies (NGS), which allow sequencing of more than one billion nucleotides per day in a single machine with a comparatively low cost.

The DNA sequence is useful for informing scientists of the kind of genetic information that is transported in a segment of DNA. Scientists can use this sequence information to determine data that can highlight changes in a gene that can cause disease or find patterns between sequences. Also, due to the advances in this technology, the amount of this data is expected to continue growing and, efficiently storing and analyzing this type of information becomes a computational challenge.

Between the techniques that have emerged as an alternative to solve these problems, are full-text indexes and data compression algorithms. For both strategies, algorithms to build suffix arrays play an important role.

One example of an application relying on suffix arrays, is the compression workflow presented in [1]. Their algorithm uses a referential compression technique that requires many alignments. For that purpose, the Burrows-Wheeler transform (BWT) and the FM-index are used, which in turn depend on efficiently building suffix arrays.

As reported in [1], the Suffix Array Constructor (SAC) is one of the parts that uses more time in the compression process, hence the interest and motivation for the development of this project with the aim of designing and implementing an algorithm for the construction of suffix arrays and obtaining better performance through the use of parallelism.

A sequential version of the algorithm was implemented in C language and profiling of its execution. After, optimizations were made and several parallel versions were implemented in order to accelerate the execution time. To test its scalability and performance, public databases were used.

The objectives of this thesis are presented in Section 1. In section 2 a review of the state of the art is presented in order to explain what is the SAC and the different approaches to obtain it. In section 3 the methodology is presented and the algorithm is described, explaining each of the main phases of this. Section 4 presents the results obtained when testing with a public dataset.

1.1. General objective.

To develop a suffix array algorithm for DNA sequences, combining different levels of parallelization with the purpose of reducing the execution time.

1.2. Specifics objectives.

- To study the related work for the construction of suffix array, different algorithms of sorting for integers and coarse-grained and fine-grained parallelization.
- To design and implement a schema of parallelization to make the groups and subgroups of sorting which can be processed separately.
- To develop a schema of parallelization for the sorting of the suffixes into each group in the indexing algorithm.
- To incorporate strategies of parallelism SIMD developed previously.
- To execute latency and scalability tests of the algorithms developed, comparing the effects of the different levels of parallelism applied and comparing the results with other relevant programs within the state of the art.

2. Related Work

This section will explain the SA and the different approaches to its construction, since this is the main structure that will be discussed along the rest of the document.

The suffix array (SA) of the string S is an integer array that provides the initial positions of the suffixes of S in lexicographic order with a lexicographically smallest suffix ($). This means that A[i] contains the initial position of the i-th suffix in ascending lexicographical order as shown in Figure 1.

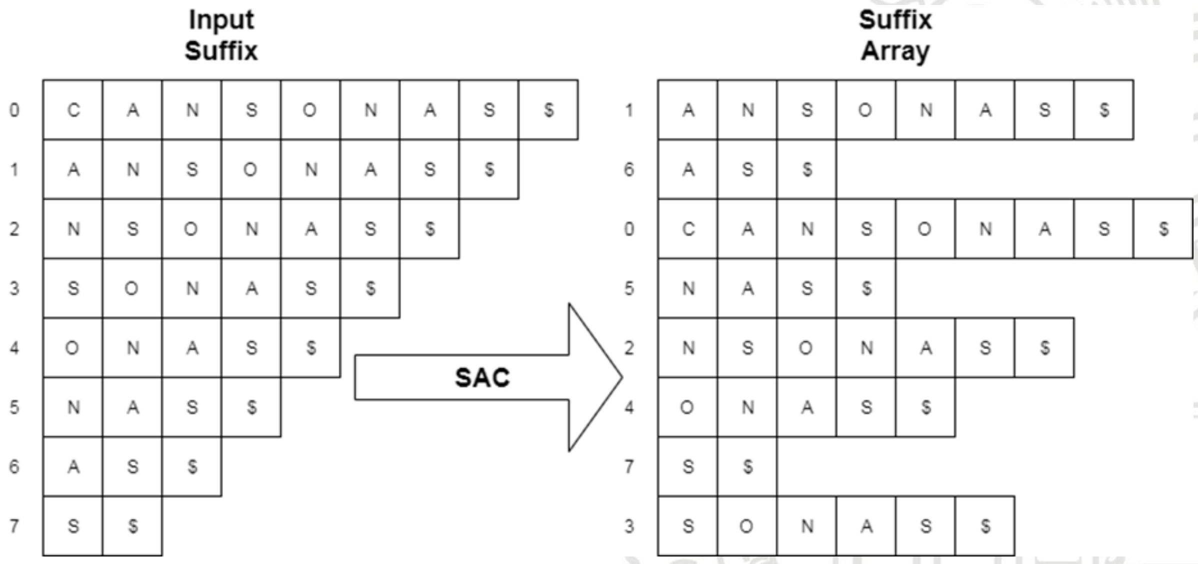

Figure 1. Suffix Array of the word "CANSONAS" after applying the SAC.

The straightforward way to generate a suffix array is to sort all suffixes using a comparison-based sorting algorithm. It starts by sorting according to only the first symbol of each suffix, then successively organize the order by expanding the evaluated part of each suffix. The main idea to develop a more efficient algorithm is to take advantage of the fact that each suffix of the string is also a suffix of another suffix of the string. The suffix array construction algorithms (SACAs) that take advantage of this property can be divided into three classes: prefix-doubling, recursive and induced copying.

The first class of SACAs, prefix-doubling, sorts the suffixes of a string by their prefixes. The idea was first applied to suffix array construction by Manber and Myers [2], and later optimized by Larsson and Sadakane [3] in order to construct a data structure more space- and cache-efficient, and simpler to construct alternative to suffix trees. The second class of SACAs recursively sorts a subset of the suffixes, this result is used to infer the sort of remaining subset, and finally merge the two sorted subsets to get the SA. The final class of SACAs, induced copying, uses already-sorted suffixes to induce the sort of all suffixes.

3. Algorithm developed

3.1. Methodology

The RadixSort and Insertion Sort algorithms were studied, characterizing the problem to statistically verify the variation in the dimension of the data as it iterates and considering the impact of the cardinality of the alphabet. After studying some sorting algorithms and since the construction of suffix array is an iterative process, these were incorporated into the partial sorting of the suffixes and data structures required for the complete sorting of the suffix array were designed and implemented.

Then, the parallel versions were developed using different approaches in order to improve the execution time of the algorithm. Finally performance tests were realized also using Vtune[1] in order to identify how much time each operating block spent and thus recognize the bottlenecks and measure the efficiency. Vtune helps showing how much time each function of the algorithm spends and the distribution of jobs between the threads that are being used.

The evaluation of the algorithm was made with the dataset obtained from the Pizza & ChiliCorpus [2] found in [4].

The algorithm was developed in C using the OpenMP API in order to add parallelism by using multithreading.

3.2. Algorithm description

To construct the suffix array it is necessary to represent the string S in suffixes (rows in Figure 2). The characters in the first column (highlighted in yellow in Figure 2) are called the Valid Chars (VC) and need to be sorted before proceeding to the next iteration.

[1] https://software.intel.com/en-us/vtune
[2] http://pizzachili.dcc.uchile.cl/texts.html

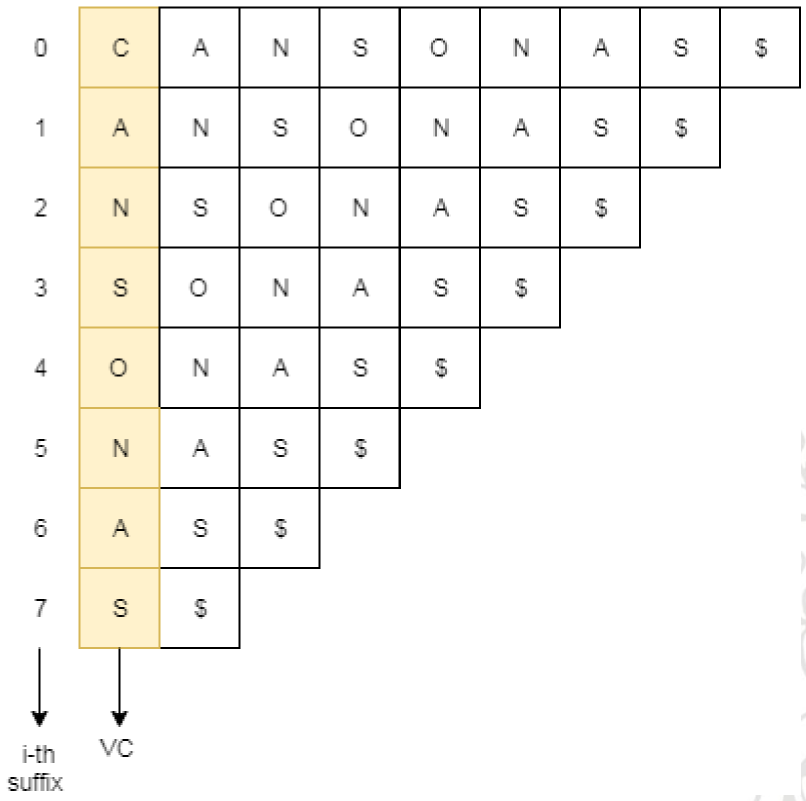

Figure 2. Suffixes array for the string "CANSONAS" and VC for the first iteration.

After sorting the VC of the first iteration, the column containing the second character of each suffix become the new VC for the second iteration (see Figure 3). It should be noted that as a result of the first iteration, the previous VC (already sorted), form what we call subgroups, as shown in different colors in Figure 3. A subgroup is a set of suffixes that have the first i-1 characters in common in the i-th iteration. For any iteration, the VC must be sorted within each subgroup.

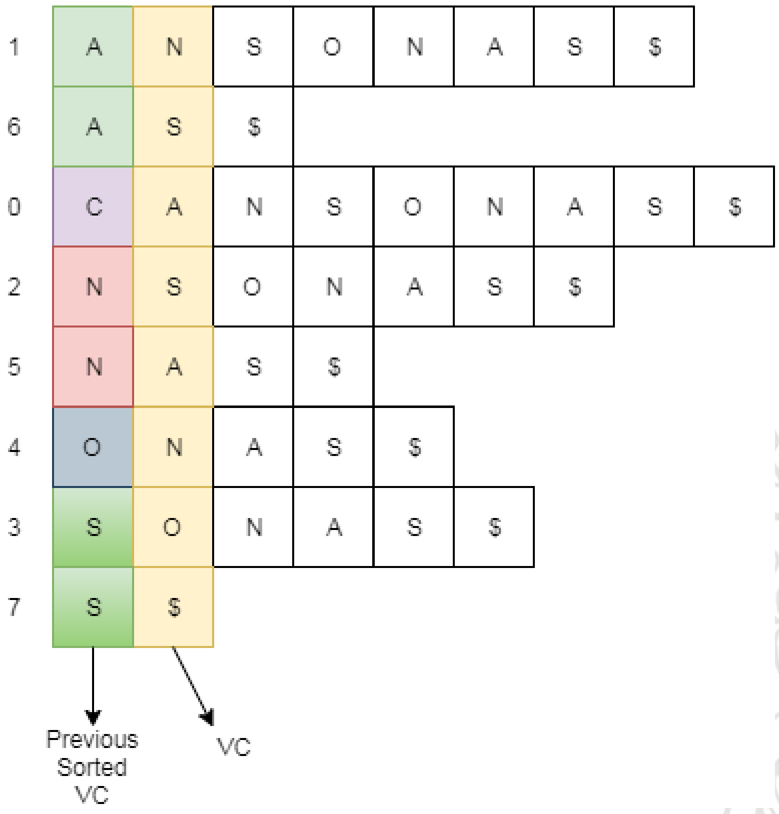

Figure 3. Second iteration in the process of suffixes array construction with the formed subgroups.

After sorting each of the characters in the second column according to the defined subgroups, the third column produces eight subgroups all containing only one character (see Figure 4). This means that all the suffixes have already been sorted and the SAC algorithm ends.

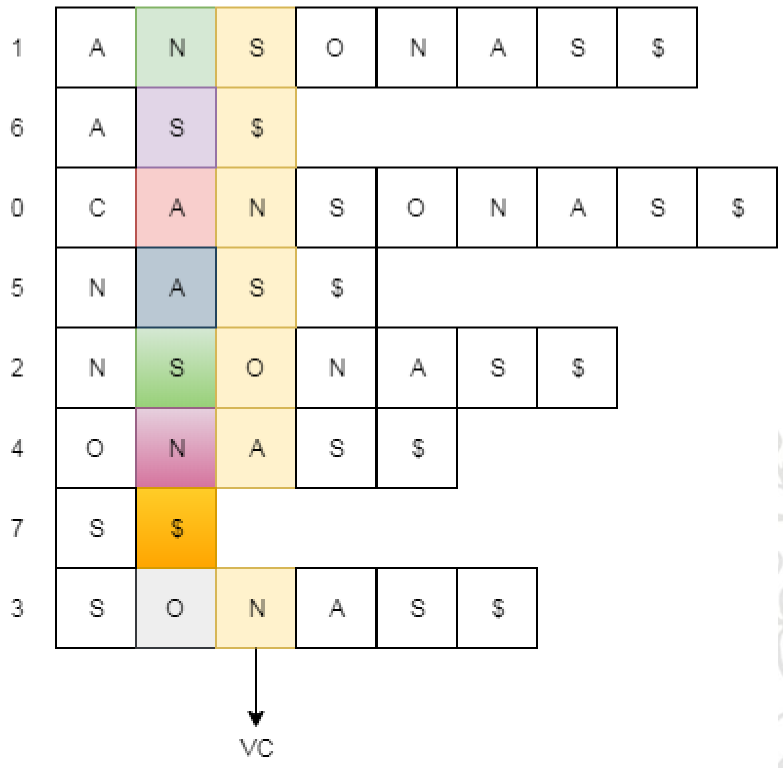

Figure 4. Third and last iteration of the suffixes array construction, each subgroup have only one element.

3.3. Algorithm implementation

The block diagram for the developed algorithm is shown in Figure 5. In the first block called *Init*, the input string S and the unsorted suffix array (SA) are received. At this stage the necessary structures are created for the algorithm development:

- The Bounds Array: this structure is a binary array and it stores the limits between the different subgroups; if a position of the array has a '1', it means that index is the last of a subgroup. The Bounds Array has the same size as SA.
- The Sorted Suffixes (SS): this array is binary and it indicates if a suffix has been fully sorted or not. A suffix has been sorted when it is the only element of the subgroup; '1' means sorted and '0' means not sorted. SS has the same size as SA.
- Valid Chars (VC): This array contains the VC of the current iteration, it has the same size of S.

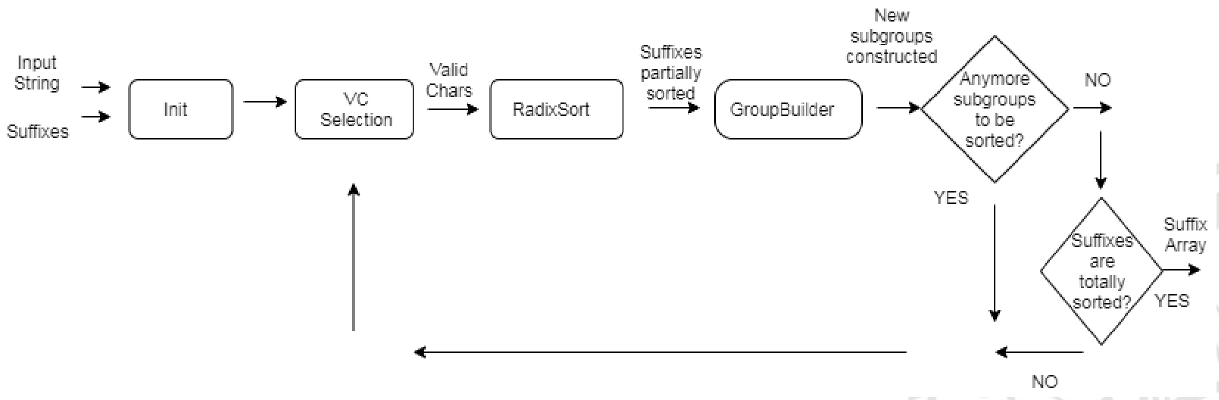
Input String
Suffixes
Init
VC Selection
Valid Chars
RadixSort
Suffixes partially sorted
GroupBuilder
New subgroups constructed
Anymore subgroups to be sorted?
NO
YES
Suffixes are totally sorted?
Suffix Array
YES
NO

Figure 5. Block diagram for the developed algorithm

The main part of the algorithm is composed by the next three blocks. VC Selection is to obtain the VC of each iteration, the VC array is filled in this stage.

The second loop block is sorting of SA with the Radix Sort algorithm, based on the VC. The Radix sort algorithm was selected after test revealed that it was the fastest for most inputs. Although we considered using Insertion Sort for the cases when it was better, the extra control cost would make it yet slower. Therefore, only Radix Sort is used.

The third is the construction of the subgroups of sorting. Group Builder does this task based on Bounds Array and SS.

At this stage, the Bounds Array is modified according to the following criteria as shown in Figure 6:

- If the previous valid character of the suffix is different from the previous valid character of the previous suffix (see first column in Figure 6), it is necessary to mark the Bounds Array with '1' in the same position of the suffix. This is because at that point they are part of different subgroups.
- If the SS array, in any position has a value of '1', it is necessary to mark the Bounds Array with '1' in the same position. This is because that subgroup contains only one character, this means that it is totally sorted.
- If the suffix was already covered, it is necessary to mark the Bounds Array with '1' in the same position. This is because the suffix is totally sorted. A suffix has been covered when the VC for the current iteration is $.

This process is performed for each subgroup, for this, it is necessary to check whether there are more subgroups before continuing with the next iteration.

All this is done iteratively until all the suffixes are totally sorted. The end of the iterative process is determined by an auxiliary variable that does the logic operation AND with each element of the SS array; if any suffix has not been sorted yet, the process is repeated.

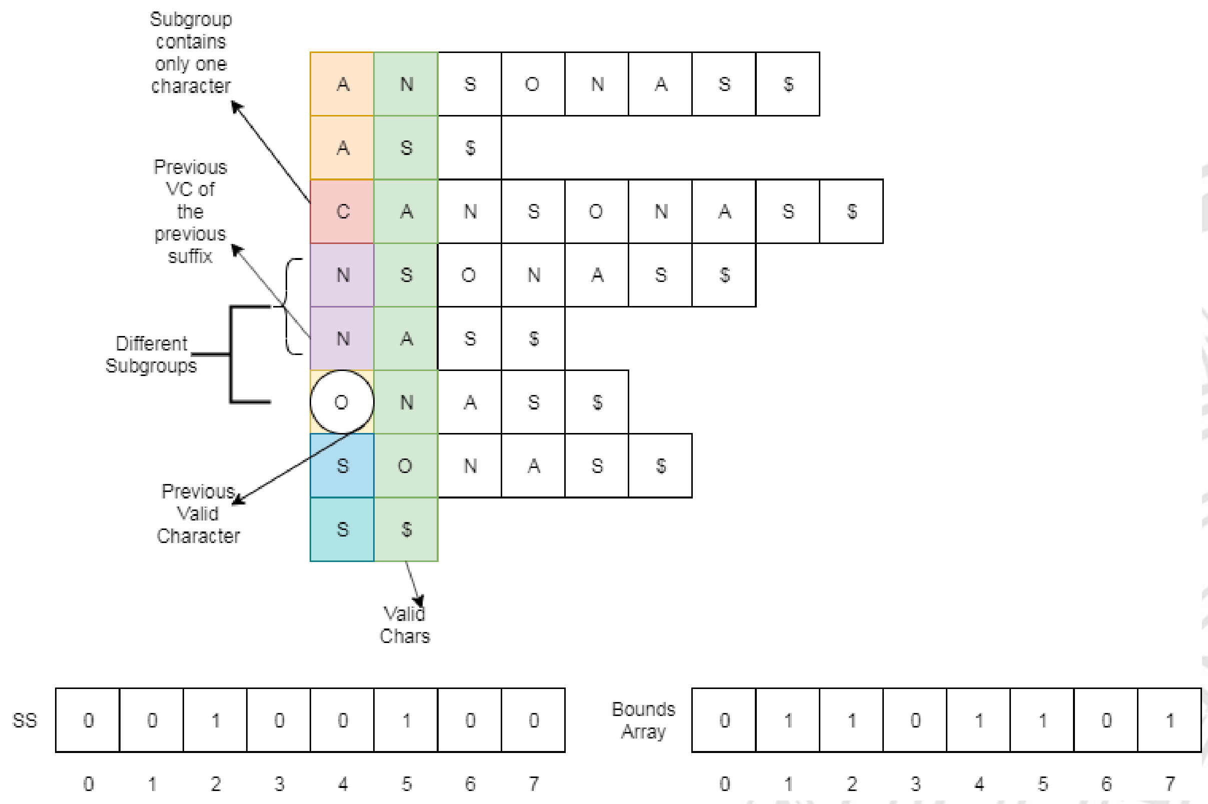

Figure 6. Second iteration of the suffixes array construction with the criteria that use de *GroupBuilder* for fill the Bounds Array.

3.4. **Parallel Versions**

After validating the Sequential version, two parallel versions were developed considering two design approaches. In the first, called Intergroup Parallel SAC algorithm, due to the conformation of the subgroups, the job of these is divided between the threads, this is an approach, one thread is responsible for reviewing the bounds array and deploy the other threads to sort a subgroup, every thread obtains the VC, sort them, and make the new subgroups as shown in the Figure 7.

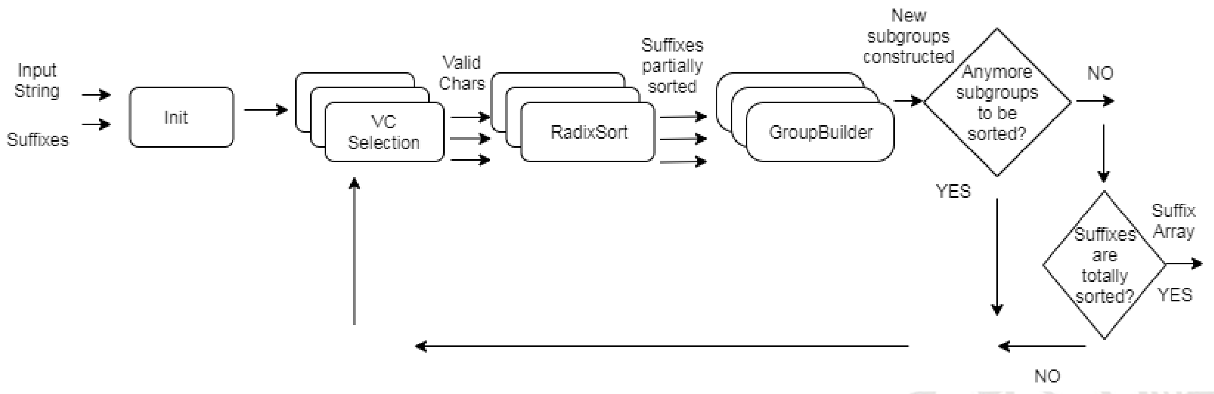
Input String
Suffixes
Init
VC Selection
Valid Chars
RadixSort
Suffixes partially sorted
GroupBuilder
New subgroups constructed
Anymore subgroups to be sorted?
NO
YES
Suffixes are totally sorted?
Suffix Array
YES
NO

Figure 7. Block diagram for the Intergroup Parallel SAC algorithm, with the job divided by threads.

Based on a parallel version of Radix-Sort [5] and VC Selection, the second version called Intragroup Parallel SAC algorithm was developed, in which in the moment of VC selection and sorting, the threads are deployed, as shown in the Figure 8, but in the final of every one is necessary to synchronize, also, the *GroupBuilder* is realized only by one thread.

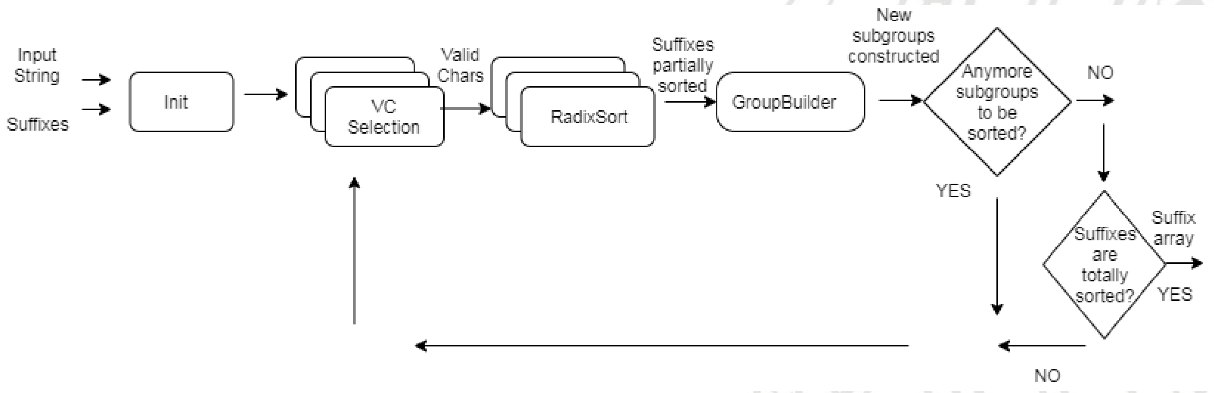
Input String
Suffixes
Init
VC Selection
Valid Chars
RadixSort
Suffixes partially sorted
GroupBuilder
New subgroups constructed
Anymore subgroups to be sorted?
NO
YES
Suffixes are totally sorted?
Suffix array
YES
NO

Figure 8. Block diagram for the Intragroup Parallel SAC algorithm, the threads are deployed in the *VC Selection* and *RadixSort*.

After the design and implementation of the algorithm, the performance tests done with Vtune showed that in the Intergroup Parallel SAC and Intragroup Parallel SAC versions, a lot of time was spent in the administration of the threads (creation and destruction), therefore other two versions were developed:

- Hybrid Intergroup: it is a combination between the intergroup parallel SAC and the Sequential version. From a certain threshold value of the subgroup size, no more threads are deployed and the sequential execution is invoked instead for the remaining subgroups sorting.
- Hybrid Intragroup: it is a combination between the intragroup parallel SAC and the Sequential version. From a certain threshold value of the subgroup

size, no more threads are deployed in the Radix-Sort algorithm and VC selection and they are executed sequentially.

4. Results and discussion

The performance tests were executed using as input the dataset DNA of Pizza & ChiliCorpus, as mentioned in section 3.1. The size values of the input strings were: 1, 5, 10 and 50 million characters.

The objective was to compare the different strategies implemented, look at the scalability of all versions and analyze the speed-ups obtained. Tests were executed with 2, 4 ,8, 16, 32, 64 threads on a workstation with a dual Intel Xeon Gold 6152, with 44 cores, 88 threads and 192 GB of RAM.

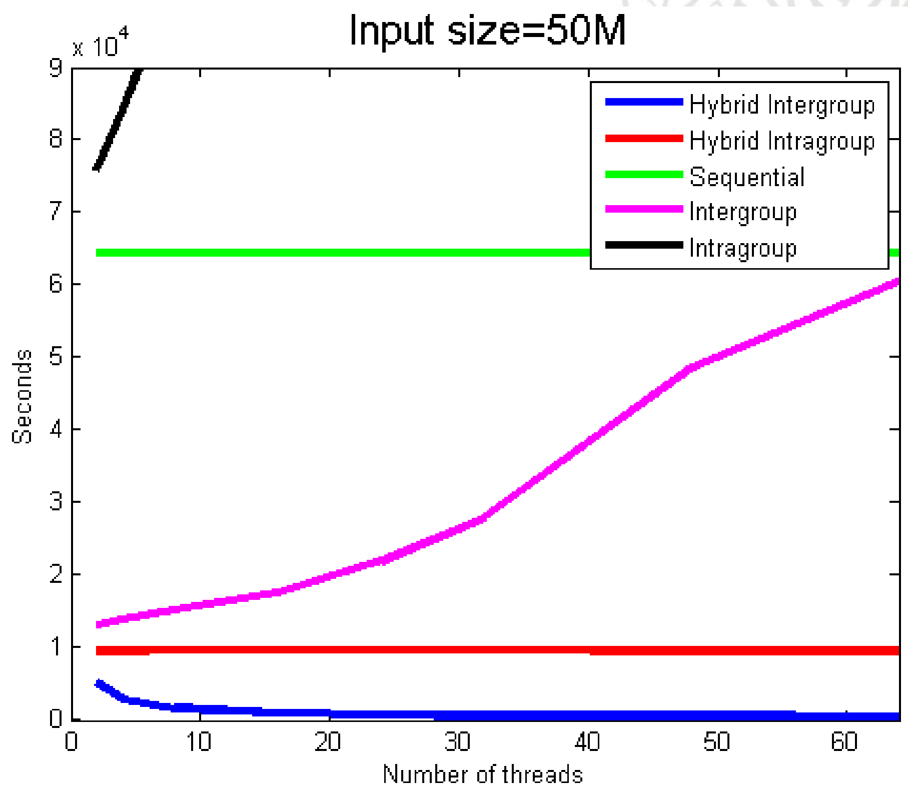

Figure 9. Comparison of the execution time between the different versions implemented with a 50 M size input and varying the number of threads.

Of the five versions developed, the Intragroup Parallel SAC version was the slowest, more than the Sequential as shown in Figure 9. As the number of threads increases, the runtime of the Intragroup version worsens; this is because every time that the Radix-Sort algorithm and the VC Selection are invoked, threads are created and many times they only have to process very small groups making the thread administration overhead relatively more expensive.

Between the Intragroup and Hybrid intragroup versions, the difference is from certain threshold value of the subgroup size, the algorithm continues to execute sequentially.

The Hybrid Intragroup version improves the execution time but it does not benefit from parallelism as shown in Figure 9, because as the number of threads increases the execution time remains constant. With the Hybrid Intragroup algorithm is possible to save time of administration of the threads for the small groups, but it is not scalable.

The Intergroup parallel SAC algorithm is better at runtime compared to the Sequential version but as the number of threads increases, the runtime also does. This is because a lot of time is spent on the administration of the threads (creation and destruction) since for small groups it is more expensive in time to create the threads than the group sorting.

In all versions, as the size of the input is greater, the runtime also increases but in almost all of them that increase is almost linear and with large slopes. In the best version, the increase in time is not more significant as shown in Figure 10.

As the input size increases, the time difference between the versions becomes more significant and the use of the best version becomes more relevant.

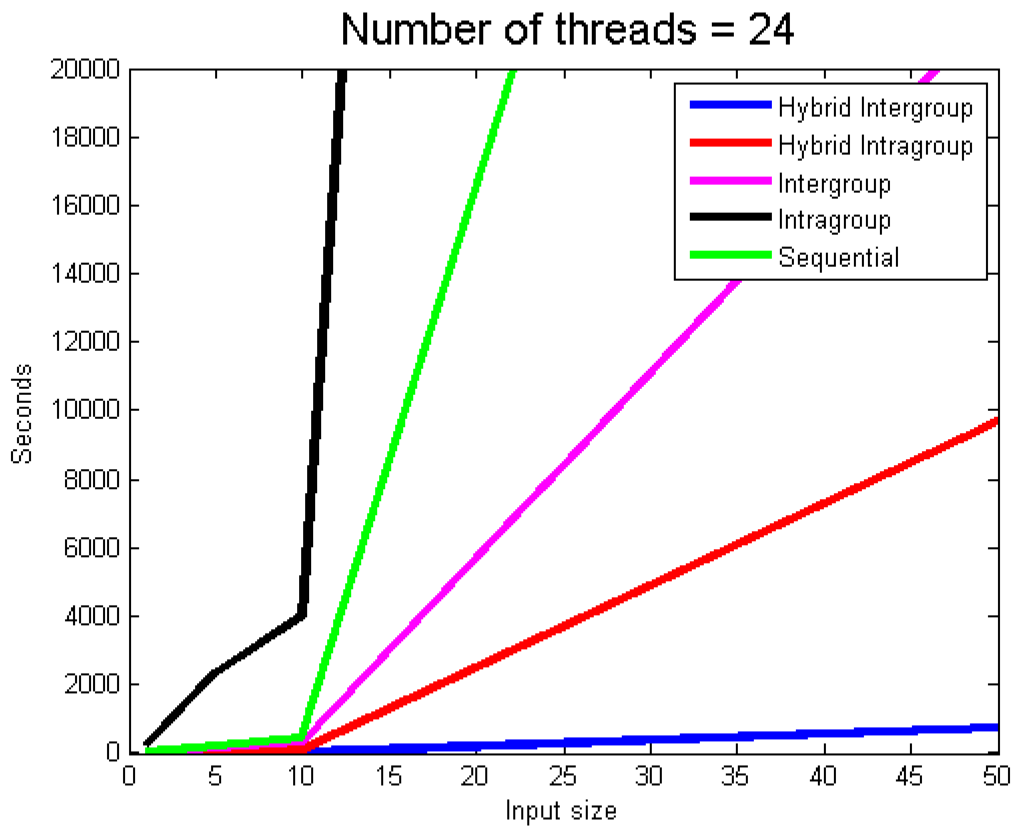

Figure 10. Comparison of the execution time between the different versions, varying the input size and with a fixed number of threads.

The best version is the Hybrid Intergroup, there is a higher speed-up and it is scalable as shown in Figure 11. It has a good distribution of tasks between the threads responsible of each subgroup and the combination with the sequential algorithm also reduces the time of administration of the threads, spending most of the time in the execution of the Radix-Sort algorithm.

The speed-up increases up to 48 threads, from there it is more or less constant and there is no significant improvement in runtime. It is important to keep in mind that in this version only as many threads are created as needed, for example, if in one of the iterations there are only 4 subgroups of ordering and it is configured for 32 threads, only 4 threads are actually created which makes it more efficient.

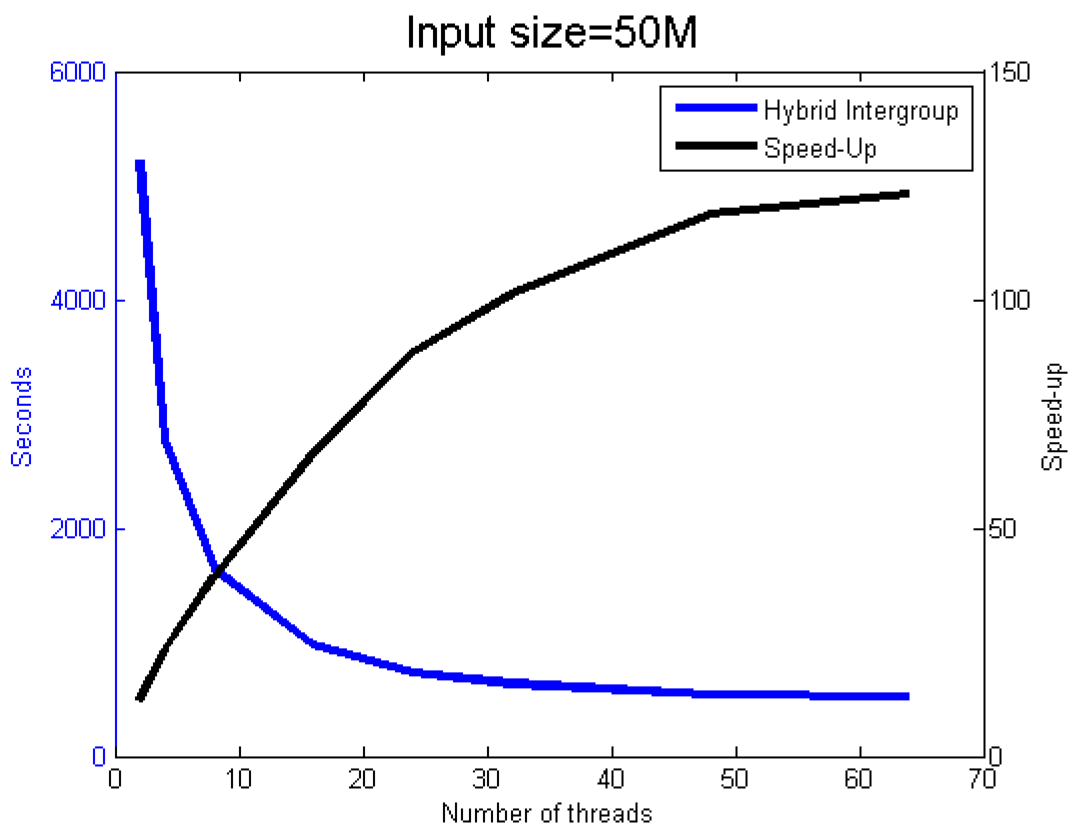

Figure 11. Execution time and speed-up of the Hybrid Intergroup version.

5. Conclusions

In this thesis, an algorithm for the construction of suffix arrays has been presented since it is a very important process within the compression of DNA and it needs considerable time to be built. Then, parallel versions were developed with the aim of improving the execution time. Latency and scalability tests were realized, obtaining as a best result the Hybrid Intergroup version with a maximum speed-up of 123 with respect a Sequential version and input size of 50 million and 64 threads.

- When studying some sorting algorithms and performing tests to choose which one to use for the construction of suffix array, it was found that from an input size less than or equal to 25 elements the Insertion-Sort is better with respect to the Radix-Sort that was the chosen finally. The reason for not using both depending on the input size is that the time saving is not as significant since it would be necessary to include another conditional to choose which of the two algorithms will sort a certain group.
- The best performing version was the Hybrid Intergroup, since tasks are better distributed between the threads and then stop to be created when the task

of administrating them is more expensive than their contribution to the algorithm, for that reason, its combination with the sequential version is beneficial.

- As the size of the input increases, the relation between the time in which the threads work and the time of their administration is greater, which makes the use of threads more relevant.
- The use of threads doesn't always translate into less execution time since their administration also requires time and an example of this were the Intergroup and Intragroup versions, which were purely parallel and it was more expensive to create and destroy a thread for small sorting groups.
- Assigning multiple independent tasks to a thread is more efficient, an example is the intergroup approach since in this, the thread is responsible for doing all the process to a sorting group during the iteration, in contrast to the intragroup approach since the threads are deployed to do only one particular task among all. Synchronizing all of them for such small tasks will be more expensive.

6. REFERENCES

[1] Guerra Aníbal, Efficient Storage of Genomic Sequences in High Performance Computing Systems, PhD Thesis, UdeA.

[2] Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. In:Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms.pp. 319–327. SODA '90 (Jan 1990)

[3] Larsson, N.J., Sadakane, K.: Faster suffix sorting. Theoretical Computer Science387(3), 258–272 (2007)]

[4] Johannes Fischer, Florian Kurpicz : Dismantling DivSufSort

[5] H. Wang's, "A faster openmp radix sort implementation," 2017, accessed: 2019-10-14. [Online]. Available: *https://haichuanwang.wordpress.com/tag/algorithm/*