Scalable Suffix Array Construction using MapReduce

Computational Genomics

Aidan Fowler, Lavanya Sivakumar, Kartik Thapar

Date: December 6, 2013

Abstract

One of the most important applications in computational genomics is sequence alignment. It is used for many applications such as analyzing genome expression, mapping differences in DNA between individuals, assembling genomes of organisms, as well as many others. One of the most important structures necessary for sequence alignment is the suffix array. It is important because most sequence alignment procedures require a pre-computed index of the sequence to aid in the alignment. Usually, these suffix arrays take many hours to construct if the sequence is large (a genome). Recently, linear time algorithms have been discovered that reduce this computation time. Below, we go a step further and introduce a new parallel scalable linear time algorithm, mrDC3, that computes a suffix array for a given reference text. Our algorithm was not tested on a large scale but in theory it should scale up linearly.

1 Introduction

1.1 Suffix Arrays & Applications

A suffix array is a lexicographically sorted array of all the indices of suffixes of a string. Suffix arrays are a simple and powerful data structure for text processing that can be used for a large variety of practical application, including full text indexes, data compression, string matching, genome analysis, and many other applications. It particularly has many applications in the field of computational biology.

Example SA: ”yabbadabbado”

```
12: $
1: abbadabbado$
6: abbado$
4: adabbado$
9: ado$
3: badabbado$
8: bado$
2: bbadabbado$
7: bbado$
5: dabbado$
10: do$
11: o$
0: yabbadabbado$
```

1.2 Goals

The suffix array has many applications. It is useful in string matching, genome analysis and text compression. A particularly useful application is using it as a full text index. One can do a binary search on a suffix array to find all occurrences of a pattern P in text T. In the recent times, there have been a few linear time algorithms discovered for creating suffix arrays. The one that is of particular interest to us is the DC3 algorithm. This algorithm makes it possible to create a suffix array for huge inputs in linear time. Our main goal for this project is to implement a parallel scalable implementation of a linear time suffix array creation algorithm. There already exists such an implementation called pDC3 created by Fabian Kulla and Peter Sanders [3]. The problem with this algorithm however, is that it is implemented using MPI (message passing interface). This is not the most useful implementation for the every day researcher as MPI requires substantial low latency and high bandwidth interconnects that are not generally available. We decided to use the MapReduce programming model using the Hadoop framework which available as open source. You can run it on a local machine or scale up as much as you want using Amazon Web Services Elastic Map Reduce. Our goals are mentioned below:

- to research fast (and parallel) suffix array construction algorithms.
- to implement a parallel implementation of the DC3 (skew) algorithm using MapReduce using the Hadoop streaming API.
- to test the parallel algorithm and compare it with other linear time algorithms

2 Prior Work

The DC3 algorithm was originally formulated by Juha Kärkkäinen, Peter Sanders and Steven Burkhard as a way to construct suffix arrays in linear time. This was motivated by the fact that at the time, linear algorithms were only possible where inputs consisted of a constant alphabet – DC3 generalized this to all strings [2].

A lot of optimizations were made to the Kärkkäinen and Sanders DC3 implementation by Puglisi et al (2007). Some of these changes involved shortening the recursion string, memory conservation, etc. They also stated in the aforementioned paper that linear suffix array creation times do not necessarily lead to better performance [5].

Further, at that point, there were only a few attempts to parallelize suffix sorting. Futamura et al. attempted a parallelization of suffix array sorting, but their algorithm was based on string sorting, which was inefficient [1].

Another attempt [4] parallelized both suffix array and Burrows-Wheeler Transform (BWT) sorting, but this was done without the use of a linear time algorithm - suffix array sorting instead used an $nlog(n)$ algorithm.

Eventually, a method of parallelizing DC3, known as pDC3 was created which implemented the algorithm using MPI. Their paper shows large improvement with respect to the original serialized algorithm, and is shown to be scalable [3].

3 Methods & Software

3.1 O(nlgn) Algorithm

Our first attempt at a parallel implementation to solve for a suffix array was to implement an $O(n \lg n)$ time algorithm using the MapReduce model [4]. The algorithm is as follows:

1. Prepare a file with a list of suffix positions from the reference text. Each mapper gets a copy of the reference string.

2. The mappers iterate over the file and emit key value pairs with the prefix of the current suffix as a key and the suffix index as the value. We select a prefix length that is much shorter than the reference text so that there are few suffixes assigned to a batch. For this case, we chose the 10 as the length.

3. The output from the mappers is shuffled so all the pairs with the same key go to the same reducer (aggregates suffixes which begin with the same prefix).

4. In the last stage, the reducer sorts the suffixes within the batches using a string sorting algorithm and outputs the suffix array. This stage was not implemented.

We abondoned any further work on this algorithm as we wanted to focus on linear time algorithms and their possible parallel implementations.

3.2 Understanding DC3

The DC3 algorithm can be explained in the following 4 steps:

3.2.1 I. Construct a Sample

In this explanation, the following text is used for reference: ‘*alphabeta*’

For $k = 0, 1, 2$, define:

$$B_k = \{i \in [0, n] | i \ mod \ 3 = k\}$$

- Sample positions: $C = B_1 \cup B_2$; Sample Suffixes: S_C
- Non-Sample positions: B_0; Non-Sample Suffixes: S_{B_0}
- `B1 = {1,4,7}`, `B2 = {2,5,8}`, `C = {1,4,7,2,5,8}`

3.2.2 II. Sort Sample Suffixes

For $k = 1, 2$, define:

$$\begin{aligned} R_k &= [t_k t_{k+1} t_{k+2}][t_{k+3} t_{k+4} t_{k+5}]...[t_{maxB_k} t_{maxB_{k+1}} t_{maxBk+2}] \\ R &= R_1 + R_2 = (lph, abe, ta0, pha, bet, a00) \end{aligned}$$

The ordering in R gives us an ordering for the sample suffixes S_C. Therefore, we sort R to get the sorted sample suffixes. Using the ranks, create a corresponding string R' with the suffix ranks. If the string has duplicates, we need to recursively find the suffix array of R'. R' is found by traversing original R, doing a dictionary lookup for triples, and substituting in the associated ranks. This, then gives us the ordering for the sorted sample suffixes. In this case, we have:

```
Sorted-Sc = [8, 4, 5, 1, 2, 7]
Suffix-Ranks = [-, 4, 5, -, 2, 3, -, 6, 1, 0, 0, 0]
```

3.2.3 III. Sort Non-Sample Suffixes.

For suffix $S_i, S_j \in S_{B_0}$:

$$S_i \leq S_j \Leftrightarrow (t_i, \textsc{Rank}(S_i + 1)) \leq (t_j, \textsc{Rank}(S_j + 1))$$

Using the condition above, we can get sorted non sample suffix indices.

```
Sorted-Sb0 = [0, 6, 3]
```

3.2.4 IV. Merge Sorted Sample & Non-Sample Suffixes

Two separate conditions for if the comparison is between $S_{B_1} - S_{B_0}$ or $S_{B_2} - S_{B_0}$.

$$\begin{aligned} i &\in B_1 : S_i \leq S_j \Leftrightarrow (t_i, \textsc{Rank}(S_i + 1)) \leq (t_j, \textsc{Rank}(S_j + 1)) \\ i &\in B_2 : S_i \leq S_j \Leftrightarrow (t_i, t_{i+1}, \textsc{Rank}(S_{i+2})) \leq (t_j, t_{j+1}, \textsc{Rank}(S_{j+2})) \end{aligned}$$

We implement the sorted set union by comparing the first elements of either sets and incrementing the counter for the index we add to the suffix array.

```
Result = [9, 8, 4, 0, 5, 6, 3, 1, 2, 7]
```

3.3 Implementing Linear-DC3

A non-recursive version of DC3 and pDC3 was implemented in python for the purposes of testing and verifying intermediate values at every point.

For DC3, these intermediate values refer to the intermediate state and steps for the creation of R and R' string, sorted S_C, suffix ranks array, sorted S_{B_0} and finally the merge step for the part suffix array. The codebase was only used as a reference point to ensure the authenticity of data at every checkpoint step. For pDC3, the values, these intermediate values refer to the P (and sorted P) array and sample and non sample index sets.

The code for the non-recursive DC3 and pDC3 algorithms is located at:

```
/code/bin/implementations/non-recursive-dc3.py
/code/bin/implementations/pdc3-methods.py
```

3.4 Implementing `pDC3`

As described in section 2, the pDC3 algorithm is a scalable parallel algorithm that efficiently uses many processors as part of the execution time dependent on the input size depends linearly on input size divided by number of processors, i.e. for number of processors p and length of text n, the execution time is given as $O(n/p)$ plus other low order terms as compared to $O(n)$ for the linear time algorithm.

Parallel-DC3 was the second parallel algorithm that was implemented as part of the overall research. The algorithm was also implemented in python using the mrjob Hadoop framework as described in appendix 7. MapReduce and Hadoop are covered in section 3.5 as prelude to the discussion on mrDC3. The source code for the MapReduce implementation of the pDC3 algorithm is located at:

```
/code/bin/implementations/pdc3.py
```

Although the algorithm is very similar to DC3 in it's implementation, there are parts that differentiate them subtly. The algorithm creates triples and assign ranks to the R string and creates the R' string much like the way in DC3 except it refers to these in a different manner. It then creates different sets S_0, S_1 and S_2 that refer to the set of non sample suffixes and sets of sample suffixes respectively. The algorithm describes implements most of the steps in a very straightforward way unlike DC3 which is difficult to comprehend at times.

Description aside, the algorithm was only implemented partly. As can be seen from the code a very large part of the algorithm was implemented using different MapReduce steps. However, there was a big setback when the team was unable to implement the merge step. The merge step as described by the algorithm isn't crystal clear although one can implement the merge in a very similar way as described in DC3. The merge step presents 4 constraints that are valid for different elements in the set. A merge or a parallel merge weren't obvious to the team. Therefore, after spending a considerable amount of time to implement the merge in the way described in the paper, the team reverted to the original DC3 algorithm and implemented a parallelized version of it using MapReduce.

3.5 MapReduce DC3 — `mrDC3`

This section introduces the MapReduce programming model and later discusses the MapReduce implementation of the DC3 algorithm.

3.5.1 MapReduce

MapReduce is a data processing paradigm for condensing large volumes of data into useful aggregated results. It is a programming model for processing and generating large data sets. In the simplest of the terms, it is a technique for dividing work across a distributed system. Programs written in the MapReduce programming model are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication.

The MapReduce framework helps developers divide a query into steps, divide the dataset into chunks, and then run those step/chunk pairs in separate physical hosts. There are two steps in a MapReduce query:

- **Map:** This is the *data collection* phase. `Map` breaks up large chunks of work into smaller ones and then takes action on each chunk. In logical terms, `map` takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain:

  $$Map(k1, v1) \Rightarrow list(k2, v2) \tag{1}$$

  After creating pairs, the MapReduce framework collects all pairs with the same key from all lists and groups them together, creating one group for each key.

- **Reduce:** This is the *data collation* or *processing phase.* `Reduce` combines the many results from the map step into a single output. In logical terms, the `reduce` function is applied in parallel to each group, which in turn produces a collection of values in the same domain:

  $$Reduce(k2, list(v2)) \Rightarrow list(v3) \tag{2}$$

3.6 Parallelizing using MapReduce

To parallelize DC3 using the MapReduce framework, the algorithm had to be divided into two major map-reduce tasks:

- The first task is to create triples and sets that is inherently used to create the string R' which is used later for sorting sample suffixes.
- The second task is to create sorted sample and non sample suffixes using R'.

Given that the algorithm executes recursively, we needed to partition these tasks into their own individual MapReduce jobs. These jobs are described below:

- Finding R'
  - **Mapper**: The mapper takes the reference text as the input and based on the length of the text, it creates sets B0, B1 and B2. It then creates the string R and releases tuples with `(k, v)` pairs of triplets from R with zero values (not necessary). It also sends the length of the text, the padded text and the R string that will be used in either the next step or the next map reduce job.
  - **Reducer**: The reducer takes in the triples and ranks them as they come. It then allots unique ranks to all unique suffixes. By using a map between triples and ranks, it creates the corresponding string R' which is saved to disk for later use. We can only use one mapper in this as we need the complete dictionary map before we create the corresponding R' string.

- Finding Suffix Array
    - **Mapper**: In the mapper, the input is read from the file using Hadoop HDFS APIs. We created the values corresponding to the depth of the recursion, the text, length of the text and the R' in the previous job. The mapper then then creates the sorted sample suffix set by recursing from the deepest level using R' and the suffix arrays whenever required. It then creates suffix ranks to create the set of sorted non-sample suffixes, after which it merges the two sorted sets. The first suffix array is created for the deepest level which is then used as R' for all the other levels.

3.6.1 Implementation

The algorithm was implemented using `mrjob` which is a wrapper over the hadoop streaming framework. The code was tested on the local nodes and external cluster using the map reduce jobs implemented using MRJob. In this, we only had one single file with both MapReduce jobs; we first created all the R' possible values and then unfolded the recursion by finding suffix array at the deepest level and using it in subsequent levels.

To get results on the cluster, the algorithm was implemented using the hadoop streaming API. The algorithm was broken down into 4 files with two mappers and two reducers that implemented the state and the recursion in a very different way. The recursion was implemented in the shell script, which is the driver for the program.

The codebase for the algorithms is located at:

```
/code/bin/mrjob/mrdc3.py
/code/bin/hadoop-streaming/
```

4 Results

4.1 mrDC3 scaling for large data sets

Our mrDC3 algorithm, as expected, scales linearly for large data sets. We ran our algorithm (lineardc3.py) using the mrjob python library. We ran it on a MacBook Pro 2.7 GHz Intel Core i7. It has 16 GB of DDR3 Memory and has 4 processors. The input reference texts for testing contain nucleotide strings ranging from sizes 100 to 1,000,000. The results are shown below. The results demonstrate that our algorithm is in fact linear.

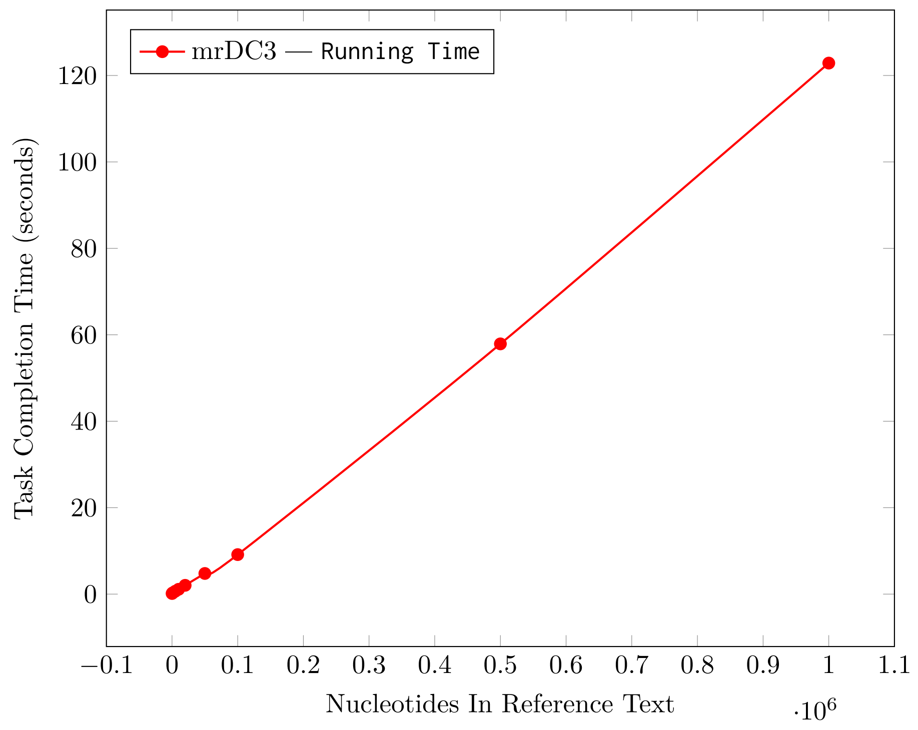

Figure 1: Running time of mrDC3 linear implementation for reference texts up to 1,000,000 nt long

4.2 Comparing mrDC3 with other algorithms

In order to compare our algorithm with the DC3 algorithm, we ran tests using code based on an implementation of DC3 written by Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. This code can be found at https://code.google.com/p/dc3/. We ran the tests on the same MacBook mentioned above using the same reference texts as inputs. Running this algorithm, we were able to verify that our algorithm produces the same exact output. The results for the running time of this algorithm are below. The DC3 algorithm was implemented in C++ whereas our algorithm was implemented in python. The two programs differ in running time by a factor of 100. This is to be expected as in general, python code can run up to 400 times slower than C++. This is because Python is interpreted and C++ is compiled. Python has no primitives, every variable is an object. Because of this Python lists take more time to parse.

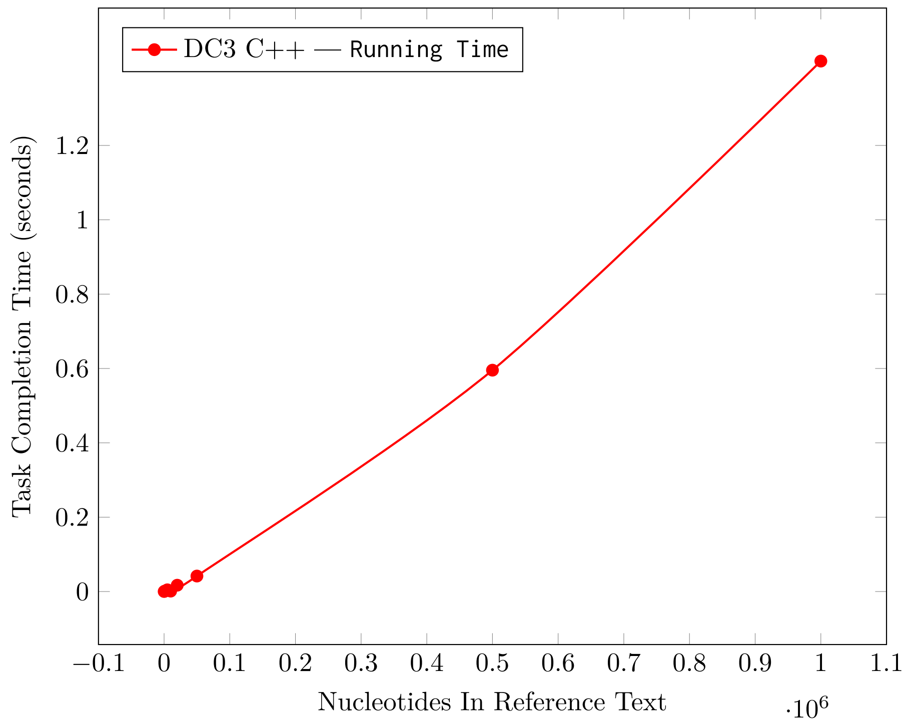

Figure 2: Running time of DC3 C++ implementation for reference texts up to 1,000,000 nt long

5 Future Work

So far, DC3 using the MapReduce framework in Hadoop was fully parallelizable, including the recursive step. However, there are many areas that will need investigation in the future. A large amount of data is stored at every point in the recursion tree – minimization of this would lead to much improved space efficiency.

We will, importantly, need to test the algorithm on a larger scale to further confirm its scalability.

Additionally, our algorithm was much slower than the pDC3 implementation. This may have been for a variety of reasons – one likely cause is that this is due to the use of Hadoop itself with an iterative algorithm. As a follow-up to this, investigation is needed into the possible use of Apache Spark instead of Hadoop, which is built around the use of iterative algorithms. Another reason for this could be the use of Python, as opposed to previous implementations in C++. Better memory management could be accomplished using another language with more direct control of memory usage.

6 Conclusions

A parallel version of the DC3 algorithm, mrDC3, was designed and implemented using MapReduce and Hadoop. DC3 has been parallelized before, the parallelized form of which is called pDC3 (Kulla and Sanders, 2007) However, mrDC3 is unique in that it implements a O(n) algorithm in the Map/Reduce framework using Hadoop streaming, which, due to its recursive properties, has not been attempted previously with any degree of success. We were also able to implement many of the optimizations suggested in the Puglisi paper [5]. Our benchmark times even scaled up linearly, as the algorithm predicted they would.

We were not able to match the speed of the linear DC3 implementation in C++. This was likely due to the use of the Python programming language in the implementation.

This linear implementation theoretically scales up far better than other implementations of suffix array construction algorithms. Efficient algorithms for suffix array construction are extremely important, as sorting a large list of strings is an inefficient task by nature. In implementing this algorithm, we have shown a a more scalable and efficient algorithm for suffix array construction and a strong alternative algorithm to pDC3.

7 Appendix A - Software

8 Appendix B - Individual Contributions

8.1 Software

8.1.1 mrDC3 USING MRJob

Algorithm

- Design: Kartik Thapar
- Algorithm: Kartik Thapar
- Implementation: Kartik Thapar
- Code: Kartik Thapar

Solving Recursion

- Design: Kartik Thapar, Aidan Fowler
- Algorithm: Kartik Thapar
- Implementation: Kartik Thapar, Aidan Fowler, Lavanya Sivakumar
- Code: Kartik Thapar, Aidan Fowler, Lavanya Sivakumar

8.1.2 Streaming mrDC3

Algorithm

- Design: Kartik Thapar

- Algorithm: Kartik Thapar
- Implementation: Kartik Thapar
- Code: Kartik Thapar, Aidan Fowler, Lavanya Sivakumar

Implementing recursion using shell scripting & saving state in HDFS

- Design: Kartik Thapar
- Implementation: Kartik Thapar
- Code: Kartik Thapar

8.1.3 Other Implementations

Suffix Array using Map/Reduce

- Design: Aidan Fowler
- Implementation: Aidan Fowler
- Code: Aidan Fowler

Non-recursive DC3

- Code: Kartik Thapar

pDC3 using Map/Reduce

- Design: Kartik Thapar
- Implementation: Kartik Thapar
- Code: Kartik Thapar

Benchmarking

- Aidan Fowler, Kartik Thapar

Presentation

- Kartik Thapar, Aidan Fowler, Lavanya Sivakumar

Report

- Lavanya Sivakumar, Aidan Fowler, Kartik Thapar

All group members participated in bug fixes over the course of the project.

References

[1] Natsuhiko Futamura, Srinivas Aluru, and Stefan Kurtz 2001 *Parallel suffix sorting*

[2] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt 2006 "Linear work suffix array construction." [p 918-936]*Journal of the ACM.*

[3] Fabian Kulla and Peter Sanders 2007 "Scalable parallel suffix array construction." [p 605-612]*Parallel Computing.*

[4] Rohith K. Menon, Goutham P. Bhat, and Michael C. Schatz 2011 "Rapid parallel genome indexing with MapReduce." *Proceedings of the second international workshop on MapReduce and its applications.*

[5] Simon J. Puglisi, William F. Smyth, and Andrew H. Turpin 2007 "A taxonomy of suffix array construction algorithms." *ACM Computing Surveys.*