TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

THESIS NUMBER: 071MSCS657

A MAP REDUCE MODEL TO FIND LONGEST COMMON SUBSEQUENCE USING NON-ALIGNMENT BASED APPROACH

SUBMITTED BY:

NARAYAN PRASAD KANDEL

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN COMPUTER SYSTEM AND KNOWLEDGE ENGINEERING

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

OCTOBER, 2016

A Map Reduce Model to Find Longest Common Subsequence Using Non-alignment Based Approach

By

Narayan Prasad Kandel

071-MSCS-657

Thesis Supervisor

Prof. Dr. Shashidhar Ram Joshi

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer System and Knowledge Engineering

Department of Electronics and Computer Engineering

Institute of Engineering, Pulchowk Campus

Tribhuvan University

Pulchowk, Lalitpur, Nepal

October, 2016

TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS, PULCHOWK

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

The undersigned certify that they have read, and recommended to the Institute of Engineering for acceptance, a thesis report entitled ”A Map Reduce Model to Find Longest Common Subsequence Using Non-alignment Based Approach” submitted by Mr Narayan Prasad Kandel in partial fulfillment of the requirement for the degree of Master of Science in Computer System and Knowledge Engineering.

....

Supervisor, Dr. Shashidhar Ram Joshi

Professor

Department of Electronics and Computer Engineering

....

External Examiner, Mr. ....

....

....

COPYRIGHT

Department Acceptance

The thesis entitled ”**A Map Reduce Model to Find Longest Common Subsequence Using Non-alignment Based Approach**”, submitted by **Narayan Prasad Kandel** in partial fulfillment of the requirement for the award of the degree of Master of Science in Computer System and Knowledge Engineering has been accepted as a bonafide record of work independently carried out by him in the department.

....

Dr. Dibakar Raj Pant

Asst. Prof. and Head of the Department

Department of Electronics and Computer Engineering,

Pulchowk Campus

Institute of Engineering,

Tribhuvan University,

Nepal.

ACKNOWLEDGEMENT

It is with great pleasure that I would like to thank my respected supervisor Prof. Dr. Shashidhar Ram Joshi for his expert guidance and mentorship. This piece of work would not have been possible without his supervision, support, motivation and co-operation within this period. I would like to acknowledge our Head of Department Dr. Diwakar Raj Pant, Prof. Dr. Subarna Shakya and Dr. Aman Shakya for their encouragement and valuable suggestions.

Also, I would like to thank our Program Coordinator (MSCSKE) Dr. Sanjeeb Prasad Panday for his encouragement and precious guidance. I would like to thanks, Mr. Janeshower Bohara for his incredible help and suggestions in doing the thesis. Along with that, I would like to thank all my friends for their incredible help in doing thesis.

Sincerely,
Narayan Prasad Kandel
(071-MSCS-657)

ABSTRACT

Biological sequences Longest Common Subsequence (LCS) identification has significant applications in bioinformatics. Due to the emerging growth of bioinformatics applications, new biological sequences with longer length have been used for processing, making it a great challenge for sequential LCS algorithms. Few parallel LCS algorithms have been proposed but their efficiency and effectiveness are not satisfactory with increasing complexity and size of the biological data. An non-alignment based method of sequence comparison using single layer map reduce based scalable parallel algorithm is presented with some optimization for computing LCS between genetic sequences.

Keywords: Bioinformatics, Longest Common Subsequence, MapReduce, Hadoop

Contents

List of Figures

List of Tables

Abbreviations

CREW Concurrent Read Exclusive Write

DNA Deoxyribonucleic Acid

LCS Longest Common Subsequence

PRAM Parallel Random Access Machine

1. INTRODUCTION

1.1. Background

Biological sequence comparison programs have revolutionized the practice of biochemistry, molecular and evolutionary biology. Pairwise comparison is the method of choice for many computational tools developed to analyze the deluge of genetic sequence data [1]. A fundamental operation in bioinformatics involves the comparison of genetic (DNA) sequences. The similarity between genetic sequences is a strong indicator of evolutionarily preserved characteristics. This property has been successfully used in determining pathologically important bacteria, viruses and fungi. Among the many sequence comparison tools for mining genetic information, an extremely common technique includes the alignment-based methods. These involve aligning the entire (global alignment, Needleman-Wunsch [2]) or smaller sections (local alignment, Smith - Waterman [3]) of the genetic sequences. The choice of global or local alignment is based on the type of analysis desired. However, both these methods are heavily dependent on the quality of sequence data. Slight discrepancies resulting from experimental or technical limitations can significantly affect the comparison results. An alternative approach of sequence analysis is becoming increasingly important in dealing with the exponential growth of genetic sequence data, classification and the grouping of organisms based on these sequences. Such alternative approaches include the alignment-free methods, which match the relative (as opposed to the exact) order of the base pairs in the sequence. Advancements in sequencing technology have provided a deluge of genetic data. The Genbank, a public repository of genetic sequence data, reported 194463572 sequence records in its 214th release on June 15, 2016. Analyzing such large datasets on uniprocessor machines is an extremely time-consuming process. It is imperative, therefore, to harness the power of high-performance computing to facilitate our understanding of this high throughput data.

1.2. Problem Definition

Longest Common Subsequences approach required a large amount of time for the deluge of genetic data which is represented by billions of characters. Time for finding longest common subsequence can be reduced tremendously if we can be able to solve the problem using non-alignment based distributed algorithm.

1.3. Objective

The purpose of this study is to investigate non-alignment based map-reduce model to compute longest common subsequence.

1.4. Scope of Work

When a new gene sequence is found, It is important to know what other sequences it is most similar to. Sequence comparison has been used successfully to establish the link between cancer-causing genes and a gene evolved in normal growth and development. One way of detecting the similarity of two or more sequences is to find their LCS.

Because of the larger number of gene sequences increasing day by day and extremely large length of single gene sequence, there is a need for parallel computing to find the LCS of gene sequences. MapReduce programming is emerging technologies for distributed computing especially in the case of big data. So, the non-alignment based LCS using map-reduce based model is really the good candidate of research.

The scope of this thesis work is to develop non-alignment based LCS finding algorithm using map-reduce based model.

2. Literature Review

A large number of research has been conducted in finding similarities between two gene species. The NeedlemanWunsch [2] algorithm was the first application of dynamic programming which provides a global alignment between two sequences. This algorithm leads to the evolution of various efficient LCS algorithms. It is only suitable if the two sequences are of similar length. The Hirschberg [4] algorithm evolved from Needleman- Wunsch algorithm provides optimize version of Needleman-Wunsch. Hunt-Szymanski [5] propose an optimization to Hirschberg algorithm. Various parallel algorithms like CREW PRAM model, Systolic arrays have been proposed in the earlier days to reduce the computation time. In the recent time Wan, Liu, Chen proposed Fast LCS algorithm [6]. Fast LCSs efficiency has been further improved by Efficient Fast Pruned LCS EFP_LCS [7]. A parallel LCS algorithm [8] based on dynamic programming has also been proposed. Li, Wang Bao [9] tried to solve LCS problem using automaton based technique in multi-level hadoop mapreduce. Beside that, Bohara Joshi [10] implement multi-level alignment based hadoop mapreduce technique to solve the lcs problem.

2.1. Needleman-Wunsch algorithm

The NeedlemanWunsch algorithm performs a global alignment of two sequences. It is commonly used in bioinformatics to align protein or nucleotide sequences. The algorithm was published in 1970 by Saul B. Needleman and Christian D. Wunsch. The NeedlemanWunsch algorithm is an example of dynamic programming and was the first application of dynamic programming to biological sequence comparison. It is sometimes referred to as the Optimal matching algorithm. This global sequence alignment method explores all possible alignments and chooses the best one (the optimal global alignment). It does this by reading in a scoring matrix and a gap penalty (penalties) that contains values for every possible residue or nucleotide match and summing the matches taken from the scoring matrix.

2.2. Hirschberg algorithm

Hirschberg's algorithm [4] is a dynamic programming algorithm that finds the optimal sequence alignment between two strings. Optimality is measured with the Levenshtein distance, defined to be the sum of the costs of insertions, replacements, deletions, and null actions needed to change one string into the other. Hirschberg's algorithm is simply described as a divide and conquer version of the NeedlemanWunsch algorithm. Hirschberg's algorithm is commonly used in computational biology to find maximal global alignments of DNA and protein sequences. If x and y are strings, where $length(x) = n$ and $length(y) = m$, the Needleman-Wunsch algorithm finds an optimal alignment in $O(nm)$ time using $O(nm)$ space. Hirschberg's algorithm is a clever modification of the Needleman-Wunsch Algorithm which still takes $O(nm)$ time, but needs only $O(min\{n, m\})$ space.

2.3. Hunt-Szymanski algorithm

Hunt-Szymanski algorithm [5] present an improved version of the Hirschberg algorithm. It used row-wise processing technique where right to left traversal is done to find the lcs. It solves the problem of recovering an LCS in $O(|M|log(n))$ where $|M|$ denotes the number of all matches.

2.4. Fast LCS algorithm using Map Reduce

Li, Wang Bao [9] purpose finite automaton based technique that implements fast lcs [6] algorithm to find the multiple longest common subsequences. The authors suggested that time required for calculating MLCS is reduced significantly using FACC Technique in compared to the time required using fast LCS [6], it uses multilevel map reduce technique and map reduce is used particularly to construct successor table of different string. Multilevel map reduce program based on fast lcs algorithm is presented by Bohara Joshi in [10]. Though it is implemented via map reduce and time might be reduced if number of the cluster devices increase but much of the time is invested in creating, managing the map reduce jobs and waiting for the results of the previous job so it fails in utilizing the power of the distributed computing due to the multilevel map reduce strategy.

3. THEORETICAL BACKGROUND

3.1. DNA

Deoxyribonucleic acid (DNA) is a molecule that encodes the genetic instructions used in the development and functioning of all known living organisms and many viruses. Along with RNA and proteins, DNA is one of the three major macromolecules essential for all known forms of life. Most DNA molecules are double-stranded helices, consisting of two long biopolymers of simpler units called nucleotideseach nucleotide is composed of a nucleobase (guanine, adenine, thymine, and cytosine), recorded using the letters G, A, T, and C, as well as a backbone made of alternating sugars (deoxyribose) and phosphate groups (related to phosphoric acid), with the nucleobases (G, A, T, C) attached to the sugars. DNA is well-suited for biological information storage, since the DNA backbone is resistant to cleavage and the double-stranded structure provides the molecule with a built-in duplicate of the encoded information

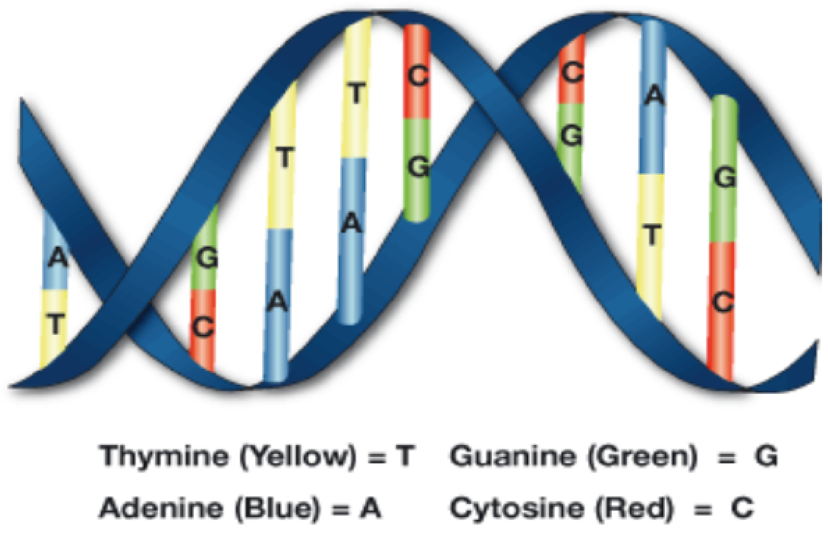

Figure 3.1: Structure of DNA Sequence

3.2. MapReduce

MapReduce [11] is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs and a reduce function that merges all interme-

diate values associated with the same intermediate key. Programs written in this functional style 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. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system. A typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Googles clusters every day. MapReduce provides an abstraction that involves the programmer defining a ”mapper” and a ”reducer,” with the following signatures:
Map: (value 1, key1) list (key2, value2)
Reduce: (key2, list (value2) list (value2).

3.2.1. Execution Overview

The Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of M splits. The input splits can be processed in parallel by different machines. Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., hash(key) mod R). The number of partitions (R) and the partitioning function is specified by the user.

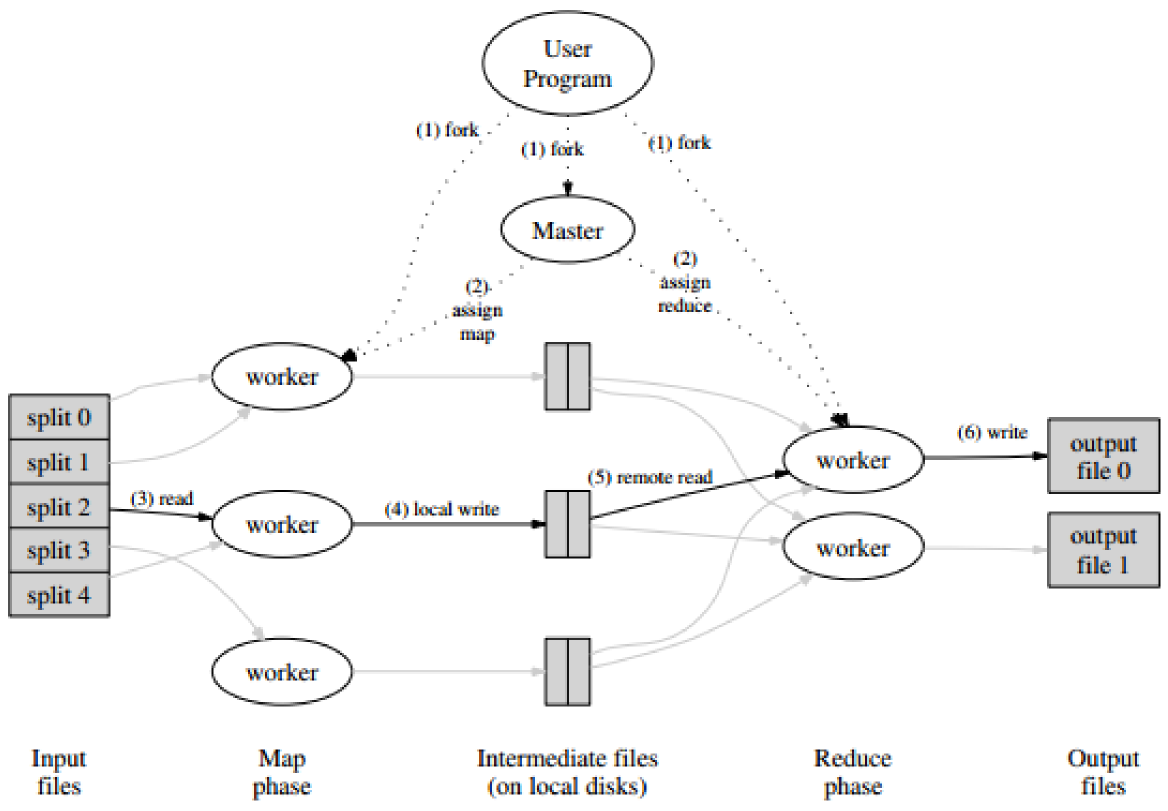

Figure 3.2: MapReduce Execution overview

Figure 3.2 shows the overall flow of a MapReduce operation in the implementation. When the user program calls the MapReduce function, the following sequence of actions occurs (the numbered labels in Figure 3.2 correspond to the numbers in the list below):

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.

2. One of the copies of the program is special the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.

3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.

4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the users Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code. After successful completion, the output of the mapreduce execution is available in the R output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these R output files into one file they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

3.3. Hadoop

Hadoop [12] is a popular open source implementation of MapReduce, which is a powerful tool designed for deep analysis and transformation of very large datasets which is inspired by Google's MapReduce and Google File System. It enables applications to work with thousands of nodes and petabytes of data.

Hadoop uses a distributed file system called Hadoop Distributed File System (HDFS), which creates multiple replicas of data blocks and distributes them on computer nodes throughout a cluster to enable reliability and has extremely rapid computations to store data as well as the intermediate results. The Hadoop runtime system coupled with HDFS manages the details of parallelism and concurrency to provide ease of parallel programming with reinforced reliability. In a Hadoop cluster, a master node controls a group of slave nodes on which the Map and Reduce functions run in parallel.

4. RESEARCH METHODOLOGY

The longest common subsequence algorithm finds the longest subsequence between two strings. In contrast to the substring, the subsequence denotes a series of letters from the string which while being in order, need not be consecutive. For example, between ATCG and CTCAG, the longest common substring is TC, while the longest common subsequence is TCG.

LCS can help identify the key nucleotides across genetic sequences and is considerably less affected by the occasional sequencing error. This method is also useful for identifying potential regions of small mutations by analyzing the portions of the string not present in the LCS.

4.1. Computing LCS using Row-wise processing Technique

The row-wise processing technique [5] is inherited from the traditional approach for filling the dynamic programming table. However, this time, we concentrate only on those table entries which correspond to a match. Each dominant match defines a new corner to a contour line. To maintain the columns where all contour lines cross the current row, we use the array MinYPrefix[1..p], where MinYPrefix[l] gives the Y-index where the l'th contour line is located. As the name of the array suggests, the value of MinYPrefix[l] may be regarded as a cursor, which indicates the minimum length prefix of Y that is needed to produce a common subsequence of length l with the first i elements of X. Value p denotes r(X[1..i], Y[1..n]), that is, the number of contour lines crossing row i. Initially, the values of MinYPrefix are initialized to 'undefined'.

Given the example strings X=abcdbb and Y=cbacbaaba, the values of the array change as follows (undefined values are represented by n+1; the leftmost entry acts as sentinel and is set to zero):

To maintain the MinYPrefix values when moving from row to row, we need the result

Table 4.1: Table for computing lcs

Row	MinYPrefix						
	0	1	2	3	4	5	6
0	0						
1	0	3					
2	0	2	5				
3	0	1	4				
4	0	1	4				
5	0	1	2	5			
6	0	1	2	5	8		

given in [5].

Update rule: Let us assume that we are processing row i. For each open interval MinYPrefix[l]..MinYPrefix[l+1], (l=0..r), find the matches (i,j) which fall into it (i.e. matches for which the j value is in the interval). The right boundary of the interval is kept unchanged, of no such match exists. Otherwise, it is updated to the smallest such j value (leftmost match in the interval). Note that the updates are simultaneous.

For example, when moving from row 2 to row 3 in the above example, we notice that X[3] = Y[4] and MinYPrefix[1] <4 <MinYPrefix[2], so we update MinYPrefix[2] to 4. The general scheme for advancing in the dynamic programming table is the following

```
begin
    for i:=1 to m do MinYPrefix[i] := n+1;
    MinYPrefix[0] := 0; r := 0;
    for i := 1 to m do
        /* Update the array values for row i. */
        for j := 0 to r
            if range[MinYPrefix[j]+1..MinYPrefix[j+1]-1]
                contains matches then
            begin
                MinYPrefix[j+1] :=
                       min{l|(i, l) is a match in this range};
                if j=r then r:=r+1;
            end
    return r;
end;
```

The Algorithm used to backtrack the LCS is as follow

```
begin
    last_char = True
    lcs_str1_index = []
    for i := lcs_length to 0
        for j := x_str_length to 0
            if i+1 > len(MinYPrefix[j-1]) or
                    MinYPrefix[j][i] < MinYPrefix[j-1][i]
                if last_char
                    last_char = False
                lcs_str1_index.append(j-1)
            elif MinYPrefix[j][i] == MinYPrefix[j-1][i] &
                    not last_char &
                    str1[j-1] = str2[MinYPrefix[j][i]]
                lcs_str1_index.append(j-1)
    lcs_str = ''
    for s_index in lcs_str1_index
        lcs_str = str1[s_index] + lcs_str
    return lcs_str
end;
```

In this backtracking algorithm, different logic is implemented between last char of the LCS index and other indexes. This is done in order to extract the LCS that has minimum ending index among all possible ending indexes.

4.2. Parallel Implementation

A scalable parallel version of the LCS algorithm purposed by Bastola [1] is outlined in 4.1. First, each string is divided across the processors and the LCS of the substrings in each processor (LCS1 and LCS2) are computed. Then the portions of strings (gray areas) that were beyond the first and last positions in the LCS are interchange and LCS for these previously unused strings is computed. Finally, the respective portions are combined to obtain the complete LCS.

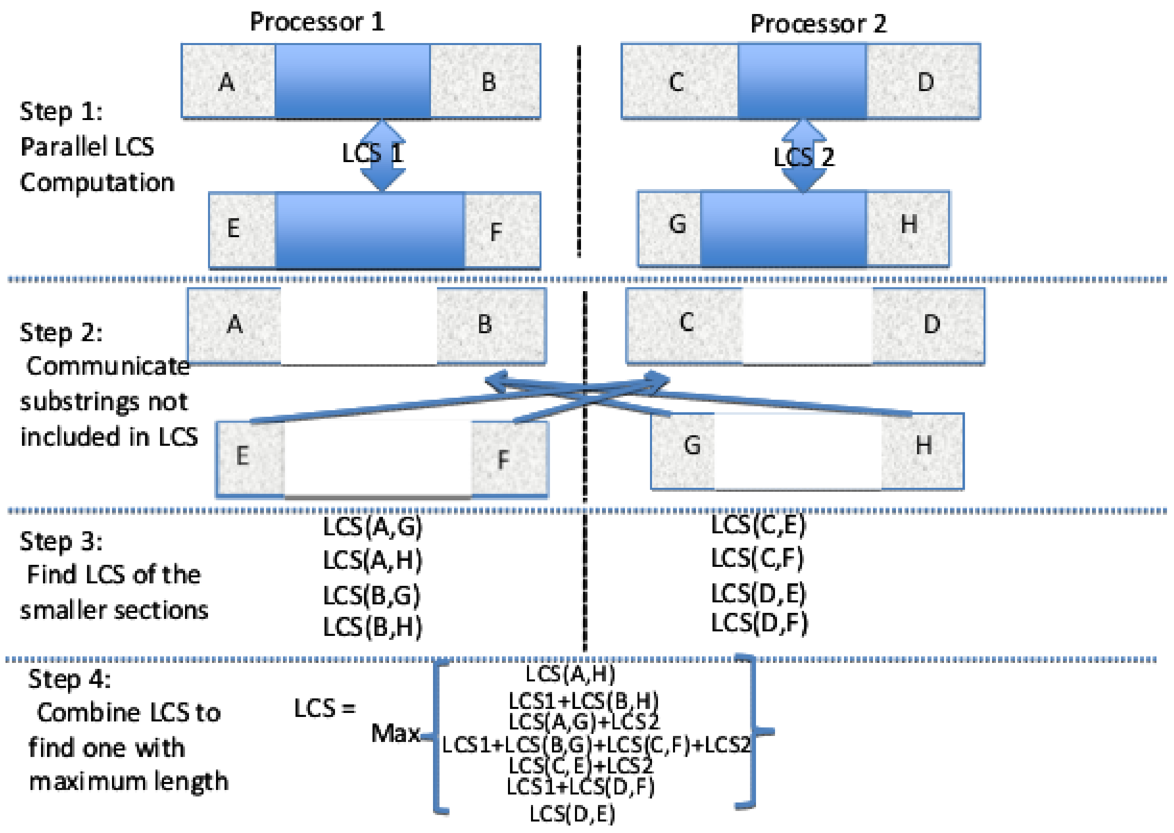
Processor 1
Processor 2
Step 1: Parallel LCS Computation
A
B
C
D
LCS 1
LCS 2
E
F
G
H
Step 2: Communicate substrings not included in LCS
Step 3: Find LCS of the smaller sections
LCS(A,G)
LCS(A,H)
LCS(B,G)
LCS(B,H)
LCS(C,E)
LCS(C,F)
LCS(D,E)
LCS(D,F)
Step 4: Combine LCS to find one with maximum length
LCS = Max
LCS(A,H)
LCS1+LCS(B,H)
LCS(A,G)+LCS2
LCS1+LCS(B,G)+LCS(C,F)+LCS2
LCS(C,E)+LCS2
LCS1+LCS(D,F)
LCS(D,E)

Figure 4.1: A Schematic Diagram of the Parallel LCS Algorithm

4.3. Map Reduce Strategy

In order to apply map-reduce strategy on the given two string, first, we do some preprocessing task to make data fit to map-reduce process. Those preprocessed data is fed to mapper process which applies row-wise LCS computing technique to compute LCS between two small sub-string. After that, parallel algorithm is applied in combiner and reducer to merger small sub-string LCS. The basic program flow is shown in 4.2.

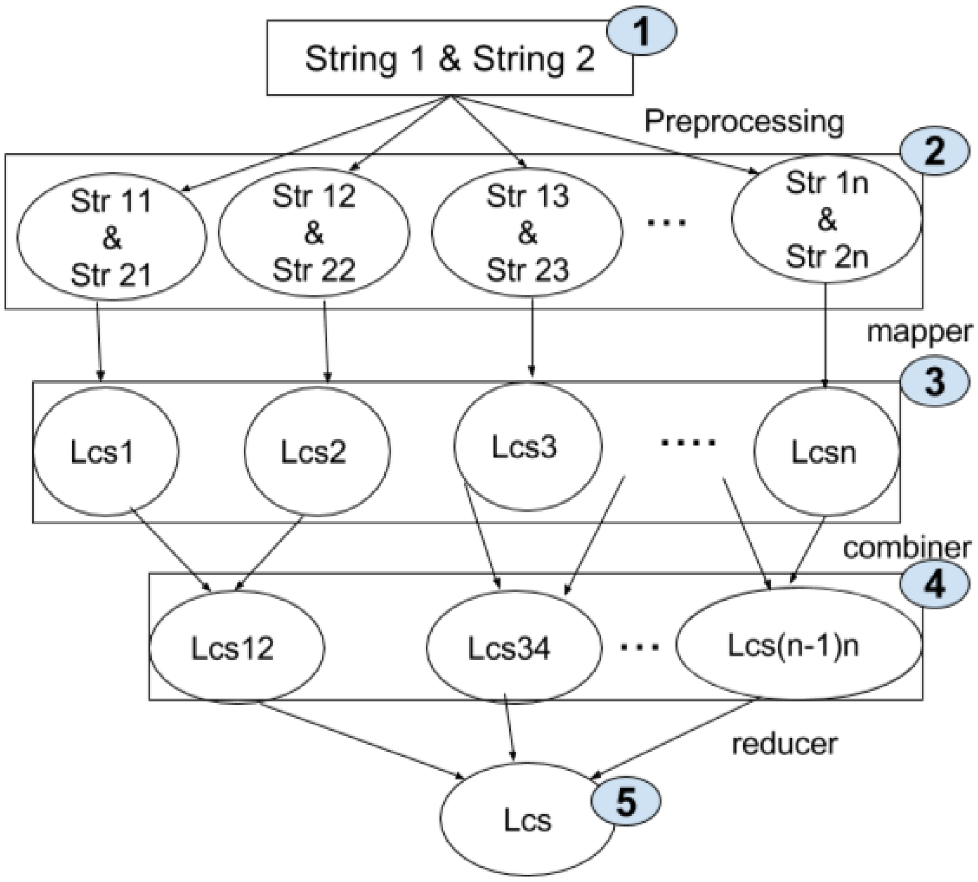

Figure 4.2: Program Execution Block Diagram

The basic block diagram of the program execution is shown in figure 4.2 and explained below.

1. Input string. It is converted to hadoop input by splitting it into multiple parts. The process of splitting given string to the multiple parts is known as preprocessing.

2. Splitted String. It is now fed to mapper to calculated the LCS of each small chunk.

3. Preliminary LCS. This is the output of the mapper. Now it is feed to combiner which used parallel algorithm to find intermediate LCS.

4. Intermediate LCS. This is output of the combiner. In reducer, Multiple intermediate LCS are merged together using parallel algorithm recursively until final LCS is generated.

5. Final LCS. An output of the reducer.

4.4. Data Collection

Sample data is collected from GenBank. (www.ncbi.nlm.nih.gov/genbank/)

4.5. Tools Used

For doing this thesis, following tools will be used

- Git - for Version Controlling
- PyCharm - ids for coding
- Sharelatex - for documentation
- Elastic Map Reduce framework

4.6. System description

System description that will be used to run the algorithm is as follow.

1. Local Development
   - OS: Ubuntu 14.04
   - Processor: Core i5
   - RAM: 8GB
   - Python: 2.7
   - Hadoop: 2.7.2
2. Hadoop Test Environment
   - instance type: m1.medium (Compute Unit:2, vCPU: 1)
   - OS: Ubuntu
   - RAM: 3.75 GB
   - Hadoop: 2.7.2

5. RESULT AND DISCUSSION

In implementing MapReduce strategy, the program is divided into 4 steps, that are 1. Preliminary Step, 2. Mapper Step, 3. Combiner Step and 4. Reducer Step. The output of the one step is fed to the consequent next step as input and final step output is received as program output. We have used JSON format as an intermediate data format. The example of input data and output data format for each of the processes is as follows:

1. Preliminary Step

   Here, we have to calculate the LCS of strings str1 and str2. First, we take partition size as 15 and divide each string as a 15 char substring and keep the string sequence order. This work is done in Preliminary Step.

   Input

   str1=ABCBDABATCGACGATCGGGGTTCTTCACCACGGGGTTCTTCACCAG
   AGTTATCT

   str2=BDCABACTCAGGCACCGCAGTGACAAAAGTCGCAGTGACAAAAGTCA
   GGACGGC

   Partition size: 15

   Output

   1 ”a”: ”ABCBDABATCGACGA”, ”index”: 0, ”b”: ”BDCABACTCAGGCAC”

   2 ”a”: ”TCGGGGTTCTTCACC”, ”index”: 1, ”b”: ”CGCAGTGACAAAAGT”

   3 ”a”: ”ACGGGGTTCTTCACC”, ”index”: 2, ”b”: ”CGCAGTGACAAAAGT”

   4 ”a”: ”AGAGTTATCT”, ”index”: 3, ”b”: ”CAGGACGGC”

2. Mapper Step

   The preprocess string is fed to the mapper. Mapper Process is responsible for finding LCS of the small substring of str1 and str2. Along with LCS, it also calculates A, B, E, F and its index which is helpful to calculate combine LCS in an upcoming step.

   Input

   Mapper1: ’a’: ’ABCBDABATCGACGA’, ’index’: 0, ’b’: ’BDCABACTCAGGCAC’

Mapper2: ’a’: ’TCGGGGTTCTTCACC’, ’index’: 1, ’b’: ’CGCAGTGACAAAAGT’

Mapper3: ’a’: ’ACGGGGTTCTTCACC’, ’index’: 2, ’b’: ’CGCAGTGACAAAAGT’

Mapper4: ’a’: ’AGAGTTATCT’, ’index’: 3, ’b’: ’CAGGACGGC’

3. Combiner Step

   Combiner step combines the output of the 2 mappers within the system to give intermediate output.

   Input

   Combiner1:

   Key: 0

   Value: [[0, ’A’: ’ABC’, ’a’: 0, ’B’: u”, ’E’: u”, ’lcs’: ’BDABATCAGA’, ’F’: ’C’, ’s2’: ’BDCABACTCAGGCAC’, ’s1’: ’ABCBDABATCGACGA’, ’f’: 15, ’b’: 15, ’e’: 0], [1, ’A’: ’T’, ’a’: 0, ’B’: u”, ’E’: u”, ’lcs’: ’CGGTAC’, ’F’: ’AAAAGT’, ’s2’: ’CGCAGTGACAAAAGT’, ’s1’: ’TCGGGGTTCTTCACC’, ’f’: 15, ’b’: 15, ’e’: 0]]

   Combiner2:

   Key: 1

   value: [[2, ’A’: ’A’, ’a’: 0, ’B’: u”, ’E’: u”, ’lcs’: ’CGGTAC’, ’F’: ’AAAAGT’, ’s2’: ’CGCAGTGACAAAAGT’, ’s1’: ’ACGGGGTTCTTCACC’, ’f’: 15, ’b’: 15, ’e’: 0], [3, ’A’: u”, ’a’: 0, ’B’: u”, ’E’: ’C’, ’lcs’: ’AGGAC’, ’F’: ’GGC’, ’s2’: ’CAGGACGGC’, ’s1’: ’AGAGTTATCT’, ’f’: 9, ’b’: 10, ’e’: 0]]

4. Reducer Process

   Finally, reducer step reduces the intermediate output from all combiner to one final lcs.

   Input:

   Key: lcs

   value: [[0, ’a’: 0, ’A’: u”, ’b’: 30, ’e’: 0, ’lcs’: ’BDABATCAGACCGGTAC’, ’f’: 30, ’s2’: ’BDCABACTCAGGCACCGCAGTGACAAAAGT’, ’s1’: ’ABCBDABATCGACGATCGGGGTTCTTCACC’, ’F’: u”, ’B’: u”, ’E’: u”], [1, ’a’: 0, ’A’: u”, ’b’: 25, ’e’: 0, ’lcs’: ’CGGTACAGGAC’, ’f’: 24, ’s2’: ’CGCAGTGACAAAAGTCAGGACGGC’,

’s1’: ’ACGGGGTTCTTCACCAGAGTTATCT’, ’F’: u”, ’B’: u”, ’E’: u”]]

Output:
”length”: 28, ”lcs”: ”BDABATCAGACCGGTACCGGTACAGGAC”

The parallel algorithm suggested by [1] highly depend on both starting index and ending index of LCS in both string. If we are able to extract LCS that lie in middle of both given string then the parallel algorithm would give us the sequential equivalent result in much more less time.

With modified LCS backtracking algorithm, for the repetition character, the index of the character that lies toward center is selected. The example cases are:

1. LCS(’TCG’, ’TCAGGGGGGGGGGGGGGG’) = ’TCG’
   index: str1 = [0,1,2], str2 = [0,1,3]
2. LCS(’TTTTTTTTTTTTTTTTCG’, ’TCAG’) = ’TCG’
   index: str1 = [15, 16, 17], str2 = [0,1,3]
3. LCS(’TCGGGGGGGGGGGGGGGG’, ’TCAG’) = ’TCG’
   index: str1 = [0,1, 2], str2 = [0, 1, 3]
4. LCS(’TCG’, ’TTTTTTTTTTCAG’) = ’TCG’
   index: str1 = [0,1, 2], str2= [0, 12, 10]

Here, cases 1, 2, 3 work as expected but case 4 doesn’t work as expected. This is because, when constructing matrix we only take least possible index of string_2 required to get LCS of length x when selecting n first character in string_1.

5.1. Complexity Analysis

The time complexity of core LCS computing algorithm is $O(|M|log(n))$ where $|M|$ denotes the number of all matches. The space complexity of the algorithm is $O(n^2)$.

5.2. Run Time of Algorithm

Performance is measured by running this algorithm for two input sequences. Table 5.1 shows the time taken to get result with a single node. Two input sequences are run for 10 times and the average time is computed. The time taken by this algorithm is listed below.
Time to compute LCS between 2 strings with length 17990 and 17990 with per process length of 1500, 500, 100 and 50 is shown in Table 5.1.

Table 5.1: Output time (in sec) comparison with different per process length of the string

S.N.	Per Process Length			
	1500	500	100	50
1	188	23	2.010	.940
2	190	22	1.951	.961
3	204	22	2.030	.919
4	197	22	2.024	.958
5	200	22	2.093	.961
6	191	23	2.012	.943
7	197	22	1.960	.961
8	195	22	1.981	.927
9	206	38	2.057	.909
10	204	24	2.021	.949
average	197	24	2.014	.943
lcs length	17984			
actual length	17984			

As shown in Table 5.1, time taken by algorithm is decrease with decreasing number of per process string length.

5.3. Hadoop Runtime Analysis

The scalability of the algorithm is studied by measuring the time taken to compute LCS on a different number of node which is shown on Table 5.2 and Table 5.3 respectively.

Table 5.2: LCS time (in sec.) comparison on different per process string length and processor number of 2 string with length 10,000

No. of Processor	Per Process String Length				
	200	500	1000	1500	2000
1	132.9	184	262.4	499.7	609
3	119.8	133.2	160.6	332.3	435
5	116.6	128.1	182.4	286.5	343.5
10	-	109.5	-	-	-
20	-	106.2	-	-	-
lcs length	6317	6416	6464	6466	6485
actual length	6514				

Table 5.3: LCS time (in sec.) comparison on different processor number of 2 string with length 200,000

No. of Processor	Per Process String Length
	500
10	609.8
20	587.3
lcs length	128,667
actual length	130,814

With reference to the time taken to computer LCS as presented in Table 5.2, it is fair to say that the algorithm is scalable. The accuracy of the algorithm goes on increasing with increase in per process length as with increase in per-process length there are less number of parallel merge.

Table 5.3 shows the time required to compute LCS of the strings with length 2,00,000. It is discovered that the time required to compute the LCS of strings having length 2,00,000 with 500 per process string length using 10 nodes is equal to the time taken to compute the LCS of strings having length 10,000 with 2000 per process string length using 1 node. This shows that, it is feasible to compute the LCS between 2 strings having very large length using group of low processing power computer. This will help to do cost effective analysis of the similarities between genetic sequences within no time.

The time required to find the LCS between 2 string is better than the MapReduce algorithm proposed by Bohara et. al [10]. Bohara et. al performed the study to find whether it is possible or not to find LCS using MapReduce approach. Bohara suggested that it is possible to calculate the LCS using MapReduce and time required will be reduced with the addition of the node. But Bohara didn't included the length of the input string so it is difficult to conclude how fast this algorithm is.

5.4. Verification

The output result is verified by comparing it with sequential program result. Total runtime is calculated by taking an average of 10 runtimes and it is later compared with corresponding sequential program runtime.

6. Conclusion

A basic model for MapReduce-based parallel algorithm for gene sequence comparison has been developed. Although there are few parallel algorithms for LCS computation, they are not reliable as the MapReduce-based solution in the context of fault tolerance and concurrency control. This MapReduce-based model handles all the different aspects of distributed computing from load balancing to synchronization automatically. The algorithm is highly scalable and cost effective. Hence, the large number of gene data can be processed at short period if we use a large number of nodes created from commodity computers. The time to calculate LCS can also reduce by decreasing the per process string length. But it might result in a decrease in LCS length, as there is trade-off between per process string length and accuracy.

6.1. Limitation

The parallel algorithm might not return longest common subsequence as multiple starting and ending point are possible for the same length of the LCS substring between two strings. This is because, with a different value of starting and ending sequence, we have different A, B, E, F. So, there is always a possibility of not having longest subsequence.

6.2. Further Enhancement

There is a special case where this algorithm dont work which is listed in limitation. So we can optimize algorithm to overcome the listed problem. This thesis can be extended to study relation between occurrences of multiple LCS across the same species to intra-species mutations. Also, we can enhance it to compare the runtime between different distributed platform like Apache Spark, Google dataflow and Hadoop cascading.

References

[1] Bhowmick, Sanjukta, et al. ”A Parallel Non-Alignment Based Approach to Efficient Sequence Comparison using Longest Common Subsequences.” Journal of Physics: Conference Series. Vol. 256. No. 1. IOP Publishing, 2010.

[2] Needleman, Saul B., and Christian D. Wunsch. ”A general method applicable to the search for similarities in the amino acid sequence of two proteins.” Journal of molecular biology 48.3 (1970): 443-453.

[3] Smith, Temple F., and Michael S. Waterman. ”Comparison of biosequences.” Advances in applied mathematics 2.4 (1981): 482-489.

[4] Hirschberg, Daniel S. ”A linear space algorithm for computing maximal common subsequences.” Communications of the ACM 18.6 (1975): 341-343.

[5] Hunt, James W., and Thomas G. Szymanski. ”A fast algorithm for computing longest common subsequences.” Communications of the ACM 20.5 (1977): 350-353.

[6] Chen, Yixin, Andrew Wan, and Wei Liu. ”A fast parallel algorithm for finding the longest common sequence of multiple biosequences.” BMC bioinformatics 7.4 (2006): 1.

[7] Eswaran S and RajaGopalan SP, ”An Efficient Fast Pruned Parallel Algorithm for finding LCS in Biosequences”, Anale Seria Informatica. Vol. VIII fasc.1, 2010.

[8] Dhraief, Amine, Raik Issaoui, and Abdelfettah Belghith. ”Parallel computing the Longest Common Subsequence (LCS) on GPUs: efficiency and language suitability.” The 1st International Conference on Advanced Communications and Computation (INFOCOMP). 2011.

[9] Li, Yanni, Yuping Wang, and Liang Bao. ”FACC: a novel finite automaton based on cloud computing for the multiple longest common subsequences search.” Mathematical Problems in Engineering 2012 (2012).

[10] Bohara Jnaneshwar, Joshi Shashidhar Ram, ”A MapReduce Based Parallel Algorithm for Finding Longest Common Subsequence in Biosequences”, IOE Graduate Conference Journal (2013)

[11] Dean, Jeffrey, and Sanjay Ghemawat. ”MapReduce: simplified data processing on large clusters.” Communications of the ACM 51.1 (2008): 107-113.

[12] White, Tom. Hadoop: The definitive guide. ” O’Reilly Media, Inc.”, 2012.