TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

CENTRAL CAMPUS, PULCHOWK

THESIS NO: 069MSCS670

A MapReduce Based Parallel Algorithm for Finding Longest Common Subsequence in Biosequences

By

Jnaneshwar Bohara

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

LALITPUR, NEPAL

NOVEMBER, 2014

A MapReduce Based Parallel Algorithm for Finding Longest Common Subsequence in Biosequences

By

Jnaneshwar Bohara

069MSCS670

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, Central Campus

Tribhuvan University

Pulchowk, Lalitpur, Nepal

November, 2014

COPYRIGHT

TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

CENTRAL 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 MapReduce Based Parallel Algorithm for Finding Longest Common Subsequence in Biosequences" submitted by Mr Jnaneshwar Bohara in partial fulfillment of the requirement for the degree of Master of Science in Computer System and Knowledge Engineering.

Spuervisor, Dr. Shashidhar Ram Joshi
Professor
Department of Electronics and Computer Engineering

External Examiner, Mr. Saroj Shakya
Associate Professor
Nepal College of Information Technology

Committee Chairperson, Dr. Shashidhar Ram Joshi
Professor
Department of Electronics and Computer Engineering

Date

Departmental Acceptance

The thesis entitled "A MapReduce Based Parallel Algorithm for Finding Longest Common Subsequence in Biosequences", submitted by Jnaneshwar Bohara 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,
Central Campus, Pulchowk
Institute of Engineering,
Tribhuvan University,
Nepal.

ABSTRACT

The Longest Common Subsequence(LCS) identification of biological sequences has significant aplications in bioinformatics. Due to the emerging growth in bioinformatics applications, new biological sequences with longer length have been used for processing, making it great challenge for sequenctial LCS algorithms. Few parallel LCS algorithms have been proposed but their efficiency and effectiveness are not satisfactory with increasing complexity and size of biological data. To overcome limitations of existing LCS algorithms and considering MapReduce programming model as promising technology for cost effective high performace parallel computing, MapReduce based parallel algorithm for LCS has been developed. This algorithm adopts the concepts of successor tables, identical character pairs, successor tree and traversal of successor tree to find Longest Common Subsequence. The hadoop framework is used for the realization of MapReduce model.

Keywords

Bioinformatics, Longest Common Subsequence, MapReduce, Hadoop

ACKNOWLEDGMENT

I would like to express my deep gratitude to Department of Electronics and Computer Engineering for implementing thesis work as a part of our syllabus through which we can start our first step towards research field.

I am grateful to my supervisor Prof. Dr. Shashidhar Ram Joshi for his encouragement and valuable guidance during the thesis work.

I am also thankful to my respected teachers Prof. Dr. Subarna Shakya, Dr. Arun Timilsina, Dr. Aman Shakya, and Dr. Sanjeeb Prasad Panday for their suggestions and comments regarding this thesis.

Last but not least I am deeply obliged to Assoc. Prof. Dr. Dhundy Raj Bastola from School of Interdisciplinary Informatics, University of Nebraska, Omaha for inspiring me to do research on bioinformatics and providing the necessary datasets.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

DNA	Deoxyribonucleic Acid
EFP	Efficient Fast Pruned
FACC	Finite Automaton based on Cloud Computing
HDFS	Hadoop Distributed File System
LCS	Longest Common Subsequence

CHAPTER 1: INTRODUCTION

1.1 Background

Biological sequence[1] can be represented as a sequence of symbols. DNA sequences can be represented as sequences of four letters A, C, G and T corresponding to the four sub-molecules Adenine, Cytosine, Guanine, and Thymine. When a new biological sequence is found, we want to know what other sequence 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.
Among the many sequence comparison algorithms, one 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. Even slight discrepancies resulting from experimental or technical limitations, can significantly affect the comparison results.

Alternative approaches of sequence analysis are becoming increasingly important in dealing with the exponential growth of genetic sequence data, and the 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. One way of detecting the similarity of two or more sequences using the alignment-free methods is to find their Longest Common Subsequences.

A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. This makes the subsequence different from the substring as a substring should appear in a contiguous relative order. A common subsequence is a

longest subsequence if it is of maximum length. For example, between ATCG and CTCAG, the longest common substring is TC, while the longest common subsequence is TCG.

Advancements in sequencing technology are changing the scale of genetic data. The Genbank, a public repository of genetic sequence data, reported 178322253 sequence records in its 204th release in Oct 2014. Analyzing such large datasets, including the 3 billion bases of the human reference genome, on uniprocessor machines is an extremely time consuming process. Therefore, for efficient computation of LCS, Parallel algorithms are used.

In this thesis work, a parallel algorithm for finding the longest common subsequence (LCS), across genetic sequences is developed. This algorithm uses the open-source implementation of MapReduce[4] called Hadoop[5] to schedule, monitor, and manage the parallel execution. MapReduce is the software framework invented by and used by Google to support parallel execution of their data intensive applications.

Computation in MapReduce is divided into two major phases called map and reduce, separated by an internal grouping of the intermediate results. This two-phase computation was developed after recognizing that many different computations could be solved by first computing a partial result (map phase), and then combining those partial results into the final result (reduce phase). The power of MapReduce is that map and reduce functions are executed in parallel over potentially hundreds or thousands of processors with minimal effort by the application developer.

1. 2. Problem Definition

Searching for the longest common sequence (LCS) between biosequences is one of the most fundamental tasks in bioinformatics. There are many sequential algorithms available in this area and few parallel algorithms are making use of the technology to arrive at a quick solution. The parallel algorithms are implemented using CREW PRAM model, Systolic arrays and MPP parallel computing model. The parallel implementations of these algorithms are of a certain difficulty due to their complicated concurrency, synchronization, and mutual exclusion, that is, none of these algorithms employed simple and cost-effective high performance parallel computing framework such as MapReduce for implementing their algorithms. There is very less work done for LCS using MapReduce.

In this thesis work, a parallel algorithm is developed to speed up the computation for finding LCS using the MapReduce programming model. MapReduce is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster. The main challenge with MapReduce model is to split the problem in the form of map and reduce operations.

1. 3. Objectives

The main objectives of this thesis are

- To develop a MapReduce based parallel algorithm to find Longest Common Subsequences
- To use developed algorithm for biological sequence comparison

1.4 Scope of the work

When a new gene sequence is found, we want 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 larger number of gene sequences increasing day by day and extremely large length of single gene sequence there is a need of parallel computing to find the LCS of gene sequences. MapReduce programming with Hadoop framework is emerging technology for distributed computing especially in case of big data. So, parallel implementation of LCS using MapReduce programming model really the good candidate of research.

The scope of this thesis work is to develop a MapReduce based parallel algorithm that could be used to find the longest common subsequence of gene sequences

1.5 Organization of the thesis report

This thesis report is mainly divided into six sections: Introduction, Literature Review, Theoretical Background, Research Methodology, Results and Discussions and Conclusions.

Introduction section covers Background of thesis work, Problem Definition, Objectives and Scope of the work.

Under Literature Review brief introduction of some existing algorithms are included. Needleman–Wunsch algorithm, Smith–Waterman algorithm and Fast LCS algorithm are included in this section.

Third chapter,which is Theoretical Background includes brief introduction of Deoxyribonucleic acid (DNA), MapReduce and Hadoop and the Hadoop Distributed File System.

Research Methodology section covers Model Development, Algorithm Development and MapReduce Strategy developed for realization of the algorithm developed.

Results and Discussions section includes Sample output, Complexity Analysis and Run Time of the Algorithm.

Finally the thesis report is concluded with Conclusions section which includes conclusions and recommendations.

CHAPTER 2: LITERATURE REVIEW

Since the LCS problem is essentially a special case of the sequence alignment, all the algorithms for the sequence alignment can be used to solve the LCS problem.

The Needleman–Wunsch[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 Smith- Waterman [3] algorithm evolved from Needleman- Wunsch algorithm provides a local alignment of biological sequences.

Various parallel algorithms based on the CREW PRAM model, Systolic arrays have been proposed in the earlier days to reduce the computation time. Later, Wan, Liu, Chen proposed Fast LCS algorithm [6]. Fast LCS's efficiency has been further improved by the Efficient Fast Pruned LCS (EFP_LCS) [7]. Further inspired from Fast LCS algorithm, Bhowmick, Shafiullah, Rai and Bastola have proposed a Parallel Non-Alignment Based Approach to LCS [8]. In the recent days Li ,Wang and Bao have proposed a finite automaton based on cloud computing called FACC[9] for LCS. This is also mostly inspited by Fast LCS algorithm.

2.1 Needleman–Wunsch algorithm

The Needleman–Wunsch algorithm performs a global alignment on 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 Needleman–Wunsch 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 Smith–Waterman algorithm

The Smith–Waterman algorithm performs local sequence alignment; that is, for determining similar regions between two strings or nucleotide or protein sequences. Instead of looking at the total sequence, the Smith–Waterman algorithm compares segments of all possible lengths and optimizes the similarity measure.

The algorithm was first proposed by Temple F. Smith and Michael S. Waterman in 1981. Like the Needleman–Wunsch algorithm, of which it is a variation, Smith–Waterman is a dynamic programming algorithm. As such, it has the desirable property that it is guaranteed to find the optimal local alignment with respect to the scoring system being used (which includes the substitution matrix and the gap-scoring scheme). The main difference to the Needleman–Wunsch algorithm is that negative scoring matrix cells are set to zero, which renders the (thus positively scoring) local alignments visible. Backtracking starts at the highest scoring matrix cell and proceeds until a cell with score zero is encountered, yielding the highest scoring local alignment.

2.3 Fast LCS algorithm

The algorithm first constructs a novel successor table to obtain all the identical pairs and their levels. It then obtains the LCS by tracing back from the identical character pairs at the last level. The key technique of this algorithm is the use of several effective pruning operations. In the process of generating the successors, pruning techniques can remove the identical pairs which cannot generate the LCS so as to reduce the search space and accelerate the search speed.

CHAPTER 3: THEORETICAL BACKGROUND

3.1 Deoxyribonucleic acid (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 nucleotides—each 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.

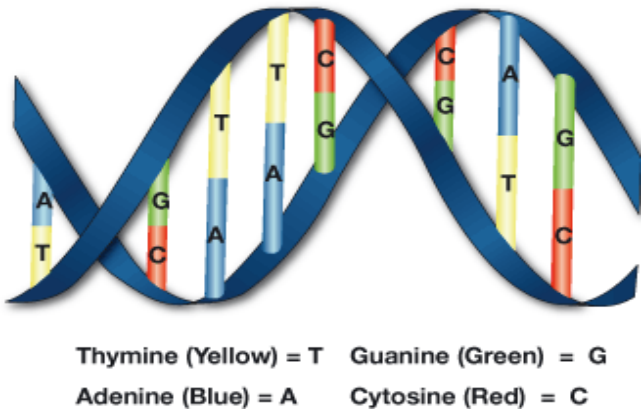


Figure 3.1. Structure of DNA sequence

3.2 MapReduce

MapReduce 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 intermediate 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 Google's 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 are specified by the user.

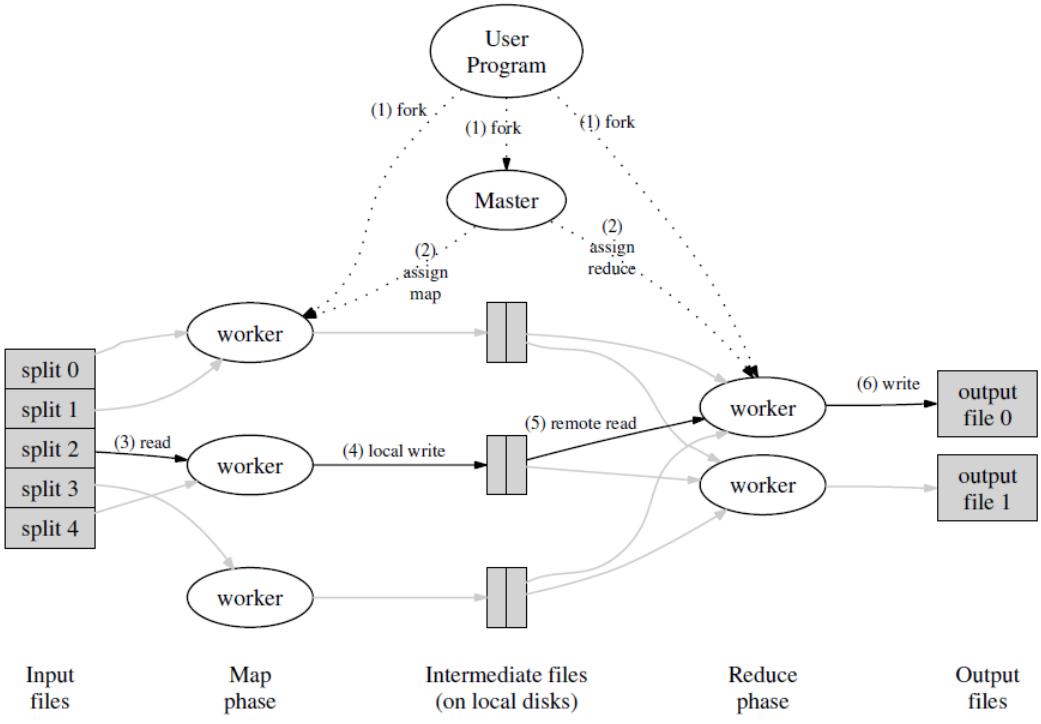


Figure 3.2. MapReduce Execution Overview

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 areM 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 user's 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.

3.2.2 Master Data Structures

The master keeps several data structures. For each map task and reduce task, it stores the state (idle, in-progress, or completed), and the identity of the worker machine (for non-idle tasks).

The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have in-progress reduce tasks.

3.3 Hadoop and the Hadoop Distributed File System

Hadoop 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.

3.3.1 NameNode and DataNodes

HDFS has a master/slave architecture. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on. HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.

The NameNode and DataNode are pieces of software designed to run on commodity machines. These machines typically run a GNU/Linux operating system (OS). HDFS is built using the Java language; any machine that supports Java can run the NameNode or the DataNode software. Usage of the highly portable Java language means that HDFS can be deployed on a wide range of machines. A typical deployment has a dedicated machine that runs only the NameNode software. Each of the other machines in the cluster runs one instance of the DataNode software. The architecture does not preclude running multiple DataNodes on the same machine but in a real deployment that is rarely the case.

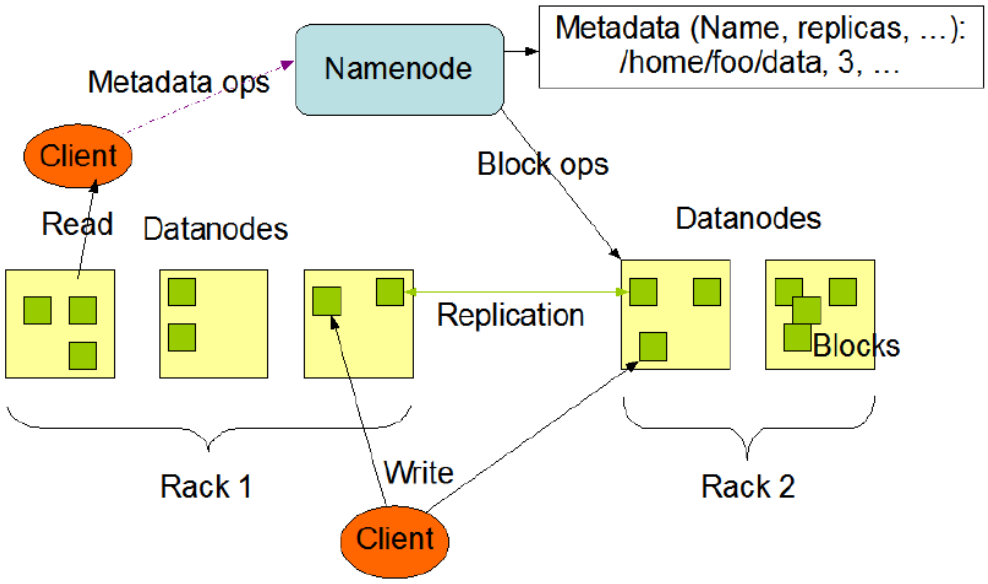


Figure 3.3. HDFS Architecture

The existence of a single NameNode in a cluster greatly simplifies the architecture of the system. The NameNode is the arbitrator and repository for all HDFS metadata. The system is designed in such a way that user data never flows through the NameNode.

3.3.2 The File System Namespace

HDFS supports a traditional hierarchical file organization. A user or an application can create directories and store files inside these directories. The file system namespace hierarchy is similar to most other existing file systems; one can create and remove files, move a file from one directory to another, or rename a file. HDFS does not yet implement user quotas or access permissions. HDFS does not support hard links or soft links. However, the HDFS architecture does not preclude implementing these features.

The NameNode maintains the file system namespace. Any change to the file system namespace or its properties is recorded by the NameNode. An application can specify the number of replicas of a file that should be maintained by HDFS. The number of copies of a file is called the replication factor of that file. This information is stored by the NameNode.

3.3.3 Data Replication

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later. Files in HDFS are write-once and have strictly one writer at any time.

The NameNode makes all decisions regarding replication of blocks. It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster.

Receipt of a Heartbeat implies that the DataNode is functioning properly. A Blockreport contains a list of all blocks on a DataNode.

The placement of replicas is critical to HDFS reliability and performance. Optimizing replica placement distinguishes HDFS from most other distributed file systems. This is a feature that needs lots of tuning and experience. The purpose of a rack-aware replica placement policy is to improve data reliability, availability, and network bandwidth utilization. The current implementation for the replica placement policy is a first effort in this direction. The short-term goals of implementing this policy are to validate it on production systems, learn more about its behavior, and build a foundation to test and research more sophisticated policies.

To minimize global bandwidth consumption and read latency, HDFS tries to satisfy a read request from a replica that is closest to the reader. If there exists a replica on the same rack as the reader node, then that replica is preferred to satisfy the read request. If HDFS cluster spans multiple data centers, then a replica that is resident in the local data center is preferred over any remote replica.

CHAPTER 4: RESEARCH METHODOLOGY

4.1 Model Development

Below are the steps followed in Model development. These include construction of successor table, finding initial identical character pairs, producing successors and creating successor tree, and finally traversing successor tree to find LCS.

4.1.1 Constructing successor tables

Suppose X = $(x_1, x_2, \ldots, x_n)$ and Y = $(y_1, y_2, \ldots, y_n)$ are two biosequences where $x_i, y_i \in$ {A, C, G, T}. We can define an array CH of the four characters such that CH(0) = 'A', CH(1)='C', CH(2) = 'G' and CH(3) = 'T'. The successor tables of the identical characters of X and Y are represented by TX and TY. Entries in successor tables are defined as:

$$T(i,j) = \begin{cases} \min\{k | k \in S(i,j)\}, & S(i,j) \neq \phi \\ -, & otherwise \end{cases} \quad (1)$$

$$S(i,j) = \{k | x_k = CH(i), k > j\}$$

Where i= 0,1,2,3 and j = 0,1,2,…,n i.e. length of sequence.

If T(i,j) is not "-", it gives the position of the next character identical to CH(i) after the j^{th} position in the sequence, otherwise it means there is no such character after the j^{th} position.

Example 1:

Let X = "ATCG" and Y = "CTCAG". Their successor tables TX and TY are shown in Table 4.1.

Table 4.1. Successor Tables TX and TY

TX:

i	CH(i)	j				
		0	1	2	3	4
0	A	1	-	-	-	-
1	C	3	3	3	-	-
2	G	4	4	4	4	-
3	T	2	2	-	-	-

TY:

i	CH(i)	j					
		0	1	2	3	4	5
0	A	4	4	4	4	-	-
1	C	1	3	3	-	-	-
2	G	5	5	5	5	5	
3	T	2	2	-	-	-	

4.1.2 Finding initial identical character pairs

For the sequences X and Y, if $x_i = y_j$ = CH(k), then (i, j) is called an identical pair of CH(k). Initial identical character pairs of X and Y computed from TX and TY are represented as (TX(k,0), TY(k,0)), where k = 0,1,2,3. In example 1, these are A(1,4), C(3,1), G(4,5) and T(2,2).

4.1.3 Producing successors and creating successor tree

At first all the direct successors of initial identical character pairs are produced in parallel using successor tables. Then the direct successors of those successors computed in the previous step are produced in parallel. This process of producing direct successors is repeated until there are no successor to be produced.

For an identical character pair (i, j), the direct successors can be produced as:

$$(i,j) \rightarrow \{(TX(k,i),TY(k,j))|k = 0,1,2,3\ TX(k,i)\neq\text{'-'}\ \text{and}\ TY(k,j)\neq\text{'-'}\} \quad (2)$$

For example, the successors of the identical character pair A (1,4) in Example 1 can be obtained by coupling the elements of the 1st column of TX and the 4th column of TY. It is G(4, 5).

Operation of Pruning

While producing director successor and creating successor tree, pruning techniques can be used to remove the identical pairs which do not contribute to the LCS so as to reduce the search space and improve the efficiency.

Pruning Operation 1

If there are two identical character pairs (i, j) and (k, l) satisfying (k, l)>(i, j) on the same level, then (k, l) can be pruned without affecting the correctness of the algorithm in obtaining the LCS of X and Y.

For example, C(3, 3) and G(4, 5) in Example 1 are the successors of the identical pair T(2, 2). Since they are on the same level and G(4, 5) > C(3, 3), we can prune G(4, 5).

Pruning Operation 2

If on the same level, there are two identical character pairs (i_1, j) and (i_2, j) satisfying $i_1 < i_2$, then (i_2, j) can be pruned without affecting the correctness of the algorithm in obtaining the LCS of X and Y.

Pruning Operation 3

If there are identical character pairs (i_1, j) , (i_2, j), ..., (i_r, j) and $i_1 < i_2 < ... < i_r$, then we can prune (i_2, j), ..., (i_r, j).

Finally a successor tree is created from the successors produced.We consider a dummy node ε(0,0) as a root.

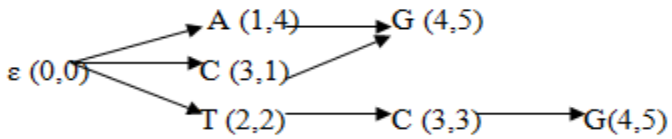


Figure 4.1. Successor tree to compute LCS of sequences X and Y

4.1.4 Traversing successor tree to find LCS

The successor tree is traversed using depth first search method. Longest path gives the LCS.

For example, in Example 1, longest path traversed is

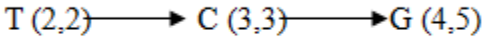


Figure 4.2. Longest Common Subsequences of sequences X and Y

Therefore, Longest Common Subsequence is TCG.

4.2 Algorithm Development

Input:

Two bio-sequences X and Y of length m and n respectively.

Output:

String(s) LCS of length |LCS| which is (are) the longest common subsequence(s) of X and Y.

Procedure

1. Create successor table TX and TY for X and Y sequences over the alphabets of CH in parallel using MapReduce model

2. Initial identical character pairs

    2.1 Find all the initial identical character pairs:(TX(k, 0),TY(k, 0)), where k = 0,1,2,3

    2.2 Apply pruning techniques on the initial identical character pairs obtained

3. Finding the direct successors level wise and producing the successor tree

Repeat the steps 3.1 through 3.2 until no more successors are found

    3.1 For all the current level identical pair, do in parallel using MapReduce model

        3.1.1 Identify all direct successors

        3.1.2 Apply pruning techniques

    3.2 Add the potentially useful successor to the successor tree

4. Traversing successor tree to search for LCS

    4.1 By depth first search method, successor tree is traversed from root to every leaf node

    4.2 Longest path traversed will be the LCS

4.3 MapReduce strategy

First of all two input files containing DNA sequences are loaded to HDFS. Input files are text files which contain combinations of characters A, C, G and T.

4.3.1 Finding the positions of characters

This single Map Reduce task is to find positions of all characters in both input sequences. Input for this MapReduce job are input files loaded into HDFS under input directory.

For this, the Map and Reduce tasks are defined as follows:

Map Procedure:

Input(k,v) = (line_number, dna_sequence)

Output(k,v) = (character, position_of_character)

For each line in value

- Extract each character
- Set extracted character(A, C, G and T) as output key
- Set position of extracted character as output value

Reduce Procedure:

Input(k,v) = (character, position_of_character)

Output(k,v) = (character, position_list)

For each position values of characters in a sequence

- Create a list of positions of each characters in a sequence file
- Set thus created list as output value of reducer
- Character value plus sequence file name will be the output key of reducer

Output of this MapReduce job is written to HDFS under “positons” sub directory of output directory.

4.3.2 Creating successor tables

This single Map Reduce task is to create successor tables TX and TY in parallel. This MapReduce job takes as input from “positons” sub directory of output directory.
For this, the Map and Reduce tasks are defined as follows:

Map Procedure:

Input(k,v) = (line_number, position_list)

Output(k,v) = (one, position_list)

For each line in value, pass the content in line (position list) to the reducer as it is.

Reduce Procedure:

Input(k,v) = (one, position_list)

Output(k,v) = (character, entry_in_successor_table)

Here,

HBase table is used to store the successor tables created. Since successor table is used as lookup table for computing direct successors at each level, it is effective solution to store this in HBase and retrieve whenever needed.

The HBase table created to store values of successor tables is given the name “successor_table”. Two column families “seq1” and “seq2” are defined to store values corresponding to TX and TY. There are four row keys defined with names 0,1,2 and 3 corressponding to four characters A,C,G and T.

4.3.3 Finding initial identical pairs

This single Map Reduce task is to compute initial identical pairs from successor tables TX and TY in parallel.

Here,

Input to the Mapper is the data from HBase table “successor_table”.

For this, the Map and Reduce tasks are defined as follows:

Map Procedure:

Input(k,v) = (rowKey, columns)

Output(k,v) = (identical_character, position_pair)

Combine values in columns “seq1:0” and “seq2:0” to compute the identical character pairs.

Reduce Procedure:

Input(k,v) = (identical_character, position_pair)

Output(k,v) = (identical_character, pruned_ position_pair)

Perform prunings on the position pairs and only the remaining pairs after pruning will be treated as initial identical character pairs and written to HDFS. Output of this MaprReduce job is given name “level0” in output directory.

4.3.4 Finding LCS levelwise

This is Multi Level Map Reduce task. Input to the First level Map Reduce job to compute LCS are the initial identical pairs stored under the location “levelo”. And output will be stored under “level1”.

For next level MapReduce job, input is “level1” and output is “level2” and so on until there are no direct successors based on successor tables.

For this the Map and Reduce tasks are defined as follows:

Map Procedure:

Input(k,v) = (identical_character, position_pair)

Output(k,v) = (identical_character, direct_successors)

For each identical character pair at current level, compute all possible direct successors.

Reduce Procedure:

Input(k,v) = (identical_character, direct_successors)

Output(k,v) = (identical_character, pruned_direct_successors)

For each identical character pairs at a level

- Compute all possible direct successors from successor tables
- Apply pruning techniques
- Identical character pair will remain as key of reducer output
- Direct successors after pruning will be value of reducer output

Here,

Besides input from previous level LCS, successor tables are read from HBase in every mapper as the look up table to compute the direct successors.

The identical character pairs at each level need to remember the value of the identical character pairs at previous levels. Therefore, identical characters pairs are created by concatenating the LCS at previous levels.

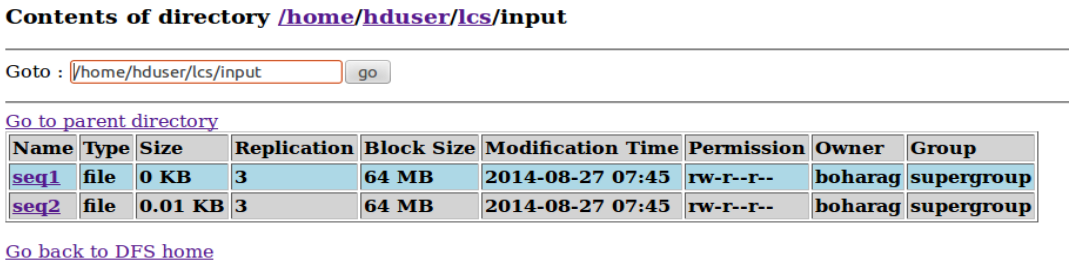
CHAPTER 5: RESULTS AND DISCUSSIONS

The basic Map Reduce Model for computing the Longest Common Subsequence has been developed, which includes as listed below.

5.1 Sample Output

Below are the sample outputs at different stages.

5.1.1 Input directory in HDFS



Contents of directory /home/hduser/lcs/input

Goto : /home/hduser/lcs/input go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
seq1	file	0 KB	3	64 MB	2014-08-27 07:45	rw-r--r--	boharag	supergroup
seq2	file	0.01 KB	3	64 MB	2014-08-27 07:45	rw-r--r--	boharag	supergroup

Go back to DFS home

Figure 5.1. Input directory in HDFS

5.1.2 Input sequence one in HDFS

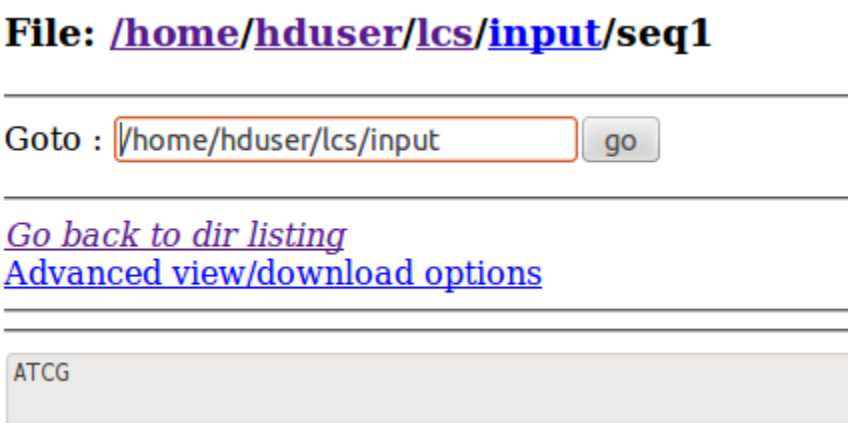


Figure 5.2. Input sequence one in HDFS

5.1.3 Input sequence two in HDFS

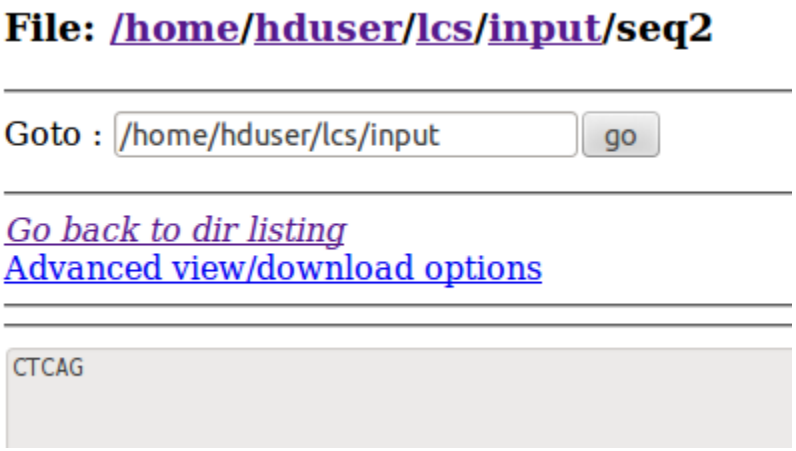
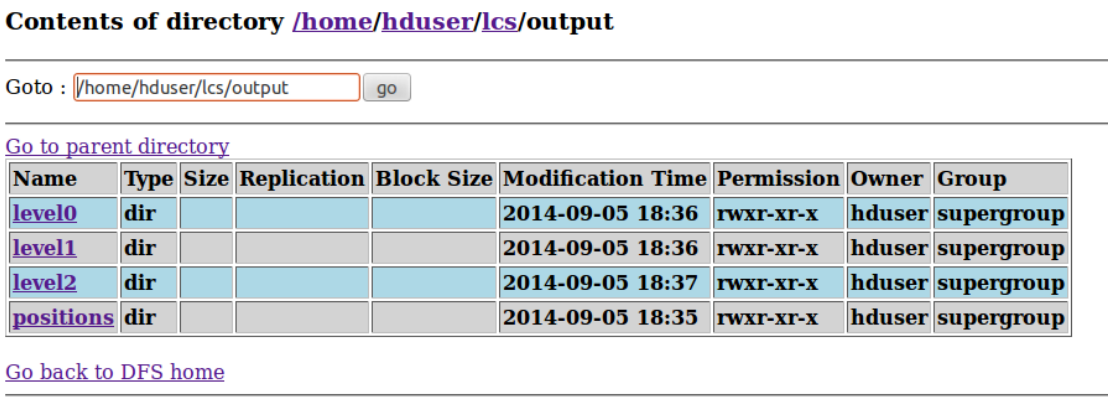


Figure 5.3. Input sequence two in HDFS

5.1.4 Output directory in HDFS



Contents of directory /home/hduser/lcs/output

Goto : /home/hduser/lcs/output go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
level0	dir				2014-09-05 18:36	rwxr-xr-x	hduser	supergroup
level1	dir				2014-09-05 18:36	rwxr-xr-x	hduser	supergroup
level2	dir				2014-09-05 18:37	rwxr-xr-x	hduser	supergroup
positions	dir				2014-09-05 18:35	rwxr-xr-x	hduser	supergroup

Go back to DFS home

Figure 5.4. Output directory in HDFS

5.1.5 Output of position finding

File: /home/hduser/lcs/output/positions/part-r-00000

Goto : /home/hduser/lcs/output/positio go

Go back to dir listing
Advanced view/download options

```
seq1::A [1]
seq2::A [4]
seq1::C [3]
seq2::C [1, 3]
seq1::G [4]
seq2::G [5]
seq1::T [2]
seq2::T [2]
```

Figure 5.5. Output of position finding

5.1.6 Output of successor table creation

```
hbase(main):013:0> scan 'successor_table'
ROW                   COLUMN+CELL
 A                    column=seq1:\x00\x00\x00\x00, timestamp=1409924130040, val
                      ue=1
 A                    column=seq1:\x00\x00\x00\x01, timestamp=1409924130040, val
                      ue=-
 A                    column=seq1:\x00\x00\x00\x02, timestamp=1409924130040, val
                      ue=-
 A                    column=seq1:\x00\x00\x00\x03, timestamp=1409924130040, val
                      ue=-
 A                    column=seq1:\x00\x00\x00\x04, timestamp=1409924130040, val
                      ue=-
 A                    column=seq2:\x00\x00\x00\x00, timestamp=1409924130058, val
                      ue=4
 A                    column=seq2:\x00\x00\x00\x01, timestamp=1409924130058, val
                      ue=4
 A                    column=seq2:\x00\x00\x00\x02, timestamp=1409924130058, val
                      ue=4
 A                    column=seq2:\x00\x00\x00\x03, timestamp=1409924130058, val
                      ue=4
 A                    column=seq2:\x00\x00\x00\x04, timestamp=1409924130058, val
                      ue=-
 A                    column=seq2:\x00\x00\x00\x05, timestamp=1409924130058, val
                      ue=-
 C                    column=seq1:\x00\x00\x00\x00, timestamp=1409924130060, val
                      ue=3
 C                    column=seq1:\x00\x00\x00\x01, timestamp=1409924130060, val
                      ue=3
 C                    column=seq1:\x00\x00\x00\x02, timestamp=1409924130060, val
                      ue=3
 C                    column=seq1:\x00\x00\x00\x03, timestamp=1409924130060, val
                      ue=-
 C                    column=seq1:\x00\x00\x00\x04, timestamp=1409924130060, val
                      ue=-
 C                    column=seq2:\x00\x00\x00\x00, timestamp=1409924130065, val
                      ue=1
 C                    column=seq2:\x00\x00\x00\x01, timestamp=1409924130065, val
```

Figure 5.6. Output of successor table creation

5.1.7 Output of initial identical character pair finding(level 0 LCS)

Figure 5.7. Output of initial identical character pair finding(level 0 LCS)

5.1.8 Output of level 1 LCS

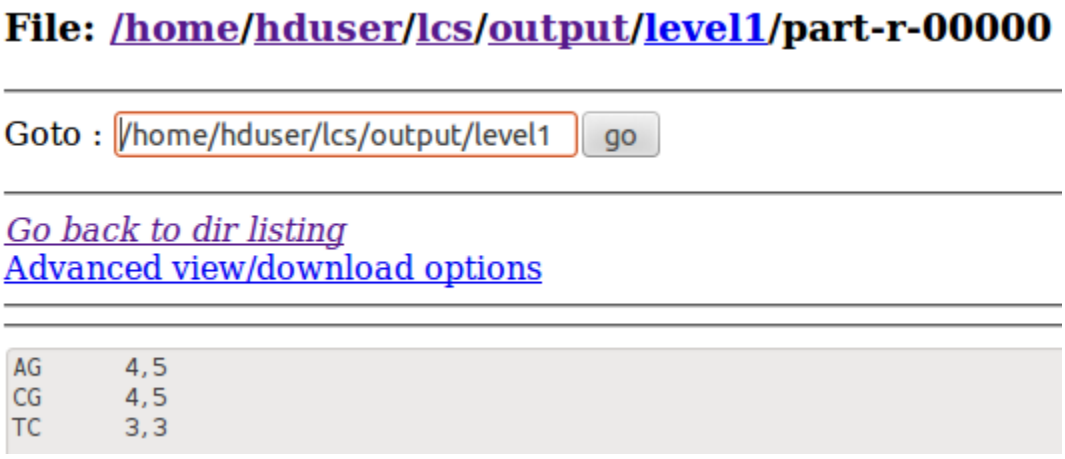


Figure 5.8. Output of level 1 LCS

5.1.9 Output of level 2 LCS

Figure 5.9. Output of level 2 LCS

5.2 Complexity Analysis

Each input sequence is traversed once to find the position of identical characters. Based on positions found, successor tables are constructed and finally LCS is computed. Therefore, assuming both sequences with length n, time complexity is O(n). Again storage space is proportional to the size of input sequences. Therefore, the space complexity is also O(n).

5.3 Run Time of Algorithm

Performance is measured by running this algorithm for two input sequences. Below are the results for the time taken with single node and multi node cluster with 5 working nodes. The configuration for each node is Intel Core 2 Duo CPU E7500 2.93GHz with 2 cores and 2GB of RAM. The hadoop version used is 1.0.4 and the linux version for hadoop cluster setup is ubuntu 12.04. Two input sequences are run for 10 times in both single node and multi node cluster and average time is computed. Table 5.1 shows the time taken by this algorithm while running in single node. Table 5.2 shows the time taken while running in the multi node cluster.

Table 5.1. Time taken by each MapReduce job for single node

No of iterations	Time taken by each MapReduce job(in ms)			
	Position Pair	Successor Table	Initial Identical Pair	LCS
1	34167	33491	33594	32640
2	33089	32491	33535	32545
3	33302	33582	33553	32471
4	33034	33594	32922	33467
5	35196	32511	33580	32501
6	34015	33457	32667	33521
7	34043	32546	33548	32861
8	34027	33553	32718	33480
9	32088	33557	32491	33446
10	32111	33569	32480	33447
Average	**33507.2**	**33235.1**	**33108.8**	**33037.9**

Table 5.2. Time taken by each MapReduce job for multi node cluster

No of iterations	Time taken by each MapReduce job(in ms)			
	Position Pair	Successor Table	Initial Identical Pair	LCS
1	15353	14484	19429	28885
2	15296	14514	19437	28900
3	20218	14517	14433	28885
4	15238	14512	14440	29858
5	15320	14480	14469	33844
6	15274	14533	14448	28838
7	20283	14527	14411	33866
8	15156	14517	14462	33839
9	15233	14499	14429	28866
10	15252	14496	14449	28864
Average	**16262.3**	**14507.9**	**15440.7**	**30464.5**

As shown in Table 5.1 and Table 5.2, time taken by the algorithm is decreased in case of multi node cluster compared with that in the single node. Thus, it shows the performance of this algortihm is scalabe with number of nodes in the cluster.This algorithm can be used to compute LCS of very long bio sequences in short time if we increase the processing power by adding more nodes in the cluster.

CHAPTER 6: CONCLUSIONS

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. Thus the algorithm developed in this thesis work uses the cutting edge technology to address the problem with the LCS computation. This MapReduce based model handles all the different aspects of distributed computing from load balancing to synchronization automatically. Simulation results show that the algorithm is scalable with respect to the number of nodes in the cluster. Hence, the large number of gene data can be processed at short period if we use the large number of nodes created from commodity computers.

The algorithm developed in this thesis work uses the HBase to store the successor tables. The HBase being one of member in NoSQL family, we can use all benefits NoSQL provides. Thus, we can have any number of columns in successor table which can grow dynamically. This makes possible to store successor tables of bio sequences which have large number of base pairs.

Yet there is much room for optimization and improvements for better accuracy and efficiency. Efficiency is to be achieved by looking for alternative approaches to address the same problem. Currently to compute LCS level wise, different MapReduce jobs are created at each level. The efficiency could be improved if we could compute the LCS using the single MapReduce job.

REFERENCES

[1] Hao B, Zhang SY, The manual of bioinformatics, Shanghai Science and Technology Publishing Company 2000.

[2] Needleman SB and Wunsch CD, "A general method applicable to the search for similarities in the amino acid sequence of two proteins" J Mol Biol **48** 443-53, 1970

[3] Smith TF and Waterman MS, "Comparison of biosequences", Adv. Appl. Math. **2** 482– 89, 1981

[4] Dean J and Ghemawat S, "MapReduce: Simplified Data Processing on Large Clusters", OSDI '04: 6th Symposium on Operating Systems Design and Implementation, 2004

[5] White T, "Hadoop: The Definitive Guide" O'Reilly|Yahoo Press, 2009

[6] Chen Y, Wan A and Liu W, "A fast Parallel Algorithm for finding the Longest Common Subsequence of multiple biosequences", BMC Bioinformatics **7** (suppl 4), 2006

[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] Bhowmick S, Shafiullah1 M, Rai H and Bastola D, "A Parallel Non-Alignment Based Approach to Efficient Sequence Comparison using Longest

Common Subsequences", High Performance Computing Symposium (HPCS2010) , Journal of Physics: Conference Series **256** (2010) 012012, 2010

[9] Li Y, Wang Y and Bao L, "FACC: A Novel Finite Automaton Based on Cloud Computing for the Multiple Longest Common Subsequences Search", Hindawi Publishing Corporation, Mathematical Problems in Engineering, Volume 2012