2013

Parallel Suffix Tree Construction for Genome Sequence Using Hadoop

Umesh Chandra Satish
Louisiana State University and Agricultural and Mechanical College, usatis1@tigers.lsu.edu

PARALLEL SUFFIX TREE CONSTRUCTION FOR GENOME SEQUENCE USING HADOOP

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in System Science

in

School of Electrical Engineering and Computer Science

by
Umesh Chandra Satish
B.E. Computer Science and Engineering,
Visvesvaraya Technological University, 2008
December 2013

ACKNOWLEDGEMENTS

I would like to first express my gratitude to the almighty for giving me the strength to live my dreams.

Next, I would like to thank my advisor Dr. Seung-Jong Park for having guided me through my Masters course. His continuous guidance and encouragement helped me immensely in my research. I would like to thank Praveen Kumar Kondikoppa and Manish Patil for their technical advice and suggestions which helped me to overcome hurdles and kept me enthusiastic and made this work a wonderful learning experience. I would also like to thank Dr. Rahul Shah for his timely help during the course of this thesis. I also thank Dr. Bijaya Kharki for being a part of my thesis committee and supporting me through it.

I would like to thank Mr. John Rials, Mr. Alex cook, Mr. Nash Hassan, Mr. Ray Leipine and the department of continuing education at Louisiana State University for supporting and funding for my Masters program.

I would now like to thank my Parents, grandmother, and my brother for constantly being a source of encouragement at all times during my stay at school. This thesis would not have been possible without their support.

I would like to thank Priyanka Rotti for being my critic and proof reading my thesis. Katrina Weis for proof reading my thesis. Abhishek Pandey for his support in writing my thesis document. Ajay Panyal for his timely technical suggestions. A special node of thanks to Sashank, Pratap, Kaushik Parasuraman, Arpitha, Sunada, Madhavi, Abhijit Rao, Karthik, Shabana and Narendra for being a constant source of support throughout my thesis and Masters program.

I would also like to express my sincere gratitude to Sandesh, Srikanth, Surabhi, Sheetal, Priyanka, and Shyam for always being my source of inspiration through this journey.

I am sure I have missed mentioning a lot of other people who made this possible for me. I express my sincere gratitude to everyone who have been with me though this journey.

TABLE OF CONTENTS

ABSTRACT

Indexing the genome is the basis for many of the bioinformatics applications. Read mapping (sequence alignment) is one such application to align millions of short reads against reference genome. Several tools like BLAST [11], SOAP [7], BOWTIE [10], Cloudburst [14], and Rapid Parallel Genome Indexing with MapReduce [13] use indexing technique for aligning short reads.

Many of the contemporary alignment techniques are time consuming, memory intensive and cannot be easily scaled to larger genomes. Suffix tree is a popular data structure which can be used to overcome the demerits of other alignment techniques. However, constructing the suffix tree is highly memory intensive and time consuming.

In this thesis, a MapReduce based parallel construction of the suffix tree is proposed. The performance of the algorithm is measured on the hadoop framework over commodity cluster with each node having 8GB of primary memory. The results show a significantly less time for constructing suffix tree for a big data like human genome.

CHAPTER 1
INTRODUCTION

The amount of data stored in this world is increasing rapidly every day. It has been roughly estimated that the data available is close to the order of zeta bytes [15]. Big data usually includes data sets with sizes beyond the ability of commonly used software tools to capture, curate, manage, and process the data within a tolerable elapsed time [1].

There are a number of large data sets like:

- Number of patients records in millions of hospitals across the world.
- Data from different stock exchanges across the world.
- Different types of large data like pictures, texts, videos, etc in various search engines and social networking sites.
- Data from different grocery stores like Walmart, Albertsons, etc.
- Employee database in all the companies across the world.
- Weather data for ships.

These large data sets are called big data. Big data spans four dimensions: volume, velocity, variety, and veracity. [2]

Volume: The volume of data is used for processing information.

- Approximately 12 terabytes of data created per day on Twitter is used for product sentiment analysis . [2]
- Approximately 350 million annual meter readings are considered to calculate power consumption. [2]

Velocity: Many tasks, which need to be completed in a short time, involve big data.

- Identify potential frauds among 5 million trade events. [2]

- Identify the customers who use the telephone the most number of times in a day amidst the daily 500 million calls. [2]

Variety: Big data can be structured or unstructured like text, images, paintings, audio, video, binary, flowcharts, trees, etc. These different types of big data are used daily to analyze various things [2].

- Monitor hundreds of live video feeds from surveillance cameras [2].
- Manage different kinds of data like texts, images, audio files, video files, etc., being uploaded on social networking sites.

Veracity: As the variety and number of sources grows, establishing the trust in big data is challenging [2]

Although the big data has made its presence felt in almost all sectors, the methods to store it effectively and analyze it are rare. For example, read mapping. In the last decade the efficiency and throughput of DNA sequencing is greatly increased. A typical sequencer produces billions of short reads per execution. One of the important bioinformatics application which is based on short reads is read mapping, where the reads are mapped against a reference genome to find similar or dissimilar regions. Read mapping can also be used to align the short reads to form the genome. In the last several years many alignment tools have been proposed which invariably rely on a precomputed index structure which allows rapid matching and searching. As a result of enormous sequence data produced and growing number of organisms considered for sequencing, an effective data structure is essential for indexing these sequence data.

1.1 DNA sequencing and the need for it

In DNA sequencing, genes are identified based on their DNA sequences. The structure of DNA is shown in figure 1.1 [8]. DNA sequencing is important for comparing the sequences of

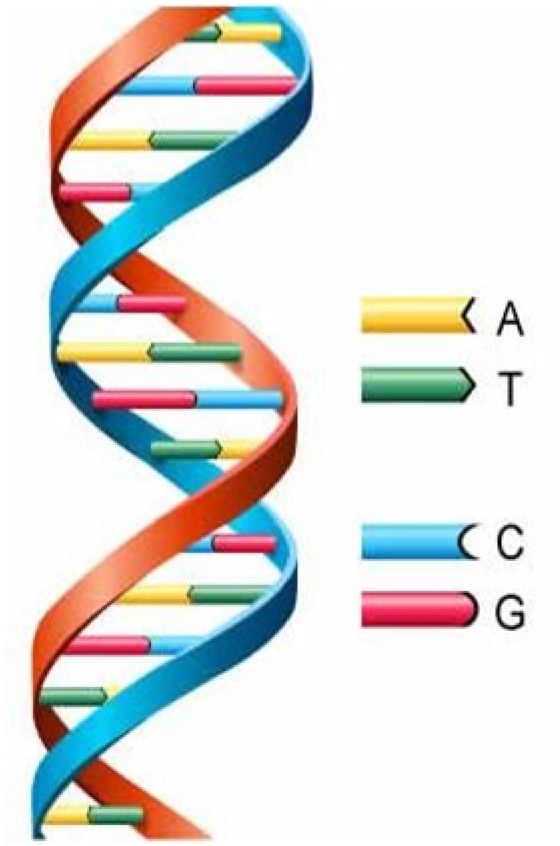


Figure 1.1: Structure of a DNA

samples to determine relationships. For example, the DNA of a chimpanzee can be compared to that of a human to determine the relationship between them. DNA sequencing has other important applications like diagnosing various kinds of diseases, crime forensics, etc.

The advent of DNA sequencing significantly accelerated biological research and discovery. The rapid speed of sequencing attained with modern DNA sequencing technology has been instrumental in the sequencing of the human genome, in the human genome project. Related projects, often by scientific collaboration across continents, have generated the complete DNA sequences of many animals, plants, and microbial genomes [3].

The process of DNA sequencing translates the DNA of a specific organism into a format that is decipherable by researchers and scientists. DNA sequencing has given a massive boost to numerous fields such as forensic biology, biotechnology and more. By mapping the basic sequence of nucleotides, DNA sequencing has allowed scientists to better understand genes and their role in the creation of the human body [3].

Forensic biology uses DNA sequences to identify the organism to which it is unique. Identifying a specific individual is not currently precise. However, as the process evolves, direct comparisons of large DNA segments, and maybe even genomes, will be more practical and viable and allow accurate identification of an individual. Furthermore, scientists will be able to isolate the genes responsible for genetic diseases like Cystic Fibrosis, Alzheimers disease, myotonic dystrophy, etc., which are caused by the inability of genes to work properly.

Agriculture has been helped immensely by DNA sequencing. It has allowed scientists to make plants more resistant to insects and pests, by understanding their genes. Likewise, the same technique has been utilized to increase the productivity and quality of the milk, as well as the meat, produced by livestock.

Recent improvements in sequencing technology (”next-gen” sequencing platforms) have sharply reduced the cost of sequencing. The 1000 Genomes Project is the first project to sequence the genomes of a large number of people, to provide a comprehensive resource on human genetic variation [4]

The goal of the 1000 Genomes Project is to find the most genetic variants that have frequencies of at least 1% in the populations studied. This goal can be attained by sequencing many individuals lightly. To sequence a person’s genome, many copies of the DNA are broken into short pieces and each piece is sequenced. The multiple copies of DNA mean that the DNA pieces are more or less randomly distributed across the genome. The pieces are then aligned to the reference sequence and joined together.

1.2 Contribution of the thesis

In this thesis, a MapReduce based parallel construction of suffix tree for human genome. In our approach we divide the task of constructing the suffix tree into multiple subtrees. These subtrees are constructed in parallel across multiple computing nodes and hence we optimize the time for constructing the entire suffix tree. We also make sure that the size of the subtree will not exceed the available primary memory. Executing our algorithm on Hadoop framework, an open source implementation of Map-Reduce, using as many as 120 cores, we have reduced the time for constructing the suffix tree.

1.3 Thesis Organization

This thesis is organized as follows: Chapter 2 discusses previous work that led to this thesis. Chapter 3 discusses the suffix tree and the Hadoop Framework. It also explains the Map-Reduce programming model. Chapter 4 discusses the algorithm which has been used as a basis for our work and details the implementation specifics of our parallel algorithm which is used to construct the suffix tree. Chapter 5 discusses the experimental results and analysis of those results. Various experimental scenarios are considered and the behavior of the algorithm for those different scenarios are observed and analyzed. The experimental results are also compared with that of [Rapid Parallel Genome Indexing with MapReduce]. Chapter 6 gives a conclusion for this thesis and discusses a few points of the thesis that can be used as a base for future work.

CHAPTER 2
RELATED WORK

For more than two decades, read mapping has created a lot of interest in the field of computational biology. A lot of work has been done over the years to improve the read mapping on large datasets.

2.1 Sequential Algorithms for read mapping

2.1.1 BLAST (Basic Local Alignment Search Tool)

BLAST is one of the oldest techniques proposed in 1990 [11]. It makes use of an algorithmic technique called seed-and-extend. BLAST first indexes the short sequence to be searched in the reference genome sequence. These short sequences are called query sequences and the resulting strings, after indexing, are called words. The length of a word is assumed to be 11 to have a rapid comparison between the words and the sequences. BLAST then scans the reference genome sequence to find the similar match of the indexed words. BLAST uses a scoring technique where a positive score is assigned for a character pair match and a negative score is assigned for character pair mismatch. Only the alignments, which are within a set threshold, are said to be hits. After finding the match, BLAST extends the match in both right and left directions to find a longer match and the extension does not stop until the accumulated score begins to decrease.

The time required to execute BLAST is the sum of the times required for the following:

- To compile a list of short sequences that can score atleast the threshold value T when compared with the reference sequence.
- To scan for the number of hits and to extend all the hits to get the matches which are above the threshold. The time for this task is proportional to the number of hits which depends on the threshold T.

For experimental purposes, the human genome was compared to the rabbit genome. The

Table 2.1: The time and sensitivity of BLAST on DNA sequences as a function of w

w	Time	Words	Hits	Matches
8	15.9	44,587	118,941	136
9	6.8	44,586	39,218	123
10	4.3	44,585	15,321	114
11	3.5	44,584	7345	106
12	3.2	44,583	4197	98

BLAST algorithm was used to locate similar regions. It used a score of 5 for a match and a score of -4 for a mismatch. The word length was set to 12. The algorithm found 98 alignments scoring over 200. 57 alignments had a score of over 350 and there were 45 paired genes. The table 2.1 [11] shows alignments for various word lengths.

There are different BLAST programs available [6]:

- Nucleotide-Nucleotide BLAST: Returns the most similar DNA sequences.
- Protein-protein BLAST: Returns the most similar protein sequences.
- Blastx: searches for a translated nucleotide in protein sequences.
- tBlastn: searches for a protein in translated nucleotide sequences.
- tBlastx: searches for translated nucleotide in translated nucleotide sequences.

2.1.2 SOAP (Short Oligonucleotide Alignment Program)

The BLAST algorithm is inefficient if a large number of short sequences must be searched in a reference sequence. To overcome this problem and to have both gapped and ungapped alignment, Ruiqiang Li et.al came up with another algorithm called SOAP. SOAP allows, by default, two mismatches and a continuous gap with a size ranging from one to three base pairs (bp) for aligning a short sequence onto the reference sequence [12].

SOAP uses the seed and hash look-up algorithm to accelerate the alignment. Both the short sequence and the reference sequence are encoded with 2-bits per base pair. SOAP then performs an XOR operation between the short sequence and the reference sequence. The resulting value is used as a suffix to check the look-up table to know the bases which are different [SOAP]. SOAP loads the reference sequence as an unsigned 3-byte array. The seed index table is then built for all the reference sequences. Then, seeds are created for the short sequences and the corresponding index table is searched for a match.

The RAM required for storing the reference sequences and seed index tables can be calculated as:

$RAM = \frac{L}{3} + (4 * 3 + 8 * 6) * (4^S) + (4 + 1) * 3 * \frac{L}{4} + 4 * (2^{24})$

where L is the total length of the reference sequences and S is the seed size.

The results showed that to run the SOAP algorithm for a human genome of size L=3 GB and a seed size S = 12 bp, 14 GB of RAM is needed. When SOAP was evaluated with Nucleotide-Nucleotide BLAST using a dataset generated from a 5MB human genome, SOAP was almost 300 times faster for a gapped alignment and 1200 times faster for an ungapped alignment.

Like BLAST, SOAP has also evolved over the years. Following are the different SOAP programs [7]:

- SOAPsv: A program to detect the structural variation.
- SOAPindel: Developed to find the insertion and deletion for re-sequence technology.
- SOAPsnp: Calculates a quality score for each consensus base.
- SOAPsplice: Designed to use RNA-Seq reads for genome-wide abinitio detection of splice junction sites and identification of alternative splicing (AS) events.
- SOAPaligner/soap2: A program for faster and efficient alignment for short oligonucleotide onto reference sequences.
- SOAP3/GPU: GPU based software for aligning short reads with a reference sequence.
- SOAPdenovo: A short read de novo assembly tool, is a package for assembling short oligonucleotide into contigs and scaffolds.

2.1.3 Bowtie

Although the SOAP algorithm is efficient when compared to BLAST, while searching for large number of short sequences from a reference sequence, the computational cost to align many short sequences to a reference sequence is very large. The study from Ley and co-workers has indicated that SOAP would take more than 3-CPU years to align 140 billion bases. The main issue with the SOAP algorithm is that the seed-and-extend is used to align the short sequences [10].

To improve the computational cost of searching for large numbers of short sequences, Ben Langmead et al, proposed a new algorithm called Bowtie. Bowtie uses a different and novel indexing strategy to create an ultrafast, memory efficient short read aligner. Bowtie indexes the reference genome sequence using Burrows-Wheeler transform (BWT) [17] and the FM index [18,19]. It considers conditions like sequencing errors and genetic variations while searching for the short sequence.

In the experiments conducted, Bowtie aligns 35-base pair (bp) and reads at a rate of more than 25 million reads per CPU-hour, which is more than 300 times faster than SOAP. Bowtie can index the reference human genome using a RAM space of 1.3 GB, which implies that a machine with a RAM space of 2 GB is sufficient to index the entire human genome of 2.2 GB. The index is small enough to be distributed over the internet and to be stored on disk and reused. Experiments showed that Bowtie could align 14.3x coverage worth of human illumine reads in about 14 hours on a single desktop computer with four processor cores.

There are a few issues associated with Bowtie. If one or more exact matches exist for a read, then Bowtie is guaranteed to report one. However, if the best match is an inexact one, then Bowtie is not guaranteed to find the highest quality alignment in all cases. With its highest performance settings, Bowtie may fail to align a small number of reads with valid alignments, if those reads have multiple mismatches. Bowtie supports options that increase the accuracy but with some performance cost. With its default options, Bowtie's sensitivity measured in terms of reads aligned is equal to SOAP. Command line options allow the user to increase sensitivity at the cost of greater running time, and to enable Bowtie to report multiple hits for a read.

Bowtie conducts a quality-aware, greedy, randomized, depth-first search through the space of possible alignments. If a valid alignment exists, then Bowtie finds it. Because the search is greedy, the first valid alignment encountered by Bowtie will not necessarily be the 'best' in terms of number of mismatches or in terms of quality. The user may instruct Bowtie to continue searching until it can prove that any alignment it reports is 'best' in terms of number of mismatches (using the option –best). This mode is two to three times slower than the default mode.

The aligner, in some cases, encounters sequences that cause excessive backtracking. This occurs when the aligner spends most of its effort fruitlessly backtracking to positions close to the end of the query. Bowtie mitigates excessive backtracking with the novel technique of 'double indexing'. Two indices of the genome are created: one containing the BWT of the genome, called the 'forward' index, and a second containing the BWT of the genome with its character sequence reversed called the 'mirror' index. Bowtie proceeds in two phases corresponding to those two cases.

- Phase 1 loads the forward index into memory and invokes the aligner with the constraint that it may not substitute at positions in the query's right half.

- Phase 2 uses the mirror index and invokes the aligner on the reversed query, with the constraint that the aligner may not substitute at positions in the reversed query's right half.

Unfortunately, it is not possible to avoid excessive backtracking fully when alignments are permitted to have two or more mismatches. In the experiments it has been observed that excessive backtracking is significant only when a read has many low-quality positions and does not align, or aligns poorly to the reference. These cases can trigger in excess of

200 backtracks per read because there are many legal combinations of low-quality positions to be explored before all possibilities are exhausted. This cost has been mitigated in Bowtie by enforcing a limit on the number of backtracks allowed before a search is terminated. This limit prevents some low quality alignments from being reported.

Bowtie allows the user to select the number of mismatches permitted in the high-quality end of a read. The default value is the first 28 bases. The first 28 bases on the high-quality end of the read are termed the 'seed'. The seed consists of two halves: the 14 bp on the high-quality end and the 14 bp on the low-quality end, termed the 'hi-half' and the 'lo-half', respectively. Assuming the default policy (two mismatches permitted in the seed), a reportable alignment will fall into one of four cases:

- no mismatches in seed (case 1).
- no mismatches in hi-half, one or two mismatches in lo-half (case 2).
- no mismatches in lo-half, one or two mismatches in hi-half (case 3).
- One mismatch in hi-half, one mismatch in lo-half (case 4).

All cases allow any number of mismatches in the nonseed part of the read and they are also subject to the quality distance constraint. The Bowtie algorithm consists of three phases that alternate between using the forward and mirror indices:

- Phase 1 uses the mirror index and invokes the aligner to find alignments for cases 1 and 2.
- Phase 2 finds partial alignments with mismatches only in the hi-half
- Phase 3 attempts to extend those partial alignments into full alignments.
- Finally, phase 3 invokes the aligner to find alignments for case 4.

The performance of Bowtie was evaluated using the reads from the 1000 Genome project. A total of 8.84 million reads, about one lane of data from an Illumina instrument, were trimmed to 35 bp and aligned to the human reference genome. All runs were performed on a single CPU. Bowtie speed-ups were calculated as a ratio of wall-clock alignment times. The time required to build the Bowtie index was not included in the Bowtie running times. Unlike competing tools, Bowtie can reuse a pre-computed index for the reference genome across many alignment runs. It is assumed that the users downloads such indices from a public repository. Results were obtained on two hardware platforms: a desktop workstation with 2.4 GHz Intel Core 2 processor and 2 GB of RAM and a large-memory server with a quad core 2.4 GHz AMD Opteron processor and 32 GB of RAM. These are denoted 'PC' and 'server', respectively. Both PC and server run Red Hat Enterprise Linux AS release 4.

The performance of Bowtie was compared to SOAP. Because its physical memory requirement exceeds the PCs physical memory, SOAP could not run on a PC. The number of reads aligned indicates that SOAP (67.3%) and Bowtie -v 2 (67.4%) have comparable sensitivity. Of the reads aligned by either SOAP or Bowtie, 99.7% were aligned by both, 0.2% were aligned by Bowtie but not SOAP, and 0.1% were aligned by SOAP but not Bowtie. Less than 5 hours is required for Bowtie to both build and query a whole human index with 8.84 million reads from the 1000 Genome project on a server.

2.2 Parallel Algorithms for read mapping using Hadoop

2.2.1 CloudBurst

Algorithms like BLAST, SOAP, and Bowtie are designed for execution on a single computing node and therefore require a long running time or they limit the sensitivity of the alignments found. CloudBurst is a new parallel read-mapping algorithm optimized for mapping next-generation sequence data to the human genome and other reference genomes, for use in a variety of biological analysis including SNP discovery, genotyping and personal genomics. It reports either all alignments or an unambiguous best alignment for each read with any number of mismatches or differences. This level of sensitivity could be prohibitively time consuming, but CloudBurst uses the open-source Hadoop implementation of MapReduce to parallelize execution using multiple compute nodes [14].

CloudBurst is a highly sensitive parallel seed-and-extend read-mapping algorithm optimized for mapping single-end next generation sequence data to reference genomes. It reports all alignments for each read with up to a user specified number of differences including mismatches. CloudBurst is a MapReduce based read-mapping algorithm . It runs in parallel on multiple machines with Hadoop. It is optimized for mapping many short reads from next-generation sequencing machines to a reference genome allowing for a user specified number of mismatches or differences. The seed size is $s = \frac{m}{k+1}$ is computed from the minimum length of the reads (m) and the maximum number of differences or mismatches (k). It attempts to extend the exact seeds to count the number of mismatches in an end to end alignment using that seed, and reports alignments with atmost k mismatches. Alternatively, like BLAST, it can extend the exact seed matches into end-to-end gapped alignments.

As a MapReduce algorithm, CloudBurst is split into map, shuffle and reduce phases. The map function emits k-mers of length s as seeds from the reads and reference sequences. The shuffle phase groups together K-mers shared between the read and reference sequences. Finally, the reduce function extends the shared seeds into end-to-end alignments allowing mismatches. The input to the application is a multi-fasta file containing the reads and a multi-fasta file containing one or more reference sequences. These files are first converted to binary Hadoop Sequence Files and copied into the HDFS. The DNA sequences are stored as the keyvalue pairs (id, SeqInfo), where SeqInfo is the tuple (sequence, start-offset) and sequence is the sequence of bases starting at the specified offset. By default, the reference sequences are partitioned into chunks of 65 kb overlapping by 1 kb, but the overlap can be increased to support reads longer than 1 kb. The overview of the CloudBurst algorithm is shown in the figure 2.1 [14] .

CloudBurst was evaluated in a variety of configurations for the task of mapping random

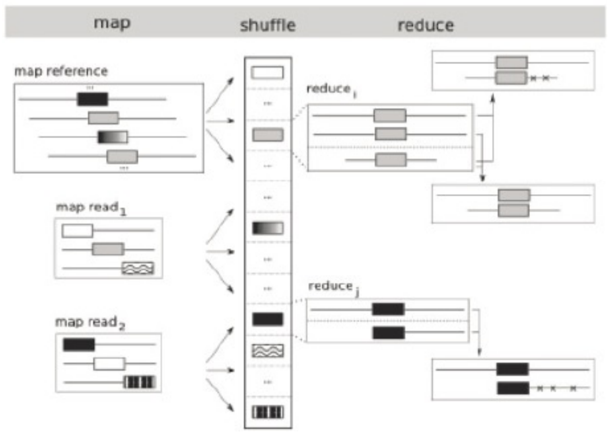


Figure 2.1: Cloudburst Algorithm

subsets of 7.06 million publicly available Illumina/Solexa sequencing reads from the 1000 genome project (accession SRR001113) to portions of the human genome (NCBI Build 36) allowing up to four mismatches. All reads were exactly 36 bp long. The test cluster had 12 compute nodes, each with a 32 bit dual core 3.2 GHz Intel Xeon (24 cores total) and 250 GB of local disk space. The compute nodes were running RedHat Linux AS Release 3 Update 4, and Hadoop 0.15.3 set to execute two tasks per node (24 simultaneous tasks total).

In a test which was conducted to test how CloudBurst scales as the number of reads increases and as the sensitivity of the alignment increases, the reads were mapped to the full human genome. To improve load balancing across the cores, the number of mappers was set to 240, the number of reducers was set to 48, and the redundancy for low-complexity seeds was set to 16. The results show that CloudBurst scales linearly in execution time as the number of reads increases. Aligning all 7M reads to the genome with four mismatches failed to complete after reporting close to 25 billion mappings due to lack of disk space availability. Allowing more mismatches increases the runtime superlinearly, because higher sensitivity requires shorter seeds with more chance occurrences.

CloudBurst was next evaluated on the Amazon EC2. This environment provides unique opportunities for evaluating CloudBurst, because the performance and size of the cluster are configurable. The first test compared two different EC2 virtual machine classes with the local dedicated 24-core Hadoop cluster described above. In all three cases, the number of cores available was held constant at 24, and the task was mapping all 7M reads to the human chromosome 22 with up to four mismatches, with runtimes averaged over three runs. The first configuration had 24 Small Instance slaves running Hadoop 0.17.0 and provides one virtual core with approximately the performance of a 1.01.2 GHz 2007 Xeon processor. The second configuration had 12 High-CPU Medium Instance slaves, also running Hadoop 0.17.0 but offers two virtual cores per machine and have been benchmarked to have a total performance approximately five times the small instance type. The running time for the High-CPU Medium Instance class was 1667 seconds, and was substantially better per dollar than the Small Instance class at 3805 seconds, and even exceeds the performance of the local dedicated cluster at 1921 seconds.

The final experiment evaluated CloudBurst when the size of the cluster increases for a fixed problem. In this experiment, the number of High-CPU Medium Instance cores was varied between 24, 48, 72 and 96 virtual cores for the task of mapping all 7M reads to human chromosome 22. Figure 2.2 [14] shows the running time on these clusters averaged over three runs. The results show CloudBurst scales very well as the number of cores increases: the 96-core cluster was 3.5 times faster than the 24-core cluster.

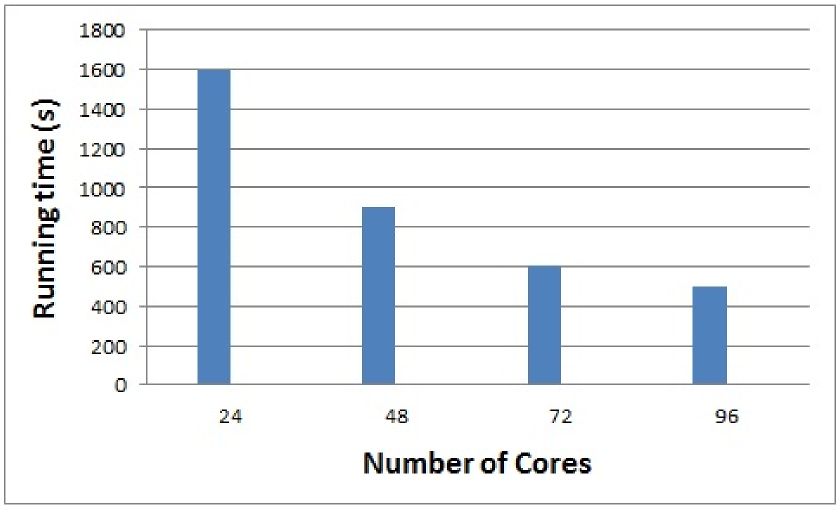


Figure 2.2: Cloudburst Results

2.2.2 Rapid Parallel Genome Indexing with MapReduce

DNA sequence alignment is an application in computational biology where four different nucleotides are matched or arranged to find the maximal similarity. The genome of an organism varies in length from a few million nucleotides for simple bacterial organisms, three billion nucleotides for the human genome, to hundreds of billions of nucleotides for the largest known genomes. Pre-computing the index structure of these genomes helps in rapid searching of patterns without loss of sensitivity and exhaustively scanning every individual nucleotide. Two of the most important indexing techniques considered are suffix array and Burrows-Wheeler Transform (BWT) [13].

The suffix array consists of the lexicographically sorted list of all suffixes of the string. Because the suffixes are sorted, this simple index allows for rapid binary search algorithms for matching query sequences of any length. More advanced algorithms even allow differences between the reference and query sequences during the suffix array matching. The closely related BWT reduces the space requirements of the suffix array, which requires > 12 GB for the human genome, by recording as an index a (reversible) permutation of the string based on the ordering of the suffix array. As a permutation, the size of the index is the same size as the string itself. So it will be 3 GB for the human genome.

As the construction of suffix array and BWT require large computation time on a desktop workstation, an approach using MapReduce has been proposed in this research. The algorithm uses the inherent data processing capabilities of Map-Reduce to divide the suffix array construction into multiple independent ranges that are then independently solved.

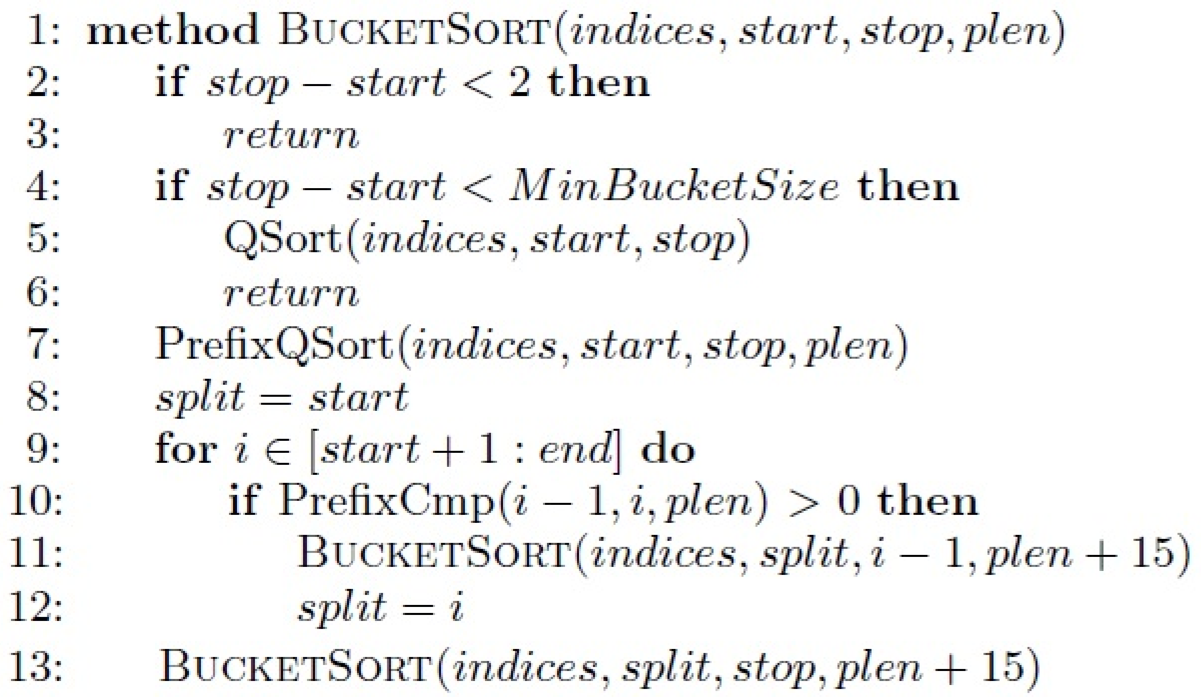
The basic index construction technique has two major problems: first, the reference human genome sequence contains a region consisting of > 21 million Ns, representing a large region of ambiguity in the genome assembly. The partitioner used in basic index construction technique uniformly divides the space of mers, but there is a non-uniform distribution of mers in the genome: all 21 million of these suffixes have the same prefix value consisting of P Ns, while most prefix mers occur just a few thousand times.

Secondly, these repeats are very expensive to sort, since many millions of characters will be examined to find the first difference between them. This causes the runtime per reducer to approach O((n^2) log n) for repetitive regions of length n.

Rapid Parallel Genome Indexing with MapReduce algorithm improves load balancing among reducers by a new partitioner that samples the genome to select the boundaries of the batches, based on true sequence distribution and the batch sorting performance is improved by an optimized recursive bucket sort that pre-computes and determines problematic repeats on the fly.

The sampling partitioner uses dynamic ranges to balance the number of suffixes per batch, rather than the number of mers per batch as proposed in the basic algorithm. The boundaries between batches are determined by R 1 variable length mers computed from a sorted list of suffixes uniformly sampled from the reference string, where R is the desired number of reducers. The variable length boundary mers enable long repetitive sequences, with correspondingly long identical prefixes, to be subdivided into separate batches. The boundary mers are recorded using a runlength encoding so that they contain at least 2 distinct characters. These boundary mers are stored in a file and distributed to all of the partitioners at launch. Furthermore, the partitioner quickly computes the partition for each suffix by exploiting the relationship between consecutive suffixes. Finally, the partition represents each character of the reference string with at most 3 bits, and reduces the memory requirement for analyzing the human genome from 3.0 GB to 1.2 GB.

The recursive bucket sort is very effective for sorting most sets of suffixes because it restricts the number of suffixes that will be compared over their full length. The algorithm first orders the suffixes based on a plen length prefix (initially plen = 15) using a version of Quicksort that only compares the first plen characters of a suffix. It then scans the list, and recursively sorts blocks of suffixes with the same prefix. To limit the recursion depth, if the size of the current range is less than MinBucketSize (default: 10% of the original range), the algorithm reverts to a Quicksort over the entire suffix length. This algorithm also optimizes the single and multiple character repeats. The bucket sort algorithm is as shown in the figure 2.3 [13] .



```
method BUCKETSORT($indices, start, stop, plen$)
    if $stop - start < 2$ then
        $return$
    if $stop - start < MinBucketSize$ then
        QSort($indices, start, stop$)
        $return$
    PrefixQSort($indices, start, stop, plen$)
    $split = start$
    for $i \in [start + 1 : end]$ do
        if PrefixCmp($i - 1, i, plen$) $> 0$ then
            BUCKETSORT($indices, split, i - 1, plen + 15$)
            $split = i$
    BUCKETSORT($indices, split, stop, plen + 15$)
```

Figure 2.3: Bucket Sort Algorithm

The code was executed on Hadoop clusters with 30, 60, or 120 concurrent tasks, using machines leased from the Amazon Elastic Compute Cloud (EC2). The experiments were performed using High-Memory Double Extra Large (m2.2xlarge) instances, which provide 4 hyper-threaded cores at 3.2 EC2 Compute Units (roughly 3.2 GHz), 34.2 GB of RAM, and 850 GB of storage. For the 30 and 60 core evaluations, 10 and 20 nodes were used with 3 tasks per node; for the 120 core evaluation 20 nodes were used with 6 tasks per node. Figure 2.4 [13] and figure 2.5 [13] show the end-to-end runtime for constructing the suffix arrays in 3 cluster environments. The figures show the performance greatly improved for as many as 120 processing cores, although the large genomes demonstrate better scaling to large numbers of cores. The suffix array construction for HG19 using 60 cores was 1.96x faster than the 30 core cluster, and the 120 core cluster was 2.37x faster than the 30 core cluster.

Figure 2.4 [13] and figure 2.5 [13] the running time as a function of the genome size for the

5 test genomes as 3 cluster environments. For a given cluster size, there is an approximately linear relationship between the genome size and the runtime, indicating that the algorithm performs well in the presence of complicated repeats.

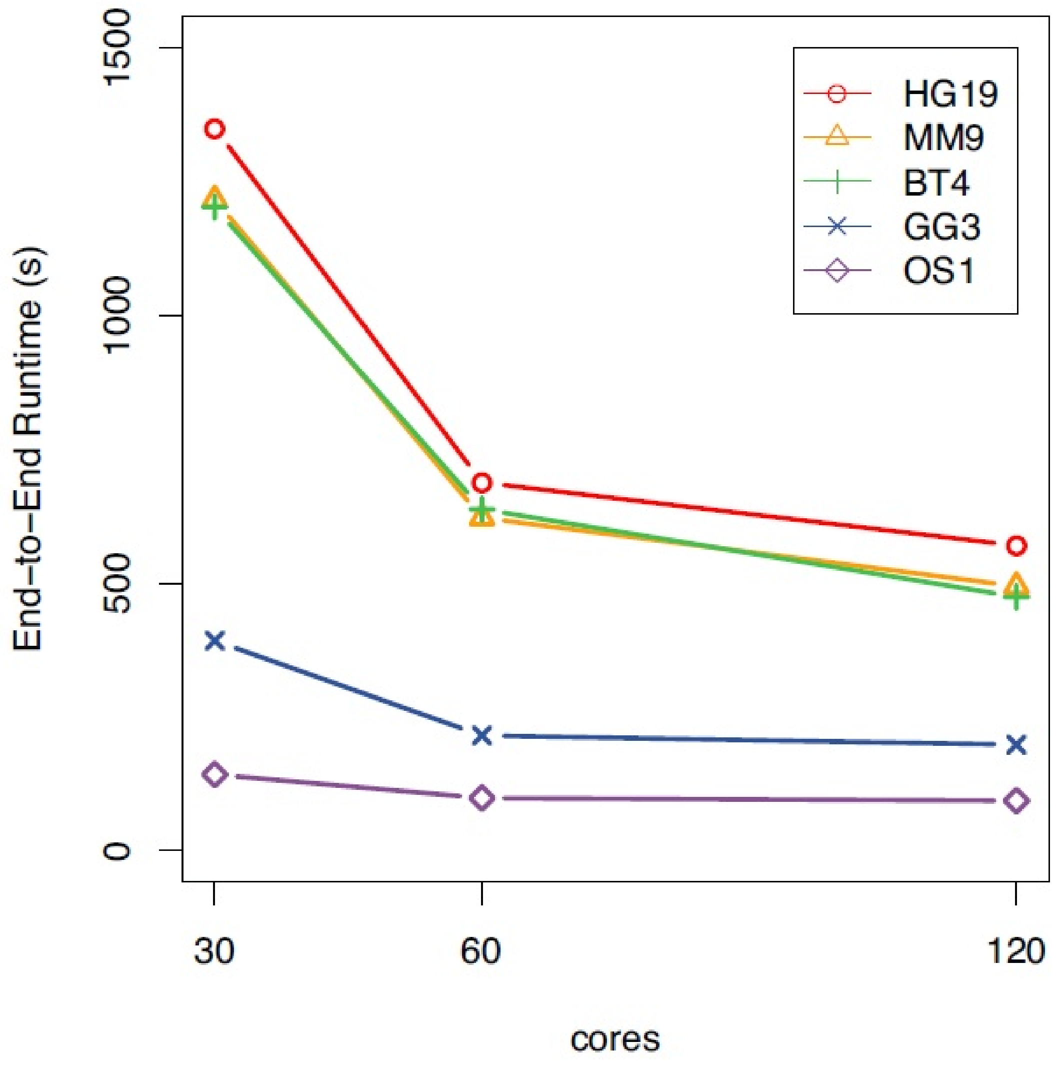


Figure 2.4: Suffix Array Results

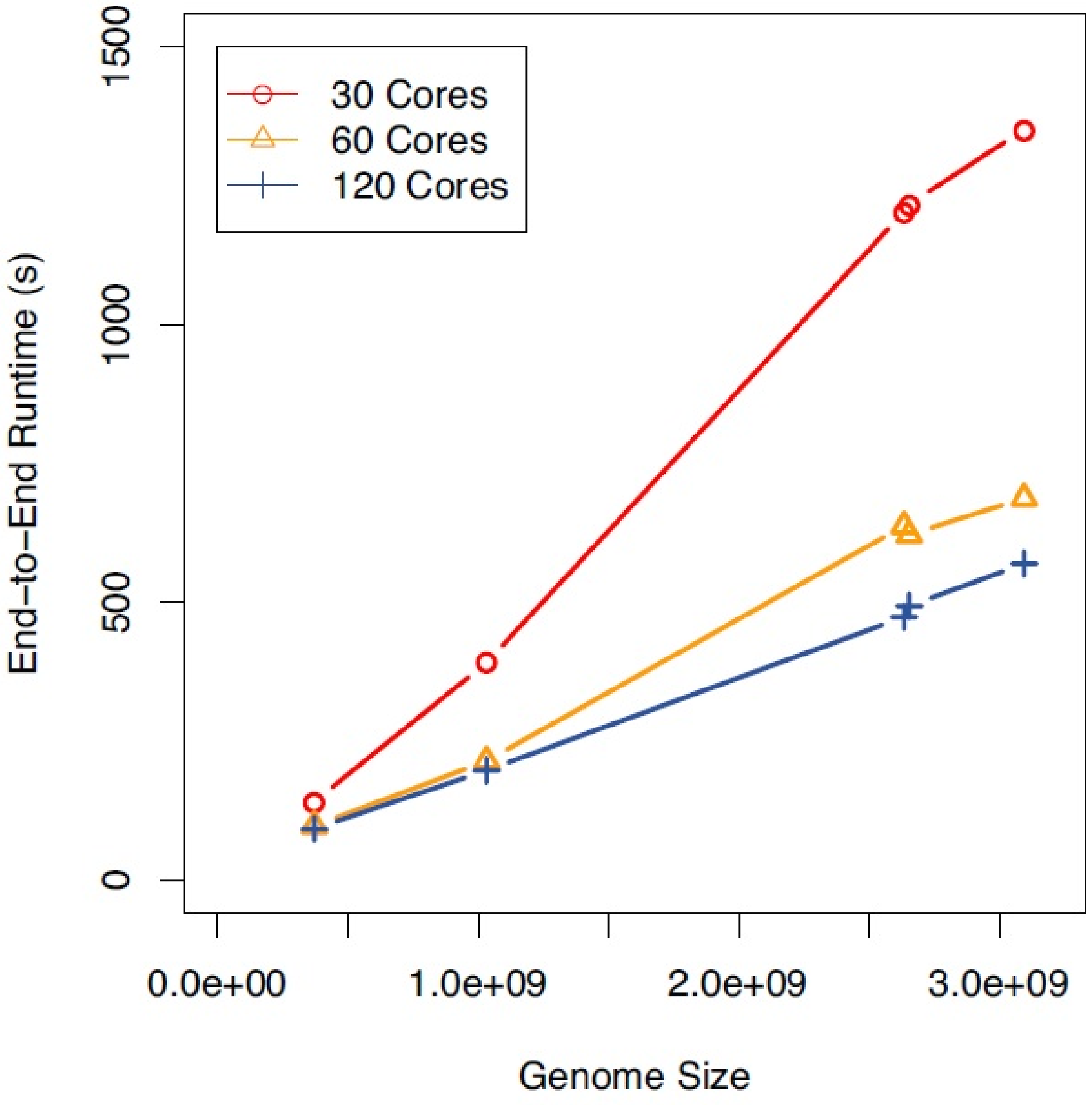


Figure 2.5: Suffix Array Results

CHAPTER 3
SUFFIX TREE AND HADOOP FRAMEWORK

3.1 Suffix Tree

Let S = s0s1s2s3s4. . .sn, where s0,s1,s2. . .sn represent the characters in the string and n represents the length of the string. The suffixes of the string S are s0s1s2s3. . .sn, s1s2s3s4. . .sn, s2s3s4. . .sn, and so on. The last suffix of the string S is sn. The suffix tree is a tree that indexes all the suffixes of a string. For the string S, the suffix tree will have n leaves and the edges are labeled by the characters in the string. If a suffix tree is traversed from the root r to a leaf i, then it indicates a suffix S[r,i] of the string S. The suffix tree can be used to accelerate searching of short sequences in reference human genome. For example, searching for a short sequence of length p in a suffix tree takes O(p) time.

Some of the important properties of the suffix tree are:

- The path from the root to each leaf in the suffix tree corresponds to one suffix of the reference human genome.
- Each internal node of a suffix tree has atleast two children and each edge is labeled with a substring of the string.
- The edges coming out of a node will have labels starting from different alphabets.

In the case of human genome, which is a string comprising of four characters, A,C,G, and T, each node can have a maximum of four edges coming out of them.

The suffix tree for a string S = GTCCGAAGCTCCGG$ is shown in Figure 3.1:

The suffixes of string S have prefixes, which we refer in this thesis as S-prefixes. In the figure 3.1,

- The suffixes sharing the S-prefix A are AGCTCCGG$ and GCTCCGG$
- The suffixes sharing the S-prefix C are CGAAGCTCCGG$, CGG$, GAAGCTCCGG$, GG$, and TCCGG$

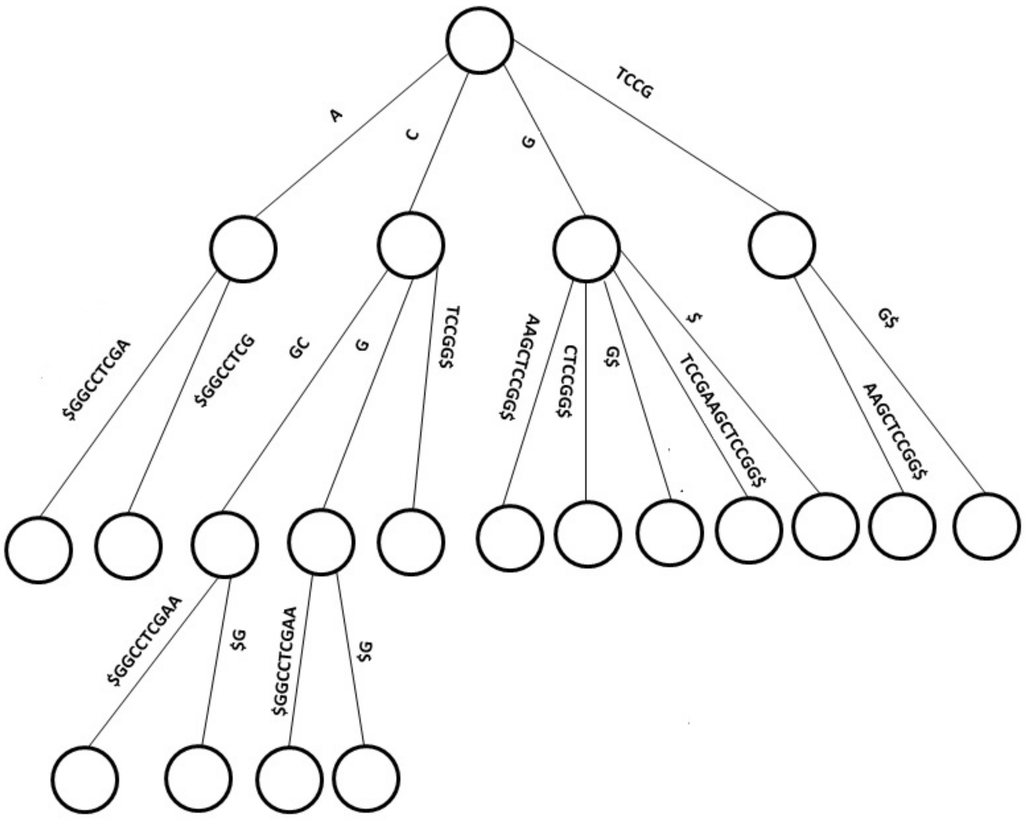


Figure 3.1: Suffix Tree

- The suffixes sharing the S-prefix G are AAGCTCCGG$, CTCCGG$, G$, and TCC-GAAGCTCCGG$
- The suffixes sharing the S-prefix TCCG are AAGCTCCGG$ and G$ The string S is padded with a symbol $. This is a terminal symbol which ensures that no suffix Sa is a proper S-prefix of any other suffix Sb and there will be n leaf nodes, one for each of the n suffixes of S.

3.2 MapReduce and Hadoop

A computationally intense task can be executed at a faster rate by running it in parallel. If the task can be divided into many splits which have no dependency on each other, then we can create different processes for each split using the available hardware threads on a machine. However, there are few problems associated with this technique:

- Dividing the task into different splits is not easy. The split size may vary and as a result each process might end at a different time.
- Combining the results from each independent processes will need further processing.
- As all the processes run on the same machine, the task completion is still dependent on the processing capacity of that machine.

To overcome such problems, Google developed a new programming model called MapReduce. It was mainly developed for scanning trillions of webpages to compute the most relevant pages for a search query. The MapReduce can intelligently distribute computation across a cluster with hundreds or thousands of computers, each analyzing a portion of the dataset stored locally on the compute node. After an initial round of independent parallel computation, the machines efficiently exchange intermediate results, from which the final results are computed in parallel.

Parallel computation in MapReduce is structured into three major phases called map, shuffle, and reduce. The map phase scans the input dataset and emits key-value pairs representing some relevant information of the data tagged by the key. These key-value pairs are then distributed and shuffled so that all values associated with a given key are collected into a single list. Finally, these key-value lists are then processed by the reduce function to compute the final results [13]. The illustration of MapReduce phase is shown in figure 3.2 [13].

Hadoop is a framework which uses the MapReduce programming model to perform computationally intense task in parallel. It is a reliable shared storage and analysis system where the storage is provided by HDFS (Hadoop Distributed File System) and the analysis is provided by MapReduce. Hadoop can store any type of data on a large scale and can do computationally intense analysis of the data easily and quickly. It can detect system failures and can easily recover from it. It provides scalability, reliability, and fault tolerant services

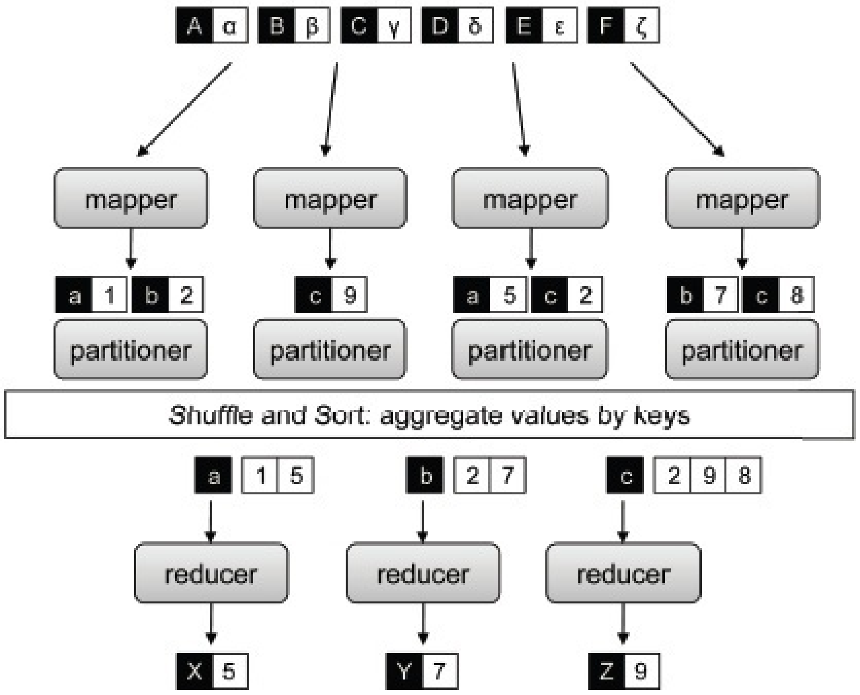


Figure 3.2: MapReduce

for data storage and analysis at very low cost. Hadoop doesnt require expensive and highly reliable hardware. It has been designed to run on clusters of commodity hardware [15].

One of the important sub-systems of the Hadoop framework is HDFS (Hadoop Distributed File System). HDFS is a file system designed for storing very large files with streaming data access patterns, running on clusters on commodity hardware. Very large in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size. There are Hadoop clusters running today that store petabytes of data. HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. A dataset is typically generated or copied from source, and then various analyses are performed on that dataset over time. Each analysis will involve a large proportion, if not all, of the dataset, so the time to read the whole dataset is more important than the latency in reading the first record [15].

Files in HDFS are broken into block-sized chunks, which are stored as independent units. Unlike a file system for a single disk, a file in HDFS that is smaller than a single block does not occupy a full blocks worth of underlying storage. The default block size is 64MB. A HDFS cluster has two types of nodes operating in a master-worker pattern: a name node (the master) and a number of datanodes (workers). The namenode manages the file system namespace. It maintains the file system tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log. The namenode also knows the datanodes on which all the blocks for a given file are located, however, it does not store block locations persistently, since this information is reconstructed from datanodes when the system starts. Without the namenode, the file system cannot be used. Datanodes store and retrieve blocks when they are told to (by clients or the namenode), and they report back to the namenode periodically with lists of blocks that they store

Another important subsystem of Hadoop is MapReduce. Hadoop runs the Map Reduce job by dividing it into two tasks: map tasks and reduce tasks. There are two types of nodes that control the job execution process: a jobtracker and a number of tasktrackers. The jobtracker coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers. Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of each job. If a task fails, the jobtracker can reschedule it on a different tasktracker

Hadoop divides the input to a MapReduce job into fixed-size pieces called input splits, or just splits. Hadoop creates one map task for each split, which runs the user- defined map function for each record in the split. The whole process of how Hadoop runs a MapReduce is shown in the figure 3.3 [15].

The runJob method on JobClient is a convenience method that creates a new JobClient instance and calls submitJob on it. Having submitted the job, runJob polls the jobs progress once every second, and reports the progress to the console if it has changed since the last

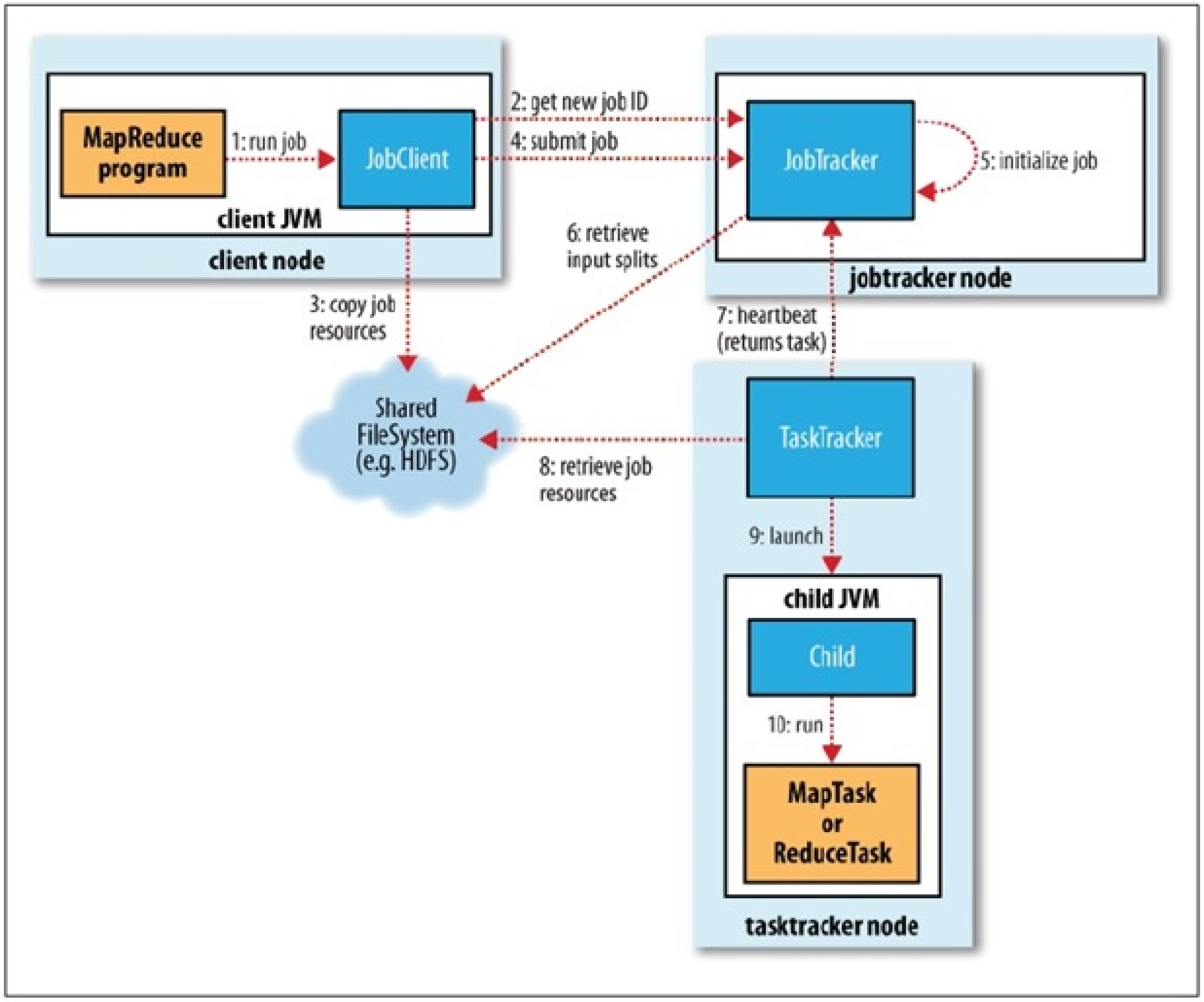


Figure 3.3: Hadoop Phases

report. When the job is complete, if it was successful, the job counters are displayed. Otherwise, the error that caused the job to fail is logged to the console.

The job submission process implemented by JobClients submitJob method does the following:

- Asks the jobtracker for a new job ID by calling getNewJobId on JobTracker.
- Checks the output specification of the job. For example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the MapReduce program.

- Computes the input splits for the job. If the splits cannot be computed, because the input paths dont exist, for example, then the job is not submitted and an error is thrown to the MapReduce program.

- Copies the resources needed to run the job, including the job JAR file, the configuration file and the computed input splits, to the jobtrackers filesystem in a directory named after the job ID. The job JAR is copied with a high replication factor (controlled by the mapred.submit.replication property, which defaults to 10) so that there are lots of copies across the cluster for the tasktrackers to access when they run tasks for the job.

- Tells the jobtracker that the job is ready for execution.

When the JobTracker receives a call to its submitJob method, it puts it into an internal queue from where the job scheduler will pick it up and initialize it. To create the list of tasks to be run, the job scheduler first retrieves the input splits computed by the JobClient from the shared file system. It then creates one map task for each split. The number of reduce tasks to create is determined by the mapred.reduce.tasks property in the JobConf, which is set by the setNumReduce Tasks method, and the scheduler simply creates this number of reduce tasks to be run. Tasks are given IDs at this point.

Tasktrackers run a simple loop that periodically sends heartbeat method calls to the jobtracker. Heartbeat methods tell the jobtracker that a tasktracker is alive. As a part of the heartbeat, a tasktracker will indicate whether it is ready to run a new task. If it is, the jobtracker will allocate a task to it. This is communicated to the tasktracker using the heartbeat return value. Tasktrackers have a fixed number of slots for map tasks and for reduce tasks. When the task tracker has been assigned a task, the next step is for it to run the task. First, it localizes the job JAR by copying it from the shared file system to the tasktrackers file system. It also copies any files needed by the application from the distributed cache to the local disk. Second, it creates a local working directory for the task, and un-jars the contents of the JAR into this directory. Third, it creates an instance of

TaskRunner to run the task.

TaskRunner launches a new Java Virtual Machine to run each task, so that any bugs in the user-defined map and reduce functions dont affect the tasktracker. The child process communicates with its parent through the umbilical interface. This way, it informs the parent of the tasks progress every few seconds until the task is complete. When a task is running, it keeps track of its progress. If a task reports progress, it sets a flag to indicate that the status change should be sent to the tasktracker. The tasktracker sends heart beat signals to the jobtracker every five seconds and hence the jobtracker is informed about the progress from all the tasktrackers. When the jobtracker receives a notification that the last task for a job is complete, it changes the status for the job to successful. Then, when the JobClient polls for status, it learns that the job has completed successfully, so it prints a message to tell the user, and then returns from the runJob method.

CHAPTER 4
SUFFIX TREE CONSTRUCTION

4.1 Ukkonens suffix tree construction algorithm

The Ukkonen algorithm [16] constructs the suffix tree in O(n) time by fetching one character from the input string in each step of the algorithm. Here, n represents the length of the string S whose suffix tree is to be built. In each step, an implicit suffix tree is built, the last of which is converted to an actual suffix tree of the input string. Generally, a termination character like $ is appended to the end of the input string while constructing the suffix tree. An implicit tree for a string S is defined as a tree obtained from the suffix tree for S$ by removing every copy of the terminating symbol $ from the edge labels of the tree, then removing any edge that has no label, and then removing any node that does not have at least two children.

The algorithm begins by reading the first character of S. A root node is created and an edge for the first character is created from the root node. This is the first implicit tree. As the algorithm proceeds, this tree is expanded by one character. In each step, all the suffixes will be expanded to take into account the new character, there by expanding the implicit suffix tree [9].

The expansion of the tree is done as follows:

- If the suffix is found to be on the edge connected to a leaf, then the new character is just added to the edge.
- If the suffix ends at a node then a new edge is created from that node and the new character is added to the new edge.
- If the suffix ends in the middle of an edge, then the edge is split by adding a new internal node and attaching a new edge to that node. The new character is added to the new edge.

- If the suffix is found to be a part of an already found suffix, then nothing is done in that step.

If a new character is to be added to the existing branches in the implicit suffix tree, then it requires navigating through all the nodes in the implicit suffix tree. This can be avoided by having suffix links.

As shown in figure 4.1, let T[i, j] be a string on the edge from root R to node U and let this be called path(U). The suffix link of node U can be defined as a pointer to another node V where path(V) is obtained by deleting the first character from path(U).

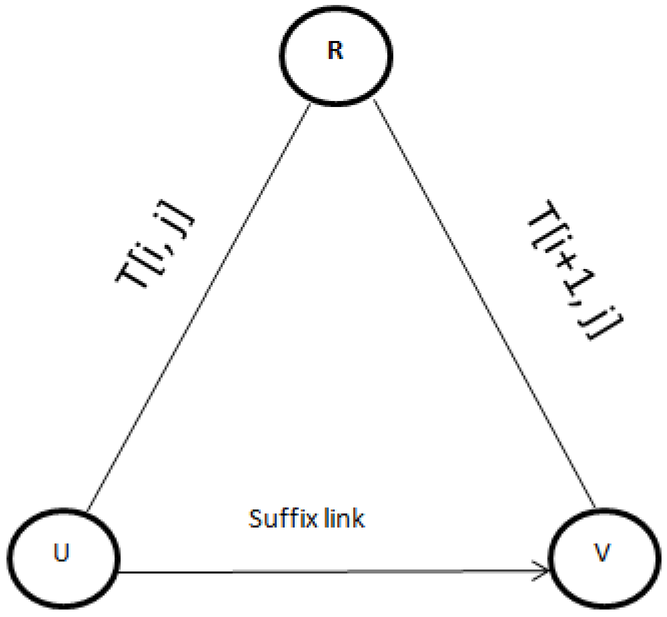


Figure 4.1: Ukkonen Suffix Tree

Suffix link helps in navigating between the branches by keeping track of the path followed as the implicit tree is expanded.

4.2 MapReduce based Parallel Suffix Tree Construction

The computation time for constructing a suffix tree for big data like the human genome is high. This algorithm improves the computation time for constructing the suffix tree for such big data. In this thesis, a distributed algorithm has been proposed to construct the suffix tree for big data like human genome. Ukkonens algorithm is considered as base in this thesis. The problem of constructing the suffix tree is divided into vertical partitions. Vertical partitioning, splits the suffix tree into sub-trees that fit into the available physical memory. The subtrees for S-prefixes are constructed in parallel. There are two ways to achieve vertical partition:

4.2.1 Fixed length S-prefixes

- This technique starts by choosing the S-prefix length as one and identify all the unique characters of length one. Then an analysis is done to verify that the subtrees, if built for those characters, fit into the available memory.
- If not, then the S-prefix length is increased to the next size, unique S-prefixes of that size are fetched and analysed if the subtrees were built using these S-prefixes then would they fit into the available memory.

The step 2 is continued until the analysis of subtrees for all the S-prefixes fit into the available memory.

For example, the human genome is a string consisting of four unique characters A, C, G, and T. S-prefix length of one is chosen and analysed if the subtrees were to be built for A, C, G, and T then would they fit into the available primary memory. If not, then the prefix length is increased to the next size.

4.2.2 Variable length S-prefixes

This technique starts similar to that of fixed length S-prefix. But at any stage, if the characters whose corresponding subtrees fit into the available memory, then those characters are not considered when the S- prefix length is increased in the next stage. Hence the technique ends up getting variable length S-prefixes.

All the S-prefixes of the string are generated such that the subtrees for all the S-prefixes form the actual suffix tree for the reference genome.

In the fixed length S-prefixes technique, the available memory might be underutilized by building small trees in different map tasks. Load balancing is not possible as some map tasks may construct trees of very small size and some map tasks may construct trees of size equivalent to the available memory. Hence, variable length S-prefixes technique is considered in this thesis so that all the map tasks construct subtrees of same size and thereby achieve load balancing and never underutilize the available memory.

For the String S=GTCCGAAGCTCCGG$, whose suffix tree is shown in figure 4.1, some of the prefixes generated by variable length prefix technique are A, CG, CCG, and TCCG. The subtrees of these prefixes are as shown in the figure 4.2, figure 4.3, figure 4.4, and figure 4.5.

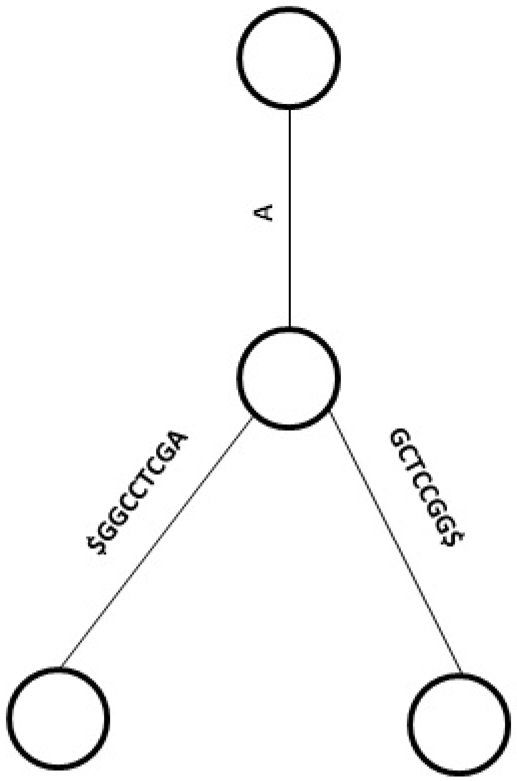


Figure 4.2: Subtree for A

The MapReduce based Parallel Suffix Tree Construction algorithm extends the existing

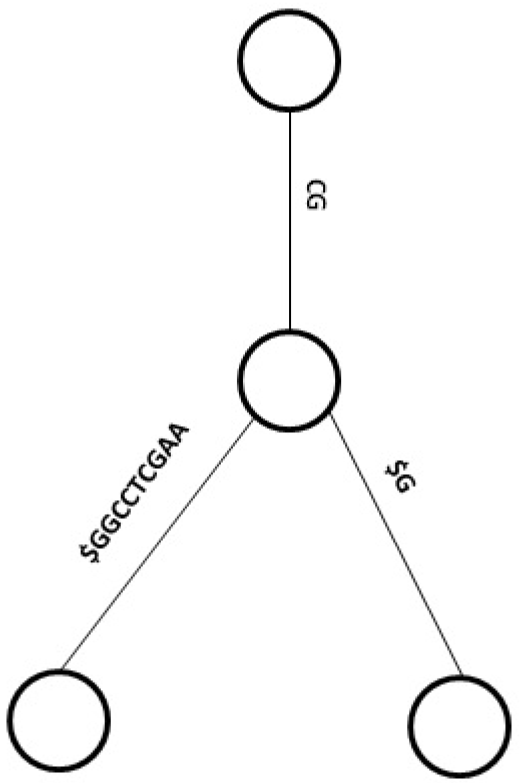


Figure 4.3: Subtree for CG

Ukkonens algorithm. The algorithm starts from the first character of the S-prefix. A root node is created and an edge for the first character is created from the root node. All the characters of the S-prefix are added to this edge. This is the first implicit tree. In each step, all the suffixes of the human genome, starting from the input S-prefix, will be expanded to take into account the new character.

The expansion of the tree is done as follows:

- If the suffix of S, starting from the input S-prefix, is found to be on the edge connected to a leaf, then the new character is just added to the edge.
- If the suffix of S, starting from the input S-prefix, ends at a node then a new edge is created from that node and the new character is added to the new edge.
- If the suffix of S, starting from the input S-prefix, ends in the middle of an edge, then the edge is split by adding a new internal node and attaching a new edge to that node.

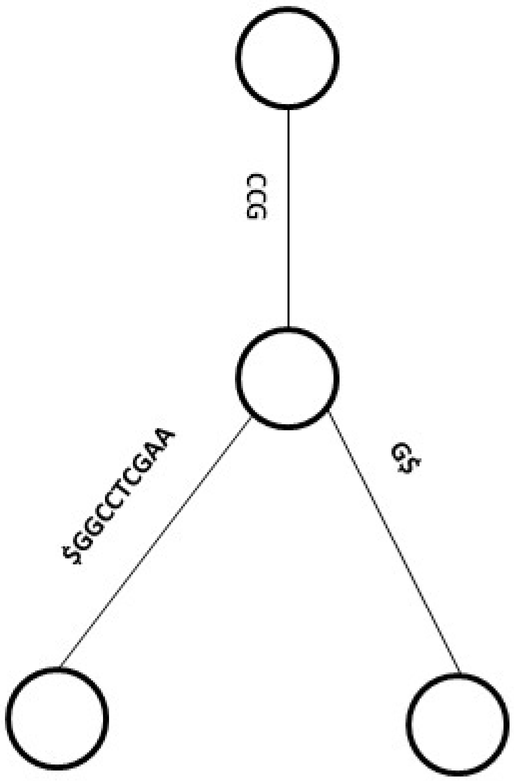


Figure 4.4: Subtree for CCG

The new character is added to the new edge.

- If the suffix of S, starting from the input S-prefix, is a part of an already found suffix of the S-prefix, then nothing is done in that step.
- After the subtree is constructed, each suffix of S, starting from the input S-prefix, is indicated by the path from the root to a leaf of the subtree.

The main improvement of the MapReduce based algorithm from the Ukkonens algorithm is handling of the suffix link. In the case of Ukkonens algorithm, the suffix link helps in navigating between the branches of the implicit suffix tree. In the MapReduce based algorithm, as the subtree of the prefix is getting built, the suffix link helps in navigating between the branches of the implicit subtree of the prefix.

As shown in figure 4.6, let T[i, k-1] be a prefix on the edge from root R to node U and let this be called path(U).Let T[k, j] be a substring of the reference human genome starting

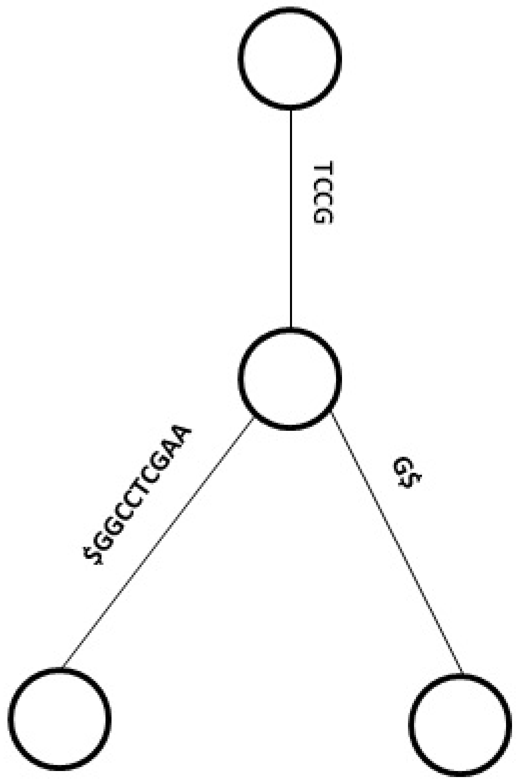


Figure 4.5: Subtree for TCCG

from the prefix T[i, k-1] and let this string be on the edge from U to V. Then path(V) can be obtained by concatenating T[i, k-1] and T[k, j]. Hence, path(V) is T[i, j].

The suffix link of node V can be defined as a pointer to another node W where path(W) is obtained by deleting m characters from T[i, j] in path(V) where m represents the number of characters between the end character of the prefix to the starting character of the same prefix in the next occurrence in the string. In the figure 4.6, m represents characters between T[k-1, k+m].

As the subtree is getting constructed, there will not be a suffix which is not a part of the reference human genome starting from the prefix in the subtree. Hence, the suffix link can never point to a node of another branch which is not a part of the subtree being constructed.

The MapReduce based algorithm considers a pointer called active point which has three parameters active node, active edge, and active length. For example, if the active point, in

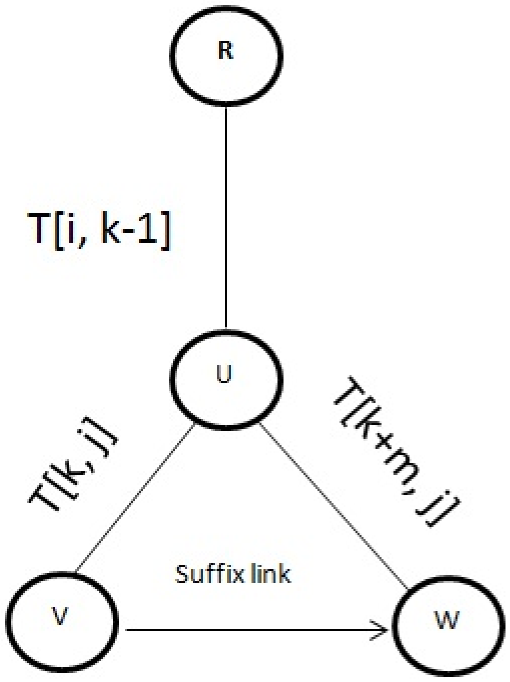


Figure 4.6: MapReduce based Suffix Tree

the current step, is (root,\0, 0) then it indicates that a new edge is inserted from the root node in the current step at position 0. Active point is very useful for expanding the implicit suffix tree. The active point is updated in each step. Hence, the current node is specified by the active node and the position to insert the currently fetched character is specified by the active length.

When a new character is fetched, then it is added to active edge. After adding the new character to the active edge the algorithm checks if there is a suffix link from the current node, if not then it finds the nearest node with a suffix link, travels through the path of the suffix link, checks for the last position in that branch, updates the active point, and adds the character to the active edge.

```
1. Method constructSuffixTree(referenceHumanGenome, prefix)
2. prefix_startposition := starting position of the prefix in the referenceHumanGenome
3. prefix_length := prefix.length
4. for position := prefix_startposition to referenceHumanGenome. length 1 do
          5. if position = 0 then
               6. Create root node
               7. Add referenceHumanGenome [position] to a fresh edge
               8. Insert edge to the subtree
          9. else if position < prefix_length then
               10. Add referenceHumanGenome [position] to the current edge
          11. else if referenceHumanGenome [position] := starting character of the edge
              from the root
               12. Add the character to the active edge
               13. Check for the nearest suffix link
               14. Travel through the path of the suffix link
               15. Find the edge with the referenceHumanGenome [position - 1]
               16. Update the active point
               17. Add referenceHumanGenome [position] to the active edge
```

CHAPTER 5
EXPERIMENTAL RESULTS AND ANALYSIS

The parallel suffix tree construction algorithm was implemented for Hadoop 0.20.203.0. The construction of the suffix tree was implemented in the mapper using Java. The code was executed on Hadoop cluster with 30, 60, and 120 nodes. The nodes were requested from Queen bee cluster. Each node had 2.33 GHz quad core xeon 64-bit processors, 8GB RAM, 10Gbps infiniband network interface, and Red Hat Enterprise Linux operating system. The suffix tree construction time was analysed for human genome of size 500MB, 1GB, 2GB, and 3GB with 30, 60, and 120 nodes.

The map tasks were generated using the variable length technique. Each of the map task constructed a set of subtrees of variable length.

Table 5.1 shows the number of map tasks of different human genome size.

Table 5.2, table 5.3 and table 5.4, the evaluation of the suffix tree construction for the human genome of size 500MB, 1GB, 2GB, and 3GB with 30, 60, and 120 nodes.

As it can be observed from the table 5.1, table 5.2, and table 5.3, the best map time, the average map time, and the worst map time remain same for the construction of suffix tree for the human genome of size 500MB, 1GB, 2GB, and 3GB with 30, 60, and 120 nodes. But, the construction time decreases as the number of nodes increase. This indicates that the scalability improves with the increase in number of nodes. figure 5.1 shows the variation of best map time, avarage map time, and worst map time for human genome of size 500MB, 1GB, 2GB, and 3GB.

As shown in figure 5.2, the construction time for a human genome of size 500MB remains

Table 5.1: Number of Map tasks for different genome size

Genome Size	Number of Map tasks
500 MB	16
1 GB	52
2 GB	124
3 GB	224

Table 5.2: Evaluation with 30 nodes (Time in minutes)

Genome Size	Construction Time	Best Map Time	Average Map Time	Worst Map Time
500 MB	6.28 mins	3 mins, 58 sec	4 mins, 53 sec	5 mins, 54 sec
1 GB	9.51 mins	2 mins, 31 sec	4 mins, 14 sec	5 mins, 52 sec
2 GB	34.45 mins	4 mins, 25 sec	7 mins, 16 sec	13 mins, 38 sec
3 GB	132.36 mins	7 mins, 34 sec	16 mins, 53 sec	28 mins, 47 sec

Table 5.3: Evaluation with 60 nodes (Time in minutes)

Genome Size	Construction Time	Best Map Time	Average Map Time	Worst Map Time
500 MB	6.28 mins	3 mins, 36 sec	4 mins, 45 sec	5 mins, 50 sec
1 GB	6.58 mins	2 mins, 35 sec	4 mins, 17 sec	5 mins, 51 sec
2 GB	20.66 mins	3 mins, 50 sec	6 mins, 54 sec	13 mins, 7 sec
3 GB	68.96 mins	7 mins, 25 sec	17 mins, 3 sec	28 mins, 1 sec

same with 30, 60, and 120 nodes. This is because the number of map tasks for 500 MB is 16 which can be executed using a maximum of 16 nodes. Similarly, we observe in figure 5.3 that the construction time for a human genome of size 1GB remains same with 60 and 120 nodes. This is because the number of map tasks for 1GB is 52 which can be executed using a maximum of 52 nodes.

Figure 5.5, figure 5.6, and figure 5.7 show the evaluation with 30, 60, and 120 nodes are shown. These graphs substantiate the scalability of the MapReduce algorithm and also show the exponential increase in the construction time with the size of genome and this behavior is consistent with 30, 60, and 120 nodes.

Figure 5.8 shows a good comparison of constructing the suffix tree with 30, 60, and 120 nodes.

As shown in figure 5.4, the construction time decereases with increase in number of nodes

Table 5.4: Evaluation with 120 nodes (Time in minutes)

Genome Size	Construction Time	Best Map Time	Average Map Time	Worst Map Time
500 MB	6.28 mins	4 mins, 4 sec	4 mins, 52 sec	5 mins, 49 sec
1 GB	6.58 mins	2 mins, 34 sec	4 mins, 21 sec	6 mins, 9 sec
2 GB	12.5 mins	3 mins, 47 sec	6 mins, 48 sec	12 mins, 8 sec
3 GB	39.31 mins	7 mins, 33 sec	16 mins, 57 sec	30 mins, 51 sec

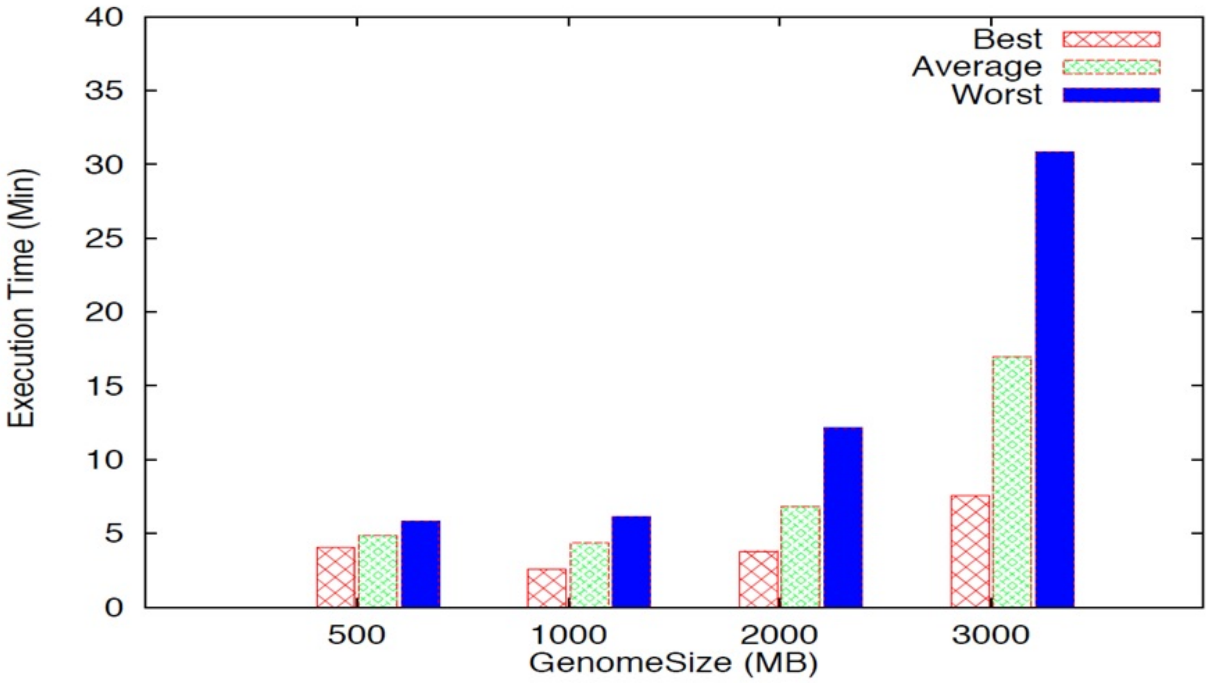


Figure 5.1: Average, Worst, and Best map task times for different sizes of genome data

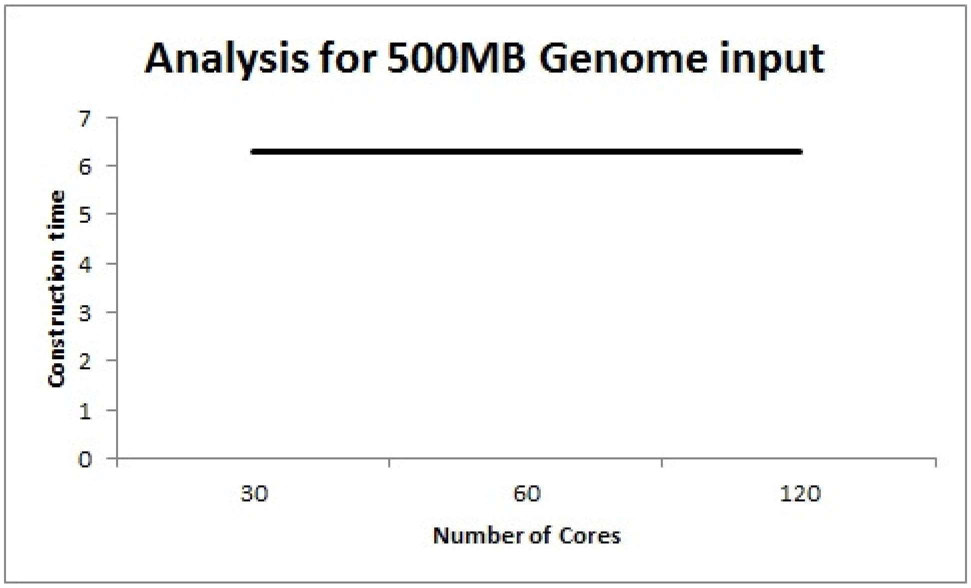


Figure 5.2: Analysis for 500MB Genome Input

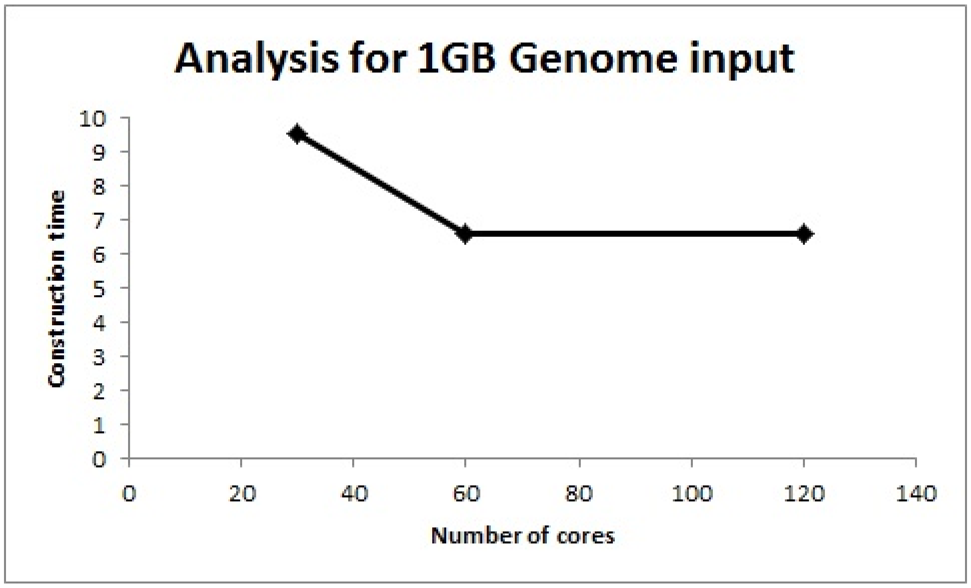


Figure 5.3: Analysis for 1GB Genome Input

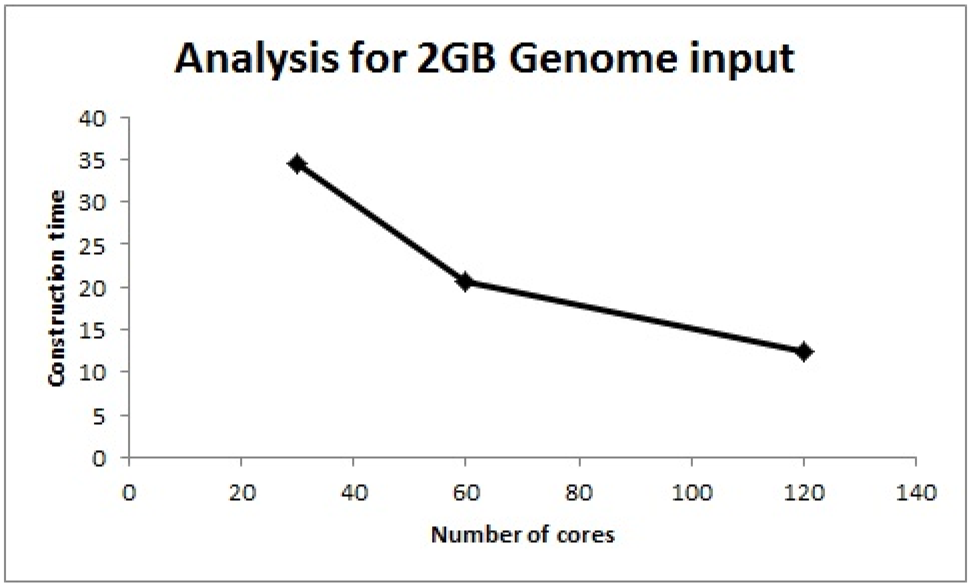


Figure 5.4: Analysis for 2GB Genome Input

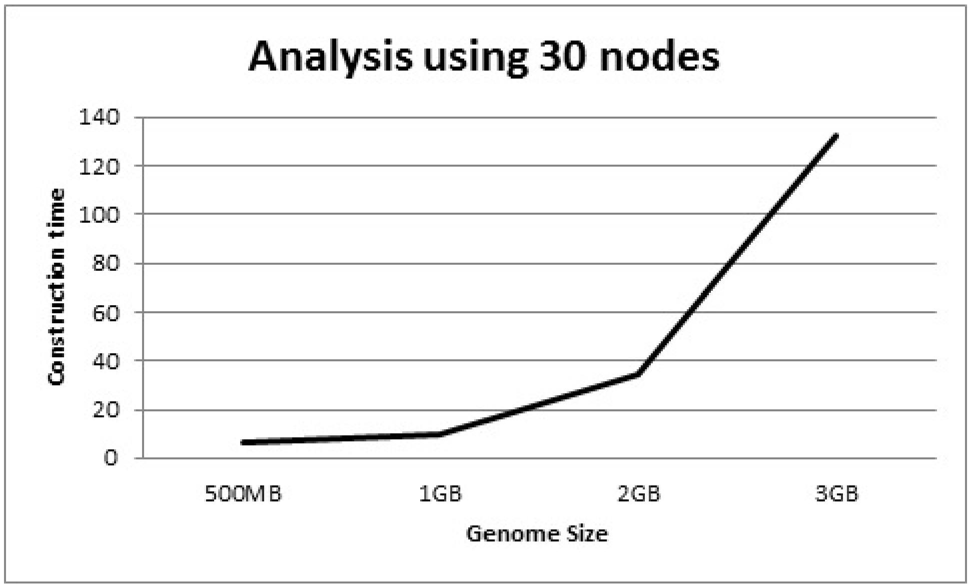


Figure 5.5: Analysis using 30 nodes

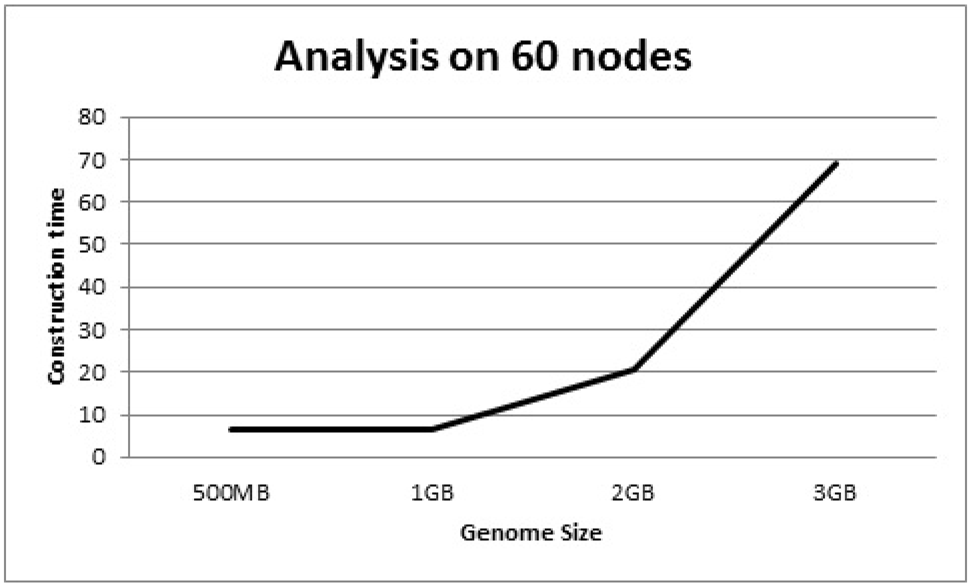


Figure 5.6: Analysis on 60 nodes

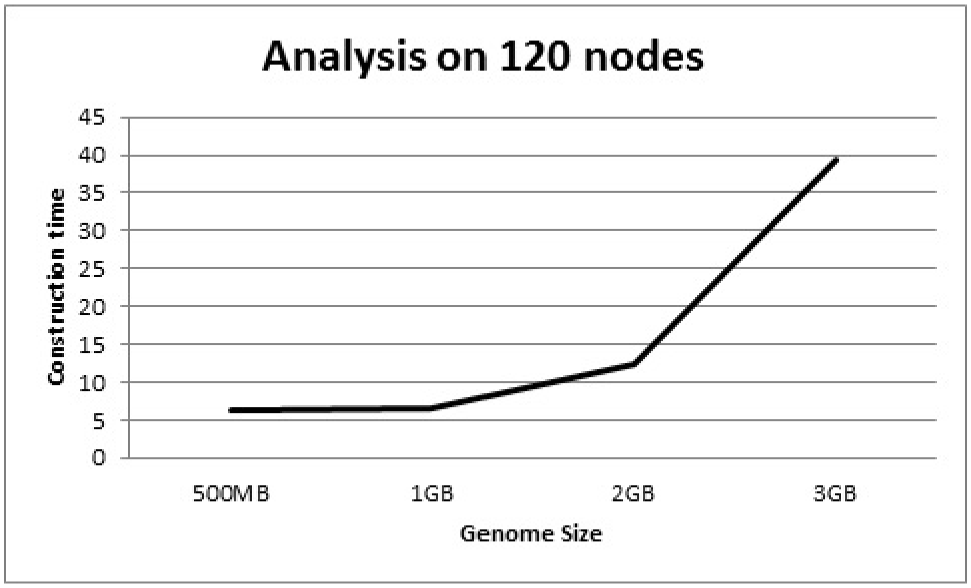


Figure 5.7: Analysis on 120 nodes

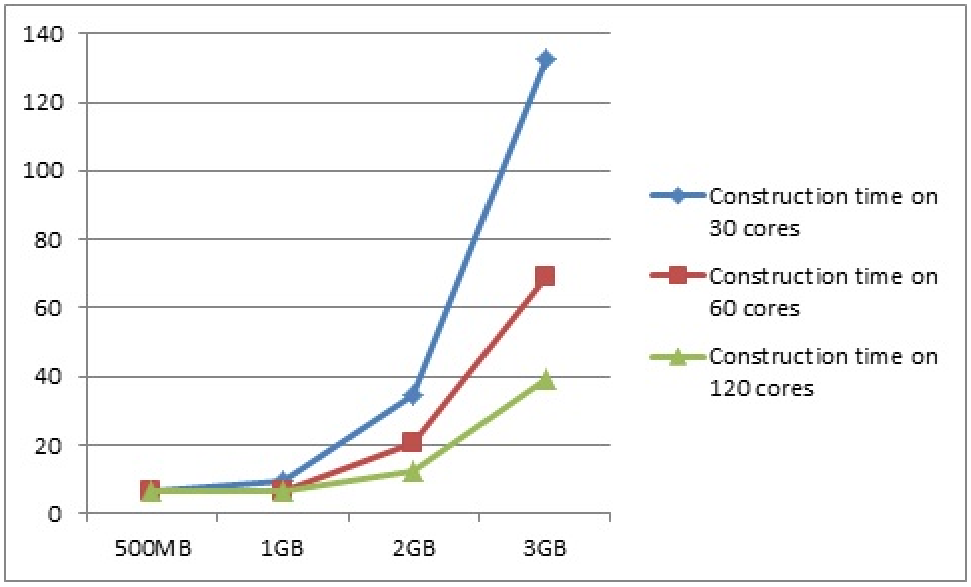


Figure 5.8: Construction Time on 30,60 and 120 cores

CHAPTER 6
CONCLUSION AND FUTURE WORK

Read mapping is one of the many bioinformatics application which is used to align millions of short reads against reference genome. Many of the contemporary alignment techniques are time consuming, memory intensive and cannot be easily scaled to larger genomes. Suffix tree is a popular data structure which can be used to overcome the demerits of other alignment techniques. However, constructing the suffix tree is highly memory intensive and time consuming. In this thesis, a MapReduce based parallel construction of the suffix tree is proposed.

6.1 Conclusions

Following are the conclusions that can be drawn constructing the suffix tree using MapReduce algorithm:

- As shown in figure 5.5, figure 5.6, and figure 5.7 the suffix tree construction time on a fixed number of nodes increases exponentially with the increase in size of human genome data.
- As shown in figure 5.8, the construction time for a human genome of a particular size decreases with the number of nodes.
- Since the number of map tasks for genome data of different size remains same, the best map time, the average map time, and the worst map time remain same if the number of nodes is more than the number of map tasks.
- The scalability of the algorithm improves with the increase in number of nodes.
- The construction of suffix tree can completed in less than the allocated number of nodes if the number of map tasks is less than the allocated number of tasks.

6.2 Future Work

- The distributed suffix tree construction algorithm proposed in this thesis is based on Ukkonen's suffix tree construction algorithm. Similar to Ukkonens algorithm, the distributed algorithm in this thesis is also an in-memory algorithm where the subtrees constructed will be written to primary memory. The Algorithms, which belong to other categories like semi-disk based and out-of-core technique, can be constructed in parallel using hadoop to check for memory efficient algorithm.
- The maximum size of the human genome considered for analysis in this thesis is 3GB. The distributed algorithm can be analysed for higher data size to check if the scalability factor of the algorithm remains consistent.
- The suffix tree constructed by the MapReduce algorithm can be used for read mapping and the time taken and memory consumed can be compared with other contemporary alignment techniques.
- Currently the algorithm is implemented in java. The algorithm can be implemented in C++ to get better performance.

REFERENCES

[1] http://en.wikipedia.org/wiki/Big_data.

[2] http://www-01.ibm.com/software/data/bigdata/.

[3] http://www.dnasequencing.org/.

[4] http://www.1000genomes.org/about.

[5] http://en.wikipedia.org/wiki/Suffix_tree.

[6] http://blast.ncbi.nlm.nih.gov/http://en.wikipedia.org/wiki/BLAST.

[7] http://soap.genomics.org.cn.

[8] http://www.google.com/imgres?imgurl=http://www.astrochem.org/sci_img/dna.jpg&imgrefurl=http://www.astrochem.org/sci/Nucleobases.php&h=1046&w=800&sz=266&tbnid=TGomYMQjmEs3YM:&tbnh=103&tbnw=79&zoom=1&usg=___VwTr5V0ESiLA1d5PwQp6tYB1es=&docid=WPu1jh_U7BgfuM&sa=X&ei=OrP1UZidOofM9gSFxoEw&ved=0CGEQ9QEwDw&dur=526.

[9] http://programmerspatch.blogspot.com/2013/02/ukkonens-suffix-tree-algorithm.html.

[10] Ultrafast and memory-efficient alignment of short DNA sequences to the human genome.*Genome Biology*, 10:R25–10, 2009.

[11] RM Casey. Blast sequences aid in genomics and proteomics. *Business Intelligence Network*, 2005.

[12] Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang. Soap: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008.

[13] Rohith K. Menon, Goutham P. Bhat, and Michael C. Schatz. Rapid parallel genome indexing with mapreduce. In *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce '11, pages 51–58, 2011.

[14] Michael C. Schatz. Cloudburst. *Bioinformatics*, 25(11):1363–1369, June 2009.

[15] Tom White. *Hadoop: The Definitive Guide.* 2009.

[16] Ukkonen, E. *On-line Construction of Suffix-trees.* 1993.

VITA

Umesh Chandra Satish was born in Bangalore, India in 1986 to Satish and Jayashree. He obtained his Bachelor of Engineering in Computer Science and Engineering from Visvesvaraya Technological University, Belgaum, India in 2008. He worked as a Systems Engineer at Siemens Information Systems Limited at Bangalore, India. In Spring 2011, he came to the US in pursuit of a Masters degree. He completed her Masters in Systems Science at Louisiana State University, Baton Rouge in December 2013.