2014

Performance Improvement of Distributed Computing Framework and Scientific Big Data Analysis

Praveenkumar Kondikoppa
Louisiana State University and Agricultural and Mechanical College

PERFORMANCE IMPROVEMENT OF DISTRIBUTED COMPUTING FRAMEWORK AND SCIENTIFIC BIG DATA ANALYSIS

A Dissertation

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
Doctor of Philosophy

in

The Department of Electrical Engineering and
Computer Science

by
Praveenkumar Kondikoppa
B.E., Visvesvaraya Technological University, 2004
August, 2014

To my late father, for discipline
To my mother, for strength and courage

Acknowledgments

Many people contributed to the successful completion of my research. This journey would not have been possible without the encouragement, support and guidance of very important people in my life. I hope they share this tremendous joy and achievement with me.

I am deeply thankful to my advisor Dr. Seung-Jong Jay Park for giving me the freedom, flexibility and time to develop as a researcher. I thank you for introducing me to Big data and genome assembly. Most importantly I thank you for making me a better person. Committee members Dr. Rahul Shah, Dr. Bijaya B. Karki and Dr. Charles Delzell provided valued advice.

I am indebted to Manish Patil for helping me throughout my graduate study starting from course work, qualifying exam, general exam and collaborating with me for research papers.

I am so grateful to my mentors and friends, Prashanth Hallur, Asha Latha mam, Bhavana Shankar, Prakash Mukre, Shrishail Nagathan and Kavitha Ponnuswamy for the guidance and encouragement.

Special thanks my under graduate friends, Sachin Desai, Vinayak Nadakarni, Santosh Umari, Vinay Patil, Tushar Sashittal and Santosh Kurahatti for supporting me during my initial travel to US.

I will always remember the fun times I had with my friends from Baton Rouge, Priyanka Rotti, Arpitha Prakash, Madhavi Divakar, Umesh Chandra, Sunada Chakravarthy, Sai Sashankh Rao, Pratap Rao, Pradeep Pai, Abhishek Bharad and Narendra NA.

Friendship is one of the things that make graduate school tolerable, Rajesh, Naveen Jagadish and Aditya Kohli who kept me sane and humorous in the last five years.

I appreciate my cousins Naveen Shettar, Anand Revannavar and all my nephews for cheering me up all the time.

I am grateful to my brother Basavaraj Kondikoppa K and sister Ratna for their blind support in all my endeavors.

Finally I thank you Lord for giving me the opportunity to pursue Ph.D.

Table of Contents

List of Tables

List of Figures

Abstract

Analysis of Big data to gain better insights has been the focus of researchers in the recent past. Traditional desktop computers or database management systems may not be suitable for efficient and timely analysis, due to the requirement of massive parallel processing. Distributed computing frameworks are being explored as a viable solution. For example, Google proposed MapReduce, which is becoming a de facto computing architecture for Big data solutions. However, scheduling in MapReduce is coarse grained and remains as a challenge for improvement.

Related with MapReduce scheduler when configured over distributed clusters, we identify two issues: data locality disruption and random assignment of non-local map tasks. We propose a network aware scheduler to extend the existing rack awareness. The tasks are scheduled in the order of node, rack and any other rack within the same cluster to achieve cluster level data locality. The issue of random assignment non-local map tasks is handled by enhancing the scheduler to consider the network parameters, such as delay, bandwidth and packet loss between remote clusters.

As part of Big data analysis at computational biology, we consider two major data intensive applications: indexing genome sequences and de Novo assembly. Both of these applications deal with the massive amount data generated from DNA sequencers. We developed a scalable algorithm to construct sub-trees of a suffix tree in parallel to address huge memory requirements needed for indexing the human genome.

For the de Novo assembly, we propose Parallel Giraph based Assembler (PGA) to address the challenges associated with the assembly of large genomes over com-

modity hardware. PGA uses the de Bruijn graph to represent the data generated from sequencers. Huge memory demands and performance expectations are addressed by developing parallel algorithms based on the distributed graph-processing framework, Apache Giraph.

Chapter 1
Introduction

1.1 Big Data

The volume of data produced from both academic research and industry is growing at an enormous rate. The term ”Big data” is widely used to characterize the complexity and volumes of data sets generated from scientific research, social networks, internet search indexing, high throughput sequence data from a genome, and many other application domains. Researchers have focussed on developing computational frameworks to address the compute and data-intensive computations. In the next several sections I introduce existing distributed computing frameworks widely used for Big data analytics.

1.2 Compute Intensive Computing

High performance computing (HPC) clusters and computational grids [18] [10] target compute-intensive applications. HPC clusters have been in use for large scale scientific computing for years. The primary goal of HPC is to use more processors to reduce the end to end computation time. Each node in the cluster can be a single-processor or multiprocessor equipped with large memory, I/O devices and local storage. Furthermore, each node has access to shared file system interconnected with high speed interconnects such as 10GbE or infiniband.

Batch computing is the simplest form of applications executed over HPC, in which, a large job is split into multiple batches, which are independently processed over multiple processors. For example, Autodock [6] is a widely used software for molecular docking to predict the binding orientation of small molecule to their protein targets in order to predict the affinity and activity of the small molecule. In the parallel version, the protein targets are batched into subsets and executed in parallel over multiple nodes. In this case the batch computing is suitable since docking does not require communication between processors.

However, not all parallel computations can be expressed as batch processing. Many of the parallel algorithms require coordination between the computing processors and the communication between processors is achieved through message passing interface (MPI). Although, MPI gives great flexibility to the programmer, but requires explicit handling of the data flow and fault tolerance. Furthermore, scalability of HPC systems is an issue and shown to be poor with linear speedup. Many of the compute intensive problems can be solved with HPC, however data intensive applications require an efficient, easily scalable and fault tolerant distributed computing framework.

1.3 Data Intensive Computing

Cloud computing provides a unique solution to storage and computing power for big data analysis. Emerging infrastructure as a service clouds such as Amazon EC2 [2], Openstack [15], Rackspace [17], OpenCirrus Eucalyptus [34] and IBMs Blue Cloud [11] are big leap towards providing computing power over the internet. On demand provisioning, ease of use, seamless computing, storage and the fact that the computing power is provided through low cost commodity clusters draws huge attention both from academia and industry. Many of these commercial clouds provide unlimited computing power. In addition to those commercial clouds, many universities and research laboratories have deployed their own private clouds for research and education. Cloud computing services can be classified into three major categories: 1) Infrastructure as a service (IaaS) 2) Software as a service (Saas) and 3) Platform as a service (Paas).

Iaas

It is a model for providing hardware resources as service, the resources can be computing units in the form of virtual machines, storage, servers and networking components. Any request is serviced on demand, users can set up any software of their choice on the provisioned resources. The characteristics and components of IaaS include i) Utility computing service and billing model ii) Automation of administrative tasks, and iii) Dynamic scaling and Policy-based services.

Saas

In this model software hosted by a vender is provided as service to the users. This model is gaining importance as a common delivery model for providing software. SaaS has number of benefits such as, easier administration, automatic updates and software version management.

Paas

It is a service model of cloud computing which provides computing platform and solution stack as a service. In this model the vendor not only provides the underlying infrastructure resources but also the application development platform. This platform includes the automation to deploy, test and iterate applications. Operating systems, databases, middleware and up-to-date tools and services are all provided by the PaaS vendor, so that the tasks requiring manual tasks such as configuring, optimizing and continuously updating your environment are handled on users behalf.

MapReduce [27] paradigm is a distributed computing framework for data intensive applications. It is inspired by the Map and Reduce constructs of the functional programming model. The program is expressed as Map and Reduce constructs. Input and output are expressed in terms of (key,value) pairs. Map takes (key, value) as input and produces intermediate (key1 value1) output pairs. Reducer takes (key1,value1) as input and produces (key2,value2) as output. All the Map functions run in concurrent. The output of Map is sent to the Reduce function as input. Reducer will sort the input key-value pairs according to the key and combine the value of key-value pairs into a list which have the same key, at last, the reducers which are also run in parallel generate the output key-value pairs.

The MapReduce framework abstracts programmers from the complexities of input data distribution, parallelization, fault tolerance and synchronization the framework inherently handles these functionalities. The open source implementation of MapReduce framework consists of computing environment called Hadoop and distributed filesystem called Hadoop distributed filesystem (HDFS). Hadoop acts like runtime environment for the Map and Reduce functions provided by programmers. HDFS is designed to provide replication, integrity of the data and fault tolerance. The architecture of Hadoop has master services namely Jobtracker and Namenode and client services namely Tasktrackers and Datanodes. The Jobtracker receives the job submitted by the user and splits it into map and reduce tasks. It assigns tasks to Tasktrackers, monitor the status and when all the tasks are complete it reports back to the user. Tasktrackers are configured with fixed number of map and reduce slots which determine the number of tasks to run concurrently. Cloud computing coupled with MapReduce programming model have made huge strides in big data analysis. MapReduce is extensively used on compute clouds at Amazon, Facebook and Yahoo. This has led many of the organizations to set up MapReduce based private cluster for internal data processing.

1.4 Scientific Big Data Analysis: Genome Assembly

Mainly there are two types of genome assembly applications (1) Read mapping or Sequence alignment (2) De Novo Genome Assembly. Both involve processing huge data sets produced from genome sequencers.

DNA sequencing is the basis for many of bioinformatics applications such as SNP

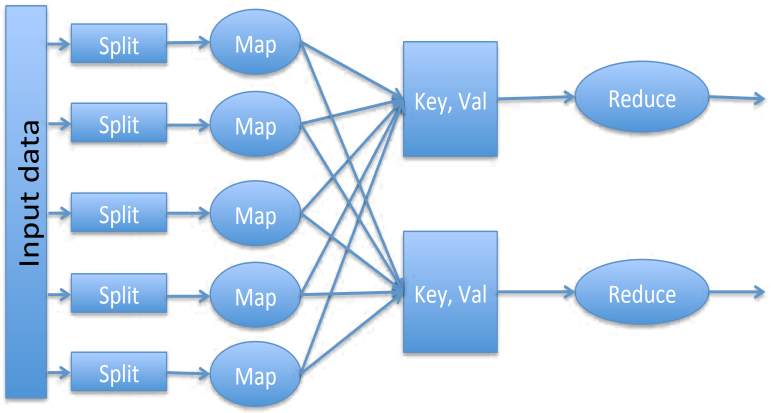


Figure 1.1: MapReduce programming model

discovery, comparative genomics, metagenomics and personal medicine. Large collection of short reads produced by sequencing technologies form a genomic dataset needed for these applications. Recent technological breakthroughs in Next Generation Sequencing (NGS) have resulted in affordable, low running costs and high throughput sequencers. However, these methods produce short sequences, for example, the Illumina-Solexa system can generate 50 million sequences of length 30-50 nucleotides [35]. The Roche-454 system can generate 400, 000 sequences of length 250-500 nucleotides in a 7.5 hour run [47]. NGS is a evolving technology and it is speculated that the sequencing costs will be reduced to as low as $1000 [50]. Low cost of producing short sequencing will trigger the use of such systems even in small laboratories. The computational demands for processing NGS data are tremendous and far exceed current capabilities. In fact, without substantial advances in high-performance, scalable algorithms, very little progress would be made to extract knowledge from such a rich set of data. Therefore, there is a need to design powerful algorithms and systems which can efficiently handle the computational challenges posed by NGSs.

1.4.1 Read Mapping

Read mapping also referred as sequence alignment an important aspect of DNA analysis. DNA is a sequence of four possible nucleotides or base pairs (bp): Adenine (A), Cytosine (C), Guanine (G) and Thymine (T). The subset of this sequence which determines the functionality of protein molecules is called genes. Even a single nucleotide difference in a gene alters the protein molecule functionality between the individuals. The purpose of sequence alignment is to compare the genes between the individuals and determine the difference in characteristics. The earliest of alignment tools is Basic Local Alignment Search Tool (BLAST), It makes use of an algorithmic technique called seed-and-extend. BLAST first indexes the short sequence to be searched in the reference genome. These short sequences are called query sequences and the resulting strings, after indexing, are called words. BLAST then scans the reference genome sequence to find the similar match of the indexed words. Tools which are based on indexing methods are SOAP [39] and BowTie [37]. These alignment tools specifically designed for reads generated from illumina-solexa machines. Not many tools are available for mapping 454 reads, because of the volumes of short reads generated, thus there is need to develop faster sequence mapping tools which can match the speed of NGS sequencers.

1.4.2 De Novo Genome Assembly

Construction of DNA sequence of an organism from short sequences produced from sequencing technique is referred as de novo genome assembly.

Sequecing

Extracting the DNA from an organism and generating fragments of the DNA is called sequencing. The sequencing field is dominated by Sanger sequencing methods until recently, in which the length of short sequences is around 500-1000bp. The modern sequencing methods known as next generation sequencing technologies produce smaller length sequence of length 30-100bp.

Short Reads

The collection of short sequence which are produced after subjecting the DNA to sequencing method. Genome sequencing projects commonly use whole genome shotgun sequencing (WGSS) approach to sequence an entire genome.

Assembly

Assembly of shot reads, in other words putting together the fragments randomly extracted from the sample to form a set of contiguous sequences called contigs, which represent the DNA in the sample. Figure 1.2 shows the overview of assembly process[1]

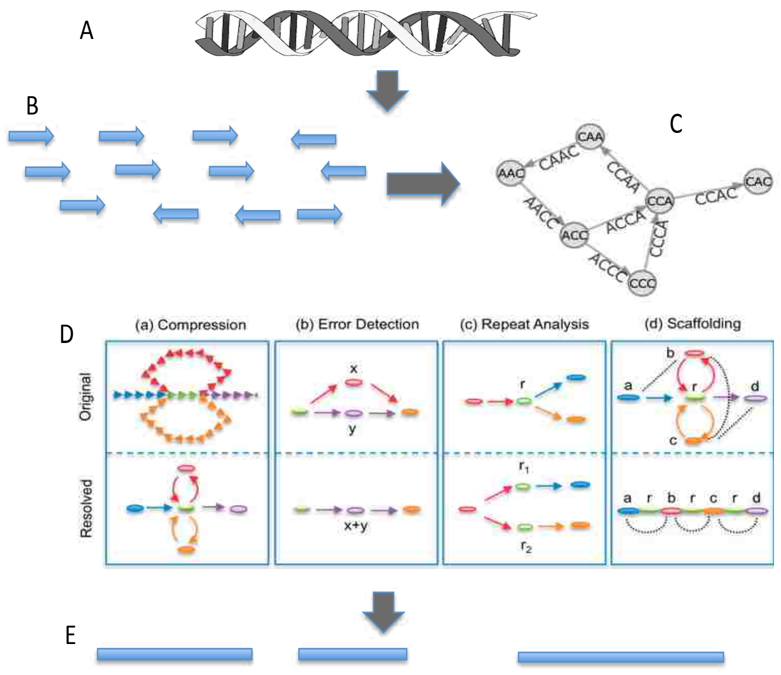


Figure 1.2: Overview of genome assembly process (A) Genomic data (B) Short reads produced from next generation sequencing methods (C) De Bruin graph representation of the unique k-mers from short reads (D) Series of graph simplification process which are applied repetitively (E) After the scaffolding phase, relatively longer sequences are produced these are called contigs

[1]Part of the image is reproduced from Contrail [8]

Chapter 2
Network-Aware MapReduce Scheduler

2.1 Introduction

Large scale scientific and engineering applications generate vast amount of data. For example, the Large Hadron Collider project generates peta bytes of data which need to be processed, analyzed and stored. Cloud computing coupled with MapReduce programming paradigm has made huge strides in big data analysis. Hadoop, which is an open source implementation of MapReduce is extensively used on compute clouds at Amazon, Yahoo and Rackspace. Many of these commercial clouds provide unlimited computing power. In addition to those commercial clouds, many universities and research laboratories have deployed their own private clouds for research and education.

To maximize the utilization of heterogenous public and private clouds, researchers want to launch their jobs over federated clouds at the same time. For example at Louisiana State University, researchers can make use of different cloud resources: LONI [13], Futuregrid [10] and Xsede [18]. We focus on how to aggregate cloud resources from private and public clusters and run MapReduce jobs more efficiently.

Conventional MapReduce(Hadoop) schedulers are optimized to operate over single cluster which connects computing nodes within local networks. However with considerable administrative efforts MapReduce can be deployed over distributed clusters. There are several issues on performance of MapReduce over distributed clusters. For example, network latency and bandwidth between a master node and slave nodes reduce the performance of MapReduce because MapReduce configuration requires a direct communication between a master node and slave nodes. Additionally, scheduling tasks among slave nodes which are scattered over remote clusters becomes an important factor on performance because of data locality.

There are several ways of deploying MapReduce Hadoop over distributed clusters. First approach is to set up independent Hadoop clusters at each site having a centralized job scheduler to manage job submission. The second approach is to aggregate the underlying physical clusters as a single virtual cluster and run MapReduce tasks on top of this virtual cluster. And the third approach is to make MapReduce framework work directly with multiple clusters without additional virtual clusters. We deployed global MapReduce over federated clusters to utilize the computing resources effectively. However, deploying MapReduce framework directly on multiple cluster degrades the performance because of failing to exploit data locality. Our research provides a method to add network awareness to global MapReduce so that a scheduler has the information about data locations to launch map tasks on nodes with the data. We refer this feature as network aware scheduler.

2.1.1 Related Works

Cardos et al. [25] suggest different architectures to set up MapReduce framework when source data and computation are distributed. They recommend three architectures, Local MapReduce where in data is moved to centralized cluster to perform computations. Global MapReduce where MapReduce cluster is established with nodes from all the clusters and Distributed MapReduce where in small independent MapReduce clusters similar to architectural overlay over the distributed computing nodes. However, their suggestions are different configurations and fail to address data locality concerns.

Yuano Luo et al. [40] introduce hierarchical MapReduce by organizing the resources into two layers: a top layer consisting of job scheduler and workload manager, and a bottom layer consisting of distributed clusters running local MapReduce. When a job is submitted, it is split into sub jobs, which are assigned to distributed clusters. A Jobtracker at each of the clusters reports the job progress and sends result back to a global controller where reduce tasks take place. The disadvantage of this approach is that it is suitable for map-intensive or map-mostly type of applications. If intermediate data generated from the map phase are huge and need to be transferred to a reducer then data transfer becomes bottleneck for the performance of Hadoop. Majority of the data intensive applications involve significant reduce phase computations. Also users need to add additional programming construct to perform global reduce at the global controller.

2.2 Scheduling in Hadoop

There exist different job schedulers for Hadoop. The mainly used job schedulers are FIFO and FAIR schedulers. We discuss each in the below sections.

2.2.1 FIFO Scheduler

Default scheduling in Hadoop is through FIFO scheduler. When a slave node with an empty map slot sends a heartbeat packet to the Jobtracker, scheduler checks the head of the job queue to get the unassigned map tasks. If the job has tasks whose input split is located on the slave node then task is assigned to the Tasktracker. If scheduler does not find local map task then it assigns only one non local map task to this Tasktracker.

2.2.2 FAIR Scheduler

FAIR scheduling ensures that every job gets an equal share of resources over time. If there is single job running then entire cluster is allocated for the job. If new jobs are submitted then task slots which free up are allocated to the newly submitted jobs, so that each job gets roughly equal share of cpu time. Small jobs get finished within reasonable time and long jobs are not starved. FAIR scheduler organizes jobs into pools and inside pool FIFO or FAIR queuing of the jobs can be configured.

2.2.3 Global MapReduce over Federated Clusters

MapReduce Hadoop cannot be easily deployed over distributed clusters of different administrative domains to form single MapReduce cluster, since master node services require direct communication between slave node services. If the clusters belonging to different administrative domains need to be federated, then a considerable amount of administrative effort is needed to make internal nodes of clusters to communicate with each other. We address this issue of internal nodes accessible from outside by introducing a virtualization on top of all the physical clusters. Cloud computing software like Eucalyptus [34] or Openstack [15] is installed on each of the underlying physical clusters. Computing units are provided as virtual machines from any of the distributed clusters. A virtual layer unifies the resources provided by all the distributed clusters into single infrastructure layer of virtual machines. MapReduce cluster is set up on top of these virtual machines where in a single Jobtracker takes the responsibility of scheduling the jobs and managing the Tasktrackers as shown in figure 2.2. The advantage of this approach is that, it is easy to manage distributed clusters through cloud computing software. However, the critical aspect in this approach is to make a scheduler be topology-aware in order to accomplish data local computation of map tasks. The proposed work focuses on improving data locality while scheduling map tasks for global MapReduce so that all types of data intensive applications such as, map-only, map-mostly and map-reduce application types are supported. In the below sections, we describe data locality issue and propose network aware scheduling to make global MapReduce scheduler data aware.

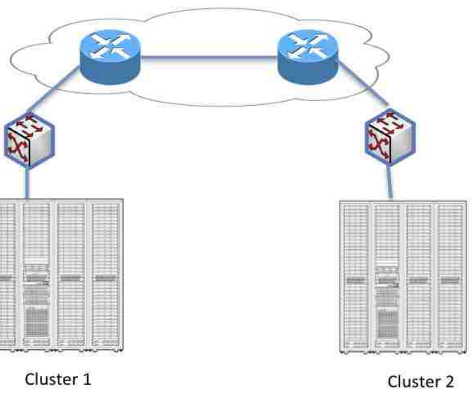


Figure 2.1: Federated Clusters

2.2.4 Case for Data Locality over Federated Clusters

Unlike HPC systems where computation and file system are decoupled, in MapReduce Hadoop filesystem (Datanode) and computation (Tasktracker) are co-located on a node. When data is transferred to HDFS it is replicated to datanodes so that Tasktracker running on that node will have data blocks prior to execution of a task. In the current implementation of MapReduce Hadooop, data locality is achieved by replicating the data at three levels of cache. At the first level, a data split is stored on a node. At the second level, a data split is saved on a node in the same rack and at the third level, a data split is stored off-rack. To achieve data locality, map tasks are scheduled on nodes which already have the data blocks (this constitutes node local). If scheduler cannot find such a node then tasks are scheduled on nodes which can fetch data blocks from any node in the same rack (this constitutes rack local). If both situations are not met then the map task is scheduled on any tasktracker requesting for a task. Thus MapReduce scheduling algorithms exploiting data locality are based on the assumption of multiple nodes in a rack, which are connected by an aggregate switch. Figure 2.1 shows the typical architecture for federated clusters which may be connected by regular internet or by dedicated link. Furthermore, if virtualization is added on top of these federated clusters to provide virtual machines as computing units, then Hadoop's assumption of racks connected through aggregate switch cannot be extended to achieve data locality. To get maximum performance of Hadoop, it is critical to configure Hadoop so that it knows the topology of the entire network while making scheduling decisions.

2.3 Network Aware Scheduling

The goal of task scheduling in Hadoop is move computation towards data. If it can not meet this objective then a task is scheduled on a node which is requesting for a task, this causes data to be transferred to compute node for processing. When resources are provisioned by distributed clusters, moving the data across the network causes the degradation in Hadoop's performance.

In case of distributed clusters, the Tasktrackers from any of underlying cluster might request for a map task. The technique to make Hadoop scheduler aware of network topology is to extend the rack aware feature of the existing Hadoop scheduler to provide one more level of caching. An administrator controlled script will hold the information about which cluster the tasktracker is associated with. In our implementation of global MapReduce architecture, network topology script has information about virtual machines and physical location of the cluster from which they are provisioned. We use Neuca [34] enabled Eucalyptus cloud computing software to provide virtual resources from the distributed clusters. The locations of the virtual machines are organized as /cluster$_N$/rack$_N$/vm$_N$. Where cluster$_N$ denotes the physical which cluster, rack$_N$ denotes the rackid and the vm$_N$ indicates the hostname for the virtual machine.

We enable delay scheduling [20] to take maximum use of data locality while scheduling a task. When head of the queue task doesn't find a compute node with data then scheduling of the task is delayed for specified delay. If any of the compute nodes become free with a data split corresponding the job being processed then scheduler assigns a map task to requesting Tasktracker. Duration for which a head of the queue map task is to be delayed is based on the average length of the map tasks for a job hence requires

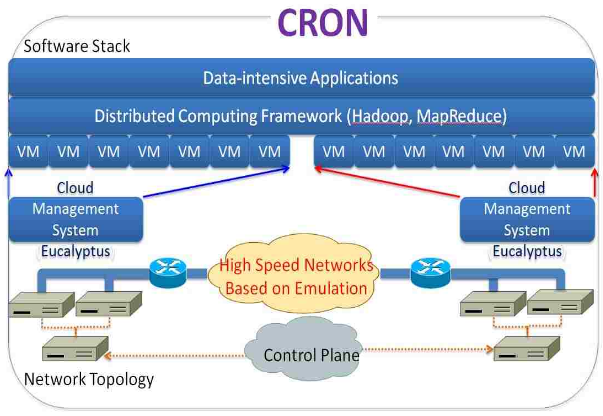

CRON
Software Stack
Data-intensive Applications
Distributed Computing Framework (Hadoop, MapReduce)
VM VM VM VM VM VM VM VM VM VM VM VM VM VM
Cloud Management System Eucalyptus
Cloud Management System Eucalyptus
High Speed Networks Based on Emulation
Control Plane
Network Topology

Figure 2.2: Experimental Setup

careful tuning.

2.4 Experimental Setup

In this section we describe the environment used for our experiments. We perform experiments over cyberinfrastructure of reconfigurable optical networking environment (CRON) [19], an Emulab based testbed that provides multiple virtual testbeds operating up to 10Gbps bandwidth. It consists of two main components: (i) hardware (H/W) components, including a switch, optical fibers, network emulators, and the workstations required to physicallycompose optical paths or function at the ends of these paths; and (ii) software (S/W) components, creating an automatic configuration server that will integrate all the H/W components to create virtual network environments based on the users requirements. All components are connected with 100/1000 Mbps Ethernet links for control. Each workstation is connected with 10 Gbps optical fibers for data movement.

We use Eucalyptus as cloud computing software, we setup two NEuca-patched [14] Eucalyptus Clouds to construct the distributed cloud scenario. The NEuca patch attaches additional exclusive virtual NICs to a VM for application data transmission. We have two computing nodes in each Cloud and each computing node accommodates three

Table 2.1: Jobs with Varying Task Lengths

Task	Average Execution Time
Map	5 sec
Map	8 sec
Map	14 sec

VMs. Then, all VMs connect to each other through the additional NICs to form a virtual cluster. We deploy network aware Hadoop over this virtual cluster as shown in figure 2.2 We avoid multiplexing VMs on a physical machine for the better and stabler execution. Each VM is allocated with 2 physical CPU cores, 2 GB of physical memory, and 10 GB of local hard disk space. Regarding the NIC, the VirtIO interface is adopted to serve the virtual NIC so that the transmission rate is as high as the Hypervisor can provide.

2.4.1 Experiments

2.4.1.1 Experiments with Native Hadoop

In this set of experiments we run Hadoop over distributed cluster without network awareness. One of the node will be a master node running Jobtracker and Namenode services. Remaining nodes from both clusters run slave services namely Tasktrackers and Datanodes. We use the word count application for evaluation. Input file size for word count application is varied so that average execution time for map task differs. Multiple jobs consisting of tasks with different execution times as shown in the table 2.1 are submitted.

2.4.1.2 Experiments with Network Aware Hadoop

In this set of experiments, Hadoop with network awareness is configured over the federated cluster. Hadoop jobs are created by using files of different sizes. A single instance of Jobtracker will manage the scheduling over all the clusters. Tasktrackers from different clusters request for map tasks as and when they get map slots freed. Jobtracker

Table 2.2: Evaluation Environment

Nodes	Quantity	Hardware and Hadoop
Master Node	1	2 CPU core, 2 GB RAM, 10Gbps NIC Jobtracker and Namenode
Slave Nodes	12	2 CPU core, 2 GB RAM 10Gbps NIC Tasktracker and Datanode Hadoop-0.20.203.0 2 Map and 2 Reduce Tasks

uses the cluster awareness to schedule tasks on these Tasktrackers there by improving the data locality. We perform experiments with varying inter-cloud bandwidth and delay and measure the performance of Hadoop with network awareness.

2.5 Results

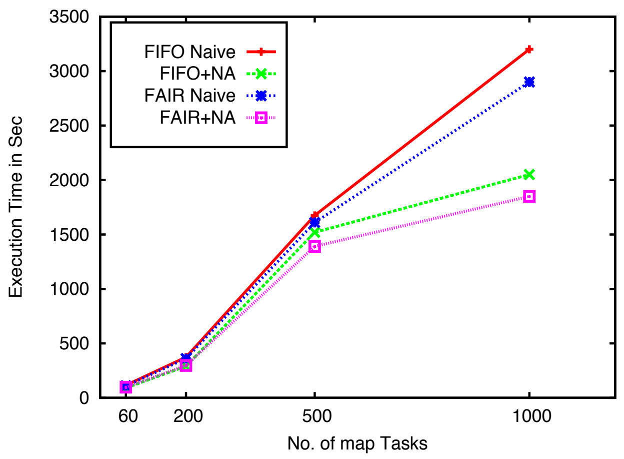


Figure 2.3: Execution times for Different schedulers for 1Gbps, 30ms inter cloud bandwidth and delay

We compare the execution times of Hadoop native scheduling with network-aware Hadoop for varying delay values. Figure 2.3 shows the execution times for varying number map tasks, FIFO scheduler with network awareness identified as FIFO+NA, shows the reduction in execution time of about 12 % on average and maximum reduction of 15 % when number of map tasks is higher compared to Hadoop Naive scheduler identified as FIFO Naive. FAIR scheduler with network awareness identified as FAIR+NA also shows the similar results, FAIR scheduler in general takes less execution time compared to FIFO, The reason being effective utilization of the cluster. Network aware Hadoop shows maximum performance improvement for map tasks between 500 to 1000. As the map task number increases, the data splits associated with the map tasks will be spread over the distributed clusters. When Jobtracker has to schedule a map task over a tasktracker, it check if the tasktracker is located in the same cluster as the datanode having the data to process. If this condition is met then only task will be assigned to the tasktracker otherwise it will be delayed for certain delay in seconds specified for the delay scheduling.

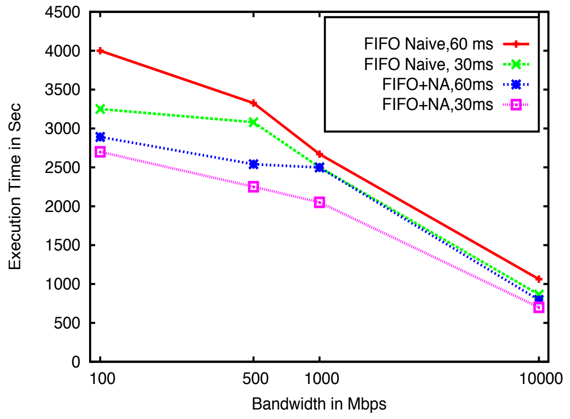


Figure 2.4: Execution times for varying inter-cloud bandwidth

Bandwidth is a significant factor for federated clusters. The performance of Hadoop greatly depends on the bandwidth between the clusters . Figure 2.4 is for varying intercloud bandwidth with fixed delay. We emulate the varying bandwidth cases on our experimental set up using dummynet [9]. At 100Mbps bandwidth between the clouds, there is significant difference in execution times for processing 10GB data. If the tasks are not executed either with node local or rack local then large amounts of data will be moved from one cluster to another cluster for map task execution. With 100Mbps of bandwidth there will be less available bandwidth between nodes. As the bandwidth provisioned between the clusters is increased overall execution time decreases. However, for processing tera bytes of data there should be sufficient bandwidth provisioned between the clusters. We performed experiments with different network conditions, in all the cases significant reduction in execution time is observed between native Hadoop scheduler and network-aware scheduler as the number of map tasks increased. It is evident that for higher inter cloud bandwidth the performance of Hadoop with network aware is greatly increased compared to native Hadoop scheduling.

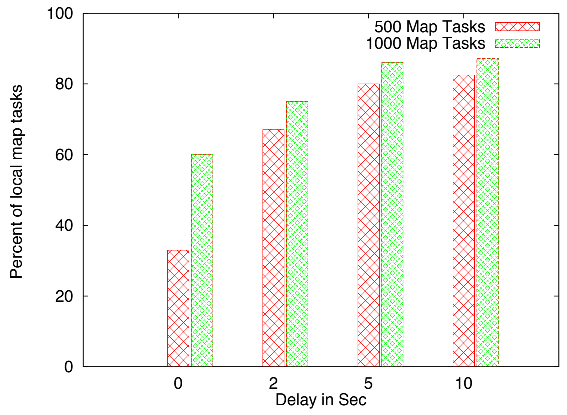
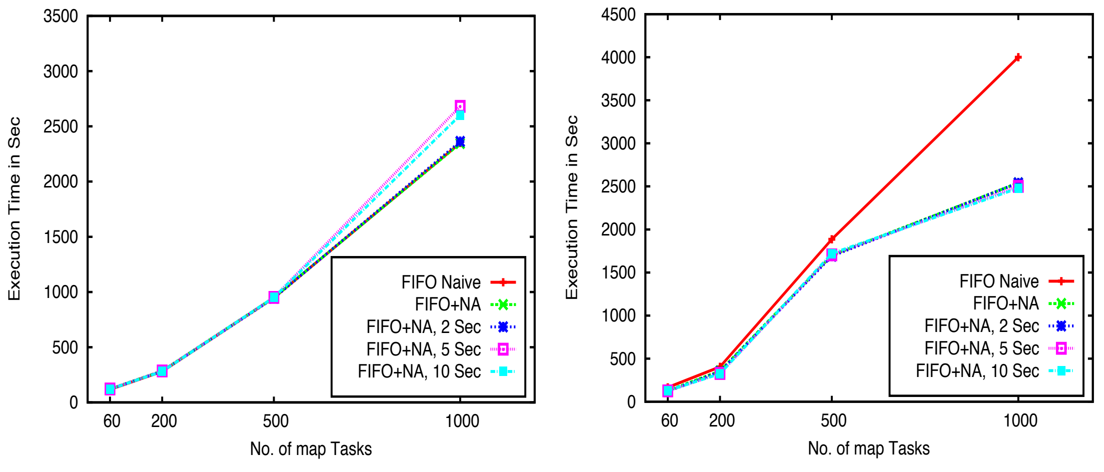


Figure 2.5: Increase in local map tasks with Network-aware

Figure 2.5 shows percentage increase in the local map tasks. As explained in the earlier sections, Hadoop scheduling follows the philosophy of moving the computation to data. All the tasks which have the data on the same node as the one requesting for the tasks are scheduled first, this is referred as node local. If scheduler cannot find such tasks then all the tasks whose data is located in the same rack as the node requesting for the task are scheduled, this is referred as rack local. Network awareness is applied to non local map tasks which require to fetch data from some other data nodes. Network aware Hadoop minimizes the data movement from one cluster to another while executing map task by adding cluster level locality. We enable delay scheduling to optimize the data locality. Delay scheduling ensures that the task is scheduled on a tasktracker which does not have the data to process will be skipped for configured amount of time. If any of the tasktracker becomes free in that duration which as the data to process then, task is scheduled on the second tasktracker. For data intensive applications, data split movement takes more time than processing of the data split. Delay parameter is a sensitive parameter and should be carefully configured. In the Figure 2.5, delay of 0 sec refers scheduling without enabling the delay scheduler. With 2 Sec delay, non local map tasks will be delayed for 2 sec to check if any of the other tasktrackers request for a task. Since the length of each task is small, it is observed that tasktrackers get freed very frequently and request for a task. It is for this reason the delay parameter depends of the length of the task execution. These results show that network awareness coupled with delay scheduling could be used to minimize the transfer of the data between the clouds.



(a) 1Gbps bandwidth and 30ms delay between clusters

(b) 100Mbps bandwidth and 30ms delay between clusters

Figure 2.6: Execution times for 30ms inter-cloud delay

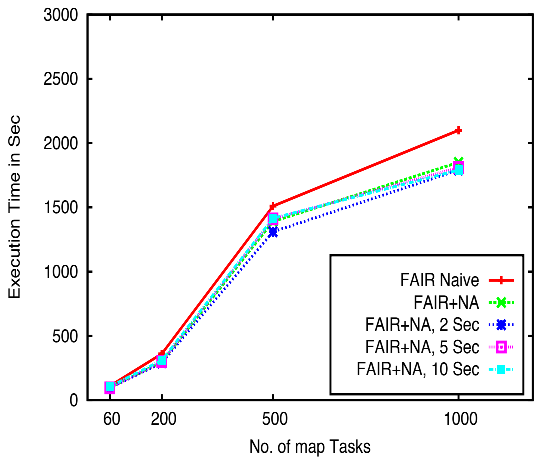
Figure 2.6 shows the results for 30ms inter cloud delay for varying bandwidths of 100Mbps and 1 Gbps. For the same number of map tasks, the execution times for a Job is less for 1Gbps bandwidth. In case of 100Mbps bandwidth the gap between curves for different scheduling schemes is less significant. Reduced available bandwidth between the clusters causes bottleneck for data split movement for tasks scheduled on distributed clusters. In figure 2.6(b), for tasks ranging from 500 to 1000 the execution times for native Hadoop and different variants for Network aware Hadoop, significant difference is observed though the inter cluster bandwidth is less.

Figure 2.6(a) shows experimental results for 1Gbps inter cloud bandwidth and 30 ms delay. Since higher bandwidth is provisioned between the clusters the overall execution times for the jobs compared to figure 2.6(b) is less. Further for jobs with higher number of tasks i.e tasks ranging from 500 to 1000 map tasks, the performance of network aware Hadoop is significantly improved. This shows that the performance Hadoop could be increased with provisioning higher bandwidth, less delay coupled with locality awareness. Similar results were observed for experiments conducted with fair scheduler.

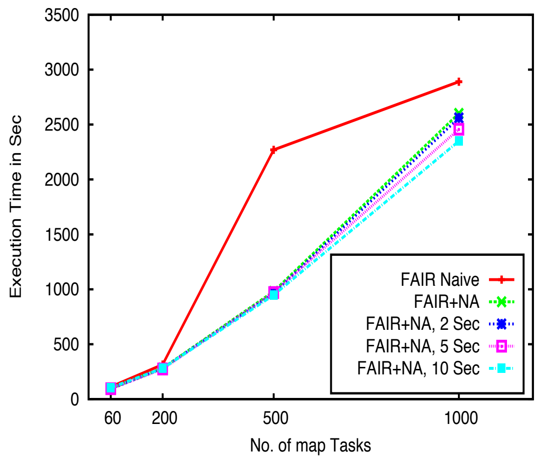
Figure 2.7 shows 60ms inter cloud delay and bandwidths of 100 Mbps and 1 Gbps. There is significant reduction in the overall job execution times between 100 Mbps and 1 Gbps scenarios. However, figure 2.7(a) shows results for 100 Mbps and 60 ms inter cloud network parameters. Because of the higher delay between the clusters, there is not significant improvement between the native Hadoop and the different variants of network aware Hadoop. Figure 2.7(b) is for 1 Gbps inter cloud bandwidth and 60 ms delay. In this case also because of higher delay between the federated clusters, less performance improvement is observed. However, it is better compared with the 100Mbps scenario.

2.5.1 Further Discussion

Figure 2.8 shows results for 10Gbps bandwidth and 30 ms delay between clusters. We observe that for higher bandwidth and low latency, overall execution times are greatly reduced for both native Hadoop and network aware Hadoop. However it is expected that network aware Hadoop to perform significantly better for higher bandwidth and low



(a) 100Mbps bandwidth and 60ms delay between clusters



(b) 1Gbps bandwidth and 60ms delay between clusters

Figure 2.7: Execution times for 60ms inter-cloud delay

latency but significant difference is not seen from the results shown in the graph. The cause of this behavior is that the virtual interfaces of VM's cannot provide throughput more than 1.2 Gbps. Even though available bandwidth is 10Gbps, the utilization from each of VM does not exceed 1.2 Gbps.

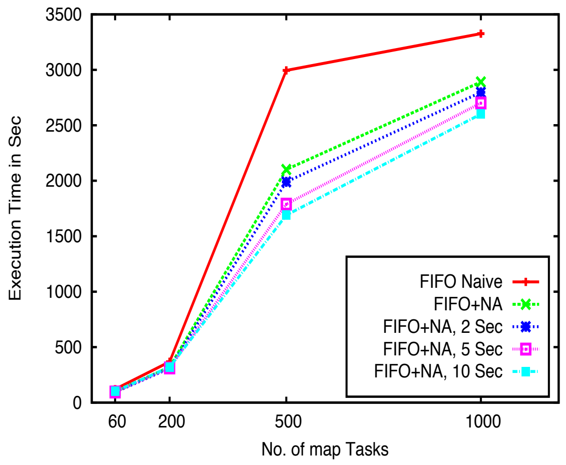


Figure 2.8: Execution times for 10Gbps and 30ms between clusters

2.6 Conclusions and Future Work

Placement of the map task on a node which has the data to process is critical for the performance of Hadoop over federated clusters. Single-cluster MapReduce architecture may not be suitable for situations when data and compute resources are widely distributed. We provide network awareness to the FIFO and FAIR schedulers in Hadoop. We evaluate our implementation on resources provided by CRON testbed. Performance improvement of 12 % to 15 % is observed in both FIFO and Fair schedulers. We plan to extend the network awareness while placing the reduce task since Reduce phase in MapReduce adds up significantly to the overall execution time.

Chapter 3
Distributed Task Scheduler

3.1 Introduction

MapReduce [27] framework is becoming a giant distributed computing framework for large scale data intensive computations. Enormous attention is given to Hadoop [5] from industry and academia. The significance of this framework is acknowledged with the emergence of commercial cloud computing such as Amazon EC2, Cloudera [7] Amazon Elastic MapReduce [3], Rackspace [17] and many other industry variants.

Current MapReduce/Hadoop deployments are over single data center, where in all the components of the framework run within the same data center. In fact Hadoop is developed assuming tree based organization of computing nodes within a single cluster and existing task schedulers are optimized to work well in this computing infrastructure.

However, there is growing demand to operate MapReduce over federated clusters. Many of the large scale scientific, Bioinformatics, Social and computational biology applications generate data from geographically dispersed data sources. Numerous applications such as scientific simulations, Web crawling, Social networking generate terabytes of data which needs to be processed and analyzed across distributed clusters. Current MapReduce setups process such a huge data by transferring it to a centralized computing cluster. Transferring large volumes of data is time consuming, inefficient and increases threat for loss of data. Another approach is to configure global MapReduce over all the distributed clusters in which the scheduling decisions are made at single site and all the other sites work as computing clusters. The challenging aspect of global MapReduce is how to set up such clusters and optimize existing single cluster MapReduce scheduler to work well over distributed clusters.

Global MapReduce provides increased computing power, at the same time poses new challenges to the MapReduce scheduler. Existing scheduler follows the pull model of task assignment to a Tasktracker which is free and requesting for a task. Scheduling involves assigning of map tasks on machines which already have stored the data block corresponding to the map task. However, if Tasktracker gets assigned a map task for which it does not have the data block locally, in this case data block needs to be downloaded from another datanode. If the datanode is located at a remote cluster then fetching the data block gets costlier and degrades the performance of Hadoop. Thus assignment of tasks randomly without considering the network characteristics between clusters becomes bottleneck for the performance of Hadoop over distributed clusters. The objective of the distributed task scheduler it to enhance MapReduce to consider network characteristics while scheduling non-local map tasks.

3.2 Background

Hadoop follows pull model of scheduling tasks rather than push model. Client services are referred as Tasktrackers(TT) and master service is referred as Jobtracker. Each TT is configured with fixed number of map and reduce slots to run map or reduce tasks in parallel. TT requests for new map task from Jobtracker(JT) through heartbeat packets. We explain scheduling in Hadoop over single cluster with an example as shown in Figure 3.1. TT_0 sends heartbeat packet to JT when its map slot gets freed. JT responds with map task M_0, since data block D_0 is located on TT_0. As we can see there is no data movement and this conforms a data local map task. When TT_1 sends a request for map task, it gets assigned any of the map tasks from the unassigned set $\{M_1..M_4\}$. Since none of these tasks have data blocks located on TT_1, there is transfer of data blocks to

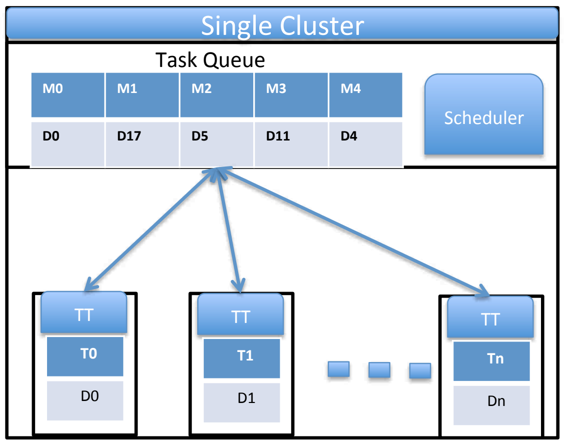


Figure 3.1: An example of scheduling in Hadoop over single cluster set up

TT_1, this constitutes non-local map tasks. However, since all the TT's are located in the same data center there isn't significant delay or network traffic generated for transferring data blocks for processing non-local map tasks. Scheduling of reduce tasks also follows the pull method but without considering the locality.

3.3 Motivation

Figure 3.2 shows the deployment of single instance of Hadoop over remote clusters. One of the nodes from any of the remote cluster runs JT and all the remaining nodes from every remote cluster run TTs. We have only shown delay and bandwidth however, other impairments such as packet loss, jitter etc. exist between the remote clusters. T_0, T_7 and T_{50} are the TTs requesting for map tasks at the moment. T_0 gets assigned map task M_0, since data block D_0 to process M_0 is located on T_0 to achieve data locality. T_7 gets assigned task M_6 since data block D_7 to process M_6 is located on T_7. JT needs to assign T_{50} with any of the unassigned tasks from the set $\{M_1,M_2,M_3,M_4,M_5,M_7,M_8,M_9\}$, these tasks are referred as non-local map tasks since none of these map tasks have data blocks stored on T_{50}. Assigning any of these unassigned map tasks results in moving

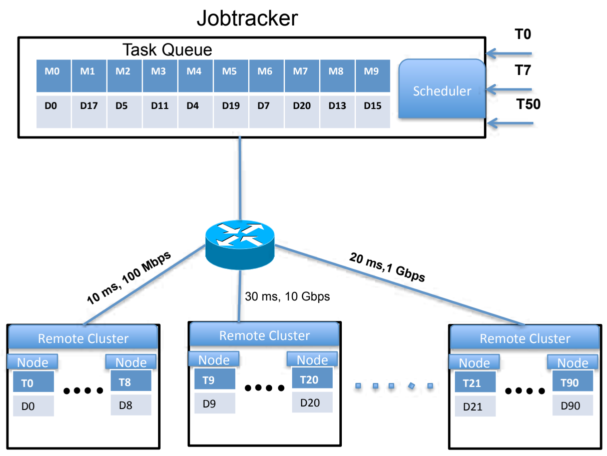


Figure 3.2: An example of task scheduling in MapReduce configured over distributed clusters. Remote clusters are geographically dispersed with significant network impairments between them.

corresponding data block from remote cluster to T_7.

Clearly, if JT assigns non-local map tasks randomly to requesting Tasktrackers without considering the underlying network impairments, then moving data block from remote cluster to the Tasktraker results in significant delay of processing of that map task and degrades the over all performance of Hadoop. We enabled JT to select a non-local map task so that the data block associated with that map task takes less time for transferring to requesting Tasktracker.

Furthermore, stability of network is critical for efficient functioning MapReduce over distributed clusters. Characteristics such as bandwidth, RTT (latencies), packet loss and Jitter (reliability of the network) illustrate the state and quality of the network which are vital for the data intensive scheduling decisions. This lead us to the development of network monitoring service(NMS) which can fetch the network status and aid scheduler to make efficient decisions.

3.4 System Overview

This section presents the goals, main challenges and basic approach underlying the design of MapReduce for distributed clusters.

3.4.1 Goal

Our goal is to make MapReduce work efficiently over geographically dispersed clusters with minimum deployment complexity. The design must be simple enough to integrate easily with the existing task schedulers and also with any of the future job schedulers for MapReduce.

3.4.2 Challenges

Data Locality : Unlike HPC systems where computation and file system are decoupled, in Hadoop, filesystem (Datanode) and computation (Tasktracker) are co-located on a node. When data is transferred to HDFS it is replicated to Datanodes so that Tasktracker running on that node will have data blocks prior to execution of a task. In the current implementation of MapReduce Hadoop, data locality is achieved by replicating the data at three levels of cache. At the first level, a data split is stored on a node. At the second level, a data split is saved on a node in the same rack and at the third level, a data split is stored off-rack. To achieve data locality, map tasks are scheduled on nodes which already have the data blocks (this constitutes node local). Once the data local map tasks assigned, tasks for which data block needs to be fetched from any other datanode within the same rack are scheduled(this constitutes rack local). If neither of node local or rack local tasks are found then the map tasks are scheduled randomly on any of the tasktrackers requesting for a task. Thus MapReduce scheduling algorithms exploiting data locality are based on the assumption of multiple nodes in a rack, which are connected by an aggregate switch. Figure 3.2 shows the typical architecture for federated clusters which may be connected by regular internet or by dedicated link. In this case the critical assumption needed for data locality is disrupted.

Network Status : Current task schedulers work well at data-center level; where in Hadoop is deployed over nodes within racks and racks connected by switches. However, if Hadoop is deployed over remote clusters connected by high speed networks, task scheduler does not respond to changing network characteristics between the clusters.

MapReduce based data intensive applications process terabytes of data and produce enormous of processed data based on type of application. Furthermore, huge amounts of data is transferred between different stages of job execution, for example data movement from map phase to reduce phase. All these operations happen over remote clusters which are interconnected by various edge network elements and links; some of which are stable and others are unreliable and often congested. The consequence of not responding to the changing network characteristics severely affects the performance of these applications. It is essential for data intensive computations to respond to these changes and change the scheduling policy.

3.4.3 Network Aware Task Scheduler

The proposed network aware scheduler can be viewed as an assignment of *n* non-local map tasks to a set of *m* tasktrackers, in order to minimize the task execution time, there by minimizing overall job execution time. We introduce a split transfer time (STT) matrix to estimate the data split transfer time associated with non-local map task. STT is a two dimensional matrix of size *i x t* where *i* is the remote datanode and *t* is the split transfer time. Each entry in STT(i)(t) indicates the estimated time to transfer data block from datanode *i* to tasktracker *t*. Model requires to compute the data split transfer time from remote datanodes to computing tasktrackers.

Data split transfer time is greatly influenced by the network parameters, which also determine the quality of the network links. Since we intend to calculate the data transfer capacity, the most important network parameter is TCP throughput. Mathis [42] formulate TCP throughput calculation using,

$$Rate <= (\frac{MSS}{RTT}) \times \frac{1}{\sqrt{p}} \tag{3.1}$$

Rate is the TCP transfer rate, MSS is the maximum segment size, RTT is the round trip time and p is the packets loss. Given the measurement of these parameters we can compute the transfer rate or bandwidth of data split from datanodes to tasktrackers. It is evident from the above formula that all these network parameters should be made part of our scheduling decisions. Thus network cost (NC) for data split from datanode *i* to tasktracker *t* is,

$$NC(i)(t) = \frac{RTT(i)(t) \times Loss(i)(t)}{Bandwidth(i)(t)} \tag{3.2}$$

We can deduce the data split transfer time from NC and the data split size with the below equation,

$$STT(i)(t) = NC(i)(t) \times Data_Split_Size \tag{3.3}$$

Hadoop uses heartbeart packet mechanism to send the status information from TT's to JT, this information is sent periodically to the jobtracker. STT is sent along with heartbeat packet periodically to the Jobtracker.

3.4.4 Task Scheduling over Distributed Clusters

In this section, we describe how scheduling of non-local map tasks is accomplished over distributed clusters with the distributed task scheduler. We consider the same example that we took while describing the motivation for our work. In the example, JT needs to assign T_{50} with any of the unassigned tasks from the set $\{M_1,M_2,M_3,M_4,M_5,M_7,M_8,M_9\}$.

Random selection of unassigned map task results in data split movement from data nodes on which splits corresponding to these tasks are located to T_{50}. In this situation JT has to make an efficient decision to choose a task which results in lesser split transfer time. JT makes use of STT entries to make an optimized task selection.

JT needs to assign T_{50} with any of the unassigned tasks from the set $\{M_1,M_2,M_3,M_4,M_5,M_7,M_8,M_9\}$, these tasks are referred as non-local map tasks since none of these map tasks have data blocks stored on T_{50}. Assigning any of these unassigned map tasks results in moving corresponding data block from remote cluster to T_7.

3.5 Implementation Details

Although Hadoop supports pluggable job scheduler design extension, same flexibility is not provided for task scheduling. This implementation has changes to jobtracker, tasktracker modules of Hadoop and developing an independent network monitoring service based on PingER [16] tool.

3.5.1 Network Measurement Service (NMS)

We have developed a network metrics measurement service to collect comprehensive up-to-date status of the network parameters. Three parameters which aptly describe a network are delay, bandwidth and throughput are periodically measured to determine the dynamic status of the network. The basis for this service is PingER tool, which allows to send packets of a user selected length to a remote node and to have it echoed back. PingER tool is ideal for our situation since it has minimal network bandwidth requirement as well as modest cpu time. Furthermore, we can modify number of configuration parameters of PingER to suit our cluster set up. PingER uses the Internet Control Message Protocol (ICMP) echo mechanism also known as ping facility. NMS is integrated into tasktracker as child process and executed periodically to collect the network statistics. At every call of NMS a sorted list of datanodes is constructed which gives the minimum split transfer time from a tasktracker to others datanodes. This sorted list is sent to jobtracker along with heartbeat packet.

3.5.2 Tasktracker

The changes to tasktracker include, call to NMS periodically, analyze the values returned and embed sorted list of datanodes into the heartbeat message. Since NMS is launched as a separate thread from tasktracker, it needs to be synchronous with tasktracker. In order to avoid blocking of tasktracker till NMS returns results, we implement the communication between tasktracker and NMS through a file. NMS writes statistics to a file and tasktracker reads that file during every heartbeat message transmit. This approach is efficient since it does not involve longer waiting for tasktracker to collect data from NMS and avoids thread blocking.

3.5.3 Jobtracker

It is here the task scheduling decisions are made to allocate non-local map tasks. For every request from a tasktracker if Jobtracker finds that, there are no local map tasks, then it checks the sorted list of datanodes sent by the tasktracker. It picks up first datanode from the list and checks if any of the map task's data split is located on that tasktracker's datanode. If it finds such task then tasktracker gets assigned corresponding non-local map task. This process is followed for every requesting tasktracker.

3.5.4 Population of STT matrix

TT and NMS to gather accomplish the task of populating STT matrix. NMS is started by TT during cluster initialization and runs periodically with period specified by the cluster administrator. At every run it collects parameters such as throughput, delay and packet loss between TT and datanodes. These values are used as parameters for equations 3.2 and 3.3 to populate STT entries, based on these a sorted list of datanodes is created. This list is written to a file which is accessed by the TT during heartbeat packet construction

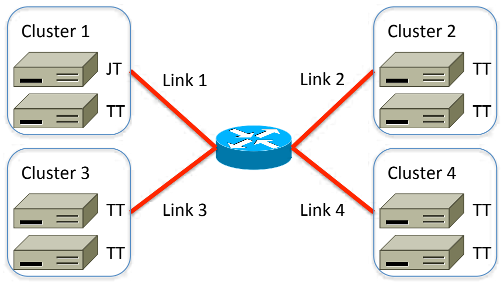


Figure 3.3: Experimental set up for distributed clusters

and sorted data list is sent to JT along with heartbeat packet.

3.6 Experiments

We used CRON [19] testbed to evaluate the performance of distributed task scheduler. The cluster consists of 4 groups of machines. Each group has 2 machines. The 4 groups inter-connect through a software router. Figure 3.3 illustrates the topology of the distributed cluster. We manipulate the bandwidth, propagation delay, and packet loss rate of the links between the 4 groups to emulate 3 categories of network conditions, 1) Worst network, 2) Average network, and 3) Best network. Tables 3.2 , 3.3 and 3.4 indicate the detailed network parameters of the 3 conditions respectively. All machines used in our experiment including the software router are Sun Fire X4240 workstations with 2 AMD Opteron 4-core 2.7GHz processor, 8GB of main memory, 500GB of disk space in a RAID 0, and multiple 10Gbps network interface cards. All machines run the Ubuntu Linux 10.04 LTS. In the links between the 4 groups of machines, we insert an Anue Network Emulator to each of them. With an emulator, the bandwidth, propagation delay, packet loss rate, and many other parameters of a link can be precisely configured.

In order to compare the performance of distributed hadoop and native hadoop, we conduct several experiments under different networking conditions as stated before. Three network conditions worst network, average network and best network conditions are chosen to be closer to the realistic network scenarios in a commodity cluster. Table 3.1

Table 3.1: Execution Workload

Workload	Input data (GB)	Output data (GB)
Wordcount	512	8.6
	256	5.2
	128	2.8
	64	1.52
Sort	512	512
	256	512
	128	512
	64	512

Table 3.2: Parameter configurations for worst network condition

Network	Bandwidth(Mbps)	Delay (ms)	Packet Loss
Link1	100	90	10^{-2}
Link2	1000	60	10^{-3}
Link3	1000	70	10^{-2}
Link4	1000	70	10^{-3}

shows the type of application and data size used for our experiments. Table also shows the output data size produced.

3.7 Results and Analysis

Figure 3.4 is for comparing performance between native hadoop and distributed hadoop for different network conditions. We used block size of 128MB for this experiment. Block size is logical unit of splitting data into files. For example if block size is 128MB and overall data size is 512GB then number of map tasks would be data size/block size,

Table 3.3: Parameter configurations for Average network condition

Network	Bandwidth(Mbps)	Delay (ms)	Packet Loss
Link1	1000	40	10^{-4}
Link2	5000	30	10^{-5}
Link3	1000	25	10^{-3}
Link4	10000	35	10^{-4}

Table 3.4: Parameter configurations for Best network condition

Network	Bandwidth(Mbps)	Delay (ms)	Packet Loss
Link1	10000	10	10^{-7}
Link2	5000	15	10^{-4}
Link3	1000	10	10^{-6}
Link4	10000	5	10^{-7}

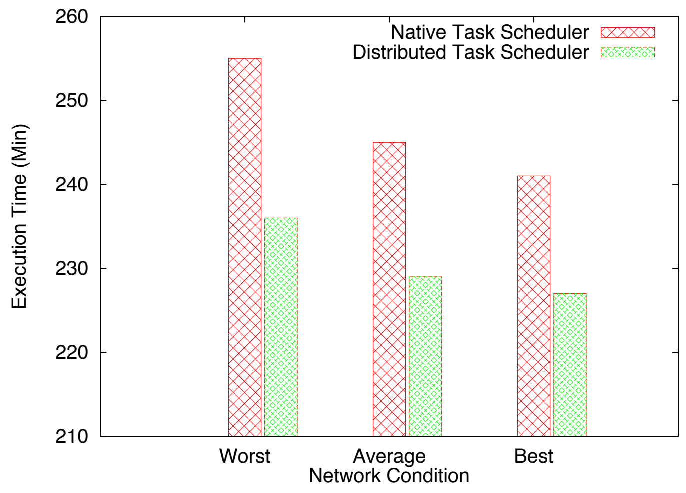


Figure 3.4: Comparison of performance between native task scheduler and distributed task scheduler for 512GB dataset and 120MB blocksize

in this case it is 4000. In worst network condition, the overall job execution time is higher for both native hadoop and distributed hadoop, indicating that network parameters need to be considered for scheduling decisions. We conducted experiments for different input data size, higher improvement was observed for 512GB case indicating that, as the input data size grows much more performance improvement can be achieved by using distributed hadoop. We used wordcount application for this experiment, we also experimented with sort benchmark application and we observed similar results in both the cases. Figure 3.5 shows the results for varying hdfs blocksize. Block size decides the number of map tasks for a job. Though smaller block size increases the parallelism, however framework may introduce overhead for launching those tasks. With distributed clusters block size should be carefully chosen so that, increase in block size

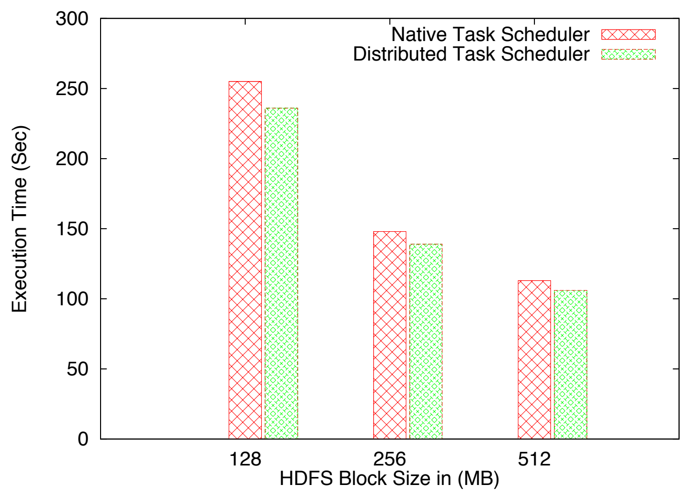


Figure 3.5: Performance comparison for 512GB data for worst network situation for varying hdfs blocksize

should not keep some of the nodes idle, which happens as more data is given to single task and load is not equally distributed amongst all the nodes. It is observed from the figure 3.5, when block size is increased from 256MB to 512 MB, though the overall job execution time is reduced but the drop factor is not same as when we decrease the block size from 128MB to 256MB. We compare the performances between native hadoop and distributed hadoop for fixed block size and varying the input data size. Figure 3.6 indicate that increasing the block size, increases the overall job execution time for both native hadoop and distributed hadoop which happens because, number of map tasks are reduced significantly and framework overhead is greatly reduced to launch less number of tasks.

3.8 Related Works

Researchers have significantly investigated on job scheduling over clusters, grids and clouds. Many of the distributed job schedulers [24] [28] deal with allocation of job based on the availability of computing power, memory and storage at participating cluster without considering the data locality. These schedulers are suitable for compute intensive applications where data movement between remote clusters is not significant.

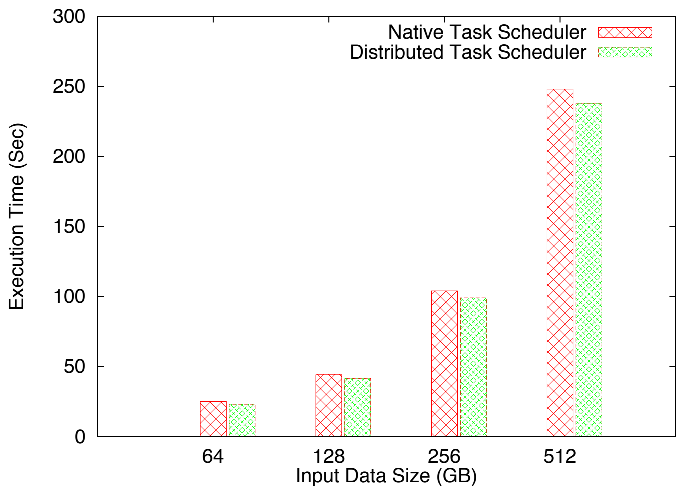


Figure 3.6: Performance comparison for varying input data for 128Mb block size and worst network condition

Cardos et al. [25] showed different architectures to set up MapReduce framework when source data and computation are distributed. They proposed three architectures, Local MapReduce where in data is moved to centralized cluster to perform computation. Global MapReduce where cluster is established with nodes from all the clusters and Distributed MapReduce where in small independent MapReduce clusters similar to architectural overlay. However, their suggestions are different configurations and fail to address data locality concerns as well optimization needed to better the task scheduler over distributed clusters.

In our approach we consider Global MapReduce, that is single Jobtracker handles scheduling of tasks over all the distributed clusters. and provide network parameters as the input to task scheduler.

3.9 Conclusion

As the demand for computing power is ever increasing there is a need to aggregate resources from geographical dispersed clusters. Hadoop task scheduler is optimized to operate over single cluster which connects computing nodes within local networks. Hadoop can be configured over remote clusters, however the scheduling of the tasks is not as efficient as on a single cluster since underlying networking parameters such as bandwidth, delay and packet loss play significant role for data movement. As far as we know not much of research has been done to include network parameters to make scheduling decisions. We proposed distributed task scheduler for optimizing task scheduling in hadoop over distributed clusters. The implementation has minimal framework overhead and can be easily integrated with existing job schedulers.

Chapter 4
Parallel Construction of Suffix Tree

4.1 Introduction

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 in reads are mapped against a reference genome to find similar or dissimilar regions, or to align the short reads to form the genome. Many alignment tools have been proposed to date [43, 39, 37, 45] which invariably rely on a precomputed index structure thus allowing 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 sequences. There are number of data structures available for indexing a string. We consider suffix tree for indexing the human genome.

Suffix tree is a widely used indexing data structure for many of the sequence based problems, such as pattern matching, finding the substrings etc. Since suffix trees greatly improve the performance of searching on the indexed string, they are ideal for indexing the human genome. In fact, many of existing sequence alignment tools use suffix tree to find matching patterns over the indexed genomic data. Due to the significance of suffix tree data structure, many sequential construction algorithms have been proposed [54, 44, 29, 32, 23, 26, 49] for the same. However, these algorithms face difficulties to scale up as the size of genome increases due to high memory requirements ($\approx$ 20 to 30 times the size of the genome). This necessitates the ability to construct and store the suffix tree in distributed manner.

With emergence of MapReduce programming model as a prominent distributed computing framework, many domains of science are turning to MapReduce to efficiently parallelize their computation. The MapReduce framework offers flexibility by expressing the computation in the form of key and value pairs and enables automatic parallelization, distribution of work. Fault tolerance is also managed intrinsically by the framework. We investigate the feasibility of parallel construction of suffix tree using MapReduce framework. Intuitively, our approach is to partition the suffix tree "vertically" and then construct each of these subtrees independently at a computing node. We note that the vertical partitioning takes into account the availability of primary memory at computing nodes as well as the number of computing nodes to be used. We use Hadoop, an open source implementation of MapReduce over 32 node cluster to analyze the performance of the proposed parallel suffix tree construction technique. We construct suffix tree for different genomes of varying sizes from 500MB to human genome of 3.5 GB.

4.2 Background on Suffix Tree

Given a string $S[1...n]$ drawn from an alphabet set Σ, a substring $S[i...n]$ with $1 \leq i \leq n$ is called a suffix of S. The lexicographic arrangement of all n suffixes of T in a compact trie is known as the *suffix tree* of S [56], where the ith leftmost leaf represents the ith lexicographically smallest suffix. Some of the key properties of the suffix tree are listed below:

- Except for the root, every internal node has at least two children.
- Each edge in the suffix tree is labeled by a (non-empty) character string and for any node u, $path(u)$ is the string formed by concatenating the edge labels from root to u.
- For any leaf v, $path(v)$ is exactly the suffix corresponding to v.

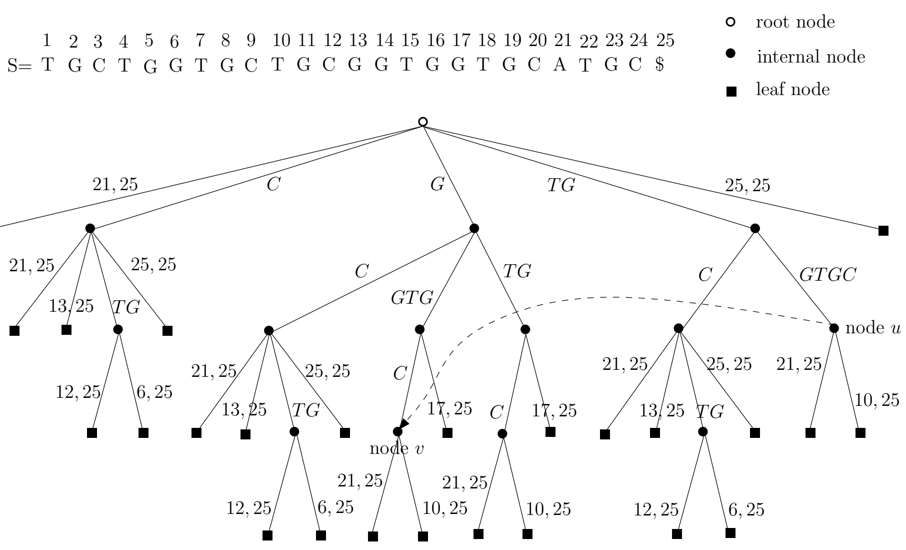


Figure 4.1: Input string S, where $\Sigma = \{A, C, G, T\}$, and corresponding suffix tree

- No two edges starting out of a node can have string-labels beginning with the same character in set Σ.

For a given pattern P, a node u is defined as the *locus node* of P if it is the node u closest to the root such that P is a prefix of $path(u)$. Then all leaves in the subtree of the locus node represent the occurrences of the pattern P in string S. Such a node can be determined in time linear to the length of P and this optimal search time is a primary reason for the popularity of suffix tree in string searching and other related research areas. Figure 4.1 shows the suffix tree for string $S = TGCTGGTGCTGCGGTGGTGCATGC\$$ with $\Sigma = \{A, C, G, T\}$ that we use in a running example throughout this chapter. Note that we assume the input string $S[1...n]$ to be appended with a termination character \$ not present anywhere else in the string i.e., $\$ \notin \Sigma$. The termination character is needed since the construction algorithm is online and every substring considered while adding each character needs to be unique.

4.3 Suffix Tree Construction

We begin this section by first briefly reviewing the well known linear time suffix tree construction algorithm by Ukkonen [54] and then adopt it for parallel construction of suffix tree using MapReduce framework.

4.3.1 Ukkonen's Suffix Tree Construction Algorithm

Ukkonen's algorithm works in phases where in phase $i + 1$ it builds implicit suffix tree of $S[1...i+1]$ from that of $S[1...i]$ in an incremental way. Finally, implicit suffix tree for $S[1...n]$ is then converted to a suffix tree as required. An implicit suffix tree for a given string P can be thought of as a stripped-off version of a suffix tree for string $P\$$. Given the suffix tree of $P\$$ every copy of the termination characters i.e., $ is removed and then all edges with empty string as a label due to such removal are deleted as well. Further, from the resulting tree all nodes that do not have at least two children are eliminated to obtain the implicit suffix tree of P.

Ukkonens algorithm processes string S by reading one character at a time and introducing the consecutive prefixes up to that character. In phase $i + 1$ algorithm considers the prefix $S[1...i + 1]$ and insert all its suffixes $S[j...i + 1]$ for $j = 1$ to $j = i + 1$ in the implicit suffix tree built for $S[1...i]$. To insert the suffix $S[j...i + 1]$ we need to locate the end of the path in the current tree that reads as string $S[j...i]$ and then extend it by character $S[i + 1]$. Following three scenarios can arise during the $i + 1$ suffix insertions (extensions) to be performed.

1. There exists a leaf node v such that $path(v) = S[j...i]$. In this case, we only need to append $S[i + 1]$ to the label for edge leading to the node v.

2. Path for $S[j...i]$ does not end at a leaf node and following character on the edge label is $S[i + 1]$. Then, suffix $S[j...i + 1]$ is already present in the tree and no further action is required.

3. Path for $S[j...i]$ does not end at a leaf node and following character does not match $S[i+1]$. To handle such a scenario, we split the edge and create a node node u along with a leaf node v such that edge label connecting u to v is given by $S[i+1]$.

Though a naive way of implementing the suffix insertions may result in cubic complexity, Ukkonen's algorithm achieves linear construction time using *Suffix links* along with other techniques. Here we focus only on suffix links as the other techniques are easily adoptable for the parallel construction described in next subsection. Suffix links are pointers between internal nodes of the suffix tree. Let $c \in \Sigma$ be a single character and P be any arbitrary string. Given two internal nodes in the suffix tree u and v such that $path(u) = cP$, $path(v) = P$, suffix link points from u to v. Figure 4.1 shown an example of a suffix link with $path(u) = TGGTGC$ and $path(v) = GGTGC$. Using suffix links insertion of suffix $S[j...i+1]$ during $i+1$th phase can be achieved in constant time. Thus careful maintenance of suffix links leads to improved construction time. Following lemma is the essence of suffix link creation and maintenance in Ukkonen's algorithm.

Lemma 4.1. *Any newly created node while inserting a suffix $S[j...i+1]$ during $i+1$th phase will have a suffix link at the end of insertion of suffix $S[j+1...i+1]$.*

4.3.2 Parallel Suffix Tree Construction Algorithm

Though Ukkonnen's algorithm constructs suffix tree for the input string $S[1...n]$ in $O(n)$ time, it implicitly assumes the availability of the entire suffix tree in primary memory. With suffix tree occupying $\approx$ 20-30 times input string size in practice, necessity to hold the suffix tree in primary memory limits the scalability of the algorithm. Though many algorithms have been proposed with space-time tradeoff in RAM model they face similar challenges while dealing with massive data sets. To overcome this difficulty our approach is to divide the suffix tree into multiple smaller sub-trees and build each of these sub-trees parallelly independent of each other using (adopted) Ukkonnen's algorithm. Before moving on to the suffix tree partitioning, we introduce the notion of S-prefix. A non-empty string that appears as a prefix for any suffix of input string S is a S-prefix. Thus by definition any substring of a string S is a S-prefix. Due to lexicographic sorting of suffixes in the suffix tree, all suffixes beginning with a given S-prefix P form a sub-tree in the suffix tree rooted at u, u being the locus node of P.

4.3.3 Suffix tree Partitioning

With the goal of decomposing the problem of suffix tree construction, we partition the suffix tree into a collection of sub-trees (or corresponding S-prefixes) such that each of them can be accommodated in primary memory and collectively it represents the suffix tree in its entirety. Without loss of generality, we assume all computing nodes to have primary memory of same size and is a input parameter for our parallel construction algorithm. Note that in practice, we need to reserve some portion of memory for scanning the string S as well as working memory for the execution of the program and only left over can be used to hold the sub-tree for a S-prefix. Moreover, we can easily estimate the size of sub-tree of a S-prefix as twice its frequency in S using suffix tree properties. Two common ways to achieving the vertical partitioning of the suffix tree are fixed length and variable length partitioning [32, 30]. In either case, we being with an empty output collection and a working collection of S-prefixes such that every symbol in the alphabet set Σ corresponds to a S-prefix. Both the collections are then refined iteratively. In each iteration, string S is scanned to compute the frequencies of each S-prefix in the working collection. For fixed length partitioning, we extend each S-prefix in working collection by one character with symbols in set Σ after each iteration and we iterate till sub-tree for each of these S-prefixes can be maintained in available primary memory. Then, working collection itself becomes the output collection. Whereas for variable length partitioning, at the end of each iteration, each S-prefix whose sub-tree can be accommodated in available memory is moved from working collection to output collection. The remaining S-prefixes in working collection are then extended by one symbol as before and the process is repeated until the working collection is empty. It can be easily seen that fixed length portioning will lead to lower memory utilization as compared to its counterpart when computing nodes are assigned the task to building the sub-trees one S-prefix per at a time per node due to unbalanced sub-tree sizes. Hence we opt for variable length partitioning for our purpose. Though the partitioning can take quadratic time in worst case, it is known to perform well in practice [30].

Let us assume that we need to partition the suffix tree showing in Figure 4.1 with the restriction that the main memory can hold the sub-tree for the S-prefix having frequency at most 8. It can be observed that variable length partitioning would produce a S-prefix C whereas fixed length portioning is forced to extend it further as S-prefixes G, T have frequencies more than the desired threshold 8 thus resulting in lower memory utilization.

4.3.4 Adopting Ukkonen's Algorithm

We now modify the Ukkonen's algorithm described earlier to build only the sub-tree for a given S-prefix P of length t. As before algorithm works in phases such that in phase $i+1$ it considers the prefix $S[1...bocc_{i+1}+t-1]$, where $bocc_{i+1}$ represents the beginning location of $i+1$the occurrence of S-prefix P in string S. For simplicity let $eocc_{i+1} = bocc_{i+1} + t - 1$ denotes the ending location of $i+1$the occurrence of P. Since we are interested only in the sub-tree for P in entire suffix tree of S, we only need to take into account the suffixes of the prefix $S[1...eocc_{i+1}]$ that being with P. Thus we insert all suffixes $S[bocc_j...eocc_{i+1}]$ for $j = 1$ to $j = i+1$ in the implicit suffix tree built so far. To insert the suffix $S[bocc_j...eocc_{i+1}]$ we need to locate the end of the path in the current tree that reads as string $S[bocc_j...eocc_i]$ i.e. suffix added in phase i that begins at position $bocc_j$. We then extend it with string $S[eocc_i+1...eocc_{i+1}]$. Once again three scenarios can arise during the suffix insertions (extensions) as before and are handled in a similar way.

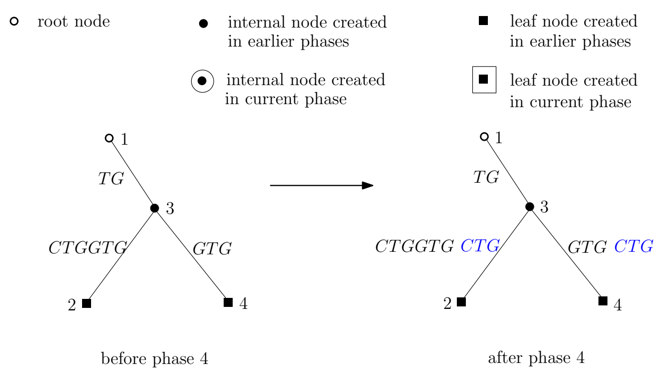
1. There exists a leaf node v such that $path(v) = S[bocc_j...eocc_i]$. In this case, we only need to append $S[eocc_i+1...eocc_{i+1}]$ to the label for edge leading to the node v.

2. Path for $S[bocc_j...eocc_i]$ does not end at a leaf node and following characters on the edge label match the string $S[eocc_i+1...eocc_{i+1}]$. Then, suffix $S[bocc_j...eocc_{i+1}]$ is already present in the tree and no further action is required.

3. Path for $S[bocc_j...eocc_i]$ does not end at a leaf node and the same path can not be continued to spell out the string $S[eocc_i+1...eocc_{i+1}]$ completely. Let $k < eocc_{i+1}$ be the maximal location in S such that we can continue the path of $S[bocc_j...eocc_i]$ and can read $S[eocc_i + 1...k]$. To handle such a scenario, we split the edge and create a node node u along with a leaf node v such that edge label connecting u to v is given by $S[k + 1...eocc_{i+1}]$.

In the event that the S-prefix P has only $i+1$ occurrences in S, $i+1$th phase is the last phase and we consider the entire string $S[1...n]$ instead of just the prefix $S[1...eocc_{i+1}]$. Moreover, we insert suffixes $S[bocc_j...n]$ for $j = 1$ to $j = i + 1$ in the current implicit tree. These changes can be accommodated by simply replacing $eocc_{i+1}$ by n in three cases listed above.

Continuing the example under consideration, we focus on the sub-tree for S-prefix TG. Note that TG will be a part of output S-prefix collection of variable length partitioning as its frequency stratifies the threshold 8. Figure 4.2 shows the implicit suffix sub-tree built for S-prefix TG before and after phase 4. For simplicity nodes have been numbered in the order of their creation and we refer to the edge as a pair (u, v) of nodes. First and second suffix extension in phase 4 simply results in extending the edge labels for edges (3,2) and (3,4) respectively by string CTG using case 1. Whereas for the third



root node
internal node created in earlier phases
leaf node created in earlier phases
internal node created in current phase
leaf node created in current phase
1
TG
3
CTGGTG
GTG
2
4
before phase 4
1
TG
3
CTGGTG CTG
GTG CTG
2
4
after phase 4

Figure 4.2: Constructing sub-tree for S-prefix TG: phase 4

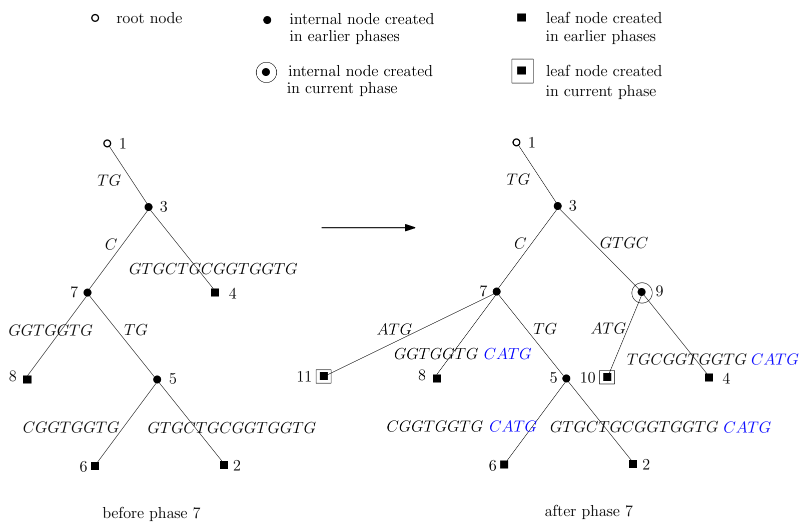


Figure 4.3: Constructing sub-tree for S-prefix TG: phase 7

extension suffix $TGCTG$ already exists and no action is necessary (case 2). Changes required in the implicit suffix sub-tree that exists at the end of phase 6, while considering the prefix $S[1...19]$ in seventh phase of the algorithm are outlined in figure 4.3. In fifth extension of the phase, we need to insert the suffix $S[15...23] = TGGTGCATG$. To achieve this, in the current implicit suffix sub-tree we first locate the end of suffix $S[15...19] = TGGTG$ that was inserted in the previous phase. We observe that, we can continue to match only one additional character i.e. $S[20] = C$ from the desired extension string $S[20...23] = CATG$. Then, by using case 3 we create a new internal node (node 9) and a new leaf node (node 10) as seen in figure 4.3. Similarly, extension sixth also create a new leaf node (node 11). To be able to execute each of the three cases efficiently (constant time), we turn our attention to suffix links. Since we would like to build only the subtree of S-prefix P, we define suffix links to have origin as well as target within the subtree itself. For any internal node u within the subtree of S-prefix P, we can write $path(u) = P\alpha$. Lets split α using smallest (possibly empty) string β such that $\alpha = \beta P\gamma$. Then suffix link points from u to node v having $path(v) = P\gamma$. If such a string β does not exists we assume suffix link from u points to the root node of the subtree of P. Lemma 4.2 can now be rewritten as follows which captures the essence of suffix link maintenance in parallel suffix tree construction.

Lemma 4.2. *Any newly created node while inserting a suffix $S[bocc_j...eocc_{i+1}]$ during $i+1$th phase will have a suffix link at the end of insertion of suffix $S[bocc_{j+1}...eocc_{i+1}]$.*

Proof. A new internal node u is created while inserting a suffix $S[bocc_j...eocc_{i+1}]$ (of phase $i+1$) only when extension rule 3 applies. For any $j < i+1$, let us express the string $S[bocc_j...eocc_i] = P\beta P\gamma'$, β being the smallest possible string. If such a β does not exist then by definition suffix link of u can be established immediately by pointing it towards the root node of the subtree being built. Otherwise, creation of a new node u implies that in the current suffix insertion, the path labeled $S[bocc_j...eocc_i]$ can be continued to spell out $\gamma'' = S[eocc_i + 1...k]$ only up to some $k < eocc_{i+1}$, which is then followed by some character other than $S[k+1]$, lets say $c \in \Sigma$. Thus, in extension $j+1$, there is already a path labeled $P\gamma'\gamma''$ in the tree and it certainly has a continuation with character c possibly along with some other characters as well. Then, there are two cases to consider: Either the path labeled $P\gamma'\gamma''$ continues only with character c or it continues with some additional character. When $P\gamma'\gamma''$ is continued only by c, once again extension rule 3 will create a node v at the end of path $P\gamma'\gamma''$ to which suffix link of u will point to. When $P\gamma'\gamma''$ is continued with two different characters, then there exists a desired node v at the end of path $P\gamma'\gamma''$ already. Thus, the Lemma is proved in either case.

$\square$

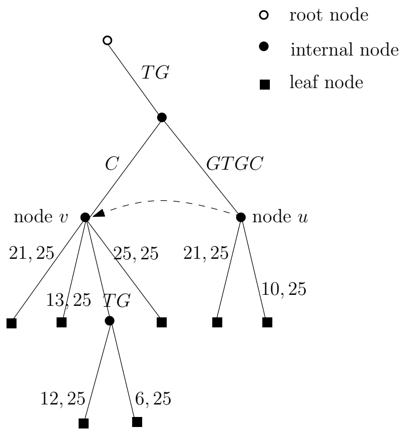


Figure 4.4: Subtree for S-prefix TG and sample suffix link in its context

Following the above lemma, all internal nodes in the changing subtree will have suffix links from them, except for the most recently added internal node, which will receive its suffix link by the end of the next suffix insertion/extension. Figure 4.4 shows the example of a suffix link in the context of a subtree for S-prefix TG. Here $path(u) = P\beta P\gamma$ such that $\beta = G$ and $\gamma = C$ and accordingly $path(v) = P\gamma = TGC$. Note that when node u (node 9 in figure 4.3) is created during fifth extension of phase seven, node v (node 7 in figure 4.3) already exists in the tree thus verifying the above lemma for the suffix link under consideration. Therefore using suffix links along with skip and count trick, edge-label compression, a stopper and once a leaf, always a leaf trick subtree [] for the given S-prefix can be built by a simple scan the string S. It can be easily be verified that the all the performance improvement techniques other than the suffix links works exactly in the same way as they would in the original Ukkonen's algorithm.

4.3.5 Hadoop Implementation Details

The basic idea is to provide each map task a S-prefix from the set of S-prefixes generated using variable length partitioning method. A S-prefix and its starting positions constitutes the key-value input format need for Hadoop. Every map task scans the input string and group together all of its suffixes. Map tasks run in parallel and produces a S-prefix subtree. For the completion of suffix tree of input string, we can decide to build the trie of all generated S-prefixes in reduce phase. In our experiments, construction time of such a trie was found to be insignificant as compared to the best map task execution time. Thus, we choose to combine the trie building with S-prefix generation and avoid overheads associated with shuffle, reduce phase by making them optional.

Input String: The algorithm distributes the input string S to every worker machine. Though a single copy of string can be stored on a shared file system, simultaneous access to shared file from all the map task would introduce a file access bottleneck and severely degrade the performance of the algorithm.

InputFormat: A file containing a S-prefix as a key and its frequency in S as the value. Though the adopted Ukkonnens algorithm does not require the frequency of S-prefix explicitly it aids in allocating only desired amount of main memory instead of blocking all the available space. Moreover, since it is not tractable to distribute S-prefixes explicitly, S-prefix key is formed by a pair (pos, len), where pos represents the index position in S having first occurrence of given S-prefix and len presents its length.

Map Task: The mapper executes our adopted Ukkonnen' algorithm to construct the sub-tree for the S-prefix it received. Starting with an empty tree, it inserts all suffixes of S beginning with input S-prefix. On successful completion mapper writes S-prefix sub-tree to the local disk.

We note that the vertical partitioning technique described earlier accepts the the available primary memory as input. Thus the number of S-prefixes generated may not be sufficient to achieve complete parallelization offered by the number of worker nodes available. To overcome this, we use an empirical technique, which artificially reduces the main memory required to generate more S-prefixes than the number of available worker nodes. We can estimate the size of subtree to be produced by each worker node by dividing the total suffix tree size by number of worker nodes. However, such an optimal load balancing can not be achieved due to data skewness. Therefore, we allocate additional 5%-20% of the load balanced share and use this as an input parameter for generating the S-prefixes. On the other hand, when the number of generated S-prefixes are more than the number of worker nodes, we can partition them into groups to achieve load balancing among worker nodes. However, such grouping is difficult as it needs to balance the number of S-prefixes in group as well as their combined frequency, both being key to algorithms performance. Therefore, we simply rely on the Hadoop scheduling and let jobtracker schedule the map tasks to worker nodes.

4.3.6 Evaluation

This section presents the performance evaluation for the parallel suffix tree construction. Our experiments were conducted on Hadoop configured Supermike (LONI HPC) [13] cluster. Each computing node is a two 2.6 GHz 8-Core Sandy Bridge Xeon 64-bit processor with 32 GB RAM. However, in each node we used only one core and limited the memory usage to 8GB. We used real data sets as shown in Table 4.2 for performance evaluation[1].

4.4 Performance Results

We perform experiments to show (a) Strong scalability - Indicates the results for increasing cores and constant genome size (b) Weak scalability - The ratio of input size and cores is constant. These two types of experiments are commonly used methods for

[1]http://hgdownload.cse.ucsc.edu/downloads.html

Table 4.1: Evaluation Environment

Node Types	Hardware and Hadoop
Master Node	2.33 GHz quad core xeon 64-bit processors 8 GB RAM Jobtracker Namenode
Slave Nodes	2.33 GHz quad core xeon 64-bit processors 8 GB RAM Tasktracker Datanode

Table 4.2: Genomes used for Suffix Tree Construction

Genome	**Name**	**Size (Bytes)**
Rice (Oryza sativa)	OS	370,792,118
Chicken (Gallus gallus)	GG	1,031,883,471
ZebraFish(Danio rerio)	DR	1,412,476,020
Cow (Bos taurus)	BT	2,634,413,324
Human (Homo sapiens)	HG	3,095,677,412

evaluating the construction algorithms. We also conduct experiments to compare the speed up gained compared to the single core version of Ukkonen's suffix tree construction algorithm. However, since Ukkonen's algorithm does not scale to larger genomes, we perform speed up experiments only for small sized genomes. The performance results do not include the overhead of variable length prefix construction since it was minimal ($\approx$ 2 min at most) compared to the overall suffix tree construction. We follow the same approach used in the existing construction methods for generating variable length prefixes.

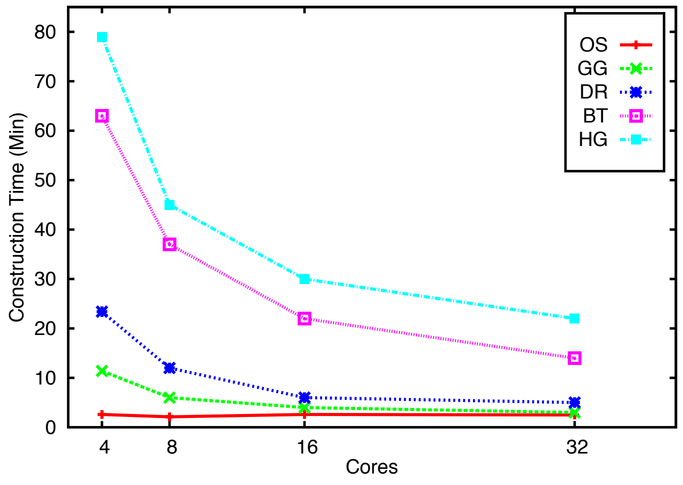


Figure 4.5: Overall execution time by number of cores

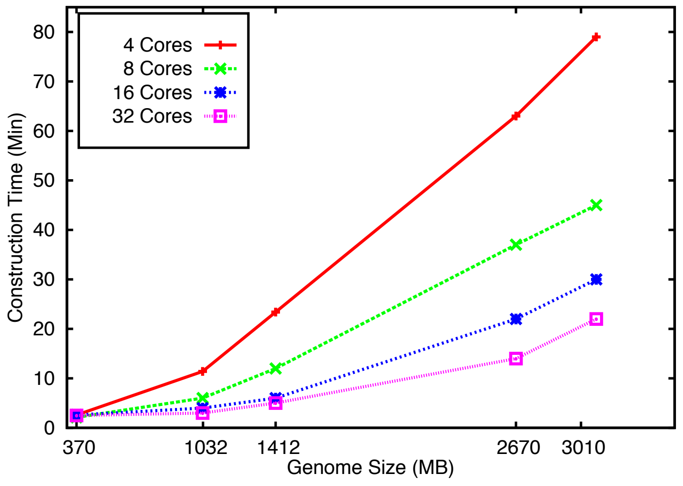


Figure 4.6: Overall execution time by genomes

Figure 4.5 shows the results for strong scalability. We keep the genome size same and increase the number of cores. For each of the genome we increase the number of cores from 4,8,16 and 32. The construction time for human genome over 4 nodes is 79 minutes and comes down to 22 min over 32 nodes. The nature of the curve indicates that, as the cores are increased the construction time decreases as expected for strong scalability. However, the results for smaller genomes indicate that the construction time does not decrease after certain number of cores and remains flat even if we increase the number of cores. The reason for this behavior is that, even if input string size is constant, as the number cores are increased the size of subtree that a core has to build decreases but it still has to load and scan the entire input string. After certain point the overhead of loading and scanning the input string dominates the construction of subtree that a core has to build. Thus every genome as critical point after which the overhead of loading and scanning dominates the construction of subtree. Figure 4.5 does not include the single core execution times for genomes BT and HS because of insufficient main memory to construct suffix tree for these genomes. Figure 4.6 is overall suffix tree construction time as function genomes size.

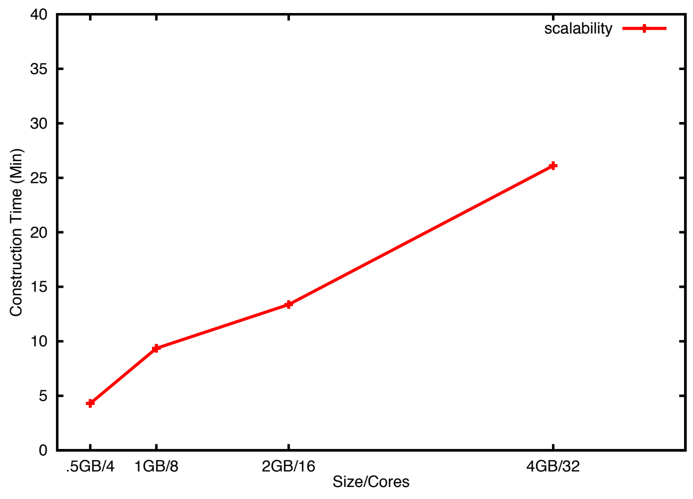


Figure 4.7: Results for weak scalability

Figure 4.7 shows results for week scalability. We vary the genome size from 512MB on 4 nodes to 4096MB on 32 nodes, keeping 8GB memory per map task, so that the ratio of genome size to number of cores is constant, which represents typical set up for weak scalability. In optimal set up the construction time should remain constant. However, as the genome size increases, the overall number of subtrees to be constructed increases and each map task has to scan the entire genome to build subtree, which increases the overhead of constructing subtrees proportional to genome size. We can see from the

Figure 4.7 that, the curve is not flat rather increases with increase in genome size indicating that optimal weak scalability results are not achieved. We compare the speedup

Table 4.3: Speedup achieved as compared to sequential execution

Genome	Speedup
Rice (OS)	0.92
Chicken (GG)	0.81
ZebraFish (DR)	0.56

achieved by parallel execution of program as against its execution on standalone machine for genomes OS, GG and DR. Table 4.3 shows results for the same with 4 cores in use. We observed the decline in speedup for each of the tested genome with increase in parallelism (refer to Figure 4.5).

4.5 Related Works

Suffix tree construction algorithms have evolved into broadly two main classes: In-memory and Disk-based. In-memory algorithms can be used when both input sting S as well as its suffix tree can be accommodated in the available main memory. Well known suffix tree construction algorithms by McCreight [44] and Ukkonen [54] fall in this category. Though Ukkonen's algorithm can build the suffix tree in optimal $O(n)$ time, this class of algorithms is shown to suffer from poor locality of reference [53]. This issue is highlighted when suffix tree can not be accommodated in relatively faster main memory resulting in $O(n)$ expensive random disk I/Os on an average. This severally restricts the use Ukkonen's algorithm in practice. Suffix tree partitioning technique has been successfully used to overcome the problem of locality of reference [32, 53, 49, 21, 30]. Hunt et al. [32] proposed one of the fist disk based algorithms. In the preprocessing step, the method first computes a set of fixed-length S-prefixes of the input string, such that the suffix subtree for each of the S-prefix fits entirely in main memory. The subtree for each fixed-length S-prefix is then constructed by inserting all suffixes starting with the given S-prefix. As pointed out earlier, suffix tree partitioning using fixed-length prefixes faces difficulty in handling data skew and produces unbalanced suffix subtrees. In [49] authors have proposed to partition the input string itself into several substrings instead of its suffix tree. Suffix tree is then built independently for each of the substring. Finally merge phase aggregates all suffix subtrees into a final suffix tree. This technique is useful when input string can be accommodated in the main memory. Otherwise, due to the problem of locality of reference, merging phase would result in lot of random disk I/Os. Technique in [21] also relies on input string portioning, however it builds suffix array along with LCP array for each of the substring. The algorithm proceed in batches to avoid the costly I/Os during merging phase. Finally, both [32, 30] use variable length portioning making sure that making sure that each S-prefix subtree fits in the main memory. To minimize the I/O cost, they access the input string strictly in sequentially order. Theoretical parallel suffix tree construction algorithms have been studied in the past [31, 36]. However, no practical implementations of these algorithms have been reported, to the best of our knowledge.

4.6 Conclusion

We have proposed a novel approach to construct suffix tree in parallel by extending the well known in-memory Ukkonens algorithm. Since suffix tree is the basis for many bioinformatic applications, building suffix tree for larger genomes would benefit genomic research community. As a future work, we would like to improve the proposed construction algorithm further by taking into account the skewness of the subtrees to be built resulting due to variable length partitioning.

Chapter 5
PGA: Parallel Giraph based Assembler

5.1 Introduction

Next generation sequencing (NGS) technnologies produce massive amount of short read data. Reducing the overall cost and producing high throughput reads are the main goals of NGS. Sequencing the entire human genome is made possible because of NGS technologies. 454 Life Sciences [1] and the llumina Genome Analyzer (GA) [12] were the earliest NGS technologies to enter the market. These methods produce short reads of length 50 to 500 which is significantly smaller than the overall size of the genomes being sequenced. Short read approaches come with cost of significant overhead for assembly as the number of short reads produced is in terms of billions, and short reads within the repeat regions require complex methods to remove ambiguities.

The earliest of assembly tools used overlap-layout-consensus(OLC) methods. In this approach each read is compared with every other read to determine overlaps, and reads with overlap are connected to build the basic assembly graph. In the subsequent stages the graph is compressed, and the final genome sequence is determined by finding a path in the overlap graph by visiting every vertex exactly once which is a Hamiltonian path problem. Although OLC methods proved to be effective for small genomes, it does not scale to larger genomes because of the quadratic time complexity associated with overlap computation and Hamilton path finding during consensus stage, which is a NP hard problem, efficient algorithms to solve this problem are unknown.

A fundamentally different approach was proposed by Pevzner [48] based on de Bruijn graph. A de Bruijn graph is constructed by further splitting the reads into small sequences of length k known as k-mers, if two k-mers differ by single nucleotide base an edge is added between the vertices. This simple k-mer based graph condenses all the repeats into single vertex of the DB graph. Since the human genome is highly repetitive in nature, many of the complexities involved in aligning the repeat regions are inherently handled in de Bruijn graph. From the algorithmic perspective, the problem of finding Hamiltonian path in OLC is reduced to a Eulerian path finding problem in de Bruijn graphs, which can be solved with efficient algorithms.

Although, de Bruijn graphs provide an exceptional advantage over OLC, the huge computational memory demands present a major practical limitation even for small sized genomes. Constructing a single DB graph from the billions of short reads is a complex endeavor. Furthermore sequencing errors from the sequencing technologies aggravate the memory demand and reduce the accuracy of the assembly. Removing errors from the data is therefore a necessary step for accurate assembly. Commonly, sequencing errors are corrected by using several heuristic approaches in the beginning of the assembly process known as error correction phase. However, current algorithms used for correcting the errors are not very accurate and highly memory intensive [33].

Huge demands for memory needed for de Novo Assembly can be addressed by using distributed computing. Recently, a number of distributed computing frameworks, such as MapReduce, Apache Giraph and many others have been proposed for data intensive applications. These frameworks provide a programming paradigm which divides the large datasets into chunks such that these chunks of data are independently processed over small computing environments. The simplicity and the ability to express the de Novo assembly as a distributed graph processing endeavor makes these frameworks more feasible for assembly of large genomes. ABySS [51] and SoapDeNovo [38] are some of the existing tools which use parallel programming models and have shown to be scalable to larger genomes, however these still require substantially larger memory.

We present Parallel Giraph [4] Assembler (PGA) for large scale genomes based on Apache Giraph and de Bruijn graphs. PGA utilizes the distributed memory across the cluster to process the de Bruijn graph in parallel. PGA incorporates the general approach followed in other tools for constructing a distributed DBG and removing of errors in short reads. We demonstrate the performance of PGA by assembling three genomes 1) small E.Coli real paired-end dataset, 2) relatively larger, Thellungiella Salsuginea paired-end dataset and 3) larger genome with 3.5 billion paired end reads from the whole genome sequencing Yoruban male. The results show that for smaller genomes the performance of PGA is comparable to Velvet and significantly better than ABySS, and Contrail. For large genomes PGA performance is significantly improved compared to ABySS and Contrail, and at the same time PGA needs small memory per core.

5.2 Background

5.2.1 De Bruijn Graph

A generalized DBG of order k denoted by G(k), is a bi-directed graph with d^k vertices labelled by a k-mer string over d alphabets. Although DBGs are used for variety of purposes our discussion is limited to its usage for de Novo assembly. Pevzner [48] was the first to introduce DBGs in the context of de Novo genome assembly. In this context, an edge exists between node N_1 and N_2 if the suffix of length k-1 of N_1 overlaps with prefix of length k-1 of N_2. Edges also carry type label with a string in {FF,FR,RF,RR}. The first character edge type indicates, if the prefix or suffix of N_1 overlaps with suffix or prefix of N_2, represented by the second character. Figure 5.1 shows the standard de Bruijn graph for k-mers of length three. The possible k-mers in the this example are GAT, ATT, TTA, TAC, ACA, CAT, ATT, TTA, TAC, ACA and CAA, each of which is represented by a node. The advantage of such representation is that how the repeated regions of the genome are handled. Let R be a long repeated substring of genome and $|R| < k$. Each repeated instance of R generates the same sequence of k-mers. As the de Bruijn graph only contains distinct k-mers, there is a single vertex for each of these k-mers. Finally, the candidate genome is determined by looking for a Hamiltonian path which will also be a minimum length string since every node is visited exactly once. However, this problem again belongs to a class of problems that are called NP-Complete: Efficient solutions to these problems have not yet been discovered. There is need to convert this from NP hard to solvable Eulerian method.

5.2.2 De Bruijn Graph to Assembly Graph

Pevzner [48] showed an alternate representation of a k-mer graph. In this representation, every distinct prefix or suffix of a k-mer is represented as a node and a directed edge is added between two (k-1)-mers if they share a k-mer. For example, given a k-mer GCT, form two nodes with GC (prefix) and CT (suffix) and add an edge labelled GCT between them. The target genome is found by visiting every edge exactly once, which is equivalent to visiting every k-mer. This method of representing and traversing the graph translates the Hamiltonian problem into a solvable Eulerian path problem. This method is employed in most of the assemblers which are based on de Bruijn graphs. An extended example of this approach is shown in figure 5.2.

[0]Part of the figure 5.2 is reproduced from [48]

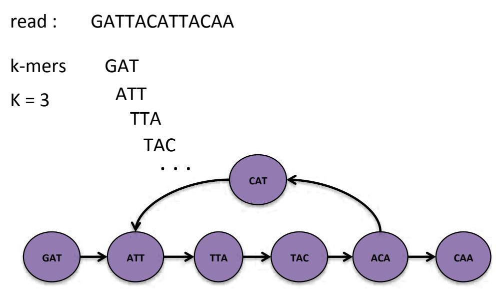


Figure 5.1: Standard de Bruijn graph representing k-mers as nodes. Target genome is found with Hamilton cycle

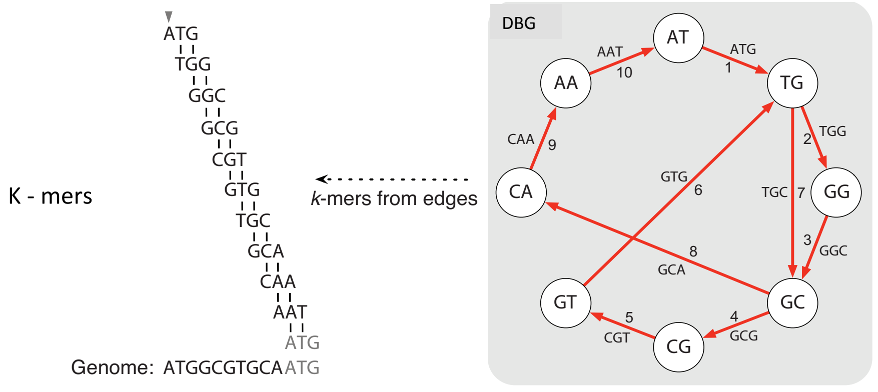


Figure 5.2: Illustration of de Bruijn graph for genome assembly. (k-1) mers form the nodes and assembly is achieved by Eulerian cycle

5.2.3 Apache Giraph

Apache Giraph is an open source implementation of Pregel [41], the graph processing architecture developed at Google. The framework introduced by Pregel and used by Apache Giraph is based on the Bulk Synchronous Parallel (BSP) computation model [55]. The input to a Giraph computation is a graph composed of vertices and directed edges. At the beginning of the computation, the framework splits the graph into several partitions and distributes them across compute nodes. The computation proceeds as a sequence of iterations, called supersteps. In each superstep every vertex executes the compute method provided by the user. A vertex can modify its own value or its out going edges or even modify the topology of the graph. The compute method embodies the graph processing algorithm as specified by the user. At the end of each superstep, all compute nodes synchronize before starting the next superstep. In every iteration the compute method performs the below operations:

- Receives messages sent by all the adjacent vertices in the previous superstep,
- Iterates over all the messages, performs the computation and send the messages to adjacent vertices and,
- Vote to halt if nothing else to compute.

The issues of parallel processing of graph such as, data race and dead-locks are inherently handled by the synchronicity of the Giraph framework. Additionally, the programming interface is simple enough to develop complex graph processing algorithms. Writing Giraph programs involve subclassing the vertex API. Each vertex has a value type as specified by the user. The compute method is applied to each vertex in every superstep. All the graph processing computation is embedded in the compute method.

5.3 Related Works

In the past decade, a considerable number of Novo assemblers have been proposed. Assemblers which are designed specifically for first and second generation sequencing technologies are based on overlap-consensus-layout method(OLC). The assembly is carried out by first building the assembly graph, then converting the unambiguous paths into contigs. In the final stage, the overlap graph is analyzed to find the Hamiltonian path to arrive at the target genome. Assemblers such as [22, 52, 46] are based on the OLC approach. The major difficulty with assemblers based on OLC is polynomial time algorithms for finding the Hamiltonian path in the overlap graph. Accordingly, these assemblers are not used for large genomes.

NGS technology produces massive amounts of data. Although whole genome sequencing is made possible with this technique the length of the reads is small. Consequently OLC approach is not feasible. The class of assemblers which are designed to match the characteristic of massive amounts of data and short read lengths are based on de Bruijn graph. Pevzner [48] showed the simpler representation of the inherent repeats in the large genomes with de Bruijn graph. In this approach, the short reads are further broken into shorter sequences of uniform length which makes the repeated regions to condense into single node in the DBG. As such, the assembly problem is cast as finding a path visiting every edge exactly once, a more feasible Eulerian Path Problem.

The earliest DBG based assemblers is Velvet. It has four stages: finding the unique k-mers by hashing the reads, graph construction, error correction, and repeat resolution. Each step of the assembly has different computational requirements. The bottleneck for velvet, in terms of time and memory, is the graph construction. The results for the assembly of Streptococcus indicate the need of 2.0 GB RAM. For the larger genomes the memory requirement would be greater. Consequently, Velvet can not be easily applicable to the assembly of large genome.

Another assembler which has gained significant traction is ABySS which uses MPI (message passing interface) for parallelization. The assembly is performed in two major steps. First, contigs are extended by resolving all the unambiguous paths. In the second step the paired-end information is used to resolve ambiguities and merge contigs. The general work flow of the assembler is building the de Bruijn graph and storing across cluster nodes, error correction, which includes cleaning the dead-ends formed by the reads that are a mixture of correct and incorrect k-mers, and bubbles, which are topological imperfections formed due to repeated regions. The results showed that assembly of the human genome was performed in roughly three and half days using 21 nodes, each having 16GB of RAM and two quad-core Intel Xeon 2.66GHz processors, giving a total of 168 cores. Although ABySS scales to larger genomes, the time for assembly is huge. Furthermore, assembly is very fragmented with an N50 length of 1.5 kilobases (kb) which is not long enough for structural variation detection between human individuals.

Micheal Shwatz introduced Contrail [8] to addresses the problems associated with the assembly of larger genomes. It uses the power of MapReduce and cloud computing to parallelize computation across cluster consisting of commodity hardware. The steps followed by contrail are based on existing assembly algorithms which required to be embedded in MapReduce framework. Assembly of E. coli K12 substrain MG1655 using 20.8 million paired-end 36 bp of Illumina reads was achieved in about 8 hours using 20 3.2 GHz Intel Xeon cores. However, the assembly of the human genome (HapMap DNA identifier NA18507) using 3.5 billion reads (accession no. SRA000271) was completed till the scaffolding phase in 74 hours with 188 cores.

The main reason for performance bottleneck of Contrail is MapReduce framework itself. The problem of de Novo assembly using de Bruijn graph is essentially a graph-mining problem involving many iterative computations. Contrail uses MapReduce as a parallelization framework, which is designed to process massively parallel dataset in single iteration. Series of iterative steps in Contrail are represented as separate MapReduce jobs, and a driver program is used to submit necessary MapReduce job at each iteration. This approach has three main disadvantages,

- For each iteration, data must be read from the underlying file system called the Hadoop Distributed Filesystem(HDFS) and results must be written back to the HDFS for next iteration.

- Fault tolerance is usually achieved through data replication which means that replication of terabytes of data at each iterative step wastes significant amounts of precious network bandwidth, I/O, and CPU cycles.

- The de Bruijn graph simplification is achieved through Map, Shuffle and Reduce constructs of MapReduce. The mapper emits the combination of node and edge information as a key. Shuffler sorts all the keys emitted by each of the Mapper to collect the connected node information. Sorting is inherent to the framework and it is a way of identifying interconnected nodes in the graph. This sorted information is sent to a Reducer. The functionality such compression or removal of nodes is performed in the Reducer. Expressing graph simplification in terms of Map, Shuffle and Reduce introduces huge overhead on system resources.

- For example if the graph size is in terms of terabytes then in each iteration Shuffle phase needs to sort Terabytes of data and if the data does not fit in the memory then external sorting algorithms are applied which consumes huge amount of time to complete the entire assembly process.

Although, MapReduce helps to achieve scalability but the performance is compromised. This is evident from the performance comparison results of Contrail and ABySS. A scalable solution to overcome the drawbacks of MapReduce can be implemented using Bulk Synchronous Parallel (BSP) processing model. The overheads of iterative computing needed for parallel graph processing such as synchronization, fault tolerance are automatically handled by the BSP frameworks. Apache Giraph is a BSP based model and we have developed PGA using this framework to process de Bruijn graph in parallel as well as all the algorithms needed to arrive at target genome.

5.4 Methods

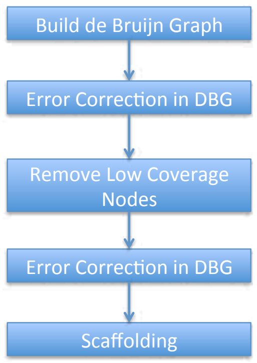


Figure 5.3: de Novo assembly pipeline

PGA's parallel algorithms are built on apache giraph. The primary goal is to reduce the memory requirements for assembly of larger genomes by making use of distributed memory. Figure 5.3 shows the PGA's assembly pipeline. At high level it has five stages (i) Building distributed de Bruijn graph (ii) Error correction which includes compression, tip and bubble removal (iii) Removal of low coverage nodes (iv) Error correction same as stage two and (v) Scaffolding. PGA also follows the same approach used in

Contrail for building the de Bruijn graph by using MapReduce as the parallelization framework. The structure of the graph is similar with some additional parameters and methods. The graph is stored in distributed manner across the cluster of computers. Each partition of the graph is independently processed by the giraph workers. All the graph processing algorithms are implemented in the compute API. In every superstep the workers run the compute method over all the nodes of the partition of the graph assigned to it from the giraph framework. Each worker is allocated a partition of the graph, the size of the partition is such that it can fit in the main memory of the machine. This is the primary reason why PGA requires low memory for assembly.

The next phase after building the de Bruijn graph is error correction. Majority of the existing assemblers employ error correction as an important step since it reduces the graph size significantly and also aides in the scaffolding phase. The billions of short reads resulting from the shotgun sequencing methods consists of errors at certain base positions. There is no confirmed way of knowing which positions in the short reads have errors. However, since each nucleotide position is covered in multiple short reads the incorrect nucleotide results in either tip or bubble in the de Bruijn graph. Error correction state mainly deals with identifying and eliminating such nodes in the de Bruijn gaph. The functional flow of error correction is as shown in figure 5.4.

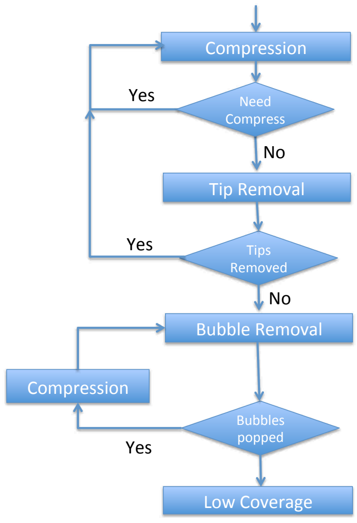


Figure 5.4: Functional flow of error correction in de Bruijn graph

5.4.1 Compression

The first step that follows after building the graph is compressing the linear chain of nodes in the graph. As shown in figure 5.5 the non-branching paths of nodes can be compressed into single node without the risk of misassembly. If there are "p" nodes in the linear path then in every superstep "p/2" nodes are compressed. The algorithm compression in Giraph is as shown in algorithm 1. The algorithm is based on on "Parallel Random List Ranking". In $Superstep0$ each compressible node is tagged as either head or tail with equal probability. In $Superstep1$ the head nodes are compressed with tail nodes and updated information is sent to adjescant nodes. In the subsequent even numbered $Supersteps$, if the nodes to be compressed are remaining then compression phase is repeated. Since, in every repeat of compression p/2 nodes are compressed the complexity of algorithm is $log(p)/w$ where w is the number of workers running in parallel. Furthermore compression is performed at different stages of assembly pipeline, first soon after building the graph, second time after removing the low coverage nodes, third time after tip and bubble removal phases and after every iteration of scaffolding.

There many advantage of compression, it reduces the overall size of the graph after every repeat of compression. This indirectly aides will resolving the ambiguous paths in the de Bruijn graph.

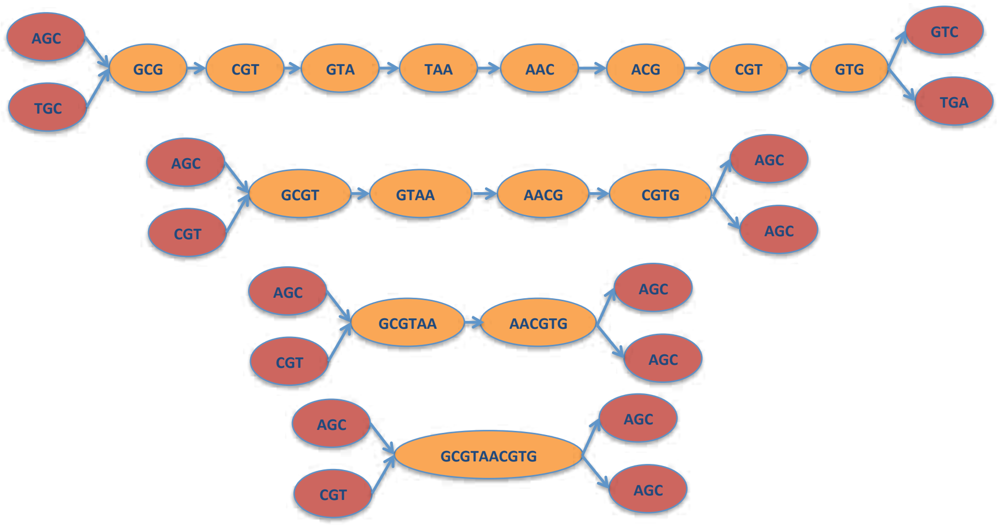


Figure 5.5: An example of linear path compression in assembly graph

```
$Compressible \leftarrow 1$
**while** $Compressible > 0$ **do**
    **if** $Superstep == 0$ **then**
        Mark Nodes to Compress
        Send Message to Neighbors
    **else**
        **switch** $Superstep\%2 == 0$ **do**
            **case** $1$:
                Merge Nodes
                Send Updated Information to Neighbors
            **endsw**
            **case** $0$:
                Mark Nodes to Compress
                $Compressible \leftarrow Remaining$
            **endsw**
        **endsw**
    **end**
**end**
```

Algorithm 1: COMPRESSION OF DE BRUIJN GRAPH USING GIRAPH

5.4.2 Topological Error Correction

In this section we describe the two error correction methods which greatly decrease the size the assembly graph and aid in resolving repeats. Errors are introduced because of sequence technologies produce incorrect nucleotides certain positions is the short reads. Tips and bubbles are the two topological imperfections caused by errors in short reads.

5.4.3 Tip Removal

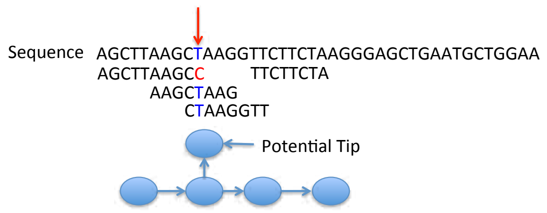


Figure 5.6: Tip formation in de Bruijn Graph

An example of short reads with errors and resulting in tips is shown in figure 5.6. Tips are formed because of errors in the end of the short read. In the example shown, Short reads are produced from the “sequence”. The base “T” is the proper nucleotide at position indicated by arrow. However in one of the short reads the sequencer interprets this as “C”. In all other short reads this nucleotide position is interpreted as “T”. This short read when converted into k-mers forms a tip in the assembly graph. Tips can be a single node or sequence of nodes which are connected at one end and disconnected at the other end. They have only incoming edge but no out going edge. The tip removal is straightforward since dropping edges connecting to tips does not affect the other parts of the graph, however certain restraint is followed to avoid loosing genuine sequence. The general approach followed in other assemblers is to keep the tip if the tip length is greater than 2k. The algorithm to remove tips using Giraph is shown as shown in algorithm 2. In $Superstep$ 0 nodes having an in-degree of one are identified as po-

```
switch $Superstep\%2 == 0$ do
    case 0:
        Nodes with only incoming edges
        Send message to connected node
    endsw
    case 1:
        Mark Nodes to Compress
        $Compressible \leftarrow Remaining$
    endsw
    case 1:
        Mark Nodes to Compress
        $Compressible \leftarrow Remaining$
    endsw
endsw
```
Algorithm 2: TIP REMOVAL IN DE BRUIJN GRAPH USING GIRAPH

tential tips. In $Superstep$ 1 if such node has length less than 2k then it is marked for deletion. In $Superstep$ 2 the parent node which connected to the potential tip updates is edge and removes the edge which is connected to the tip node. Removal of tips leads to linear paths in the graph at certain places. After removing the tips, there will be some linear chain of nodes created. We perform compression after removing the tips as well. Usually compression is performed after removing the tips, if no tips are removed then the next phase of the assembly which is bubble removal is performed.

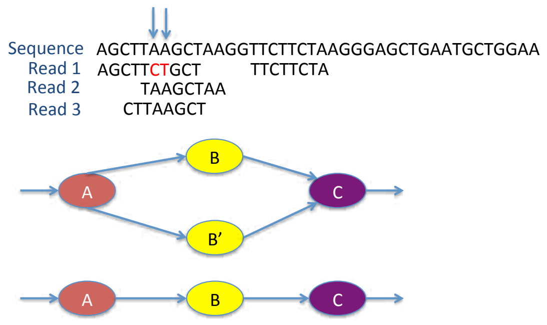

Sequence AGCTTAAGCTAAGGTTCTTCTAAGGGAGCTGAATGCTGGAA
Read 1 AGCTTCTGCT TTCTTCTA
Read 2 TAAGCTAA
Read 3 CTTAAGCT
A
B
B'
C
A
B
C

Figure 5.7: Bubble formation in de Bruijn Graph

5.4.4 Bubble Removal

Bubbles are introduced in the DBG because of errors in the middle of the short reads. Figure 5.7 shows formation of bubble. The two nucleotides “AA” indicated by arrows in the $sequence$ are misinterpreted in $read$ 1 as “CC”, however $read$ 2 and $read$ 3 correctly interpret the two nucleotide positions. The resulting DBG will have two nodes indicated as B and B'. The node B' is created because of k-mers that are produced from the short read $read$ 1. The node B is created from k-mers that are produced from short reads $read$ 2 and $read$ 3. The objective of bubble removal is to identify such nodes in DBG and keep the nodes which are created by the high frequency k-mers, and remove the nodes which have low frequency support. The algorithm 3 describes the implementation of bubble removal in DBG using apache giraph. The algorithm works as follows, the parameter

```
$max\_bubble\_len \leftarrow 5 * kmer\_length$
switch $Superstep == 0$ do
    case 0:
        if $NodeLength < max\_bubble\_len$ then
            Potential bubble
            Send Message to tail node

    endsw
    case 1:
        Find the similarity between node strings
        Node with maximum coverage is retained
    endsw
    case 2:
        Delete potential bubble
        Adjust edge info at head and tail nodes
    endsw
endsw
```

Algorithm 3: BUBBLE REMOVAL IN DE BRUIJN GRAPH USING GIRAPH

max_bubble_len is set to 5*kmer_length. This heuristic is adopted in majority of the assembler tools. Nodes with length less than max_bubble_len, which have in-degree as well as out-degree of one are considered as potential bubbles in the figure 5.7 the nodes B and B' correspond to potential bubbles. In $Superstep$ 0 every node matching this criteria sends message to their tail node. In the figure 5.7, C corresponds to a tail node and node A corresponds to a head node. In $Superstep$ 1 tail nodes which received the messages compute dissimilarity between the node messages, in this case node C compute dissimilarity between nodes B and B'. The dissimilarity between the nodes B and B' is computed using the edit-distance algorithm[GIve reference Here]. This algorithm checks the number of characters in the strings that can be changed to equalize the two strings. The complexity of the edit-distance algorithm is $O(n^2)$ where n refers to the length of the 2 strings. If the dissimilarity between the nodes is within the threshold then one of the node is a bubble. Then, the length of the nodes and, the k-mer frequency associated with both the nodes are compared. The one with higher k-mer frequency is considered as valid node and included in the path between the head node and tail node. In $Superstep$ 2 the node with lower k-mer frequency is purged and edges are adjusted for head and tail nodes. After removing the bubbles, there will be linear paths node. We can compress these nodes without loss of information. After every repeat of bubble removal, compression is performed.

5.4.5 Scaffold Construction

The assembly graph at this stage is error corrected with edges between overlapping contigs. The ambiguity of over- lapping edges can not be removed without the help of mate- pair information. Short read position information coupled with insert length forms the basis for bundling the mate-pair information with assembly graph. The input to the scaffolding phase mate bundled de Bruijn graph, which has the linking information with every node. We developed a graph hop method to find the exact path between the linked nodes. In this method, in superstep 0 each unique node sends message to its neighbor consisting of its node id, the direction of hop and the estimated distance. If the path traversed so far is equal to the estimate distance then current path traversed so far is used to resolve the ambiguity.

```
Input ← Mate bundled DBG if $Superstep == 0$ then
    msg ←node,destination,expected dist, current path
    Send Message to Neighbors(msg)
else
    if Superstep <MAX_HOP then
        Process Message to check if message is destined for current node
        msg ← append current node to incoming message
        Send Message to Neighbors(msg)
    else
        halt computation
    end
end
```

Algorithm 4: HOP METHOD TO FIND THE PATH BETWEEN LINKING NODES

5.5 Results

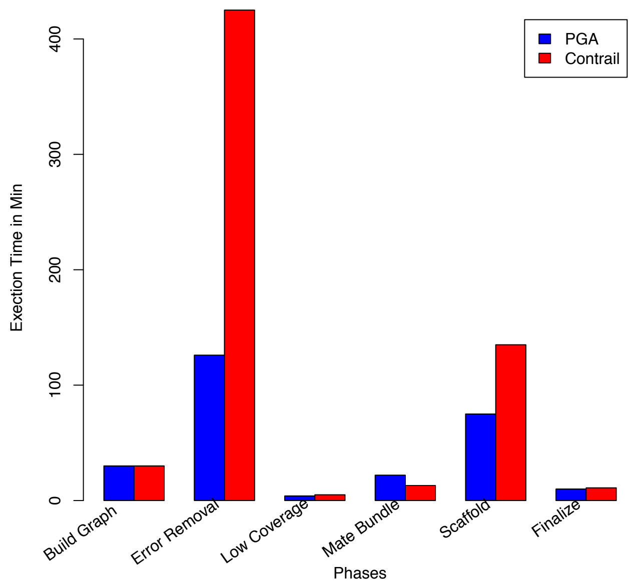


Figure 5.8: Comparing the execution times with every phase of Contrial and PGA

5.5.1 Comparison of PGA with Short Read Assemblers for *E.Coli*

E.Coli data set consists of 20.8 million paired-end, 36 bp Illumina reads with 200 bp insert size (NCBI Short Read Archive, accession no. SRX000429). This data has been assembled by many of the assemblers to evaluate the accuracy and performance. We also consider this dataset to assess PGA against Velvet, Abyss and Contrail. The evaluation was done on a cluster with 8 nodes giving a total of 16 cores and 400 GB local disk. Initial k-mer size of 27 is used for all the evaluations, as the quality of assembly is shown to be optimal with this k-mer size. Experiments were also conducted by varying k-mer sizes but report the results only for k-mer size which gave optimum results. The performance comparisons are shown in the table 5.1. The quality of assembly in terms of N50 length is better than Abyss and is comparable to Contrail. However, in terms of memory requirements Contrail and PGA need smaller memory compared to Abyss.

Furthermore, the memory requirement for PGA can be smaller compared to Contrail. The notion for this inference is that the aggregate memory needed for graph processing using giraph framework is equal to the size of the graph. However, in table 5.1 we have specified the memory for PGA as 4GB, since the graph building phase in PGA is based on MapReduce and this requires more memory for the reduce phase of DGB construction. In terms of the execution speed, The overall assembly was completed in 40.2 minutes which is a performance gain of 69% compared to Contrail. Compared to ABySS we get around 27%. Velvet shows better performance compared to other tools but it has peak memory usage of 32 GB. In general Velvet has better performance for smaller genomes.

5.5.2 Assembly of Thellungiella Salsuginea

In order to show the scalability of PGA we consider a relatively larger genome Thellungiella Salsuginea [57]. It consists of 215 million paired-end, 101 bp Illumina GA II

Table 5.1: Performance comparison results for PGA with short read assemblers for E.Coli data

	Velvet	ABySS	Contrail	PGA
No.of Scaffolds	190	251	273	234
N50	73180	96,308	119782	191103
Mean	55400	37,381	39296	37996
Max	112,000	268,283	236834	237843
Memory	32	8	4	4
Cores	1	16	16	16
Time (Minutes)	20	29	133.3	40.2

Table 5.2: Performance comparison of PGA Short Read Assemblers for Thellungiella Salsuginea dataset

	Velvet	Abyss	Contrail	PGA
No.of Scaffolds	351	410	300	287
N50	18000	16500	15000	15500
Mean	3100	2500	1850	1751
Max	21000	20508	18051	18100
Memory (GB)	256	256	8	8
Cores	1	17	16	16
Time (Min)	333	420	840	312

system reads with 180 insert size (GenBank[1] database accession no. AHIU00000000; PID 80723). Assembly of this genome was performed using the same set of resources that we used for E.Coli dataset. PGA's performance is comparable to AbySS with just 4GB memory which half the memory needed for AbySS. PGA performance is 3X better than Contrail, though both use the same size of memory. Velvet performed better compared to other tools, however required larger physical memory. The assessment and performance results are shown in the Table 5.2.

5.5.3 Assembly of Human Genome Accession no. SRX016231

To demonstrate the scalability of PGA to larger genomes we assemble Yoruban male (NCBI short read archive SRA000271) with accession number SRX016231. This data

[1]`http://www.ncbi.nlm.nih.gov/bioproject/?term=txid72664`

Table 5.3: Performance of PGA for Human Genome

	PGA
No.of Scaffolds	723,040
N50	2056
Mean	1384
Max	20563
Memory (GB)	8
Cores	256
Time (Hours)	10.27

set has read length of 101, Insert length of 500 and 47X coverage. We used k-mer size of 57. The assembly was performed on 128 nodes giving a total of 256 cores and, 8 GB of RAM was allocated to each core. The complete assembly finished in 10.27 hours. Comparing these results with PASHA and ABySS, which took 21.5 and 50.6 hours respectively over 8-node cluster each having 24GB of RAM. PGA is 2X and 5X faster than PASHA and ABySS respectively. The number of node requirement for PGA can also be significantly reduced after the stages of build graph and initial graph compression. We observed that, the initial DBG was of 1.49 TB and after the compression the graph size is significantly reduced around 280 GB. The requirement of 128 nodes is specified considering the initial de de Bruijn graph size of 1.49TB.

5.6 Conclusion

We presented PGA to achieve assembly of larger genomes over commodity hardware. The parallel algorithms are based on apache giraph to take advantage of distributed memory to address the large memory requirements needed to process the de Bruijn graph. Our evaluation using three data sets of different size showed that PGA's performance is comparable to existing tools for smaller genomes and significantly better for larger genomes. Furthermore, PGA's requires small memory per core compared to any of the existing tools.

Chapter 6
Summary of Contributions and Future Work

This dissertation is two-fold. We provided a detailed study on MapReduce as distributed computing framework and overview of ecosystem of tools associated with MapReduce. It also gave overview of the genome assembly process and how MapReduce based apache giraph can be used to accomplish large scale genome assembly.

In the first part of the dissertation, we analyzed the task scheduler in Hadoop [5], which is an open source implementation of MapReduce. Specifically, Hadoop's performance on distributed clusters is evaluated. Though, number of job schedulers exist for Hadoop, the task scheduling is coarse grained and optimized for single cluster deployment. We investigate performance of Hadoop over virtual clusters as well as distributed clusters and proposed methods to optimize task scheduling by including network parameters such as delay, packet loss and bandwidth while making scheduling decisions. The chapters 2 and 3 constitute first part of the thesis. In chapter 2, issues affecting the performance of Hadoop task scheduler are investigated and a method to improve the performance is provided. In chapter 3, the factors degrading the performance of Hadoop over distributed cluster are analyzed and a network aware task scheduler implementation is provided. This work can be extended for optimizing the scheduling of reduce tasks, since the placement of reduce tasks also significantly influence the overall performance.

In the second part of the dissertation, two fundamental applications in Bioinformatics namely sequence alignment and de novo assembly are addressed. Both the applications involve processing of billions of short reads generated from NGS technologies. The size of short read data sets produced from NGS is ever increasing and requires a scalable distributed computing framework to store, analyze and process to construct genomes of the order of human genome. The goal of the second part of the thesis was to develop scalable algorithms for indexing reference genome and to develop de novo genome assembly tool by using MapReduce based ecosystem of tools. The chapters 4 and 5 make second part of the thesis. In chapter 4, scalable, distributed algorithms for constructing the suffix tree to index larger genomes is proposed. In chapter 5 describes why genome assembly is Big data challenge and why existing assembly methods are inefficient for assembly of genomes of the order of human are investigated. We propose parallel giraph based Novo genome assembler (PGA), a tool to achieve assembly of larger genomes over small computing environment.

PGA can be extended to the assembly of metagenomes. The field of metagenomes deals with assembly of community of species by sampling the DNA of several organisms. The volumes of the short reads generated from sequencing of metagenomes is in terms of terabytes. Which makes assembly of metagenomes a more complex challenge than assembly of single genome. Furthermore, the throughput of NGA sequencing is increasing fast, indicating that only distributed analysis systems can help bridge the gap of scalability.

Bibliography

[1] 454 Life Science Sequencing Technology. http://www.454.com/.

[2] Amazon elastic compute cloud. http://aws.amazon.com/ec2/.

[3] Amazon Elastic MapReduce. http://aws.amazon.com/elasticmapreduce/.

[4] apache giraph. http://giraph.apache.org/.

[5] Apache Hadoop. http://hadoop.apache.org/.

[6] Autodock tool for docking. http://autodock.scripps.edu/.

[7] Cloudera Hadoop. http://www.cloudera.com/content/cloudera/en/home.html.

[8] De Novo Assembly of Large Genomes Using Cloud Computing.

[9] Dummynet software emulator. http://info.iet.unipi.it/luigi/dummynet/.

[10] Futuregrid compute cluster. https://portal.futuregrid.org/.

[11] IBM Blue Cloud Computing Platform. http://www.ibm.com/cloud-computing/us/en/.

[12] Illumina's Genome Analyzer. http://www.illumina.com/systems/genomeanalyzeriix.ilmn.

[13] LONI. http://www.loni.org.

[14] Neuca Eucalyptus Cloud Computing Software. https://geni-orca.renci.org/trac/wiki/Eucalyptus-2.0-Setup/.

[15] Openstack cloud computing platform. http://www.openstack.org/.

[16] Pinger tool . http://www-iepm.slac.stanford.edu/monitoring/general.html.

[17] Rackspace cloud computing platform. http://www.rackspace.com/.

[18] Xsede Computation Grid. https://www.xsede.org/.

[19] CRON Project: Cyberinfrastructure for Reconfigurable Optical Networking Environment, 2011. http://www.cron.loni.org/.

[20] ACM. *Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling*, 2010.

[21] Marina Barsky, Ulrike Stege, Alex Thomo, and Chris Upton. Suffix trees for very large genomic sequences. In David Wai-Lok Cheung, Il-Yeol Song, Wesley W. Chu, Xiaohua Hu, and Jimmy J. Lin, editors, *CIKM*, pages 1417–1420. ACM, 2009.

[22] Serafim Batzoglou, David B. Jaffe, Ken Stanley, Jonathan Butler, Sante Gnerre, Evan Mauceli, Bonnie Berger, Jill P. Mesirov, and Eric S. Lander. ARACHNE: A Whole-Genome Shotgun Assembler. *Genome Research*, 12(1):177–189, January 2002.

[23] Srikanta J. Bedathur and Jayant R. Haritsa. Engineering a fast online persistent suffix tree construction. In *ICDE*, pages 720–731, 2004.

[24] Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid, 2000.

[25] Michael Cardosa, Chenyu Wang, Anshuman Nangia, Abhishek Chandra, and Jon Weissman. Exploring mapreduce efficiency with highly-distributed data. In *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce '11, pages 27–34, New York, NY, USA, 2011. ACM.

[26] Ching-Fung Cheung, Jeffrey Xu Yu, and Hongjun Lu. Constructing suffix tree for gigabyte sequences with megabyte memory. *IEEE Trans. Knowl. Data Eng.*, 17(1):90–105, 2005.

[27] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI 2004*, pages 137–150.

[28] Zhaohui Ding, Xiaohui Wei, Yuan Luo, Da Ma, Peter W. Arzberger, and Wilfred W. Li. Customized plug-in modules in metascheduler csf4 for life sciences applications. *New Generation Comput.*, 25(4):373–394, 2007.

[29] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.

[30] Amol Ghoting and Konstantin Makarychev. Serial and parallel methods for i/o efficient suffix tree construction. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 827–840, New York, NY, USA, 2009. ACM.

[31] Ramesh Hariharan. Optimal parallel suffix tree construction. In *STOC*, pages 290–299, 1994.

[32] Ela Hunt, Malcolm P. Atkinson, and Robert W. Irving. A database index to large biological sequences. In *In VLDB*, pages 139–148, 2001.

[33] Lucian Ilie, Farideh Fazayeli, and Silvana Ilie. Hitec: accurate error correction in high-throughput sequencing data. *Bioinformatics*, 27(3):295–302, 2011.

[34] Eucalyptus Systems Inc. Installing eucalyptus 2.0. http://open.eucalyptus.com/wiki/EucalyptusInstallationv2.0.

[35] Hui Jiang and Wing Hung H. Wong. SeqMap: mapping massive amount of oligonucleotides to the genome. *Bioinformatics (Oxford, England)*, 24(20):2395–2396, October 2008.

[36] Gad M. Landau, Baruch Schieber, and Uzi Vishkin. Parallel construction of a suffix tree (extended abstract). In Thomas Ottmann, editor, *ICALP*, volume 267 of *Lecture Notes in Computer Science*, pages 314–325. Springer, 1987.

[37] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, 10(3):R25–10, March 2009.

[38] R Li, H Zhu, J Ruan, W Qian, X Fang, Z Shi, Y Li, S Li, G Shan, K Kristiansen, S Li, H Yang, J Wang, and J Wang. De novo assembly of human genomes with massively parallel short read sequencing. *Genome Res*, 20:265–272, 2010.

[39] Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang 0004. Soap: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008.

[40] Yuan Luo, Zhenhua Guo, Yiming Sun, Beth Plale, Judy Qiu, and Wilfred W. Li. A hierarchical framework for cross-domain mapreduce execution. In *Proceedings of the second international workshop on Emerging computational methods for the life sciences*, ECMLS '11, pages 15–22, New York, NY, USA, 2011. ACM.

[41] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale

graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[42] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The macroscopic behavior of the tcp congestion avoidance algorithm. *SIGCOMM Comput. Commun. Rev.*, 27(3):67–82, July 1997.

[43] Andréa Matsunaga, Maurício Tsugawa, and José Fortes. Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, ESCIENCE '08, pages 222–229, Washington, DC, USA, 2008. IEEE Computer Society.

[44] Edward M. Mccreight. A Space-Economical Suffix Tree Construction Algorithm. *J. ACM*, 23(2):262–272, 1976.

[45] 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, New York, NY, USA, 2011. ACM.

[46] Eugene W. Myers, Granger G. Sutton, Art L. Delcher, Ian M. Dew, Dan P. Fasulo, Michael J. Flanigan, Saul A. Kravitz, Clark M. Mobarry, Knut H. J. Reinert, Karin A. Remington, Eric L. Anson, Randall A. Bolanos, Hui-Hsien Chou, Catherine M. Jordan, Aaron L. Halpern, Stefano Lonardi, Ellen M. Beasley, Rhonda C. Brandon, Lin Chen, Patrick J. Dunn, Zhongwu Lai, Yong Liang, Deborah R. Nusskern, Ming Zhan, Qing Zhang, Xiangqun Zheng, Gerald M. Rubin, Mark D. Adams, and J. Craig Venter. A whole-genome assembly of drosophila. *Science*, 287(5461):2196–2204, 2000.

[47] K. L. Patrick. Illuminating the future of genome sequencing and personalized medicine. *Yale J Biological Med*, 80, 2007.

[48] P. A. Pevzner, H. Tang, and M. S. Waterman. An eulerian path approach to dna fragment assembly. *Proc Natl Acad Sci U S A*, 98(17):9748–9753, August 2001.

[49] Benjarath Phoophakdee and Mohammed J. Zaki. Genome-scale disk-based suffix tree indexing. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 833–844, New York, NY, USA, 2007. ACM.

[50] Catherine Shaffer. Next-generation sequencing outpaces expectations. *Nat Biotechnol*, 25(2):149, February.

[51] J Simpson, K Wong, S Jackman, J Schein, S Jones, and I Birol. Abyss: a parallel assembler for short read sequence data. *Genome Res*, 19:1117, 2009.

[52] G. G. Sutton, O. White, M. D. Adams, and Ar Kerlavage. TIGR Assembler: A new tool for assembling large shotgun sequencing projects. 1:9–19+, 1995.

[53] Yuanyuan Tian, Sandeep Tata, Richard A. Hankins, and Jignesh M. Patel. Practical methods for constructing suffix trees. *VLDB J.*, 14(3):281–299, 2005.

[54] Esko Ukkonen. On-line construction of suffix trees, 1995.

[55] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[56] Peter Weiner. Linear pattern matching algorithms. In *SWAT (FOCS)*, pages 1–11. IEEE Computer Society, 1973.

[57] Hua-Jun Wu, Zhonghui Zhang, Jun-Yi Wang, Dong-Ha Oh, Maheshi Dassanayake, Binghang Liu, Quanfei Huang, Hai-Xi Sun, Ran Xia, Yaorong Wu, Yi-Nan Wang, Zhao Yang, Yang Liu, Wanke Zhang, Huawei Zhang, Jinfang Chu, Cunyu Yan, Shuang Fang, Jinsong Zhang, Yiqin Wang, Fengxia Zhang, Guodong Wang, Sang Yeol Lee, John M Cheeseman, Bicheng Yang, Bo Li, Jiumeng Min, Linfeng Yang, Jun Wang, Chengcai Chu, Shou-Yi Chen, Hans J Bohnert, Jian-Kang Zhu, Xiu-Jie Wang, and Qi Xie. Insights into salt tolerance from the genome of thellungiella salsuginea. *Proc Natl Acad Sci U S A*, 109(30):12219–24, 2012.

Vita

Praveenkumar Kondikoppa was born in Karnataka, India, in 1982. He obtained his bachelors degree in Electronics and Communications in 2004 from Visvesvaraya Technological University (VTU), Belgaum. During his doctoral studies at Louisiana State University, he has worked on interdisciplinary research projects which included improving the performance of distributed computing framework and de Novo genome assembly. During the last few years of is Ph.D. he has worked as research assistant at center for computation and technology (CCT).