Analyzing the Programmability and the Efficiency of Large-Scale Graph Processing on

A Project Report

Submitted in partial fulfillment of the requirements for the award of the degree

of

Bachelor of Technology

in

Computer Science and Engineering

Guided By:

Dr. R.C. Joshi

Submitted By:

Dishant Ailawadi **Gautam Seshadri** **Milan Kumar Mohapatra**

Department Of Electronics and Computer Engineering
Indian Institute of Technology Roorkee
Roorkee – 247 667 (India)

May, 2011

Students' Declaration

We hereby that the work being presented in this project entitled, "Analyzing the programmability and efficiency of large scale graph processing on Hadoop", in partial fulfillment of the requirement for the degree of Bachelor of Technology in Computer Science and Engineering, submitted in the department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, is an authentic record of our original work carried out under the guidance of Dr. R C Joshi, Professor, IIT Roorkee.

Further, we declare that this work hasn't been submitted for award of any other degree.

Dishant Ailawadi Gautam Seshadri Milan Kumar Mohapatra

Supervisor's Certificate

This is to certify that this project is a result of original and sincere efforts made by the above candidates. All the statements made by the candidates are correct to the best of my knowledge and belief.

Dr. R C Joshi
Professor
Department of Electronics and Computer Engineering,
Indian Institute of Technology Roorkee

Acknowledgement

We would like to thank our project guide and mentor Dr. R C Joshi for all his help and support. He has helped us in our project by giving us a clear focus to work on the problem right from day one and by helping us overcome difficulties through encouragement and guidance. His impact on the work in immeasurable.

We want to thank our senior Mr. Akhil Langar for allowing us to access through his account, the Hadoop Cluster Testbed provided by University of Illinois, Urbana Champaign.

We would also like to thank the department of Electronics and Computer Engineering and the Institute Computer Centre at IIT Roorkee for providing us with the necessary facilities for developing and demonstrating our project.

Abstract

Graphs are analyzed in many important contexts like page rank, protein-protein interaction networks, and analysis of social networks. Many graphs of interest are difficult to analyze because of their large size, often spanning millions of vertices and billions of edges. We believe that MapReduce has emerged as an enabling technology for large-scale distributed graph processing. Its functional abstraction provides an easy-to-understand model for designing scalable, distributed algorithms. The open-source Hadoop implementation of MapReduce has provided researchers with a powerful tool for tackling large-data problems.

In this work, we analyze the programmability and efficiency of implementing large scale graph algorithms on Hadoop using the MapReduce model. We have taken the problem of finding the minimum spanning tree of a large graph, which is an important building block for many graph algorithms. Minimum Spanning Tree algorithm, comprising of three stages i.e. Find-Min, Connect-Component and Merge, has been implemented with many variants in each stage. A thorough performance analysis using the Wikipedia Dataset consisting of 8.8 million nodes and 0.9 billion edges, highlighting the most successful combination of our techniques, has been carried out. Thus, it has been shown that useful graph operations can be decomposed into MapReduce steps and that the power of the cloud might be brought to large graph problems as well.

Contents

List of figures

Chapter 1: INTRODUCTION AND STATEMENT OF THE PROBLEM

1.1 Introduction

Large graphs are ubiquitous in today's information-based society and graphs are analyzed in many important contexts, including ranking search results based on the hyperlink structure of the World Wide Web, Computer Networks, mobile call networks, module detection of protein-protein interaction networks, and extracting interesting features from social networks. Typical graph mining algorithms assume that the graph fits in the memory of a typical workstation, or at least on a single disk, whereas the above mentioned graphs are difficult to analyze because of their large size, often spanning millions of vertices and billions of edges. As such, researchers have increasingly turned to distributed solutions. In particular, MapReduce has emerged as an enabling technology for large-scale graph processing and if useful graph operations can be decomposed intoMapReduce steps, the power of the cloud might be brought to large graph problems as well.

1.2 Statement for our Project Problem

The main objective of our project can be stated as follows:

a. To implement large scale algorithm like Minimum Spanning Tree algorithm on Hadoop’s MapReduce framework
b. To analyze the programmability and efficiency of the implementation

1.3 Scope of Cloud Based Computing with Hadoop

Using Cloud Computing as a technology has definitely become popular and appealing within the research community as well as in the industries.

A cloud is a large collection of commodity computers, each with its own disk, connected through a network. [1] Distributed processing on a cloud has enjoyed much recent attention. Hadoop helps in hosting MapReduce jobs, which do their work by sequencing through data stored on disk. The technique increases scale by having a large number of independent but loosely synchronized computers running their own instantiations of the MapReduce job components on their data partition.

There are indeed potential benefits of using the high-performance scientific computing on clouds. The common aspect for many important problems [2], including those in machine learning, social networking analysis, and business intelligence, is the need to analyze enormous graphs. The Web consists of trillions of interconnected pages, IMDB has millions of movies and movie stars, and sequencing a single human genome requires searching for paths between billions of short DNA fragments. At this scale, searching or analyzing a graph on a single machine would be time-consuming and may even be impossible, especially when the graph cannot possibly be stored in memory on a single computer. Fortunately, Hadoop and MapReduce Model can enable us to tackle the largest graphs around by scaling up many graph algorithms to run on entire clusters of machines.

1.4 Organization of our Report

Chapter 2 of our report gives an overview on the basics of the Hadoop and MapReduce framework with the help of a sample graph-processing problem on MapReduce Model. It also discusses the previous works done in areas related to Graph Processing on Hadoop and MapReduce.

Chapter 3 discusses the fundamentals of the Minimum Spanning Tree algorithm that we have implemented, the Test-bed used for Performance Analysis and the Test data used.

Chapter 4, 5 and **6** discuss the Find-Min Stage, Connect-Component Stage and the Merger Stage, respectively, in detail where each chapter explains their design, implementation, performance analysis and their bottlenecks and further scope for improvement.

Chapter 7 gives the conclusion and future scope of the project which is followed by **Bibliography**.

Chapter 2: OVERVIEW

This chapter discusses the Hadoop Distributed File System and working of MapReduce model in detail with the help of a sample graph processing algorithm. A brief overview of the existing Minimum Spanning Tree Algorithm (Boruvka's) is given and subsequent sub-sections discuss our approach towards the MST algorithm, the test-bed and the test data used in the project.

2.1 Hadoop and Hadoop Distributed File System

Hadoop is a free distribution open-source framework based on Java by Apache Group and funded by Yahoo! for running applications in reliable, scalable and distributed computing. [3]

The Hadoop implements a computational paradigm named MapReduce, where the application is divided into many small fragments of work, each of which may be executed or re-executed on any node in the cluster. Hadoop is designed to efficiently process large volumes of information by connecting many commodity computers together to work in parallel.

In a Hadoop cluster, data is distributed to all the nodes of the cluster as it is being loaded in. The Hadoop Distributed File System (HDFS) will split large data files into chunks which are managed by different nodes in the cluster. In addition to this each chunk is replicated across several machines, so that a single machine failure does not result in any data being unavailable. Even though the file chunks are replicated and distributed across several machines, they form a single namespace, so their contents are universally accessible.

HDFS is designed to be robust to a number of the problems that other DFS's such as NFS are vulnerable to. In particular:

- HDFS is designed to store a very large amount of information of the order of terabytes or petabytes. This requires spreading the data across a large number of machines.

- It stores data reliably. If individual machines in the cluster malfunction, data should still be available.
- It provides fast, scalable access to this information. It is quite possible to serve a larger number of clients by simply adding more machines to the cluster.
- It integrates well with Hadoop MapReduce, allowing data to be read and computed upon locally when possible.

But while HDFS is very scalable, its high performance design also restricts it to a particular class of applications.

2.2 MapReduce Model

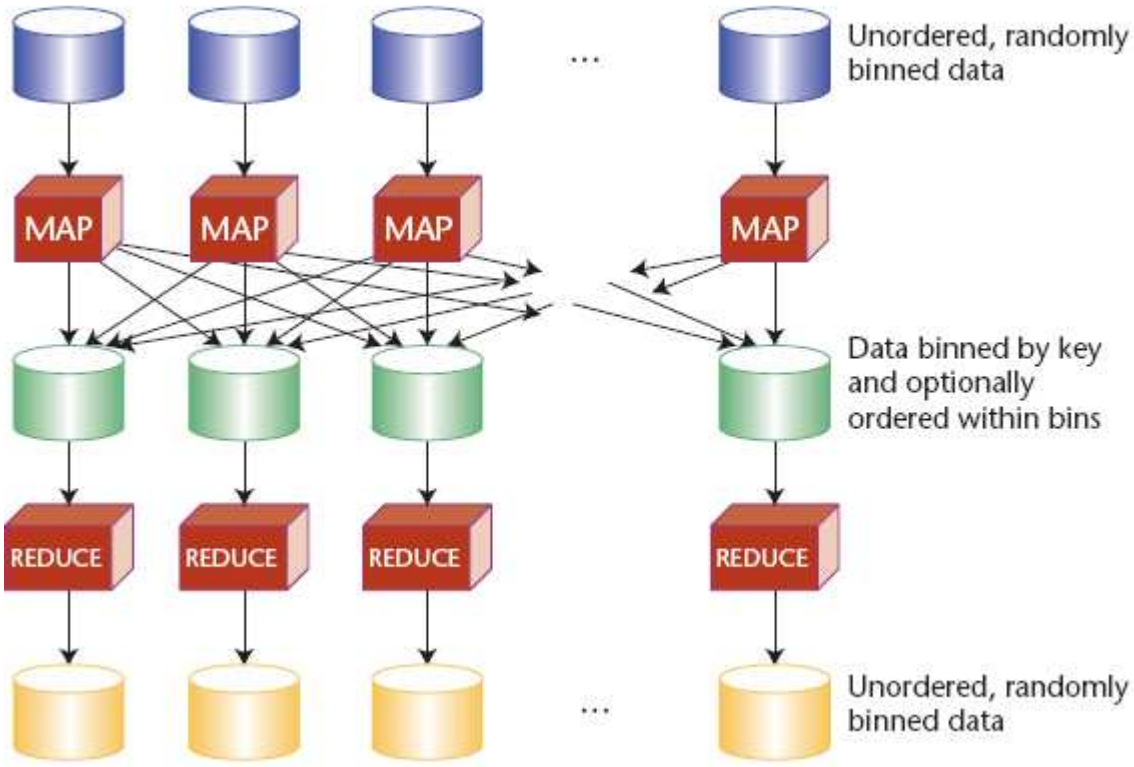

Figure 2.1: A MapReduce job.

A record consists of a key and a value. The Mapper's job is to create some number of records in response to each input record. Records presented to the reducer are binned by key, such that all records with a given key are presented as a list to the reducer. The reducer then examines the list sequentially through an iterator.

Figure 2.1 outlines a MapReduce job's function. A job operates on an input file (or more than one) distributed across different Keys and produces an output file also distributed across different keys. The system feeds input-file records in the form of Key-Value to the job's mapper instances and partitions the mapper instances' outputs globally by key into bins, producing an intermediate file. It then feeds each of these intermediate bins to a reducer instance; the reducers then produce the job's output file.

In particular, the map seeks to key its outputs so that the system places in the same bin the records that should come together in the reduce phase. To perform some of the reduction early, thereby lessening the need to store and transport records, we can also specify a combiner that operates on the mapper's output while the mapper is running.

Every job must specify both a Mapper and a Reducer, but either of them can be the identity. Although a user can employ MapReduce on a single machine, MapReduce frameworks are designed to support operation on a cloud of computers. The Hadoop system then implements a single MapReduce job as parallel mapper and reducer instances running concurrently on different computers. A simplification that MapReduce brings to parallel computing is that the only synchronization takes place when creating and accessing files. The problem here is that shared state, beyond the files, is limited.

Hadoop, implemented in Java, runs each Mapper or Reducer instance in an independent virtual machine on each computer. Consequently, the "static" fields defined in each Java class aren't static across instances. What the algorithms do use is Hadoop's facility for passing parameters to the MapReduce job environment and for each mapper or reducer instance to contribute to counts accumulated across all mapper and reducer instances. More specifically, when mapper and reducer instances are created, they're initialized with a copy of the job input parameters.

2.3 A Sample graph processing problem on MapReduce

Here, we describe a sample graph processing problem for a better understanding of the MapReduce model. This algorithm augments the edges of the graph with degree information.

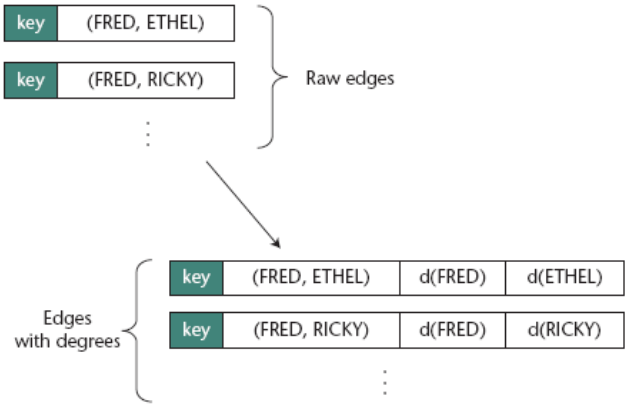

Figure 2.2: Records for representing vertex degrees.

In the figure above, the input file comprises records whose values are edges. The output file has edges augmented with degree information.

The first Map-Reduce process with will append the records with vertex-degree information, creating a new file in which the edges will have the degree info of the vertices also. Creating the desired output requires three passes through the data: one map and two reduces.

Figure 2.3a summarizes the first MapReduce job. For each edge record the mapper reads, the map emits two records, one keyed under each of the vertices that form the edge. This process will create many vertex bins corresponding to vertices such that each bin will hold records for every edge adjacent to its associated vertex. The reduce phase works on each such bin in turn. Having read the bin's contents, the reducer knows the vertex degree, which is equal to the number of records in the bin. The reducer can now emit each of the edge that was there in the bin with half the information regarding the degree of the vertex associated with that bin. As this is only half of the degree information; another reducer, or another call to the same reducer, produces the other half. The reducer keys output records by the edge so that the two halves of each edge's vertex information can come together in the next Reducer phase.

Figure 2.3b summarizes the second MapReduce job, which employs an identity Mapper. Each bin then collects the partial degree information, one record for each vertex in the edge. The reducer combines these records to produce a single record for the edge containing the desired degree information.

Thus this completes the Augmenting-Edges-With-Degree-Info part.

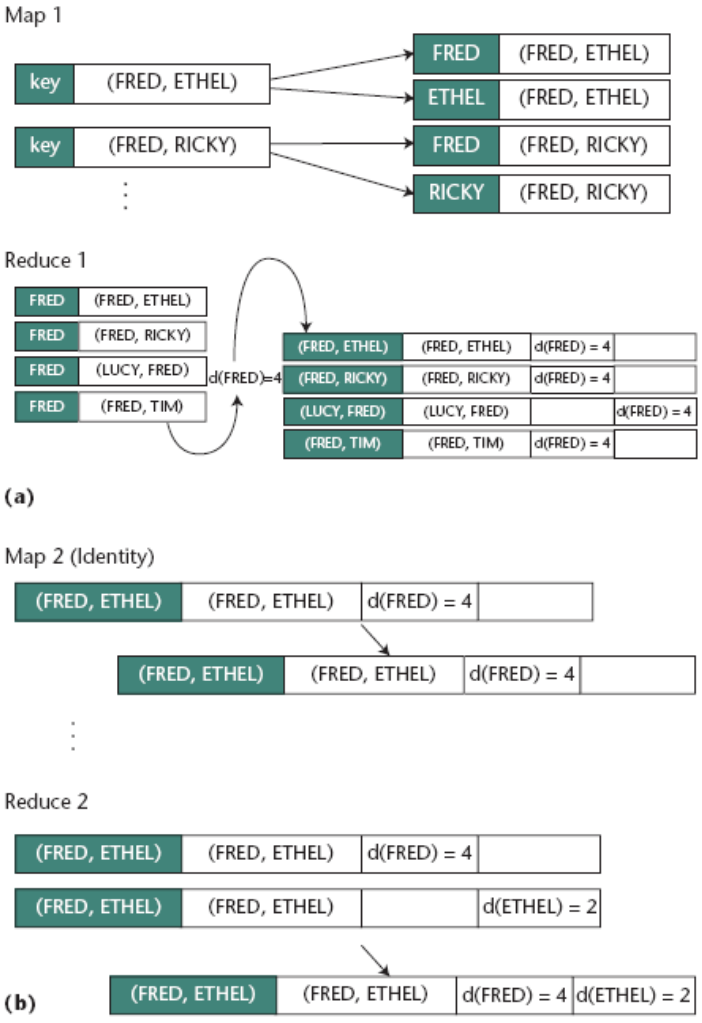

Figure 2.3.Augmenting edges with degrees.

2.4 Related Work

2.4.1 Related Work in Graph Processing on Hadoop

Recently, in the last years a number of graph algorithms on MapReduce and Hadoop have been published. Large-scale Graph-processing on Hadoop have been presented in [7,8,9].In [1], Cohen has given some MapReduce implementations of useful graph operations in a simple form, from which we tried to develop the Edge-Degree augmentation problem, thus getting an idea of how to decompose graph-operations into Map-Reduce steps. In [8], Kang et.al have described how to implement many graph mining operations such as PageRank, spectral clustering, diameter estimation, connected components in the form of a repeated matrix-vector multiplication. Using the idea presented to implement Connect-Component here, we have tried to modify the algorithm to suit our implementation and optimized on performance and also, we got inspiration on how to carry out Block- Structured Connect-Component. Jimmy-Lin[2] has provided a host of Hadoop-optimizations like Swimmy, In-Mapper Combiner and Range Partitioner, We have carried out In-Mapper combiner in our analysis also and also presented the feasibility of other operations also.

2.4.2 Related Work in MapReduce Model and the MST algorithm

In the initial phase of our project, we read some Hadoop related papers for having a proper in-depth understanding of the MapReduce model and how it works[10,11].Various MST algorithms based on shared-memory and distributed- memory approach exist in literature, though these approaches cannot be mapped effectively onto Map-Reduce Framework. But there are some which has given has insight on how to carry out MST implementation efficiently.[4] gave us a good implementation of the MST algorithm on the shared memory architecture. [12] provided us with a insight into the parallel implementation of MST algorithm using Prim’s algorithm.

Chapter 3: PROJECT DETAILS

This chapter discusses the fundamentals of the Minimum Spanning Tree algorithm that we have implemented, the Test-bed used for Performance Analysis and the Test data used.

3.1 Boruvka MST Algorithm

Boruvka's MST algorithm lends itself more naturally to parallelization,since other approaches like Prim's and Kruskal's are inherently sequential, with Prim's growing a single MST one branch at a time, while Kruskal's approach scans the graph's edges in a linear fashion.[4] Three steps comprise each iteration of parallel Boruvka's algorithm:

- *find-min*: for each vertex *v* label the incident edge with the smallest weight to be in the MST.
- *connect-components*: identify connected components of induced graph with edges found in Step 1.
- *compact-graph*: compact each connected component into a single super-vertex, remove self-loops and multiple edges; and re-label the vertices for consistency.

Steps 1 and 2 (find-min and connect-components) are relatively simple and straightforward;

Step 3 (compact-graph) shrinks the connected components and relabels the vertices. For sparse graphs this step often consumes the most time. In the following section, we describe our implementation of spanning tree generation using Boruvka approach.

3.2 Implementation of Minimum Spanning Tree algorithm

The Minimum Spanning Tree algorithm for Hadoop which we have proposed has 5 MapReduce Steps which must be iterated to some number of iterations before the Complete MST of the graph can be found. In each iteration of MST, we do the following:

- *A Find-Min MapReduce*
- *Connect-components MapReduce*: which is an iteration of 1-step MapReduce process.
- *Merger MapReduce*: 3-step Merger MapReduce process

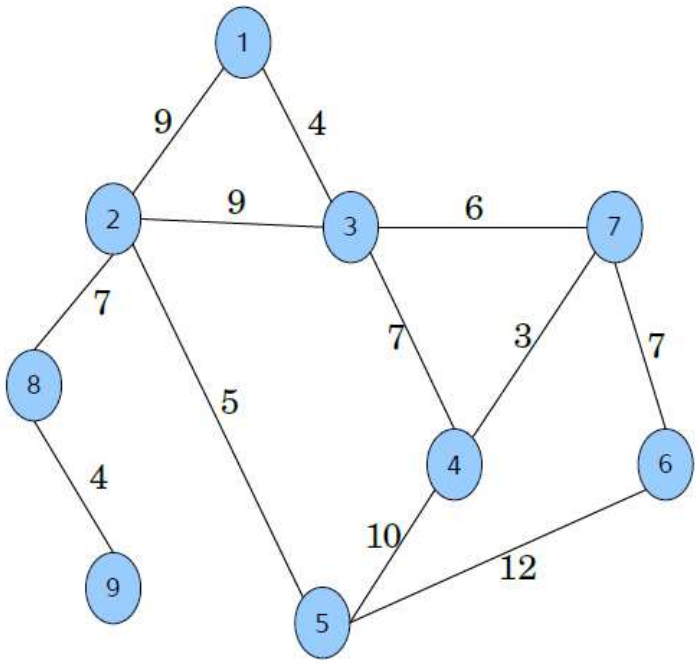

Figure 3.1: The Input Graph used for illustrating our MapReduce Implementation of MST. We have shown the figures for only the first iteration of our MST implementation.

3.3 Test Data and the Cluster Used for Performance Analysis

We have carried out our Performance Analysis of the Hadoop code on The Illinois Cloud Computing Testbed (CCT) which is the world's *first cloud testbed* aimed at supporting both systems innovation and applications research.[5]

CCT is unique in several respects: (1) it is a true cloud/datacenter testbed, e.g., its storage to computation ratio is different from that of many existing testbeds such as Emulab and PlanetLab). Currently, CCT is configured with about 500 TB of shared storage and 1000+ shared cores; (2) it is the only cloud testbed to support both applications and systems research (in contrast with the Google-IBM testbed, for example).

Given below is the specification of the Cluster:

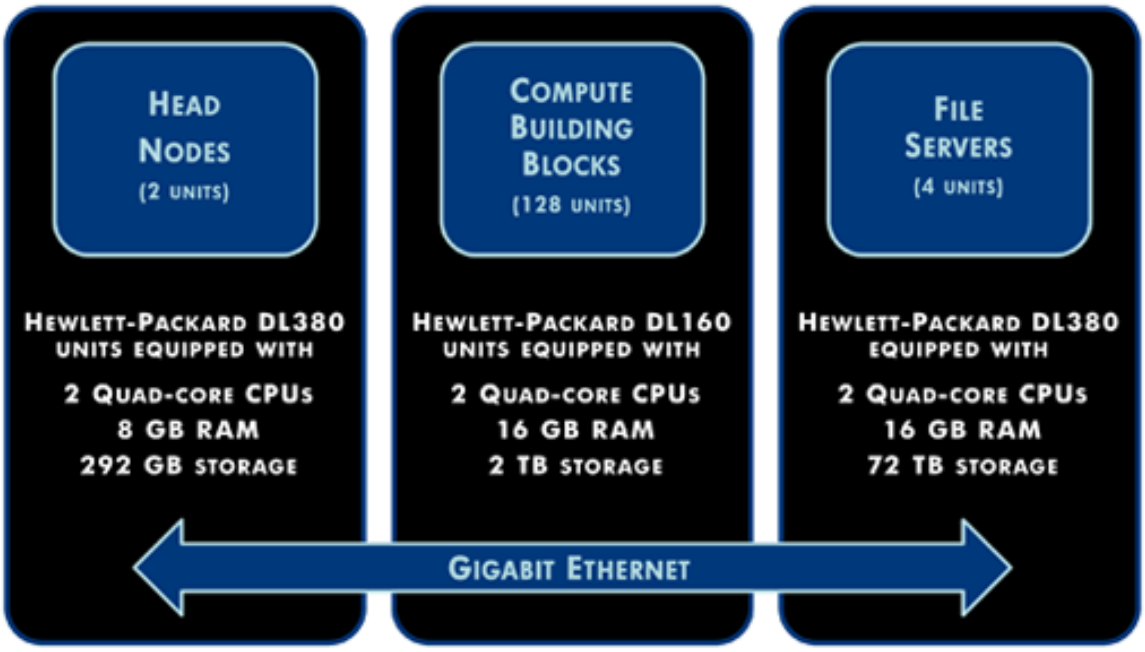

Figure 3.2: Specification of the test bed used. This Test Bed was provided by University of Illinois, Urbana- Champaign.

Initially the Link-graph is of the following form:

1 3 4

2 1

3

4 2 3

This represents a graph with four nodes: nodeId 1 points to 3 and 4; nodeId 2 points to 1, nodeId 3 is a dangling node (no neighbors); and nodeId 4 points to nodes 2 and 3.

We have worked on Wikipedia Dataset which is among the most popular datasets to process with Hadoop: it's big, it contains a lot of text, and it has a rich link structure. And most important of all, Wikipedia can be freely downloaded by anyone. The online material available on UMD Cloud9[6] has provided code for extracting the link graph from Wikipedia and contain instructions for packing Wikipedia pages from the raw XML distribution into block-compressed Sequence Files for convenient access. Once this is done, given below is the invocation for extracting the link graph in the above text format:

*hadoop jar cloud9.jar edu.umd.cloud9.collection.wikipedia.BuildWikipediaLinkGraph *
*-libjars bliki-core-3.0.15.jar,commons-lang-2.5.jar *
*/user/jimmy/Wikipedia/compressed.block/en-20101011 *
/user/jimmy/Wikipedia/edges /user/jimmy/Wikipedia/adjacency 10

The command-line arguments, in order: directory of Wikipedia source, directory for storing edge information, directory for storing adjacency lists (in the above text format), and number of partitions.

Once we have got this Adjacency List, this list has to be transformed into a symmetric Adjacency List along with weights. Random Integer weights have been assigned to the edges ranging from 0 to 1000. For doing this, a MapReduce program has to be written which converts this list into a symmetric Adjacency List with weights. This new list serves as the initial input to our MST Hadoop Implementation.

Chapter 4: FIND-MIN STAGE

This chapter discusses the design and implementation of Find-Min stage in detail. It also discusses the performance analysis, bottlenecks and further scope of improvement.

4.1 Introduction

The first stage of our MST iteration is the Find-Min step. This step involves finding for each vertex, the edge with minimum weight emanating from it in the graph. This step is computationally not very intensive, but serves as a base for the other, more challenging stages to follow. So, a careful consideration of the data structures and input formats to be used, have been done in this stage.

4.2 Our MapReduce Approach

In the Mapper Stage, the minimum-weight outgoing edge from each vertex V (let it be NodeA) is found and two Key-Value pairs (V, NodeA) and (NodeA, V) are emitted for each map.

For example, suppose we have an adjacency list (V-> (NodeA, Edge-Weight)):

*7->{(2,5),(3,7),(6,12),**(4,2)**}.*

The minimum edge is the (7-4) edge whose weight is 2.Two key-value pairs are created and sent to both Nodes 7 and 4.

In the reducer stage, for a given Key, the Value is a list of Min-Edges that have the vertex with Id Key as one of the vertices. We build up a Secondary Adjacency for node-Key and remove duplicate edges (the case when both vertices of an edge have that edge as a Min weight edge).

We also emit edges of the secondary adjacency list as part of Spanning Tree. To prevent two reducers from emitting same edge twice, the condition that smaller Vertex of edge should emit is used. Thus,

the key-Value collected from Reducer is the (NodeId, ListOfMinEdges).

In-Mapper Combiner strategy.

We can prevent so many Key-Value pairs from using the network traffic during the shuffling phase between Map and Reduce by building up a temporary secondary-adjacency list during the Mapper Task. A Hash-Map from Vertices to their secondary-adjacency lists is created during the starting of Mapper Tasks and all map calls from the Input-Split will build up this Hash-Map. After completing the Mapper-Task, we can emit all Key-List pairs from the Hash-Map. In this Case, the Reducer combines the list of various temp-secondary lists for the same Key and emits a Full-Secondary-List.

Here, instead of sending the full adjacency list onto the next stage, the list is broken down and the information is sent to each of the node separately. To illustrate, suppose the final list is:

4-> (2, 6, 8).

Instead of sending the whole list, we send the initial SuperVertex of 4 to nodes 2, 6, and 8 individually. This step helps in optimizing the Connect-Components Stage which is explained in the next chapter.

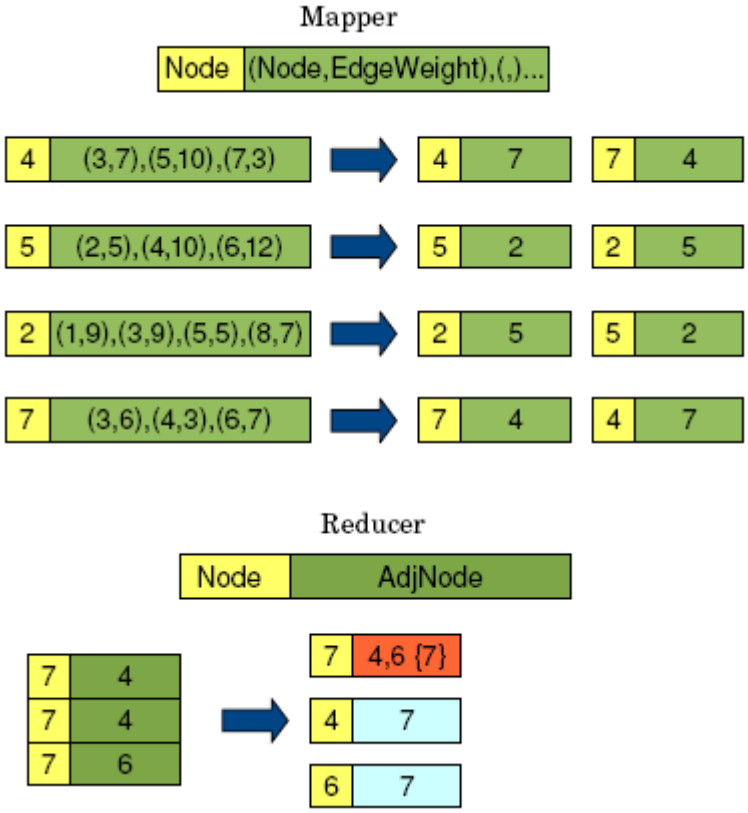

Figure 4.1: The FindMin Stage

In the above figure, FindMin Stage is illustrated. The Mapper operates on the Adjacency List of a Node and emits the edge with Minimum weight as a Key-Value pair. The Reducer constructs a secondary Adjacency List of a Node consisting of all MinEdges and emits the List and sends the initial SuperVertex number of the Node to its neighbors.

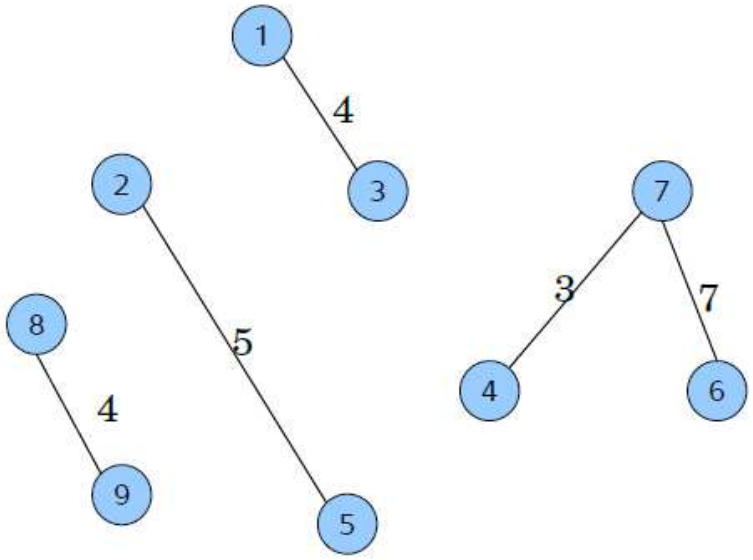

Figure 4.2: The Minimum Weight Edges found in Find-Min Stage using the original graph shown in Figure 4.1.

4.3 Performance Analysis

We carried out the analysis of Find-Min stage using with and without the In-Mapper Combiner technique.

Number of Reducer Tasks	Normal Find-Min (Time in seconds)	With In-Mapper Combiner (Time in seconds)
64	57	65
92	50	53
128	62	64
192	61	66
256	71	72
384	126	124
Number of Records Emitted from Mapper	17632054	16940451

Table 1: The table shows the running times of Find-Min Stage of first iteration for varying number of Reducer Tasks

We have varied the number of Reducer tasks in our analysis. Clearly we see that from Table 1, the performance of Normal Find-Min is better than the In-Mapper Combiner version. The In-Mapper combiner strategy would work really well when we have got some idea on the topology of the graph such that neighboring clusters or components if packed together in the same Mapper file would then lead to higher merging of records. The idea behind the same is presented in the next section.

4.4 Bottleneck and further Scope for Improvement:

- When In-Mapper Combiner Strategy is used in the Mapper Stage of Find-Min, we find that number of Output records emitted reduced by only 10%. This clearly shows that there aren't enough localities in the inputs to the Mapper in a given Mapper task. Thus, when we store up all the Key-value pairs that have to be emitted, inside the Mapper, it makes the Mapper task run slower and also the number of Key-value pairs shuffled across the network doesn't reduce to such great degree.

- A solution of trying to improve on the above situation is when we analyze far-ahead i.e. what the Connected Components are going to be and packing those nodes into a given Mapper task, then the In-Mapper combiner Strategy can prove to be advantageous. But for trying out this, we need to calculate the Connected Components which itself is highly costly and also, the whole graph structure will change for the next iteration of MST. This can be a part of our future work.

Chapter 5: CONNECT-COMPONENTS STAGE

This chapter discusses the design and implementation of Connect Component stage in detail. It also discusses the performance analysis, bottlenecks and further scope of improvement.

5.1 Introduction

After we have found out Min-Edges i.e. the edge outgoing from each vertex with minimum weight and constructed a graph from these Min-Edges, we now have to find the Connected Components in this Constructed Graph. We do so as follows:

For every Vertex v_i in the newly constructed graph, we maintain a component id c^h_i which is the minimum Vertex id within h hops from v_i. Initially, c^h_i of v_i is set to its own node id: that is, $c^0_i = i$. For each iteration, each node sends its current chi to its neighbors. Then c^{h+1}_i,component id of v_i at the next step, is set to the minimum value among its current component id and the received component ids from its neighbors.

By repeating this process, component ids of nodes in a component are set to the minimum node id of the component. We iteratively do carry out this process until component ids converge. The upper bound of the number of iterations in this algorithm is iterations where d is the diameter of the graph.[8]

Initially, we have the adjacency List for the constructed graph. Also, it should be remembered that the graph is undirected, so the Adjacency Matrix is symmetric which also means that each vertex has information about what all vertices it is connected to.

We assign the initial SuperVertex for each Vertex as its NodeId itself. In each iteration, the SuperVertex value is propagated to all its neighbors. Similarly, the same neighbors will also send the Vertex their SuperVertex numbers. Then, each vertex will find the minimum of the SuperVertex numbers received including its own and assign the same to itself. In the next iteration we repeat this process. The iterations continue until there is an iteration where none of the SuperVertex numbers of the vertices change.

5.2 Our MapReduce Approach

In our optimized implementation of this stage, we need to iterate a single Map-Reduce program until convergence. The Mapper is an Identity Mapper. At the end of Find-Min stage, the adjacency List (of the Min-Edges) for each Vertex and also the SuperVertex number of that Vertex to each of the neighbors were emitted, i.e. from each Vertex with Id V we emitted two kinds of Key-Values:

Key: V Value: Adjacency List ("Self" along with SuperVertex number V)

For each Neighbor g in the List of Vertex V,

Key: g Value: SuperVertex V as "Others"

When the first Iteration of Connect-Components Reducer begins, two kinds of Key-Value pairs are obtained: the one with "self" and List of SuperVertex numbers of neighbors with type "others". The minimum of all SuperVertex numbers is found and assigned to Vertex with Id V.

Now begins the propagation phase. For each neighbor mentioned on the List, keeping that as the Key and our newly assigned SuperVertex number as Value, the same Key-Value pair is emitted, marked as "Others". Finally, the List is emitted with the Vertex V's SuperVertex number embedded into it marked as "Self".

If newly assigned SuperVertex number is compared with its previous SuperVertex, it can be known whether the Component Id has changed and thus, a global counter can be incremented intimating the same.

Also, we create a Hash-Map which stores all the Vertex-New-SuperVertex-Mappings for all the Keys (Vertices) to the Reducer Task, by initiating the Map in Configure Method of the Reducer-Task. After completion of the reducer-Task, in the Close() method, all the Hash-Maps Key-Value set

are stored to a side-effect file. Since, we will be creating these files in each iteration of Connect-Components, the files stored in the last iteration of CC will be actually used in the Merger Stage, since that is the final converged mapping. Thus, we iterate over these Reducer steps as long as the global counter across all MapReduce Tasks reaches zero.

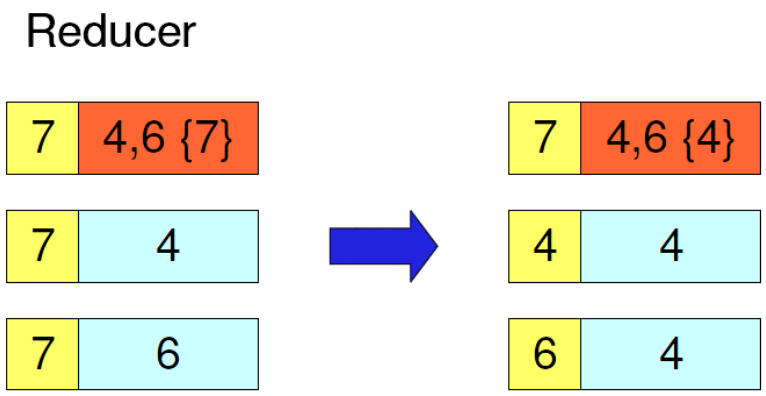

Figure 5.1: Connect-Components Stage

In the figure above, Connect-Components Stage is illustrated. The Mapper is identity. The Reducer receives for a Key, the secondary adjacency List with the Key's SuperVertex appended into it and also the neighbor's SuperVertex numbers. The Reducer obtains the minimum of all SuperVertex numbers and emits the same to those neighbors and also its own modified secondary adjacency list

Connected Components (CC) Stage using Block Multiplication Approach

Using the idea in [8], we approach the problem of Connected Components Stage in a new way. The algorithm for CC-Block is based on block multiplication. The main idea is to group elements of the input matrix into blocks or sub matrices of size b by b. Also elements of input vectors are grouped into blocks of length b. Here the grouping means all the elements in a group are put into one line of input file. In our implementation, each block contains both zero and non-zero elements of the matrix or vector. Only blocks with at least one nonzero element are saved to disk.

For performing the same, we need an additional Map-Reduce Program to convert the Adjacency List from the Find-Min Stage into Block-Format.

ConvertToBlock Stage

Let us assume that the BlockSize is b.

Mapper: The Input to the Mapper is the Adjacency List of nodes.

If the Input is of the form: *4-> 1,4,7,9,2,10,34,67*

The Key 4 will be part of the Block Node ceil(4/b). We try to assign the edges of Node 4 in partial Blocks. For example if b is 3, then

1 and 2 will map to Block 1
4 maps to 2
7, 9 maps to 3
10 maps to 4
34 maps to 12
67 maps to 23

These partial Blocks are appended with added information about which row and column of the block the edges belong to. Other Mappers will build up these partial blocks and it is the job of the Reducer to merge or combine these partial blocks to make a complete block.

Reducer: Key: BlockNode Number and Value: List of Partial Blocks

For each BlockNode, there can be a total of b blocks assigned to it. It may be possible that some of the b blocks may be completely filled with zeros. We ignore such blocks. A block contains b rows and each row is of size b. For each partial Block of size b, we assign it to some row of the block depending upon which row it must belong to.

In this manner, Block Adjacency for the BlockNode is built. Then finally, we initialize the Vector for the BlockNode as shown below.

(For example, if Block Size is 4 and Vector Node is 5, then the Vector is [17, 18, 19, 20])

Output from Reducer: For each Block in the Block-Adjacency List, using Column-No of Block as Key, Vector is emitted as the value with the BlockNode appended to it. Also the BlockNode's own Adjacency List is emitted.

Block Connect- Components(BCC)

The BCC is same as in the normal case as we need to iterate over the same Reducer methods as long as none of vectors change.

Reducer: Key: BlockNode Id and

Value: Type A-> Our own Block Adjacency List

Type B-> The Vectors of BlockNodes adjacent to it

Both the Type A and Type B Values are stored in the Reducers.

Then for each Block belonging to the BlockNode's Block-Adjacency List, the corresponding Vector is looked up that was sent to it. Block-vector multiplication is performed on the same with operation of finding the Minimum product. We store the Min-Product in a Min Vector.

The same is carried out for all the Blocks in the Block-Adjacency List and final Min-Vector across all Min-Vectors is obtained.We compare the Final Min-Vector to BlockNode's own Initial Vector. Accordingly, the global counter is incremented.

In BCC, it is also possible to decrease the number of iterations. The main idea is to multiply diagonal matrix blocks and corresponding vector blocks as much as possible in one iteration. It should be remembered that multiplying a matrix and a vector corresponds to passing node ids to one step neighbors in BCC. By multiplying diagonal blocks and vectors until the contents of the vectors do not change in one iteration, we can pass node ids to neighbors' located more than one step away. This is illustrated in Figure 5.2.

Thus, we can see in the figure that when we multiply B(i,i)with v_i several times until v_i does not change in one iteration. For example in the first iteration v_0 changed from {1, 2, 3, 4} to {1, 1, 1, 1} since it is multiplied to $B_{0,0}$ four times. This is especially useful in graphs with long chains.

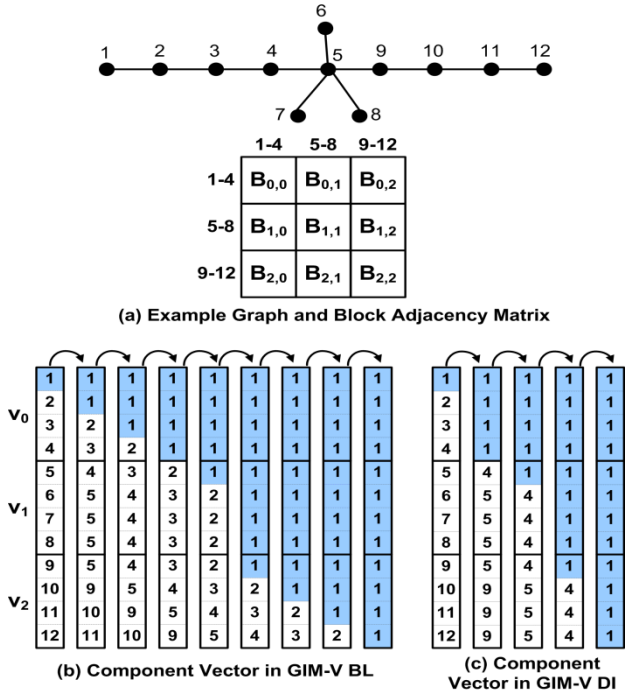

Figure 5.2: Illustration of optimization in Block Connect Component[8]

5.3 Performance Analysis

The Graph that was generated from the Find-Min stage had a total of 8816027 nodes and around 80,00,000 edges. When we ran normal Connect-Components algorithm on the same; it ran for 20 iterations before none of the SuperVertices changed their components. The running time for each iteration varied from 42 to 47 seconds.

The Connect-Components Hadoop Algorithm mentioned in [8] has 2 Map-Reduce steps since they work on directed asymmetric graphs. We implemented this version of the algorithm also. The running time was far greater than our own version's (with a single reducer) running time.

The version of Block CC which we implemented ran for the 14 iterations, an improvement over normal CC due to repeated diagonal-vector multiplications and each iteration's running time for block of size 3x3 was 1 minute 37 seconds and for block of size 5x5 was 2 mins 20 seconds.

5.4 Bottlenecks and further Scope for Improvement

The Connected Components problem has to be run on the Adjacency List that was found in Find-Min step. In each iteration of CC, the Component Ids of the node has to be sent to its neighbors and those neighbors will send the same to the node. This clearly shows how many key-value pairs has to be shuffled across the network during each iteration. For graph processing it is highly advantageous for adjacent vertices to be stored in the same block, so that any message passing between them can be processed in memory or at least without any network traffic. The general problem of partitioning a graph into multiple blocks of roughly equal size such that the intra-block links are maximized and the inter-block links are minimized is computationally quite difficult to solve and we don't pursue this option.

The Block multiplication program was run successfully by us and it gave performance similar to the original Connect-Components model. As part of future work we need to perform the following techniques before the power of this optimization can be properly harnessed.

- The Ids of the Nodes in the first iteration and the subsequent iterations are not sequential. I.e., there are many Ids missing from the List. This is because after the first iteration, the original vertices are merged into a SuperVertex and thus, we get discontinuous Ids. But for performing Block Multiplication, we require the Ids to be continuous. For converting to continuous Ids, a three step MapReduce process has to be performed before the start of each iteration of MST. Mapping from original Ids to a new IDs isn't trivial in MapReduce Model, but the performance benefits to be had from Block Multiplication compel us to perform the same as part of future work.

- For a 5x5 Block which contains 25 entries from the Adjacency List, it can be encoded in such a way that only non-zero entries are present. Since the Secondary Adjacency Matrix is very sparse, most of the blocks will have lot of zero entries and it is wasteful to send the whole blocks across the network, and also extra time in serializing and deserializing them is incurred. Although the number of Key-Value pairs exchanged are less than the original Connect-Components, but presence of so many zero-entries in the blocks implies that the amount of data transferred is much larger than the normal case.

Chapter 6: MERGER STAGE

This chapter discusses the design and implementation of Merger stage in detail. It also discusses the performance analysis, bottlenecks and further scope of improvement.

6.1 Introduction

After we have completed executing the Connect-Components Step, the SuperVertex pairing for each of the vertices is available. This is collected in a set of files in a directory. The SuperVertex for each vertex is the minimum NodeId of the Vertex to which it is a member of, i.e. if vertices 10,45,68 form a connected component, then vertices 45 and 68 will have its SuperVertex as 10.

Now, the next job is to assign the SuperVertex number to each of the vertices, reconstruct the Adjacency List in such a way that all vertices are renamed to their SuperVertex numbers and the edges are consistent. Let us take an example,

Suppose Vertex 6 is connected to 10, 17,21,89,90 with weights 4, 2,7,4,1, respectively.
i.e. *6-> {10,4},{17,2},{21,7},{89,4},{90,1}*

Also suppose that the SuperVertex pairing is as follows:
6->4
10-> 9
17->4
21->15
89->15
90->50

Then the New Adjacency List for Vertex 6 will look as
4-> {9,4},{4,2},{15,7},{15,4},{50,1}

Clearly there are edges that have the same Vertices as end-points. These have to be purged as they won't participate further in MST process. Also, it can be observed that there are multiple-proper-edges. We need to retain the one with minimum edge-weight. Thus, the final adjacency will look as: *4-> {9,4},{15,4},{50,1}*

Now, this poses many new problems. As it was seen that vertex 6 was assigned to SuperVertex 4, there may be other Vertices also who may have their SuperVertex as 4. So, each node will develop its own new partial adjacency List. These lists have to been combined in such a way that multiple edges are purged again.

Let us build up on our last example, as we saw that node 17 was also having 4 as its SuperVertex number. Then suppose Node 17 built up its Adjacency List as:
4->{20,7},{15,3},{100,45},{50,8}

And node 6 from before built up its List as:
4-> {9,4},{15,4},{50,1}

So, now we need to combine both these Lists. The new Combined List would be like:
4->{20,7},{15,3},{100,45},{50,1},{9,4}

There is also an additional problem that needs to be taken care of. Since, we are reducing each Vertex to its SuperVertex number, then, when the edges are further developed in the Spanning Tree in subsequent iterations, the end-points of the each edge may not be the original Vertices that were there in the graph initially.

So, in order to get the original edge, at the time when the new Adjacency List was created, while constructing the new edge, we store or embed the former edge into it. When we compare two same edges with different weights, the one with smaller weight is the winner and its former edge is embedded. This is the case for the first iteration of MST only. As we go on to iterations from 2nd onwards, the former end-points are also not the original edges, rather the ones embedded into it. Thus, the embedded ones are carried forward from iteration to iteration and while printing the edges of the Spanning Tree from 2nd iteration onwards, the embedded edges are printed.

6.2 Our MapReduce Approach

The Merger complications have already been mentioned. Here we discuss the implementation of the same on Hadoop. The Merger process is a 3-step MapReduce process with each MapReduce having an identity Mapper.

Merger Stage 1:

The Input Directories for this Stage are:

1. *The Original Adjacency List of the Graph for this iteration of MST(the same which was input to Find-Min Stage)*
2. *The Side-Effect files storing the Vertex-SuperVertex pairings from the last iteration of CC.*

The Mapper is identity. In the Reducer, when it receives the List for the Key/Vertex, two new Lists are created.

1. *A List X in which it has neighbors whose NodeIds are smaller than its own NodeId.*
2. *A List Y in which it has neighbors whose NodeIds are larger than its own NodeId.*

After it receives its SuperVertex number T, it appends or embeds the SuperVertex number into List Y and emit the same as

Key: Vertex/Key and Value: List X(embedded with SuperVertex number)
For each Vertex G mentioned in List X, it emits a Key-Value as:
Key: G, Value: Its Own Vertex with SuperVertex T appended.

Merger Stage 2: Identity Mapper

Reducer Process: For the Key/Vertex V, it receives two Kinds of Key-Value pairs:

1. *A List Y in which it has neighbors (let them be P) whose NodeIds are larger than its own NodeId V with its own SuperVertex number T appended onto it.*
2. *The SuperVertex number of each neighbor mentioned in P.*

When it receives Key-Values of *Type 2*, it is stored in a Hash-Map from (NodeId of Neighbor) to (SuperVertex of Neighbor). Also the List Y is stored when it is received.

After all Key-Value pairings have been received, an iteration is performed on the vertices of P.

For each Vertex W in P, lookup W in hash-Map to obtain the SuperVertex (Let it be H). If T==H ignore this edge and carry on. If T is not equal to H, create a Node N with destination Vertex H with weight equal to the weight of V-W edge and if this is Iteration 1 of MST, store the V-W Information inside this Node, else just copy the previous ancestral edge of V-W into the Node. Create a replica of Node N as M and set the destination Vertex of M as T.
Emit 2 Key-Values:

1. *Key: T, Value: N*
2. *Key: H,Value: M*

The Merger 2 process has helped in finding the new SuperVertex-SuperVertex labeling for all the edges in the original Graph for this iteration of MST. The edges where the end-point SuperVertices are same, are ignored.

Now, the duplicate edges for each SuperVertex have to be removed by checking their weights and selecting the one with minimum weight and constructing the Adjacency List for this SuperVertex. This is what we do in the Merger 3 Stage.

Merger 3 Stage:

Mapper is identity
Reducer: Key: SuperVertex V, Value: List of Nodes with weight, Destination SuperVertex and ancestral Edge mentioned.

We initialize a Hash-Map for this SuperVertex V. For each Node N, the Destination SuperVertex T is checked whether it is there in the Hash-Map. If it is not there, a new Hash Value from T-Node is created.

If it is there, stored Node Q is obtained. If weight of Node Q is > Node N, then Node Q is replaced with Node N, else proceed forward. Thus, iteration is performed on the all Nodes that are received as values. Finally, after the values are exhausted, iteration is performed on all the keys of Hash-Map, the corresponding Nodes are stored in a List L and emitted as a single Key-Value as: *Key:*

SuperVertex V, Value: List L

Thus, finally at the end of this iteration of MST, we have obtained a new symmetric Adjacency Matrix which can serve as Input to the Find-Min stage of the next iteration of MST.

In- Reducer Merge in Step 2

Suppose a series of Keys K1, K2, K3 etc is received along with their lists. For example, to a given reducer, if we get the lists (Remember, only higher node Ids will send their SuperVertex to 1)

1-> 4,6,8

4->7,9,10

10-> 12,56

and the SuperVertex numbers are:

1->1

4->1

6->6

8->6

7->1

9->6

10->10

12->10

56->23

then the new lists would look like:

1->1,6,6

1->1,6,10

10->10,23

Clearly the lists of NodeId 1 can be merged here itself instead of emitting the duplicate edges and later, taking care of them in Reducer 3 Stage of Merger. Thus, after merging the lists and also removing duplicate edges by checking weights we get,

1->6,10

10->23

and then these edges are emitted.

Thus, what we have effectively done is that the output which comes from Merger 2 is a completely merged output and now we have to perform merging globally across different reducer outputs which is performed in Reducer 3 of Merger Stage.

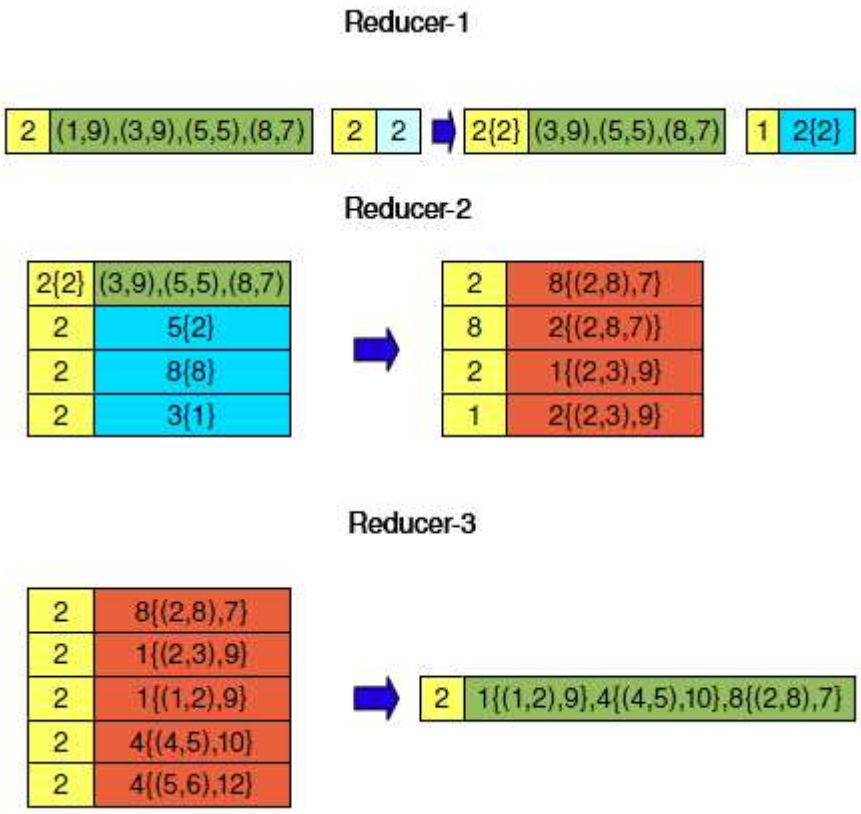

Figure 6.1: Merger Stage

In the Figure above, Merger Stage is illustrated. All the 3 Mappers are identity. Reducer 1 receives for the Key, the original List and its own final SuperVertex number. It emits two types of Lists as discussed before. Reducer 2 receives for the Key, a List and SuperVertex number of all neighbors with Ids greater than Key's ID. It emits the SuperVertex-SuperVertex pairings for all edges in List. Reducer 3 receives for a SuperVertex, all the SuperVertex pairings, it mergers all the pairings, removes all duplicate pairings based on minimum edge-weight and emits final adjacency list for next iteration of MST.

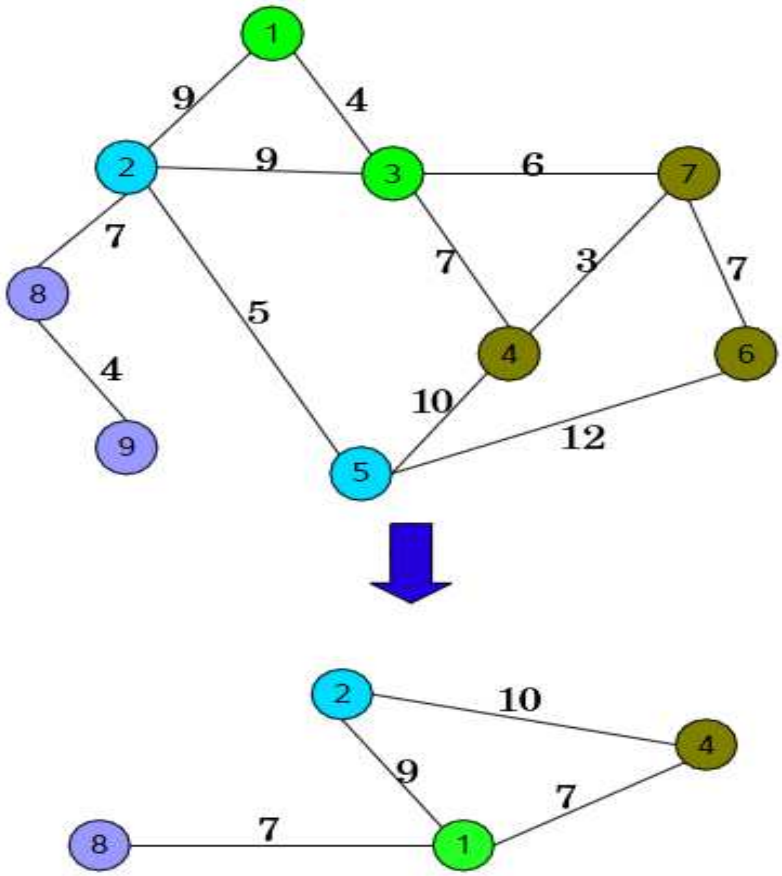

Figure 6.2: The Input Graph is transformed into a new condensed graph at the end of first iteration of our MST algorithm. The nodes with same colors are of same component and are thus subsequently merged.

6.4 Performance Analysis

It is observed from the results that the number of output records emitted when the In-Reducer Merger was used in Step 2 , **reduced by as good as 15%** to when the Non-Merging Reducer-2 was used ,though the performance was 3 times slower. This was because we needed to store the records in Memory during the whole Reducer Task which involved working on all the Keys assigned to this Reducer. Since, merging didn't happen to that large extent and lot of in-Memory storage and access was required, we went along with Reducer 2 without Merging and put the whole load of Merging on Reducer 3.

Using Normal Reducer 2 Task, all the three Reducer steps took similar amount of time. This sequence of Merger steps is a novel technique that we have implemented in this project. Changing the number of Reducer tasks doesn't seem to change the performance much, though we observed that changing **number of reducers beyond 256** increased the running time of each reducer beyond 2 minutes 20 seconds.

Below are the **statistics** for all the reducers in Merger stage of the first iteration of our MST implementation.

Reducer 1:

Launched reduce tasks=225;Reduce input groups=8816027
Map input records=17632054
Reduce output records=95498333
Map output records=17632054
Reduce input records=17632054
MR Job Total Elapsed Time: 1 min, 50 sec

Reducer 2:
Launched reduce tasks=231;Reduce input groups=4568602
Map input records=95498333
Reduce output records=164926062
Map output records=95498333
Reduce input records=95498333
MR Job Total Elapsed Time: 1 min, 42 sec

Reducer 3:

Launched reduce tasks=237
Reduce input groups=655301
Map input records=164926062
Reduce output records=655301
Map output records=164926062
Reduce input records=164926062
MR Job Total Elapsed Time: 1 min, 47 sec

6.4 Bottlenecks and further Scope for Improvement

The Merging of Connected Components into a single new SuperVertex and relabeling all the edges and removing self-loops and duplicate edges seem to be the toughest and most compute- intensive step in MST. Because of this, three Reducer steps have to be run to perform the same. During the course of our project we tried to run different versions of the Merger step, the final version that we proposed gave the best performance. Various difficulties on doing the same on Hadoop are:

- The input to the Reducer 1 is both the final Connect-Components SuperVertex labelling and the input graph to that iteration of MST. There must be a simple method to collect both the information together. This clearly **shuffles 2*N number of Key-value pairs**.

- The SuperVertex ID has to be propagated to all its neighbors. One good optimization that we have implemented is by sending it to neighbors who all are lesser in NodeIDs. In spite of this, we still need to **shuffle V + E number of edges**. There is no chance of obtaining any locality here.

- We only built up a partial adjacency List in the Reducer 2 and thus, we need to finally merge the lists into a complete one in Reducer 3. Here also, instead of emitting partial Adjacency Lists, all edges in the List have to be emitted twice so that a symmetric adjacency list can be built in reducer 3. This clearly shuffles lot of key-value pairs across the network.

Chapter 7: CONCLUSION AND FUTURE WORK

We have decomposed many useful graph operations in Boruvka's MST algorithm into MapReduce steps and implemented the same on a Hadoop cluster. Many customizable features of Hadoop like Mapper, Reducer, Combiner, Partitioner, Input-Output formats, Sorting and Network shuffling have significantly helped in distributed processing of large amount of data. We have observed that MapReduce can be seen as an enabling technology for analyzing large datasets, and can thus be efficiently exploited by developers in developing their own high performance large-scale distributed applications. However, we came across many performance bottlenecks when analyzing large graphs, because standard best practices in MapReduce do not sufficiently address serializing, partitioning, and distributing the graph across a large cluster.

In our work, Minimum Spanning Tree (MST) was taken as a basis for analyzing the efficiency and programmability of large scale graph processing because it is one of the most studied combinatorial problems and consists of different types of intricacies at each step. We have thoroughly explained and reasoned the MapReduce approaches in our implementation and have used various techniques like In-Mapper Combining, In-Reducer Combiner, and Block Partitioning to optimize the performance and also have discussed their need in our problem context in detail with their pros-cons.

Another issue regarding the practicality of cloud-based MapReduce computations on graphs that we feel is relevant is inter-processor bandwidth. Each MapReduce job has the potential to move every graph record from one processor and its disk to another. This involves serializing, deserializing, sorting and network shuffling which becomes a bottleneck in large scale graph processing.The prospect of the entire graph traversing the cloud fabric for each MapReduce job is disturbing.

Future work as discussed in earlier sections remains to improve our optimized design patterns even further. For example, Partitioning could be done to cluster based on actual graph topology. Block multiplication approach to Connect-Components can be further optimized by using techniques like block encoding and efficient re-mapping of the input nodeIds.

BIBLIOGRAPHY

1. Jonathan Cohen, “Graph Twiddling in a MapReduce World”, Volume 11, Issue 4, pp 29–41, IEEE Computing in Science & Engineering, July-Aug, 2009.
2. Jimmy Lin and Michael Schatz. Design Patterns for Efficient Graph Algorithms in MapReduce.*Proceedings of the 2010 Workshop on Mining and Learning with Graphs Workshop (MLG-2010)*, July 2010, Washington, D.C.
3. “Hadoop Tutorial-YDN-Yahoo!DeveloperNetwork” *http://developer.yahoo.com/hadoop/tutorial/*
4. D.A. Bader and G. Cong, ``A Fast, Parallel Spanning Tree Algorithm for Symmetric Multiprocessors (SMPs)," Journal of Parallel and Distributed Computing, 65(9):994-1006, 2005.
5. “Illinois Cloud Computing Testbed”, *http://cloud.cs.illinois.edu/*
6. *“Cloud9-A MapReduce Library” http://www.umiacs.umd.edu/~jimmylin/cloud9/docs/content/pagerank.html*
7. J.D. Cohen, “Trusses: Cohesive Subgraphs for Social Network Analysis,” 2008; http://www2.computer.org/cms/Computer. org/dl/mags/cs/2009/04/extras/msp2009040029s1.pdf.
8. Kang, U, Tsourakakis, C.E. and Faloutsos, C. “PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations”. *Proceedings of the* ICDM '09. Ninth IEEE International Conference on *Data Mining, 2009.*
9. Kang, U, Tsourakakis, C.E. and Faloutsos ,”HADI: Fast Diameter Estimation and Mining in Massive Graphs with Hadoop” , *Journal ACM Transactions on Knowledge Discovery from Data (TKDD*) *Volume 5 Issue 2, February 2011*
10. Abhishek Verma, Xavier Llora, David E. Goldberg, Roy H. Campbell, “Scaling Simple and Compact Genetic Algorithms using MapReduce”, *IlliGAL Report No. 2009001 October, 2009*
11. Shimin Chen, Steven W. Schlosser. "Map-Reduce Meets Wider Varieties of Applications." *Intel Labs Pittsburgh Tech Report, IRP-TR-08-05, May, 2008.*
12. Ekaterina Gonina and Laxmikant V. Kale,” Parallel Prim’s algorithm on dense graphs with a novel extension”, *2007.*