2008

Visualization Web Portal on the Grid

Sandhya Turaga
San Jose State University

VISUALIZATION WEB-PORTAL ON THE GRID

A Project Report

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

Of the requirements for the Degree

Master of Computer Science

By

Sandhya C Turaga

May 2008

Approved by: Department of Computer Science
College of Science
San Jose State University
San Jose, CA

Dr. SoonTee Teoh

Dr. Robert Chun

Dr. Agustin Araya

ACKNOWLEGEMENTS

I would like to thank Dr. SoonTee Teoh, for his guidance and support throughout this project. I thank Dr. Robert Chun and Dr. Augstin Araya for their valuable suggestions and feedback on my work.

I convey my sincere gratitude to my beloved husband, Kishore Jonnalagedda, for his support and encouragement throughout the project.

ABSTRACT

Visualization Web Portal on the Grid

By Sandhya C Turaga

Grid-based visualization portals help scientists explore data that is distributed across the globe and to visualize the data. Visualization allows scientists to explore data effectively and helps them to obtain further insights into the data. We developed a visualization grid portal whose main aim is to be able to store large data sets across machines in a cluster in a distributed fashion, and to allow users of the Grid Portal to visualize the data set effectively. This Grid portal uses HADOOP, a grid platform that facilitates flexible data storage in a distributed fashion, and supports distributed computation as well.

The main goal of the Grid portal is to support positional datasets from the user, process them on the grid efficiently, and produce the visualization. The input on the grid is partitioned into multiple pieces and each partition is executed concurrently. Current implementation of HADOOP does not consider any boundaries when it partitions the input, which limits the kind of applications that can run on the Grid. Our aim is to implement boundary-based input partition and to enable online job submission to the grid. We discuss the advantages and disadvantages of the boundary-based input split. Finally, we compare the performance of grid processing with standalone machine processing of the same dataset and determine which approach is more efficient and faster.

Table of Contents

List of Figures

List of Tables

1. INTRODUCTION

The need for high-performance computation in scientific communities is increasing, and it would be beneficial to provide high performance without having users install additional software on their machines. Grid portals are very useful for processing large amounts of data in a distributed manner to meet demanding computational needs. For large-scale applications, storing the data and processing it on a single machine is a difficult task.

Grid software helps to store large databases on multiple machines that are on the grid and to share computation on multiple machines to speed up computation when the data is large. Machines on the grid may be located across the world. Various tools for facilitating grid-enabled computations are available on the market.

Computational Grid and Data Grid are the two different types of grids available. The purpose of the Data Grid is to facilitate large-scale data storage and access among multiple machines; the purpose of the Computational Grid is to provide large-scale data computation on multiple machines to speed up computation. In addition, some grid tools provide both large-scale data storage and computation. The advantage of using grid tools is that they hide the complexities involved in distributing data and computations among machines as well as the complexities involved in connecting to multiple remote servers, collecting the data, and producing the final output.

Grid-enabled portals allow users to access and visualize the data regardless of the geographic location of the user and without having to install specialized software. In addition, once a user is logged on to one machine from a browser on the grid, the user has full access to the resources available on the grid. The grid portal that we discuss here supports any dataset in which values can be accessed through column numbers and processes the dataset on the grid, and produces the visualization of the dataset. The main goals of the grid portal are to be able to submit jobs to the grid efficiently, to enable the grid to process sequential files that have boundary conditions, and to improve the performance of the grid by distributing the tasks in an efficient way.

This report is organized as follows:

- Section 2 describes grids and discusses their characteristics and applications suitable for execution on the grid. We also describe the processing of a sample dataset that we have chosen for visualization, and results of our research into grid software available on the market.
- Section 3 describes the design and implementation of a grid portal and the challenges involved in producing visualization on the grid.
- Section 4 describes the performance results of the grid and the performance improvements that we have implemented.
- Section 5 describes various visualizations that we have produced on the grid.

- Section 6 presents our conclusions and discusses future work.

2. GRID COMPUTING

2.1 What is a Grid?

As **Andrew Grimshaw, CTO and Founder Avaki Corporation** indicates, a grid is a collection of distributed resources connected by a network at different geographical locations [7]. These computing resources share some common characteristics; for instance, they can be heterogeneous (each CPU's architecture can be different); they can belong to different organizations; and they can have different security and management policies.

"A grid gathers all the resources (e.g., CPU, data, and applications) and makes them accessible in secure manner to users and applications." [7]

Grid middleware has become popular because it manages the underlying hardware complexity of sharing resources, and providing access to resources in a secure manner. The primary aim of grid middleware is to provide transparent access to grid resources. Application developers can take the advantage of Grid middleware to produce robust grid applications since grid middleware takes care of the resource access, fault tolerance, and fault recovery of machines on the grid.

Because demand for high performance computing is constantly increasing, instead of producing new machines with higher performance, using idle CPUs on the network to meet computation requirements is increasingly cost effective for organizations. In addition, a grid helps in performing more jobs or more work in a given day, which can increase an organization's output.

Different types of grids are available on the market such as Data Grids, Computational Grids, and Science Grids. Data Grids manage the complexity of storing data across machines on a grid and provide data consistency. Data Grids also reduce the time to copy the data manually across machines. Computational Grids share computation across machines to perform tasks in a reasonable time. Computers are used extensively in weather forecasting, science, and bio-medical research to perform complex analysis of data and to perform complex calculations. Computational Grids assist users in those areas perform these complex tasks in short periods of time.

"A computational grid is hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities."[5]

2.2 Grid Characteristics

There are several definitions of grids and what they are supposed to do. The following are the characteristics of a grid taken from [2]. A grid must be able to handle a number of resources ranging from a few to millions without performance degradation. Geographically distributed resources can be part of a grid. A grid should be able to support heterogeneity; resources should be sharable across organizations and should be able to conform to the security policies defined in various organizations participating on the grid. A grid can be seen as a single virtual computer with the following characteristics:

- A grid must assure quality of service.
- Resources must be coordinated to produce aggregate computing capabilities.
- Computing resources should be transparent to the users.
- A grid should be able to adapt to dynamic configurations using automatic failure mechanisms and should be able to maximize performance from resources on the grid [2].
- A grid should be able to process large datasets that need intense computational power and intense data management [2].

2.3 Distributed application characteristics

It is important to determine what kind of applications can be used on a grid. For obvious reasons such as the time it takes to distribute data and tasks across machines on the network, the time it takes to coordinate the resources on the grid, and the overhead involved in initiating the actual computation, not all applications are suitable for grid usage. A grid is usually used in situations in which the resources of a single machine are not sufficient to perform the required tasks, or the time it takes to perform the computation on one machine is very high. Therefore, if an application is expecting results in real time, it is not an appropriate candidate for the grid. An application with tasks that need access to gigabytes of data to perform computations and an application that can be partitioned into multiple individual tasks that can run concurrently are suitable candidates for the grid [6]. If an application cannot be partitioned into multiple tasks, it defeats the purpose of distributed computing. In addition, when a task is broken down into small sub-tasks, each sub-task should be big enough to compensate for the overhead of initiating the task and should be small enough to produce results in a few hours or days. Therefore, it is necessary to assess what kind of applications and computations can be deployed on a grid.

2.4 Sample Dataset to be visualized

The Micro data sample from the 2000 U.S. census data is particularly suitable for visualization and data analysis on a grid.

2.4.1 Micro data Sample from the 2000 U.S. census

http://www.cs.umd.edu/hcil/InfovisRepository/contest-2006/info.htm

The micro sample dataset contains 5% of the results of the US census 2000. Housing unit record and number of persons per housing unit are recorded in the dataset. The large dataset contains information at the state level. The housing unit record starts with the character "H" and person records start with a character "P" and follow "H" records; every "P" record has a parent "H" record. To process the dataset, we need to process the "H" records and their dependant "P" records together. The purpose of choosing this dataset is to create an effective visualization on a geographic unit that has different information such as professions, income, expenditure, and cultural background. The size of the data is 5 GB.

2.4.2 Processing of Micro data Sample

Since this project aims to produce visualization, we have selected the Micro data Sample from the 2002 census dataset to produce visualization. This dataset provides housing unit records and person records. Housing unit records contain household languages, person's citizenship status, race, profession, industry, and cultural background. All this data is in ASCII format. Each single ASCII file has all the housing records information belonging to a single state in the U.S. Therefore, the dataset has 50 ASCII files, each with information for the people living in the corresponding state. Appendix A-1 shows a sample ASCII file from the dataset.

In each ASCII file, each Housing record starts with a letter H and each Person's record starts with a letter P. Each person record contains 314 characters. Each character or set of characters in each record specifies some information about the corresponding housing unit such as race, status, profession, state. Each housing record contains 266 characters. Each character or set of characters specifies information such as annual income for the household, social security income, number of cars owned by the household, amount of money each household spends on gas, and rent. The details of what the each column means are documented.

Generally, a grid is used to compute very intense calculations on large datasets for which it takes long time to process the data and computations on one machine. Since census information contains lot of in-depth information about a house and the persons living in it, and since the number of people living in a state is large, processing information on the census dataset is time consuming. The overall size of the dataset is 5GB, which is 5% of the US census data. Therefore, processing the data and performing various calculations on the census dataset suits a grid's prerequisites.

2.4.3 Visualization on the Grid

The total size of the dataset is 5 GB. A single machine takes considerable time to perform complex calculations on this 5 GB of data and to produce the visualization. This dataset has 50 files; processing all these files to produce a simple visualization on a single machine takes approximately 10 minutes. Since we are trying to develop a visualization web portal, there is a high probability that multiple requests to perform different calculations on this dataset can be received by the server in the same period. If one machine approximately takes 10 minutes to produce the results for one complex calculation, and if we assume that hundreds of machines are serving the clients, even then every user has to wait for 10 minutes to get the result once the job is submitted to the web server. Thus if only a limited number of machines exist, user wait time might increase. If a single user submits multiple requests to produce animated visualization effects, then the response time would increase even more. If the dataset is bigger than 5 GB, the response time of a single machine would be prohibitively high unless we use a supercomputer.

Therefore, to support above-specified situations, either finding a faster machine with huge amounts of memory or combining all machines available and utilizing those resources in their fullest are the solutions. The option of getting a faster machine is not affordable. Therefore, utilizing all available machines and their resources and performing distributed computing might be viable solution. As explained in an earlier section, a grid is suitable to process this kind of complex application and to produce results fast without having to spending more money.

2.5 Evaluation of Grid software

Grid computing software is used to create the grid environment and to deploy applications on the grid. The following are the various grid software packages that we have evaluated.

2.5.1 GRIDGAIN

www.**gridgain**.com/

This computational grid provides distributed computation. GRIDGAIN is written purely in the JAVA programming language. This software helps in creating a computational grid, deploying nodes onto the grid, creating grid tasks and grid jobs, distributing tasks among nodes, and executing the tasks. GRIDGAIN software allows splitting a task into sub-tasks; it executes each task in parallel, combines the results of each sub-task, and produces the result quickly. However, GRIDGAIN does not support data distribution. Data must be replicated across all the machines manually.

GridGain runs on both Linux and Windows operating systems. Computational grids help in situations in which it takes long time to perform computations, such as complex build processes and large-file processing.

2.5.2 HiveX

http://sourceforge.net/projects/hivex/

HiveX is a computational grid that distributes tasks among nodes (other computers) on the grid. This grid software is written in C/C++. HiveX can run on LINUX, free BSD, and UNIX operating systems.

2.5.3 Avaki

http://www.sybase.com/products/allproductsa-z/avakieii/sybaseavakifordatagrid

Sybase Avaki is data grid that provides the transparent unification of data distributed across multiple machines in a cluster. Avaki can be integrated with a computational grid without any changes to existing grid environments. Avaki is developed in JAVA. Avaki Data Grid also provides secure data sharing across organizations.

2.5.4 Globus Toolkit

http://www.globus.org/toolkit/

Globus is an open source toolkit for building computational grids. It provides data access and storage across firewalls, and it is very useful for setting up large enterprise level computational grids.

2.5.5 HADOOP

http://lucene.apache.org/hadoop/

HADOOP is grid software for running applications in a cluster of machines, providing both a data and a computational grid. In other words, HADOOP provides a framework to store data across machines in a cluster and to distribute computational work across machines. HADOOP transparently provides applications for both reliability and data movement. HADOOP implements the map/reduce paradigm to divide tasks into small fragments of work; each fragment can be executed on any node in the cluster. HADOOP's Map/Reduce is inspired by Google's Map/Reduce paradigm. HADOOP is developed in JAVA.

HADOOP implements Master/Slave architecture to distribute data and computation across machines.

3. GRID-BASED PORTAL FOR VISUALIZATION

3.1 Design overview

The main purpose of the web portal presented here is to enable users to access grid resources to perform their data and time-intensive computations in a distributed manner without having to install any special software on their machines. The project follows a client-server architecture in which the client is typically a browser, and the server is an HTTP server supported through Tomcat. We used SERVLETS and JSP pages to implement the server layer. The server interacts with the grid to submit jobs and returns results to users. Once a user registers with the grid, the user can upload his/her dataset to the grid, and submit various requests to the grid. We chose position data sets as input to computations on the grid. In a position data set, each line of the input file can be recognized by a sequence number. For instance if each line in an input file is like "H000011113334450111000000" then "H" is in position one. Therefore, if the user specifies to access a character at position two we would take "0" since the value in the second position in the line is "0." In the metadata file the user can specify, for example, to select values from one to five and perform specified computation on the values. Every user is allotted a home directory on the grid at the time of registration. To perform computations on the uploaded datasets, the user specifies metadata through an HTML form, which is then stored in an xml format on the grid in a special directory under the user's home. Metadata consists of details about a dataset, such as the definition for the contents of a dataset and the computation algorithm to be used during job processing. We chose XML and a supporting XSD for that XML structure due to the ease in defining and expanding well-defined structures. During the definition of the metadata by the user, he/she can specify whether there are any conditions that the computation must follow, such as read a line if that line contains some character or read consequent lines only if a condition on the previous line is satisfied. The server transforms those specifications into regular expressions and stores them in the generated XML. In addition, the user has to specify which operation (sum, average, count, and/or median) should be performed on the given data, which column should be read for that operation, and whether that column should be given a label. Users can create as many metadata files as are required by the dataset.

Once the metadata is ready, the user can select one or more data files as input for the job and their corresponding metadata file to perform the computation on the grid. This information is then submitted to the web server. The Map/Reduce program on the Tomcat server specifies HADOOP, which datasets HADOOP has to read to perform the computation, the logic of the computation, the set of input directories for the computation, the set of output directories where HADOOP should store the final results, and the dependencies that exist in a file, if any, to perform the computation. In addition, the Map/Reduce program invokes HADOOP's methods to distribute the specified computation and data across machines. It is HADOOP's responsibility to take the user-defined Map/Reduce program and distribute the program to multiple machines, splitting the input into multiple chunks to perform concurrent execution. Although splitting the input into multiple chunks is handled by HADOOP, due to its limited splitting functionality, we have overridden some of its functionality to extend its split function, as explained in detail in later sections.

Our project is designed such that data-intensive calculations are performed on the grid in the backend and visualization is performed on the front end in an applet to separate the presentation logic from application logic. Figure 1 illustrates the overall design of the project. As shown, there are two types of job submission—synchronous and asynchronous; the one that is used is determined according the size of the dataset by the server layer. The user is agnostic of this behavior.

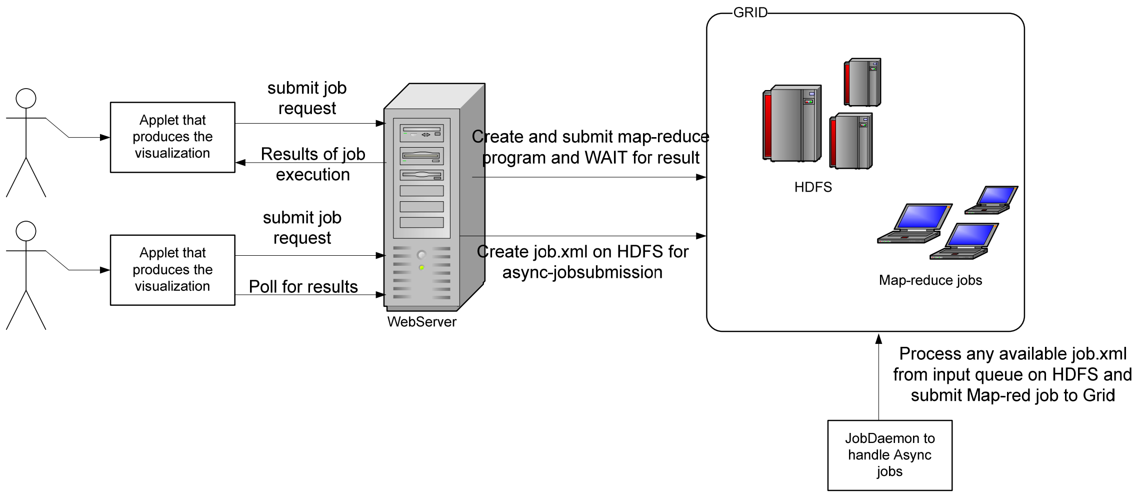

GRID
Applet that produces the visualization
submit job request
Results of job execution
Create and submit map-reduce program and WAIT for result
HDFS
Applet that produces the visualization
submit job request
Poll for results
Create job.xml on HDFS for async-jobsubmission
Map-reduce jobs
WebServer
Process any available job.xml from input queue on HDFS and submit Map-red job to Grid
JobDaemon to handle Async jobs

Figure 1 Design Overview of the project

3.2 Map/Reduce Programming Model

HADOOP uses the Map/Reduce programming model to execute a given task concurrently. In this model, a computation is expressed as a set of Map and Reduce tasks. A map task takes as input key/value pairs and produces a set of intermediate key/value pairs. All intermediate values with the same key are combined and passed to a reduce task. A reduce task takes the intermediate key and a set of values of that key, combines the values together, and produces one output value.

In a map task, each input is split into M partitions. Usually the number of partitions is configurable by the user and the input is partitioned using a split function on the intermediate key. The default split function hashes its input key. Each partitioned piece can be executed in parallel across the machines to speed up the computation.

When the user calls a Map/Reduce program, the following sequence takes place.

1. The input file or files are split into M pieces with a specific block size, and the pieces are distributed across the machines for processing. The default implementation calculates the optimal split size as:

   max (min (block_size, data/#maps), min_split_size)

   This formulation may not be workable for all datasets as explained below in our modification of the split calculation.
2. The machine processing the map task produces key/value pairs from the input split. For instance, if the input to the map task is a line-based file, then the map interface produces a sequential key per line, and the position of the starting byte in the line with respect to the start of the file is the value for that key. The framework produces the keys to guarantee the ordering of the input split. The ordering is important for producing sorted output per input split. These key/value pairs are passed to the user-defined map function. User defined Map function reads each key/value pair and produces an intermediate key/value pair. For instance, if the user-defined map function specifies to read the income of a person from columns 32-40 in a given value parameter, and to combine that with a key "AA", then AA/income are produced as intermediate key/value pairs. In generating intermediate key/value pairs, the user-defined map function defines both the key and the value.
3. A partition function partitions the intermediate key/value pairs into R regions. R is the number of reduce tasks per map task that can be controlled by the user. Data is partitioned into R regions with the partition function. The default partition function hashes the intermediate key. The location of the partitioned data is given to the reducer, which eventually generates an output file for each reduce task.
4. Before reducing the key/value pairs, the Reduce function sorts the output with the intermediate key to group the values of the same key. Essentially, a set of values for an intermediate key are generated and passed to the user-defined Reduce function. The user-defined Reduce function performs its operations on key/value pairs, and the output is appended to the output file of that reduction task.

   Figure2 shows that the input data is split into multiple blocks, and each block is distributed to a map task. Once the operations that are specified in the map class are executed on the given block of data, the intermediate output from Map is given to Reduce. The Reduce task performs operations on the given intermediate key/value pairs and produces one output file per Reduce task. Since there is only one Reduce task, only one output file is created in the figure.
5. When all map tasks are completed, one output file for each reduction is created in the final output.

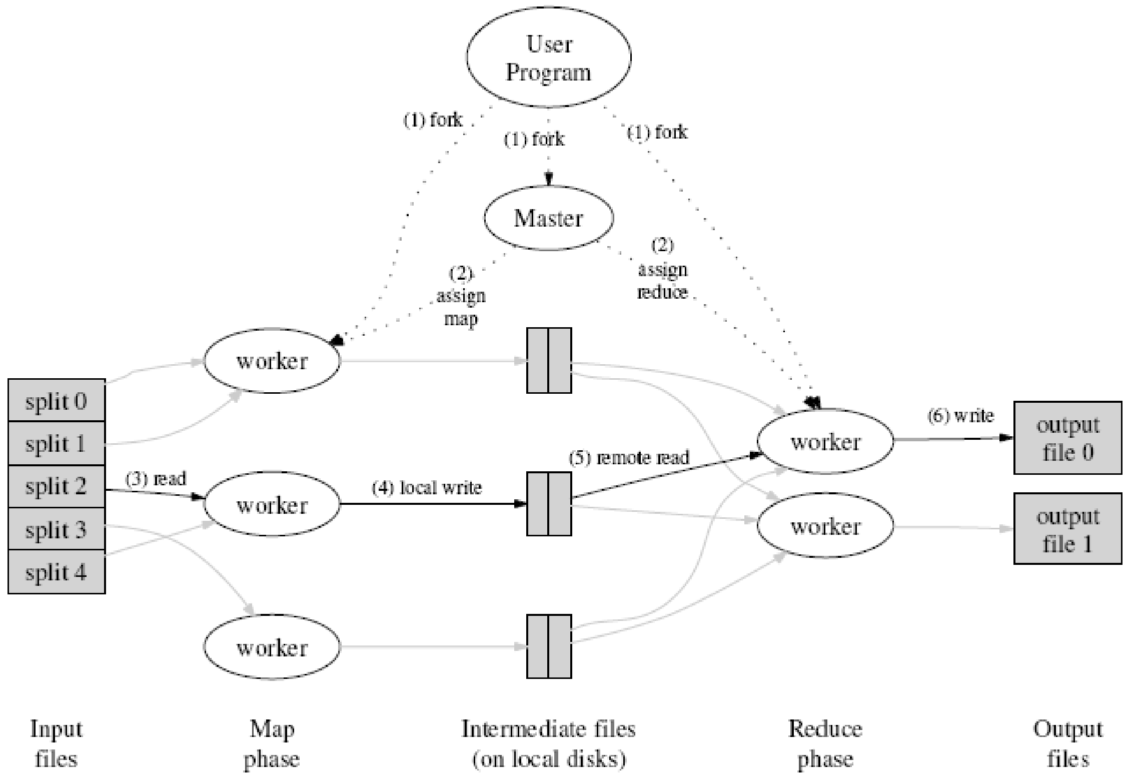


Figure 2 Map/Reduce programming paradigm [1]

3.3 HADOOP's Map/Reduce architecture

In HADOOP, before the map operation takes place, the input is logically split into M pieces by an operation called FileSplit, and each split is distributed to multiple machines. The default InputFormat class, responsible for creating key/value pairs (K1, V1), is passed to the mapping function, and creates a set of key/value pairs with each key as a sequentially generated long number and each value as one line in the input split. The user specifies the Mapper class, which generates intermediate key/value pairs (K2, V2) using the input key/value pairs (K1, V1) generated by the InputFormat. After creating the intermediate key/value pairs, the Mapper class calls OutputCollector.collect and passes the intermediate key/value pairs (K2, V2). The collected output is partitioned using an hash function on the key class. The number of partitions is configurable by the user. Each output file will be routed to a reduce task with a key and a set of values.

Before the reduce task starts, if the output files from the map tasks are distributed across machines, then these files are first copied to the local file system. Once the data is available locally, all the data is appended to a single file to produce a single input file for the reduce task. The final merged file is unsorted. To ensure a sorted order on the keys, HADOOP uses Merge-sort to generate the final input file for reduction. The following figure explains the process from mapping to reduction. Note that the combine step is optional. The final key/values (K3, V3 and K4, V4) are generated by the reduce function in each of the reduction tasks. The output is stored as one file per reduce task.

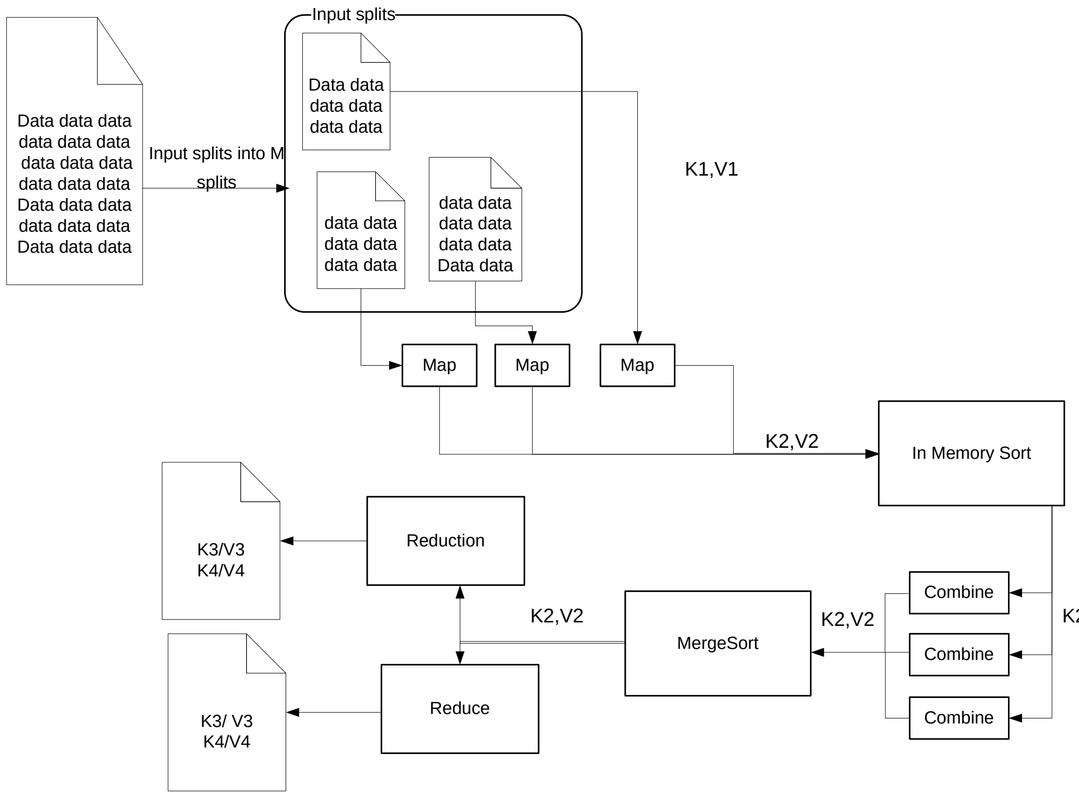

Data data data data data data data data data data data data Data data data data data data Data data data
Input splits into M splits
Input splits
Data data data data data data
data data data data data data
data data data data data data Data data
K1,V1
Map
Map
Map
K2,V2
In Memory Sort
K3/V3 K4/V4
Reduction
K2,V2
MergeSort
K2,V2
K2,V2
Combine
Combine
Combine
K3/ V3 K4/V4
Reduce

Figure 3 Map/Reduce implementation in the HADOOP framework

Figure 3 explains the sequence in which HADOOP executes a given task. Input data is split into M number of splits, and each split is distributed to a user defined map task. The map task generates intermediate key/value pairs K2, V2, shuffles them, and passes the result to the combiner class. The user-defined combiner combines all the values of the same key, performs the operations specified in combine function, and writes them to the distributed file system. The keys are passed to the reduce phase after all the map and combine tasks are finished. The final output is written to a file after the operations specified in the reduce function are executed on the input to the reduce function. One output file is created per reduce phase. If a user specifies four reductions, four output files are generated and the user has to read the four files to get the complete final output.

3.4 Steps involved In Producing Visualizations on the Dataset

1. Specify configurations to be used by HADOOP in running the submitted job. These configurations are described in detail in section 3.4.1; the xml file that has all the configuration values that we set for the project is described in Appendix B.
2. A user uploads a dataset onto the grid and specifies a split condition if any. For this dataset the split condition is “^H\\n”. The dataset is described in section 2.4.

Section 3.4.9 describes various options that we considered for performing the split operation with a boundary condition, and section 3.4.10 describes in detail the implementation of that split calculation.

3. We store the dataset on the grid and calculate the split metadata on the grid. That is, we submit a job to HADOOP to calculate the start and end position of each input split and store the final output on the grid.
4. Take the input information from the user such as input dataset location, metadata of the input dataset. In the metadata the user specifies the columns of the input data to be used in the computations and the needed computations, such as SUM, AVERAGE, COUNT, MEDIAN, CORRELATION, MULTIPLICAITON, STANDARD DEVIATION, MAX, and MIN.
5. Prepare a metadata.xml file, job.xml, and store it on the grid. Creating the metadata.xml file is described in detail in section 3.4.2, and the sample metadata.xml file is described in Table 1. The creation of job.xml is described in detail in section 3.4.3, and a sample job.xml is shown in Table 2. The creation of metadata.xml and job.xml takes place on the web server.
6. Send a request to the Map/Reduce program we have written, with metadata.xml and job.xml as arguments. The web server sends a request to the Map/Reduce program that we have written. In our project, we used TOMCAT as the web server.
7. The Map/Reduce program parses the xml files, extracts the required information, and submits a request to HADOOP in which it specifies which Map class and Reduce class HADOOP should use. We wrote these Map classes and Reduce classes to include the actual logic needed to perform any computations that the user has requested. We have described how the web server can submit a request to HADOOP in section 3.4.4 and described the Map/Reduce program in detail in section 3.4.5.
8. Along with the above information, we specify the location of the split information and the split condition that we have used in HADOOP's configuration object. In addition, we implemented the configuration object to use the "LogicalInputFormat" that we had written instead of HADOOP's "InputFormat." HADOOP's "InputFormat" is the class that actually calculates the split information, including the start index, the length, and the size of the split. We use "LogicalInputFormat" to look for the split metadata files specified in the configuration object, and we pass this split information to HADOOP. We use "LogicalInputFormat," only if the user specifies a split condition, otherwise we use HADOOP's default "InputFormat" class to calculate the split information.
9. As soon as HADOOP gets the split information from either "LogicalInputFormat" or "InputFormat," it sorts the splits by size and distributes them to machines on the grid. Each machine on the grid calls our Map class to perform operations on the split it has received. Each machine gives one line at a time to the user-defined Map class. This line from the input chunk would be written by the "LineRecordReader" class if we were to use HADOOP's default inputformat. However, if we use "LogicalInputFormat," "LogicalRecordReader" reads each

line and calls the Map class. Section 3.4.9 describes in detail how we implemented the “LogicalInputFormat” and “LogicalRecordReader.” Once all the input chunk’s processing is finished, the output from the map tasks is shuffled so that all the outputs for one key go to the same machine that runs the Reduction. Here again the Reduce class that we have written is used by HADOOP and the final output is written to a sequence file on the grid.

10. Our browser polls the output folder with the unique jobid that it has received from the web server. As soon as the output file is created on the grid, the browser reads the information from that sequence file and submits the data to the JAVA applet. The applet performs the visualization on the received data. How we performed various visualizations is described in section 4.

3.4.1 HADOOP configuration

To initialize the grid, a number of parameters must be carefully configured in HADOOP’s configuration files. Some of the important configurations are listed below.

- The location of the log directory for each node on the grid.
- The data directory location for intermediate data for each node on the grid.
- The machine to serve as the NAMENODE (Master), which coordinates the rest of the DATANODES, on the grid. NAMENODE is the one that handles the distribution of data among the other resources on the grid. Note that NAMENODE does not receive any user-data; it only acts as a broker between data nodes.
- The machine to serve as the JOBTRACKER, which distributes jobs to TASKTRACKERS and keeps track of the jobs. JOBTRACKER is the machine that distributes and coordinates jobs among worker machines and worker machines are the TASKTRACKERS .
- The number of maps to be performed on a given job. Selecting the number of maps and reduces is a tricky task since it directly affects the overall job execution time.
- The number of reductions to be performed on a given job.
- The number of tasks that can run concurrently on a machine.
- The maximum size of a chunk, to be used as an upper bound when splitting the input in computations. The default value of chunk size is 64MB.
- Ports on which the TASKTRACKER, JOBTRACKER, NAMENODE, and DATANODE processes should run.

Appendix B shows the configuration file that we used to specify the above configurations to produce visualizations for the census dataset.

3.4.2 Metadata creation

Once the user uploads a dataset, understanding the dataset is a major task to perform any computations on the data. The first step is to collect information from the user including which line and column to read for performing the computation and the values need to be matched; this interaction is done through the front-end HTML form. The information submitted by the user is translated into xml and stored in a metadata directory on the grid. The schema is defined such that the SUM, AVERAGE, COUNT, and MEDIAN functions can be called recursively on a variety of data. The schema can support mutually exclusive operations and conditionally dependant operations. Mutually exclusive operations are individual line-level operations that are dependent on some data in that line. Conditionally dependant operations are those that support relative dependency between lines when performing calculations.

Table 1 shows a sample metadata xml file. Each metadata xml file can have any number of "dataprocess" elements. Each "dataprocess" element contains selection criteria for a calculation; any number of operations with key/value items can be specified. According to the selection criteria, the user can specify an expression such as process the following lines only if the current line starts with "H"–a conditionally dependent operation. If the user does not specify the selection criteria but specifies only operations, then it is an exclusive operation. Table 1 specifies sample metadata that specifies the grid to calculate the percentage of Asians, Hispanics, and Americans living in each state. "<tns:condition>1-6=key[1]</tns:condition>" specifies that the user has to extract the value from 1-6 characters in the given input line, and has to compare it with the key 1. In that way, the user can provide an expression such as "<tns:expression>^H</tns:expression>," which specifies that the user has to consider lines that start with an H. Likewise, the user can also specify the function to be performed on the extracted data. For example the function tag in the table "<tns:function></tns:count></tns:function>" specifies to perform a count operation on the given input.

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:dataprocess>
        <tns:id>1</tns:id>
        <tns:selectioncriteria>
                <tns:expression>^H</tns:expression>
        </tns:selectioncriteria>
        <tns:storage>
                <tns:keys>
                        <tns:key>1-6</tns:key>
                        <tns:key>30-39</tns:key>
                </tns:keys>
        </tns:storage>
 </tns:dataprocess>
 <tns:dataprocess>
  <tns:id>2</tns:id>
  <tns:selectioncriteria>
   <tns:expression>^P</tns:expression>
   <tns:condition>1-6=key[1]</tns:condition>
  </tns:selectioncriteria>
  <tns:operation>
        <tns:columndetail>37</tns:columndetail>
        <tns:columnvalues>
                <tns:key>1</tns:key>
                <tns:value>Black</tns:value>
        </tns:columnvalues>
        <tns:columnvalues>
                <tns:key>2</tns:key>
                <tns:value>Asian</tns:value>
        </tns:columnvalues>
        <tns:function>
                <tns:count>1</tns:count>
        </tns:function>
  </tns:operation>
 </tns:dataprocess>
</tns:DataProcessor>
```

Table 1 User-defined Metadata xml that specifies operations to be performed on the input

3.4.3 Types of job submissions

Once the user has specified the metadata, he or she can submit jobs against that metadata file. Two different types of job submission are possible: synchronous and asynchronous. The type of job submission that should take place for a user's request depends on the size of the input dataset specified by the user for this job. If the input is in gigabytes and not enough resources exist, then the computation may take longer than the browser's time-out period for synchronous job submission. If the input size is below a certain threshold, synchronous job submission takes place.

For synchronous job submission, the user submits a request to a SERVLET, and the SERVLET submits the job to HADOOP for map/reduce operations. HADOOP returns a Boolean job status value indicating whether the job succeeded or failed. If the job succeeded, the output files are read from the grid and passed to the client to produce visualization. A unique output directory per user request is created by the servlet and submitted to HADOOP; final output from the job execution is written to that output directory.

In asynchronous job submission, as soon as the user submits a request, a job.xml file is created with information such as name of the output directory, input directory path, unique jobid, and metadata xml file location; job.xml is stored on the grid in the /jobs/inbox directory and the unique jobid is returned to the user. The browser on the client side keeps polling the web server via an AJAX call for the status of the job. The servlet in turn polls the /jobs/processing, /jobs/error, and /jobs/processed directories on the grid to check the submitted job's status with the unique job id. If the submitted job is completed successfully, job.xml is moved to /jobs/processed and the unique output directory specified in the job.xml will exist with the final output. A job controller daemon that is running on one of the TASKTRACKER machines checks the /jobs/inbox folder every five seconds for any job.xml files, reads the elements of the job.xml file, and sends them as arguments to the map/reduce program to create and submit the job to HADOOP. Table 2 shows a sample job.xml file. In the table, the "datasetname" tag specifies the location of the dataset, "filename" indicates the file that needs to be processed, "outfile" indicates the output directory location where the final output should be stored, and "configfile" indicates the location of the metadata.xml file that the Map/Reduce class is supposed to use.

Submitting a job to HADOOP consists only of specifying a set of configuration parameters such as which Map, Reduce, and Combine classes to use, which input format to use, the location of the input files, the name of the file where the output should be stored, and the boundary condition of the split, if any. Appendix B shows configuration parameters that we used in our project.

HADOOP creates the jobid folder in the /jobs/processed folder if the submitted job is successful so that the browser polling the /jobs/processed folder can read the folder to get the final output. For each submitted job there can be multiple output files created in the jobid directory since HADOOP creates as many output files as the number of reductions.

```
<?xml version="1.0" encoding="UTF-8"?>

<job:jobconfig xmlns:job="http://job.sjsu.edu/jobschema">


        <job:datasetname>
                /etc/userdatasets/sandhya/datasets/pums1
        </job:datasetname>

        <job:filename>
                /etc/userdatasets/sandhya/datasets/pums1/
        </job:filename>

        <job:jobid>
                jobid-1205344163435
        </job:jobid>

        <job:outfile>
                /jobs/processed/jobid-1205344163435
        </job:outfile>

        <job:configfile>
                /etc/userdatasets/sandhya/metadata/ethnic.xml
        </job:configfile>

</job:jobconfig>
```

Table 2 job.xml created with the user specified information

3.4.4 How to submit a job to HADOOP from TOMCAT

In the initial implementations of HADOOP, online job submission from Tomcat was not possible since HADOOP was not able to find the user-defined map and reduce classes and was throwing a “ClassNotFoundException.” The reason for that exception is that HADOOP could not load classes that were in the traditional web application’s classes folder or from a jar in the lib folder. In addition, placing the hadoop-jar file in the web application’s lib directory conflicted with the web container’s lib files because hadoop-jar contained JETTY webserver’s APIs, which conflict with Tomcat APIs. Removing the conflicting APIs from hadoop-jar did not alone resolve the problem although it is part of the solution. HADOOP was not able to load the classes that Tomcat's CLASSLOADER was loading. We had written a new URLCLASSLOADER that can obtain the classes from Tomcat's CLASSLOADER. The URLCLASSLOADER creates a class path, copies the classes from Tomcat's CLASSLOADER, and passes them to the HADOOP's JobClient. However, the JobClient did not load the classes even though it was running in the same JVM and context as the SERVLET that initiated the request. In attempting to solve this problem, we dug into HADOOP's source code and found that HADOOP's class loading mechanism is limited to loading only one jar file during the map-reduce request. Therefore, we had to prepare a single large jar file that contained the class files that we have written and all the dependant jar files that our classes use, place the large jar file in the web application's lib folder, and load this big jar file through the URLCLASSLOADER. HADOOP developers are working on being able to accept more than one jar file during a map-reduce request.

As a result of these conditions, to enable the online job submission from the Tomcat web server to HADOOP, we need to include hadoop-site.xml and hadoop-default.xml, in which the HADOOP configurations to run a job are specified, in the web application's classes folder. Apart from that, it is necessary to include the cleaned up hadoop-jar file and its dependent jar files and one big jar file that contains all the map/reduce classes and its dependent jar files in the web application's lib directory.

3.4.5 Generic Map/Reduce Program

This program is the major ingredient of the project; in it, user-submitted jobs are processed. Only three classes are passed to all the machines on the grid. The first is the class that extends MapReduceBase and implements Mapper, the second is the Combiner class that extends MapReduceBase and implements Reducer, and the third is the class that extends MapReduceBase and implements Reducer. The class that implements Mapper does the mapping-related work, the combiner acts like a local reducer for the map class, and the class that implements Reducer does the reduction-related work. Since only these three classes are distributed to all machines on the grid, any computation logic has to be included in these three classes. From now on, I refer to the class that implements the Mapper as the Map class, the class that does the local reduction as the Combiner class, and the class that implements the Reducer as the Reduce class.

Since only the Map, Combine, and Reduce classes are transmitted to the machines on the grid, any modification of global variables or any parsing of data that is done outside the Map or Reduce class cannot be executed on the grid. Therefore, metadata xml, where the user specifies operations and the data that need to be read, is parsed in these classes. Once a job is submitted to the grid, the input is split into multiple chunks and is distributed among the available machines. If the user does not have specific conditions for the split, each chunk is terminated at some calculated offset of an input file. At each TASKTRACKER, the user-defined Map class is executed, and whatever functionality is defined in that Map class is processed on the chunk that is given to the machine. This works perfectly when a user does not have any data dependencies in the executing file. If any dependency exists in the file, there is a high probability that dependant data is split across multiple chunks and are processed by different machines. In this case, the calculated result might be inaccurate. To overcome this problem, we have implemented a logical boundary-based split to ensure that all related data is executed on one machine. The boundary-based split is explained in detail in later sections. This boundary-based split releases the Map program from having to take extra measures to get all the data that is related.

Before the Map class is called, the HADOOP framework internally generates a java long number as the key and the value for the key is the offset of the first byte of the line with respect to start of the file. Essentially, each line of a chunk is assigned to a java long number. These generated key/value pairs are passed to the user-defined Map class. The user-defined Map class reads values from the formal argument, applies any selection criteria the user has specified, reads the column positions, and prepares a set of intermediate key/value pairs as specified in the metadata xml. The way we implemented the intermediate key for ease of grouping and sorting is if the user has specified a key in metadata.xml, the intermediate key is the user-defined key; otherwise, the matched value is the intermediate key. In this Map class we perform operations such as SUM, AVERAGE, MEDIAN, and COUNT on a small set of data typically the size of one line and pass it the combine phase. We use the combine phase to perform operations similar to the reduction phase but with the in-memory data processed by the map phase. The goal is to enhance performance and reduce network latency in the data transfer since the combine task generates a smaller set of output values passed to the reduction phase. The reduction phase performs the specified operations on the multiple map sets. We implemented the Map class such that it creates a mapping with the user-defined key/value pairs and passes this mapping to the Combine class and then on to the Reduce class.

In the Reduce phase, the reduce method reads the key and set of values for that key, performs the specified function, such as COUNT or SUM, and writes the final output as one value per key to the output file specified by the user. The Map class does not call the reduce class directly. Before the user-defined reduce class is called, HADOOP collects all the outputs of the same key from all the mappers, sorts them either by user-defined partitioning function or by HADOOP's predefined partitioning function, and then passes the final key and a set of values for that key to the user-defined combine class. After the combine class is done, the key/values are passed down to the user-defined reduce class. The user-defined Reduce class executes the specified operation on the set of values for a key and writes to the output file.

While submitting the job to HADOOP, the user can specify to perform computations only on one file or on a set of files in a directory. When the user specifies only one input file, then it is trivial that we submit one job. However, when a user specifies multiple files, submitting one job per file would be very inefficient since the overhead to initiate the Map tasks is repeated for each file and the metadata.xml is parsed repeatedly in the Map class. In addition, if there are multiple users submitting multiple jobs and we create a single job per file, the grid would be overwhelmed with multiple jobs, and multiple map/reduce tasks per job and the response time of the grid would increase.

HADOOP can take a directory as an input and execute all the files at once. If we choose to submit the whole directory as one job, then the keys for each file should be different in order to separate the output for each file in the directory. The reason for this restriction is that once a directory is given to the job, the data is partitioned logically to a certain size, and each partition is executed on different machine. Machines on the grid do not differentiate which partition belongs to what file, so all these machines would write the value to a single key. This approach would be useful to calculate aggregate values across multiple files. However, if we want to separate each individual file's output, we need to be able create a set of unique keys per file. For instance, if a user wants to calculate how many Asians and Hispanics live in each state, we must prepare two unique keys for each state.

Initially, we have considered the option of choosing a unique identifier per file and attaching at the end of each line in that file. For example, we could attach a unique state code at the end of each line of a file. Nevertheless, this approach introduces lot of overhead for attaching extra information at the end of every line in a file, not to mention that we are indeed changing the user's data and we had to repeat the operation for every file in the dataset.

The other solution we have considered is to record the actual filename while defining the chunk. If we attach the file name to the user-defined key in the Map class, even though multiple machines are processing two different chunks of a same file, if we know to which file the chunk belongs, separating the output among the files becomes easy.

Let us go through an example of how the Generic Map/Reducer executes a sample job. Let us consider a simple scenario in which the user wants to calculate the total number of Hispanics, Asians, and African Americans who live in each state of the US. The user defines a metadata.xml with the above information by specifying Asian, African American (AA), and Hispanic as three different keys for each state. The user also specifies the criteria for a line to be selected for processing and the column positions in that line that the Map class should read from to differentiate amongst the keys. The GenericMapReducer program takes the file name of the chunk this machine is processing and prepares a key such as Asian_filename, AA_filename, or Hispanic_filename and reads the values in the column positions specified for each key and adds them to a map. In the combine phase, all the values for the keys Asian_filename, AA_filename, and Hispanic_filename that are executed on local Mappers are collected, sorted on the key names, and sent to the reduce function of the Combine class. The combine class performs the COUNT operation on the set of keys given to it by the local map task. These keys are a subset of the overall keys for the file, the rest of which are spread across multiple map tasks. The Combine class then sends the calculated sub-total to the Reduce class as a key/value combination similar to what is given earlier by the Map phase to the Combiner phase. The Reduce class performs the COUNT operation on the set of sub-totals for each key and writes the output in the user-specified location. The result in the output file is one value per key.

The GenericMapReduce program does not guarantee the output to be in the same order in which the keys appear in the input dataset. The reason is that prior to the reduction phase, the output from the Map class of GenericMapReduce program is sorted on the intermediate-keys by HADOOP. Therefore, the order of lines in the output file changes from that of the input file.

Let us explain this problem with an example that calculates pixels on the screen from geographic latitudes/longitudes. In this example, every state has a file that contains a set of latitudes/longitudes separated by spaces. Each line has a geographic latitude/longitude. If we combine all these latitudes and longitudes and draw a polygon, we can visualize a state's boundary. However, we must convert the geographic latitudes/longitudes to pixel coordinates that can fit on the screen in the order in which they appear in the file. Therefore, when we run a formula on this set of latitudes/longitudes to generate the x, y coordinate pixels on the screen, the output size would be the same as input size. In this case, the order of the coordinates matters since we draw a polygon with these coordinates, and if the order of the coordinates changes in the output, the boundary of a state changes. In addition, the coordinates for different states should not be intermixed. To overcome this problem, we have used HADOOP's generated key instead of the user-defined key so that sorting is performed on the unique sequential key and so that the order of the output file does not change from the input file. However, this approach has resulted in inaccurate output because HADOOP's generated key is repeated across multiple files. To understand this problem we must investigate how HADOOP internally generates the key.

3.4.6 How HADOOP generates a unique key per line in a chunk

Once the input data is split into multiple chunks, each chunk is distributed to a different machine. When a machine starts processing the input chunk, it takes the offset of the first byte in the line with respect to the beginning of the file as the starting key and increments the key by the number of bytes in the line. For example, let us assume that a file with 100 lines is split into two chunks such that chunk1 contains lines 1-50, chunk2 contains lines 51-100, and each line in the file is 10 bytes. Let us say that chunk1 is executing on machine1 and chunk2 is executing on machine2. In this situation, machine1 generates the key starting from zero since this chunk's first line is beginning of the file and increments the key by 10 since the number of bytes per line is 10. Machine1 increments the key until it finishes the chunk and it restarts the key generation when it reads another chunk. On the other hand, machine2 generates the key starting from 51 since its chunk's first line and increments the key by 10 since the number of bytes in each line is 10.

From the above explanation on how HADOOP generates the key internally, if we submit multiple files in a single job, each file's start position is zero; two different file's start index for the key being generated would be identical. Therefore, in the reduction phase, two different file's outputs will contain the same key, and we will receive inaccurate results at the individual file level output.

To overcome this problem, we modified HADOOP's functionality so that the key will be the line number of the file in combination with the filename of the chunk that the machine is processing. Even though some of the indexes of different files are identical, the keys are different for different files because the filename is unique in a directory.

Now if we submit coordinate calculation for multiple files in one job, the key for every line of a file is always unique, and the order of the input file's content is not altered in the output.

3.4.7 Input Partition

To distribute tasks among the machines on the grid and to allow multiple machines work concurrently on the input, the input is partitioned into multiple parts and each part is given to a different machine on the grid. The idea behind splitting the input is that each machine can execute tasks equally so that the load can be balanced among the machines on the grid. For this purpose, the input file is logically split into multiple splits as explained below.

As soon as HADOOP receives a job to perform, it reads the input location from the configuration of the submitted job. If the input path is the directory, HADOOP reads the input files in the specified directory one by one and calculates the total number of splits. The maximum split size is limited to the block size of the chunk that the user specifies in hadoop-default.xml file. The size of the split is calculated by taking the total number of bytes in the input directory into account. HADOOP logically splits the input in such a way that all the splits are of the same size other than the last split. Only the metadata of the input split is stored before the spilt is distributed to a TASKTRACKER that would process the split. The metadata of each split contains the name and absolute path of the file on the grid, the start index of the split, and the length of the split. The actual bytes are read after the split's metadata is distributed to a TASKTRACKER. Since the time it takes to process each task and the overhead of creating the task and distributing the task is significant, keeping the split size large enough helps in finishing the job fast. The ideal minimum split size is 64MB [1], which is equal to the size of a block of files on the Distributed File System (DFS). The general formula applied for splitting a regular text file follows:

goalSize = totalSize / (requestedNumSplits == 0 ? 1 : requestedNumSplits)

SplitSize = MAX (minSize, MIN (goalSize, blockSize))

Where minSize is the minimum split size as configured in the hadoop-default.xml or overridden in hadoop-site.xml

requestedNumSplits is the number of map tasks that should be run ideally.

blockSize is the default blocksize as configured. We choose to keep it at 64MB.

Each split is then given to a Map task.

3.4.8 Challenges with Splitting on the Dataset

As explained above, in HADOOP splitting is done irrespective of any conditions or boundaries defined on the input. In addition, each split is executed on a different machine, which limits the type of applications that can run on the grid. In other words, if the first line in the input file is dependant on the consecutive lines upon a starting condition, and if the file is partitioned at a certain size, there is a high probability that the dependant lines might be shared among multiple partitions, which might change the end result. For instance, in the census dataset, each file has multiple lines, and Housing records start with an H and person records start with a P. Multiple lines that start with a P would follow the lines that start with an H. To calculate the total number of houses and the average number of people that live in a house, the same machine should process all the P lines that have a parent H line. If this file is split at a random number of bytes, there is a high probability that some P lines with the same parent might be placed in different splits, which would change the results. Another example is that, to calculate the average income per house, we have to calculate the total income per house and divide that with the number of houses that are in that file. In this scenario, to calculate the total income, we have to process the persons that live in a house and their income. Since every P line is dependant on its parent H line, splitting a file with a specified boundary condition is vital in these situations. Although a grid is suitable for non-sequential batch jobs, being able to process files that have a repetitive pattern increases a grid's capability. This follows the standard data-mining pattern:

```
cat * | grep | sort –unique –c > outfile
```

The following paragraphs explain how to perform the input partitioning with a specified boundary without imposing significant overhead.

To split the input with a specific boundary, we must be able to define a condition on the data. In addition, the condition should support some repetitive pattern. Here we need to be able to differentiate the condition that should be met when calculating an output versus the condition that can be used to split a file. The main goal of splitting the input with a condition is that all the dependant data to perform a calculation be distributed to the same TASKTRACKER.

We considered three options to perform the condition-based splitting. The first two options try to change the split size once the split is distributed to a machine. The advantage of the first two approaches is that, HADOOP's calculation of split metadata is not modified, so no overhead is imposed on the job execution. Third option overrides HADOOP's current split calculation but comes with overhead.

1. Specifying only end condition.

2. Specifying start condition and end condition of a line for splitting.

3. Preprocess the input file looking for the matching conditions, and prepare the metadata of the split ourselves on a single machine.

4. Prepare split metadata ourselves on the grid.

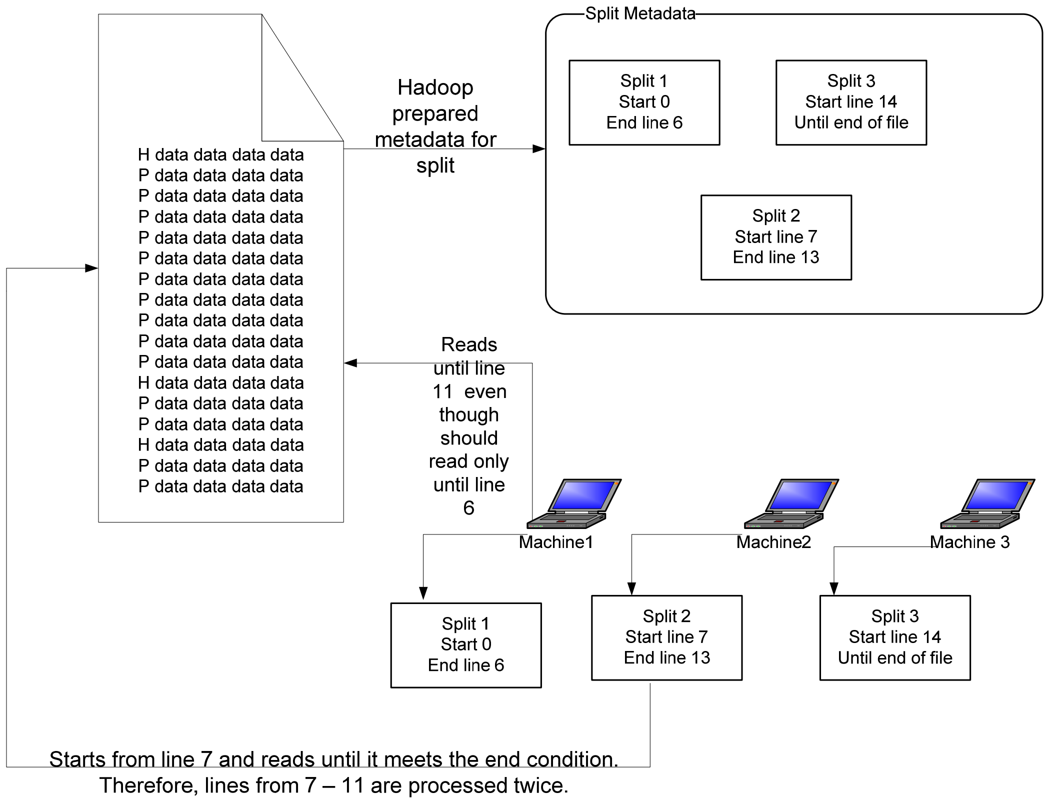
Now we analyze the pros and cons of each of the above specified options.

3.4.8.1 Specifying only the end condition

The condition that is specified is for a line, which means the condition should indicate split the file only when the line ends with some specified condition. If the user wants to split before a line beginning with H, then the end condition should be specified as \nH. In this case, a TASKTRACKER reads the chunk until the next H starts. This method ensures that the patterns for H and the corresponding P lie in the same flow and relative matching can be achieved. However, some obvious drawbacks exist in this approach.

One obvious scenario is when two machines are processing two splits concurrently and the ending H in split1 actually occurs past the beginning index of split 2. As a result, two machines might process some lines twice. Figure 4 explains this scenario. Another drawback is that some of the splits would be very large, creating load balancing problems. If the input is hundreds of gigabytes and the end condition matches after 100 GB, only one machine has to take the load of this 100 GB split, resulting in load imbalance.

The above explanation leads to option two, in which we take both start condition and end condition; if the condition is not matched even after the chunk size is read, read up to 1MB of data from another chunk to meet the condition.



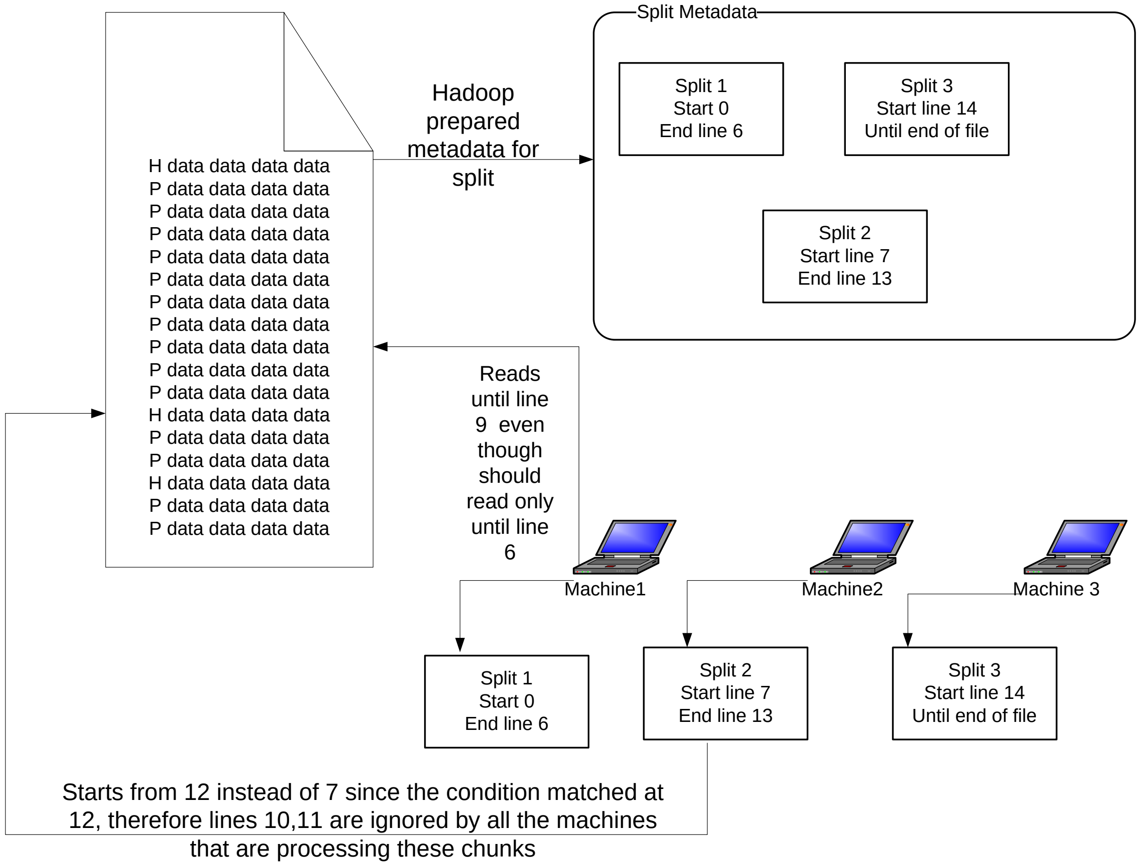
Split Metadata
Split 1 Start 0 End line 6
Split 3 Start line 14 Until end of file
Split 2 Start line 7 End line 13
Hadoop prepared metadata for split
H data data data data
P data data data data
P data data data data
P data data data data
P data data data data
P data data data data
P data data data data
P data data data data
P data data data data
P data data data data
P data data data data
H data data data data
P data data data data
P data data data data
H data data data data
P data data data data
P data data data data
Reads until line 11 even though should read only until line 6
Machine1
Machine2
Machine 3
Split 1 Start 0 End line 6
Split 2 Start line 7 End line 13
Split 3 Start line 14 Until end of file
Starts from line 7 and reads until it meets the end condition.
Therefore, lines from 7 – 11 are processed twice.

Figure 4 Split reading when only end condition specified

3.4.8.2 Specifying start condition and end condition for a line

In this scenario, the user has to specify the start condition and end condition of a pattern so that every block is split at one line before the condition. If the condition matches at the first line of the file, which would be the beginning of the file, start reading all the lines until it reaches 1MB less than the specified split size. From then on, start matching for the condition at every line so that the last block does not exceed the total split size. If the split size exceeds and the condition did not meet, we add a buffer size of 10% of the split size to accommodate more data until it matches the condition. The need for that extra buffer is created by the repetitive nature of the data within each split-size. If the condition matches for the first time before the extra buffer is read, we stop reading data into that split. If the start index of a split does not match the condition, read the lines until the start condition matches and ignore these read lines. Start the split from the position where the start condition matches. Ignoring the lines that do not match a condition ensures that more than one machine on the grid do not process these lines.

Let us explain the above scenario with an example from the PUMS dataset. In the PUMS dataset, housing record lines start with an H and are followed by a set of related person records. Therefore, the user specifies the start condition as H and the new line as the end condition. Now our split condition as a regular expression is ^H.*[\r\n]. When a machine reads the first line of a split, if it is beginning of the file, the condition matches and the machine reads all the lines until it reaches the last 1MB. Within the last 1MB of data, the condition is checked at every line to ensure that the H line and all its dependant P lines can fit into the specified split size. If the H line and all its dependant P lines cannot fit into the specified split size, we add a buffer of 10% of the split size to accommodate the H line and all its dependant P lines. If the first line that is read does not meet the condition, the machines have to ignore the lines before the condition matches. This ensures that all the H lines and the P lines dependant on these H lines go to one machine and avoids the multi processing of these lines.



Split Metadata
Split 1 Start 0 End line 6
Split 3 Start line 14 Until end of file
Split 2 Start line 7 End line 13
Hadoop prepared metadata for split
H data data data data
P data data data data
P data data data data
P data data data data
P data data data data
P data data data data
P data data data data
P data data data data
P data data data data
P data data data data
P data data data data
H data data data data
P data data data data
P data data data data
H data data data data
P data data data data
P data data data data
Reads until line 9 even though should read only until line 6
Machine1
Machine2
Machine 3
Split 1 Start 0 End line 6
Split 2 Start line 7 End line 13
Split 3 Start line 14 Until end of file
Starts from 12 instead of 7 since the condition matched at 12, therefore lines 10,11 are ignored by all the machines that are processing these chunks

Figure 5 Split reading when both start and end condition is specified

The main drawback of the above scenario is that the splitting cannot be done on the datasets that do not have a start condition. The second drawback is that if the condition does not match even after adding the 10% buffer size to the actual split size, the split takes place after the 10% limit is reached. The second machine reading the consecutive split does not know the position at which the previous split was read and ignores these lines. In that case, some of the data is not processed at all, which produces an inaccurate result. Figure 5 explains this scenario. In the example explained above, once the line starting with H is encountered but did not get the same condition even after the added 10% buffer size is reached, this portion of the data is split as soon as the 10% buffer is reached. As a result, the P lines that depend on the H line were separated into another split. In addition, the other machine ignores the rest of the P lines that depend on this H line since they do not meet the match condition, and the other machine thinks that these lines are to be read by another machine. The third drawback is that there is a little overhead to read and check for the condition in the last 1MB of data and the first few lines. This overhead adds up when multiple map tasks are reading data and trying to match conditions on the same file. The load is clearly seen on the TASKTRACKERs as well as the data nodes that actually supply the data from the DFS.

3.4.8.3 Prepare metadata on the machine that has submitted the job

Until this point, we have been utilizing the metadata of the split that is prepared by HADOOP and have tried to manipulate the metadata after the task is distributed to a machine on the grid. Now let us consider the scenario of preparing the metadata ourselves. We must read the file on the machine that has submitted the job, match the specified condition, and prepare the metadata. For instance, to match the condition ^H$\\n, when a client submits the job, open the input files and start reading the lines of each file so that a split's metadata ends exactly when a condition is matched. In this case, no conditions need to be matched at the time the TASKTRACKER is reading the input.

The worst case for this scenario occurs when the input is too large; a single machine takes a long time to read the lines in the input, match for the condition, and specify the split's metadata. The second problem is some splits might be too big, resulting in load imbalance. The third problem is reading all the input on one machine by the client imposes significant overhead on the client machine.

3.4.8.4 Calculating Split metadata on the grid

Since the metadata preparation on a single machine takes a long time and defining the metadata without any condition and matching for the condition at the time of distributing the task results in inaccurate results, we consider the option of performing the split calculation on the grid. That is, when the user requests for visualization, we take the split condition from the user and prepare the metadata for the splits on the grid. We submit a job to the grid to calculate the metadata of the split, such as start index, end index, and total number of bytes. We match the user-specified condition and define a split's start index, end index, and total number of bytes. In this scenario, the split calculation is a pre-processing step and need not be part of the actual job submission. However, this approach does not overcome the problem of load balancing since some of the splits can be larger than others. Nevertheless, one potential advantage of this approach is that since we know the actual split size before the split is distributed to the TASKTRACKER, we can consider the worker machine's available CPU, memory, and network bandwidth before we submit a large split to a TASKTRACKER. In this way, we can ensure that bigger splits can be executed on a machine with high performance and high network bandwidth.

3.4.9 The option we choose

After considering the four options specified in section 3.4.8, we have chosen option 4, which calculates split metadata on the grid. Calculating split metadata on the grid overcomes most of the disadvantages of options 1, 2, and 3. This section describes how we implemented split calculation on the grid.

Implementation of Split calculation on the grid

We chose to take the starting condition and ending condition as input parameters that can be specified in the job configuration. We then implemented this approach as two phases.

1. A job to calculate the logical split
2. An InputFormat to read the calculated logical split

3.4.9.1 Logical Split Map/Reduce program

The logical splitting mentioned earlier is most optimal if performed on the grid. We took the default implementation provided out of the box by HADOOP and ran our Map/Reduce program, LogicalSplitter. For every map task in the LogicalSplitter, we obtained the filename on which the map is executing, the start offset of the map as calculated by HADOOP, the length of the offset as calculated by HADOOP, and the start and end conditions provided by the user at configuration time. Every time the map method was called on the LogicalSplitter, we look for the starting condition and calculate the starting offset as follows:

$$\text{Logical\_start} = \text{start} + \sum_{i=1}^{n} (\text{ no.of bytes per } i^{th} \text{ value}) \qquad \text{--- (1)}$$

Where start is the optimal start calculated by HADOOP

N is the total number of bytes between the start and start + length (where length is the optimal length calculated by HADOOP)

And $1 \le i \le N$

Once start is marked, we looked for the end, starting from the very next byte of start, until we find the end or the map is finished, in which case we would not have found the end in the current split. If logical_end is found we calculated that as follows:

$$\text{Logical\_end} = \text{Logical\_start} + \sum_{i=1}^{n} (\text{ no.of bytes per } i^{th} \text{ value}) \qquad \text{---(2)}$$

The conditions defined in (1) holds for (2) also.

We then created an intermediate key with the logical start and filename, and we group the results in the reduction phase and write them out to the grid.

Note that it is not necessary to find the end condition in the same map. Although in theory there is no limit to the size of a file on the DFS or to the number of files on the DFS, the current implementation does have a set limit in the number of splits it can have for a single file. We have seen that the number of splits will be a factor of the total number of map tasks that will be executed, which is limited. Due to this limitation, we chose to have one reduction for the LogicalSplitter to provide us with an ease of implementation of the actual split file. During reduction we used the following formula, which involves iteration over the group of obtained keys to find the right start and length pair that will eventually be used by the InputFormat of the Job-submission client. Table 3 explains the logic.

```
BEGIN
        END_FOUND=false;
        PREV_K=null;
        FOR-EACH Ki in (K1, K2, K3..Kn)
            IF PREV_K = null THEN
                PREV_K = Ki
            END-IF
            IF (Vi is NOT empty ) THEN
                END_FOUND=true;
            END-IF
             IF END-FOUND THEN
                   Write  the pair of PREV_K,(Vi-PREV_K) as a valid split
                    PREV_K = null
             ELSE IF PREV_K = null THEN   /* for consecutive maps with null V */
                    PREV_K = Ki
             END-IF
        END FOR
END
```

Table 3 Algorithm to perform logical split

In Table 3, Ki is the start offset of the key of the ith key (the intermediate key is filename + start off set).

Along with the above, we also saved the actual criteria for which this map/reduce operation is valid and the filename to which the metadata belongs.

As mentioned earlier, we require that the user provide the start condition, the end condition, or both. It is obvious that in the case in which neither condition is provided, this discussion is nullified. To simplify the discussion, we break the algorithm into separate sections to discuss these multiple scenarios.

Start condition is mentioned and end condition omitted

In the case in which the start condition is mentioned and the end condition is omitted, the formula given in (1) is the only one that needs to be calculated. Logical_end will be null since (2) will not be calculated. Hence the algorithm in the reduction phase will have multiple Ki's with their corresponding Vi's as null. This case needs to be dealt with separately when all the V's are null. Then, the algorithm changes as specified in Table 4.

```
BEGIN
        PREV_K=K1
        FOR-EACH Ki in (K2, K3, K4..Kn)
            Write  the pair of PREV_K,(Ki - PREV_K -1) as a valid split
            PREV_K = Ki
        END FOR
END
```

Table 4 Algorithm to perform logical split when only start condition is specified

Start condition is omitted and end condition is supplied

The case in which the end condition is the only condition mentioned, we take the first start and ignore the others. In this case, (1) will not be calculated, but to calculate (2) the following is assumed: Logical_start = start; The calculation for (2) is then trivial.

```
BEGIN
        IF all K are EMPTY THEN
              PREV_START=0
              FOREACH Vi in (V1, V2, V3 .. Vn)
                    Write the pair of (PREV_START, Vi-PREV_START-1) as a valid
split
                    PREV_START = Vi
              END-FOR
        END-IF
```

Table 5 Algorithm to calculate the logical split when only end condition is specified

3.4.9.2 LogicalInputFormat and LogicalRecordReader

In the previous section, we discussed how the meta-data is calculated for the LogicalSplit. In this section, we discuss how that meta-data is used prior to the job submission to perform the actual Map/Reduce that the user wants on the input data set.

When the user requests a new job to be performed on a given input dataset, we check the directory holding the metadata and look for the input filename(s) on which the operation will be performed. We chose to have the sub-directory with the same name as the real input file so that the implementation is extendible for multiple reductions in the previous phase. We then iterate through any sub-directories under it to find the actual meta-data files and look to see if a meta-data file is found with the same criteria that the user is currently requesting. If no meta-data file is found, then the job is submitted without any logical splitting. Otherwise, we read the contents of the meta-data file and create the FileSplit with the start and length in it as discussed above. An array of splits is created and passed to the job submission logic.

At the time of reading the contents of the meta-data file, we needed to ensure that even the \r and \n be read and passed as input to the map program. We limited ourselves to processing one line at a time.

We found that the HADOOP implementation of DFS has a lower limit of 16KB of data when a read is called for one byte. To ensure that the line-ends are met properly in different systems, we had to read one byte of data towards the end of the split. To minimize overhead, we found it optimal to declare a buffer size of 16KB for both the input and output buffers used to read and pass data to the map program.

4. PERFORMANCE RESULTS

We have produced different types of visualization requiring computation-intensive calculations. One of the visualizations is to calculate the median income of people that belong to seven different races, the average number of people per house in seven different races, and the average number of people that speak five different languages. These three calculations are performed for each state in USA. The total data that needs to be processed to produce the visualization for 50 states in the USA is 5 GB, and the number of computations, including comparisons, additions, multiplications, and divisions, equals 26,240,015.

We have performed the calculations to produce the above visualization on a single machine with 512MB random access memory, 1.6GHz processor speed, and 10GB of free hard disk space. The single machine took 9 minutes and 30 seconds to finish the computations to produce the visualization, whereas a grid with six machines finished in 20 minutes initially but after performance tuning, finished in 5 minutes and 12 seconds.

Out of the six machines, only four machines performed the computations; the remaining two machines were the masters coordinating data and task distribution on the grid.

4.1 Performance improvement of the grid

4.1.1 Grid computation with Map/Reduce

The performance of the grid depends critically on how we write the Map/Reduce program. In the initial version of the Map/Reduce program, we identified patterns to calculate median income for different races, number of people per household per race, and the number of people speaking a certain language. This information is collected per state. The matched input lines are parsed to extract the required data. An intermediate key with “race+statecode” and a corresponding intermediate value for the data that was extracted were written to the OutputCollector. The intermediate keys looked like “statecode+race” or “statecode+ language.” That is because when sorting happens on these intermediate keys, they are sorted by statecode first and then by the secondary key that is race or language. We ensured that the intermediate keys for race or language do not collide with each other. The way HADOOP works after this is that it shuffles and sorts these intermediate keys and writes them to a common location on the DFS (Distributed File System). Once all the map tasks for the job are done, the reduction phase starts and each key and its corresponding values are processed by one reduction task. We then calculated the mean, median, mode, total number of people speaking a language, and the maximum and minimum number of people per household in the reduction phase. Note that these were calculated per race or per language per state.

In this approach, the number of bytes output by the map is proportional to the input bytes given to the map. The entire output has to be sorted in memory and written to the DFS; this requirement caused severe overhead due to network latency and resource contention during sorting. Therefore, the grid took 20 minutes to finish the job, a result far worse than the single machine’s performance.

In an attempt to increase the job’s performance, we increased the number of parallel map tasks on a machine from two to ten. Although this allowed the grid to execute computations faster, the load on the DATANODES was exhaustive. Hence, the total turn around time for the job completion increased to 20mins. Since the load on the DATANODES created a bottleneck, we attempted to compress the intermediate output from the map tasks. Although the compression decreased the load on the DATANODES, the fact that large amounts of data needed to be compressed resulted in high CPU and memory utilization on the TASKTRACKER machines. This immediately resulted in “java.lang.OutOfMemory” exceptions on the TASKTRACKER JVMs. An attempt to increase the heap size considerably was not fruitful. After analyzing the memory usage pattern in the JVM and the corresponding garbage collection statistics, we realized that JVM tuning is not an apt approach.

4.1.2 Grid computation with Map/Combine/reduce

To improve the performance of the job, we decided to decrease the number of input bytes for the reduce phase. To achieve this, we dug more deeply into HADOOP's map/reduce framework. One lead obtained from the HADOOP documentation was a combiner class. However, the affect of using a combiner class was not documented. From looking at the "TaskRunner" class, which executes a map task or a reduce class, we realized that using a combiner class to calculate intermediate aggregated values would be very effective. Therefore, we wrote a combiner class that processes the map output in memory before writing to the disk. We had to ensure the uniqueness of the keys to maintain data integrity. The result of using the combiner was the number records to be input to the reduction phase decreased from 648757504 records to 685 records. The time taken for processing also decreased significantly from 20 minutes to 13 minutes. Although this is a considerable improvement, the number of bytes output from the map remained the same. Moreover, the number of bytes was same that was sent for sorting of the intermediate keys. Hence, the network latency decreased, but CPU utilization and in-memory processing were high, and the processing time of the job was still far way from the single machine's performance of 9 minutes, 15 seconds.

4.1.3 Grid computation with several other improvements

Having tried all of the above improvements and not achieving a significant performance breakthrough, we analyzed the entire HADOOP code base for vital process improvements. We observed that HADOOP spawns a new process every time a map task is executed. Note that there is one map task for each data chunk. We set the number of map tasks that can be launched to a number equal to the number of splits or number of files, which is comparable with the actual split calculation during job submission [13]. We chose to equate the number of reduce tasks to the number of machines available for the map/reduce process [13]. In addition, we noticed that the process' life is ended as soon as it is done executing the split. Considering that each process is a JVM, this puts significant overhead on a machine that is under load. To maximize the amount of work done in the map phase, we chose to perform most computations in the map phase. To perform computations in map phase, input should be logically split with a boundary condition to get accurate results, as explained in section 3.4.9. We performed a majority of the aggregations, and we maintained sorted order of the keys in the Map class. Finally, we wrote the output towards the end of the mapping phase in the "close()" function that is called once for a map task. Upon doing this, we realized that we could eliminate the combiner phase. This elimination resulted in a huge decrease in memory usage and optimal CPU utilization. The number of records traversing the network dropped significantly, from 1512 to 672. The time taken for the reduce phase was well under a minute. Therefore, it was not essential to make significant performance improvements in the reduce phase.

In addition to the above improvements, we also recognized that accessing data over the network is very expensive. At the time of distributing tasks, HADOOP takes care of sorting the nodes from which the data should be accessed such that the least expensive node is at the beginning. Ideally, the least expensive node would be the local node. If the chunk that is about to be processed by the TASKTRACKER does not reside on the local node, the chunk is transmitted over the network from the nearest DATANODE. To transmit the data, which cumulatively might be several gigabytes, over the network is very expensive. Furthermore, when multiple TASKTRACKERS perform this type of data transfer, the overall turnaround time for job completion is significantly affected. To minimize this data transfer latency, we chose to redistribute the data on the grid at the time of the DATANODE startup. Once we executed the job with 100% local data access, we achieved a job processing time of 6 minutes and 15 seconds. At that point, the grid's performance is significantly better than that of a single machine, 9 minutes and 15 seconds.

During the job execution, HADOOP performs redundant execution of some tasks on different machines as a failsafe. This technique is called speculative execution. We analyzed speculative execution in terms of visualization to see whether we could find different ways to monitor fail-over cases. With this intention, we turned off speculative execution; doing so decreased the job completion time from 6 minutes, 15 seconds to 5 minutes, 12 seconds. Therefore, our overall job execution time was reduced from 20 minutes to 5 minutes, 15 seconds.

4.1.4 Lateral scaling of the grid

Despite the above enhancements, the addition of a new machine with the equivalent or better configuration on the grid improves overall job execution time by about 1 minute. The improvement arises from the fact that the number of splits available for processing is significantly higher than the number of machines running map tasks. This will be true as long as one map task processes one split and the input dataset can be split into logical chunks. Figure C-2 in Appendix C shows the lateral scaling of the grid.

5. VISUALIZATION

Since the dataset provides state level information, visualizing state based information in individual geographic state would be more effective. US census website (http://www.census.gov/geo/www/cob/pu_metadata.html) provided a set of geographic latitudes and longitudes per state in individual files. These set of latitudes and longitudes in a file specifies a state's boundaries. Therefore, parsing the geographic latitudes and longitudes that fits the screen size or the applet size is necessary.

5.1 Calculation of pixels from geographic latitude/longitude

The purpose of this calculation is to convert the given geographic latitudes and longitudes to pixels. From geographic US map, the whole US map is divided into seven latitudes and eight longitudes. Maximum longitude is 130 degrees and Maximum latitude is 50 degrees. In addition, the difference between the latitudes is five degrees and difference between the longitudes is ten degrees. Calculating the pixel value of width of each longitude and latitude is as follows.

Following are the list of variables that we have used to calculate the pixels for drawing US map on the screen.

MAXLON – Maximum value of a longitude on the map

MAXLAT – Maximum value of a latitude on the map

NUM_LON – The number of longitudes

NUM_LAT – The number of latitudes

W – The maximum screen width

H – The maximum screen height

Δ lon – The number of longitudes between each representation of longitudinal line

Δ lat – The number of latitudes between each representation of latitudinal line

No. of pixels between each longitude (pLON) = W / NUM_LON

No. of pixels between each latitude (pLAT) = H / NUM_LAT

The factor to display the longitudinal and latitudinal lines would be

Lon_factor(€lon) = pLON/Δ lon

Lat_factor (€lat) = pLAT/ Δ lat

For a given coordinate (LONi, LATi) we can calculate the corresponding pixel (x, y) as follows:

For a western map LONi is always less than zero. Hence to obtain the absolute LONi we perform LONi = MAX_LON – LONi

Then pixel x = LONi * €lon

Since latitude values increase up north to the poles from equator and the screen measurement of a pixel increases from north to south that is from top of the screen to bottom LATi should be represented as

LATi = MAX_LAT – LATi

Then pixel y = LATi * €lat

To increase or decrease the map size, Δ lon and Δ lat can be adjusted. In our calculation we have taken the Δ lon as 4 and Δ lat as 8 to get a decent map size that can fit in the applet with the size (600, 600).

5.2 US Map visualization

US census bureau has provided boundary files that contain geographic latitude and longitude for all states that are in USA. These coordinates are the boundary coordinates for states. Each state's boundary coordinates are specified in a separate file. We have converted these latitudes and longitudes to pixels on the grid using the calculation specified in section 5.1. WE have stored the output of the converted pixels in 52 separate files, one file per state on the grid and read all the 52 files to draw the map. Since these coordinates are boundaries a particular state, we have drawn a polygon that combines these coordinates. Each state's original boundary file contains different sections that are separated with some special characters. A new section starts whenever a new corner of a state boundary starts. Therefore, when draw a polygon whenever a new section starts. As a result each state internally has multiple polygons in it. However, some coordinates overlap with each other. As a result, the user sees the whole state as one polygon. Since all the calculations are performed on the grid, the applet reads the boundary files from the grid and draws each state one by one. The size of the map varies with the applet size. If the applet size is high map would look bigger. To make the map size relative to the applet/screen's size, user need to perform the calculations specified in section 5.1 on the boundary files. Since we have limited the applet size to 600 x 600, we have calculated the pixels only once for 600 x 600 size and stored on the grid. Figure 6 is such US map that is drawn as polygons. We calculate each state's center coordinate when we are drawing the map that is used later in the visualization.

5.3 Map Visualization with census data analysis

Visualization would be very effective when users want to analyze the census data. Therefore, we have performed such an analysis and displayed in circle format inside each state. Each circle in each state defines some information that relates to that state. We have calculated the employment and unemployment status in each state in USA. We have calculated this information on the grid for each state and stored the output on the grid. The applet reads the output files and calculates percentage relative to the circle and displays on the grid. We have used Java Applet 2D objects to draw the arcs in the circle.

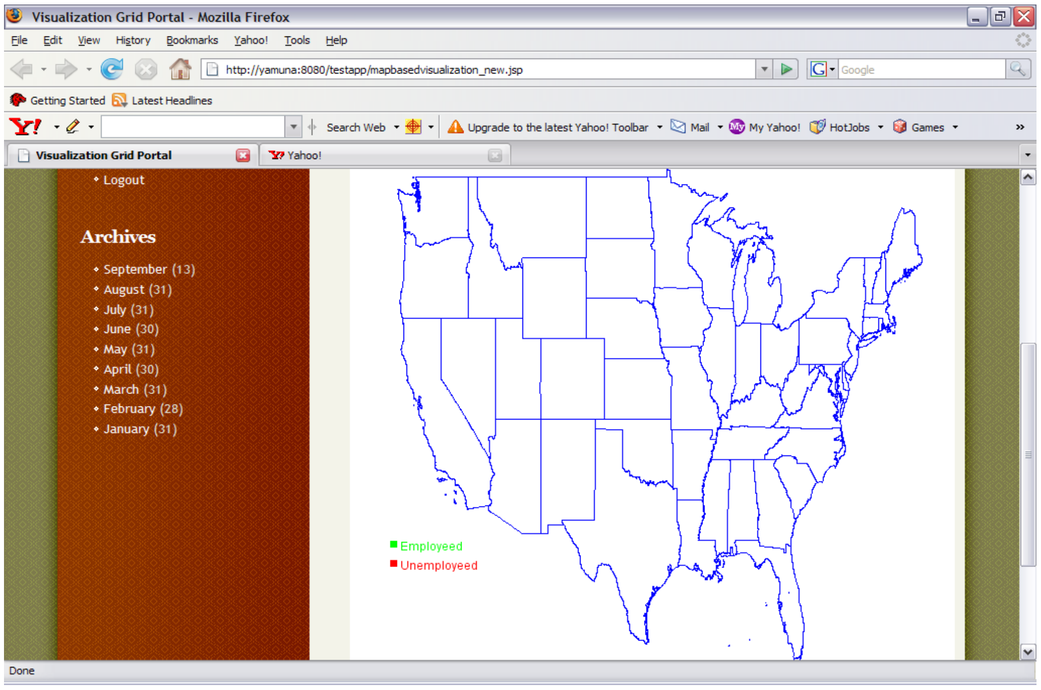

Visualization Grid Portal - Mozilla Firefox
File Edit View History Bookmarks Yahoo! Tools Help
http://yamuna:8080/testapp/mapbasedvisualization_new.jsp
Getting Started Latest Headlines
Visualization Grid Portal
Yahoo!
Logout
Archives
September (13)
August (31)
July (31)
June (30)
May (31)
April (30)
March (31)
February (28)
January (31)
🟩 Employeed
🟥 Unemployeed
Done

Figure 6 USA map that is drawn as polygons

To draw the circle we have to calculate the ARC size and draw the circle ARC by ARC. To draw the ARC, we have to calculate the start position of the ARC in degrees, and the length of the ARC in degrees, height and width of the eclipse. If the height and width of the circle are same then the eclipse becomes a circle. The following algorithm is use to calculate the ARC of a circle.

```
START = 0.0
FINISH = 0.0
FOE EACH VALUE IN Vi (V1, V2, V3)
        FINISH = START + Vi
        STARTING ANGLE F1 = MINIMUM (90, 90 - FINISH);
        LENGTH OF THE ARC F2 = MAXIMUM (90 – START, 90 - FINISH);
        START = FINISH
END FOR
```

Java Arc2D.Double constructs an ARC, initialized to the specified location, size, angular extent and type. To draw the circle inside each state, we have calculated the center latitude and longitude of each state. We have set the circle size as 20/20 which are height and width of the circle, and the coordinates of the upper left corner of the ARC are

Coordinate X = center longitude of the state – width of the circle/2

Coordinate Y = center latitude of the state – height of the circle/2

We construct the ARC using java Arc2D.Double to which we pass the X, Y, Width, Height, F1, and F2-F1. After we construct the ARC we transform the arc and append to the path. The path is drawn once all the arcs are added to the circle.

We have performed an analysis of the employment rate and unemployment rate in each state in USA. Since each state's size in the US map is small we have limited the amount of information that can be shown within each state. Figure 7 shows the visualization of US map with the employment and unemployment rate within each state as circles. From figure 7, green color indicates the employment status and red color indicates the unemployment rate. As can be seen from the visualization, one can say that unemployment rate in very little in all the states in USA.

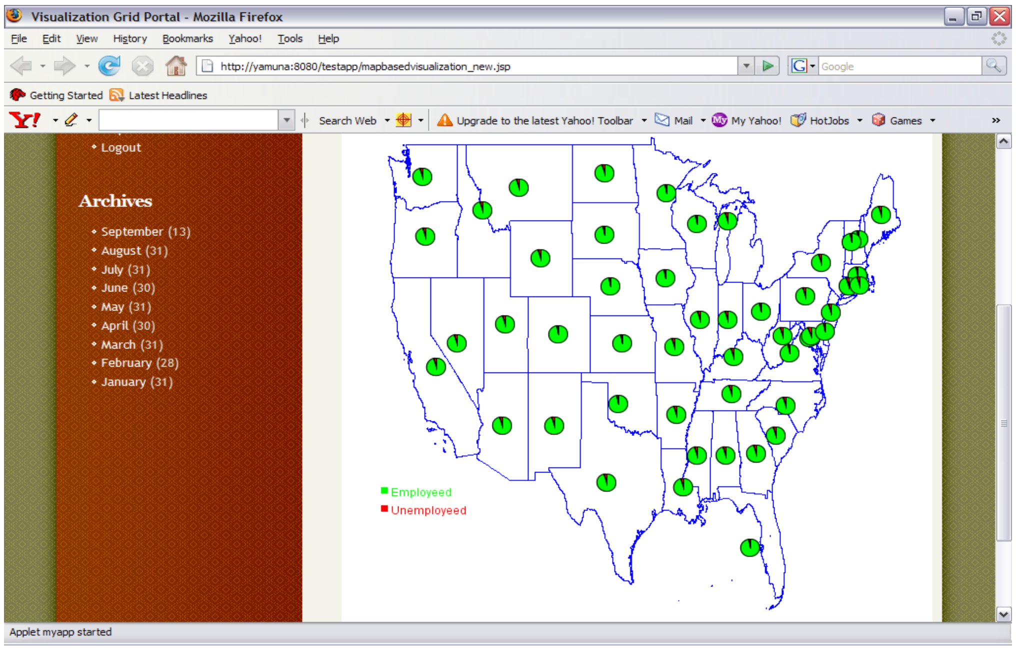

Visualization Grid Portal - Mozilla Firefox
File Edit View History Bookmarks Yahoo! Tools Help
http://yamuna:8080/testapp/mapbasedvisualization_new.jsp
Getting Started Latest Headlines
Search Web Upgrade to the latest Yahoo! Toolbar Mail My Yahoo! HotJobs Games
Logout
Archives
September (13)
August (31)
July (31)
June (30)
May (31)
April (30)
March (31)
February (28)
January (31)
🟩 Employeed
🟥 Unemployeed
Applet myapp started

Figure 7 Visualization of each state in US with employment versus unemployment rate

However, the user is looking at very limited amount of data that relates to each state. If the user is interested in looking into more details then the user can zoom each state separately and can visualize more information that is related to that state. When the user zooms a single state, the whole US map would be visualized in the corner in smaller size to accommodate the bigger single state map. However, the limited information within each state is not distracted by the bigger map. In this detailed version of the state, we have analyzed eight different types of income a person gets in each state on an average. So if the user clicks on the state that he/she is interested in we show these details that relate to that state. Since each state is drawn as a polygon, when a user clicks on a state we capture the mouse click, and get the point where the user clicked. We go through the set of polygons and find to which polygon the point belongs. Once we know the state that is clicked by the user, we multiply the each and every coordinates of that state with 2 to make the state twice as big. When we draw the zoomed state, to get the map into the center of the applet we had to add some correction factor to each coordinate in the state. Therefore, the coordinates of the zoomed state are calculated as

```
FACTOR = 2, ADDVAL = 50
LONGITUDE = CENTER_LONGITUDE * FACTOR
LATITUDE = CENTER_LATITUDE * FACTOR
LONCORRECTION = (WIDTH OF THE APPLET / 2 – LONGITUDE) + ADDVAL
LATCORRECTION = (HEIGHT OF THE APPLET / 2 – LATITUDE) + ADDVAL
FOR EACH COORDINATE IN STATE S ((X,Y), X1,Y1)
        NEW_LONGITUDE = (Xi * FACTOR) + LONCORRECTION
        NEW_LATITUDE = (Yi * FACTOR) + LATCORRECTION.
```

END FOR

In the same way, we multiply each and every coordinate of every other state in the US map with 0.5 to make the US map half the current size displayed. Figure 8 shows the visualization that includes the detailed information of state California and the US map in the corner. In the figure state California is zoomed to see detailed information about that state. In California the bigger state has different income in different colors. Out of those, the bigger part of the circle that is showed in cyan color shows that wage income is higher than the rest of the income types. In the same way the user can visualize seven different types of expenditure also in another circle.

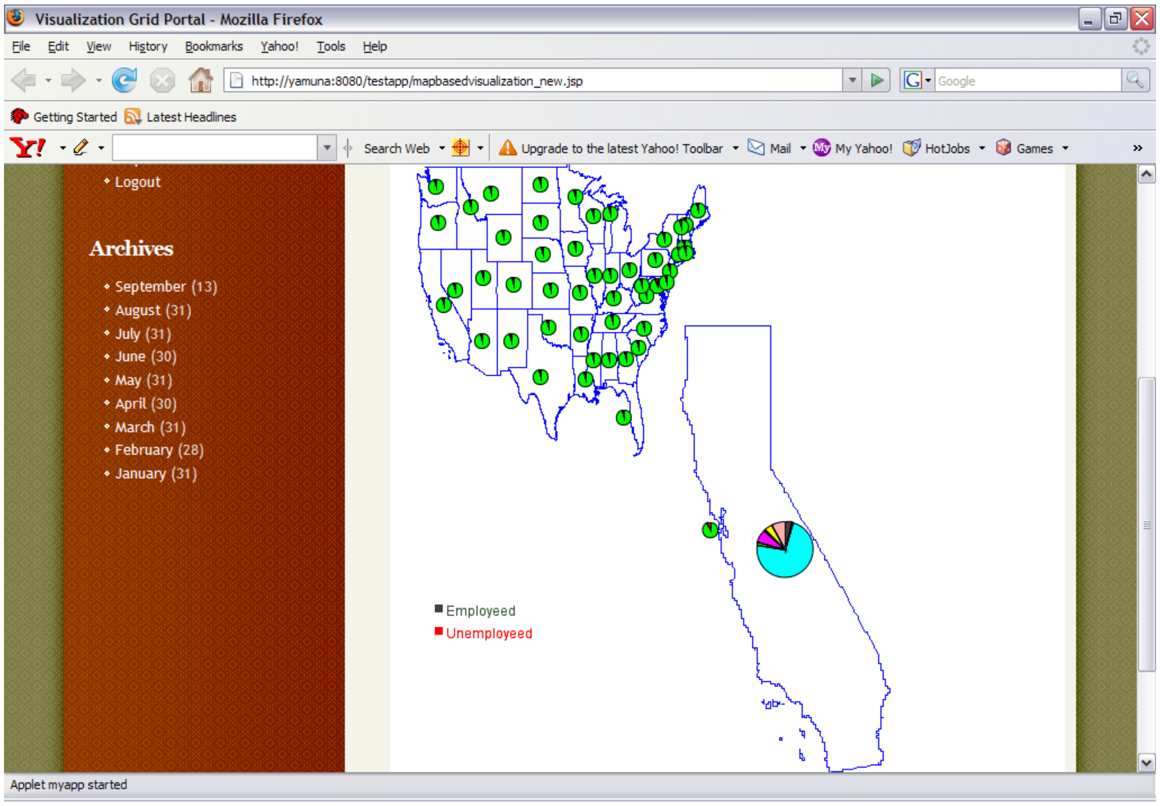


Figure 8 Visualization of each state in US with employment versus unemployment rate

5.4 Visualizing more data in matrix form

Sometimes visualizing different types of information at one time helps to derive some details. We have performed an analysis on seven different races in each state. We have calculated the median income per race within state, and the average number of people that live in a house. In addition, we have analyzed five different languages that are spoken more in each state. All this information is displayed in one visualization. Figure 9 shows the visualization that has all the details described in earlier lines.

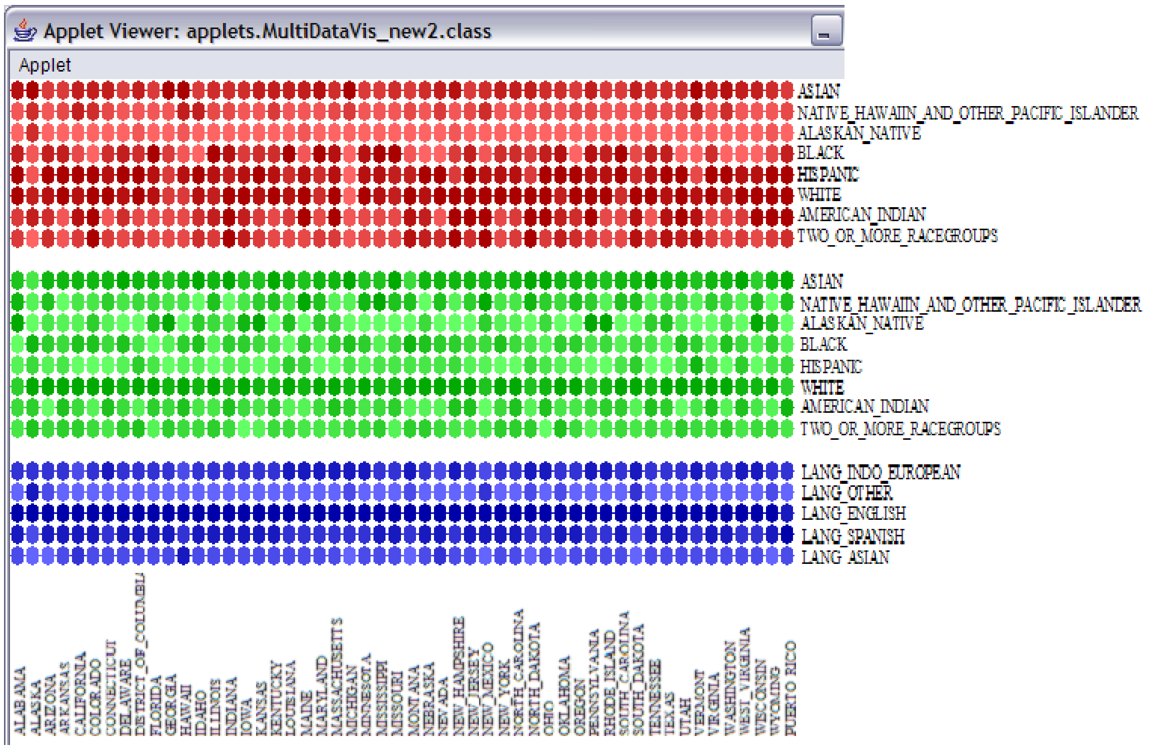


Figure 9 Matrix representation of different ethnic groups, languages that are spoken, and the average number of people per household

In figure 9, RED color represents the number of household that live per house, GREEN color represents the median income of people in each ethnic group and BLUE color represents the languages that are spoken. Each row defines either and ethnic group or the language as shown in the figure and each column represents a state. In addition, some colors are light and some colors are dark. Dark color indicates that the value is higher and light color indicates that the value is small.

To produce the above visualization, we have calculated the median income, average number of people per household, and the number of people that speak each language for each state on the grid. In addition, to produce from lighter color to darker color, we have to have the sorted order from higher value to lower value. This sorting is also done on the grid. Once the calculations are done on the backend, the applet reads the values, assigns a unique color to three sets. The dark color and light color are pre assigned to maximum value of the group and to minimum value of the group respectively. The color to the values that are in between the maximum value and minimum value is assigned by calculating relative position of the value in the array. Once a color is assigned to number that is stored so that the same number gets always the same color. The color is calculated as

DIFF = NUMEBR OF ELEMENTS BETWEEN THE MAX_VAL AND MIN_VAL

R_FACTOR = ABS(MAX_REDCOLOR – MIN_RED_COLOR) / DIFF

```
G_FACTOR = ABS(MAX_GREENCOLOR – GREEN_MIN_COLOR) / DIFF
B_FACTOR = ABS(MAX_BLUECOLOR – BLUE_MIN_COLOR)
FOR EACH ELEMENT IN ARRAY A(A1,A2,A3)
    IF ELEMENT_VAL == MAX_VAL
        ASSIGN DARK COLOR
    ELSE IF ELMENT_VAL == MIN_VAL
        ASSIGN LIGHT COLOR
    ELSE
    COLOR_RED=ABS(MIN_R_COLOR–ELEMENT_IDX *R_ FACTOR)
    COLOR_GREEN=ABS(MIN_G_COLOR–ELEMENT_IDX* G_FACTOR)
    COLOR_BLUE=ABS(MIN_B_COLOR–ELEMENT_IDX *B_ FACTOR)
END FOR
```

6. CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

Grid is useful for processing large datasets with extensive computations. It leverages the power of multiple machines by splitting the input into multiple chunks and by distributing them to the machines on the grid. The grid executes the chunks concurrently. However, the accuracy of the output will be skewed if these chunks are not split with some designated boundaries. Logical splitting helps prevent this problem by splitting the input on a well-defined boundary condition. In addition, multiple lines can be grouped into one record when the boundary is well-defined on the input chunk. As long as the chunks are proportional in size the performance of the computation is on par with HADOOP's default splitting. However, if the boundary condition causes a chunk size to be abnormally large, performance degradation might occur due to an imbalance in load. In order to avoid this, the task distribution should take individual machine's capability into consideration so that powerful machines get bigger chunks to process.

The performance of the grid is directly proportional to the size of the grid. It is possible to complete a job in constant time by determining the number of machines that it has to run on, given the size of the dataset. However, we need to consider the time it takes to distributes tasks across multiple machines, transfer the data on the network to the machines and finally aggregate the output, against the time it takes to process on a single machine, before choosing the grid. If the dataset and the number of computations are too small, the turn around time for the job completion will be higher on the grid than a single machine's turn around time.

Since the dataset that we chose for our project is large, and the number computations that we had to perform to produce the visualization are high, grid was very useful in reducing the turnaround time for the visualization.

6.2 Future work

Since network latency is involved between the master node and the slave nodes communication, it will be helpful for the logical splitting to work more efficiently if we have contiguous blocks replicated across different machines. In addition, if a slave node gets the list of splits to be processed in one request, it can better allocate its map and reduction tasks for these splits.

REFERENCES

[1] Jeffrey Dean, Sanjay Ghemawat, "MapReduce:Simplified Data Processing on Large Clusters", Google, Inc. 2004.

[2] Miguel L. Bote-Lorenzo, Yannis A. Dimitriadis, and Eduarado Gomez-Sanchez, "Grid Characteristics and Uses: a Grid Definition", School of Telecommunications Engineering, University of Valladolid 2004.

[3] Ralf Lammel, "Google's MapReduce Programming Model --- Revisited", Microsoft Corp. 2007

[4] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung,"The Google File System", Google Inc. 2003

[5] Ian Foster, "Computaitonal Grids", Mathematics and Computer Science Division, Argonne National Laboratory.

[6] Leon Erlanger, "Distributed Computing: An Introduction", 2002, http://www.extremetech.com/article2/0,1697,11769,00.asp

[7] Andrew Grimshaw, "What is a Grid?", Avaki Corporation, http://www.gridtoday.com/02/1209/100840.html

[8] NigelDaley, " Hadoop Map-Reduce Tutorial", Apache Software foundation, 2008.

[9] Dhruba Borthakur, "The Hadoop Distributed File System: Architecture and Design", 2008

[10]"Hadoop Cluster Setup", 2008, http://hadoop.apache.org/core/docs/r0.15.3/cluster_setup.htm

[11] DougCutting, " Hadoop MapReduce: How Map and Reduce operations are actually carried out" 2008

[12] AmareshwariSriRamadasu, "TaskExecutionEnvironment" 2008, http://wiki.apache.org/hadoop/TaskExecutionEnvironment

[13] LohitVijayarenu, "How Many Maps and Reduces: Partitioning your job into maps and reduces" 2007, http://wiki.apache.org/hadoop/HowManyMapsAndReduces

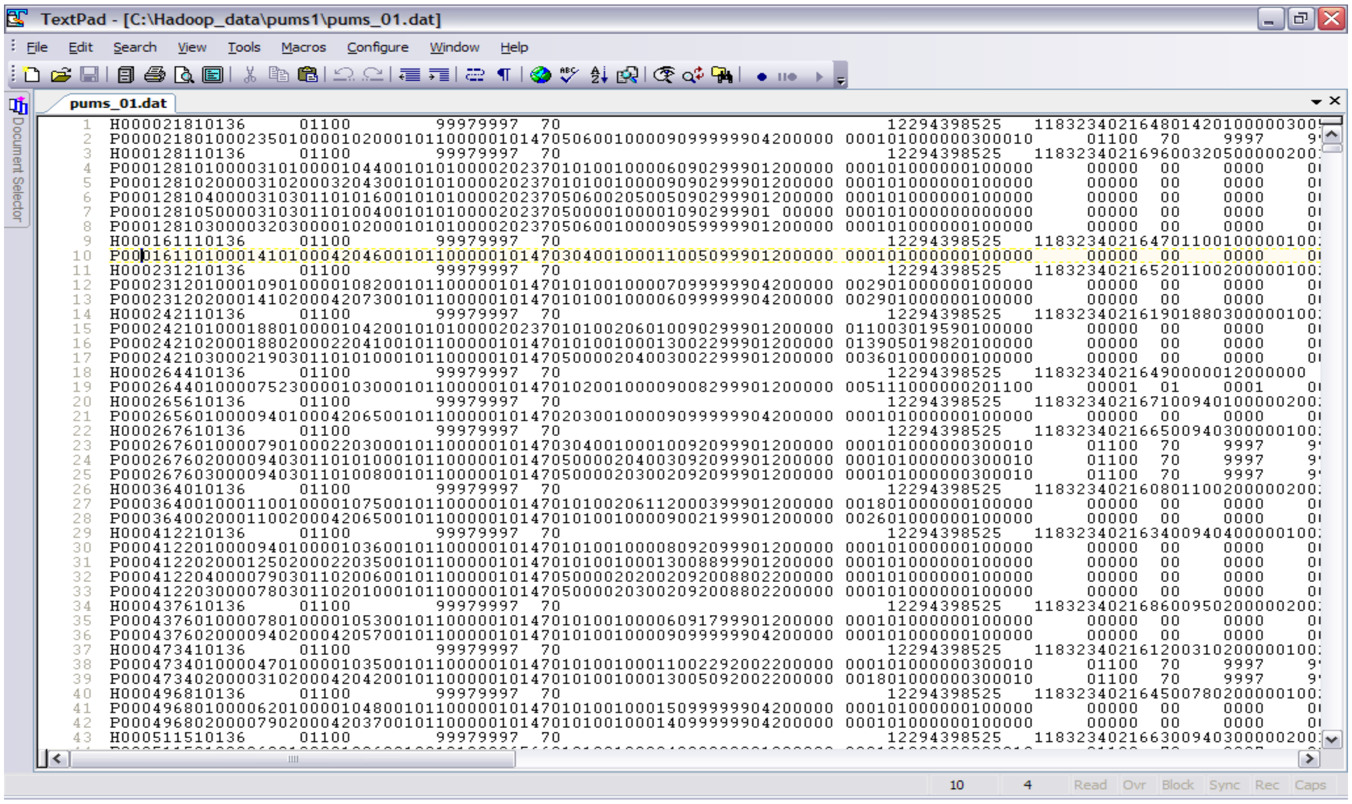
[14] Hans-Ulrich Heiss, Marcus Dormanns, "Partitioning and mapping of parallel programs by self-organizations", Department of Mathematics and Computer Science, University of Paderborn, Germany.

[15] Sebastian Kay Belle, Daniela Oelke, Sonja Oettl, Mike Sips., “Exploration of the local distribution of major ethnic groups in the USA”, http://www.cs.umd.edu/hcil/InfovisRepository/contest-2006/files/Belle.pdf.

[16] Benoit Gaudin, Mike Bennett, Brendan Sheehan, Aaron J Quigley, Member, IEEE, “Π- Flow: Flow & charts for the study of population movement”. http://www.cs.umd.edu/hcil/InfovisRepository/contest-2006/files/Gaudin.pdf

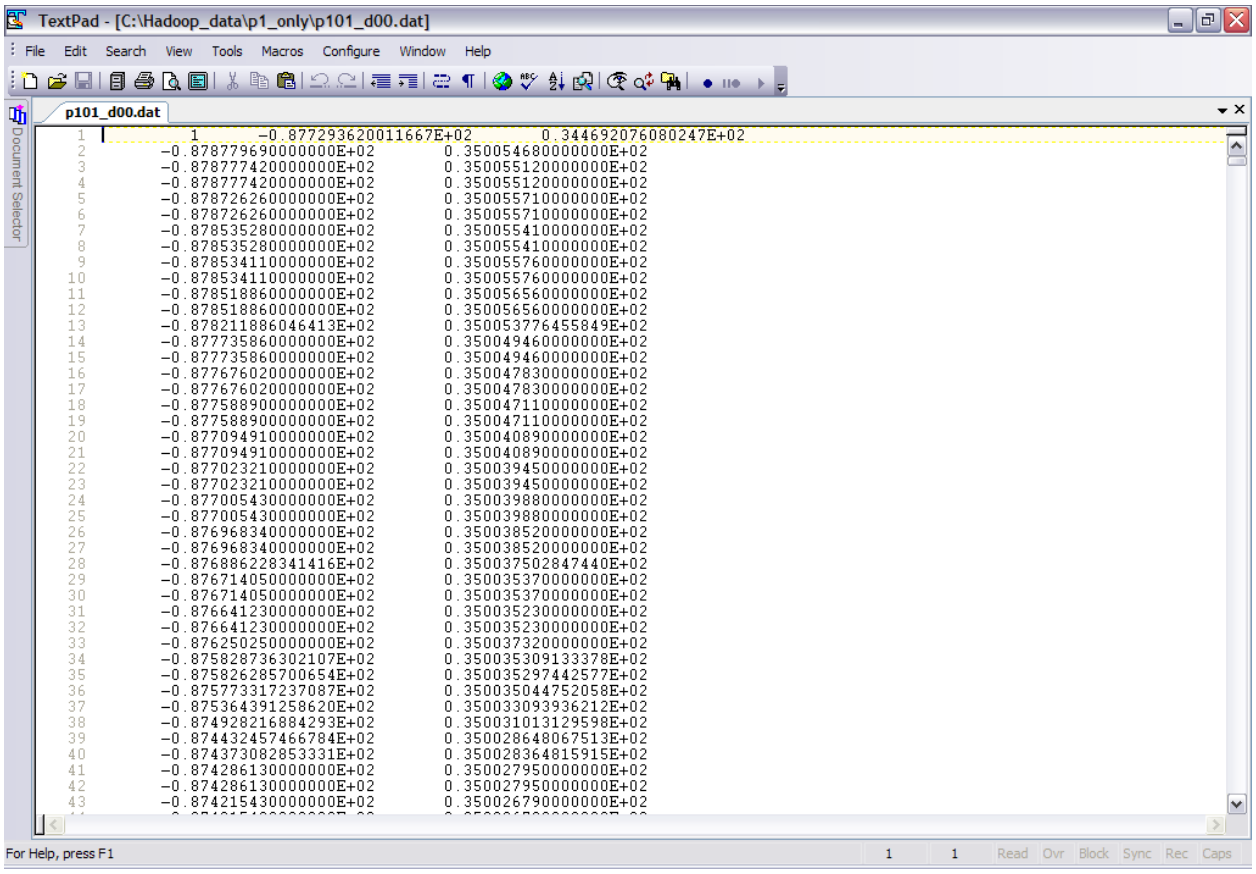
APPENDIX A - Sample input dataset and boundary files

Figure A - 1 Sample entries of input dataset


TextPad - [C:\Hadoop_data\pums1\pums_01.dat]
File Edit Search View Tools Macros Configure Window Help
Document Selector
pums_01.dat
10 4 Read Ovr Block Sync Rec Caps

```
H000021810136     01100       99979997  70                             12294398525   118323402164801420100000300
P00002180100023501000010200010110000010147050600100009099999904200000 000101000000300010     01100  70    9997    9
H000128110136     01100       99979997  70                             12294398525   118323402169600320500000200
P00012810100003101000010440010101000020237010100100006090299901200000 000101000000100000     00000  00    0000    0
P00012810200003102000320430010101000020237010100100009090299901200000 000101000000100000     00000  00    0000    0
P00012810400003103011010160010101000020237050600205005090299901200000 000101000000100000     00000  00    0000    0
P00012810500003103011010040010101000020237050000100001090299901 00000 000101000000000000     00000  00    0000    0
P00012810300003203000010200010101000020237050600100009059999901200000 000101000000100000     00000  00    0000    0
H000161110136     01100       99979997  70                             12294398525   118323402164701100100000100
P00016111010001410100042046001011000001014703040010001100509990120000 000101000000100000     00000  00    0000    0
H000231210136     01100       99979997  70                             12294398525   118323402165201100200000100
P00023120100010901000010820010110000010147010100100007099999904200000 002901000000100000     00000  00    0000    0
P00023120200014102000420730010110000010147010100100006099999904200000 002901000000100000     00000  00    0000    0
H000242110136     01100       99979997  70                             12294398525   118323402161901880300000100
P00024210100018801000010420010101000020237010100206010090299901200000 011003019590100000     00000  00    0000    0
P00024210200018802000220410010110000010147010100100013002299901200000 013905019820100000     00000  00    0000    0
P00024210300021903011010100010110000010147050000204003002299901200000 003601000000100000     00000  00    0000    0
H000264410136     01100       99979997  70                             12294398525   118323402164900000120000000
P00026440100007523000010300010110000010147010200100009008299901200000 005111000000201100     00001  01    0001    0
H000265610136     01100       99979997  70                             12294398525   118323402167100940100000200
P00026560100009401000420650010110000010147020300100009099999904200000 000101000000100000     00000  00    0000    0
H000267610136     01100       99979997  70                             12294398525   118323402166500940300000100
P00026760100007901000220300010110000010147030400100010092099901200000 000101000000300010     01100  70    9997    9
P00026760200009403011010100010110000010147050000204003092099901200000 000101000000300010     01100  70    9997    9
P00026760300009403011010080010110000010147050000203002092099901200000 000101000000300010     01100  70    9997    9
H000364010136     01100       99979997  70                             12294398525   118323402160801100200000200
P00036400100011001000010750010110000010147010100206112000399901200000 001801000000100000     00000  00    0000    0
P00036400200011002000420650010110000010147010100100009002199901200000 002601000000100000     00000  00    0000    0
H000412210136     01100       99979997  70                             12294398525   118323402163400940400000100
P00041220100009401000010360010110000010147010100100008092099901200000 000101000000100000     00000  00    0000    0
P00041220200012502000220350010110000010147010100100013008899901200000 000101000000100000     00000  00    0000    0
P00041220400007903011020060010110000010147050000202002092008802200000 000101000000100000     00000  00    0000    0
P00041220300007803011020100010110000010147050000203002092008802200000 000101000000100000     00000  00    0000    0
H000437610136     01100       99979997  70                             12294398525   118323402168600950200000200
P00043760100007801000010530010110000010147010100100006091799901200000 000101000000100000     00000  00    0000    0
P00043760200009402000420570010110000010147010100100009099999904200000 000101000000100000     00000  00    0000    0
H000473410136     01100       99979997  70                             12294398525   118323402161200310200000100
P00047340100004701000010350010110000010147010100100011002292002200000 000101000000300010     01100  70    9997    9
P00047340200003102000420420010110000010147010100100013005092002200000 001801000000300010     01100  70    9997    9
H000496810136     01100       99979997  70                             12294398525   118323402164500780200000100
P00049680100006201000010480010110000010147010100100015099999904200000 000101000000100000     00000  00    0000    0
P00049680200007902000420370010110000010147010100100014099999904200000 000101000000100000     00000  00    0000    0
H000511510136     01100       99979997  70                             12294398525   118323402166300940300000200
```

Figure A - 2 Sample entries in geographic coordinate file


TextPad - [C:\Hadoop_data\p1_only\p101_d00.dat]
File Edit Search View Tools Macros Configure Window Help
p101_d00.dat
Document Selector
For Help, press F1
1 1 Read Ovr Block Sync Rec Caps

```
         1       -0.877293620011667E+02        0.344692076080247E+02
    -0.878779690000000E+02       0.350054680000000E+02
    -0.878777420000000E+02       0.350055120000000E+02
    -0.878777420000000E+02       0.350055120000000E+02
    -0.878726260000000E+02       0.350055710000000E+02
    -0.878726260000000E+02       0.350055710000000E+02
    -0.878535280000000E+02       0.350055410000000E+02
    -0.878535280000000E+02       0.350055410000000E+02
    -0.878534110000000E+02       0.350055760000000E+02
    -0.878534110000000E+02       0.350055760000000E+02
    -0.878518860000000E+02       0.350056560000000E+02
    -0.878518860000000E+02       0.350056560000000E+02
    -0.878211886046413E+02       0.350053776455849E+02
    -0.877735860000000E+02       0.350049460000000E+02
    -0.877735860000000E+02       0.350049460000000E+02
    -0.877676020000000E+02       0.350047830000000E+02
    -0.877676020000000E+02       0.350047830000000E+02
    -0.877588900000000E+02       0.350047110000000E+02
    -0.877588900000000E+02       0.350047110000000E+02
    -0.877094910000000E+02       0.350040890000000E+02
    -0.877094910000000E+02       0.350040890000000E+02
    -0.877023210000000E+02       0.350039450000000E+02
    -0.877023210000000E+02       0.350039450000000E+02
    -0.877005430000000E+02       0.350039880000000E+02
    -0.877005430000000E+02       0.350039880000000E+02
    -0.876968340000000E+02       0.350038520000000E+02
    -0.876968340000000E+02       0.350038520000000E+02
    -0.876886228341416E+02       0.350037502847440E+02
    -0.876714050000000E+02       0.350035370000000E+02
    -0.876714050000000E+02       0.350035370000000E+02
    -0.876641230000000E+02       0.350035230000000E+02
    -0.876641230000000E+02       0.350035230000000E+02
    -0.876250250000000E+02       0.350037320000000E+02
    -0.875828736302107E+02       0.350035309133378E+02
    -0.875826285700654E+02       0.350035297442577E+02
    -0.875773317237087E+02       0.350035044752058E+02
    -0.875364391258620E+02       0.350033093936212E+02
    -0.874928216884293E+02       0.350031013129598E+02
    -0.874432457466784E+02       0.350028648067513E+02
    -0.874373082853331E+02       0.350028364815915E+02
    -0.874286130000000E+02       0.350027950000000E+02
    -0.874286130000000E+02       0.350027950000000E+02
    -0.874215430000000E+02       0.350026790000000E+02
```

APPENDIX B – HADOOP configurations

Table B - 1 Configurations that we have specified in HADOOP for the visualization project

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
<property>
  <name>hadoop.tmp.dir</name>
  <value>/tmp/hadoop-sanju</value>
</property>
<property>
  <name>fs.default.name</name>
  <value>hdfs://krishna:54310</value>
</property>
<property>
  <name>mapred.job.tracker</name>
  <value>ganges:54311</value>
</property>
<property>
  <name>dfs.replication</name>
  <value>3</value>
</property>
<property>
  <name>mapred.child.java.opts</name>
  <value>-XX:NewSize=50m –Xmx512m</value>
</property>
<property>
        <name>dfs.datanode.port</name>
        <value>yamuna:54312</value>
</property>
<property>
        <name>mapred.task.tracker.output.port</name>
        <value>yamuna:54313</value>
</property>
<property>
        <name>dfs.name.dir</name>
        <value>/cygdrive/c/Hadoop/hadoop-0.15.3/newdfs/newname/newdfsname</value>
</property>
<property>
        <name>dfs.data.dir</name>
        <value>/cygdrive/c/Hadoop/hadoop-0.15.3/newdfs/newdata/newdfsdata</value>
</property>
<property>
        <name>dfs.client.buffer.dir</name>
        <value>/cygdrive/c/Hadoop/hadoop-
0.15.3/newdfs/newclient/newbuffer/newclientbuffer</value>
</property>
<property>
        <name>mapred.local.dir</name>
        <value>/cygdrive/c/Hadoop/hadoop-0.15.3/newmapred/newlocal/newmapredlocal</value>
</property>
<property>
        <name>mapred.map.tasks</name>
        <value>10</value>
</property>
<property>
        <name>mapred.reduce.tasks</name>
        <value>4</value>
</property>
<property>
  <name>mapred.reduce.parallel.copies</name>
  <value>5</value>
  <description>The default number of parallel transfers run by reduce
  during the copy(shuffle) phase.
  </description>
</property>
<property>
  <name>mapred.tasktracker.tasks.maximum</name>
  <value>5</value>
  <description>The maximum number of tasks that will be run
  simultaneously by a task tracker.
  </description></property>
```

```
<property>
  <name>io.file.buffer.size</name>
  <value>10485760</value>
  <description>10 mb</description>
</property>

<property>
  <name>mapred.compress.map.output</name>
  <value>false</value>
  <description>Should the outputs of the maps be compressed before being
              sent across the network. Uses SequenceFile compression.
  </description>
</property>

<property>
  <name>mapred.output.compress</name>
  <value>false</value>
  <description>Should the job outputs be compressed?
  </description>
</property>

</configuration>
```

Table B - 2 Configuration parameters that we submit to HADOOP when no split condition specified

```
JobConf conf = new JobConf(GenericMapReducer.class);
 conf.setJobName("GenericMapReducer" + jobid);
conf.setOutputKeyClass(LongWritable.class);
conf.setOutputValueClass(Text.class);
conf.setOutputFormat(SequenceFileOutputFormat.class);
conf.setMapperClass(MapClass.class);
conf.setCombinerClass(Reduce.class);
conf.setNumReduceTasks(4);
conf.setReducerClass(Reduce.class);
conf.setInputPath(new Path(inDir));
 conf.set(XMLKEY, xml);
conf.set("OUTDIR", outfile);
Path tmpDir = new Path(Constants.TMP_DIR, "job" + jobid);
conf.setOutputPath(tmpDir);
JobClient.runJob(conf);
```

Table B - 3 Configuration parameters that we submit to HADOOP when a split condition is specified

```
JobConf conf = new JobConf(EthnicCalcAll.class);
conf.setJobName("EthnicCalcAll" + outfile);
cnf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(Text.class);
conf.setInputFormat(LogicalSplitTextInputFormat.class);
conf.set("mapred.logicaltextinput.metadatadir",
"/etc/userdatasets/sandhya/metadata/pums5");
conf.set ("mapred.logicaltextinput.condition", "^H.*[\n\r]?");
conf.setOutputFormat(SequenceFileOutputFormat.class);
conf.setMapperClass(MapClass.class);
conf.setSpeculativeExecution(false);
conf.setReducerClass(Reduce.class);
conf.setInputPath(new Path(inDir));
conf.setOutputPath(new Path(outfile));
JobClient jc = new JobClient(conf);
RunningJob rj = jc.submitJob(conf);
```

APPENDIX C - Performance Results

Figure C - 1 Performance improvement in Grid with Logical mapping

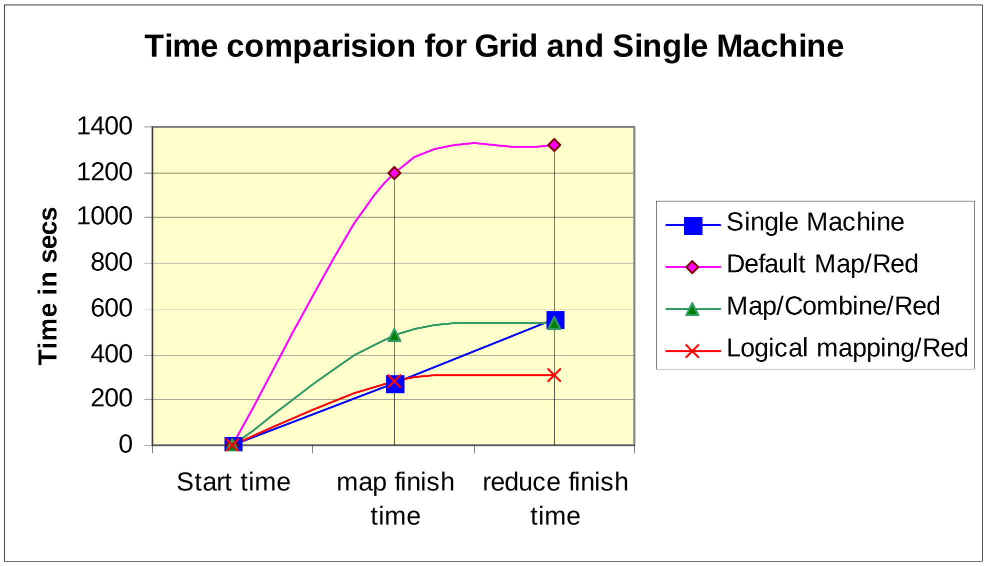
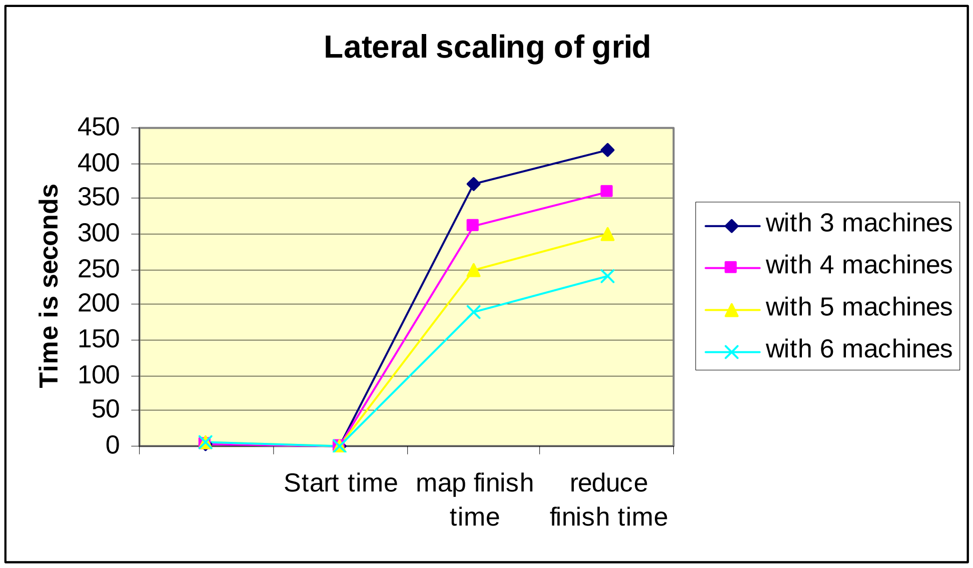


Figure C - 2 Lateral scaling of grid with more machines


Lateral scaling of grid
Time is seconds
Start time
map finish time
reduce finish time
with 3 machines
with 4 machines
with 5 machines
with 6 machines