*­TeraSort Implementation using SAGA MapReduce*

Submitted by  
*Sharath Maddineni*

Project Report for  
MS in *Engineering Science*   
(concentration in *Information Technology Engineering*)  
Louisiana State University  
*Fall 2010*

# Abstract

In recent years, there has been a large increase in the size of computational data distributed across different locations. MapReduce is a software framework for data intensive applications, which provides concurrent and distributed platform for handling computation and storage. The limitation with the available frameworks is that their implementation is tied to specific infrastructure and resources. Also, a number of applications like sorting, searching and other genomic applications were developed in those frameworks. SAGA (Simple API for Grid Applications) MapReduce framework was developed to support heterogeneous distributed environments. The objective of this project is to implement sorting using SAGA MapReduce framework. However, extracting performance is difficult using the SAGA MapReduce because of the overhead in interoperability provided by SAGA. Hence, project also involves enhancing the framework suitable to implement TeraSort effectively. Further, the results have been shown to implement the framework on clusters and across different machines.

Contents

Abstract ii

Background 1

Problem Statement and Objectives 2

Literature Review 2

Previous Implementations 2

SAGA MapReduce 2

Execution Overview of SAGA MapReduce 4

Methodology 4

Input data 6

Sort implementation 6

SAGA MapReduce on different Machines 9

SAGA MapReduce on clusters 9

Results and Analysis 10

Conclusion 12

Future Work 12

References 13

Appendix 14

# Background

MapReduce is a programming framework that supports applications, which involve parallel and distributed computations on very large data sets. It is extensively used for distributed applications such as genome searching, sorting, page ranking etc.

MapReduce computation is based on key/value pairs. From a set of input key/value pairs a set of output key/value pairs are produced. The programs in the MapReduce are written in functional style. These functions are automatically parallelized and executed on large clusters [3]. This framework typically consists of three functions: Mapper, Reducer, and Partitioner.

The Mapper takes an input pair and produces a set of intermediate key/value pairs. The framework collects all the intermediate values for a key and hands them to the Reducer. The Mapper performs most of the filtering operations like discarding the data, which not required and just picking the data, which is required in the next stage. For example, in a searching operation it just searches for the string in the input line and just emits out the result.

The Reducer gets an intermediate key and intermediate values list, and merges these to form a usually smaller set of values. Typically only a zero or one value is produced for each intermediate key, as the Reducer usually performs an aggregation of data.

The Partitioner is used by the Mapper to determine the partition number for the extracted key value pair. These partitions are assigned to the Reducer to aggregate.

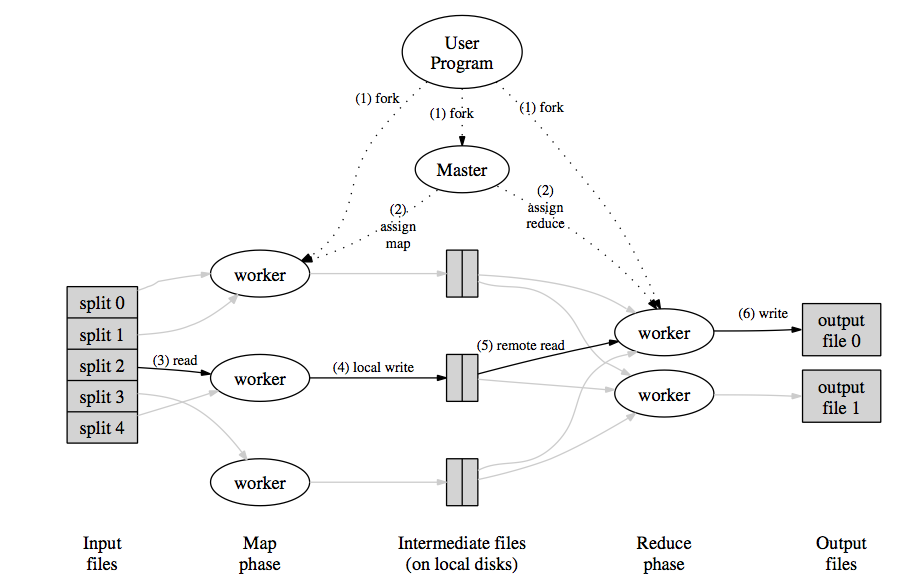


Figure 1: This figure shows the typical MapReduce process. [3]

**SAGA**

SAGA stands for Simple API (Application Program Interface) for Grid Applications. Running distributed applications involving large volumes of datasets and distributed worker machines is a difficult task for computational scientists. SAGA provides a simple Linux based interface for the users with basic programming background. SAGA has many advantages with different packages like a file package, job package, stream package and some other important packages required for grid applications. The SAGA File package handles the file transfers and browsing remote directories. The SAGA Job package handles the submitting, running jobs on remote machine and supercomputers. The SAGA Stream package takes care of the authentications required for performing any applications [2]. SAGA helps the computational scientists to perform large calculations, run distributed applications hiding the inner complexities involved.

# Problem Statement and Objectives

In recent years, there has been a large increase in the size of the data used in many organizations like hospitals where data is distributed over many locations. There are various approaches for computation of large dataset where the dataset resides on a single commodity cluster. For instance, Yahoo uses the MapReduce framework to process the large datasets by running the MapReduce based on Hadoop file system. However, the framework was designed to function with a specific infrastructure. Often, in the practical world the data will be distributed across different machines, which might be spread over different locations. Thus, the need to perform computation in heterogeneous distributed environment arises. MapReduce in SAGA enables applications to utilize the heterogeneous environment. There are several real world applications like sorting, searching, Blast and other Genomic applications which are developed using this framework. This work mainly focuses on sorting of the large datasets using the SAGA MapReduce on distributed infrastructures for open scientific research. For sorting, typical MapReduce sort algorithm is followed except that a set of keys are generated prior to start of MapReduce Framework. This work also focuses on starting the workers on different nodes of the cluster and enhancing the framework to support the sorting. Work was also done to implement the sorting using two remote machines configuring the required environment in order to use both workers on both machines.

# Literature Review

## Previous Implementations

MapReduce framework was implemented in several platforms in recent years. For example, Google has developed its own MapReduce framework in Google file system [3] while Yahoo has implemented MapReduce sorting in Hadoop [4]. However, each of them was tied their respective file systems and the implementations were limited to single cluster which could be as large as possible within the file system. [1]

## SAGA MapReduce

The SAGA MapReduce framework concentrates on processing large-scale dataset tasks implementable in heterogeneous environments supported by SAGA, so the application is not tied to any infrastructure [1]. Therefore, SAGA-MapReduce can work with different job scheduling systems like Globus, Condor and distributed file systems like HDFS, KFS and can be utilized in combination for performing MapReduce tasks [6].

**Mappers:** In order to use the Mapper, the declared Mapper class must be registered in the framework and declared in the job description of MapReduce. A user-specified mapper function is defined to emit intermediate key value pairs. To use this Map Function it has to be declared as a Mapper in the Job Description for MapReduce and must be registered with the framework.

**Reducers:** Similar to the Mapper, the declared Reducer class must be registered in the framework and declared in the job description of MapReduce. The intermediate key and value pairs generated in the Map Phase serve as the input Key In and Value In for the Reduce Phase, so these must match with the type of the intermediate keys and values of the respective Mapper class used for generating the input for this reducer. Key and Value pairs generated by Reducer after this phase denote the output type of the final keys and values.

**Partitioners:** The declared Partitioner class must also be registered in the framework and declared in the job description of MapReduce. When processing the input data by Mapper, the generated intermediate keys will get partitioned across reduce tasks using function. By default, this framework applies a partitioning function that returns the hash value of the input key modulo to the number of partitions. However, in some cases a different partitioning scheme might be required.

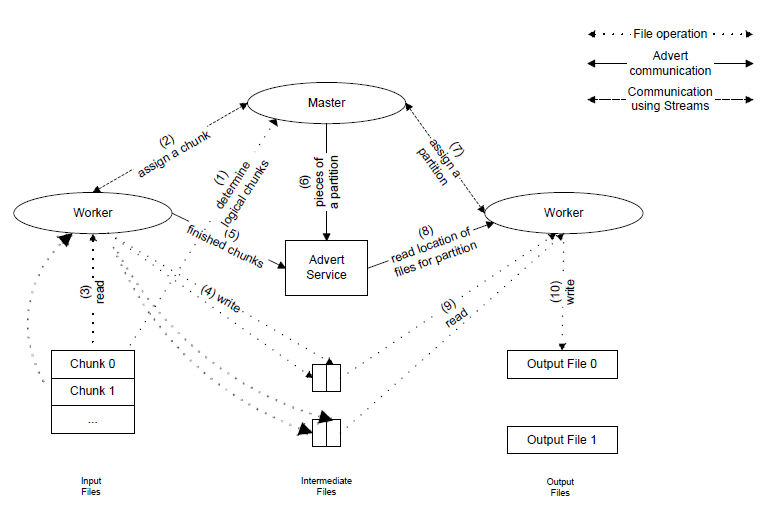


Figure 2: Figure shows the execution overview of SAGA MapReduce. [9]

## Execution Overview of SAGA MapReduce

While executing a MapReduce application, the following are the steps as provided in the User manual [6]:

1. Master is started with an executable file linked to SAGA-MapReduce libraries and the location for configuration file should be provided, which creates a session in advert service.
2. The Master takes the Input format specified in the Job Description to read and chunk the input data.
3. After preparing chunks, the Master spawns workers on the host machines, which are specified in the configuration file using the SAGA Job API.
4. Worker puts its status information into an advert directory and continues to communicate with Master using the advert service.
5. Workers will process the chunks assigned by the Master (in the Map Phase) using Map function defined by the user and partitions the data according the partition function.
6. When mapping all the chunks is finished Master moves to the Reduce Phase. In this phase, the Master assigns the partition number to be reduced to the idle workers.
7. Reducer aggregates all the given partitions and gives the sorted output partitions.

# Methodology

**Workers**

Currently SAGA MapReduce supports two kinds of workers

1. Launching the local workers using SAGA fork which just spawns the workers
2. Launching the remote and local workers through SAGA SSH adaptor

Depending on the resources required by the application, user could provide as many workers as he wants using an XML configuration file.

**Input Type**

SAGA MapReduce supports only file based input formats and from these file formats user can use either Text file formats for input or Sequence File format for input and output.

**Data Location**

Typically it reads a file as input if specified exact file path otherwise it takes all files as input which are in the specified folder.

Sorting large datasets using MapReduce is typically performed in the following steps [3][4]:

1. A Partition split list is generated before the Master is launched.
2. Master starts with generating the chunks from the input data.
3. After the chunks are generated, it starts the workers.
4. Then Master moves to Map Phase. In this Phase the master assigns the chunks to the workers.
5. If the number of chunks are more than the available workers, master puts chunks in a queue format and assigns those chunks whenever it sees a worker is free.
6. In the Map-Phase the worker behaves as a Mapper. Mapper uses the Map function specified by the user and processes the results while emitting the Key, Value pairs.
7. Typically each Mapper extracts the key value pairs and moves them into a particular partition using the partition function. In this project, the Mapper uses “Terapartitioner” which will determine the partition number based on the key value using the Partition list, which was generated previously. So, each Mapper creates multiple partitions where each partition has only particular range of keys.
8. After processing all chunks the Master moves into Reduce Phase. In this phase, Master assigns a partition number to the Worker to act as a Reducer. The Reducer combines all the partitions with that assigned partition number and stores the result in a particular output partition.
9. Thus we have the total sorted output divided in to partitions.

Using the above steps, a TeraSort is implemented by using the SAGA-MapReduce by defining a Mapper function, a Reducer function, and a Partitioning function and File output format. The partitioning list is extracted before actually starting the application. Here the TeraSort uses this list of partition splits, which are picked from the input chunks in the Partitioning function.

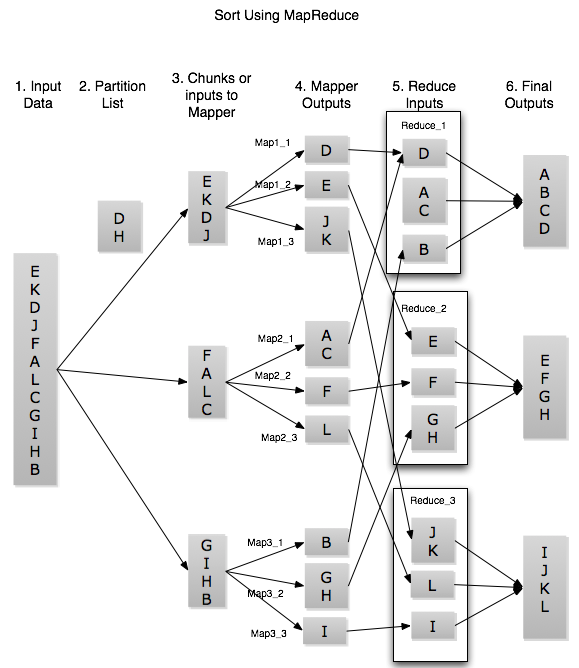


Figure 3: The steps in implementation of Sort using MapReduce. Assuming input keys from A to L.

## Input data

To generate input data for this project, a *gensort* program provided by the Sortbenchmark[5] is used. This program generates the data in the following format.

* Each record or each line is of 100 bytes of data.
* It is composed of 10 byte Key, 32 byte Row ID, and some random values.

Generating Partitioning list

For multiple workers and multiple chunks which have multiple map phases and multiple reduce phases.

* A class for building partition lists and reading the list.
* Use a new partitioner function for multiple workers to move the intermediate keys into respective partitions.

## Sort implementation

**Generate Partitioning list**

A typical partition split generator class (TeraKeyPartitionGenerator) contains these functions:

* SampleInput
* GetPartitions
* WritePartitions
* ReadPartitions

SampleInput: It reads some input chunks to generate the required number of samples from the input keys. For this function, job description should be passed, which contains the input data path and number of sample to be collected. It returns the list of collected keys.

Pseudo code:

function Sampleinput (job, sample\_size)

num\_samples=10

records\_per\_chunk := sample\_size / num\_samples;

chunk\_per\_sample := chunks.size() / num\_samples;

for i from 0 to num\_samples

chunk = chunk\_per\_sample \* i) // pick a random chunk, from chunk id’s in job

while read records in chunk

pick the 10 byte key from the random record

add these keys into an array;

If required keys are found from a sample break while;

end for

Return the collect keys in the array

GetPartitions: This function Sorts the sampled keys and creates num\_partitions-1 split points. Each key represents the starting key for that given partition.

Pseudo code:

function GetPartitions(num\_partitions)

load the keys array

sort it

samples\_per\_partition = num\_keys / num\_partitions

for i from 0 to num\_partitions

result[i] = keys(samples\_per\_partition \* i)

retrun result;

WritePartitions: This function writes the required number of partitions from the sampled keys to the given path in an output file format (partition list).

Pseudo code:

function WritePartitions(num\_partitons, path )

Initialize RecordWriter writer(path)

Initialize partitions

partitions := GetPartitions(num\_partitions)

for i from 0 to partitonssize

writer.Write(partitions[i])

ReadPartitions: This function reads partition splits from the given sequence file (partition list). This is used by the partitioning function to read the values from the generated list file.

Pseudo code:

function ReadPartitions(path)

RecordReader reader(path)

while (reader.NextRecord())

partitions.push\_back(reader.current\_key());

**Mapper Function**

The Map function gets chunk ID as the input key and one of the line which is read from the corresponding chunk as the value. By using the value each record is broken into 10-byte key to emit in the map phase and value as the entire record value. A mechanism should be used to skip the bad records.

Pseudo code:

class TeraSortMap derive from Mapper<>

public:

function Map( key, value, context)

if length(value) >99 // to skip the bad records

context Emit(string(value, 0,10), string(value, 11, 100)

In the Map phase, depending on the number of reducers specified in the job description the Mapper emits the keys and values in to the intermediate partitions. Again depending on the partition function the keys goes into the reduce phase. Thus, each worker in the map phase creates the reducer partitions which serve as the input in the reduce phase. Further, the partitioning list, which was generated, is sorted using merge sort by the SAGA-MapReduce framework. It is registered to the framework as REGISTER\_MAPPER\_CLASS (TeraSortMap, 1);

**Partitioner Function**

This function is initialized to read the keys from the partitioning list and Get partition should return partition number based on the input key. In this framework, the input keys to the Partition function are serialized. So, the input keys need to be deserialized and then compared to find the correct partition. It is registered to the framework as REGISTER\_PARTITIONER (TeraPart, 4);

Pseudo code:

Initialize() // initialize this instead of reading the file everytime.

partitions := TeraKeyPartitionGenerator.ReadPartitions(partition\_list)

function GetPartition(key, num\_partitions) //Extract the strings from serialized data

partition\_list := "file://path/to/\_partitions.lst"

for i form 0 to partitions.size()

if ascii\_value(deserialized\_key[0]) < ascii [0]

return i

else

return num\_partitions-1

fi

end for

The existing SAGA MapReduce framework did not support the initializing the partition function. In this project to achieve good results, the framework has been modified accordingly.

**Reducer Function**

Reducer Function just acts as an identity function. It takes the input key-value pairs from multiple intermediate files and emits them. Thus in the reduce phase each reducer collects the keys and values from the various intermediate partitions. The extracted key-value pairs are thus sorted.

Pseudo code:

function Reduce (int key, Iterator values, Context context)

Initialize val

Clear val // for each call to reduce because there may be same keys with different values.

while values has a next

val = val + values.Current()

Emit(key, val )

Finally, after the reduce phase, the output files are created depending on the number of reducers specified in the job description. Consequently, each partition has the keys, which are sorted where the ASCII values are less than its next partition keys ASCII value. It is registered in to the framework as REGISTER\_REDUCER\_CLASS (TeraSortReduce, 1);

**Output Format**

An output format has been declared for this TeraSort implementation. First TeraOutputRecord writer is declared using SAGA file package. The writer extracts strings from serialized data. This function should also be registered to the framework.

Pseudo code:

function Write(string key, string value)

writer\_->Write((key+1), key.size()-1)

writer\_->Write((value.c\_str()+1), value.size()-1)

writer\_->Write(new line)

**Job Description**

The job description is provided as following in the Main function of terasort.cpp. Typically it should be specified with the classes which are used to do the mapping, reducing, partitioning etc. Before initializing the framework the partitioning list should be generated as follows

Pseudo code:

JobDescription job;

job.set\_input\_format("Text"); // Specify input.

FileInputFormat::AddInputPath(job ,"file://path/to/input/");

job.set\_mapper\_class("TeraSortMap");

job.set\_reducer\_class("TeraSortReduce");

job.set\_partitioner\_class("TeraPart");

job.set\_output\_format("TeraOutput");

job.set\_num\_reduce\_tasks(5);

TeraKeyPartitionGenerator partitioner;

partitioner.SampleInput(&job, 200);

partitioner.WritePartitions(5, FileOutputFormat GetUrl(job, "\_partitions.lst")

job\_runner.Initialize(

mapreduce::g\_command\_line\_parameters["config"].as<std::string>());

job\_runner.Run(&result);

## SAGA MapReduce on different Machines

To implement the MapReduce sort on different machines, the input and output locations, which are one machine, should be mounted on the other machines as well. So, this implementation heavily depends on SSHFS fusermount to mount the required locations. A password less login from master machine to worker machine is also required by SAGA to start the workers on different machines. Configuring framework to appropriate location is also required for successful execution of the MapReduce.

## SAGA MapReduce on clusters

For large datasets a number of workers are required. Right now SAGA MapReduce does not support the submitting jobs to the PBS scheduler and generating the list of workers automatically. So a python script, which would generate the list of workers in an xml file, is written in order to run workers.

And then this script is used for submission to scheduler.

#!/bin/bash

#PBS -N maplog

#PBS -l walltime=08:00:00,nodes=16:ppn=8

#PBS -q workq

echo Job $PBS\_JOBNAME is executing in $PBS\_O\_WORKDIR

cat $PBS\_NODEFILE > path/to/nodefile.txt

cd /path/to/terasort/

python xml.py

./terasort /path/to/mrinput/ -c pbs.xml

Once the XML file with the list of required workers is generated, the Master will start the workers using SSH job submission of SAGA.

To run this MapReduce in a distributed environment all the workers and Master must have access to input and output locations. On supercomputers all the nodes have the Luster file system, where every node can access the input and output locations.

Finally, the output files from this sort of applications are validated using the *valsort* provided by the Sortbenchmark itself.

There were problems in launching the MapReduce workers directly on remote machines because of the problems in loading the remote environments. In order to overcome this problem, the required environmental variables have been presented in the job description of SAGA job when launching the MapReduce workers.

Some workers fail in the Map phase because of the bad records or lines in the given input file. So, a mechanism has been devised inside the Map function, which would skip performing the Map function on bad records.

Another factor, which affects the performance, is that SAGA MapReduce is hugely dependent on external advert service. The number of connections that are allowed on the advert server limits the number of Workers supported by SAGA MapReduce. Also, the performance also depends on the connection speed from the location where SAGA MapReduce is performed to the advert server location. As the input size increases the number of intermediate files also increases which in turn increases the number of entries in the advert server. For the reduce phase the Workers need to read their input files from the advert server. Clearly, as the input size increases the latency increases because of the large number of write operations by Master. Efforts have been made in order to reduce this communication with the advert server in the reduce phase.

On the FutureGrid machines there is a limit of disk size available. Whereas, on LONI machines like Queen Bee disk size is not a problem but is limited by the number of files (10,000)[7] allowed per directory required by the SAGA MapReduce for larger input size; this causes performance degradation of Workers.

# Results and Analysis

The three main factors that affect the performance of SAGA MapReduce are

* Chunk size
* Number of workers and
* Size of input data.

Therefore, tests were repeated three times varying one of the factors while keeping the others constant.

The results when the number of workers and input data size were varied were not surprising, i.e., the increase in the number of workers improved the performance proportionally and when the input size was increased the time taken also increased proportionally. However, there was an anomaly when the chunk size was varied.

This experiment was conducted on FutureGrid cluster Sierra. The input data size was 4Gb. Number workers were 8, Number of reduces were 8.

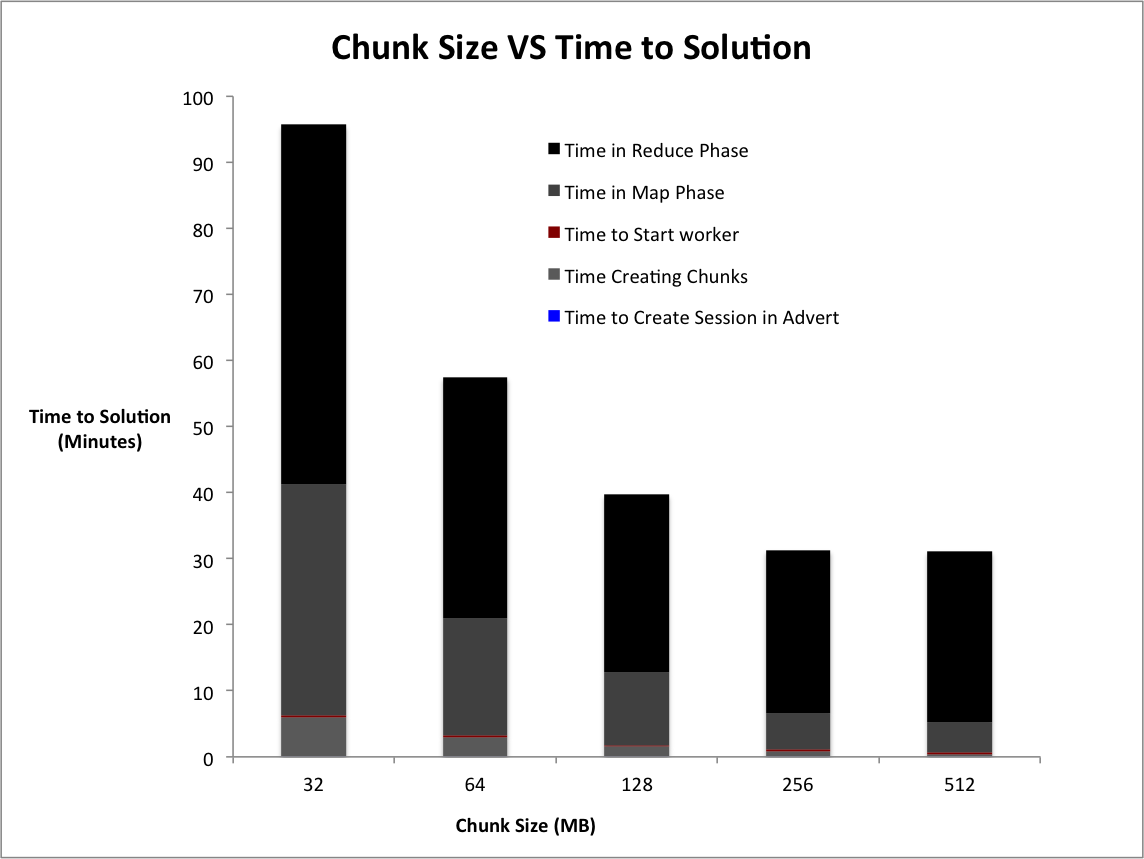


Figure 4: Time to solution vs Chunk size

Clearly from the above graph, increase in the chunk size from 16MB to 256MB showed an increase in the performance whereas from 256MB onwards there was no significant improvement in the performance. Also, from the graph above it can be noticed that with the increase in chunk size

* the time to create chunks decreased in the Chunk phase; Master creates fewer entries for chunks in the advert server.
* Time spent in the Map phase is most affected; number of Workers in Map phase decreases and time taken to process a chunk is less than time taken to process 2 chunks half its size until 256MB after which time spent in the Map phase doesn’t change much due to other factors such as CPU speed, RAM, number of Workers etc. have a bigger effect.
* Time spent in Reduce phase also decreases; less number of intermediate files are generated.

32 GB is the maximum input size for successfully running this sorting application. The factors are 256mb chunk size, 32 partitions and 16 workers. The total time to solution was 172 minutes.

This sort application was also tested to run 1 GB or input data size across two machines (cyd01.cct.lsu.edu and cyder.cct.lsu.edu). The master and the data are on the same machine and there are 8 remote workers. Chunk size was 128 MB and the time to solution was 15.23 minutes.

# Conclusion

In this project, data was generated using *gensort* and validated using *valsort* programs provided by Sortbenchmark. Two approaches for sorting using SAGA MapReduce were implemented on Distributed machines and clusters.

The main factor that affected the performance in the distributed implementation was mounting of input and output locations, which heavily relied on the network speed. In the cluster implementation, the file system mounted to nodes using luster filesystem was fast but limited by the number of files allowed in a single directory.

The applications where the data is on a single cluster should be used because it is faster and involves very less network overhead. In a case where data and workers are distributed it is good to use distributed machines, which would lower the data transfer time.

# Future Work

The future work should mainly focus on improving the performance of SAGA MapReduce framework. This involves reducing Master-Worker communication with advert service, which slows the execution. Also, the framework’s capability to handle large number of intermediate files should be improved. For heterogeneous environments an intelligent framework can be developed to distribute the workers intelligently based on the computation power of the machines used.

# References

1. Saurabh Sehgal1, Miklos Erdelyi, Andre Merzky, Shantenu Jha.: 2010, Understanding Application-Level Interoperability, Scaling-Out MapReduce Over High-Performance Grids and Clouds, Retrieved from

http://saga.cct.lsu.edu/publications/papers/journals/understanding-application-level-interoperability-scaling-out-mapreduce-over-high-performance-grids-and-clouds

1. Merzky, A., Stamou, K., Jha, S.: 2009, Application Level Interoperability between Clouds and Grids, Retrieved from, http://saga.cct.lsu.edu/publications/papers/confpapers/Application-Level-Interoperability-between-Clouds-and-Grids/saga\_cloud\_interop.pdf/view
2. Dean, J. and Ghemawat, S.: 2008, “Map-Reduce: simplified data processing on large clusters, Commun. ACM 51(1), 107- 113”. Retrieved from

http://static.googleusercontent.com/external\_content/untrusted\_dlcp/labs.google.com/en/us/papers/mapreduce-osdi04.pdf

1. Owen O’ Malley, May 2008,TeraByte Sort on Apache Hadoop ,Retrived from,

http://www.hpl.hp.com/hosted/sortbenchmark/YahooHadoop.pdf

1. May 2010, Sort Benchmark , Retrieved from, http://sortbenchmark.org/
2. Miklos Erdelyi, May 2010, User Manual SAGA MapReduce, Retrieved from,

https://svn.cct.lsu.edu/repos/saga-projects/applications/MapReduce/branches/MapReduce.2009/docs/user\_manual/tex/user\_manual.tex

1. November 2010, Queen bee storage, https://docs.loni.org/wiki/File\_Storage\_on\_Linux\_x86\_Clusters

# Appendix

Specifications of machines used as of November

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | Type | Processor | Cores  per node | Ram  per node | Total Disk space Avaialable |
| India | FutureGrid  Cluster | Intel(R) Xeon(R) CPU ,2.50GHz | 8 | 24GB | Shared Disk space  (< 150GB free space) |
| Sierra | FutureGrid  Cluster | Intel(R) Xeon(R) CPU ,2.50GHz | 8 | 32GB | 50 GB |
| Queen Bee | LONI  Cluster | Intel(R) Xeon(R) CPU, 2.33GHz | 8 | 16GB | Shared disk space  ( < 20 TB free space) |
| Cyd01 | Single Server | Dual-Core AMD Opteron, 1GHz | 4 | 8GB | 450GB |
| Cyder | Single Server | Six-Core AMD Opteron(tm) 2.4 GHz | 12 | 64GB | 5.2 TB |

The tera\_utils.hpp is as follows

#ifndef TERA\_UTILS\_HPP\_

#define TERA\_UTILS\_HPP\_

#include "mapreduce/input\_output.hpp"

using mapreduce::TextInputFormat;

using mapreduce::SerializationHandler;

class TeraKeyPartitionGenerator {

public:

TeraKeyPartitionGenerator() {}

// Reads some input chunks to generate the required number of samples from the

// input keys.

void SampleInput(JobDescription\* job, long sample\_size) {

//find the number of chunks to it collect samples

boost::scoped\_ptr<RawInputFormat> input\_format(new TextInputFormat);

std::vector<InputChunk\*> chunks = input\_format->GetChunks(\*job);

int num\_samples = std::min(10, static\_cast<int>(chunks.size()));

long records\_per\_chunk = sample\_size / num\_samples; // collect only these number of keys per chunk

int chunk\_per\_sample = chunks.size() / num\_samples; // use only these number of chunks

long num\_records = 0;

std::string key\_buffer;

// check for max sample size

for (int i = 0; i < num\_samples; ++i) {

boost::scoped\_ptr<RawRecordReader> reader(input\_format->GetRecordReader(

chunks[chunk\_per\_sample \* i], job));//take a chunk to collect keys

while (reader->NextRecord()) {

// Extract key from line of text.

ZeroCopyInputStream\* value = reader->current\_value();

key\_buffer.clear(); //use a key\_buffer to store the entire record but clear it before using for next record

SerializationHandler<std::string>::Deserialize(value, &key\_buffer);

keys\_.push\_back(key\_buffer.substr(0, 10));

// if are at max records\_per\_chunk for this sample break and goto next chunk.

if (num\_records >= (i + 1)\*records\_per\_chunk) {

break;

}

++num\_records; //incement records

}

}

}

// Sorts the sampled keys and creates num\_partitions-1 split points

std::vector<std::string> GetPartitions(int num\_partitions) {

int num\_keys = static\_cast<int>(keys\_.size());

if (num\_keys < num\_partitions) {

throw new std::runtime\_error("Requested more partitions than keys available");

}

//sort the keys

std::sort(keys\_.begin(), keys\_.end());

// Get each nth sample to get partitioning keys.

float samples\_per\_partition = static\_cast<float>(num\_keys) /

static\_cast<float>(num\_partitions);

std::vector<std::string> result(num\_partitions - 1);

for (int i = 1; i < num\_partitions; ++i) {

result[i - 1] = keys\_[static\_cast<int>(round(samples\_per\_partition \* i))];

}

return result; //return the array

}

// Writes the required number of partitions from the sampled

void WritePartitions(int num\_partitions, saga::url path) {

//declare writer using input\_output.hpp

RecordWriter<std::string, std::string> writer(

new SequenceFileRecordWriter(path));

std::vector<std::string> partitions = GetPartitions(num\_partitions);

std::string null\_value;

// write these partitions into given path

for (int i = 0; i < static\_cast<int>(partitions.size()); ++i) {

writer.Write(partitions[i], null\_value);

}

writer.Close();

}

// Reads partitions from the given SequenceFile.

static std::vector<std::string> ReadPartitions(saga::url\* path) {

// declaration of reader is form the input\_output.hpp

RecordReader<std::string, std::string> reader;

boost::scoped\_ptr<RawRecordReader> record\_reader(

new SequenceFileRecordReader(\*path));

reader.Initialize(record\_reader.get());

std::vector<std::string> partitions;

while (reader.NextRecord()) {

partitions.push\_back(reader.current\_key());

}

reader.Close();

//return the patitiions

return partitions;

}

private:

std::vector<std::string> keys\_;

};

#endif /\*TERA\_UTILS\_HPP\_\*/

Terasort.cpp file is as follows

#include <sstream>

#include <string>

#include "mapreduce.hpp"

#include "master/DistributedJobRunner.hpp"

#include "tera\_utils.hpp"

using std::stringstream;

using std::string;

using mapreduce::Mapper;

using mapreduce::JobDescription;

class TeraOutputRecordWriter : public RawRecordWriter {

public:

TeraOutputRecordWriter(saga::url& path) {

writer\_.reset(new SagaFileOutputStream(saga::filesystem::file(

path, saga::filesystem::Write)));

}

~TeraOutputRecordWriter() {}

// RawRecordWriter implementation. Extracts original strings from

// serialized data directly (assumes their length is stored on 1 byte).

void Write(const std::string& key, const std::string& value) {

writer\_->Write((key.c\_str()+1), key.size()-1);

writer\_->Write((value.c\_str()+1), value.size()-1);

writer\_->Write(static\_cast<const void\*>("\r\n"), 2);

}

void Close() {

writer\_.reset();

}

private:

boost::scoped\_ptr<SagaFileOutputStream> writer\_;

};

// Output format for writing key/value pairs for TeraSort.

class TeraOutputFormat : public FileOutputFormat {

public:

RawRecordWriter\* GetRecordWriter(TaskDescription\* task) {

// Default work path for task.

saga::url default\_url = FileOutputFormat::GetUrl(\*task,

FileOutputFormat::GetUniqueWorkFile(task));

return new TeraOutputRecordWriter(default\_url);

}

};

class TeraPart : public Partitioner {

public:

#define PARTITION\_LIST "\_partitions.lst"

void InitPart(JobDescription\* job) {

saga::url partition\_list = “/path/to/partition/list/”

std::vector<std::string> partitions = TeraKeyPartitionGenerator::ReadPartitions(

&partition\_list);

}

int GetPartition(const std::string& key, int num\_partitions) {

std::string deserialized\_key;

ArrayInputStream input\_stream(mapreduce::string\_as\_array(&(const\_cast<std::string&>(key))), key.size());

mapreduce::SerializationHandler<std::string>::Deserialize(&input\_stream, &deserialized\_key);

saga::url partition\_list = "file://localhost/work/smaddi2/workerop/\_partitions.lst"

std::vector<std::string> partitions = TeraKeyPartitionGenerator::ReadPartitions(&partition\_list);

for (int i=0; i< partitions.size() ; i++)

{

std::string ps = partitions[i];

if ( int(deserialized\_key[0]) < int(ps[0]))

return i;

}

return num\_partitions-1;

}

};

REGISTER\_PARTITIONER(TeraPart, 4);

REGISTER\_OUTPUTFORMAT(TeraOutput, TeraOutputFormat);

// Mapper for extracting key/value from TeraSort input line.

#define TERASORT\_KEY\_LENGTH 10

class TeraSortMap : public Mapper<int, string, string, string> {

public:

void Map(const int& key, const string& value, Context\* context) {

// Get key/value.

std::string keys;

keys= key;

context->Emit(string(value, 0,10), value );

}

};

REGISTER\_MAPPER\_CLASS(TeraSortMap, 1);

class TeraSortReduce : public Reducer<string, string, string, string> {

public:

void Reduce(const string& key, Iterator<string> &values, Context\* context) {

std::string val ;

val ="";

val.clear();

while (values.Next()) {

val += values.Current();

}

context->Emit(key, " " + val );

}

};

REGISTER\_REDUCER\_CLASS(TeraSortReduce, 1);

int main(int argc, char\*\* argv) {

std::string input\_loc;

input\_loc = argv[1];

// std::cerr << input\_loc;

try {

// Will not continue execution if we are supposed to be a worker.

// Otherwise jobs will be submitted.

mapreduce::MapRunner\* runner = mapreduce::MapRunnerFactory::get\_by\_key("TeraSortMap");

assert(runner);

mapreduce::ReduceRunner\* runner2 = mapreduce::ReduceRunnerFactory::get\_by\_key("TeraSortReduce");

assert(runner2);

mapreduce::Partitioner\* runner3 = mapreduce::PartitionerFactory::get\_by\_key("TeraPart");

assert(runner3);

mapreduce::InitFramework(argc, argv);

MapReduceResult result;

JobDescription job;

// Specify input.

job.set\_input\_format("Text");

FileInputFormat::AddInputPath(job,

input\_loc);

job.set\_mapper\_class("TeraSortMap");

job.set\_reducer\_class("TeraSortReduce");

job.set\_partitioner\_class("TeraPart");

job.set\_output\_format("TeraOutput");

job.set\_num\_reduce\_tasks(20);

job.set\_chunk\_size(64);

TeraKeyPartitionGenerator partitioner;

partitioner.SampleInput(&job, 2000);

partitioner.WritePartitions(20, FileOutputFormat::GetUrl(job, "\_partitions.lst") );

std::cerr << "Done part";

mapreduce::master::DistributedJobRunner job\_runner(job);

job\_runner.Initialize(mapreduce::g\_command\_line\_parameters["config"].as<std::string>());

job\_runner.Run(&result);

} catch (const std::exception& e) {

std::cerr << "std exception caught: " << e.what() << std::endl;

} catch (const saga::exception& se) {

std::cerr << "saga exception caught: " << se.what() << std::endl;

}

}

**XML configuration file:**

<?xml version="1.0" encoding="ISO-8859-1"?>

<MRDL version="1.0"

xmlns="http://cct.lsu.edu/MRL-1-0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema\_instance" >

<!-- Description of the session - a file can contain multiple sessions -->

<MapReduceSession name="Terasort"

version="0.1"

user="maddy"

priority="1"

experimentID="AgentTest"

eventLevel="DEBUG">

<!-- The orchestrator host (AdvertDB) -->

<OrchestratorDB>

<Host>

advert://fortytwo.cct.lsu.edu/5432

</Host>

</OrchestratorDB>

<!-- List of hosts we want to run comp. agents on -->

<TargetHosts>

<Host arch="x86\_64" OS="redhat">

fork://localhost

</Host>

</TargetHosts>

<MasterAddress>tcp://localhost:8001</MasterAddress>

<!-- Application binaries for different platforms/architectures -->

<ApplicationBinaries>

<BinaryImage arch="x86\_64" OS="redhat" extraArgs="">

/Users/Maddy/Desktop/mapreduce/examples/terasort/terasort

</BinaryImage>

</ApplicationBinaries>

<!-- Output file system schema and base location -->

<OutputBase>file://localhost//work/smaddineni/workerop</OutputBase>

</MapReduceSession>

</MRDL>

Python Script in used for generating list of workers

Code

hnm= gethostname()

file = open nodelist file and load its contents

initialize sseq, list, r, lst

while sseq is not null :

sseq = read each line of file

kl = sseq

lst= clean kl

if sseq is not null

if (lstris equal to lst):

if (r < 3): //number of workers on a single node

list = list + '<Host arch="x86\_64" OS="redhat"> ssh://%s/ </Host>\n' %lst

r= r +1

lstr = lst

else if r=7

r=0

else:

r=r+1

else:

list = list + '<Host arch="x86\_64" OS="redhat"> ssh://%s/ </Host>\n' %lst

lstr = lst

r=r+1

close file

nfile = open("/path/to/pbs\_use.xml","rb")

sseq = "test"

newxml = ""

while sseq is not null :

sseq = nfile.readline()

if ("replaceme" in sseq):

newxml = newxml + list

elif ("replaceme2" in sseq):

newxml = newxml + " <MasterAddress>tcp://” +hnm+ ”:80011/</MasterAddress>"

else:

newxml = newxml + sseq

nfile.close()

FILE = open("/path/to/terasort/pbs.xml”)

FILE.writelines(newxml)