COMPARISON BETWEEN JAVA AND MAPREDUCE PARALLEL PROGRAMMING TECHNIQUES

FINAL YEAR BACHELOR OF ENGINEERING IN INFORMATION TECHNOLOGY PROJECT

SUBMITTED BY

ABHIRUP BHABANI
001011001028

AND

SANTANU DUTTA
001011001036

UNDER THE SUPERVISION OF
Mr. UTPAL KUMAR RAY

DEPARTMENT OF INFORMATION TECHNOLOGY
JADAVPUR UNIVERSITY

2014

ACKNOWLEDGEMENT

We would like to thank our project guide Mr. Utpal Kumar Ray for his guidance and support and the faculty members of the department for their cooperation. We extend our sincere thanks to our classmates who have helped us in our hours of need.

ABSTRACT

The past few years have marked the start of a historic transition from sequential to parallel computation. The necessity to write parallel programs is increasing. The problem size is increasing, and to tackle that, we need technologies which will provide faster and accurate computation. We are constantly striving for massive scalability across distributed systems consisting of hundreds and thousands of nodes in a cluster.

The term MapReduce actually refers to two separate and distinct tasks that Hadoop programs perform. The first is the map job, which takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). The reduce job takes the output from a map as input and combines those data tuples into a smaller set of tuples.

This project report presents the concept of Map Reduce, its various advantages and limitations and the concept of writing code using Map Reduce under Hadoop File System.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

1.1. NEED FOR PARALLEL COMPUTING

For the past decade, distributed computing has been one of the biggest buzz phrases in the computer industry. At this point in the information age, we know how to build networks; we use thousands of engineering workstations and personal computers to do our work, instead of huge behemoths in glass-walled rooms. Surely we ought to be able to use our networks of smaller computers to work together on larger tasks. And we do an act as simple as reading a web page requires the cooperation of two computers (a client and a server) plus other computers that make sure the data gets from one location to the other. However, simple browsing (i.e., a largely one-way data exchange) isn't what we usually mean when we talk about distributed computing. We usually mean something where there's more interaction between the systems involved.

Parallelism is not automatically faster than performing operations serially, although it can be if you have enough data and processor cores. While aggregate operations enable us to more easily implement parallelism, it is still the responsibility of the programmer to determine if the application is suitable for parallelism.

- Computing things in parallel by breaking a problem into smaller pieces enables us to solve larger problems without resorting to larger computers. Instead, we can use smaller, cheaper, easier to find computers.

- Large data sets are typically difficult to relocate, or easier to control and administer located where they are, so users have to rely on remote data servers to provide needed information.

- Redundant processing agents on multiple networked computers can be used by systems that need fault tolerance. If a machine or agent process goes down, the job can still carry on.

1.2. MOTIVATION

Using Java

Many applications ranging from scientific computing to data mining to interactive virtual reality applications need powerful computer resources and have to deal with distributive system equipped with parallelism, heterogeneity and reconfigurability. The main target of these parallel and distributed applications is networks and clusters of workstations. Object-oriented languages are widely used in many areas of computing and provide a practical solution for embedding application domain programming models into frameworks easing the work of programmers. These languages offer a convenient way to separate design from implementation by efficiently encapsulating implementation designs into classes.

Using MapReduce

Over the past years, studies are going on to successfully implement hundreds of special-purpose online data computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues. As a reaction to this complexity, a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library, is successfully achieved in Hadoop.

CHAPTER 2
USING MAPREDUCE TECHNIQUE

2.1. INTRODUCTION

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The runtime system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required intermachine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system. MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

2.2. PROGRAMMING MODEL

The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: Map and Reduce. Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function. The Reduce function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

Conceptually the map andreduce functions supplied by the user have associatedtypes:

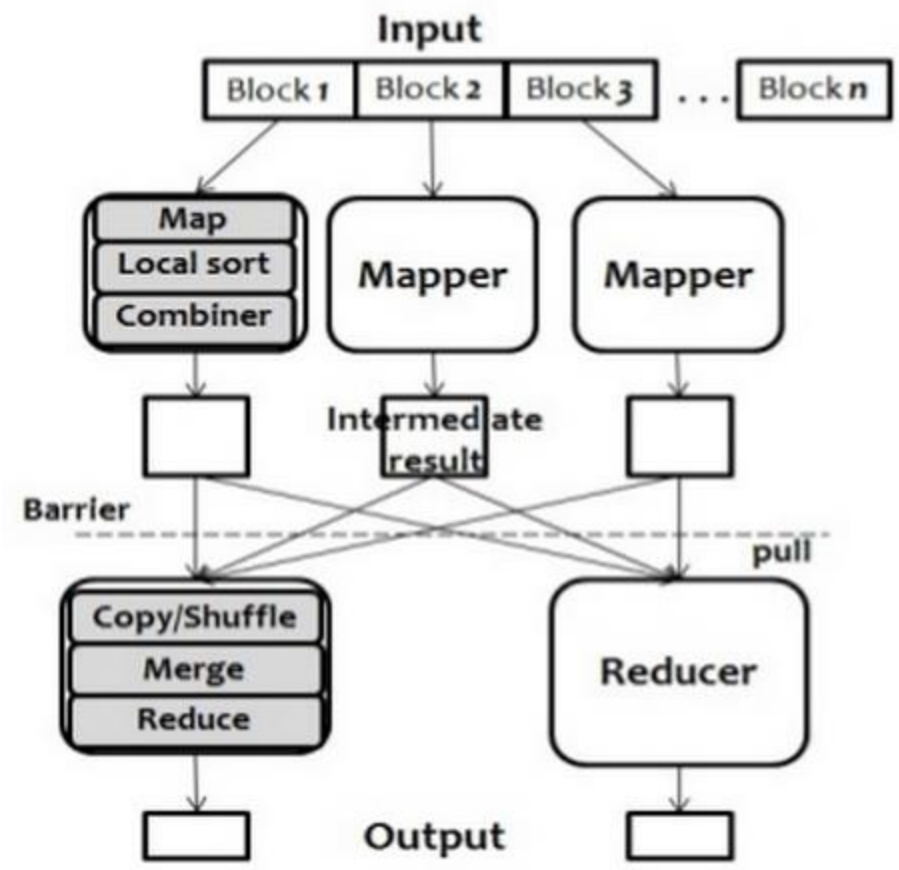
Map	(k1,v1)	= list(k2,v2)
Reduce	(k2,list(v2))	= list(v2)

i.e., the input keys and values are drawn from a differentdomain than the output keys and values. Furthermore,the intermediate keys and values are from the same domainas the output keys and values.

2.3. HADOOP ARCHITECTURE

MapReduce utilizes the Google File System(GFS) as an underlying storage layer to read input and store output. GFS is a chunk-based distributed file system that supports fault tolerance by data partitioning and replication. Apache Hadoop is an open source java implementation of MapReduce. Hadoopconsists of two layers: a data storage layer called Hadoop DFS(HDFS) and a data processing layer called Hadoop MapReduce Framework. HDFS is a block-structured file system managed by a single master node.

A single MapReduce(MR) job is performed in two phases: Map and Reduce stages. The master picks up idle workers and assigns a map or a reduce task according to the stage. Before starting the map task, an input file is loaded on the distributed file system, and partitioned into multiple data blocks of same size. Each block is then assigned to a mapper, a worker which is assigned the map task, and the mapper applies Map() to each record on the data block. The intermediate outputs produced by the mappers are then sorted locally for grouping key value pairs sharing the same key. After local sort, Combine() is optionally applied to perform preaggregation on the grouped key-value pairs so that communication cost taken to transfer all the intermediate outputs to the reducers is minimized. The mapped outputs are stored in the local disks of the mappers, partitioned into R, where R is the number of Reduce tasks i n the MR job.



When all Map tasks are completed, the MapReduce scheduler assigns Reduce tasks to workers. The intermediate results are shuffled and assigned to reducers via HTTPS protocol. Since all mapped outputs are already partitioned and stored in local disks, each reducer performs the shuffling by simply pulling its partition of the mapped outputs from the mappers. A reducer reads the intermediate results and merges them by intermediate keys, i.e. key2. Then each reducer applies Reduce() to the intermediate vakues for each key key2 it encounters. The output of the reducers are stored and triplicated in HDFS.

2.4. IMPLEMENTATION OF THE MODEL

Counting the number of occurrencesof each word in a large collection of documents

The map function emits each word plus an associatedcount of occurrences.The reduce function sums together all counts emittedfor a particular word.In addition, the user writes code to fill in a MapReduce specification object with the names of the input and output files, and optional tuning parameters. The user theninvokes the MapReduce function, passing it the specification object.

Distributed Grep

The map function emits a line if itmatches a supplied pattern. The reduce function is anidentity function that just copies the supplied intermediatedata to the output.

Count of URL Access Frequency

The map functionprocesses logs of web page requests and outputs<URL, 1>. The reduce function adds together all values for the same URL and emits a <URL, total count>pair.

Reverse Web Link Graph

The map function outputs<target, source> pairs for each link to a targetURL found in a page named source. The reducefunction concatenates the list of all source URLs associatedwith a given target URL and emits the <target,list(source)>

Inverted Index

The map function parses each document, and emits a sequence of <word, document ID>pairs. The reduce function accepts all pairs for a givenword, sorts the corresponding document IDs and emits a<word, list(document ID)> pair. The set of all outputpairs forms a simple inverted index. It is easy to augmentthis computation to keep track of word positions.

Distributed Sort

The map function extracts the key from each record, and emits a <key, record> pair. Thereduce function emits all pairs unchanged. This computationdepends on the partitioning facilities.

2.5. APPLICABILITY

The calculations that need to be run have to be composable. This means it should be possible to run the calculation on a subset of data, and merge partial results.

The dataset size is big enough (or the calculations are long enough) that the infrastructure overhead of splitting it up for independent computations and merging results will not hurt overall performance.

The calculation depends mostly on the dataset being processed; additional small data sets can be added using HBase, Distributed cache or some other techniques.

2.6. NON-SUITABILITY OF THIS TECHNIQUE

MapReduce is not applicable, in scenarios when the dataset has to be accessed randomly to perform the operation. MapReduce calculations need to be composable - it should be possible to run the calculation on a subset of data, and merge partial results.

MapReduce is not applicable for general recursive problems because computation of the current value requires the knowledge of the previous one, which means that you can't break it apart to sub computations that can be run independently..

If a data size is small enough to fit on a single machine memory, it is probably going to be faster to process it as a standalone application. Usage of MapReduce, in this case, makes implementation unnecessary complex and typically slower.

MapReduce is an inherently batch implementation and as such is not applicable for online computations

CHAPTER 3
USING JAVA TECHNIQUE

3.1. INTRODUCTION

Parallel computing involves dividing a problem into subproblems, solving those problems simultaneously (in parallel, with each subproblem running in a separate thread), and then combining the results of the solutions to the subproblems. Java SE provides the fork/join framework, which enables us to more easily implement parallel computing in your applications. However, with this framework, we must specify how the problems are subdivided (partitioned). With aggregate operations, the Java runtime performs this partitioning and combining of solutions.

One difficulty in implementing parallelism in applications that use collections is that collections are not thread-safe, which means that multiple threads cannot manipulate a collection without introducing thread interference or memory consistency errors. The Collections Framework provides synchronization wrappers, which add automatic synchronization to an arbitrary collection, making it thread-safe. However, synchronization introduces thread contention. It is preferable to avoid thread contention because it prevents threads from running in parallel. Aggregate operations and parallel streams enable us to implement parallelism with non-thread-safe collections provided that we do not modify the collection while you are operating on it.

3.2. COMPONENTS

A distributed application is built upon several layers. At the lowest level, a network connects a group of host computers together so that they can talk to each other. Network protocols like TCP/IP let the computers send data to each other over the network by providing the ability to package and address data for delivery to another machine. Higher-level services can be defined on top of the network protocol, such as directory services and security protocols. Finally, the distributed application itself runs on top of these layers, using the mid-level services and network protocols as well as the computer operating systems to perform coordinated tasks across the network.

At the application level, a distributed application can be broken down into the following parts:

Processes

A typical computer operating system on a computer host can run several processes at once. A process is created by describing a sequence of steps in a programming language, compiling the program into an executable form, and running the executable in the operating system. While it's running, a process has access to the resources of the computer (such as CPU time and I/O devices) through the operating system. A process can be completely devoted to a particular application, or several applications can use a single process to perform tasks.

Threads

Every process has at least one thread of control. Some operating systems support the creation of multiple threads of control within a

single process. Each thread in a process can run independently from the other threads, although there is usually some synchronization between them. One thread might monitor input from a socket connection, for example, while another might listen for user events (keystrokes, mouse movements, etc.) and provide feedback to the user through output devices (monitor, speakers, etc.). At some point, input from the input stream may require feedback from the user. At this point, the two threads will need to coordinate the transfer of input data to the user's attention.

Objects

Programs written in object oriented languages are made up of cooperating objects. One simple definition of an object is a group of related data, with methods available for querying or altering the data (getName(), setName()), or for taking some action based on the data (sendName(OutputStreamo)). A process can be made up of one or more objects, and these objects can be accessed by one or more threads within the process. And with the introduction of distributed object technology like RMI and CORBA, an object can also be logically spread across multiple processes, on multiple computers.

Agents

While a process, a thread, and an object are pretty well-defined entities, an agent is a higher-level system component, defined around a particular function, or utility, or role in the overall system. A remote

banking application, for example, might be broken down into a customer agent, a transaction agent and an information brokerage agent. Agents can be distributed across multiple processes, and can be made up of multiple objects and threads in these processes. Our customer agent might be made up of an object in a process running on a client desktop that's listening for data and updating the local display, along with an object in a process running on the bank server, issuing queries and sending the data back to the client. There are two objects running in distinct processes on separate machines, but together we can consider them to make up one customer agent, with client-side elements and server-side elements.

3.3. PROGRAMMING MODEL

The Client/Server Model (Socket Programming)

The client/server model is a form of distributed computing in which one program (the client) communicates with another program (the server) for the purpose of exchanging information. In this model, both the client and server usually speak the same language, a protocol that both the client and server understand, so they are able to communicate. While the client/server model can be implemented in various ways, it is typically done using low-level sockets. Using sockets to develop client/server systems means that we must design a protocol, which is a set of commands agreed upon by the client and server through which they will be able to communicate. The HTTP protocol that provides a method calledGET, which must be implemented by all web servers and used by web clients (browsers) in order to retrieve documents, is an example.

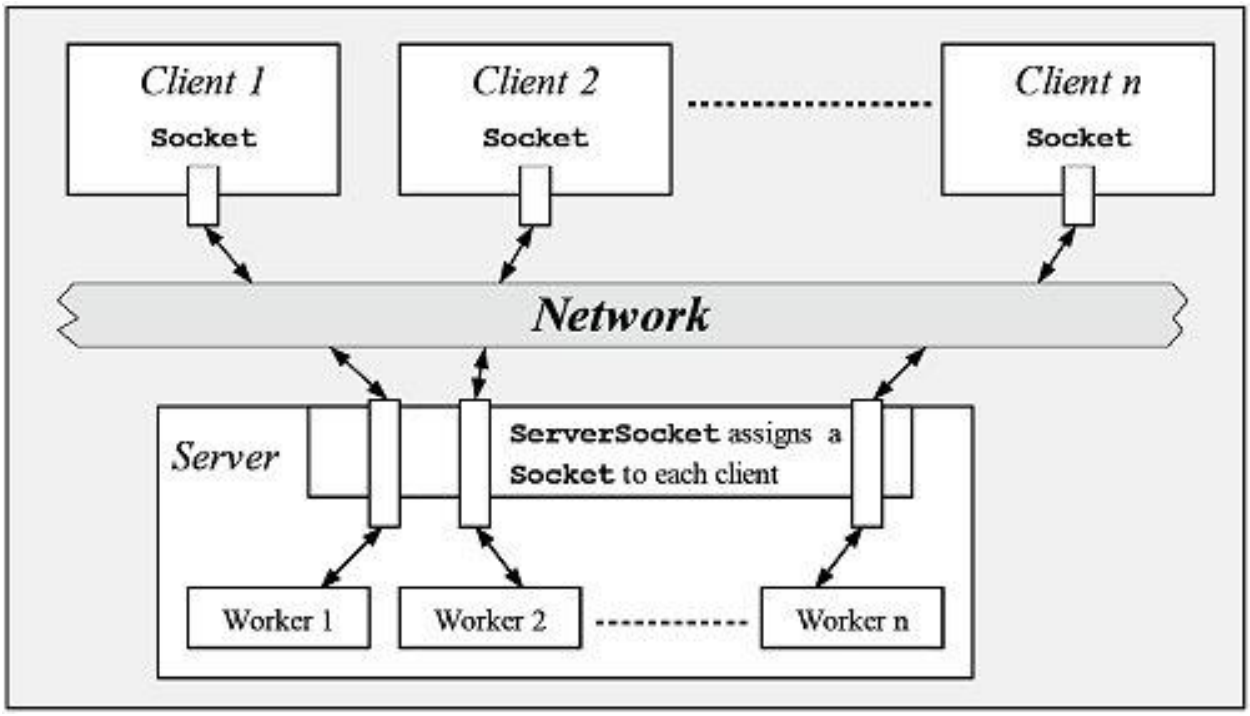


Figure showing the implementation of Socket Programming

The Distributed Objects Model

A distributed object-based system is a collection of objects that isolates the requesters of services (clients) from the providers of services (servers) by a well-defined encapsulating interface. In other words, clients are isolated from the implementation of services as data representations and executable code. This is one of the main differences that distinguish the distributed object-based model from the pure client/server model.

In the distributed object-based model, a client sends a message to an object, which in turns interprets the message to decide what service to perform. This service, or method, selection could be performed by either the object or a broker. The Java Remote Method Invocation (RMI) and the Common Object Request Broker Architecture (CORBA) are examples of this model.

Remote Method Invocation (RMI)

RMI is a distributed object system that enables you to easily develop distributed Java applications. Developing distributed applications in RMI is simpler than developing with sockets since there is no need to design a protocol, which is an error-prone task. In RMI, the developer has the illusion of calling a local method from a local class file, when in fact the arguments are shipped to the remote target and interpreted, and the results are sent back to the callers.

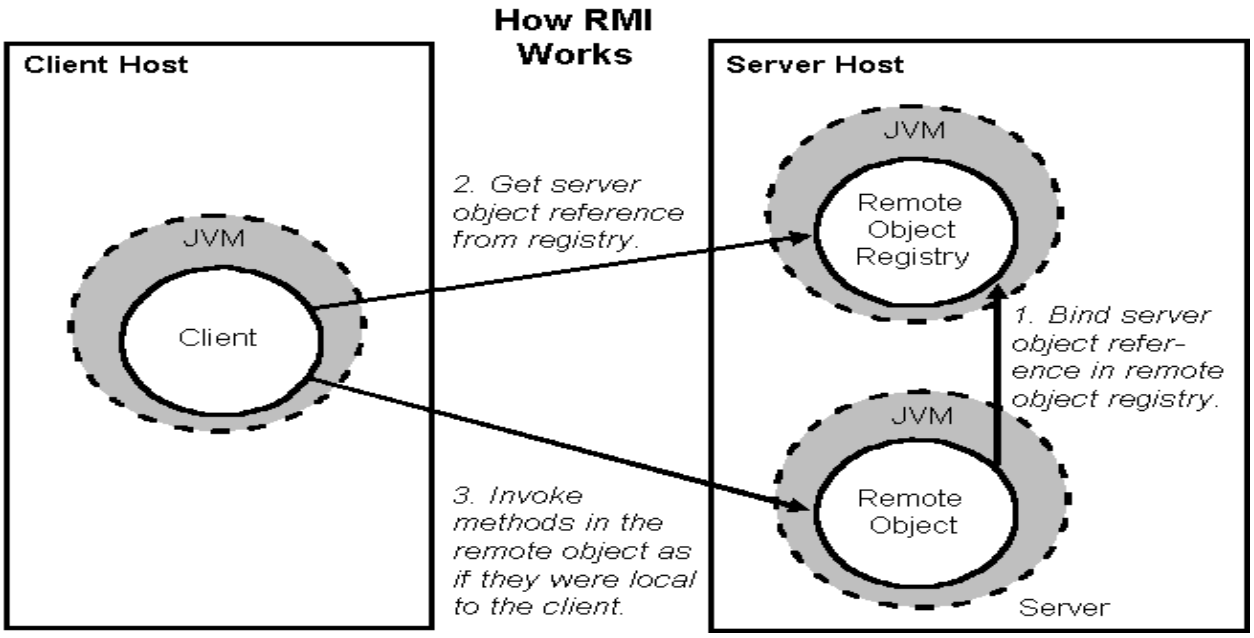


Figure showing the implementation of RMI

CORBA

The Common Object Request Broker Architecture (or CORBA) was developed to aid in distributed objects programming. It is important to note that CORBA is simply a specification. A CORBA implementation is known as an ORB (or Object Request Broker). CORBA was designed to be platform and language independent. Therefore, CORBA objects can

run on any platform, located anywhere on the network, and can be written in any language that has Interface Definition Language (IDL) mappings.

Similar to RMI, CORBA objects are specified with interfaces. Interfaces in CORBA, however, are specified in IDL.

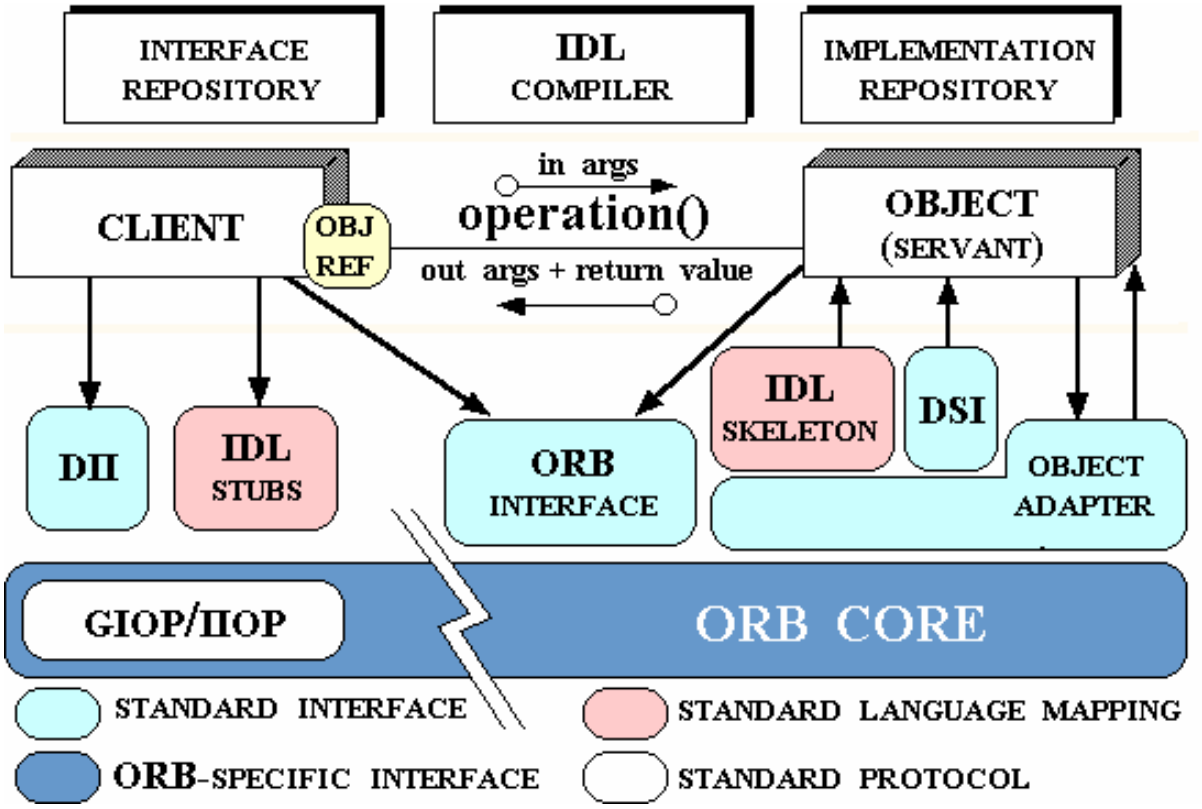


Figure showing the implantation of CORBA

3.4. DISCUSSION

Developing distributed object-based applications can be done in Java using RMI or JavaIDL (an implementation of CORBA). The use of both technologies is similar since the first step is to define an interface for

the object. Unlike RMI, however, where interfaces are defined in Java, CORBA interfaces are defined in the Interface Definition Language (IDL). This, however, adds another layer of complexity where the developer needs to be familiar with IDL, and equally important, and its mapping to Java.

CHAPTER 4
IMPLEMENTATION AND RESULTS

4.1. PROBLEM DEFINTION

The basic problem lies in multiplying two matrices of dimension M X M to produce a third resultant matrix,

Serial Matrix Multiplication

Suppose we have to multiply to matrices of dimensions M x M. The code should be :

```
void matrixmultiply (mt_A, mt_B, mt_C, int M)
{
     int i, j, k;
     for(i=0;i<n;i++)
     {
          for(j=0;j<n;j++)
          {
               for(k=0;k<n;k++)
               {
                    C[i][j]=C[i][j]    +    mt_A[i][k]    *
mt_B[k][j];
               }
          }
     }
}
```

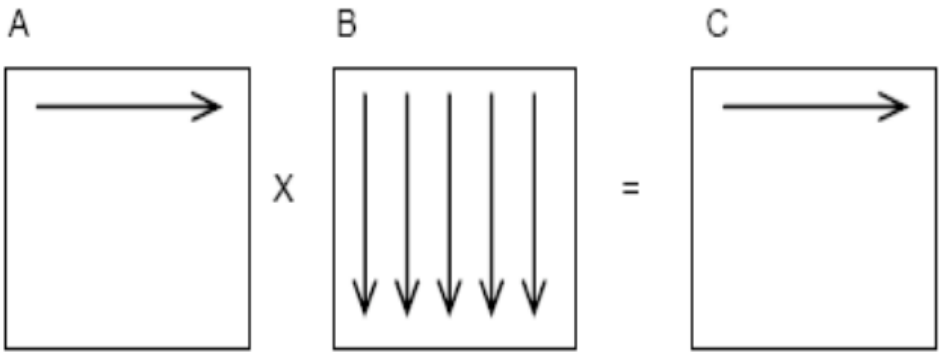


Figure showing the implementation of Serial Matrix Multiplication

Parallelizing Matrix Multiplication

We shall use the Direct Row – Stripped Partition to parallelize matrix multiplication. It is a master-slave program approach.

- One Master, several Slaves
- One node will run the master program, the rest will run the slave program.

A0	B	C0
A1		C1
A2		C2
Matrix A is partitioned equally depending on the number of slaves. Each node is sent to different slaves.	Matrix B is sent to all the slaves.	Matrix multiplication is performed on different slaves and the result is finally integrated on the master.

4.2. JAVA IMPLEMENTATION

There are 1 Master Nodes and N Slave Nodes.

The dimensions of the matrices are M x M.

Depending on the number of Slave nodes, one of the matrix's rows are divided into M/N blocks.

The Master Node sends one matrix of dimension M x M and another matrix of dimension M/N x M.

The matrix multiplication operations are performed on the respective Slave Nodes.

The resultant matrix is of dimension M/N x M.

The resultant matrix is sent to the Master Node.

The Master Node integrates the resultant matrices to get the final product matrix.

CODE

Master Side :

```
import java.io.*;
```

```
import java.net.*;
import java.util.Random;

public class TCPClientMatrix
{
     public static void main (String argv[]) throws Exception
     {
          int number_of_machines=4;
          String no_of_machines;
          String dim;
          String result;
          String server1;
          String server2;
          String server3;
          String server4;

          BufferedReader inFromUser = new BufferedReader (new
InputStreamReader (System.in));

          System.out.println("Enter IP Server :");
          server1 = inFromUser.readLine();

          System.out.println("Enter IP Server :");
          server2 = inFromUser.readLine();

          System.out.println("Enter IP Server :");
          server3 = inFromUser.readLine();

          System.out.println("Enter IP Server :");
          server4 = inFromUser.readLine();

          Socket clientSocket1 = new Socket(server1,6001);
          Socket clientSocket2 = new Socket(server2,6002);
          Socket clientSocket3 = new Socket(server3,6003);
          Socket clientSocket4 = new Socket(server4,6004);

          DataOutputStream outToServer1 = new
DataOutputStream(clientSocket1.getOutputStream());
          DataOutputStream outToServer2 = new
DataOutputStream(clientSocket2.getOutputStream());
          DataOutputStream outToServer3 = new
DataOutputStream(clientSocket3.getOutputStream());
          DataOutputStream outToServer4 = new
DataOutputStream(clientSocket4.getOutputStream());

          BufferedReader inFromServer1 = new BufferedReader(new
InputStreamReader(clientSocket1.getInputStream()));
```

```
          BufferedReader inFromServer2 = new BufferedReader(new
InputStreamReader(clientSocket2.getInputStream()));
          BufferedReader inFromServer3 = new BufferedReader(new
InputStreamReader(clientSocket3.getInputStream()));
          BufferedReader inFromServer4 = new BufferedReader(new
InputStreamReader(clientSocket4.getInputStream()));



          System.out.println("Enter the dimensions");
          dim   =   inFromUser.readLine();
          int dimension = Integer.parseInt(dim);

          System.out.println("Sending data...");

          String rows=Integer.toString(dimension/4);
          outToServer1.writeBytes(rows + '\n');
          outToServer2.writeBytes(rows + '\n');
          outToServer3.writeBytes(rows + '\n');
          outToServer4.writeBytes(rows + '\n');

          String columns=Integer.toString(dimension);
          outToServer1.writeBytes(columns + '\n');
          outToServer2.writeBytes(columns + '\n');
          outToServer3.writeBytes(columns + '\n');
          outToServer4.writeBytes(columns + '\n');

          System.out.println("Data sent...");

          long start = System.currentTimeMillis();

          /* for (int rowA=0;rowA<dimension/4;rowA++)
          {
               for (int colA=0;colA<dimension;colA++)
               {
                    String input = "1";
                    outToServer1.writeBytes(input + '\n');
                    outToServer2.writeBytes(input + '\n');
                    outToServer3.writeBytes(input + '\n');
                    outToServer4.writeBytes(input + '\n');

               }
          }



          // MatrixB
```

```
            for (int rowB=0;rowB<dimension;rowB++)
            {
                 for (int colB=0;colB<dimension;colB++)
                 {
                      String input = "1";
                      outToServer1.writeBytes(input + '\n');
                      outToServer2.writeBytes(input + '\n');
                      outToServer3.writeBytes(input + '\n');
                      outToServer4.writeBytes(input + '\n');
                 }
            }*/

            // To output matrixC (C = A.B)

            System.out.println("on Client's Screen :");
            System.out.println("Matrix C ");

            /* for(int rowC=0; rowC<dimension/4; rowC++)
            {
                 for(int colC=0; colC<dimension; colC++)
                 {

                 }
            }*/

            result = inFromServer1.readLine();
            System.out.println(result);
            result = inFromServer2.readLine();
            System.out.println(result);
            result = inFromServer3.readLine();
            System.out.println(result);
            result = inFromServer4.readLine();
            System.out.println(result);
            System.out.println("Resultant Matrix Received");
            System.out.println("Time elapsed: " +
(System.currentTimeMillis() - start));

            clientSocket1.close();
            clientSocket2.close();
            clientSocket3.close();
            clientSocket4.close();
      }
}
```

Slave Side :

```
import java.io.*;
import java.net.*;
import java.util.Random;

public class TCPServerMatrix2
{
      public static void main (String argv[]) throws Exception
      {
            String rows;
            String columns;
            String clientInput;
            String serverResult;
            ServerSocket welcomeSocket = new ServerSocket(6001);
            while(true)
            {
                  Socket connectionSocket = welcomeSocket.accept();
                  System.out.println("Connection accepted");
                  BufferedReader inFromClient = new BufferedReader (new
InputStreamReader(connectionSocket.getInputStream()));
                  DataOutputStream outToClient = new
DataOutputStream(connectionSocket.getOutputStream());

                  rows = inFromClient.readLine();
                  int rowsMat= Integer.parseInt(rows);

                  columns = inFromClient.readLine();
                  int colsMat = Integer.parseInt(columns);

                  System.out.println("Receiving data...");


                  // To Create matrices
                  double[][] matrixA = new double[rowsMat][colsMat];
                  double[][] matrixB = new double[colsMat][colsMat];
                  double[][] matrixC = new double[rowsMat][colsMat];
                  // Get input from client To fill Matrices
                  // MatrixA
                  for (int rowA=0;rowA<rowsMat;rowA++)
                  {
                        for (int colA=0;colA<colsMat;colA++)
                        {
                              matrixA[rowA][colA] = rowA + colA;
                        }
```

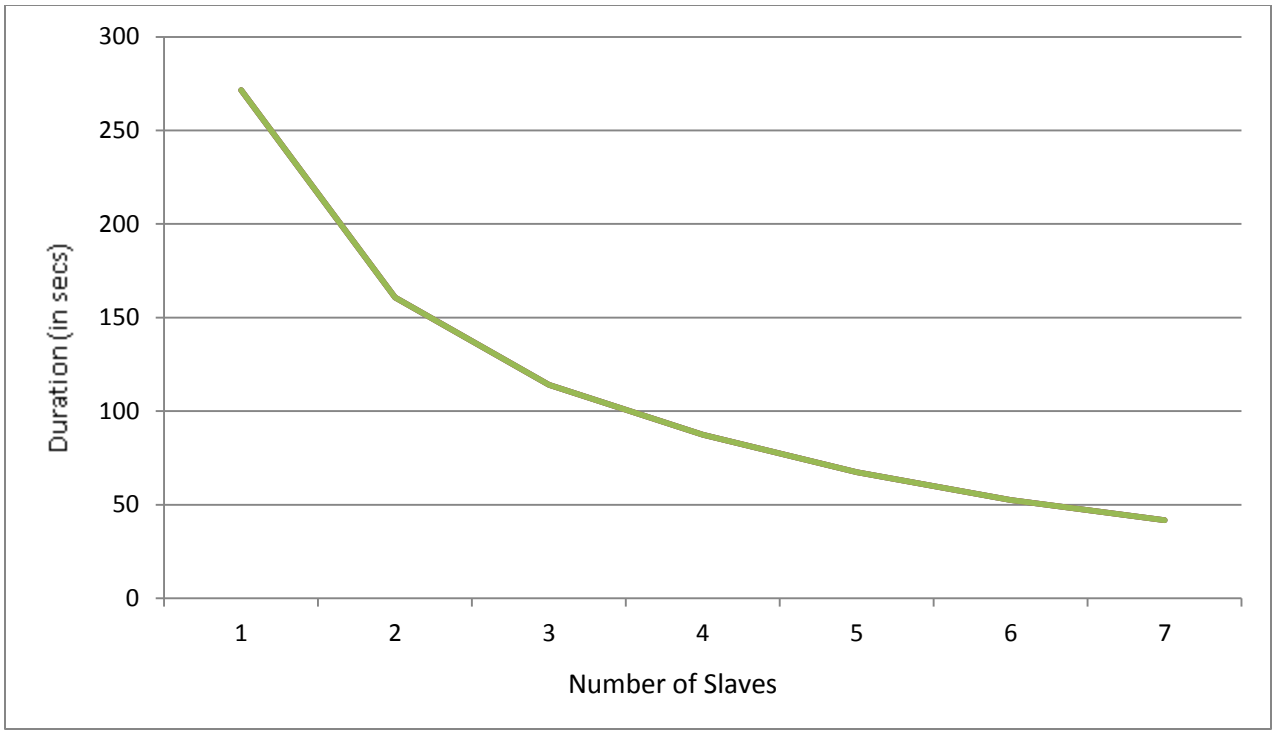
```
                }
                // MatrixB
                for (int rowB=0;rowB<colsMat;rowB++)
                {
                      for (int colB=0;colB<colsMat;colB++)
                      {
                            matrixB[rowB][colB] = rowB + colB;
                      }
                }

                // Multiplication C =A.B
                for(int rowA=0; rowA<rowsMat; rowA++)
                {
                      for(int colB=0; colB<colsMat; colB++)
                      {
                            for(int element=0; element<colsMat;
element++)
                            {
                                  matrixC[rowA][colB] +=
matrixA[rowA][element]*matrixB[element][colB];
                            }
                      }
                }

                System.out.println("Sending data...");

                outToClient.writeBytes("Server 2 done " + '\n');

                // To output matrixC (C = A.B)
                for(int rowC=0; rowC<rowsMat; rowC++)
                {
                      for(int colC=0; colC<colsMat; colC++)
                      {
                            serverResult = matrixC[rowC][colC]+'\n';
                            outToClient.writeBytes(serverResult);
                      }
                }
                System.out.println("Data sent");
                connectionSocket.close();
          }
     }
}
```

4.3. EXPERIMENTAL RESULTS

The code was executed with varying number of slave nodes and duration of computation was obtained. The result was then tabulated and a graph was drawn based on it.

Dimension	Number of Slave Nodes	Duration (in secs)
2100 x 2100	1	271.468
2100 x 2100	2	160.701
2100 x 2100	3	113.995
2100 x 2100	4	87.406
2100 x 2100	5	67.360
2100 x 2100	6	52.503
2100 x 2100	7	41.782

Table showing duration of computation with the number of slave nodes



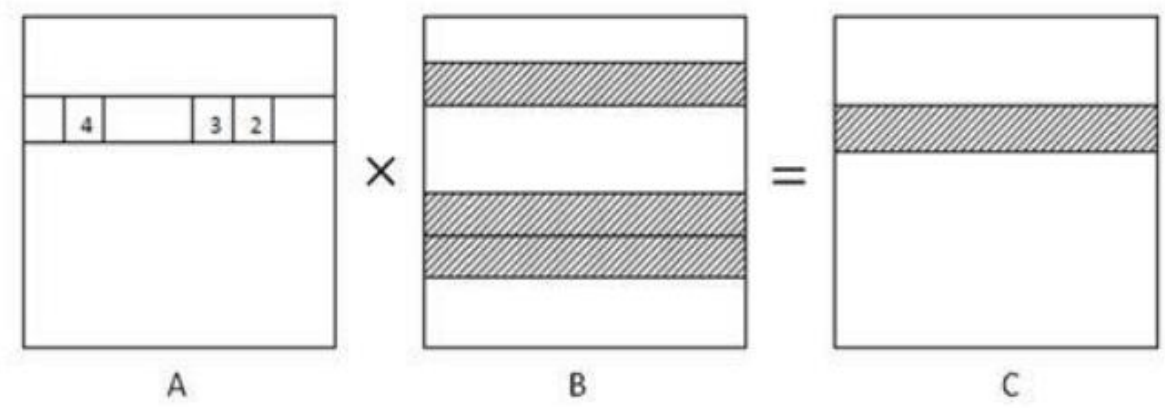
Graph depicting variation of duration of computation with number of slaves

4.4. MAPREDUCE IMPLEMENTATION

One of the matrices is divided in partsize x partsize blocks. The other matrix is divided into partsize row strips.

A map multiplies block of the first matrix with the row of the second matrix and produces partial results.

Every block multiplication is handled by one MAP task. Matrix data is stored in a Hadoop file.



CODE

```
importjava.io.IOException;
importjava.util.*;

importorg.apache.hadoop.fs.Path;
importorg.apache.hadoop.conf.*;
importorg.apache.hadoop.io.*;
importorg.apache.hadoop.mapreduce.*;
importorg.apache.hadoop.mapreduce.lib.input.FileInputFormat;
importorg.apache.hadoop.mapreduce.lib.input.TextInputFormat;
```

```
importorg.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
importorg.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

publicclassOneStepMatrixMultiplication {

    publicstaticclassMap extendsMapper<LongWritable, Text, Text, Text> {
        publicvoidmap(LongWritable key, Text value, Context context)
throwsIOException, InterruptedException {
            Configuration conf = context.getConfiguration();
            intm = Integer.parseInt(conf.get("m"));
            intp = Integer.parseInt(conf.get("p"));
            String line = value.toString();
            String[] indicesAndValue = line.split(",");
            Text outputKey = newText();
            Text outputValue = newText();
            if(indicesAndValue[0].equals("A")) {
                for(intk = 0; k < p; k++) {
                    outputKey.set(indicesAndValue[1] + ","+ k);
                    outputValue.set("A,"+ indicesAndValue[2] + ","+
indicesAndValue[3]);
                    context.write(outputKey, outputValue);
                }
            } else{
                for(inti = 0; i < m; i++) {
                    outputKey.set(i + ","+ indicesAndValue[2]);
                    outputValue.set("B,"+ indicesAndValue[1] + ","+
indicesAndValue[3]);
                    context.write(outputKey, outputValue);
                }
            }
        }
    }

    publicstaticclassReduce extendsReducer<Text, Text, Text, Text> {
        publicvoidreduce(Text key, Iterable<Text> values, Context context)
throwsIOException, InterruptedException {
            String[] value;
            HashMap<Integer, Float> hashA = newHashMap<Integer, Float>();
            HashMap<Integer, Float> hashB = newHashMap<Integer, Float>();
            for(Text val : values) {
                value = val.toString().split(",");
                if(value[0].equals("A")) {
                    hashA.put(Integer.parseInt(value[1]),
Float.parseFloat(value[2]));
                } else{
```

```
                    hashB.put(Integer.parseInt(value[1]),
Float.parseFloat(value[2]));
                }
            }
            intn = Integer.parseInt(context.getConfiguration().get("n"));
            floatresult = 0.0f;
            floata_ij;
            floatb_jk;
            for(intj = 0; j < n; j++) {
                a_ij = hashA.containsKey(j) ? hashA.get(j) : 0.0f;
                b_jk = hashB.containsKey(j) ? hashB.get(j) : 0.0f;
                result += a_ij * b_jk;
            }
            if(result != 0.0f) {
                context.write(null, newText(key.toString() + ","+
Float.toString(result)));
            }
        }
    }

    publicstaticvoidmain(String[] args) throwsException {
        Configuration conf = newConfiguration();
        // A is an m-by-n matrix; B is an n-by-p matrix.
        conf.set("m", "2");
        conf.set("n", "5");
        conf.set("p", "3");

        Job job = newJob(conf, "MatrixMatrixMultiplicationOneStep");
        job.setJarByClass(OneStepMatrixMultiplication.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);

        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        FileInputFormat.addInputPath(job, newPath(args[0]));
        FileOutputFormat.setOutputPath(job, newPath(args[1]));

        job.waitForCompletion(true);
    }
}
```

CHAPTER 5
CONCLUSION

5.1. CONCLUDING REMARKS

The focus of the project was in analyzing the time required for parallel computation using two separate techniques using two large matrices. Though it has been implemented using two square matrices, the code can be extended and developed to include non-square matrices also.

In this project we implemented and analyzed the parallel matrix multiplication implementations with Java using master/slaves algorithm and on a Hadoop cluster using Map/Reduce algorithm to achieve high performance. It has been shown that the performance of parallel model is reliable and saves more time than the serial model. It has also been shown that MapReduce works better.

REFERENCES

References:

[1] www.google.com

[2] www.wikipedia.com

[3] Java Network Programming – 4th Edition – O' Reilly Media

[4] Data-Intensive Text Processing with Map Reduce

[5] www.oracle.com/technetwork

[6] www.javaworld.com

[7] hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html

[8] www.ibm.com/software/data/infosphere/hadoop/mapreduce/

[9] www.youtube.com

APPENDIX A : SYSTEM INFORMATION

OS NAME	Microsoft Windows 7 Home Basic
Version	6.1.7601 Service Pack 1 Build 7601
Processor	Intel(R) Core™ i5-2450M CPU @ 2.5 GHz
BIOS Version	Phoenix Technologies Ltd. 03QB, 12/14/2011
Windows Directory	C:\Windows
System Direction	C:\Windows\system32
Boot Device	\Device\HarddiskVolume4
RAM	4.00 GB
Physical Memory	3.92 GB
Virtual Memory	7.83 GB

APPENDIX B : COMPILING AND RUNNING INFORMATION

The Source files (Master side and slave side) are first copied to the respective machines.

Compilation :

javac <filename>.java

Running :

java filename

First the slave side programs are compiled and executed. Then, the master side program is compiled and executed.

In the master side, the IP Addresses of the slave side nodes are given and the connection is set up.

APPENDIX C : SOURCE FILE INFORMATION

The source files are written and compiled in Java JDK 1.7

APPENDIX D : JAVA FUNCTION CALLS

new Socket(<Server Address>, <Port No>)

It creates the socket connection with the server address on the specified port no.

new DataOutputStream(<Socket>.getOutputStream());

It is a link for the outgoing messages on the specified socket.

new BufferedReader(new InputStreamReader (<Socket>.getInputStream()));

It is a link for the incoming messages on the specified socket.

<socket>.writeBytes(rows + '\n')

It is used to send the message over the socket link.

inFromServer1.readLine()

It is used to receive the message over the socket link.

new ServerSocket(<port no>)

It opens a link for the socket connection on the specified port no.

Integer.parseInt(<string>)

It converts a string into an integer.

<socket>.accept()

It accepts a socket connection.

<socket>.close()

It closes the socket connection.

APPENDIX E : MAP REDUCE FUNCTION CALLS

context.getConfiguration()

It retrieves the configuration of the system.

line.split(<identifier>)

It splits the line on occurrence on the specified identifier.

context.write(<Key>, <Value>)

It appends the specified value on the specified key.

publicvoidmap()

It implements the mapping operation.

publicvoidreduce()

It implements the reduce operation.

hashA.put()

It adds the values to the hash map.

hashA.containsKey(<key>)

It checks whether the value is present or not.

context.write()

It outputs the result.

conf.set()

It sets the configuration parameters.

newJob()

It assigns a new job according the configuration parameters with a job name.

setJarByClass()

It creates the class file.

setMapperClass()

It assigns the mapper function to the associated class.

setReducerClass()

It assigns the reducer function to the associated class.

setInputFormatClass()

It sets the format in which the input is to be taken.