Hadoop, MapReduce paradigm and its applications

Abstract—**There is an ever increasing need for faster computation. Being able to compute faster is crucial for Natural language processing because most of its algorithms are based on machine learning which need to process lots of data. Among many solutions available Hadoop provides the most simple and effective solution for faster computation. This article intends to serve as tutorial for Hadoop and provides examples to begin programming in Hadoop.**

Keywords—Hadoop, parallel-processing, LCS in hadoop, Large Dna sequence matching.

I. Introduction

There is always a heavy demand for faster computation as new algorithms begin to process more and more data, which consumes a large amount of time. Historically people have tried out many solutions, some include :

- Message passing
- Hadoop

There are two major ways of achieving faster computation. One of them is to use a fast hardware i.e. a set of fast processors, large amount of ram. The current speed at which processors operate is not enough and at times not necessary when operations are I/O bound. Another problem arises when we want many processors to work in a system, which requires special designs of motherboards which are very limited in design and are not scalable. This method is not economical.

Second option is clustered computing. A cluster is set of computer working together to perform a single operation. It initially began in 1995 when people invented the message passing interface which allowed programs to synchronize and communicate messages and share data over the lan. It laid the basics for the MapReduce paradigm.

Hadoop is a framework which follows the MapReduce paradigm. Because of its easy to understand interface it is being widely adapted by more and more people everyday. The idea for hadoop started when Google published a paper on parallel processing in a cluster of commodity computers. Hadoop is an open source framework which intends to replicate the ideas given in the paper. The basic idea in Hadoop follows a divide and conquer approach, given a large amount of data, the data is chunked into pieces and distributed among systems which the process on the data independently and finally combine the results of all the independent processes and produce the output. Formally this is known as Map and Reduce.

In this article we provide a basic tutorial for anyone interested in programming in Hadoop. It also provides applications where hadoop has been successfully applied. The rest of the article is organized as follows. In section 2 we explain the map reduce paradigm and explain how to setup a Hadoop cluster. In section 3 we explain how Hadoop works and the components that make Hadoop. In section 4 we explain the examples that we implemented in Hadoop. They include:

- Least common subsequence
- Dna sequence matching

In section 5 we describe the various other possibilities that Hadoop opens up for natural processing related tasks. Finally in section 6 we give the conclusions.

II. Hadoop and the MapReduce paradigm

A. MapReduce paradigm

Hadoop is suitable for embarrassingly parallel problems (i.e. those problems which do not need much effort to be parallelized) and I/O intensive problems. The map reduce paradigm is taken from functional programming map operation applies a function to a set of data produces a set of results and reduce operation applies a function to a set of data and produces a scalar.

The same approach was applied in MPI (message passing interface) where the scatter operation divided the data set into chunks and distributed it to other systems for processing and reduce collected all the data and was the final stage of processing.

MapReduce takes the same approach with a slight variation. In the Map phase a particular function is applied to a part of the data, not on the whole. Many such mapper tasks are run simultaneously, thus providing parallel computation. Reduce collects output of the map tasks and produces the final output.

B. Hadoop cluster installation

Setting up the cluster is just a matter of editing the configuration files in the etc/hadoop/ directory in the hadoop package. Our setup consisted of 3 machines and their ip address where 192.168.1.1(master), 192.168.1.2(slave1), 192.168.1.3(slave2) with their /etc/hosts file edited properly to match their ip addresses. All the relative reference of the directories is from the directory of the Hadoop installation.

```
#create a new id as hduser in all the systems, this
    id will be the administrator for the cluster
useradd hduser
adduser hduser sudo
passwd hduser
#login as hduser in all the machines
ssh-keygen -t rsa -C "cluster" #generating the ssh
    key so that the machines can work together.
ssh-copy-id localhost # add the generated key to
    your own machine
ssh-copy-id hduser@192.168.1.2 # add the generated
    key to slave1
ssh-copy-id hduser@192.168.1.3 # add the generated
    key to slave2
```

Next use any text editor and edit these configuration files and paste these configurations.
/etc/hadoop/core-site.xml:

```
<property>
  <name>fs.default.name</name>
  <value>hdfs://master:9002</value>
</property>
<property>
  <name>io.file.buffer.size</name>
  <value>131072</value>
</property>
```

Edit these files in the Namenode(the master server) and create the directory hadoopNN

/etc/hadoop/hdfs-site.xml:

```
<property>
  <name>dfs.namenode.name.dir</name>
  <value>/hadoopNN</value>
</property>

<property>
  <name>dfs.blocksize</name>
  <value>67108864</value>
</property>
```

Edit these files in the Datanode(all the machines of the cluster) and create the directory hadoopDN

/etc/hadoop/hdfs-site.xml:

```
<property>
  <name>dfs.datanode.data.dir</name>
  <value>/hadoopDN</value>
</property>
```

Edit the file etc/hadoop/slaves and add the host names of the machines in your cluster.Also edit the file etc/hadoop/hadoop-env.sh and set the JAVA_HOME variable in the file.

```
# login as hduser
su - hduser
#To start hadoop, format the namenode
<HADOOP_PREFIX>/bin/hdfs namenode -format
#Then start hdfs
<HADOOP_PREFIX>/sbin/start-dfs.sh
#Now you can submit jobs by running the command
hadoop jar <jar file> mainClass args
```

III. Hadoop components

Hadoop consists of mainly 2 components

- The hadoop distributed file system
- The hadoop job scheduler

These two components are the the core part of hadoop and are the main selling points of hadoop. The hadoop distributed file system or The HDFS for short is a distributed file system which automatically manages the files on the cluster. The basic operation of the HDFS is to break up the given file into blocks (these are different from the blocks in the filesystem) each of maximum size as given in the configuration files. Generally large clusters have a very large block size.

The HDFS also stores the meta data of the files stored as blocks, in the cluster. The concept of blocks is different from the one in filesystems. In filesystem a block cannot be reused to store more that one file even if the block is empty, whereas in Hadoop because the block size is much larger (in the order of MB) one cannot afford to waste the space so it stored more than one file in the block. The block structure of HDFS is important because it plays a crucial role in parallel processing. HDFS also maintains a replica of the each block in more than one system of the cluster. An advantage one can clear see is that if any system crashes, HDFS can always recover data, so it provides reliability on commodity hardware.

Just as in functional programming a MapReduce job applies a particular function on the data. When a MapReduce job is submitted to Hadoop, Hadoop inspects the input for the program and calculates the number of blocks (aka to as InputSplits in the API) that the input files occupy and for each inputSplit it creates a new mapper process and applies it on the split.

So if a particular file occupies 128 MB and the block size is 64 MB then 2 mapper (by default, more can be configured) processes will run and each acting on its own split. Another advantage of storing a block in more than one system is that when a MapReduce task is run, the framework makes sure that the mapper works on the files directly available on the hard drive. In this way the job load is equally shared among all the systems in the cluster.

As soon as a mapper task completes, the results of all the mappers are transfered to the reducer. Generally there is only one reducer (we can have more than one). The reducer then processes on the intermediate results and writes the output to the filesystem. Such a system opens up a range of possibilities for faster and parallel processing. Few examples are:

A. *Reading log files*

Websites and softwares generate lots of log files. These log files can contain so much information, take the example of flipkart, it provides suggestions to its users on the similar products when a user chooses a particular product. All this is done by processing these log files in parallel.

B. *Indexing*

A very crucial part of NLP is indexing. Indexing a large corpus is a very time consuming task. It can be done very efficiently by using Hadoop.

C. *The job scheduler*

The job scheduler takes care that application are given the amount of resources they request and makes sure that all the mappers complete properly.

IV. Projects in Hadoop

A. *Least common subsequence problem*

Being able to generate summaries is great but we need a basis to rank various algorithms. Traditionally evaluation of summarization involved human judgments for different metrics like grammar, coherence. However manual evaluation of summaries on a large scale requires lots of human effort and is an inefficient option. It is not feasible on a frequent basis.

Many methods have been proposed for automatically comparing generated summaries to a human written reference summary. These methods are : cosine similarity, unit overlap (i.e. unigram or bigram), and longest common subsequence. Here we have taken the problem of longest common subsequence and parallelized it because it takes a very long time without parallelization. The evaluation method has been taken from the paper ROUGE : A Package for Automatic Evaluation of Summaries, Chin-Yew Lin.

Longest common subsequence: A sequence $Z = [z_1, z_2, \ldots, z_n]$ is a subsequence of another sequence $X = [x_1, x_2, \ldots, x_m]$, if there exists a strict increasing sequence $[i_1, i_2, ..., i_k]$ of indices of X such that for all $j = 1, 2, \ldots, k$, we have $x_{ij} = z_j$ (Cormen et al. 1989). Given two sequences X and Y, the longest common subsequence (LCS) of X and Y is a common subsequence with maximum length.

1) Sentence level LCS: We see the given sentences as a list of words. The intuition is longer the common subsequence the more similar they are. We have used the F-measure proposed in the paper. Given X(reference summary) and Y(candidate summary) with length m and n respectively, the F-measure is defined as follows :

$$R_{lcs} = \frac{LCS(X,Y)}{m} \quad (1)$$

$$P_{lcs} = \frac{LCS(X,Y)}{n} \quad (2)$$

$$F_{lcs} = \frac{(1+\beta^2)R_{lcs}P_{lcs}}{R_{lcs} + \beta P_{lcs}} \quad (3)$$

where LCS(X,Y) is the length of the longest common subsequence of the strings X and Y and $\beta = P_{lcs}/R_{lcs}$ when $\partial F_{lcs}/\partial R_{lcs} = \partial F_{lcs}/\partial P_{lcs}$

2) Summary level LCS: The concept of sentence level lcs can be extended to summary level. When applying to summary level, we take the union of LCS matches between a reference summary sentence, r_i ,and every candidate summary sentence, c_j . Given a reference summary of u sentences containing a total of m words and a candidate summary of v sentences containing a total of n words, the summary level LCS-based F-measure can be computed as follows:

$$R_{lcs} = \frac{\sum_{i=1}^{u} LCS(X,Y)_{\cup}(r_i, C)}{m} \quad (4)$$

$$P_{lcs} = \frac{\sum_{i=1}^{u} LCS(X,Y)_{\cup}(r_i, C)}{n} \quad (5)$$

$$F_{lcs} = \frac{(1+\beta^2)R_{lcs}P_{lcs}}{R_{lcs} + \beta P_{lcs}} \quad (6)$$

here $\sum_{i=1}^{u} LCS(X,Y)_{\cup}(r_i, C)$ is the sum of the LCS of each reference line and the candidate summary. For example,if $r_i = w_1w_2w_3w_4w_5$, and C contains two sentences: $c_1 = w_1w_2w_6w_7w_8$ and $c_2 = w_1w_3w_8w_9w_5$, then the longest common subsequence of r_i and c_1 is w_1w_2 and the longest common subsequence of r_i and c_2 is $w_1w_3w_5$. The union longest common subsequence of r_i, c_1, and c_2 is $w_1w_2w_3w_5$ and $LCS(X,Y)_{\cup}(r_i, C) = 4/5$.

3) Implementing it in Hadoop: In the problem, each sentence in the candidate summary is compared with every sentence in the reference summary, it is basically a cross product of the sentences of the two input files. Calculating cross product is a very difficult task(and the most time consuming) in hadoop since files need to be present in each system or they will be share at runtime (which will add to the computation time due to the latency in the network). So the basic algorithm is as follows:

1) L= compute cross product of input files X,Y
2) for each tuple in L compute LCS

In order to solve the problem of runtime sharing of files we changed the replication factor of the HDFS to 3 so that every file is available in all the 3 systems. The LCS algorithm is a slightly modified algorithm from wikipedia where it compares for charecters, here we compare for words.

```
public class SSMap extends Mapper<LongWritable, Text
    ,IntWritable,LongWritable>{

        private Path inFile = new Path("/user/hduser
            /input/test2.txt");

        public ArrayList<String> lcs(ArrayList<
            String> a, ArrayList<String> b) {
            int[][] lengths = new int[a.size()+1][b.
                size()+1];

            // row 0 and column 0 are initialized to
                0 already

            for (int i = 0; i < a.size(); i++)
                for (int j = 0; j < b.size(); j++)
                    if (a.get(i).compareTo(b.get(j))
                        ==0)
                        lengths[i+1][j+1] = lengths[
                            i][j] + 1;
                    else
                        lengths[i+1][j+1] =
                            Math.max(lengths[i+1][j
                                ], lengths[i][j+1]);

            // read the substring out from the
                matrix
            ArrayList<String> sb = new ArrayList<
                String>();
            for (int x = a.size(), y = b.size();
                x != 0 && y != 0; ) {
                if (lengths[x][y] == lengths[x-1][y
                    ])
                    x--;
                else if (lengths[x][y] == lengths[x
                    ][y-1])
                    y--;
                else {
                    assert a.get(x-1).compareTo(b.
                        get(y-1))==0;
                    sb.add(a.get(x-1));
                    x--;
                    y--;
                }
            }
            Collections.reverse(sb);
            return sb;

        public void map(LongWritable key, Text value
            ,
                    Mapper<LongWritable, Text,
                        IntWritable,
                        LongWritable>.Context
                        context) throws
```

```
                                IOException ,
                                InterruptedException
        {
                FileSystem fs = FileSystem.get(
                    Driver.conf);

                FSDataInputStream fr = fs.open(
                    inFile);
                Scanner scn = new Scanner(fr);

                HashSet<String> set = new HashSet<
                    String>();
                while(scn.hasNextLine())
                {

                        String s = scn.nextLine();
                        set.addAll( lcs(new
                            ArrayList<String>(Arrays
                            .asList(value.toString()
                            .split(" "))), new
                            ArrayList<String>(Arrays
                            .asList(s.split(" "))) )
                            );
                }
                long size = set.size();
                context.write(new IntWritable(new
                    Integer(1)), new LongWritable(
                    size));
                scn.close();
        }
}
```

Initially before starting the program we calculate the number of words in the two documents using the "wc -w" linux utility. Say X, Y are the reference and candidate summaries with m, n words respectively. When the program runs Hadoop reads each line from the InputSplit and gives it to the map function and for each line that the map function receives we calculate the lcs of it and each line from the second file. We store the words that the LCS function return in a hashed set so that we dont have duplicate elements.

After the given line is compared to each line in the second file, we will have the set of words which is the longest common subsequence of that line and the second file. We emit the key,value pair (1,sizeof(the set)). We use 1 as the key because internally hadoop shuffles and aggregates all the values of one key into one list. So when the reducer starts, the reducer will receive the input key as 1 and input value as list(LCS of every line of file 1). The reducer simply calculates the sum of every entry in the list and writes the output.

The reducers code:

```
public class SSR extends Reducer<IntWritable ,
    LongWritable , Text , LongWritable> {

        public void reduce(IntWritable key, Iterable
            <LongWritable> values , Reducer<
            IntWritable ,LongWritable ,Text ,
            LongWritable>.Context context) throws
            IOException , InterruptedException
        {
                long sum = 0;
                for (LongWritable t:values)
                {
                        sum = sum + t.get();
                }
                context.write(new Text("sum : "),
                    new LongWritable(sum));
        }
}
```

4) Choosing which file will load in every map call: Initially the calculated values of m, n should be used to determine which to load every time(the one will small size will be loaded).

5) Controlling the Parallelization: The important part of the this program is controlling the parallelization. For each InputSplit, generally this is set to the block size of the cluster, a mapper process will be started. But this can be configured in hadoop using the configuration property "mapred.max.split.size" which can be set to how big the input split must be. For example if the input file is 900 kB we can start 3 mappers by setting this variable to 300 kB. One should remember that this variable must not be set to a very small value as it will increase the overhead of Hadoop thus will slow down the program.

B. DNA Matching problem

A main drawback of Hadoop is that one has to write code in java and also understand the API properly. In order to solve this problem the developers of Hadoop have given feature called as Hadoop streaming. It is very simple and easy to use command line interface which can run map reduce in any programming language. Through the following DNA matching problem written in c++ we illustrate the usefulness of Hadoop streaming.

Pattern matching is at the heart of bioinformatics. Pattern matching is used for matching a particular DNA string to a large DNA database to find out which organisms have a particular trait, then classify organisms and build phylogenic trees.

Fast algorithms are available for DNA matching but they can be still sped up by using Hadoop as matching on large databases with different organisms can be easily parallelized (aka embarrassingly parallel problem). Here we discuss one such Fast matching algorithm which matches one pattern with another DNA string and then we use Hadoop streaming to parallelize it.

1) The algorithm: The whole algorithm boils down to indexing the given DNA string and the search string so that patterns can be easily found in the string. This algorithm is implemented as described in the paper Exact Multiple Pattern Matching Algorithm using DNA Sequence and Pattern Pair (Raju Bhukya et. al).

DNA is the basic blueprint of life and is represented as a long sequence of 4 alphabets: Adenine (A), Cytosine (C), Guanine (G), and Thymine (T). The same algorithm can be applied to RNA as RNA is composed of the same 4 components with thymine replace with Uracil (U). Since there are 4 alphabets we would have 16 pairs. They are {AA, AC, AT, AG, CA, CC, CT, CG, TA, TC, TT, TG, GA, GC, GT, GG}. The algorithm is as follows:

Input: String S of n characters and pattern P of m characters.

1) Store all the indexes and the frequency of occurrence of every consecutive pair of characters in S as a list.

2) Store all the indexes and the frequency of occurrence of every pair of characters in P as a list. Difference here is that each character belongs to only one pair.
3) Find the least frequent pair in P.
4) Align P's least frequent pair and the String using the index created.
5) Match the aligned strings to find if they match. While matching we use the created index of P to match in the descending order of the pairs occurring in P.

```
int return_positions()
{
  string S,P;

  P=string("<set the pattern here>");

  getline(cin,S);
  int n,m;
  n=S.size();

  m=P.size();
  vector<int> stab[16];
  int sidx[16]={0};
  vector<int> ptab[16];
  int pidx[16]={0};
  int ssort[16] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
       11, 12, 13, 14, 15};
  int psort[16]= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
      11, 12, 13, 14, 15};
  int found=1,ncmp=0,npat=0;

  bool odd = false;
  if(m%2 == 1)
  {
    odd = true;
  }

  short *sp,*pp;
  int i,j;

  for (i=0;i<n;i++)
  {
    sp= (short*) &S[i];
    int index = (*sp & 1536)>>9 | (*sp & 6)<<1;
    (stab[index]).push_back(i);
    sidx[index]++;
  }
  int t;

  for (i=0;i<16;i++)
  {
    for (j=i+1;j<16;j++)
    {
      if (sidx[i]>sidx[j])
      {
        t = sidx[i];
        sidx[i]=sidx[j];
        sidx[j]=t;

        t = ssort[i];
        ssort[i]=ssort[j];
        ssort[j]=t;
      }
    }
  }


  for (i=0;i<m;i=i+2)
  {
    pp= (short*)&P[i];
    int index = (*pp & 1536)>>9 | (*pp & 6)<<1;
    (ptab[index]).push_back(i);
    pidx[index]++;
  }

for (i=0;i<16;i++)
{
  for (j=i+1;j<16;j++)
  {
    if (pidx[i]<pidx[j])
    {
      t = pidx[i];
      pidx[i]=pidx[j];
      pidx[j]=t;

      t = psort[i];
      psort[i]=psort[j];
      psort[j]=t;
    }
  }
}

int y;
for (i=15;i>=0;i--)
{
  if(pidx[i]!=0)
  {
    y = psort[i];
    break;
  }
}
int diff;
for (i=0;i<(m/2)*2;i=i+2)
{
  pp = (short*)&P[i];
  int index = (*pp & 1536)>>9 | (*pp & 6)<<1;
  if (index == y)
  {
    diff=i;
    break;
  }
}


int k,b=14,l,c=0,a;

for (i=0;i<=sidx[y];i++)
{
  found = 1;
  k = stab[y].at(i) - diff;
  if(k<0)
  {
    continue;
  }
  sp = (short*)&S[k];

  ncmp +=2;
  for (l=0;l<16;l++)
  {

    if (pidx[l]!=0)
    {
      for (c=0;c<ptab[psort[l]].size();c++)
      {
        a = ptab[psort[l]].at(c);
        sp = (short*)&S[k+a];
        if( ((*sp & 1536)>>9 | (*sp & 6)<<1) !=
            psort[l] )
        {
          found = 0;
          break;
        }
      }
    }
    if (found==0)
    {
      break;
    }
  }
```

```
      if(odd==1 && S[k+m-1]!=P[m-1])
      {
        found =0;
      }
      if (found==1)
      {
        npat=npat+1;
        cout << 1 <<"\t"<< k<<endl;
      }
    }
}
```

2) How streaming works: The concept of streaming is same as that of MapReduce except we use another language instead of java. Just like in MapReduce , for each InputSplit a new mapper process is started, here for each InputSplit the given executable is executed. Once it is executed the data is inserted line by line into the program thats the reason there is a getline in the function. Suppose a large genome database is constructed with the name of the species and its genome with a tab space between them, then Hadoop will provide us with the full string which we can preprocess to extract the species name and its DNA sequence and then apply the algorithm.

In the output one can provide the species name(from the preprocess) as key with a tab space and the position where the pattern occurs. In this way Hadoop will recognize the species name as key and position as value. In the reduce phase it will shuffle and sort based on key and output it into the file. A simple example of Hadoop streaming: their ip addresses

```
hadoop jar hadoop-streaming -2.7.1.jar \
  -input myInputDirs \
  -output myOutputDir \
  -mapper <executables path>
```

the -input parameter takes directory which will contain the genome database similarly the -output myOutputDir directory is where the output will be written to.

V. Hadoop, Natural language processing and Future work

Hadoop is very useful for NLP related tasks as they involve processing a large number of files and extract information from them. Machine learning is very frequently used in NLP which learns various features and characteristics by reading a large number of files. Most of these file operations are independent i.e. features extracted from one file do not affect the extraction of features of another file.

A. Mahout

Mahout is a framework for scalable machine learning and data mining applications. It contains many machine learning algorithms which automatically use the power of Hadoop to parallelize the work. It has a large number of classifiers which are very useful in NLP.

B. Terrier

Terrier is a IR engine which supports Hadoop out of the box (though they do not yet support the new version). Terrier can index a large corpora using Hadoop in a quick and efficient way. It also contains may learning algorithms which are used for ranking the search results. It has the state of art retrieval.

C. Future work

We are working on automatic gazette creation for named entity extraction. Basic intro to the project:

The project automatically creates a gazette of named entities from a large corpus. Examples of named entities are, names of diseases, names of mechanical tools, names of lawyers. It takes a list of small number of examples (known as the seed list) and creates a bigger list. Also known as semi-supervised algorithm. Our focus was on extracting lawyers from a legal corpus. The basic algorithm is as follows. Extract the names of the people using a NER (based on a maximum entropy model from Mahout) and then use certain features of lawyers from the seed list to find out if the names detected by the NER is a lawyer or not. Since there is no interdependency in the files in the corpus this problem can be easily parallelized using Hadoop.

VI. Conclusion

Hadoop and the MapReduce paradigm are becoming very popular because it is quite simple, yet efficient way to parallelize and speed up processing. This article describes a lot about Hadoop but it is just the tip of the ice berg. There is a lot more to learn and lots interesting possibilities open up with Hadoop.

Acknowledgment

The author would like to thank Dr. Prasenjit Majumder for his help and support and for introducing him to the subject.

References

[1] The complete documentation for Hadoop is available at http://hadoop.apache.org/docs/current/

[2] LCS algorithm https://en.wikipedia.org/wiki/Longest_common_subsequence_problem

[3] Chin-Yew Lin, 2004, *ROUGE : A Package for Automatic Evaluation of Summaries* The F-measure for the LCS summarization.

[4] Raju Bhukya and DVLN Somayajulu *Exact Multiple Pattern Matching Algorithm using DNA Sequence and PatternPair* International Journal of Computer Applications, 2011

[5] About Mahout : mahout.apache.org

[6] Sachin Pawar, Rajiv Srivastava and Girish Keshav Palshikar *Automatic Gazette Creation for Named Entity Recognition and Application to Resume Processing*