Distributed Model Checking Using Hadoop

Rehan Abdul Aziz

October 25, 2010

Abstract

Various techniques for addressing state space explosion problem in model checking exist. One of these is to use distributed memory and computation for storing and exploring the state space of the model of a system. DiVinE is an open-source model checker whose infrastructure supports distributed model checking. In this report, an approach for distributed model checking is described that uses *Apache Hadoop* which is an open-source implementation of the programming paradigm called MapReduce. The algorithm is a distributed version of Breadth First Search which makes use of the DiVinE LIBRARY combined with the MapReduce framework.

Contents

1 Introduction

In Computer Science, it is sometimes of essential value to check that a system meets some particular specifications. One method, used very widely in the software industry is *testing* [1] in which the system's actual behaviour is checked against the expected behaviour over several manually designed *test cases*. *Testing* can also be applied to *simulation* of a system rather than the original system. Testing the system manually becomes more time and resource consuming as the size of the system under consideration increases. Moreover, this method does not guarantee that the sytem will be free of errors and bugs since some cases might be missed. Another approach is *deductive verification* in which a system is proved mathematically to meet specific properties with the help of automated theorem provers acting as proof checkers. This approach demands the involvement of specialists in the field, therefore it is very expensive and is rarely used.

A very promising method for testing hardware and software systems is called *model checking* [2]. Model checking is automatically checking the model of a system for given properties. The fact that model checking can be automated accounts for its growing popularity. A model checker works by exploring all the states that the system under consideration can reach. The set of all states of a system is called its *state space*. If the system does not meet a specific criterion, then the erroneous execution also called the *counterexample* is returned, which means that the error can be reproduced or analyzed. The model checker that is discussed in this report is DiVinE [3, 4, 5] which is the acronym for Distributed Verification Environment.

MapReduce [6] is a programming model introduced by Google for processing and generating large data sets using distributed computing. Users do not need to know any details of how the scheduling and distribution of tasks work, all such detail is abstracted in the MapReduce framework. On the programming level, only two functions namely *map* and *reduce* need to be specified. Very briefly, mapreduce works as follows: the input is given as a file or a directory of files. The input is divided into equal parts and given as key/value pairs to several instances of a user defined *map* function which run on several worker nodes in the distributed computing cluster. The output from the map function is also in the form of key/value pairs and is fed into several instances of a user defined *reduce* function and the output is written in a directory. This simple model can cover a wide range of real world problems, some of which are mentioned in [6]. There are several implementations of MapReduce, but we shall only refer to the open-source Hadoop [7] in our report.

In model checking, as the variables in the model of a system are increased,

its state space increases exponentially. Distributing the state space over different memory locations, as well as using distributed computing units is one solution addressing the issue of a blow up in the state space size. In this report, a combination of the DiVinE model checker with Hadoop is used to describe an algorithm which uses the MapReduce paradigm to distribute DiVinE model checker over a computing cluster. The rest of the report is organized as follows, Section 2 describes the working of MapReduce and Hadoop with the help of a detailed running example, which can also be viewed in isolation as a short tutorial for MapReduce and Hadoop. In Section 3, we briefly discuss distributed model checking and DiVinE model checker. Section 4 explains in detail our algorithm that uses Hadoop to distribute DiVine. In Section 5, the timings of our algorithm for a particular model are mentioned. Finally, in Section 6, we conclude the report and comment on potential future work.

2 MapReduce and Hadoop

MapReduce is a programming paradigm developed at Google for handling large data in a distributed computing environment. A typical MapReduce job consists of two functions, *map* and *reduce*, both defined by the user. Figure 1 shows a general MapReduce computation. The input is first divided into equal sized chunks, and each of them is fed to a user defined map function. The map function processes the data as instructed by the user and presents its output as key/value pairs. After all map functions have finished, MapReduce distributes the values according to the keys to a number of reducers. Each reducer first sorts all the keys, and then runs the user defined reduce function over them. Each reduce function finally outputs key/value pairs. Optionally, the programmer can also provide a *combiner* whose purpose is to process the data against each key on the same node as the mapper, before the output is sent to the reducers. A combiner can therefore be thought of as a 'local reducer'. A very intuitive example, used frequently in the literature is to explain the flow of a MapReduce word-count job. Suppose we wish to count the frequency of all the words present in a large input text. MapReduce breaks the input data into equally sized blocks. We define the map function to simply output 1 (value) against each word (key) it encounters. MapReduce collects all the outputs against each key. The reduce function is programmed to add up all the 1's against each key, and in this way, we get the frequency of each word. In the coming subsection, we shall take a look at a more detailed example regarding MapReduce.

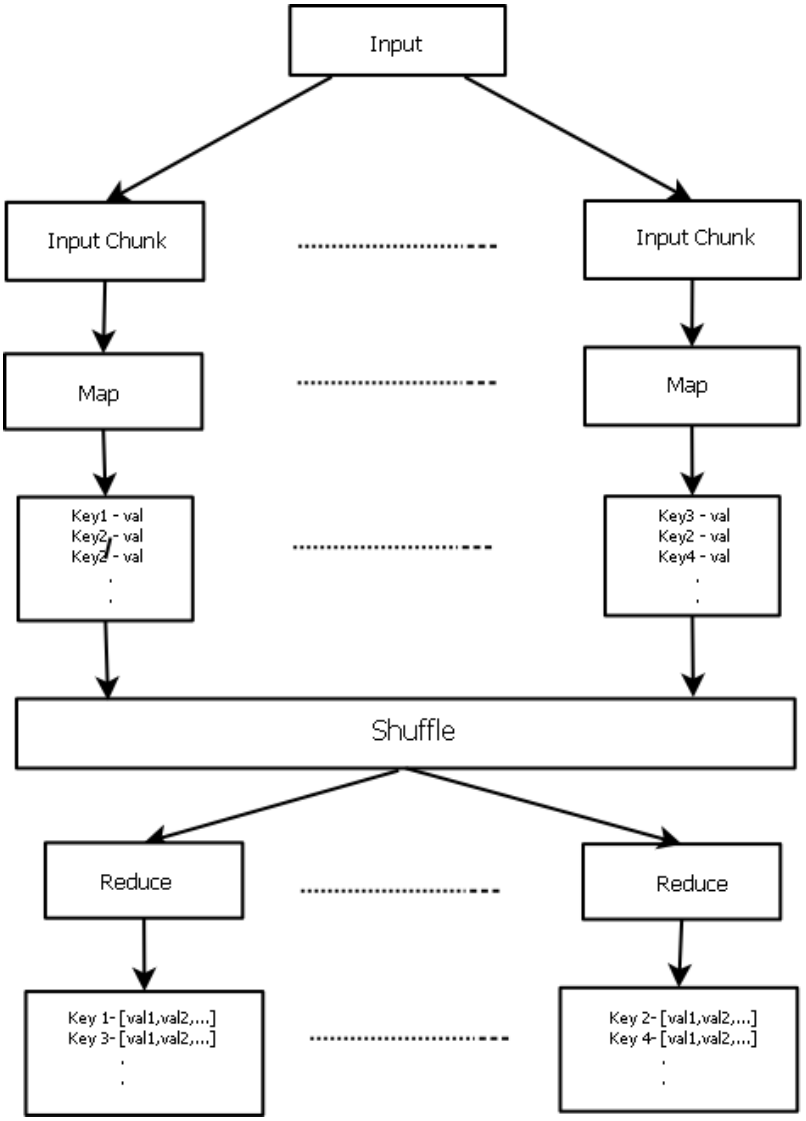

Figure 1: A typical MapReduce computation

Hadoop is an open-source implementation of MapReduce, with Yahoo! as its major contributor. Hadoop uses its own *Hadoop Distributed File System (HDFS)* with MapReduce. The structure of HDFS is quite similar to the Google File System (GFS) having a single master managing a number of slaves. The master is called the *namenode* while the slaves are called *datanodes*. The namenode stores the directory structure and also stores metadata such as replica locations related to the files. HDFS uses a large block size of 64 MB by default, and the files are broken down into chunks of this size to be stored on the HDFS.

MapReduce may seem like a very limited programming model with almost no co-ordination between the map and reduce tasks, and the restriction of viewing all data as key/value pairs, and there is some truth in this intuition. But the high-level nature of this programming model saves the programmer from a great amount of repetitive and cumbersome work, i.e. of deciding how the nodes communicate with each other, where data is stored, how requests are handled etc. With this limited control and high abstraction level comes convenience. Another feature of MapReduce that makes it very appropriate for our purpose is that theoretically, it scales almost linearly with the amount of input data and the size of the cluster.

2.1 Hadoop Running Example

To better elaborate the use of MapReduce and Hadoop, we will go through a simple running example in detail. This example can be considered as a brief tutorial on how to get started with MapReduce and Hadoop, after installing Hadoop. For setting up Hadoop on a single node or a cluster, we refer the reader to a very good tutorial written by Michael G. Noll which is available online. [8] From here onwards, we will assume that the reader has successfully setup Hadoop and is able to run MapReduce jobs.

2.1.1 Problem Statement

Java's Random class claims that the random integers it generates in a certain range follow a uniform distribution. Suppose we are given the task to check the validity of this claim very non-rigorously by generating a large amount of integers using its function Random.nextInt(int range) and analyzing them for uniformity. Our input consists of rows of data in the following format:

```
Year Count
```

For example:

```
1347 312
1470 243
1789 400
```

The first variable Year is a random number between 1 and 1999, with both the numbers being inclusive. The the second variable Count is another random variable between 1 and 499, both inclusive. Both the numbers are generated using the nextInt(int range) function of the Random class. If the distribution is indeed uniform, then the following conditions should be true, given a large amount of data:

1. Number of each year in the first column must be roughly equal.

2. The counts for each year must have an average near (1+499)/2 = 250.

Before moving on to a MapReduce interpretation of the problem, let us see briefly how the input is generated, so that the reader has a better grasp on the problem.

2.1.2 Generating the Input

Our input will consist of records in a file in the format given in the problem statement, where each row in the file consists of a year and a count separated by a space. In Hadoop, it is more convenient and efficient to use a small number of large files to represent the data instead of choosing a large number of smaller files. For this purpose, and for simplicity, we will generate all our input in one large file named number_counts.

```
public static void main(String[] args) throws Exception {

  String fileName = "//your//own//path//number_counts";
  BufferedWriter writer =
    new BufferedWriter(new FileWriter(fileName, true));

  //Instantiate a variable of Random class
  Random random = new Random();

  int year, count;
  //Generate 10 million records
  for (int i = 0; i < 10000000; i++) {
      year = random.nextInt(1000) + 1000;
      count = random.nextInt(500) + 1;
      writer.write(year + " " + count);
      writer.newLine();
  }
  writer.close();
}
```

Before running a MapReduce job, we should copy this input file to the Hadoop filesystem. This can be done using the following command:

```
$./hadoop dfs -copyFromLocal
 /your/own/path/number_counts number_counts
```

Now that the input is ready to be executed, we move on to define our map and reduce functions for the job, and also see how to configure them for our specific problem.

2.1.3 *Map* Function

It is more convenient to write code for MapReduce in any IDE rather than writing it in a simple text editor, thus we recommend programming in a good Java IDE. Please note that the jar file called *hadoop-version-core.jar* found in hadoop's installation directory must be included in the project's libraries. In our map function, we simply split each line of input by a space, separate the year and the count, and store the count against the year. The code for our mapper class is as follows:

```
public class MyMapper extends MapReduceBase
  implements Mapper<LongWritable, Text, Text, IntWritable> {

    public void map(LongWritable key, Text value,
      OutputCollector<Text, IntWritable> output,
      Reporter reporter) throws IOException {

      //Split the line by " " to separate year and count
      String [] lineValues = value.toString().split(" ");

      //Get the number and count
      Text number = new Text(lineValues[0]);
      int count = Integer.parseInt(lineValues[1]);

      //Store the value in the output
      output.collect(number, new IntWritable(count));
    }
}
```

Any mapper class should be an extension of the class MapReduceBase and must implement the interface Mapper. The Mapper interface has four arguments: input key, input value, output key, and output value. The input key is an integer offset of the current line from the beginning of the file, and is not relevant to us in the current example. The input value is the text of the line read, so this contains a row of data. Our output key is the year, which can be both text as well as an integer, but for the sake of diversity, we will treat it as a string here. Finally, our output value is a number, which is actually the randomly generated count against a year.
There is another important point to observe in this class. Instead of using Java's primitive data types, Hadoop uses its own data types that are suited optimally to network serialization. For String, the equivalent in these data types is Text; for int, it is IntWritable; for long, it is LongWritable etc.

2.1.4 *Reduce* Function

After executing the mappers, Hadoop collects the data from all the map outputs, distributes them according to the key to a number of reducers. For each key, all the values are returned in a single iterator, which is handed to a reducer. In our case, each reducer will have a year (Text) as an input key, an iterator of all the counts (IntWritable) for that year, the year (Text) again as the output key, and a text specifying the count and the average (Text) as the output value.

```
public class MyReducer extends MapReduceBase
  implements Reducer<Text, IntWritable, Text, Text> {

    public void reduce(Text key, Iterator<IntWritable> values,
      OutputCollector<Text, Text> output,
      Reporter reporter) throws IOException {

      int sum = 0, recordCount = 0;
      double average = 0;

      //Take the sum of all counts of a single year
      while (values.hasNext()) {
        sum += values.next().get();
        recordCount++;
      }

      //Take the average
      average = sum/recordCount;

      //Output the sum and the average as a comprehensible string
      output.collect(key,
        new Text("OccurenceCount:"+recordCount+"
          Average:"+average));
    }

}
```

Just like the mapper class, any reducer class must be an extension of MapReduceBase and must implement the Reducer interface, which is similar in structure as the Mapper interface.

2.1.5 Executing the Program

Now that we have defined the map and reduce functions, all that remains is to configure how we want the job to be run. For this, we use the JobConf class which allows to setup parameters for the job.

```
public static void main(String[] args) throws IOException {

  JobConf conf = new JobConf(Main.class);
  conf.setJobName("Average count");

  /* Specify the input and output paths
  * These paths are in HDFS and not in user's local dir
  */
  FileInputFormat.addInputPath(conf, new Path(args[0]));
  FileOutputFormat.setOutputPath(conf, new Path(args[1]));

  //Configure mapper and reducer classes
  conf.setMapperClass(MyMapper.class);
  conf.setReducerClass(MyReducer.class);

  //Set the types for input and output keys and values
  conf.setMapOutputKeyClass(Text.class);
  conf.setMapOutputValueClass(IntWritable.class);
  conf.setOutputKeyClass(Text.class);
  conf.setOutputValueClass(Text.class);

  //Run the job
  JobClient.runJob(conf);
}
```

This function takes two parameters, input and output paths. If the input path is a file, then that file will be used as the input, and in case of a directory, all the files will be treated as input. We specify these paths in the JobConf object along with the input and output types of keys and values of our map and reduce classes.
To execute the program, we compile the folder, note the path to its jar file, and run it as follows.

```
$ ./hadoop jar
 /path/to/your/MapReduceProgram.jar number_counts output
```

By looking at some rows from the output file, we see that the function behaves quite uniformly, and the results would have been even more precise had we generated more than 5 million records.

```
1628 OccurenceCount:4980 Average:247.0
1629 OccurenceCount:4863 Average:252.0
1630 OccurenceCount:5038 Average:251.0
1631 OccurenceCount:5065 Average:250.0
1632 OccurenceCount:5004 Average:249.0
1633 OccurenceCount:4956 Average:250.0
```

3 Distributed Model Checking

3.1 Motivation

Consider a model of a system that has n variables, each of which can have two possible values. We can see that the system itself can be in one of 2^n possible states at any given time. If our variables can have more than 2 values, say k, then the number of possible states becomes k^n. In general, a system of concurrent processes, each with a certain number of variables, can reach a number of states that is exponential in size to the number of processes and variables. This is called the *state space explosion* problem [9], and is one of the challenges of model checking. As the number of variables and processes increase in a system, the memory required in order to distinguish between explored and unexplored states increases exponentially.

Various techniques have been developed over the years to deal with the state space explosion problem. For example, *abstraction* reduces the number of states by creating a simpler model that preserves all the interesting properties of the original model, *state compression* [10] attempts to reduce the size of each state so that more states can be stored etc. The technique that is used in this report is *Distributed Model Checking* and the idea is that during the state space generation, the states can be stored on a distributed memory, and even multiple processing units can be used to tackle the state space explosion problem.

3.2 DiVinE

DiVinE (Distributed Verification Environment) [3, 4] is an open-source model checker which specializes in distributed model checking. DiVinE uses its own language called DVE for modelling systems, but can also be used with PROMELA models. [11] DiVinE is basically composed of two modules, a lower level DiVinE LIBRARY and a higher layer built using the library called the DiVinE TOOL. The library provides basic operations on a model, e.g. obtaining the initial state, getting the successors of a state, storing a state, checking membership of a state, etc.

4 Breadth First Search using Hadoop

In this section, an algorithm for generating state space using the Breadth First Search (BFS) algorithm with Hadoop is described. This algorithm was designed by Keijo Heljanko and Mark Sevalnev. Before we move on, let us look first at how the state space with BFS can be generated.

4.1 Simple BFS

Procedure 1 illustrates a simple BFS algorithm that can be used to generate the state space for any given model M. At first, the queue is initialized with just the initial state. Then, the successors of each unseen state are enqueued, and this is repeated until the queue is empty. It can be sees that only two operations *GetInitialState* and *GetSuccessors* are required for the model in order to run this algorithm successfully. Note that the only concern is generating the complete state space, and not exactly checking properties for each state at the moment.

Procedure 1 GenerateStateSpaceBFS(M)

```
1:  s_init ← GetInitialState(M)
2:  S ← ∅
3:  Q ← {s_init} {Q is a FIFO structure}
4:  while Q ≠ ∅ do
5:     s ← dequeue(Q)
6:     if s ∉ S then
7:        S ← S ∪ {s}
8:        S_succ ← GetSuccessors(S)
9:        for each s_succ ∈ S_succ do
10:          Q ← Q ∪ s_succ
11:       end for
12:    end if
13: end while
```

Now that the simple BFS algorithm for generating the state space has been described, our next step is to parallelize the storage and computation somehow to alleviate the state space explosion problem. That is done using MapReduce, discussed in the earlier section, because of its efficient implementation and ease of use.

4.2 Some Notation

To keep the pseudo-code brief and comprehensible, we introduce some notation here that will be used in the algorithms.

- A state is represented in terms of its two properties, its name and another property that we shall call the *status* of the state. The status of the state is represented by two constants which we shall refer to as $\mathbf{s_s}$ and $\mathbf{s_f}$. The status basically is a part of the algorithm, and it helps us to keep track of which states have been seen by the algorithm,

and which have not been explored yet. Note that this status is not a property associated with each state in the original model. The original model, of course, just consists of the names and relationships of states. So, a single state name may have different statuses in different sets.

In the pseudo-code, we represent a state as the tuple $(name, status)$ where $status \in \{\mathbf{s_s}, \mathbf{s_f}\}$. In the actual implementation, it is represented by the name and s or f (for *seen* and *frontier* respectively) separated by a tab. We will use that convention where we reproduce any actual code used in the implementation.

- Given a state i, we can refer to its status as $status(i)$ and to its name as $name(i)$.

4.3 Map and Reduce Functions

Before we look at the main algorithm in Section 4.4, it is important to first discuss all the map, reduce, and combine functions that the algorithm uses. The algorithm basically runs in iterations, and in each iteration, there are two MapReduce computations. Let us call them Phase 1 computations and Phase 2 computations. We will denote our functions with the same convention using the numbers 1 and 2, e.g. map1 and map2 refer to the map functions of first and second phase respectively. Right now, it is not important to see how they will fit in the algorithm, rather, it is simpler to just discuss them in isolation, and then when we move on to the main algorithm, see how they can be used.

4.3.1 Phase 1

Procedure 2 describes the map function of Phase 1. It is represented in pseudo-code since we have implemented this as a shell script and not a Java class. This can be done using Hadoop streaming library. This is necessary since DiVinE LIBRARY can only be used with C/C++ right now.

Procedure 2 $map1(I)$

```
1: for each i ∈ I do
2:    if status(i) = s_s then
3:       output i
4:    else
5:       status(i) = s_s
6:       output i
7:       for each c ∈ GetSuccessors(name(i)) do
8:          output (c, s_f)
9:       end for
10:   end if
11: end for
```

Procedure 2 takes a set of states as an input. It first checks if the status of the state is seen, it simply outputs the state. Otherwise, if the status of the state is frontier, it first changes it to seen, then outputs it, and then outputs all the children of the state with the status frontier. Recall that the output from map functions is fed to the combiner (locally) functions first, and the output from the combiner functions is fed to the reducer function. The combiner function for Phase 1 computation is given below.

Procedure 3 $combine1(key, I)$

```
1: for each i ∈ I do
2:    if status(i) = s_s then
3:       output (key, s_s)
4:       return
5:    end if
6: end for
7: output (key, s_f)
```

The actual combine1 function in Java is given in Appendix A. The above combine function removes duplicates of states on each node where map is run. Like mentioned earlier, it is an intermediate function between map and reduce and the purpose is to handle different values of a single key on the same machine where map is run, in order to reduce the co-ordination between nodes of a network. Here, the function takes as an input key/value pairs which are nothing but state $(name, status)$. For a given state, if there is any status which is equal to $\mathbf{s_s}$, it disregards all other values and outputs the key/value pair $(key, \mathbf{s_s})$. Otherwise, in case it finds no status equal to $\mathbf{s_s}$,

it outputs $(key, \mathbf{s_f})$. Note that this output will be fed as key/value pairs to the reducer function, which is given below.

Procedure 4 $reduce1(key, I)$

```
1: for each i ∈ I do
2:    if status(i) = s_s then
3:       return
4:    end if
5: end for
6: output (key, s_s)
```

The actual reduce1 function in Java is given in Appendix A. The reduce function above filters all the states which have even one $\mathbf{s_s}$ in their set of values. This is the final output of the whole Phase 1 computation and this is refered to as $MapReduce1(I)$ in the pseudo-code.

4.3.2 Phase 2

Phase 2 computation is rather simple compared to the Phase 1 computation. There is only one map function here and there are no combiner and reducer for this phase. The map function is given below.

Procedure 5 $map2(i)$

```
1: if status(i) = s_s then
2:    output i
3: else
4:    output (name(i), s_s) {if status(i) = s_f}
5: end if
```

The actual map2 function in Java is given in Appendix A. The map function above simply traverses through all the states in the input, converts their statuses to $\mathbf{s_s}$ and outputs them. This output is the final output of the Phase 2 computation and will be refered to as $MapReduce2(I)$ in the pseudo-code. The purpose of this function may not be very obvious at this stage, but it will become clearer as we see the main algorithm next, and an example to demonstrate the whole algorithm after that.

4.4 State Space Exploration Using Hadoop Algorithm

Procedure 6 describes our main algorithm for exploring state space of a model. It is actually implemented as a script.

Procedure 6 *ExploreStateSpace*(M)

1: $s_{init} \leftarrow GetInitialState(M)$
2: $T \leftarrow \{(s_{init}, \mathbf{s_f})\}$
3: $S \leftarrow \{(s_{init}, \mathbf{s_s})\}$
4: **loop**
5: $\quad T \leftarrow MapReduce1(T \cup S)$
6: $\quad$ **if** $|T| = 0$ **then**
7: $\quad\quad$ **break**
8: $\quad$ **end if**
9: $\quad S \leftarrow MapReduce2(T \cup S)$
10: **end loop**

Let us examine the algorithm closely and see how it works. The function takes a model as an input. In the actual implementation, this models can be any valid model expressed in DVE. Throughout the algorithm, two sets T and S are stored. The first set T stores all the states that are discovered by the algorithm and whose children are yet to be explored, and S stores all the states which have been discovered by the algorithm.

First, the initial state of the model is obtained and both S and T are initialized with it with the statuses $\mathbf{s_s}$ and $\mathbf{s_f}$ respectively since the initial state has already been seen and its children are yet to be explored. Next, there is a loop that goes on until there are no more states whose children are yet to be discovered, in other words, until T is empty. As soon as the algorithm enters the loop, the output of $MapReduce1$ computation with the input as the union of S and T is assigned to T. Now the emptiness of T is checked the program is terminated if it is empty. Otherwise, the output of $MapReduce2$ (with the union of S and T as the input) is assigned to S and the execution goes back to the start of the loop.

4.5 Example

In this section, we demonstrate the flow of the algorithm described above with the help of an example. For the sake of simplicity, let us assume that the algorithm is run on a single machine, this assumption only makes the analysis simpler in the sense that case analysis at *combine*1 is not taken into consideration. Otherwise, at each *combine*1, we will have to consider all different combinations for different *map*1 running on different machines, and the example will become quite complicated.

Consider the model M in figure 2 where each node of the graph represents a state in the model. Looking at the graph, we can get the following values.

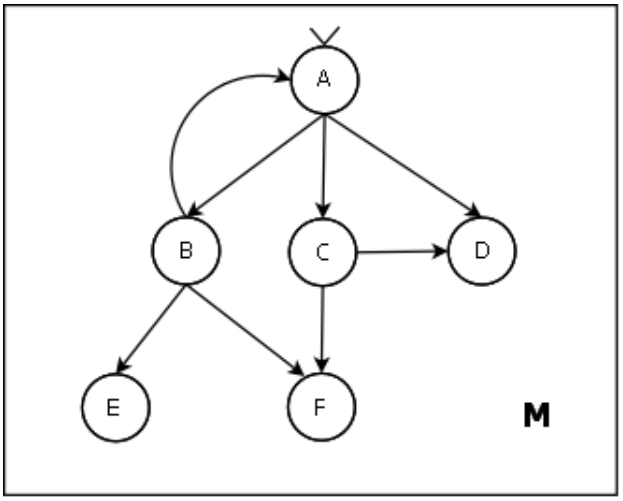

Figure 2: An example model M represented as a graph

$GetInitialState(M) = \{A\}$
$GetSuccessors(A) = \{B, C, D\}$
$GetSuccessors(B) = \{A, E, F\}$
$GetSuccessors(C) = \{F, D\}$
$GetSuccessors(D) = \emptyset$
$GetSuccessors(E) = \emptyset$
$GetSuccessors(F) = \emptyset$

Let us run the Procedure 6 on this model now. After executing steps 1-3, $T = \{(A, \mathbf{s_f})\}$ and $S = \{(A, \mathbf{s_s})\}$. Now we enter the loop, and here we will see what happens in each iteration.

1. (a) $map1(T \cup S)$ will give the following states:
   $(A, \mathbf{s_s})$
   $(B, \mathbf{s_f})$
   $(C, \mathbf{s_f})$
   $(D, \mathbf{s_f})$
   $(A, \mathbf{s_s})$

   (b) The above output from $map1$ will be fed to $combine1$ which will give the following states:
   $(A, \mathbf{s_s})$
   $(B, \mathbf{s_f})$
   $(C, \mathbf{s_f})$
   $(D, \mathbf{s_f})$

(c) The output from *combine*1 will act as an input to *reduce*1 which will give the following states:
$(B, \mathbf{s_f})$
$(C, \mathbf{s_f})$
$(D, \mathbf{s_f})$

(d) The result from above is the output of $MapReduce1(T \cup S)$ and now $T = \{(B, \mathbf{s_f}), (C, \mathbf{s_f}), (D, \mathbf{s_f})\}$.

(e) Since $|T| \neq 0$, we continue.

(f) $map2(T \cup S)$ is called and it returns the following output:
$(B, \mathbf{s_s})$
$(C, \mathbf{s_s})$
$(D, \mathbf{s_s})$
$(A, \mathbf{s_s})$

(g) Since there is no combine and reduce function, the above states are the output of $MapReduce2(T \cup S)$. Therefore,
$S = \{(A, \mathbf{s_s}), (B, \mathbf{s_s}), (C, \mathbf{s_s}), (D, \mathbf{s_s})\}$.

2. (a) $map1(T \cup S)$ will give the following states:
$(B, \mathbf{s_s})$
$(A, \mathbf{s_f})$
$(E, \mathbf{s_f})$
$(F, \mathbf{s_f})$
$(C, \mathbf{s_s})$
$(D, \mathbf{s_f})$
$(F, \mathbf{s_f})$
$(D, \mathbf{s_s})$
$(A, \mathbf{s_s})$
$(B, \mathbf{s_s})$
$(C, \mathbf{s_s})$
$(D, \mathbf{s_s})$

(b) The above output from *map*1 will be fed to *combine*1 which will give the following states:
$(B, \mathbf{s_s})$
$(A, \mathbf{s_s})$
$(E, \mathbf{s_f})$
$(F, \mathbf{s_f})$
$(C, \mathbf{s_s})$
$(D, \mathbf{s_s})$

(c) The output from *combine*1 will act as an input to *reduce*1 which will give the following states:

$(E, \mathbf{s_f})$
$(F, \mathbf{s_f})$

(d) The result from above is the output of $MapReduce1(T \cup S)$ and now $T = \{(E, \mathbf{s_f}), (F, \mathbf{s_f})\}$.

(e) Since $|T| \neq 0$, we continue.

(f) $map2(T \cup S)$ is called and it returns the following output:
$(E, \mathbf{s_s})$
$(F, \mathbf{s_s})$
$(A, \mathbf{s_s})$
$(B, \mathbf{s_s})$
$(C, \mathbf{s_s})$
$(D, \mathbf{s_s})$

(g) Since there is no combine and reduce function, the above states are the output of $MapReduce2(T \cup S)$. Therefore,
$S = \{(A, \mathbf{s_s}), (B, \mathbf{s_s}), (C, \mathbf{s_s}), (D, \mathbf{s_s}), (E, \mathbf{s_s}), (F, \mathbf{s_s})\}$.

3. (a) $map1(T \cup S)$ will give the following states:
$(E, \mathbf{s_s})$
$(F, \mathbf{s_s})$
$(A, \mathbf{s_s})$
$(B, \mathbf{s_s})$
$(C, \mathbf{s_s})$
$(D, \mathbf{s_s})$
$(E, \mathbf{s_s})$
$(F, \mathbf{s_s})$

(b) The above output from $map1$ will be fed to $combine1$ which will give the following states:
$(E, \mathbf{s_s})$
$(F, \mathbf{s_s})$
$(A, \mathbf{s_s})$
$(B, \mathbf{s_s})$
$(C, \mathbf{s_s})$
$(D, \mathbf{s_s})$

(c) The output from $combine1$ will act as an input to $reduce1$ which will give the following states:
$\emptyset$

(d) The result from above is the output of $MapReduce1(T \cup S)$ and now $T = \emptyset$.

(e) Since $|T| = 0$, we terminate the program.

5 Experiments and Results

5.1 Setup and Configuration

We tested the implementation of the algorithm described above on Pythia, which is a computing cluster at the ICS department. 16 compute nodes, each having two 2.0GHz Intel Xeon 5130 dualcore processors and running Rocks Linux were used. The DiVinE model chosen for the experiment is adding.3.dve. The number of mappers and reducers for phase 1 was not specified, which means that Hadoop used the default value. The default number of mappers depend on the size of the input such that there is one mapper for one HDFS block. For phase 2, the number of mappers per node was 4, and the number of reducers was 0 since there is no reduce function for phase 2. Since DiVinE library is only accessible in C/C++, Hadoop streaming was used to run the implementation of map function of phase 1. The timings obtained by running the implementation are given below.

5.2 Running time

5.2.1 Initial time

Since Hadoop leaves a lot of files on the system where it is run, all the files required for its setup were extracted to a single folder which would be deleted after the execution. The extraction time of Hadoop is not more than half a minute in our experiments. After extraction, the time required to get Hadoop started is about a minute on average. Though the starting time for Hadoop is significant, it becomes relatively insignificant if the model we are interested in exploring has a big state space, since that will require many iterations to be completely explored.

5.2.2 BFS Iterations

Figure 3 shows the total time of Algorithm 6 after the initial steps, for the first 50 iterations. With each iteration, as we go deeper into the search, the number of states increase, and as a result, the time increases.

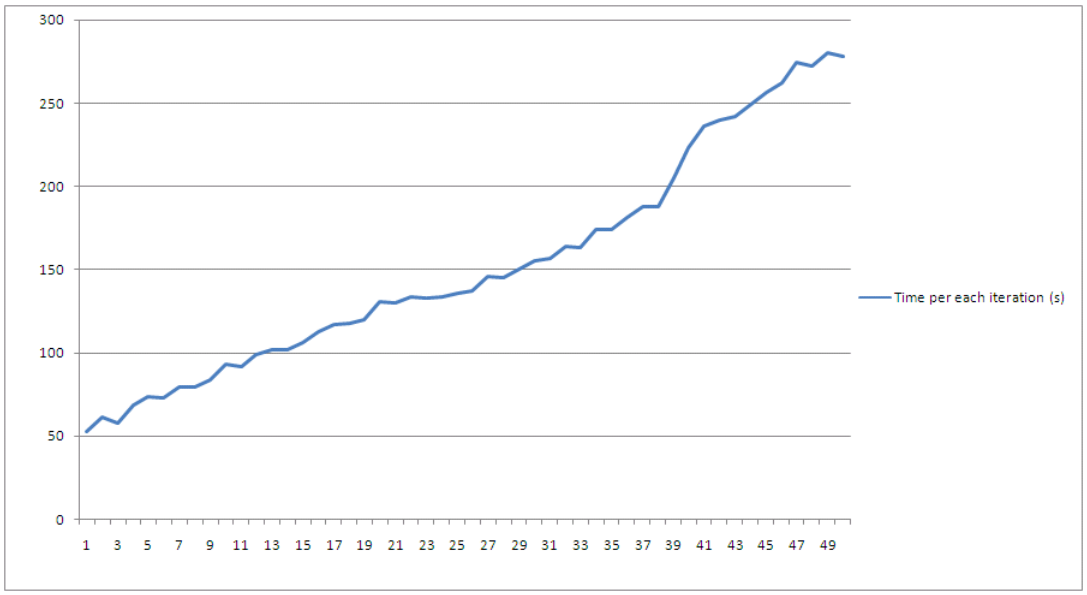

Figure 3: Time taken in the first 50 iterations in seconds

In each iteration, we have broken down the total time into three stages, since these stages are the only ones that take non-negligible time. Figure 4 shows these timings. The first two are the timings of Phase1 and Phase2 of the algorithm. The third stage is copying which consists of copying the output of phase 2 of current iteration to the input of the next one. Out of these three stages, this stage takes mostly the smallest time, as can be seen from the graph.

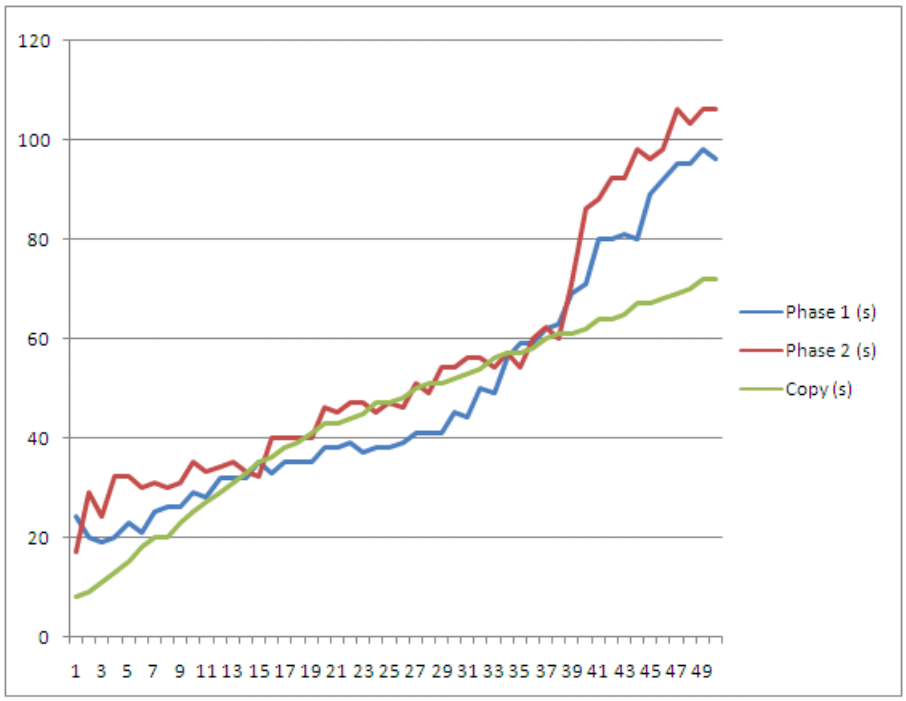

Figure 4: Time break down of each iteration in seconds

6 Conclusion

We described a Breadth First Search algorithm for exploring state space of a model using DiVinE model checker in combination with Hadoop in the report. The timings for one DiVinE model were also recorded and plotted for each phase in the algorithm.

There can be several extensions for the future work of the approach described in this report. The performance of the presented algorithm can be compared and evaluated against other benchmarks in distributed model checking, especially DiVinE's own distributed model checking techniques. Another possible extension could be studying the timing in each phase of MapReduce, and investigating whether some optimization could be derived based on the observations regarding the breakdown of timings.

A Java code for different functions

```
public void combine1(Text key,
  Iterator<Text> values, OutputCollector<Text, Text> output,
  Reporter reporter) throws IOException {

  while (values.hasNext()) {
    if (values.next().toString().equals("s")) {
      output.collect(key, "s");
      return;
    }
  }
  output.collect(key, "f");
}
```

```
public void reduce1(Text key,
  Iterator<Text> values, OutputCollector<Text, Text> output,
  Reporter reporter) throws IOException {

  while (values.hasNext()) {
    if (values.next().toString().equals("s")) {
      return;
    }
  }
  output.collect(key, "f");
}
```

```
public void map2(LongWritable key, Text value,
  OutputCollector<Text, IntWritable> output,
  Reporter reporter) throws IOException {

  String word = value.toString();

  if (word.charAt(word.length() - 2) == 's')
    output.collect(value, null);
  else { // if (word.charAt(word.length() - 1) == 'f')
    word =
      new String(word.substring(0, word.length() - 1) + "s");
    output.collect(new Text(word), null);
  }
}
```

References

[1] I. Burnstein, *Practical Software Testing: A Process-oriented Approach*. New York, NY, USA: Springer Inc., 2003.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, Massachusetts: The MIT Press, 1999.

[3] J. Barnat, L. Brim, I. Černá, and P. Šimeček, "DiVinE – The Distributed Verification Environment," in *Proceedings of 4th International Workshop on Parallel and Distributed Methods in verifiCation* (M. Leucker and J. van de Pol, eds.), pp. 89–94, July 2005.

[4] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček, "DiVinE – A Tool for Distributed Verification (Tool Paper)," in *Computer Aided Verification*, vol. 4144/2006 of *LNCS*, pp. 278–281, Springer Berlin / Heidelberg, 2006.

[5] "DiVine." `http://divine.fi.muni.cz/page.php`.

[6] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *CACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] "DiVine." `http://hadoop.apache.org/`.

[8] "Running Hadoop on Ubuntu Linux by Michael G. Noll." `http://www.michael-noll.com/wiki/Running_Hadoop_On_Ubuntu_Linux_(Single-Node_Cluster)`.

[9] A. Valmari, "The state explosion problem," *Lecture Notes in Computer Science*, vol. 1491, pp. 429–528, 1998.

[10] G. J. Holzmann, *The Spin Model Checker, Primer and Reference Manual*. Reading, Massachusetts: Addison-Wesley, 2003.

[11] J. Barnat, V. Forejt, M. Leucker, and M. Weber, "DivSPIN – A SPIN compatible distributed model checker," in *Proceedings of 4th International Workshop on Parallel and Distributed Methods in verifiCation* (M. Leucker and J. van de Pol, eds.), pp. 95–100, July 2005.