International Journal of Applied Engineering Research
ISSN 0973-4562 Volume 10, Number 11 (2015) pp. 27513-27524
© Research India Publications
http://www.ripublication.com

Comparison of Checkpointed Aided Parallel Execution Against Mapreduce

Nisha Rani N, Shiju Sathyadevan
Amrita Center For Cyber Security Systems & Network, Amrita University, Kollam, India nisharani9028@gmail.com, shiju.s@am.amrita.edu
Eric Renault, Viet Ha Hai
Institute telecom – Telecom Sudparis, Evry, France Franceeric.renault@telecom-sudparis.eu,
haviethaivn@yahoo.com

Abstract

Researchers have been actively working for the past few decades in parallelizing programs so as to cut through massive data chunks for faster response. Current day processors are faster and have more number of cores. So as to utilize the computational capabilities of the processors to its full extend, processes need to be run in parallel. A task can be performed in lesser time by using parallel programming. But writing a parallel programming manually is a difficult and time consuming task. So we have to use tools to convert a sequential program to a parallel one automatically. Open-MP (Open Multi-Processing) is a set of directives which can be used to generate parallel programs written in c, c++, FORTRAN to efficient parallel programs. A new paradigm called CAPE (Check-pointing Aided Parallel Execution) is introduced that uses check-pointing technique to generate parallel programs from sequential programs provided with Open-MP directives. Map-reduce is a programming model for performing parallel processing. In this paper we have compared the performance and coding complexity of map-reduce against CAPE under different levels of difficulties.

Keywords: CAPE; Check pointing Aided Parallel Execution; Hadoop; Single Instruction Multiple Data; SIMD

Introduction

Several research initiatives are active in finding new means for improving the performance of processors for high throughput, faster processing. There are several advantages for parallel programs over sequential programs. In sequential programming, the processes execute in a sequential order one after the other. But in parallel programming, we have multiple processes and threads that execute simultaneously at the same time. In this paper we are looking into the capabilities of a new platform CAPE (Check pointing Aided Parallel Execution) for converting sequential program to parallel program thus improving the overall efficiency. CAPE is an automatic paradigm for doing this conversion. CAPE is capable of converting simple as well as complex sequential programs to efficient parallel programs with ease achieving better performance against similar processing frameworks. CAPE is able to achieve this conversion with the help of Open MP directives [3].

Section 2 provides an overview about CAPE, its architecture, working and general algorithm that is being considered in this paper. In section 3, there is a brief description about Hadoop and where its architecture and how it works is discussed. Section 4 details SIMD architecture and how it is used in CAPE and. This section also explains how both CAPE and uses SIMD architecture. Section 5 does the performance evaluation of CAPE and. We monitor the behavior of parallel programs while running in CAPE and in. We evaluate CAPE and on the basis of different criteria: no of lines of code, complexity in coding, and execution time. Section 6 explains the advantages of using CAPE over based on the above criteria. Section 7 and section 8 deals with conclusion and future work respectively.

Related Works

There are many works related to parallel computing since it is necessary to utilize the capability of fast processors and to obtain high throughput. Viet Hai Ha and Eric Renault has proposed checkpoint for converting sequential program to parallel program [1], [2], [4]. Their works deal with different check pointing approaches that can be engaged in automatic conversion of sequential programs to run in parallel mode. The approaches are:

- Continuous check pointing: All the information is considered from the beginning of execution of a program. Advantage of this approach is its simplicity. Disadvantage is that the parts that are not modified are also examined.
- Discontinues check pointing: Modified part since the beginning of a program is considered. Advantage is the size of the checkpoint. Drawback is the decrease in efficiency due to continuous monitoring of process memory.
- Incremental discontinuous check pointing: Information regarding where to add the checkpoints is considered. The drawbacks of above approaches are eliminated in this approach.

CAPE

CAPE stands for Check pointing Aided Parallel Execution. It transforms a sequential program to a distributed program. A sequential program is split into different segments and these segments are distributed over a set of machines called nodes to execute in parallel. The parts of the original code that have to be executed in parallel are identified by programmers while using Open MP pragma directives. CAPE is able to work with the Open MP directives equipped with the programs written in c, c++ programming languages. CAPE is language independent and hence can be extended to any other programming languages.

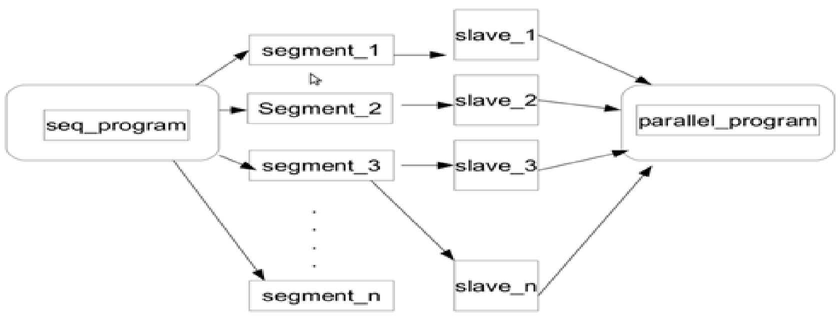

Figure 1: Architecture of CAPE

Check pointing is a technique for inserting fault tolerance into computing systems [11]. It stores a snapshot of the current application state and uses for resuming the execution in case of failure. It is the backbone for rollback recovery, playback debugging, process migration, job swapping, load balancing, periodic backup, and many other functions. The main attraction is that, they can also be used in order to transform a sequential program to a parallel program automatically. CAPE use check pointing to transform a sequential program to a parallel or distributed program.

CAPE is based on Master-Slave Architecture model. The sequential program resides in the master node as shown in Figure1. Master node splits the program into different segments and distributes it into a set of slave nodes. Slave nodes execute the respective segments and produce pre-output results. The pre-outputs are then fed back to the master node. Master node combines these pre-outputs from the slave nodes and produces the target result. Let us take the example of matrix multiplication. The sequential matrix multiplication program first resides in the master node. Master node adds check points at different locations. It adds check-points with the help of pragma directives. By this a parallel executing matrix program is generated. This program is divided into different segments and given to the slave nodes. Slave nodes execute the segments they got and send back the results to the master node. Master node merges the results and produces the final result of matrix multiplication. CAPE work on the basis of a set of primitives as depicted in Figure 2.

```
#pragma omp (i, j, k)
for ( i ; j ; k )
code segment
```

automatically converted to

↓

```
1) if ( master ( ) )
2) start ( )
```

```
3) for ( i ; j ; k )
4) create ( original )
5) send ( original, slave_n )
6) stop ( )
7) wait for ( target )
8) inject ( targeti )
9) if ( ! last parallel ( ) )
10) merge ( final, targeti )
11) broadcast ( final )
12) else
13) receive ( targeti )
14) inject ( targeti)
15) start ( )
16) else
17) create ( targeti )
18) stop ( )
19) send ( targeti , master )
```

Figure 2: Check pointing Enabled General algorithm for Open MP directives

Line 1 & 2	:	The master node starts the checkpointing.
3 to 7	:	Master node creates checkpoints and sends the same to a set of slave node. It stops checkpointing and wait for the result from slave nodes.
8	:	Inject the result from slave nodes to a final file.
9 & 10	:	If the result received by the master node is the last output from the slave node, then it merge all the result and form the final target parallel program.
11	:	Master node broadcast the final result to the slave nodes.
12 to 15	:	If the received result from slave node is not the final one then master node continue step 8
16 & 17	:	Each slave node creates checkpoints and process the segment which they get from the master node.
18 &19	:	After processing they stop checkpointing and send their respective results to the master node.

Hadoop

Hadoop is a radical approach to the problem of distributed computing [6], [12]. It is actually a software platform that lets one to write and execute applications that process huge amounts of data. Hadoop is also designed as master-slave architecture. In Hadoop the master node is called name node and slave nodes are called data nodes. Hadoop consists of two core components: Hadoop Distributed File System and. HDFS is responsible for storing data in a cluster. It consists of two phases: map and reduce. Map perform the mapping of input data given and reduce function reduce the intermediate output and produce the reduced final output.

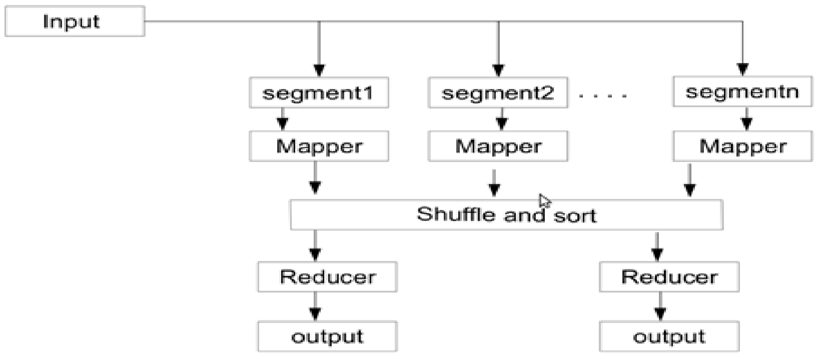

Figure 3: Architecture of Map Reduce

The main steps as shown in Figure 3 are:

1. Data in the name node is split into blocks and distributed across multiple data nodes in the cluster.
2. These blocks are given to the map function in each data nodes. Each Map task operates on a single block of data.
3. Then is passed to shuffle and sort function which sorts the intermediate data from all map functions.
4. This result is passed to the reducers and they operate on these intermediate results and produce the final output file.
5. The output file is stored in HDFS.

SIMD Concepts and Implementation In CAPE and Mapreduce

Single Instruction Multiple Data (SIMD) is a type of parallel execution architectural model [7]. It is classified under Flynn's taxonomy. This consist of a set of processors each having its own local memory. Each processor executes same set of instructions simultaneously. Each of them works on different pieces of data which are stored in their local memory. Both CAPE and are based on SIMD architecture. In SIMD model same instruction is passed to all parallel executing nodes and each node work on different data sets. In CAPE same instruction is passed to the slave nodes but each slave node perform their task on different segments. In same instruction is carried out in map function but each data node is operating on different sets of data. In reduction function also the instruction is same but the data nodes are working on different data sets.

Evaluation between CAPE and Map Reduce

In this section we identified three tasks that range in complexity from medium to semi-complex problems: 'matrix multiplication', 'K-Means' algorithm and word count. Matrix multiplication and K-Means were used to compare the coding complexities where as matrix multiplication and word count were used to compare the overall performance in CAPE and environments. K-Means was excluded from performance comparison.

Size of the Program in CAPE and Map Reduce

Lines of the code, in the case of CAPE program was very much less than that of program. This is evident from the code shown under Fig.4 and Fig.5 associated with matrix multiplication program and the code snippets in Fig.6 and Fig.7 associated with K-Means. In Map Reduce there was separate driver class section for matrix multiplication and for K-Means programs. Similarly separate map and reduce section for these programs. But in CAPE, for implementing a new program it was much simpler. It expected to just specify the program name, its path and number of nodes in already built program block. Another major difference in CAPE was that there were no separate map/reducer sections.

Complexity of the Program in CAPE and Map Reduce

In CAPE, was up to the programmer to identify how & where to partition the given problem into mapping and reducing module. The programmers need to separate the problem into map and reduce block. But in CAPE programmers need not be worried about the same. CAPE expects developers to push the new code segment into an already built program block. In short, the programmer need to set driver class, identify and code the map and reduce sections etc in case of whereas in CAPE there is no need to develop map and reduce functionalities as the complier with the help of check-points will detect where to map and reduce.

```
 folder=/home/abc
 prog=matrix
 num_nodes=2
//Parallel block(i,j,k)
for (i = 0 ; i < m ; i++ )
for ( j = 0 ; j < q ; j++ )
for ( k = 0 ; k < p ; k++ )
sum = sum + first[i][k]*second[k][j];
```

Figure 4: pseudo code of CAPE program for matrix multiplication

---

Driver class section begins:

```
public class matrix {
public static void main(String[] args)throws Exception{
Job job = new Job();
```

```
                    job.setJarByClass(matrix.class);
               job.setJobName("matix multiplication");
               FileInputFormat.setInputPaths(job,new
              Path(args[0]));
               FileOutputFormat.setOutputPath(job,new
              Path(args[1]));
               job.setMapperClassMapper.class);
               job.setReducerClass(Reducer.class);
               job.setOutputKeyClass(Text.class);
               job.setOutputValueClass(IntWritable.class);
               boolean success = job.waitForCompletion(true);
               System.exit(success ? 0 : 1);
                  } }
```

Driver class section ends

Map function begins:

```
if (indicesAndValue[0].equals("A"))
for (int k = 0; k < p; k++)
outputKey.set(indicesAndValue[1] + "," + k);
outputValue.set("A," + indicesAndValue[2] + "," + indicesAndValue[3]);
```

Map function ends:

Reduce function begins:

```
for (int j = 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;
```

Reduce function ends:

Figure 5: pseudo code of program for matrix multiplication

```
                          folder=/home/abc
                                prog=kmeans
                                num_nodes=2
                             for(;;)
                 cenLx=newcentral[i][0]-oldcentral[i][0];
cenLy=newcentral[i][1]-oldcentral[i][1];
cenL=sqrt(cenLx*cenLx+cenLy*cenLy);
```

Figure 6: Pseudo code of CAPE program for k-means algorithm

Driver class begins:

```
                         public class matrix {
 public static void main(String[] args)throws Exception{
                                                              Job job = new Job();
                                                               job.setJarByClass(kmeans.class);
                                                               job.setJobName("kmeans clustering");
             fileInputFormat.setInputPaths(job, new Path(args[0]));
 fileOutputFormat.setOutputPath(job,new path(args[1]));
                                      job.setMapperClassMapper.class);
                                                              job.setReducerClass(Reducer.class);
                                                   job.setOutputKeyClass(Text.class);
                                                  job.setOutputValueClass(IntWritable.class);
                                                              boolean success = job.waitForCompletion(true);
                                                              System.exit(success ? 0 : 1);
                                                              } }
```

Driver class ends:

Map function begins:

```
 while ((line = readCentroids.readLine()) != null) {
 StringTokenizer centroidTokenizer =newStringTokenizer    (line,delim);
                                                              numAttributes                              =
centroidTokenizer.countTokens();
                                                              double[] point = new double[numAttributes];
                                                              for (int I = 0; I < numAttributes; i++) {
                                                                          String              token              =
centroidTokenizer.nextToken();
                                                                          point[i] = Double.parseDouble(token);
                                                              }
                                                              centroids.add(point);}
```

Map function ends:

Reduce function begins:

```
for (int i = 0; i < numAttributes; i++)
sum[i] += Double.parseDouble(temp[i].toString());
for (int i = 0; i < numAttributes; i++)
             average[i] = sum[i] / length;
 for (int i = 1; i <= numAttributes; i++)
                                                               csv += d[i - 1];
 if (i == numAttributes)
                                                   csv += "\n";
 else
                                                   csv += delim;
```

Reduce function ends:

Figure 7: pseudo code of program for k-means program

Speed of the Program in CAPE and Map Reduce
Test environment consist of three virtual machines where in, one of them was configured as the master node while the remaining two were configured as slave nodes. CAPE program executed much faster than and hence CAPE could cut through large volume data sets in a much shorter time. This can be observed by analyzing the Figure 8. and Table 1.

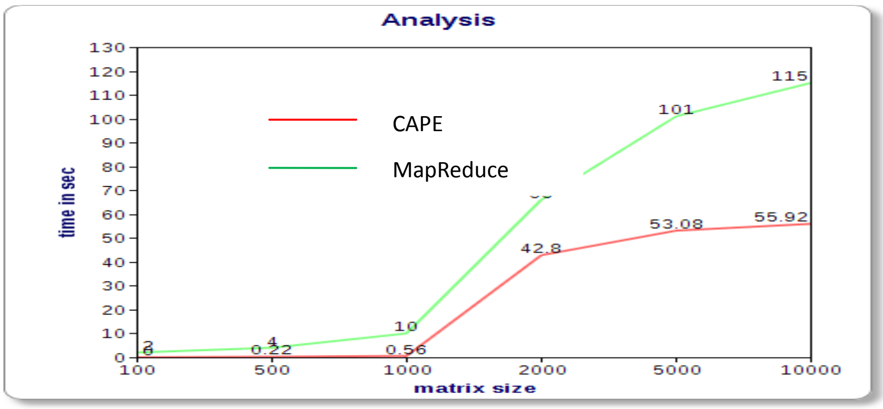

Figure 8: Time of execution (in sec) vs. matrix size

This figure highlights the execution speed for different matrix sizes. It is evident that when the size of matrix is 100 x 100, the execution time for is 2 seconds and that for CAPE is 0 seconds (negligible in value to measure). For a 10000 x 10000 matrix, the execution time for CAPE is 55.92 seconds and that for is 115 seconds. Reason behind this is that does not differentiate between the code segments that can be executed sequentially or in parallel mode. Name node transfers the entire program to the data nodes to execute. But in CAPE the sequential part of a program is executed by the master node itself. After completing sequential part it split the code into segments which can execute in parallel and send to the slave nodes.

Table 1: Execution time in seconds for matrix multiplication

Row size x column size	CAPE (in sec)	Map Reduce (in sec)
100 x 100	0	2
500 x 500	.22	4
1000 x 1000	.56	10
2000 x 2000	42.8	66
5000 x 5000	53.08	101
10000 x 10000	55.92	115

Table 2 compares the execution time for the word count program. In case of word count program CAPE took only 6 seconds to work through an input file of size 200MB and took 14 seconds. A 750 MB file was processed by CAPE in 43 seconds whereas took 122 seconds.

K-Means is excluded from the comparison because CAPE is not able to process larger data sets against the algorithm. Large data sets against K-Means algorithm failed with heap error. Current version of CAPE supports only OpenMP parallel for directive; CAPE can only extract the calculated results in the data memory region and it cannot extract in the heap region. K-means algorithm uses dynamically allocated variables (in the heap), and hence will need to modify CAPE.

Table 2: Execution time in seconds for Word Count

Input size in MB	CAPE (in sec)	Map Reduce (in sec)
100	6	14
200	15	50
500	25	99
750	43	122

Working of Program in CAPE and Map Reduce

In CAPE first the master node split the original program into different blocks. These blocks are assigned to a set of slave nodes. The slave nodes perform their task on the block segment which they received from the master node. Each slave node sends their results back to the master node. Master node merges the results and furnishes the target program and then broadcast to the slave nodes. In short, mapping is done by the slave nodes and reducing function is done by the master node. But in the slave nodes perform both mapping and reducing part. I.e. the results of each slave nodes are merged by the slave nodes and the final or target program is stored in the distributed file system (HDFS).

Advantages of CAPE

1. Performance: CAPE increases speed of a program by converting a sequential program to a distributed program in easy steps with least programming hassles. It does the required conversions automatically using Open MP pragma directives.
2. Flexibility: It allows selecting those sections or segments to be checkpointed in a given program.
3. Size: Size of code is fewer than that of since in CAPE there is no need of map and reduce function for each program.
4. Lesser complex: The programmer need not bother about the complexity in partitioning the problem for separate map and reduce function as encountered in programming.

Conclusion

CAPE is a new but evolving technique in transforming a sequential program to a distributed program. It is capable of doing this conversion automatically so that the programmer need not worry about the underlying complexities. The programmer only needs to push his code segment to an already built CAPE program block. More than 75% of the work is done by installing the CAPE package as it is. With its current capabilities and limitations, CAPE is performing much faster than map-reduce.

Future Works

For CAPE to be widely accepted it need to support majority of the features currently provided by. CAPE does not have the capability to connect itself with Hadoop File System (HDFS) so that large volume data can be read from the file system. This can be done by using libhdfs. so file in HDFS library. CAPE as of now has several restrictions. Large data sets against K-Means algorithm failed with heap error. Current version of CAPE supports only Open MP parallel for directive; it can extract the calculated results in the data memory region and it cannot extract in the heap region. K-means algorithm uses dynamically allocated variables (in the heap), and hence will need to modify CAPE. A team is working in adding this feature to CAPE so that larger data sets can be tested against K-means and other clustering and classification algorithms.

References

[1] Viet Hai Ha & Eric Renault, Discontinuous Incremental: A new approach towards extremely lightweight checkpoints, International Symposium on Computer Networks and Distributed Systems, Tehran : Iran, Islamic Republic, 2011, pp. 1-6

[2] Viet Hai Ha & Eric Renault, Improving Performance of CAPE using Discontinuous Incremental Checkpointing, IEEE International Conference on High Performance Computing and Communications, 2011, pp. 1-6

[3] Viet Hai Ha, Eric Renault, Design and Performance Analysis of CAPE based on Discontinuous Incremental Checkpoints, IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, 2011, pp. 1-6.

[4] Rabi Prasad Padhy, Big Data Processing with Hadoop-MapReduce in Cloud Systems. International Journal of Cloud Computing and Services Science (IJ-CLOSER), 2013, pp. 4-10.

[5] Xuejun Yang, Panfeng Wang, Hongyi Fu, Yunfei Du, Zhiyuan Wang, Compiler-Assisted Application-Level Checkpointing for MPI Programs, The 28th International Conference on Distributed Computing Systems, 2008, pp. 252-256.

[6] Mandeep Kaur, Rajdeep Kaur, A Comparative Analysis of SIMD and MIMD Architectures. International Journal of Advanced Research in Computer Science and Software Engineering, 2013, pp. 1154-1155.