

# Optimized Common Parameter Set Extraction Framework by Multiple Benchmarking Applications on a Big Data Platform

Jongyeop Kim<sup>1,\*</sup>, Abhilash Kancharla<sup>2</sup>, Jongho Seol<sup>2</sup>, Indy N. Park<sup>3</sup>, Nohpill Park<sup>2</sup>

<sup>1</sup>Math and Computer Science, Southern Arkansas University, Magnolia, AR 71753, USA

<sup>2</sup>Computer Science, Oklahoma State University, Stillwater, OK USA

<sup>3</sup>Computer Science, Oklahoma City University, Oklahoma City, OK, USA

## ARTICLE INFO

### Article History

Received 27 June 2018

Accepted 20 September 2018

### Keywords

Big data  
Hadoop  
configuration  
performance tuning

## ABSTRACT

This research proposes the methodology to extract common configuration parameter set by applying multiple benchmarking applications include TeraSort, TestDFSIO, and MrBench on the Hadoop distributed file system. The parameter search space conceptually conducted named  $\Omega(x)$  to hold status of all parameter values and its evaluation results for every stage to eventually reduce benchmarking cost. In the process of determining parameter set for each stage, one parameter and its associated values selected which is reduced system performance in terms of overall execution time difference that are measured by multiple applications on a Hadoop cluster. The experimental results demonstrate the proposed extended greedy manner provide a feasible benchmark model for the multiple MapReduce tasks. This model classified several candidate parameter value sets that can be reduced the overall execution time by 27% of the values against Hadoop default settings. Moreover, we propose e-heuristic greedy with alternative parameter selection model to evaluate second candidate parameter value which will lead global optimum by returning back to the previous stage if local minimum is not found at the current stage compare to the previous ones.

© 2018 The Authors. Published by Atlantis Press SARL.

This is an open access article under the CC BY-NC license (<http://creativecommons.org/licenses/by-nc/4.0/>).

## 1. INTRODUCTION

The Apache Hadoop Distributed File System (HDFS) [1] is one of the prominent engines as a big data processing framework [2] with its distributed processing capabilities over a cluster that composed of multiple nodes [3]. The core technology of this open source is called map and reduce, which is accomplished by appropriately splitting a big task into each node and merging it through inter process communication.

From the hardware perspective, the performance of Hadoop increases in proportion to the number of nodes constituting a cluster and their computing powers of each node. One the other aspect of a given HW environment is that we can consider the performance improvements by adjusting the proper configuration parameters to achieve site specific goals.

In Hadoop, various configuration parameters [4] are firmly related to MapReduce [5] interfacing mechanism among the master node and numerous slaves. Finding the site specific set of parameters will be a way to maximize the performance of Hadoop. In other words, preparing a cluster with optimized tuning [6,7] conditions means finding parameter combinations that yields optimal performance.

To classify an optimized set of parameter, a professional expert with Hadoop has to intervene to tune the cluster which is time-consuming and tedious [4].

This is because there are many parameters [4,8] to consider and difficulty in accurately grasping the correlation between parameters. Moreover, the parameter values that we expected best performance should be verified on the real Hadoop clusters with specialized MapReduce or given benchmarking applications.

Fortunately, there have been several studies conducted for efficient tuning automated manner such as Naïve exhaustive [9], random [10], and heuristically-based [11]. For the Naïve exhaustive and random approach, the TestDFSIO is used and the heuristically-based is evaluated with the TeraSort [12]. Since the methodology we have proposed applies single benchmarking application given by Hadoop distribution to evaluate each algorithm [11], it needs to be improved to verify more complex MapReduce task environments.

In this paper, we proposed enhanced version of heuristically-based algorithms to apply multiple benchmark application such as TeraSort, TestDFSIO, and MrBench.

The rest of the paper is organized as follows. Section 2 introduces preliminary works and reviews. Section 3 presents the extended heuristically-based algorithms. Section 4 analyzes experimental results. Finally in Section 5 concluding remarks are drawn along with future works.

\* Corresponding author. Email: [jkim@saumag.edu](mailto:jkim@saumag.edu)

## 2. PRELIMINARY

In this section, the background knowledge of the Hadoop configuration parameters, parameter domains, couple of benchmarking techniques and several proposed algorithms will be reviewed.

### 2.1. Hadoop Parameters

The most important and fundamental concept of Hadoop is to reduce the overall processing time by distributing one task to multiple cluster nodes. To control the efficient operation between master and slave nodes [13], tuning of configuration parameter according to system environment is needed. Well-tuned parameter values will keep the system in optimal condition and ultimately improve system performance.

Apache Hadoop has more than 80 parameters [14] and 25 of these parameters are significantly affected map and reduce performance [4]. Those parameters are defined at the \$hadoop\_home/conf/mapred-site.xml as one of the properties. For example, io.sort.mb with default value 100 and io.sort.record.percent with value 0.05 can be described as follows:

- Configuration for mapred-site.xml

```
<configuration>
<property>
<name>io.sort.mb</name>
<value>100</value>
</property>
<property>
<name>io.sort.record.percent</name>
<value>0.05</value>
</property>
<property>
<name>io.sort.spill.percent</name>
<value>0.80</value>
</property>
<property>
<name>io.sort.factor</name>
<value>10</value>
</property>
<property>
<name>mapred.job.shuffle.merge.percent</name>
<value>0.66</value>
</property>
<property>
<name>mapred.job.shuffle.input.buffer.percent</name>
<value>0.70</value>
</property>
</configuration>
```

There are several steps to set up parameter values static way, first stop all the processes (Hadoop SecondaryNameNode, DataNode, TaskTracker, JobTracker, NameNode) on the cluster using stop-all.sh, manually modify the configuration, and then use start-all.sh to start all the processes. Apache Hadoop operates using the default value defined in \$hadoop\_home/docs/hdfs-default.xml, if required parameters are not defined in the \$hadoop\_home/conf/mapred-site.xml.

At the previous work we have developed a java application framework to modify configuration files without human intervention and applied at the several parameter evaluations [9–12].

As shown in Table 1, we have selected six important parameters for evaluation set which includes a domain of double or integer type. Each parameter distinguishes the minimum value from the maximum value based on the default value. When the tuning is completed, one value will be determined for each parameter.

### 2.2. Parameter Value Domains

As shown in Table 2, all parameter values are constantly subdivided by ranging from −90% up to +90% based on its low and high values [11]. The reason for configuring this arity table is to broaden the selection that one parameter can have.

### 2.3. Parameter Space

The parameter space [9,11], denoted by  $\Omega(x)$  is a sample space composed of parameter combinations taken from arity table and its time difference generated by map and reduce applications.

Depending on the algorithm chosen, the number of parameter combinations can be variably changed. If we apply Naïve exhaustive algorithm, the size of the space will hold  $a^n$  cases with its evaluation results (where  $a$ : arity,  $n$ : number of parameter).

While the heuristically greedy algorithm requires multiple  $\Omega(x)$  spaces which will hold each cases respectively, e.g.,  $a * p$ ,  $a * (p - 1)$ ,  $a * (p - 2)$ , ...,  $a * (p - (p - 1))$ .

### 2.4. Benchmarking Applications

Apache Hadoop distribution provide predeveloped several stress testing tools such as TestDFSIO, TeraSort, MrBench and MnBench in forms of hadoop-test-1.2.1.jar or hadoop-examples-1.2.1.jar.

**Table 1** | Parameters and its values

Symbol	Parameter and description	Default	Low	High
P0(i.s.m)	io.sort.mb	100	50	120
P1(i.s.r.p)	io.sort.record.percent	0.05	0.01	0.09
P2(i.s.s.p)	io.sort.spill.percent	0.80	0.40	0.90
P3(i.s.f)	io.sort.factor	10	1	20
P4(m.j.s.m.p)	mapred.job.shuffle.merge.percent	0.66	0.44	0.88
P5(m.j.s.i.b.p)	mapred.job.shuffle.input.buffer.percent	0.70	0.40	0.90

**Table 2** | Arity table of parameters

Par \ Arity%	−90%	−80%	−70%	−60%	−50%	−40%	−30%	−20%	−10%
i.s.m	50	55	61	66	72	77	83	88	94
i.s.r.p	0.01	0.014	0.019	0.023	0.028	0.032	0.037	0.041	0.046
i.s.s.p	0.4	0.444	0.489	0.533	0.578	0.622	0.667	0.711	0.756
i.s.f	1	2	3	4	5	6	7	8	9
m.j.s.m.p	0.22	0.269	0.318	0.367	0.416	0.464	0.513	0.562	0.611
m.j.s.i.b.p	0.2	0.256	0.311	0.367	0.422	0.478	0.533	0.589	0.644
Par \ Arity%	+10%	+20%	+30%	+40%	+50%	+60%	+70%	+80%	+90%
i.s.m	111	122	133	144	155	166	177	188	200
i.s.r.p	0.056	0.061	0.067	0.072	0.078	0.083	0.089	0.094	0.1
i.s.s.p	0.811	0.822	0.833	0.844	0.856	0.867	0.878	0.889	0.9
i.s.f	11	12	13	14	15	16	17	18	20
m.j.s.m.p	0.684	0.709	0.733	0.758	0.782	0.807	0.831	0.856	0.88
m.j.s.i.b.p	0.722	0.744	0.767	0.789	0.811	0.833	0.856	0.878	0.9

The TestDFSIO [12] for measuring throughput, average IO rate mb/s, IO rate std deviation, and execution time on the HDFS clusters are as follows:

(e.g. -nrFiles 10, -fileSize 100)

```

----- TestDFSIO -----: write
Date & time: Fri Oct 19 21:38:29 EDT 2018
Number of files: 10
Total MBytes processed: 1000
Throughput mb/s: 1.6357509732718292
Average IO rate mb/s: 1.6582612991333008
IO rate std deviation: 0.2057749407824309

Test exec time sec: 82.103

----- TestDFSIO -----: read
Date & time: Fri Oct 19 21:44:03 EDT 2018
Number of files: 10
Total MBytes processed: 905
Throughput mb/s: 6.272530530132189
Average IO rate mb/s: 35.75289535522461
IO rate std deviation: 88.22740064015464

Test exec time sec: 40.019

```

The TeraSort [12] is a benchmark that combines testing the HDFS and MapReduce of a Hadoop cluster [15]. It provides MapReduce framework, File system counters, and Job Counters.

```

----- Tera Sort: (sort 30 K)
Map input bytes = 30000
Reduce input groups = 300
Combine output records = 0
Reduce output records = 300
Map output records = 300

CPU time spent (ms) = 850

```

The MrBench [12] is an application to measure performance by executing several small-scale jobs multiple times, as opposed to large tasks on TeraSort. Following example shows average time consumption for 30 runs.

```
$ hadoop jar hadoop-*test*.jar mrbench -numRuns 30
```

```

DataLines  Maps      Reduces    AvgTime (ms)
      1      2          1      31414

```

## 2.5. Cost of the Parameter Selection Algorithms

The execution time of map and reduce task will not be estimated before evaluation process has finished. This is because the correlation between one value of parameter and the others does not have a linear relationship. To verify this, we have to create a combination of values and run each case in the actual Hadoop environment. If it takes too much time, benchmarking will be less effective.

Considering three methods we have proposed, we can express the execution time as a Big O notation as follows:

### 2.5.1 Cost comparison

> Naïve exhaustive:  $O(a^p)$

> Random method:  $O(1 * p * r) \approx O(p^2)$

> Greedy:  $O(a * (p + (p - 1) + (p - 2) + \dots + (p - (p - 1)))) \approx O(p^3)$

Where,  $p$ : number of parameters, 1: constant time,  $a$ : number of parameter values.

The execution time of Naïve exhaustive method can be expressed as  $O(a^p)$  because you have to take into account the number of all cases, which in other words can be called the combinatorial problem. This approach is beneficial for finding appropriate parameters, but there is a time-consuming disadvantage.

To compensate for these drawbacks, the random method proposed. In this method, a parameter value is picked from the arity table by generating random numbers ranging from zero to 17. With this method, we could shorten the execution time, but we could not find a good candidate parameter compared to the Naïve exhaustive.

Finally, the heuristically greedy proposed to save benchmarking time and find acceptable parameters. An important point of this method was to fix the parameter value that most drastically reduced execution time in one parameter space and to test other parameters based on it.

Figure 1 shows how the benchmarking time changes as the number of parameters to be tested increases. Here we can see the heuristically greedy lies between Naïve and Random, but almost similar to Random's execution time.

It is difficult to determine a linear correlation between each parameter combination.

The heuristically greedy method is less efficient compared to the random [10], but it has been verified in previous studies as a computationally feasible way to find the best combination of parameters. The heuristically-based parametric optimization took slightly over quadratic time or at most cubic time and the performance is also great with feasible optimization time [11].

### 3. e-HEURISTIC GREEDY MODEL

The algorithms we proposed for extracting parameter sets is single benchmark application exercised to a one algorithm. For example, the TestDFSIO was applied to Naïve exhaustive and Random, the TeraSort was tested on the heuristically greedy.

However, in actual systems, it is also important to identify the parameter set that are applicable to common tasks because multiple map reduce operations are performed with different characteristics. So, we present the extended model called e-heuristic greedy that can be fulfilled for the multiple applications by extending heuristically greedy.

In this model, we evaluate map reduce performance with TeraSort, TestDFSIO, and MrBench on the Hadoop HDFS. Likewise, the heuristically-based [11] greedy, in the selection process of each stage assumes one parameter value contributing to the most significant performance reduction in the parameter space is selected to determine the subsequent parameters.

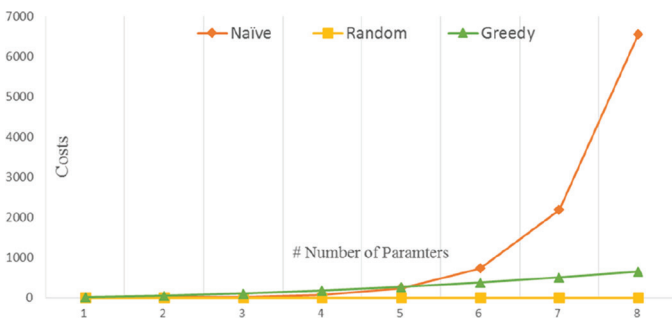


Figure 1 | Optimization cost of e-Heuristic greedy

### 3.1. e-Heuristic Greedy Structure and Operations

The overall main logic of e-Heuristic greedy is similar to heuristically greedy, but in the e-heuristic greedy, two more functions are combined to hold MrBench and TestDFSIO benchmarks. This model takes the form of batch processing. Once it is executed, the Hadoop configuration is created and three benchmarking programs are sequentially executed based on this configuration. This is because Hadoop HDFS must be restarted to apply changed parameter values for the new task. The component of the parameter space that we set conceptually defined are stored in a single text file in the Linux system.

### 3.2. e-Heuristic Greedy – Parameter Space

The parameter evaluation space denoted by  $\Omega$  composed of  $n - 1$  spaces from  $\Omega(0)$ ,  $\Omega(1)$ , ...,  $\Omega(n - 1)$  where  $n$ : number of configuration parameters. The  $\Omega(0)$  is an initial evaluation space for all parameters which includes set of values of parameter and its evaluation results generated by benchmarking applications.

The first benchmarking uses the default parameter values (i.s.m 100, i.s.r.p 0.05, i.s.s.p 0.80, i.s.f 10, m.j.s.m.p 0.66, m.j.s.i.b.p 0.70) provided by Hadoop distribution. Based on this subsequent processing continues with this value and the result is stored in the  $\Omega(x + 1)$ .

As shown in Table 3, the  $\Omega(x)$  space is composed of parameter values taken from the arity table and the execution time difference value of benchmarking compare to the result by generating Hadoop default parameter values. In order to find out how each parameter affects the execution result, 18 test cases are performed for each parameter while others are retained default value.

In Table 3 consider the fifth row, highlighted yellow, this is an example for the parameter  $p2$  indexed by column  $3^{rd}$  which is i.s.s.p (io.sort.spill.percent).

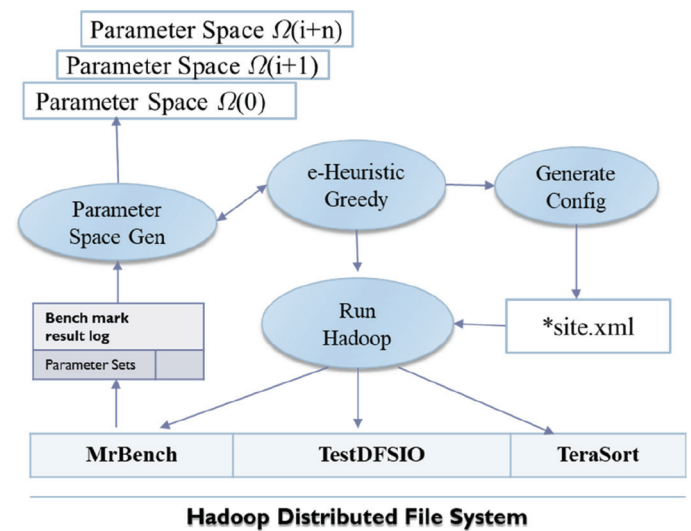


Figure 2 | e-Heuristic greedy model



**Table 3** |  $\Omega(x)$  space components

Inc	%	inx	p0	p1	p2	p3	p4	p5	Diff
+	10	0	111	0.05	0.8	10	0.66	0.7	70
+	30	4	100	0.05	0.8	10	0.733	0.7	620
–	30	5	100	0.05	0.8	10	0.66	0.533	1020
+	90	2	100	0.05	0.9	10	0.66	0.7	–2770
–	90	3	100	0.05	0.8	2	0.66	0.7	1270

The i.s.s.p increases the default value of 0.8 by +90% to obtain a value of 0.9, and the remaining five parameters retain the default value  $\{p0: 100, p1: 0.05, p2: 0.9, p3: 10, p4: 0.66, p5: 0.7\}$ . After changing the Hadoop configuration file with the given one set of parameter combinations,  $p0$ – $p5$ , three benchmarking programs are run under the same conditions.

The execution time difference, denoted by Diff, is calculated by comparing the result obtained on the combination of default parameter values and newly updated ones.

- Calculate run time differences

Diff = (diff TestDFSIO read + diff TestDFSIO write + diff TeraSort exec + diff TeraSort CPU + diff MrBench exec)

diff TestDFSIO read = (TestDFSIO read – default TestDFSIO read)

diff TestDFSIO write = (TestDFSIO read – default TestDFSIO write)

diff TeraSort exec = (TeraSort exec – default TeraSort)

diff TeraSort CPU = (TeraSort CPU – default CPU)

diff MrBench exec = (MrBench exec – default CPU)

### 3.3. e-Heuristic Greedy Model – Determining Parameter

This model also assumes the values that can be locally minimized the benchmarking results in the current stage will ultimately lead to a global optimum.

Before moving on to the next stage  $\Omega(n+1)$  from the current  $\Omega(n)$ , the parameter and its corresponding value that played a significant role in reducing system performance need to be determined by evaluating Diff value.

The only one parameter and associated value is fixed for each stage, and in the next stage, the values in the attribute table are applied to each parameter to pin down. By doing this the parameter space will be eventually decreased.

- $\Omega(x)$  space reduction

$$\Omega(0) = a * p, \Omega(1) = a * (p - 1), \dots, \Omega(n) = a * (p - n)$$

where  $p$ : number of parameters,  $a$ : arity,  $n$ : number of stages.

- Total cost of benchmarking

$$\sum_{i=0}^n ((a * (p - i)))$$

**Table 4** shows how parameters are fixed when there are six parameters. For example, if  $p = 6$  and  $a = 18$ , the stage for tuning requires five  $\Omega$  spaces, one additional parameter is fixed as the stage increases, and the remaining one parameter is determined in the final stage.

### 3.4. e-Heuristic Greedy with Alternate Parameter Selection

In the e-heuristic greedy model, the parameter is fixed based on the local minimum for each stage. However, if the local minimum value is less than previous stage, it is necessary to consider next possible candidate parameter from the previous stage and re-evaluate current stage with newly selected one to expect leading better performance.

**Figure 3** shows the processing steps and its evaluation costs. Each bubble indicates the state that exists in the evaluation spaces and arrows show the progression direction between the stages.

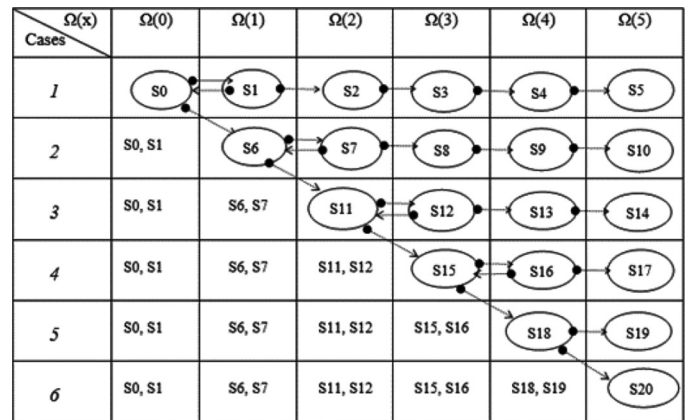
For example, in Case 2, the selected Diff value is larger than a state S1 in  $\Omega(1)$  compare to the state S0 in  $\Omega(0)$ . In this case, after selecting the candidate parameter and its values which is second smallest Diff from S0 and the S1 is re-evaluated with the newly updated sets and the result is saved at S6.

This methodology, each state can be performed once more, so the following additional cost is required. However, we expect to find better results than in e-heuristic greedy, and the performance test for this model will proceed in the next study.

**Table 4** | Parameter fixed for each space

$\Omega(0)$	$\Omega(1)$	$\Omega(2)$	$\Omega(3)$	$\Omega(4)$	$\Omega(5)$
*i.s.m	*i.s.m	*i.s.m	*i.s.m	*i.s.m	*i.s.m
i.s.r.p	*i.s.r.p	*i.s.r.p	*i.s.r.p	*i.s.r.p	*i.s.r.p
i.s.r.p	i.s.r.p	*i.s.s.p	*i.s.s.p	*i.s.s.p	*i.s.s.p
i.s.f	i.s.f	i.s.f	*i.s.f	*i.s.f	*i.s.f
m.j.s.m.p	m.j.s.m.p	m.j.s.m.p	m.j.s.m.p	*m.j.s.m.p	*m.j.s.m.p
m.j.s.i.b.p	m.j.s.i.b.p	m.j.s.i.b.p	m.j.s.i.b.p	m.j.s.i.b.p	*m.j.s.i.b.p

\* Parameters determined for the each  $\Omega(x)$

**Figure 3** | Parameter fixed for each space

$$\text{Case 1} = \sum_{i=0}^n ((a * (p - i))) \quad (1)$$

$$\text{Case 2} = \sum_{i=0}^1 ((a * (p - i))) + \sum_{i=1}^n ((a * (p - i))) \quad (2)$$

$$\text{Case 3} = \sum_{i=2}^2 ((a * (p - i))) + \sum_{i=2}^n ((a * (p - i))) \quad (3)$$

$$\text{Case } n = \sum_{i=0}^j ((a * (p - i))) + \sum_{i=j}^n ((a * (p - i))) \quad (4)$$

where  $p$ : number of parameters,  $a$ : arity,  $n$ : number of stages,  $j$ : number of cases.

### 3.5. e-Heuristic Greedy Procedures

The e-heuristic greedy extended from the heuristically-based greedy [11] composed of several functions includes `runHadoop()`, `runTeraSort()`, `runTestDFSIO()`, `runMrBench()`. Each function activated by the main, and in these functions, the `start()` method can be invoked repeatedly supported by the `ProcessBuilder` Class [15].

#### 3.5.1. e-Heuristic Greedy main

It manages all functions related to the whole benchmarking process and number of iterations for each parameter evaluation space. The `runHadoop` function responsible for control Hadoop HDFS processors on the master and slave nodes such as `NameNode`, `DataNode`, `JobTracker`, `TaskTracker`, and `SecondaryNameNode`. Any combination of parameter values should be determined and adapted to the configuration file for the next iteration of process.

**Algorithm 1.** e-Heuristic Greedy main [11]

```
Function: e-Heuristic Greedy main

Default_Par[ ] = {100, 0.05, 0.80, 5, 0.1,
                  0.66, 0.70}

Fixed_Par[ ] = { }

for (int i = 0; i < runs, i++) {
    for (int j = 0; j < n, j++) {
        Selected_Par[ ] = DeterminePar
            (IncDec, Default_Par, percentInc,
             parIdx, Low, High);
        if (!(Fixed_par.contains(Selected_Par)) {
            runHadoop( );
            runTeraSort( );
            runTestDFSIO( );
            runMrBench( );
        }
    }
}

Fixed_Par[idx + 1] = min ext_CPU of OMEGA(i)
OMEGA(i + 1) = min t_CPU of OMEGA(i)
endfor
```

#### 3.5.2. TestDFSIO

This class object is activated by main function with newly updated parameter configuration for each iteration. As shown in Algorithm 2, the write and read operation of `TestDFSIO` using `start` method to create a process instances supported java class `ProcessBuilder`.

**Algorithm 2.** Function `runTestDFSIO` [9]

```
Function: runTestDFSIO
Input: nrFiles, fileSize, numFiles
Output: Execution time (Write + Read)

ProcessBuilder TestDFSIO_write =
    NewProcessBuilder("ssh",
        "hduser@master", "hadoop", "jar",
        "~/hadoop-test-1.1.2.jar", "TestDFSIO",
        "write", "-nrFiles", fileSize,
        "-fileSize", numFiles);

run_Process(TestDFSIO_write);
run_Process(ProcessBuilder pb) {
    Process process = pb.start( );
}

ProcessBuilder TestDFSIO_read =
    NewProcessBuilder("ssh",
        "hduser@master", "hadoop", "jar",
        "~/hadoop-test-1.1.2.jar", "TestDFSIO",
        "read", "-nrFiles", fileSize,
        "-fileSize", numFiles);

run_Process(TestDFSIO_read);
run_Process(ProcessBuilder pb) {
    Process process = pb.start( );
}
```

#### 3.5.3. TeraSort

To evaluate the performance of a `TeraSort`, we follow through three steps as shown in Algorithm 3: `TeraGen`, `TeraSort`, and `TeraValidate`. Function `run_TeraGen` randomly creates data, `run_TeraSort` sorts this data, and `run_TeraValidate` obtains output results.

**Algorithm 3.** Function `runTeraSort` [11]

```
Function: runTeraGen
Input: genSize
Output: Execution time

ProcessBuilder run_TeraGen =
    new ProcessBuilder("ssh",
        "hduser@master", "hadoop", "jar",
        "~/hadoop-examples*.jar",
```

```

    "TeraGen", genSize, "/teraInput");
run_Process(run_TeraGen);
ProcessBuilder run_TeraSort =
    new ProcessBuilder ("ssh",
        "hduser@master", "hadoop", "jar",
        "~/hadoop-examples*.jar",
        "TeraSort", "/teraInput", "/teraOutput");
run_Process(run_TeraSort);
ProcessBuilder run_TeraValidate =
    new ProcessBuilder ("ssh",
        "hduser@master", "hadoop", "job",
        "-history", "all",
        "/teraInput", ">",
        "/TeraSort_results.log"
run_Process(run_TeraValidate);

```

### 3.5.4. MrBench

MrBench is also implemented in the same way as TestDFSIO. In this way, for the benchmarking, it is possible to substitute the manual typing on the command line of the Linux OS by the process builder in succession.

**Algorithm 4.** Function runMrBench

```

Function: runMrBench
Input: processDir, numRuns
Output: Execution time

ProcessBuilder run_MrBench =
    new ProcessBuilder ("hadoop", "jar",
        "~/hadoop-*test*.jar", "mrbench",
        "-baseDir",
        processDir, "-numRuns", numRuns);
run_Process(run_MrBench);
run_Process(ProcessBuilder pb) {
    Process process = pb.start( );
}

```

## 4. EVALUATION AND ANALYSIS

In this study, one Hadoop cluster consist of one master node and four slave nodes are used. Master node is configured with Intel(R) Core(TM) i5-6500 CPU, 3.20 GHz and 8 GB memory; and slave nodes with Intel(R) Core(TM) i3-6100T CPU, 3.20 GHz, and 8 GB memory. Each machine is bundled via 100 Mbps SAU Local Area Network. Hadoop version 1.2.1 and CentOS Linux release 7.3.1611 (Core) is used for the evaluation.

As shown in Table 5, a master node includes DataNode, NameNode, SecondaryNameNode, TaskTracker, JobTracker, on each slave node has TaskTracker and DataNode.

## 4.1. Evaluation Results

The performance of HDFS and MapReduce workloads are measured by TeraSort with Gensize 30 MB, MrBench with five runs, and TestDFSIO with filesize 20 MB. As shown in the following steps, one parameter and its associated values are fixed in each evaluation space. For the display purpose, the Diff value is multiplied by 10 to visualize on the graph.

### 4.1.1. Default parameter evaluation

To set up the reference value of the parameter tuning, default values are applied to the six selected parameters. The total execution time of 58.01 s is set as the reference value.

- Exec time: 58.01 s
- Default: {i.s.m: (100), i.s.r.p: (0.05), i.s.s.p: (0.80), i.s.f: (10), m.j.s.m.p: (0.66), m.j.s.i.b.p: (0.70)}

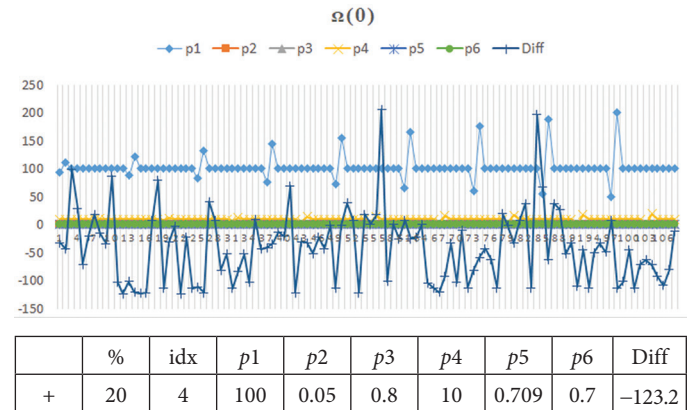
### 4.1.2. Parameter evaluation space $\Omega(0)$

In this parameter space  $\Omega(0)$ , the parameter that has the greatest effect on the total execution time is fixed to m.j.s.m.p. with its value 0.709.

- Idx: 4, m.j.s.m.p (mapred.job.shuffle.merge.percent)
- Increased +20%: 0.709, Exec: 45.69, Diff: -12.32 s

**Table 5** | Processors on Hadoop clusters

Nodes	Master node	Slaves 1	Slaves 2
<b>IP Address</b>	10.106.*.203	10.106.*.22	10.106.*.136
<b>Processors on nodes</b>	6931 TaskTracker 7043 RunJar 6744 JobTracker 7208 RunJar 6345 DataNode 6571 S-NameNode 6143 NameNode	6928 TaskTracker 6795 DataNode Slaves 3 10.106.*.164 13040 TaskTracker 12935 DataNode	12844 TaskTracker 12735 DataNode Slaves 4 10.106.*.163 3795 TaskTracker 3659 DataNode



**Figure 4** | Parameter trends on  $\Omega(0)$

- Fixed: {m.j.s.m.p: (0.709)}
- Default: {i.s.m: (100), i.s.r.p: (0.05), i.s.s.p: (0.80), i.s.f: (10), m.j.s.i.b.p: (0.70)}

### 4.1.3. Parameter evaluation space $\Omega(1)$

In  $\Omega(1)$ , the i.s.r.p with 0.032 is selected as the second candidate. We have here established two parameters m.j.s.m.p and i.s.r.p.

- Idx: 1, i.s.r.p (io.sort.record.percent)
- Increased –40%: 0.032, Exec: 43.91, Diff: –14.31 s
- Fixed: {m.j.s.m.p: (0.709), i.s.r.p: (0.032)}
- Default: {i.s.m: (100), i.s.s.p: (0.80), i.s.f: (10), m.j.s.i.b.p: (0.70)}

### 4.1.4. Parameter evaluation space $\Omega(2)$

- Idx: 3, i.s.f (io.sort.factor)
- Increased +60%: 16, Exec: 41.81, Diff: –16.2 s
- Fixed: {m.j.s.m.p: (0.709), i.s.r.p: (0.032), i.s.f: (16)}
- Default: {i.s.m: (100), i.s.s.p: (0.80), m.j.s.i.b.p: (0.70)}

### 4.1.5. Parameter evaluation space $\Omega(3)$

- Idx: 2, i.s.s.p (io.sort.spill.percent)
- Increased +10%: 0.811, Exec: 42.91, Diff: –15.1 s

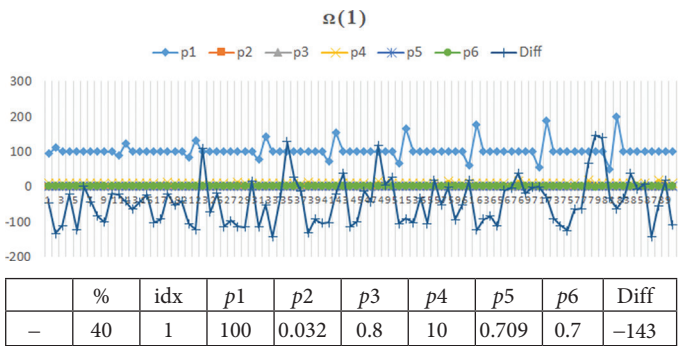


Figure 5 | Parameter trends on  $\Omega(1)$

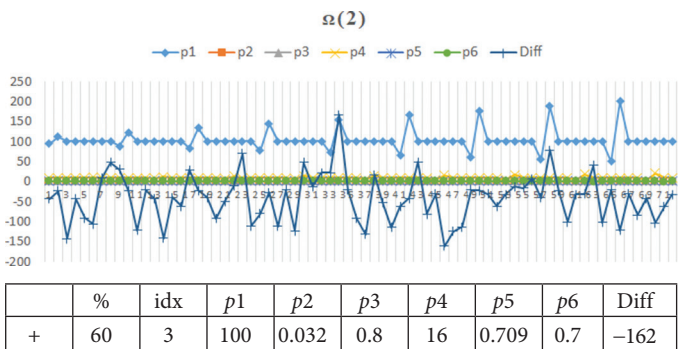


Figure 6 | Parameter trends on  $\Omega(2)$

- Fixed: {m.j.s.m.p: (0.709), i.s.r.p: (0.032), i.s.f: (16), i.s.s.p: (0.811)}
- Default: {i.s.m: (100), m.j.s.i.b.p: (0.70)}

### 4.1.6. Parameter evaluation space $\Omega(4)$

- Idx: 0, i.s.m (io.sort.mb)
- Increased –60%: 66, Exec: 40.81, Diff: –17.2 s
- Fixed: {m.j.s.m.p: (0.709), i.s.r.p: (0.032), i.s.f: (16), i.s.s.p: (0.811), i.s.m: (66)}
- Default: {m.j.s.i.b.p: (0.70)}

### 4.1.7. Parameter evaluation space $\Omega(5)$

- Idx: 5, m.j.s.i.b.p (mapred.job.shuffle.input.buffer.percent)
- Increased –10%: 0.533, Exec: 42.81, Diff: –15.2 s
- Fixed: {m.j.s.m.p: (0.709), i.s.r.p: (0.032), i.s.f: (16), i.s.s.p: (0.811), i.s.m: (66), m.j.s.i.b.p: (0.533)}
- Default: { }

## 4.2. Total Benchmarking Space Trend Analysis

Through the e-heuristic greedy algorithm, parameters are identified for each evaluation space as shown in Table 6. We did not find the most optimal parameter set in  $\Omega(5)$  at the last stage, but fortunately we could identify the set that optimizes performance

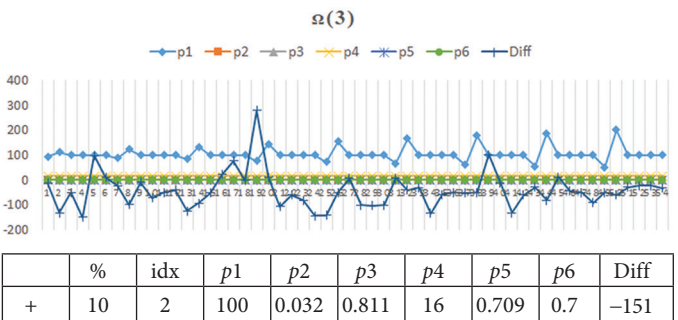


Figure 7 | Parameter trends on  $\Omega(3)$

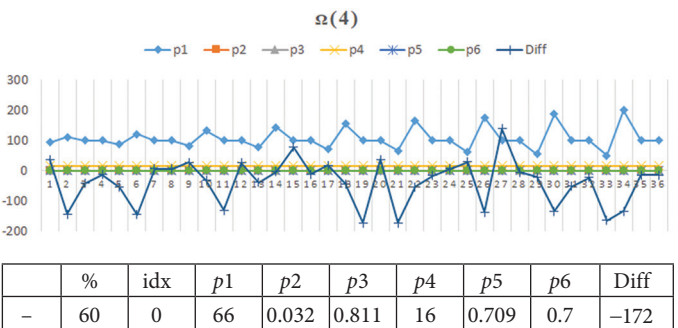


Figure 8 | Parameter trends on  $\Omega(4)$



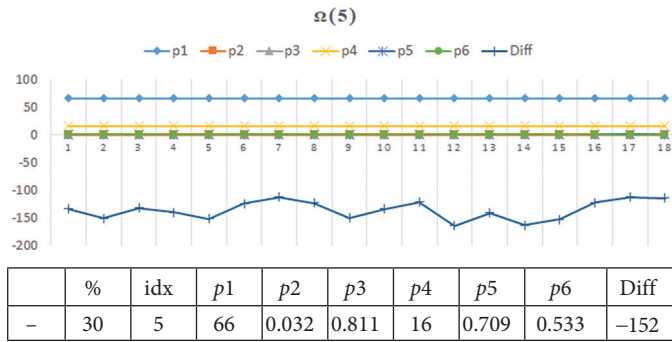
Figure 9 | Parameter trends on  $\Omega(5)$ 

Table 6 | Total parameter trend on all spaces

$\Omega \backslash p$	$p1$	$p2$	$p3$	$p4$	$p5$	$p6$	Diff
$\Omega(0)$	100	0.05	0.8	10	0.709	0.7	–12.32
$\Omega(1)$	100	0.032	0.8	10	0.709	0.7	–14.31
$\Omega(2)$	100	0.032	0.8	16	0.709	0.7	–16.18
$\Omega(3)$	100	0.032	0.811	16	0.709	0.7	–15.11
$\Omega(4)$	66	0.032	0.811	16	0.709	0.7	–17.23
$\Omega(5)$	66	0.032	0.811	16	0.709	0.533	–15.23

among the six spaces in  $\Omega(4)$  with a parameter set and values {m.j.s.m.p: (0.709), i.s.r.p: (0.032), i.s.f: (16), i.s.s.p: (0.811), i.s.m: (66), m.j.s.i.b.p: (0.70)} to minimize execution time of –17.2 s. It can be said that the local minimum value selected at each stage contributed to finding a globally optimal value.

## 5. CONCLUSION

This paper has presented a method to facilitate the extraction of common parameter set for a Hadoop cluster by applying multiple benchmarking application include TeraSort, TestDFSIO and MrBench.

This model developed by extending the heuristically-based greedy to adapt more complex map and reduce tasks. In each parameter evaluation space, one parameter and its associated value is selected that is considered to be the most locally optimum value by considering total sum of execution time. Since we assume that the selected value will ultimately lead to a global optimum, the value of this parameter is fixed in the next evaluation space and the remaining available parameters are tested.

As a result of selecting sequence, this model has found one set of parameter that shortens the execution time by 27% compared with default one at the evaluation space  $\Omega(4)$ . In each space, of course we could classify the parameter combinations with better processing time than the default one, but we found that it may not be possible to find better combinations as move on to the next space.

## ACKNOWLEDGEMENT

This study is supported by the FY1718 faculty research grant from Southern Arkansas University (10-2860-5-020, 10-2860-6180).

## REFERENCES

- [1] Apache Hadoop Home Page, <http://hadoop.apache.org/> (Last accessed – Dec 20, 2017).
- [2] H. Han, Y. Wen, T.-S. Chua, X. Li, Toward scalable systems for big data analytics: a technology tutorial, in: IEEE Access, Vol. 2, IEEE, 24 June 2014, pp. 652–687 (Big Data Framework).
- [3] S. Ghemawat, H. Gobioff, S.-T. Leung, The Google file system, in: Proceedings of the 19th ACM symposium on Operating systems principles, Vol. 37, ACM, New York, NY, USA, 19–22 October 2003, pp. 29–43.
- [4] S. Babu, Towards automatic optimization of MapReduce programs, in: Proceedings of the 1st ACM symposium on Cloud computing, ACM, New York, NY, USA, 10–11 June 2010, pp. 137–142.
- [5] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Communications of the ACM – 50th anniversary issue, ACM, New York, NY, USA, Vol. 51, January 2008, pp. 107–113.
- [6] Technical White Paper, Performance measurement of a Hadoop cluster, AMAX, 2012.
- [7] M. Kontagora, H. González-Vélez, Benchmarking a MapReduce environment on a full virtualization platform, in: 2010 International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), IEEE, Krakow, 2010.
- [8] The Fresh Open Source Software Archive, <http://fossies.org/linux/misc/hadoop-1.2.1.tar.gz/hadoop-1.2.1/docs/mapred-default.html>. Web. 20 January 2015.
- [9] J. Kim, T.K. Ashwin Kumar, K.M. George, N. Park, Performance evaluation and tuning for MapReduce computing in Hadoop distributed file system, in: 2015 IEEE 13th International Conference on Industrial Informatics (INDIN), IEEE, Cambridge, UK, 22–24 July 2015.
- [10] J. Kim, N. Park, Identification of the optimal Hadoop configuration parameter sets for MapReduce computing, in: 2015 3rd International Conference on Applied Computing and Information Technology/2nd International Conference on Computational Science and Intelligence, IEEE, Okayama, 12–16 July 2015.
- [11] J. Kim, N.-J. Park, N. Park, A study of heuristically-based parametric performance improvement/optimization algorithms for bigdata computing, in: 2016 4th Intl Conf on Applied Computing and Information Technology/3rd Intl Conf on Computational Science/Intelligence and Applied Informatics/1st Intl Conf on Big Data, Cloud Computing, Data Science & Engineering (ACIT-CSII-BCD), IEEE, Las Vegas, NV, USA, 12–14 December 2016.
- [12] M.G. Noll, Benchmarking and stress testing a Hadoop cluster with TeraSort, TestDFSIO & co., <http://www.michael-noll.com/blog/2011/04/09/benchmarking-and-stress-testing-an-hadoop-cluster-with-terasort-testdfsio-nnbench-mrbench>, 2011.
- [13] T. White, Hadoop: The Definitive Guide, O'Reilly Media, Inc., 2011, pp. 177–178.
- [14] Hadoop 1.2.1 Documentation, Mapred-default Configuration Parameters, <https://hadoop.apache.org/docs/r1.2.1/mapred-default.html>. Web. 20 January 2015.
- [15] TeraSort Benchmark Comparison for YARN, TeraSort Benchmark Comparison for YARN | MapR, <https://mapr.com/resources/terasort-benchmark-comparison-yarn/>.
- [16] ProcessBuilder (Java Platform SE 7), 23 June 2018, <https://docs.oracle.com/javase/7/docs/api/java/lang/ProcessBuilder.html>.