


Article

Best Trade-Off Point Method for Efficient Resource Provisioning in Spark

Peter P. Nghiem 

Department of Computer Engineering, School of Engineering, Santa Clara University, 500 El Camino Real, Santa Clara, CA 95053, USA; pnghiem@scu.edu; Tel.: +1-408-554-6805

Received: 20 September 2018; Accepted: 16 November 2018; Published: 22 November 2018



Abstract: Considering the recent exponential growth in the amount of information processed in Big Data, the high energy consumed by data processing engines in datacenters has become a major issue, underlining the need for efficient resource allocation for more energy-efficient computing. We previously proposed the Best Trade-off Point (BToP) method, which provides a general approach and techniques based on an algorithm with mathematical formulas to find the best trade-off point on an elbow curve of performance vs. resources for efficient resource provisioning in Hadoop MapReduce. The BToP method is expected to work for any application or system which relies on a trade-off elbow curve, non-inverted or inverted, for making good decisions. In this paper, we apply the BToP method to the emerging cluster computing framework, Apache Spark, and show that its performance and energy consumption are better than Spark with its built-in dynamic resource allocation enabled. Our Spark-Bench tests confirm the effectiveness of using the BToP method with Spark to determine the optimal number of executors for any workload in production environments where job profiling for behavioral replication will lead to the most efficient resource provisioning.

Keywords: Apache Spark; Hadoop MapReduce; YARN; algorithm for best trade-off point; optimization; resource provisioning; performance efficiency; energy efficiency; elbow curve

1. Introduction

The Gartner Inc. research firm has forecasted that the rapidly-growing cloud ecosystem will have up to 25 billion IoT sensor devices connected by 2020 [1]. This large number of devices will generate hundreds of zettabytes of information in the cloud to be analyzed by Big Data processing engines, such as Hadoop MapReduce and Apache Spark, to deliver practical value in business, technology, and manufacturing processes for better innovation and more intelligent decisions. In such an era of exponential growth in Big Data, energy efficiency has become an important issue for the ubiquitous Hadoop and Spark ecosystems.

It is now established that the energy consumption cost of a server over its useful life has far exceeded its original capital expenditure [2]. Gartner estimated in a July 2016 report that Google at the time had 2.5 million servers and counting [3]. As such, the energy consumption associated with datacenters has become a major concern, as more companies have more datacenters with over a million servers. In fact, datacenter electricity consumption is projected to increase to roughly 140 billion kilowatt-hours annually by 2020, the equivalent annual output of 50 power plants, costing American businesses \$13 billion annually in electricity bills, and emitting nearly 100 million metric tons of carbon pollution per year [4]. This energy expense, largely incurred for Big Data processing, could be reduced through more efficient resource provisioning in MapReduce and Spark among other data processing frameworks.

We previously proposed the innovative Best Trade-off Point (BToP) method and algorithm for obtaining the best trade-off between performance and computing resources for energy efficiency in any

workload running on Hadoop MapReduce [5]. Since then, we have further explored more software modules and data processing engines running on the Hadoop ecosystem including Apache Spark for similar or different approaches to optimize resource provisioning for performance efficiency. In this paper, we address the same question of how to allocate the optimal number of executor resources for a workload in Apache Spark.

Since Spark is already quite efficient in resource utilization with its built-in Dynamic Resource Allocation (DRA) mechanism, we initially did not expect to gain any further performance improvement by applying the proposed BToP method to Spark. However, our Spark-Bench Terasort experimental results prove that Spark using the BToP method could still outperform Spark with DRA enabled, particularly in production environments, where job profiling for behavioral replication will lead to the most optimal resource provisioning.

In the MapReduce programming model, multiple MapReduce jobs must be strung together to create a data pipeline with data read from disk in between every stage of the pipeline and written back to disk when completed. Apache Spark effectively minimizes these excessive disk I/O bottlenecks by keeping all activities in memory whenever possible. The performance difference between Hadoop and Spark has been evaluated by many researchers.

Taran et al. [6] tested the performances of Hadoop and Spark using Wordcount dataset of various sizes ranging from 287 KB to 5.5 GB and observed that Spark's performance was better than Hadoop. However, as the dataset size grew, the running times grew very quickly, even faster than a power function, and speedup decreased significantly. Samadi et al. [7] confirmed that the Wordcount speedup of Spark over Hadoop was influenced by the input data size. It increased when moving from 1 GB to 5 GB and decreased when moving from 5 GB to 10 GB. The authors in [7] conducted High-Bench benchmarks and found that Spark was more efficient than Hadoop in dealing with a large amount of data in major cases. Unlike Hadoop, which stores the data on disk after each map or reduce action, Spark processes data in-memory based on RDDs. If the system has insufficient memory to support huge input size, the performance will become limited when Spark runs out of the memory it needs for the creation of new RDDs. As such, Spark performance can be seriously affected by not having enough memory and can even become slower than Hadoop.

In their evaluation of several major architectural components in MapReduce and Spark frameworks, Shi et al. [8] observed that for batch jobs, Spark was a better choice for smaller datasets, since MapReduce could be several times faster than Spark when experimenting with larger datasets. For Terasort of 1 GB input, Spark was faster overall than MapReduce, because Spark had lower control overhead than MapReduce. However, for Terasort of 100 GB and 500 GB inputs, MapReduce was 1.5 times and 1.8 times faster than Spark, respectively, since MapReduce had a more efficient execution model for shuffling data than Spark. For iterative operations in machine learning such as K-Means, Spark was 1.5 times faster than MapReduce in its first iteration, and more than 5 times faster in subsequent operations. RDD caching effectively reduced the CPU overhead for parsing text to objects which was often the bottleneck for each iteration. The experimental results showed that Spark was faster than MapReduce by approximately 2.5 times for Wordcount, and 5 times for both K-means and PageRank. Kang and Lee [9,10] examined the iterative algorithms including PageRank, K-means, and logistic regression in five resource management frameworks including Hadoop and Spark, and found that Spark could reduce significant performance overheads, including network I/O, disk I/O, scheduling, and synchronization, which slowed down MapReduce, by using RDD caching and reusing it in MapReduce-like parallel operator to optimize iterative and interactive operations.

In comparing the memory-enhanced and iterative-aware version of Flame-MR-It, a Java-based MapReduce framework that improves Hadoop performance without modifying its programming APIs, with Hadoop and Spark on Amazon EC2 cloud platform, Veiga et al. [11] proved that Flame-MR could reduce Hadoop execution times by up to 65% through memory management optimization while providing competitive results compared to Spark without modifying the source code of the applications, and also using less memory. Spark consumes 62% and 25% more memory than Flame-MR-It for PageRank

and Connected Components, respectively. The performance of iterative applications is improved by the utilization of long-lived Workers, the avoidance of writing intermediate results to HDFS, and the implementation of data cache to maximize in-memory processing and minimize the use of disk.

Although the emerging Spark with its high-speed in-memory computations does not replace Hadoop MapReduce for low latency data process, many interactive data mining, iterative algorithms of machine learning, and data streaming applications previously available on MapReduce have now become deprecated and substituted by equivalent applications on Spark. Apache Spark also has the functionality to process structured data in Hive and SQL, streaming data from Flume, Twitter, and HDFS, and graphs. Therefore, a large portion of datacenter workloads will eventually be processed by Apache Spark. Hence, we expect that there will be many more research groups and Big Data companies including Databricks [12], a company founded by the creators of Apache Spark, working on the improvement of Spark's performance efficiency and energy saving upon its maturity.

Meanwhile, at the high level, many of the prior research works on Hadoop MapReduce resource provisioning to accomplish certain application performance goals through dynamic job profiling, as in Babu [13], cluster size elasticizing, as in Herodotou et al. [14], scaling past execution results to meet given Service Level Objectives (SLO), as in Verma et al. [15], and matching an application's resource consumption with a database of similar signatures of other applications, as in Kambatla et al. [16], could still be referenced when applying the BToP method in Nghiem and Figueira [5] to Apache Spark.

This paper makes the following contributions:

We further expand the use of the Best Trade-off Point (BToP) method, which was first proposed in the author's previous work on MapReduce, for efficient resource provisioning in Apache Spark, to prove that it is applicable to any system relying on a trade-off elbow curve for making good decisions (Figure 1).

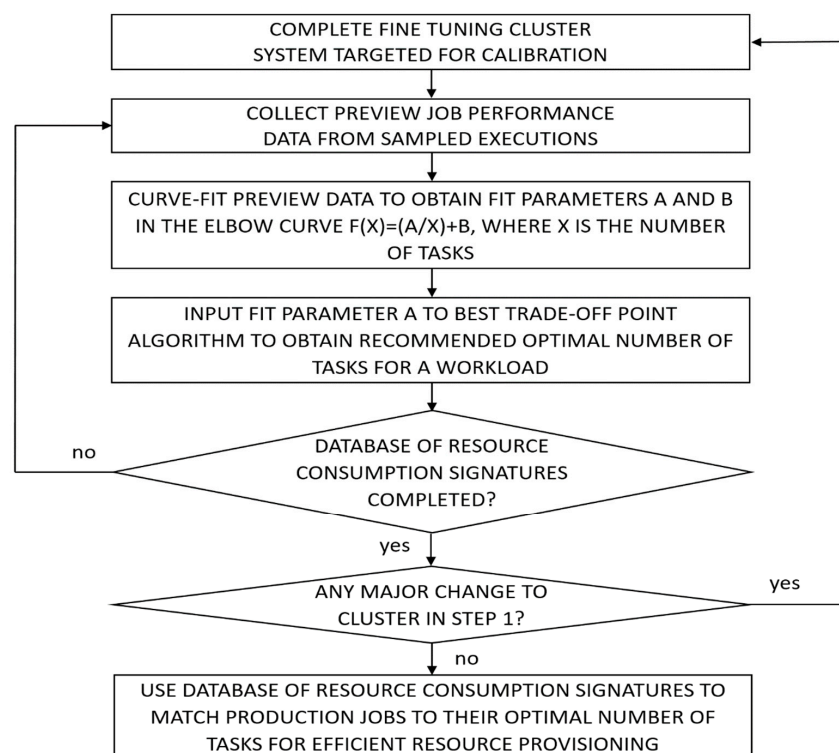


Figure 1. Flowchart of Best Trade-off Point (BToP) Method.

We further formulate the BToP method as a general approach and technique based on an algorithm with mathematical formulas to find the best trade-off point on an elbow curve, non-inverted as $f(x) = (a/x) + b$ or inverted as $f(x) = -(a/x) + b$, of performance vs. resources for a large variety of applications and systems.

We perform Spark-Bench Terasort tests with Spark's built-in DRA enabled, and then rerun the tests with DRA disabled by using the *-num-executors* option to manually assign the numbers of executors recommended by the BToP method. We then analyze and compare the job runtime performances and potential energy savings from the results of both approaches.

The results of our Spark-Bench Terasort evaluation show that the optimal numbers of executors recommended by the BToP method are consistently much better than the numbers of executors configured and used by Spark's built-in DRA during the most part of a job execution, particularly for small datasets. In addition, the overall runtime performance of Spark with DRA enabled is consistently slower than Spark using the BToP method.

Thus, our experiment with Spark-Bench Terasort confirms that Spark using the BToP method to determine the optimal number of executors for a workload not only saves energy consumption, but also improves job runtime performance in comparison to Spark with built-in DRA enabled (Tables 1 and 2 in Section 5). These improvements could add up quickly to have a significant impact on performance and cost for numerous jobs with similar profiles in a production environment.

The remainder of this paper is organized as follows:

Section 2 presents the BToP method. Section 3 presents some pertinent background knowledge on Apache Spark. Section 4 discusses our Spark-Bench Terasort experiment with the BToP method. Section 5 analyzes the performance gain and energy saving in using Spark with the BToP method vs. Spark with DRA enabled. Section 6 discusses related work. Section 7 concludes our work and proposes future research on the application of the BToP method to any systems characterized by an elbow curve.

2. Best Trade-Off Point (BToP) Method

The innovative Best Trade-off Point (BToP) method helps us find the exact optimal number of executor resources for a workload as the best trade-off point between performance and computing resources on a runtime elbow curve fitted from sampled executions of the target cluster. In addition to the cluster-computing frameworks Hadoop and Spark, the proposed BToP techniques could be used in a large variety of systems and applications which utilize a trade-off curve as a powerful tool for making informed decisions [5]. This paper will evaluate only the current most prominent cluster-computing framework, Apache Spark, running on YARN for efficiency in resource provisioning.

The seven sequential steps for implementing the proposed BToP method on a cluster-computing system are as follows (Figure 1):

- Step 1. Complete the configuration and fine tuning of the architecture, software, and hardware of the production cluster-computing system targeted for calibration.
- Step 2. Collect necessary preview job performance data from historical runtime performances or sampled executions on the same target production system, configured exactly as in step 1, as reference points for each workload.
- Step 3. Curve-fit the preview data to obtain the fit parameters a and b in the runtime elbow curve function $f(x) = (a/x) + b$, where x is the number of executor resources.
- Step 4. Input the fit parameter a to the BToP algorithm to obtain the recommended optimal number of executors for a workload (Algorithms 1 and 2).
 - a. The algorithm computes the number of executors over a range of slopes from the first derivative of $f(x) = (a/x) + b$ and the acceleration over a range of slopes from the second derivative (Algorithms 1 and 2).
 - b. The algorithm applies the Chain Rule to search for break points and major plateaus on the graphs of acceleration, slope, and executor resources over a range of incremental changes in acceleration per slope increment (Algorithms 1 and 2).
 - c. The algorithm extracts the exact number of executors at the best trade-off point on the elbow curve and outputs it as the recommended optimal number of executors for a workload (Algorithms 1 and 2).

- Step 5. Repeat steps 2–4 to gather enough resource provisioning data points for different workloads to build a database of resource consumption signatures for subsequent job profiling.
- Step 6. Repeat steps 1–5 to recalibrate the database of resource consumption signatures if there are any major changes to step 1.
- Step 7. Use the database of resource consumption signatures to match dynamically submitted production jobs to their recommended optimal number of executors for efficient resource provisioning.

The crux of the proposed BToP method is not tied to any specific system. These seven steps for implementing the BToP method and algorithm for efficient resource provisioning could also be used with other types of software components or data processing engines in the Hadoop ecosystem, any computing system, network data routing system, cluster microarchitecture system, payload engine system including but not limited to vehicle, aircraft/plane/jet, boat/ship, and rocket. This method could also be used with the yield curve of various types of securities, the convex iso yield curve, also known as convex isoquant curve, and the indifference curve in economics and manufacturing such as the semiconductor/IC yield curve, the manufacturing quality control curve, and any other types of trade-off curves with an elbow shape, non-inverted or inverted, for making intelligent decisions.

In general, the proposed Best Trade-off Point method can be used in any other types of applications which rely on an elbow curve $f(x) = (a/x) + b$ or an inverted elbow curve $f(x) = -(a/x) + b$ of performance vs. resources for good decision making including efficient resource provisioning. An elbow curve is a form of an exponential function with a negative growth rate. It is an exponential decay function $f(x) = (a/x) + b$ where $f(x)$ approaches b instead of zero when x approaches infinity (Algorithm 1). On the other hand, an inverted elbow curve is a function with inverted exponential growth $f(x) = -(a/x) + b$ where $f(x)$ approaches b instead of infinity when x approaches infinity (Algorithm 2).

Algorithm 1 Best Trade-off Point algorithm for a non-inverted elbow curve $f(x) = (a/x) + b$

Algorithm 1: Best Trade-off Point for a non-inverted elbow curve $f(x) = (a/x) + b$

Input: Parameter a for a workload with runtime curve $f(x) = (a/x) + b$

Output: Optimal number of executors

foreach incremental slope value **do**

 output number of executors $x = \text{sqrt}(-a/\text{slope})$;

end

foreach incremental slope value **do**

 output absolute value of acceleration $= 2a/\text{pow}(\text{slope}, 3)$;

end

foreach incremental target value of change in acceleration **do**

foreach incremental slope value **do**

if change in acceleration in the current slope increment is \geq to the target value AND change in acceleration in the next slope increment is $<$ the target value **then**

 output acceleration, slope, and number of executors;

 store number of tasks in an array register;

 break;

end

end

end

foreach incremental target of change in acceleration **do**

if number of executors do not change in 7 increments **then**

 output number of executors as recommended optimal value;

 break;

end

end

Algorithm 2 Best Trade-off Point algorithm for an inverted elbow curve $f(x) = -(a/x) + b$

Algorithm 2: Best Trade-off Point for an inverted elbow curve $f(x) = -(a/x) + b$

Input: Parameter a for a workload with runtime curve $f(x) = -(a/x) + b$

output: Optimal number of executors

foreach incremental slope value **do** output number of executors $x = \text{sqrt}(a/\text{slope})$;**end****foreach** incremental slope value **do** output absolute value of acceleration $= -2a/\text{pow}(\text{slope}, 3)$;**end****foreach** incremental target value of change in acceleration **do** **foreach** incremental slope value **do** **if** change in acceleration in the current slope increment is \geq to the
target value AND change in acceleration in the next slope increment
is $<$ the target value **then**

output acceleration, slope, and number of executors;

store number of tasks in an array register;

break;

end **end****end****foreach** incremental target of change in acceleration **do** **if** number of executors do not change in 7 increments **then**

output number of executors as recommended optimal value;

break;

end**end**

Typically, in Data Science applications, we could have an inverted elbow curve, also known as a knee curve (negative elbow curve), of performance vs. data, where increasing the size of data no longer improves performance beyond a certain point. Thus, the BToP algorithm for an inverted elbow curve could simply be derived from the BToP algorithm for a non-inverted elbow curve as shown in Algorithm 2. And the sequential steps of the BToP method will still be the same, with the exception that the elbow curve function is now inverted as $f(x) = -(a/x) + b$.

3. Resource Provisioning in Apache Spark

Our BToP method for optimal resource provisioning is expected to work for many other types of parallel processing frameworks running on Hadoop YARN beyond MapReduce. One such example is Apache Spark, which has recently gained momentum in terms of popularity for in-memory processing of Big Data analytic applications with better sorting performance for large clusters. Apache Spark, which can access HDFS datasets without being tied to the two-stage MapReduce paradigm [17], also supports running application JARs in HDFS. In this experiment, we evaluate the effectiveness of resource allocation of Apache Spark in Hadoop YARN cluster mode.

The BToP method is a breakthrough method for optimal resource provisioning in the scientific, computing, and economic communities, since there has been no solution for finding the best trade point between performance and resources in the past. It is a unique and innovative method for efficient resource provisioning in Spark as well as MapReduce, among many other applications and systems. Although there is an abundance of on-going development and research work to improve many features of Spark, we focus on the dynamic resource allocation mechanism in this paper, since it is the main feature directly related to the objective of the BToP method. This feature is not available in Hadoop

MapReduce, which relies on YARN for resource allocation at the container level. However, containers allocation is inefficient and does not support a high-level of cloud elasticity.

Here, we closely investigated Spark's built-in DRA because we wanted to show that despite its robustness and sophistication, our innovative BToP method can still outperform it in terms of optimal resource provisioning for efficient performance and energy saving, as shown in our experiment comparing Spark with DRA enabled and Spark using the BToP method with DRA disabled. If other features of Spark, which are not directly related to executor resource allocation, also improve its performance, such as the use of RDDs and in-memory processing vs Hadoop MapReduce's use of persistent storage and batch processing, they have already been indirectly taken into account by the intelligent BToP method and algorithm, which generate the optimal resource provisioning values from the actual historical system performance data.

In YARN cluster mode, each instance of SparkContext runs an independent set of executor processes, while YARN provides facilities for scheduling across applications. Multiple Spark jobs initiated by different threads may run concurrently within each Spark application which gets its own executor processes. Spark runs *long-running* processes and threads, which stay up through the entire duration of the application and execute tasks in multiple threads, to avoid the overhead of repeatedly invoking tasks [9,10]. Allocation of executor resources on the cluster can be controlled by Spark YARN client using the *-num-executors* option, which overrides Spark's built-in DRA mechanism [18]. We use this option to apply our BToP method and algorithm to determine the optimal static number of executors required for a workload throughout its entire execution.

Apache Spark could relinquish executors when they are no longer used, and acquire executors when they are needed, according to its DRA mechanism to gracefully decommission an executor by preserving its state before its removal using timeout [19,20]. Spark's DRA offering an elastic resource scaling ability, which is missing in MapReduce, helps prevent under-utilization of cluster resources allocated for an application and starvation of others in a multi-tenant system environment. To determine whether Spark with DRA enabled or Spark using the BToP method with DRA disabled would be most suitable for certain circumstances, we need to examine Spark's architecture with its RDD, its distributed execution and DRA mechanism.

3.1. Spark Architecture and Resilient Distributed Dataset (RDD)

Spark's driver programs access Spark through a SparkContext object, which represents a connection to a computing cluster. A driver program typically manages a few Worker Node processes, also known as Executors, which are launched at the beginning of a Spark application, and typically run for the entire lifetime of an application. Each Executor represents a unique resource unit in Spark, which is a combination of a certain number of CPU cores and memory, as specified through *spark.executor.cores* and *spark.executor.memory*, for requesting resources from the cluster manager (Figure 2).

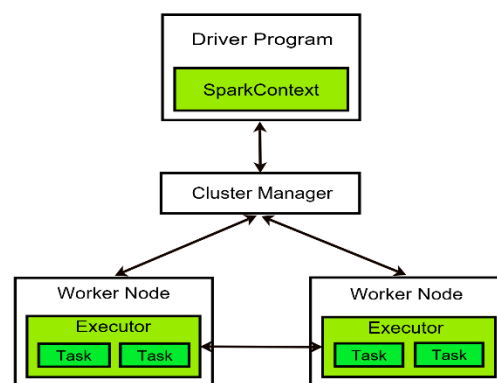


Figure 2. Spark Architecture in Cluster Mode.

The number of CPU cores occupied by each task is defined through *spark.task.cpus* as the smallest running unit in Spark execution layer. The number of required executors is derived from the number of task execution units. SparkContext is used to build an RDD, which is Spark's fundamental abstraction for distributing data and computation, and for running parallel operations. RDDs are immutable collections of objects supporting in-memory data storage distributed across clusters. Spark's efficiency is achieved through parallelization of processing across multiple cluster nodes and minimization of data replication between those nodes. Its fault-tolerance is achieved in part by logging the lineage of transformations applied to coarse-grained sets of data for recovery when needed. In case of data loss or node failure, an RDD will have sufficient information of how it was derived from other RDDs to recover the lost partition without any costly replication [21–23].

The Spark programming model is based on two types of high order functions which are transformations to create new RDD and actions to compute results without changing the original data. Transformations are lazily evaluated, since they are not executed until a subsequent action has the need for the result. This lazy operator approach further improves performance by avoiding unnecessary data processing when result is not subsequently needed. However, it can also introduce processing bottlenecks and cause application stall. For better cluster performance, RDDs persist in memory whenever possible.

Spark jobs often comprise multiple sequential stages. When an action is called on an RDD, a Directed Acyclic Graph (DAG) of stages is built from the RDD lineage graph at the low level, and submitted to the DAG scheduler. Operators are divided into stages which contain pipelined transformations with narrow dependencies. Each stage contains some tasks based on the partition of the input data with one task for every partition. A task is a combination of data and computation which is assigned to an executor pool thread. The stages are passed on to the Task Scheduler which launches tasks through cluster manager [24].

3.2. Distributed Execution in Spark

Although Spark starts up in YARN, it does all task scheduling, task forming, and process execution independent of YARN. The Spark driver runs inside of YARN's ApplicationMaster, which is responsible for driving the application and requesting resources from YARN. Spark job runs as a Java process on a JVM.

Typically, Spark program execution starts by calling `sc.textFile()` to create input RDDs from external data. New RDDs are then defined by using narrow transformations such as `filter()`, `map()`, and `union()`, which does not require data to be shuffled across the partitions, or wide transformations such as `groupByKey()` and `reduceByKey()`, which does require data to be shuffled. Any intermediate RDDs which will need to be reused are kept in memory through `persist()`. The sequence of commands creates an RDD lineage, a DAG of RDDs, which could later be used in a called action. Output results are obtained by launch actions, such as `reduce()`, `collect()`, `count()`, `first()`, and `take()`, on RDDs to begin parallel computation, which is then optimized and executed by Spark.

A lot of Spark's API revolves around passing functions to its operators to run them on the cluster. Spark automatically takes the user's function and distributes it to executors. A user's code in a single driver program is automatically distributed for execution on multiple cluster nodes. All work is expressed in either creating new RDDs, transforming existing RDDs, or calling operations on RDDs to compute a result [18,21–23].

3.3. Spark's Dynamic Resource Allocation (DRA)

Although pre-allocating a large enough number of resources for a workload could improve performance, it might lead to underutilization of cluster resources and starvation of other applications. In addition, unlike the MapReduce task, which resides in a process and is killed when the task is finished, the Spark task is a thread residing in a process known as executor, which is launched at the start of Spark application and is not killed until the application is finished. To solve these problems,

an elastic resource scaling ability, also known as Dynamic Resource Allocation feature, has been added to Spark since version 1.2 as a built-in mechanism for acquiring and releasing executors during runtime according to the load of current application. This robust feature is particularly useful when multiple applications share resources in a Spark cluster.

To enable DRA, we set *spark.dynamicAllocation.enabled* to true, and enable *spark.shuffle.service.enabled* for external shuffle data transmission to support dynamically-added and -removed executors. The minimum and maximum numbers of executors that should be allocated to an application can be specified through *spark.dynamicAllocation.minExecutors* and *Spark.dynamicAllocation.maxExecutors*. The initial number of executors can be set in the *spark.dynamicAllocation.initialExecutors* parameter. The behavior of DRA can be further fine-tuned with several properties listed in Spark's official document for DRA [19]. To disable DRA for a Spark job without hard-coding the properties in the SparkConfig file, we specified the *-num-executors* in Spark-Bench env.sh settings at time of job submission.

Spark's *ExecutorAllocationManager* calculates the maximum number of executors it requires through pending and running tasks. If the current target executor number exceeds the actual number which is needed, Spark stops adding new executors and notifies cluster manager to cancel any extra pending requests. If the new target executor number remains the same, Spark stops sending requests to the cluster manager. Unlike static allocation as in the BToP method, where all resources are provisioned at the start of the job execution, the DRA mechanism could request and remove resources dynamically during run-time. If the current executor number cannot satisfy the desired executor number, Spark updates the target number and boosts it with the number of executors to add for the first round which is doubled in each subsequent round.

To acquire executors when they are needed and to relinquish executors when they are no longer used, Spark uses several different timeouts for its request and remove policies, respectively. The request for executors is triggered when there have been pending tasks for *spark.dynamicAllocation.schedulerBacklogTimeout* seconds, and then triggered again every *spark.dynamicAllocation.sustainedSchedulerBacklogTimeout* seconds thereafter if the queue of pending tasks persists. To efficiently deal with the common slow start and fast ramp-up in actual needs, the number of executors requested in each round increase exponentially, in the order of 1, 2, 4, 8, 16, and so on, from the previous round. Any unused executor is removed when it has been idle for more than *spark.dynamicAllocation.executorIdleTimeout* seconds (the default value is 60 s) [19].

To handle multiple parallel jobs from separate threads which can run simultaneously inside a SparkContext instance, Spark's scheduler runs jobs in a FIFO fashion with an option to use a round robin fair scheduler to prevent large jobs at the head of the queue from significantly delayed, later jobs in the queue [18,19].

4. Spark-Bench Terasort Experiment with BToP Method

4.1. Test System Specifications

To illustrate our BToP method for obtaining the optimal number of executors for different workloads, we use the Teragen and Terasort components of the Spark-Bench Terasort test to experiment with 10 GB, 100 GB, and 1 TB datasets. Spark-Bench, which contains a comprehensive set of benchmark workloads for applications in machine learning, graph computation, SQL queries, and streaming application, is a Spark specific benchmarking suite proposed by Li et al. from IBM TJ Watson Research Center as a standard benchmark suite for Apache Spark [25–27]. The Terasort component among many others not mentioned in [26,27] was later added to the updated Spark-Bench on GitHub [28].

We conducted the Spark-Bench Terasort tests on a real 24-node homogeneous Hadoop cluster running Cloudera CDH-5.6 YARN with Spark 1.5.0. The cluster has 2 racks of 12 nodes each. HDFS storage is 261 TB (Raw), and 80 TB (usable after replication overhead). The NameNodes are VM of 4 cores and 24 GB of RAM, each running on Intel Xeon E5-2690 physical hosts of 8 cores and 16 threads,

2.9 GHz base frequency and 3.8 GHz max turbo frequency, and TDP of 135 W. The DataNodes are physical systems running Intel Xeon E3-1240 v3, 3.4 GHz base frequency and 3.8 GHz max turbo frequency, and TDP of 80 W. Each node has 4 cores, 8 threads, 32 GB of RAM, two 6 TB hard disks and 1 Gbit network bandwidth. All nodes are connected to a switch with a backplane speed of 48 Gbps.

4.2. Spark Configuration in the Experiment

For Spark running in YARN cluster mode, the `-num-executors` option to the Spark YARN client controls how many executors it will allocate on the cluster (as `spark.executor.instances` configuration property), while `-executor-memory` (as `spark.executor.memory` configuration property) and `-executor-cores` (as `spark.executor.cores` configuration property) control the resources per executor. When the `-num-executors` option is specifically assigned to a certain number of executors at runtime, it supersedes the `spark.dynamicAllocation.enabled` configuration which is then disabled for that job submission. For the three 10 GB, 100 GB, and 1 TB workloads, we first ran Spark-Bench Terasort with DRA enabled, and then ran Spark Bench Terasort again over a dozen times to have enough sampled data points for plotting a smooth curve, each time with a different `-num-executors` number specifically assigned through the Spark option `SPARK_EXECUTOR_INSTANCES = (-num-executors)` in Spark-Bench env.sh settings, which automatically turned off dynamic allocation.

Spark is designed for Big Data processing, but its version 1.5.0 used in our experiment is unstable when we run Spark-Bench Terasort of 1 TB of data on our 24 nodes homogeneous Hadoop cluster. To process heavy workloads in Spark, we need to occasionally maximize disk and network utilization which can cause timeout exceptions in many operations in Spark such as RPC timeout, heartbeat timeout, connection timeout, acknowledgement wait, storage blockManagerSlave timeout, and storage blockManagerMaster timeout. We find that the default value of RPC communication timeout set to 120 s is clearly insufficient for a heavy 1 TB workload processed by Spark 1.5.0 with DRA enabled. To get Spark-Bench Terasort to complete the job, we increase the timeouts by a large margin in setting the `SPARK_RPC_ASKTIMEOUT = 800 s` and adding the umbrella `SPARK_NETWORK_TIMEOUT = 800 s` to cover all timeout exceptions in Spark for heavy workloads.

On the other hand, there is no problem running Spark-Bench Terasort on a 1 TB dataset by manually assigning a static Spark's `-num-executors` without changing any default settings of timeouts in Spark. It appears that the DRA mechanism generates more disk and network utilization overhead in constantly monitoring, ramping up, and releasing executor resources, and therefore, causes timeout problems in processing heavy workloads. Spark's inherent timeout errors in processing heavy workloads have been reported by many users, especially for Spark versions older than version 1.6.0. Even on Spark version 1.6, Ullah et al. [29], in their evaluation of major big data resource management systems in the context of cloud computing environment, reported that their experimentations crashed on several occasions when the dataset was larger than 500 GB. Spark is memory intensive and all operations take place in memory. As such, it has limitations in handling very large dataset applications, and may crash when memory is no longer available for further operations. The authors in [29] confirm that before the release of version 1.5, Spark was not capable of handling datasets larger than the size of RAM and the problem of handling larger dataset persists in the newer releases with different performance overheads.

Generally, Spark's performance could easily be improved by just adding more cores and more memory. By increasing the number of cores per executor, we can force an increase in percentage of utilization and a reduction in execution time. To max out the cluster, we can start with evenly-divided memory and cores [30]. For a node with 32 GB of RAM, using YARN's *DefaultResourceCalculator* setup, which only takes the available memory into account when doing its calculation, the default resource allocation can be potentially up to 32 containers per node. However, each Worker Node in our 24-node Hadoop cluster has only 8 vCores as available logical cores from the 4 physical cores with hyperthreading, which gives 8 threads. Although each node has 32 GB of RAM and YARN's default

container allocation size is 1 GB, we are restricted to running only 8 containers of 1 vCore each per node under YARN's *DominantResourceCalculator* setup to prevent overutilization of CPU resources [31].

Following the same approach for CPU base resource allocation, Spark does its own task scheduling and process execution independent of YARN. Its configured Spark driver and executor cannot be larger than the size of a YARN container. And the total resources requested in terms of CPU and memory cannot exceed the available resources on the cluster for a user's job. In YARN cluster mode, the ApplicationMaster as a non-executor container runs the Spark driver which takes up its own resources assigned through the *-driver-memory* and *-driver-cores* properties. Thus, the available resources are determined by the following equations [32]:

$$\text{Total vCores available} \geq \text{spark.executor.cores} \times \text{spark.executor.instances} + 1 \text{ core for driver}$$

$$\begin{aligned} \text{Total Memory available} \geq & (\text{spark.executor.memory} + \text{executor memory overhead}) \\ & \times \text{spark.executor.instances} + \text{spark.driver.memory} + \text{driver memory overhead} \end{aligned}$$

In our experiment, we set the *spark.executor.memory* = 1 GB and the *spark.executor.cores* = 1 to have a total of 192 executors with 8 executors per node. The total memory used by 24 nodes is equal to only 192 GB out of 768 GB of available RAM. As such, we can optionally increase the heap size up to 3 GB without affecting the number of executors per node. In general, a single executor should not have more than 64 GB of RAM to avoid excessive garbage collection delays. We notice that with a single core executor, the overall Spark-Bench Terasort performance result in our experiment is quite slow, since the benefits of running multiple tasks in a single JVM have been eliminated by the single core executor configuration. To fine tune Spark's performance, we can increase the number of cores per executor to raise the percentage of CPU utilization at the expense of the total available number of executor instances. Ideally, to achieve full write throughput, up to 5 vCores should be assigned to each executor to support up to 5 concurrent tasks since beyond that, HDFS client could have trouble with too many concurrent threads [33].

Li et al. [26] observed in their Spark-Bench experiment that the workload execution time decreased as the number of vCores increases up to 6 cores, since Spark could launch more tasks concurrently. Nevertheless, overcommitting CPU resources, as in assigning 8 vCores, could lead to a performance degradation due to resource contention. Li et al. suggested setting 1.5 tasks per core as a rule of thumb for better performance across workloads. Moreover, Spark users should configure one big executor per node instead of many small executors with fewer allocated resources resulting in higher resource utilization, which might lead to worse performance due to resource contention.

In our experiment on a small 24-node Spark YARN cluster, we are more interested in the performance improvement impacted by the BToP method on Spark than the further fine-tuning of Spark at the expense of the total available number of executors. Therefore, we run the experiment with a single core executor to have the largest possible available number of executors for testing. In other words, due to the resource constraint in our small 24-node cluster, we run the experiment with the minimum value of vCore set for an executor to have the full performance spectrum of up to 192 executors, which is the maximum numbers of executors configurable for that cluster. Such configuration eliminates the benefits of running multiple tasks in a single JVM. However, our purpose here is to find the ratio of improvement in resource utilization, performance, and energy saving in using the BToP method on the same cluster under test. If the cluster could be properly fine-tuned with more resources as in a production environment, the performance curve is expected to shift, while the ratio of improvement should still be consistent for using Spark with the BToP method instead of the built-in DRA mechanism.

4.3. BToP Method Implementation in Spark

We now implement the seven steps outlined in Section 2 for efficient resource provisioning in Spark to apply the BToP method to Spark-Bench Terasort (Figure 1).

Step 1. Complete the configuration and fine tuning of the architecture, software, and hardware of the production cluster-computing system targeted for calibration.

Step 2. Collect necessary preview job performance data from historical runtime performances or sampled executions on the same target production system, configured exactly as in step 1, as reference points for each workload (Figure 3).

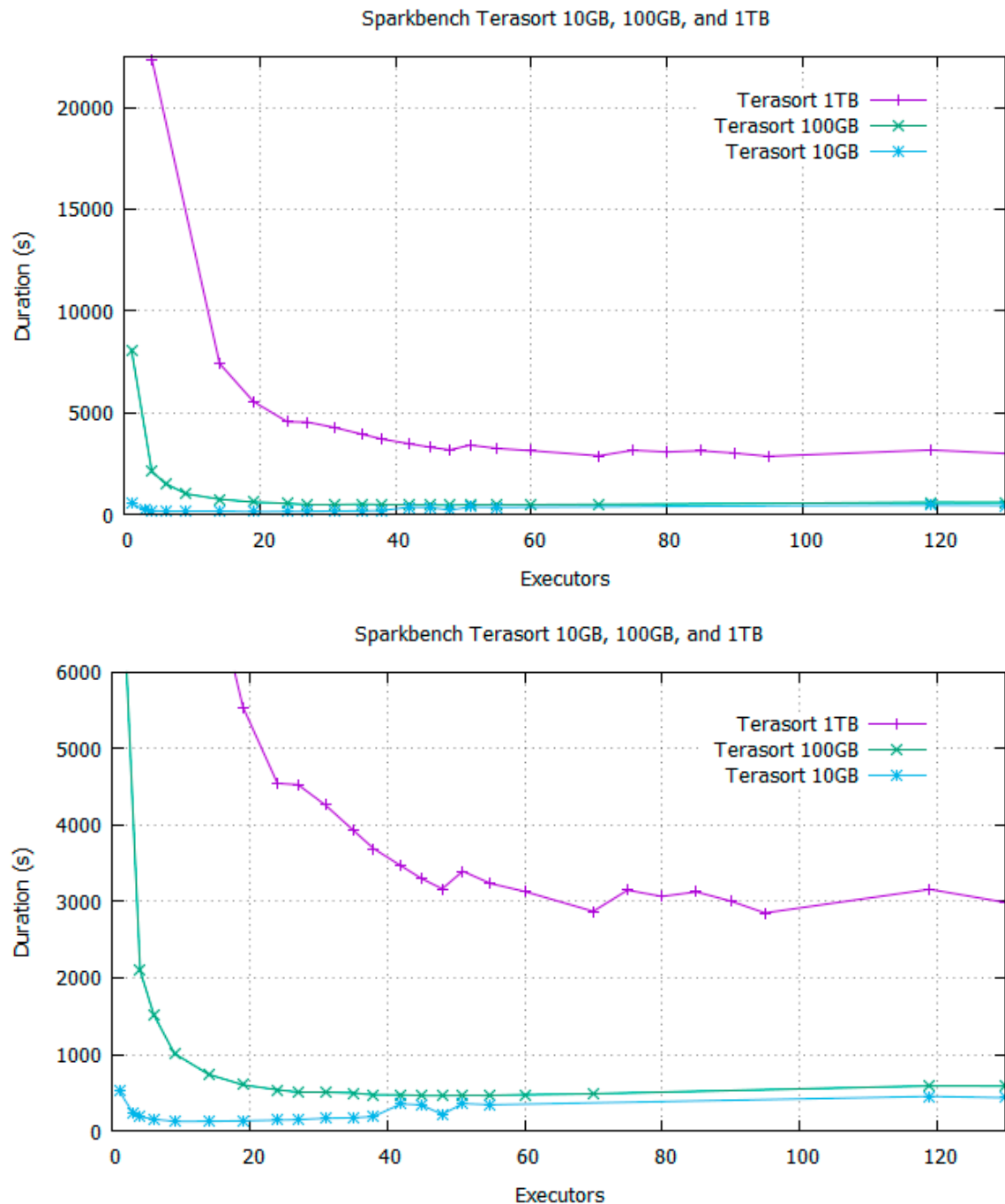


Figure 3. Plots of preview data points of job duration vs. executors for Spark-Bench Terasort 10 GB, 100 GB and 1 TB workloads at different vertical scales.

In our experiment, we gathered approximately 20 sample points for each workload to plot the elbow curves. However, a slightly smaller number of sample runs would also be enough for us to obtain the fitted duration elbow curves for computation of the recommended number of executors

for efficient resource allocation. In a production environment, if the necessary preview data to fit the elbow curve could be extracted from the system's existing historical performance data, then no further sample runs would be required. Additionally, the BToP method imposes no overhead on the framework since it neither makes any invasive change to the cluster computing system nor burdens any of its resources after the database of resource consumption signatures has been established.

The preview data of Spark-Bench Terasort indicates that when the allocated number of executors is too small for a workload, the job duration is very high due to insufficient processing resources. However, provisioning executor resources more than what is needed for a workload might result in an actual decrease in performance at some point in time after the turn of the elbow on the runtime curve. Although increasing parallelization would normally improve job performance, the growing framework overhead and potential resource contention due to an excessive number of allocated executors could eventually degrade the overall performance. In our experiment, this observed phenomenon of performance degradation of Spark on YARN despite the steady increase in executor resources is more obvious with 10 GB and 100 GB workloads than with a 1 TB workload (Figure 3).

Step 3. Curve-fit the preview data to obtain the fit parameters a and b in the runtime elbow curve function $f(x) = (a/x) + b$, where x is the number of executor resources (Figure 4).

```
gnuplot> f1(x) = (a1/x) + b1                                # Define the shape of the fitting function
gnuplot> a1 = 77521.832; b1 = 2988.399;                    # Initial estimates for the fitting parameters a1 and b1
gnuplot> fit f1(x) 'SB_Terasort_1TB.dat' using 1:7 via a1, b1
iter   chisq      delta/lim  lambda    a1          b1
  0  2.8730068229e+07  0.00e+00  3.92e+03  7.752183e+04  2.988399e+03
  1  2.6849921349e+06 -9.70e+05  3.92e+02  7.972362e+04  1.917349e+03
  2  2.4446378372e+06 -9.83e+03  3.92e+01  8.098939e+04  1.790669e+03
  3  2.4446374119e+06 -1.74e-02  3.92e+00  8.099137e+04  1.790503e+03
iter   chisq      delta/lim  lambda    a1          b1

After 3 iterations the fit converged.
final sum of squares of residuals : 2.44464e+006
rel. change during last iteration : -1.73993e-007

degrees of freedom    (FIT_NDF)                : 20
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 349.617
variance of residuals (reduced chisquare) = WSSR/ndf   : 122232

Final set of parameters      Asymptotic Standard Error
=====
a1      = 80991.4            +/- 1505            (1.859%)
b1      = 1790.5             +/- 90.53            (5.056%)

correlation matrix of the fit parameters:
      a1      b1
a1    1.000
b1   -0.568  1.000
```

Figure 4. Fitting Spark-Bench Terasort preview data points for 1 TB to the function $f(x) = (a/x) + b$.

The best trade-off point on the runtime elbow curve should be the location where no further significant decrease in job duration could be obtained by continuing to increase the number of executors. Since the rate of descending of the job duration is the downhill slope of the graph, the target point could be found in the area where the slope is gentle and no longer steep, and the vertical movement is nearly flat. To find the slope, we take the derivative of the polynomial function

$$f(x) = \left(\frac{a}{x}\right) + b \quad (1)$$

where x is the number of executors.

The derivative of $f(x)$ is a slope of a tangent line at a point x on a graph $f(x)$. It is equivalent to the slope of a secant line between two points x and $x + \Delta x$ on the graph, where Δx approaches 0.

$$f'(x) = \lim_{\Delta x \rightarrow 0} \frac{[f(x + \Delta x) - f(x)]}{\Delta x} \quad (2)$$

From (1),

$$f'(x) = -ax^{-2} \quad (3)$$

and therefore,

$$x = \sqrt{-a/f'(x)} \quad (4)$$

where $f'(x) < 0$ for a downhill slope with a negative value.

Using GNUplot [34] to curve-fit the preview data points, we obtain the fit parameters a and b of the graph function $f(x) = (a/x) + b$. GNUplot fit command uses Levenberg–Marquardt Algorithm (LMA), also known as the damped least-squares (DLS) method, which is used to solve non-linear least squares problems. LMA interpolates between the Gauss–Newton algorithm (GNA) and the method of gradient descent. However, LMA is more robust than the Gauss–Newton algorithm since, in many cases, it could find a solution even if it starts very far off the final minimum. The GNUplot fit command is used to find a set of parameters that best fits the input data to the user-defined function, which is $f(x) = (a/x) + b$ here. The fit is judged based on the Sum of Squared Residuals (SSR) between the input data and the function values, evaluated at the same places on the curve. LMA will try to minimize the weighted SSR or chisquare. A reduced chisquare much larger than 1.0 may be caused by incorrect data error estimates, data errors not normally distributed, systematic measurement errors, ‘outliers’, or an incorrect model function. The parameter error estimates, which are readily obtained from the variance-covariance matrix after the final iteration, is reported as “asymptotic standard errors” (Figure 4).

With the obtained fit parameter a , we then plot the three fitted elbow curves of job duration vs. executors for Spark-Bench Terasort 10 GB, 100 GB, and 1 TB workloads (Figure 5). The resulting performances of Spark with DRA enabled for all three workloads indicate that Spark’s DRA mechanism is already quite effective in achieving generally good performance. As such, it appears initially from the graphs that there might be no more room for further improvement in running Spark with the BToP method which disables the built-in DRA.

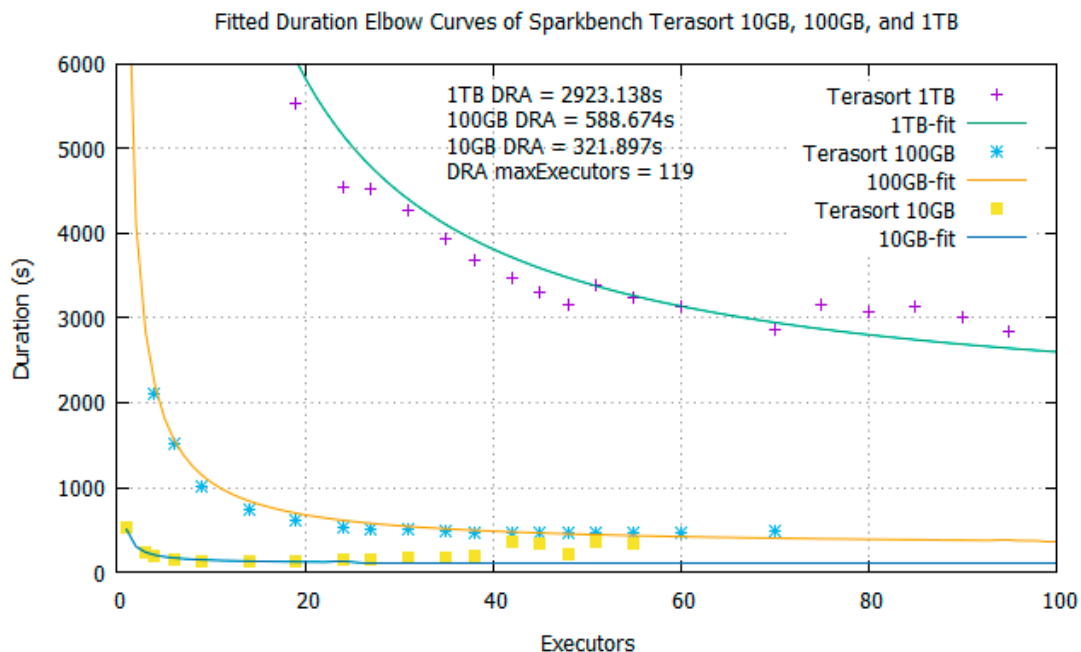


Figure 5. Fitted elbow curves of job duration vs. executors for Spark-Bench Terasort 10 GB, 100 GB, and 1 TB workloads, and their corresponding runtimes with Dynamic Resource Allocation (DRA) enabled.

Step 4. Input the fit parameter a to the BToP algorithm (Algorithm 1) to obtain the recommended optimal number of executors for a workload (Figure 6).


```

# Table of number of executors over range of slopes from -0.25 to -39.25 for x=sqrt(-a/slope)
# Slope      Terasort 10GB      Terasort 100GB      Terasort 1TB
-0.25        40.24        164.62        569.18
-1.25        18.00        73.62        254.54
-2.25        13.41        54.87        189.73
-3.25        11.16        45.66        157.86
-4.25         9.76        39.93        138.05
-5.25         8.78        35.92        124.21
-6.25         8.05        32.92        113.84
-7.25         7.47        30.57        105.69
-8.25         7.01        28.66         99.08
-9.25         6.62        27.06         93.57
-10.25        6.28        25.71         88.89
-11.25        6.00        24.54         84.85
...
#Table of acceleration over range of slopes from -0.25 to -39.25 for |accele|=|2a(1/pow(slope,3))|
# Slope      Terasort 10GB      Terasort 100GB      Terasort 1TB
-0.25       51818.24       867164.16       10366899.20
-1.25       414.55       6937.31       82935.19
-2.25       71.08       1189.53       14220.71
-3.25       23.59       394.70       4718.66
-4.25       10.55       176.50       2110.10
-5.25        5.60       93.64       1119.41
-6.25        3.32       55.50       663.48
-7.25        2.12       35.56       425.06
-8.25        1.44       24.13       288.47
-9.25        1.02       17.12       204.67
-10.25       0.75       12.58       150.42
-11.25       0.57        9.52       113.77
...
# Table of acceleration, slope, and number of executors over incremental change in acceleration
#Accele      Terasort 10GB      Terasort 100GB      Terasort 1TB
#Change      Accele Slope Executor      Accele Slope Executor      Accele Slope Executor
1           3.32 -6.25  8.05       5.82 -13.25 22.61       10.06 -25.25 56.64
2           5.60 -5.25  8.78       9.52 -11.25 24.54       16.88 -21.25 61.74
3          10.55 -4.25  9.76       12.58 -10.25 25.71       22.71 -19.25 64.86
4          10.55 -4.25  9.76       17.12 -9.25 27.06       31.56 -17.25 68.52
5          23.59 -3.25 11.16       24.13 -8.25 28.66       37.75 -16.25 70.60
6          23.59 -3.25 11.16       24.13 -8.25 28.66       37.75 -16.25 70.60
7          23.59 -3.25 11.16       24.13 -8.25 28.66       45.67 -15.25 72.88
8          23.59 -3.25 11.16       35.56 -7.25 30.57       55.98 -14.25 75.39
9          23.59 -3.25 11.16       35.56 -7.25 30.57       55.98 -14.25 75.39
10         23.59 -3.25 11.16       35.56 -7.25 30.57       55.98 -14.25 75.39
11         23.59 -3.25 11.16       35.56 -7.25 30.57       69.63 -13.25 78.18
12         23.59 -3.25 11.16       55.50 -6.25 32.92       69.63 -13.25 78.18
13         23.59 -3.25 11.16       55.50 -6.25 32.92       69.63 -13.25 78.18
...
*****
For Terasort 10GB, the recommended number of executors is 11.16
For Terasort 10GB, the recommended number of executors is 13.41
For Terasort 10GB, the recommended number of executors is 18.00
For Terasort 100GB, the recommended number of executors is 32.92
For Terasort 100GB, the recommended number of executors is 35.92
For Terasort 100GB, the recommended number of executors is 39.93
For Terasort 1TB, the recommended number of executors is 84.85
For Terasort 1TB, the recommended number of executors is 88.89
For Terasort 1TB, the recommended number of executors is 93.57
For Terasort 1TB, the recommended number of executors is 99.08
*****
First recommended number of executors for same workload provides highest efficiency
in performance/energy ratio. Subsequent number(s) slightly improves job runtime.

```

Figure 6. Applying BToP algorithm to Spark-Bench Terasort 10 GB, 100 GB, and 1 TB workloads to tabulate the number of executors over range of slopes, acceleration over range of slopes, and recommended acceleration, slope, and optimal number of executors over range of incremental changes in acceleration per slope increment, to output the final recommended optimal numbers of executors for each workload.

- a. The algorithm computes the number of executors over a range of slopes from the first derivative of $f(x) = \left(\frac{a}{x}\right) + b$ and the acceleration over a range of slopes from the second derivative (Figure 6).

Taking the second derivative of the function $f(x)$, which is the derivative of the slope, we have the acceleration of the rate of change in number of executor resources:

$$f''(x) = f'(f'(x)) \quad (5)$$

$$= \lim_{\Delta x \rightarrow 0} \frac{\left[\left(\frac{f(x+\Delta x) - f(x)}{\Delta x} \right) - \left(\frac{f(x) - f(x-\Delta x)}{\Delta x} \right) \right]}{\Delta x} \quad (6)$$

$$= \lim_{\Delta x \rightarrow 0} \frac{[f(x+\Delta x) - 2f(x) + f(x-\Delta x)]}{\Delta x^2} \quad (7)$$

as the second symmetric derivative.

From (2),

$$f''(x) = 2ax^{-3} \quad (8)$$

The BToP algorithm finds the optimal number of executors recommended for a workload by locating the best trade-off point at the bottom of the elbow curve where assigning more resources no longer significantly reduces the job duration, and therefore, reduces the overall system efficiency in resource utilization and energy consumption. Taking the parameter a in $f(x) = (a/x) + b$ as input, our program computes and tabulates the number of executors over a range of slopes from -0.25 to -39.25 for $x = \sqrt{-a/\text{slope}}$, and the acceleration over a range of slopes from -0.25 to -39.25 for $|\text{acceleration}| = |2a \times \text{slope}^{-3}|$.

- b. The algorithm applies the Chain Rule to search for break points and major plateaus on the graphs of acceleration, slope, and executor resources over a range of incremental changes in acceleration per slope increment (Figures 6 and 7).

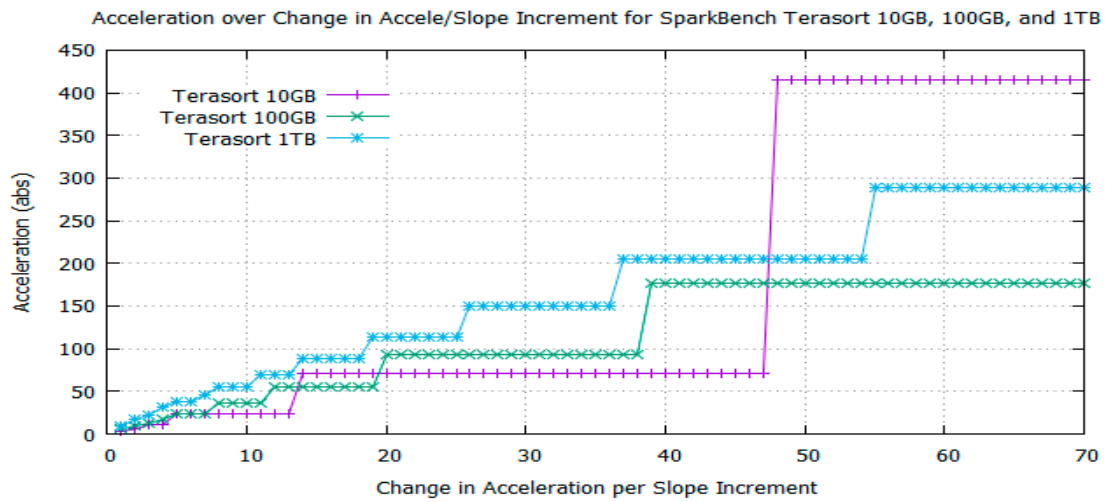
Applying the Chain Rule, the rate of change in acceleration with respect to executors is:

$$\frac{d(\text{acceleration})}{d(\text{executors})} = \frac{d(\text{acceleration})}{d(\text{slope})} \times \frac{d(\text{slope})}{d(\text{executors})} \quad (9)$$

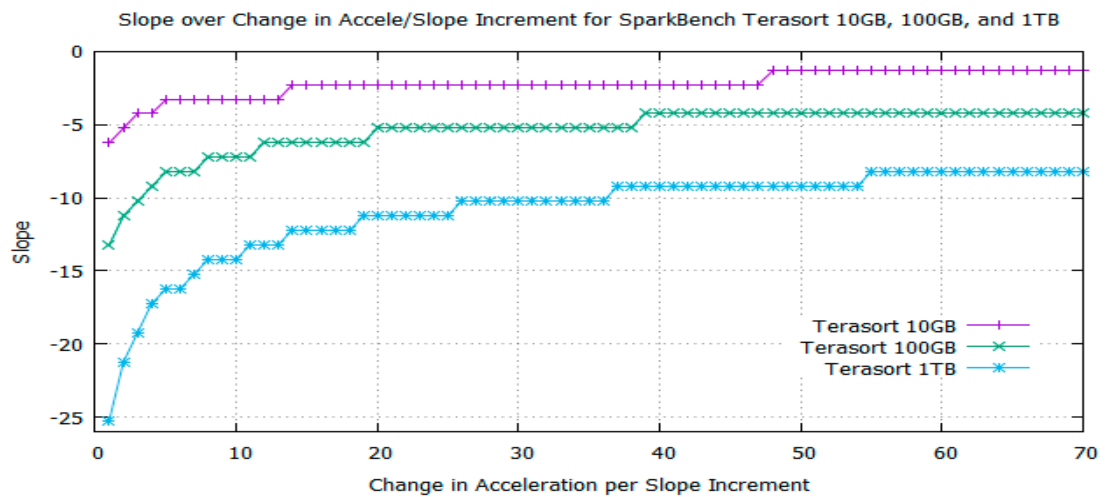
The BToP algorithm looks for break points on the graphs to compute a table of recommended acceleration, corresponding slope, and optimal number of executors, when the change in acceleration in the current slope increment is greater than or equal to the target value of change in acceleration per slope increment, and the change in acceleration in the next slope increment is less than the target value of change in acceleration per slope increment (Figures 6 and 7).

Finally, the BToP algorithm searches for all major plateaus lasting at least 7 increments of change in acceleration on the graph of executors versus change in acceleration per slope increment, which corresponds to the graph of slope versus change in acceleration per slope increment and the graph of acceleration versus change in acceleration per slope increment. The minimum plateau length of 7 increments of change in acceleration per slope increment specified for a major plateau could be adjusted by user to further fine tune the search for the first recommended optimal executor resource value. For example, if we decrease the minimum plateau length to 5 increments from 7, the first recommended number of executors for Terasort 1 TB rounded to the nearest whole number will be 81 executors instead of 85 executors according to the plot of executors versus change in acceleration per slope increment in Figure 5. Similarly, if we increase the minimum plateau length to 8 increments from 7, the first recommend number of executors for Terasort 1 TB rounded to the nearest whole number will be 89 executors instead of 85 executors. However, if we change the minimum plateau length from 7 increments to 5 or 8 increments for Terasort 10 GB and 100 GB, the first number of recommended executors will not change. Apparently, the fine-tuning option has a more sensitive impact to the first number of recommended executors when the workload is heavy such as in the range of 1 TB or larger. As such, users have the option to set a smaller value for the minimum plateau length in the BToP algorithm to further get a slightly more efficient resource provisioning when the workload is heavy.

(a)



(b)



(c)

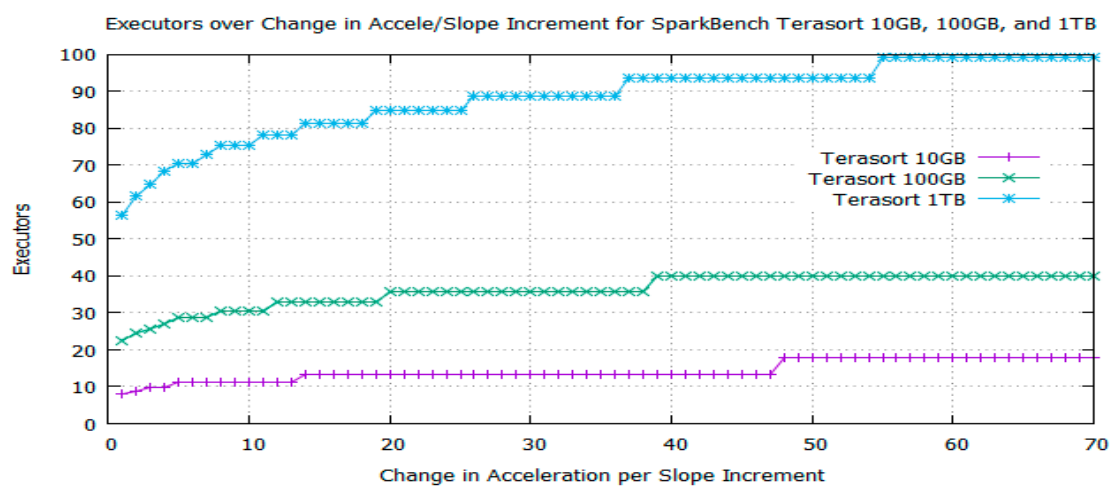


Figure 7. The optimal numbers of executors for Spark-Bench Terasort 10 GB, 100 GB, and 1 TB workloads are identified by the major plateaus lasting at least seven increments on the graphs. The algorithm searches for break points in the changes in acceleration and outputs: (a) recommended acceleration, (b) corresponding slope and (c) executors versus change in acceleration per slope increment.

- c. The BToP algorithm extracts the exact number of executors at the best trade-off point on the elbow curve and outputs it as the recommended optimal number of executors for a workload (Figures 6 and 7).

The BToP algorithm finds the optimal number of executors recommended for a workload by locating the best trade-off point on the elbow curve where assigning more executors no longer significantly reduces the job execution time, and therefore, reduces the overall system efficiency in resource utilization and energy consumption. The first recommended number of executors for the same workload provides the highest efficiency in system performance and energy consumption ratio. The subsequent recommended number(s) of executors lowers the job runtime a little bit more, but at a much less efficient performance/energy ratio. However, increasing the number of executors beyond the recommended range does not necessarily translate into any further performance gain and, on the contrary, might adversely increase the job duration. The experimental results prove that Spark using the BToP method could still consistently outperform Spark with DRA enabled in all three tested workloads of 10 GB, 100 GB, and 1 TB, despite the sophistication and robustness of Spark's DRA.

Step 5. Repeat steps 2–4 to gather enough resource provisioning data points for different workloads to build a database of resource consumption signatures for subsequent job profiling.

Step 6. Repeat steps 1–5 to recalibrate the database of resource consumption signatures if there are any major changes to step 1.

The BToP method is based on behavioral replication, which is accomplished by the creation and maintenance of a database of resource consumption signatures for job profiling to match dynamically-submitted jobs to their recommended optimal static number of resources computed by the BToP algorithm. However, we do not suggest the simple scaling of the insights gained from a small-scale run to a later large-scale execution due to the expected difference in the behavior of several parts of each cluster size including the communications, synchronizations, and convergences of its nodes. Even for the same cluster-computing system, step 6 of the BToP method requires the user to “[r]epet steps 1–5 to recalibrate the database of resource consumption signatures if there are any major changes to step 1”, which could be a change in the configuration or fine tuning of the architecture, software, and hardware of the cluster targeted for calibration.

Step 7. Use the database of resource consumption signatures to match dynamically-submitted production jobs to their recommended optimal number of executors for efficient resource provisioning.

For compute-intensive or I/O intensive applications, which are practically different kinds of workloads, their differences are inherently factored into their unique historical performance data used by the BToP algorithm to create their own specific executor resource consumption signatures for later job profiling and behavior replication of corresponding workloads.

5. Analysis of Spark with BToP method vs. Spark with DRA Enabled

5.1. Performance Gain

From the graphs of job duration vs executors of Spark-Bench Terasort, we notice that the durations of 10 GB, 100 GB, and 1 TB workloads eventually get longer instead of shorter, even though the number of executors continue to increase after 20, 60, and 95, respectively. This clearly indicates that adding far more executor resources long after the best trade-off point adversely increases Spark's runtime instead of reducing it. This phenomenon is more noticeable with small workloads of 10 GB and 100 GB than with the large 1 TB workload, due to growing framework overhead and resource contention. As such, for the 10 GB workload, we had to limit the number of preview data points used for curve fitting to only the first 7 sampled data points within the range from 1 to 20 executors on the x-axis to get the best fit of its actual elbow curve and to avoid any skew caused by the outliers beyond that range. Similarly, for the 100 GB workload, we only used the first 17 sampled data points within the range from 1 to 60 executors on the x-axis to get the best fit of its actual elbow curve. In the same way, for the 1 TB

workload, we only used the first 20 sampled data points within the range from 4 to 95 executors on the x-axis to get the best fit of its actual elbow curve for computation in the BToP method.

Our goal is to have the best fitted elbow curve from the limited preview data points within the range of interest where the best trade-off point could be located to provision only the optimal number of executors needed for a workload to reserve the remaining executor resources for other jobs in a multi-tenant Spark YARN cluster to improve the overall system throughput. As such, we could ignore the outlier data points far beyond the turn of the elbow on the runtime curve when curve fitting the elbow graph for implementing the BToP algorithm. In fact, we do not recommend provisioning more executor resources beyond the best trade-off points, which offers rapidly diminishing returns and, possibly, adverse effects, when it comes to runtime performance and energy efficiency.

Running Spark with DRA enabled, we notice that Spark dynamically fluctuates the utilized resources up to a maximum of 80 executors for the 10 GB workload and 119–120 executors for the 100 GB and 1 TB workloads. The Spark-Bench Terasort tests with DRA enabled result in 321.897 s for 10 GB, 588.674 s for 100 GB, and 2923.138 s for 1 TB (Figure 5 and Table 1).

Running Spark-Bench Terasort with BToP method (DRA disabled), the recommended optimal numbers of executors are 11–13–18 for 10 GB, 33–36–40 for 100 GB, and 85–89–94–99 for 1 TB. By extracting the corresponding runtimes from the fitted elbow curves of Spark-Bench Terasort (Figure 5), we obtain the durations of Spark with BToP method (DRA disabled) for 10 GB, 100 GB, and 1 TB workloads as shown in Table 1.

These corresponding durations derived from the optimal numbers of executors recommended by the BToP method consistently outperform the durations of Spark with DRA enabled for all three workloads. The performance gains are up to 10.76% for 1 TB, up to 18.85% for 100 GB, and up to 64.37% for 10 GB (Table 1).

Table 1. Performance of Spark-Bench Terasort with Dynamic Resource Allocation (DRA) enabled vs. Spark using BToP method (with DRA disabled).

Spark-Bench Terasort	DRA Enabled (s)	BToP Method (DRA Disabled)		Improved Performance (%)
		Executors	Duration (s)	
10 GB	321.897 s	11	129.43	59.79
		13	122.06	62.08
		18	114.70	64.37
100 GB	588.674 s	33	513.50	12.77
		36	498.77	15.27
		40	477.73	18.85
1 TB	2923.138 s	85	2739.04	6.30
		89	2694.84	7.81
		94	2644.34	9.54
		99	2608.56	10.76

The question remains whether the difference in runtimes for performance gain is significant enough to justify the additional ground work required for applying the BToP method to Spark, particularly for applications outside of a production environment. The answer to that question depends on the performance criteria and Service Level Objectives (SLO) in each individual case. Users will have to decide whether to apply the BToP method by weighing the additional benefits in performance gain and energy saving against the extra work involved in creating a database of resource consumption signature for dynamic job profiling.

Our Spark-Bench Terasort experiment proves that the BToP method is more advantageous for production runs since the BToP algorithm recommends the optimal number of executors by matching a workload to its equivalent predetermined resource consumption signature for behavioral replication.

The dynamic job profiling is expected to be more precise for identical repetitive jobs in production environment. Moreover, its database of resource consumption signatures is built from historical data and sampled executions of the same target cluster system which has been configured exactly as it was initially set up at time of calibration for later production runs.

Thus, Spark with BToP method appears to be always slightly faster than Spark with DRA enabled due to the inherent optimization of the BToP method. Nevertheless, when DRA is enabled, Spark relies on its built-in DRA mechanism, its executor request and executor remove policies, its job scheduler, and users' configurations of the dynamic allocation properties to efficiently allocate and deallocate resources as needed. Therefore, Spark with the BToP method might be more suitable for further optimizing the throughput of a target cluster in production environment, while Spark with DRA enabled is already efficient enough for general purpose applications.

We choose Spark-Bench Terasort tests for evaluating the effectiveness of Spark using the BToP method, since the datasets of different sizes generated through a random sequence by Teragen to be sorted by Terasort will always be consistent and their results can be replicated. For many other kinds of workloads, having several different numbers of incremental data sizes needed in the experiment would require us to generate several custom datasets, which will not be public standard datasets. In that case, readers might not be able to verify the test results. On the other hand, even if different kinds of workloads are used, they will not change the effectiveness of the BToP method based on its ingenious design and intelligent nature in behavioral replication. Noticeably, the difference in workloads will be inherently factored into the unique historical performance data of each specific kind of workload used by the BToP method and its algorithm to generate the recommended number of executor resources for efficient allocation as the optimal resource consumption signatures for subsequent behavioral replication. In fact, applying the BToP method to Spark-Bench Teragen gave us the same consistent improvement in performance results as Terasort which we consider as redundant and unnecessary, as it would repeat a point which has already been proven for the BToP method.

5.2. Energy Saving

The performance gain in using Spark with the BToP method also leads to some energy savings, which can be estimated and quantified by using the following energy consumption equation:

$$\text{Energy}(N) = [\text{Time}_{\text{run}}(N) \times \text{Power}_{\text{active}}(N)] + [\text{Time}_{\text{idle}} \times \text{Power}_{\text{idle}}] \quad (10)$$

The energy consumption per job can be computed from the linear sum of job duration multiplied by active power and idle duration multiplied by idle power [5,35,36]. These power models based on a linear interpolation of CPU utilization have been verified to be accurate with I/O workloads for this class of server, since network and disk activity contribute negligibly to dynamic power consumption [37,38].

Although the power consumption at the node level varies with the computation phases and workload, it could be computed at the executor level based on the number of allocated executors at each different phase. Spark's DRA mechanism provides an elastic scaling ability which monitors the load of the current application and allows executors to be allocated as needed through the acquisition and release of executors accordingly during run time. As such, we simply estimate the worst-case values of energy consumption of each workload by using the maximum number of executors assigned for DRA. For the BToP method, which is based on behavioral replication to effectively match an optimal static number of executors from the database of resource consumption signatures with its corresponding workload, we can directly compute the actual energy consumption of each specific workload.

Since the actual executor resources allocated and utilized by a workload constantly change in Spark with DRA enabled, it would be difficult to ascertain an apparent fluctuating power consumption. However, for a heavy workload of 1 TB, we observe from the logged output files that the 119 to 120 executors, as the maximum number of executors for DRA, are fully utilized most of the time in

Spark-Bench Terasort execution. Moreover, even if the maximum number of single task executor resources as single thread processes were not fully used, they would not be released until the long running application has finished. So, it is reasonable to use the maximum number of executors assigned in DRA for the heavy workload of 1 TB to estimate its energy consumption.

For Spark-Bench Terasort of 1 TB, we compare the estimated energy saving between the maximum number of executors utilized in DRA (120 executors equivalent to 15 nodes) and the highest number of executors recommended by the BToP algorithm (99 executors equivalent 13 nodes) based on a 24-node Spark YARN cluster with a maximum of 8 executors per node. For an active power consumption per node of 250 W, idle power of 235 W, and an average job arrival time of 3000 s, we have the following energy consumptions results:

$$E(15) = [2923.14 \text{ s} \times (250 \text{ W} \times 15)] + [(3000 \text{ s} - 2923.14 \text{ s}) \times 235 \text{ W}] \\ = 10,979,37.1 \text{ J} = 3049.95 \text{ Wh per job}$$

$$E(13) = [2608.56 \text{ s} \times (250 \text{ W} \times 13)] + [(3000 \text{ s} - 2608.56 \text{ s}) \times 235 \text{ W}] \\ = 8,569,808.4 \text{ J} = 2380.5 \text{ Wh per job}$$

Hence, by provisioning executor resources with the BToP method, we reduce the energy consumption by about 21.95%. This translates to $(0.66945 \text{ kWh saved per job}) \times [((365 \times 24) \text{ h/year}) / ((3000/3600) \text{ h/job})] = 7037.26 \text{ kWh saved per year}$. According to the US Department of Energy, the May 2017 average retail price of electricity for commercial customers in California was \$0.1493 per kWh. Thus, the annual energy saving amounts to \$1050.66 for the given 1 TB computing job with arrival time of 3000 s [39] (Table 2).

From the output files of Spark-Bench Terasort of 10 GB and 100 GB, we observe that Spark with DRA enabled frequently ramps up and down within the range of 0 to 80 and 120 executors, respectively, according to the need of each stage of the jobs. However, the fluctuating resource utilization rarely reached the maximum available number of executors allocated for DRA in both 10 GB and 100 GB workloads. As previously rationalized for the 1 TB workload, since their estimated energy consumptions would be inconsistent if they are modeled after a moving target, we use their maximum numbers of executors assigned for DRA, 80 executors for 10 GB workload and 120 executors for 100 GB workload, as ceiling values to calculate their energy consumption for the worst case. As such, the energy consumption for Spark with DRA enabled could typically be slightly less than the estimated worst-case values. Be that as it may, using Spark with the BToP method instead of Spark with DRA enabled for Spark-Bench Terasort of 10 GB and 100 GB workloads with an average job arrival time of 3000 s will result in an energy efficiency of 52.71% per job and 57.11% per job, respectively. That is equivalent to an annual energy saving of \$329.57 for the given 10 GB computing job and \$690.66 for the given 100 GB computing job (Table 2).

Table 2. Energy savings in using Spark with BToP method in lieu of Dynamic Resource Allocation (DRA) enabled for Spark-Bench Terasort of 10 GB, 100 GB, and 1 TB.

Spark-Bench		DRA Enabled		BToP Method (DRA Disabled)			Energy Saving		
Terasort	Max Exec.	Equiv. Nodes	Energy (Wh/job)	Static Exec.	Equiv. Nodes	Energy (Wh/job)	Per Job (%)	Per Year (kWh)	Per Year (\$)
10 GB	80	10	398.36	18	3	188.37	52.71	2207.41	329.57
100 GB	120	15	770.60	40	5	330.53	57.11	4626.02	690.66
1 TB	120	15	3049.95	99	13	2380.5	21.95	7037.26	1050.66

6. Related Work

There have been many studies on resource allocation in Spark to improve its performance. But our BToP method is the first to prove that there is still room for further optimization of resource provisioning at the executor level in Apache Spark, despite its sophisticated built-in DRA mechanism.

6.1. Performance Prediction Models

For a better resource allocation framework, Wang and Khan [40] leveraged the multi-stage execution structure of Apache Spark jobs to develop several hierarchical models for predicting job execution time, memory footprint for RDD creation, and I/O overhead. These performance metrics were extracted from a simulated run of an actual job in a limited scale on real homogeneous cluster by analyzing Spark event log files. To alleviate the effect of data skew in the simulation, input data was divided into multiple sections where sampled data was extracted with equal probability. The authors predicted the job performance based on the limited scale execution job performance data and evaluated their framework with four real-world applications. In each case, the prediction accuracy was high for execution time and memory, but varied with different applications and simulation setup for I/O cost. Although this performance prediction approach is also based on modeling system behavior, its experimental evaluation process is very time consuming and complex, with required calculations of the Stagetime, Tasktime, Startup time, and Cleanup time of job and stage, the I/O cost details for each task through TaskIOWrite and TaskIORead, and the average RDD memory footprint for each stage. In contrast to these complex performance prediction models, which might not always provide accurate and consistent results at different stages, the BToP method provides a much simpler and more reliable database of executor resource consumption signatures optimized by its algorithm from the historical job performance data of each specific application and system for unwavering precision in behavioral replication.

6.2. Optimal Resource Ratio by Distribution Probability

In exploring the effectiveness of executor resource quota connecting to stage performance with various input sizes, He et al. [41] used the time distribution probability and quota ratio of CPU cores and memory to determine the optimal ratio for each executor for a typical Spark job. By combining the resource ratio, variance probability and variant total time with the specific resource quota configuration, the authors found the correlated inflection point for best performance to allocate resources accordingly. However, such tedious executor quota calculation is quite complex, especially when it is very difficult to manually fine tune Spark performance for numerous different job characteristics to overcome varying bottlenecks at every processing stage. That is what we try to avoid with the BToP method which automates the resource provisioning process after a database of optimal executor resource consumption signatures has been generated by its algorithm from historical performance data of different workloads.

6.3. Resource Scheduling Algorithm

In the resource scheduling area, Xu et al. [42] proposed the Prophet scheduling algorithm to dynamically dispatch executors in terms of expected resource fragmentation, over-allocation, resource contention, and subsequent dramatic performance degradation due to unexpected over-allocation with its task back-off mechanism. Prophet chooses the optimal number of executors based on a greedy approach and the predictability of future executor resource demand. Although Prophet can learn and leverage the patterns in the executors' scheduling to best match time-varying executor demands to resource availability to reduce the makespan of YARN's default capacity and fair schedulers, it operates only at the scheduling level, and is specifically designed for data-parallel computation frameworks. Our BToP method, which is not really a resource scheduler, offers a novel and universal solution for optimal resource provisioning in any system characterizable by a trade-off elbow curve, and does not require any modification to the underlying system. As such, the BToP method could be used in conjunction with any resource scheduling methods, including Prophet, to get the most out of system performance improvement in Spark.

6.4. Energy Conservation Algorithm

To decrease energy consumption in Spark, Duan et al. [36] proposed a scale-down algorithm to remove under-utilized nodes to reduce the clusters' size and a Virtual Machine (VM) migration algorithm to properly adjust the load on over-utilized nodes of Spark clusters on the Cloud. Noticeably, the energy consumption of Spark clusters in the experiment was only simulated on CloudSim and not tested in real working conditions. These scaling down of operational clusters and VM migration resembles the energy conservation approach proposed for Hadoop MapReduce in Leverich et al. [37]. However, the authors in [36] did not consider the effect of Spark's built-in DRA, a sophisticated dynamic allocation mechanism which is not available in Hadoop MapReduce. Spark's DRA already handles the dynamic adjustment of resource allocation at the executor level by scaling down or ramping up the number of dispatched executors according to demand during run time. Thus, the real effectiveness of applying the additional scaling down algorithm on top of Spark's built-in DRA has yet to be verified in real working conditions since there might be cases where the algorithm could be in discord and interfere with the built-in DRA or vice versa. Since our invented BToP method matches static resources at the executor level to workloads according to a database of optimal resource consumption signatures, it can be safely combined with Duan et al.'s scale-down algorithm for better energy saving in Spark.

7. Conclusions

Spark-Bench Terasort test confirms that our proposed Best Trade-off Point method for efficient resource provisioning in Apache Spark consistently outperforms Spark with its built-in Dynamic Resource Allocation enabled, in both runtime performance and energy efficiency. Although Spark's robust and sophisticated DRA mechanism is effectual for general purpose applications including any workload with erratic and unpredictable behavior, the BToP method might be preferable in production environments, where workload behaviors are predictable and repetitive, and can be replicated for optimal performance and energy saving. By optimizing resource provisioning, the BToP method improves overall system throughput and prevents cluster underutilization as well as starvation among applications running in Spark's multi-tenancy environment.

This study shows that significant aggregate annual energy saving can be achieved when our proposed BToP method is adopted and consistently applied to all Big Data processing jobs running on all Spark and Hadoop clusters in today's large datacenters. However, since Spark could process large-scale data up to 100x faster than MapReduce in memory, or 10x faster on disk, the additional incremental gain in performance and energy saving through a more efficient resource provisioning with the BToP method may not be completely desirable, considering the extra work involved in building an extensive database of resource consumption signatures for this further optimization. Users will have to weigh the benefits, performance criteria, and Service Level Objectives (SLO) in each individual case to determine applicability and best practices.

In general, our innovative BToP method and algorithm to compute the best trade-off point for efficient resource provisioning is applicable whenever and wherever there is an elbow curve of performances vs. resources. Besides MapReduce and Spark, researchers and users will find the BToP method effective in many other applications with different software frameworks and data processing engines, any computing system, network data routing system, cluster microarchitecture system, payload engine system including but not limited to vehicle, aircraft/plane/jet, boat/ship, and rocket, and any other elbow yield curves in data science, economics, and manufacturing. In addition, the BToP method could also be coded into Artificial Intelligence in any system which could be characterized by an elbow curve of performance vs resources for efficient resource provisioning. In brief, our BToP method will work with any fundamental trade-off curves with an elbow shape, non-inverted or inverted, for making good decisions.

8. Patents

Best Trade-off Point on an elbow curve for optimal resource provisioning and performance efficiency, US patent pending (PPA #62/289159, UPA #15/418859).

Funding: This research received no external funding.

Acknowledgments: The author gratefully acknowledges use of the Hadoop cluster in the Design Center of Santa Clara University School of Engineering. The author would like to thank Sylvia M. Figueira, System Administrator Chris Tracy, and the reviewers for their feedback and continued support.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Gartner's Forecast of 25 Billion IoT Devices Connected by 2020. Available online: <http://www.gartner.com/newsroom/id/2905717> (accessed on 29 August 2018).
2. Koomey, J. Growth in data center electricity use 2005 to 2010. *A Report by Analytical Press, Completed at the Request of The New York Times*. 2011, 9. Available online: https://www.missioncriticalmagazine.com/ext/resources/MC/Home/Files/PDFs/Koomey_Data_Center.pdf (accessed on 29 August 2018).
3. Datacenter Knowledge. Available online: <http://www.datacenterknowledge.com/archives/2017/03/16/google-data-center-faq> (accessed on 29 August 2018).
4. Whitney, J.; Delforge, P. Data center efficiency assessment. *Nat. Resour. Def. Counc.* **2014**. Available online: <https://www.nrdc.org/sites/default/files/data-center-efficiency-assessment-IP.pdf> (accessed on 29 August 2018).
5. Nghiem, P.P.; Figueira, S.M. Towards efficient resource provisioning in MapReduce. *J. Parallel Distrib. Comput.* **2016**, *95*, 29–41. [CrossRef]
6. Taran, V.; Alienin, O.; Stirenko, S.; Gordienko, Y.; Rojbi, A. Performance evaluation of distributed computing environments with Hadoop and Spark frameworks. In Proceedings of the 2017 IEEE International Young Scientists Forum on Applied Physics and Engineering (YSF), Lviv, Ukraine, 17–20 October 2017; pp. 80–83. [CrossRef]
7. Samadi, Y.; Zbakh, M.; Tadonki, C. Performance comparison between Hadoop and Spark frameworks using HiBench benchmarks. *Concurr. Comput. Pract. Exp.* **2018**, *30*, e4367. [CrossRef]
8. Shi, J.; Qiu, Y.; Minhas, U.F.; Jiao, L.; Wang, C.; Reinwald, B.; Özcan, F. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proc. VLDB Endow. ACM* **2015**, *8*, 2110–2121. [CrossRef]
9. Kang, M.; Lee, J.G. An experimental analysis of limitations of MapReduce for iterative algorithms on Spark. *Clust. Comput.* **2017**, *20*, 3593–3604. [CrossRef]
10. Kang, M.; Lee, J.G. A comparative analysis of iterative MapReduce systems. In Proceedings of the Sixth International Conference on Emerging Databases: Technologies, Applications, and Theory, ACM, Jeju Island, Korea, 17–19 October 2016; pp. 61–64. [CrossRef]
11. Veiga, J.; Expósito, R.R.; Taboada, G.L.; Tourino, J. Enhancing in-memory efficiency for MapReduce-based data processing. *J. Parallel Distrib. Comput.* **2018**. [CrossRef]
12. Databricks. Available online: <https://databricks.com/spark/about> (accessed on 29 August 2018).
13. Babu, S. Towards automatic optimization of MapReduce programs. In Proceedings of the 1st ACM symposium on Cloud computing, ACM, Indianapolis, IN, USA, 10–11 June 2010; pp. 137–142. [CrossRef]
14. Herodotou, H.; Dong, F.; Babu, S. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In Proceedings of the 2nd ACM Symposium on Cloud Computing, Cascais, Portugal, 26–28 October 2011; p. 18. [CrossRef]
15. Verma, A.; Cherkasova, L.; Campbell, R.H. Resource provisioning framework for mapreduce jobs with performance goals. In Proceedings of the ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing, Lisbon, Portugal, 12–16 December 2011; Springer: Berlin/Heidelberg, Germany, 2011; pp. 165–186. [CrossRef]
16. Kambatla, K.; Pathak, A.; Pucha, H. Towards Optimizing Hadoop Provisioning in the Cloud. *HotCloud* **2009**, 9, 12.
17. Apache Hadoop. Available online: <http://hadoop.apache.org> (accessed on 29 August 2018).
18. Apache Spark. Available online: <http://spark.apache.org> (accessed on 29 August 2018).

19. Apache Spark Dynamic Resource Allocation. Available online: <http://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation> (accessed on 29 August 2018).
20. Cloudera. Spark Dynamic Allocation. Available online: http://www.cloudera.com/content/www/en-us/documentation/enterprise/latest/topics/cdh_ig_running_spark_on_yarn.html#concept_zdf_rbw_ft_unique_1 (accessed on 29 August 2018).
21. Karau, H.; Konwinski, A.; Wendell, P.; Zaharia, M. *Learning Spark: Lightning-Fast Big Data Analysis*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2015; ISBN 9781449359065.
22. Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauley, M.; Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, San Jose, CA, USA, 25–27 April 2012; p. 2.
23. Zaharia, M.; Chowdhury, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Spark: Cluster computing with working sets. *HotCloud* **2010**, *10*, 95.
24. Databricks: Understanding your Apache Spark Application Through Visualization. Available online: <https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html> (accessed on 29 August 2018).
25. Agrawal, D.; Butt, A.; Doshi, K.; Larriba-Pey, J.L.; Li, M.; Reiss, F.R.; Xia, Y. SparkBench—A spark performance testing suite. In Proceedings of the Technology Conference on Performance Evaluation and Benchmarking, Kohala Coast, HI, USA, 31 August 2015; Springer: Cham, Switzerland, 2015; pp. 26–44. [CrossRef]
26. Li, M.; Tan, J.; Wang, Y.; Zhang, L.; Salapura, V. SparkBench: A spark benchmarking suite characterizing large-scale in-memory data analytics. *Clust. Comput.* **2017**, *20*, 2575–2589. [CrossRef]
27. Li, M.; Tan, J.; Wang, Y.; Zhang, L.; Salapura, V. Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark. In Proceedings of the 12th ACM International Conference on Computing Frontiers, Ischia, Italy, 18–21 May 2015; p. 53. [CrossRef]
28. Sparkbench: Benchmark Suite for Apache Spark. Available online: <https://sparktc.github.io/spark-bench/> (accessed on 29 August 2018).
29. Ullah, S.; Awan, M.D.; Sikander Hayat Khiyal, M. Big Data in Cloud Computing: A Resource Management Perspective. *Sci. Progr.* **2018**. [CrossRef]
30. IBM Hadoop Dev/Tech Tip/Spark/Beginner's Guide: Apache Spark Troubleshooting. Available online: <https://developer.ibm.com/hadoop/2016/02/16/beginners-guide-apache-spark-troubleshooting/> (accessed on 29 August 2018).
31. Hortonworks. Managing CPU resources in your Hadoop YARN clusters, by Varun Vasudev. Available online: <https://hortonworks.com/blog/managing-cpu-resources-in-your-hadoop-yarn-clusters/> (accessed on 29 August 2018).
32. DZone/Big Data Zone. Using YARN API to Determine Resources Available for Spark Application Submission: Part II. Available online: <https://dzone.com/articles/alpine-data-how-to-use-the-yarn-api-to-determine-r> (accessed on 29 August 2018).
33. Cloudera. How-to: Tune Your Apache Spark Jobs (Part 2). Available online: <http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/> (accessed on 29 August 2018).
34. GNUplot. Available online: <http://www.gnuplot.info/> (accessed on 29 August 2018).
35. Chen, Y.; Keys, L.; Katz, R.H. *Towards energy efficient mapreduce*. EECS Department, University of California, Berkeley, Technical Report; UCB/EECS-2009-109, 120; University of California, Berkeley: Berkeley, CA, USA, 2009.
36. Duan, K.; Fong, S.; Song, W.; Vasilakos, A.V.; Wong, R. Energy-Aware Cluster Reconfiguration Algorithm for the Big Data Analytics Platform Spark. *Sustainability* **2017**, *9*, 2357. [CrossRef]
37. Leverich, J.; Kozyrakis, C. On the energy (in) efficiency of hadoop clusters. *ACM SIGOPS Oper. Syst. Rev.* **2010**, *44*, 61–65. [CrossRef]
38. Rivoire, S.; Ranganathan, P.; Kozyrakis, C. A Comparison of High-Level Full-System Power Models. *HotPower* **2008**, *8*, 32–39.
39. U.S. Energy Information Administration. Electric Power Monthly Data for May 2017. Available online: https://www.eia.gov/electricity/monthly/epm_table_grapher.php?t=epmt_5_06_a (accessed on 29 August 2018).

40. Wang, K.; Khan, M.M.H. Performance prediction for apache spark platform. In Proceedings of the 2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conference on Embedded Software and Systems (ICESS), New York, NY, USA, 24–26 August 2015; pp. 166–173. [[CrossRef](#)]
41. He, H.; Li, Y.; Lv, Y.; Wang, Y. Exploring the power of resource allocation for Spark executor. In Proceedings of the 2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS), Beijing, China, 26–28 August 2016; pp. 174–177. [[CrossRef](#)]
42. Xu, G.; Xu, C.Z.; Jiang, S. Prophet: Scheduling executors with time-varying resource demands on data-parallel computation frameworks. In Proceedings of the 2016 IEEE International Conference on Autonomic Computing (ICAC), Wuerzburg, Germany, 17–22 July 2016; pp. 45–54. [[CrossRef](#)]



© 2018 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).