6-2017

Towards Efficient Resource Provisioning in Hadoop

Peter P. Nghiem
Santa Clara University, pnghiem@scu.edu

Recommended Citation

Nghiem, Peter P., "Towards Efficient Resource Provisioning in Hadoop" (2017). *Engineering Ph.D. Theses*. 10.
http://scholarcommons.scu.edu/eng_phd_theses/10

Santa Clara University

Department of Computer Engineering

June 2017

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

Peter P. Nghiem

ENTITLED

Towards Efficient Resource Provisioning in Hadoop

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND ENGINEERING

Chair of Doctoral Committee

Member of Doctoral Committee

Member of Doctoral Committee

Member of Doctoral Committee

Member of Doctoral Committee

Chair of Department

Santa Clara University

Department of Computer Engineering

June 2017

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

Peter P. Nghiem

ENTITLED

Towards Efficient Resource Provisioning in Hadoop

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND ENGINEERING

Chair of Doctoral Committee

Member of Doctoral Committee

Member of Doctoral Committee

Member of Doctoral Committee

Member of Doctoral Committee

Chair of Department

TOWARDS EFFICIENT RESOURCE PROVISIONING IN HADOOP

by

Peter P. Nghiem

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in Computer Science and Engineering
School of Engineering
Santa Clara University

Santa Clara, California

June 2017

Abstract

Considering 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 better energy-efficient computing. This thesis proposes 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 and Apache Spark. Our novel BToP method is expected to work for any applications and systems which rely on a trade-off curve with an elbow shape, non-inverted or inverted, for making good decisions. This breakthrough method for optimal resource provisioning was not available before in the scientific, computing, and economic communities.

To illustrate the effectiveness of the BToP method on the ubiquitous Hadoop MapReduce, our Terasort experiment shows that the number of task resources recommended by the BToP algorithm is always accurate and optimal when compared to the ones suggested by three popular rules of thumbs. We also test the BToP method on the emerging cluster computing framework Apache Spark running in YARN cluster mode. Despite the effectiveness of Spark's robust and sophisticated built-in dynamic resource allocation mechanism, which is not available in MapReduce, the BToP method could still consistently outperform it according to our Spark-Bench Terasort test results. The performance efficiency gained from the BToP method not only leads to significant energy saving but also improves overall system throughput and prevents cluster underutilization in a multi-tenancy environment. In General, the BToP method is preferable for workloads with identical resource consumption signatures in production environment where job profiling for behavioral replication will lead to the most efficient resource provisioning.

Keywords

Best Trade-off Point; Hadoop MapReduce; Apache Spark; YARN; Optimal resource provisioning; Dynamic resource allocation; Energy efficiency; Performance efficiency; Yield curve; Exponential decay curve; Elbow curve.

Acknowledgments

First, I would like to thank my PhD advisor, Professor Dr. Silvia M. Figueira for accepting me to the PhD program under her supervision, supporting my research, guiding me throughout the academic process, having faith in my innovative ideas and hard work, and examining and proofreading my journal research papers.

I would like to thank all the members of my doctoral committee, who are Professor Dr. Nicholas Tran, Professor Dr. Yi Fang, Professor Dr. Weijia Shang, and Professor Dr. Ahmed Amer, for their continuing support, encouragement, suggestion, and advice in my research and studies for the PhD degree in Computer Science and Engineering. I particularly would like to express my appreciation to Professor Dr. Nicholas Tran who thoroughly examined my journal research paper and provided good feedback in every step of the way. I also particularly would like to express my appreciation to Professor Dr. Yi Fang for letting me use his entire group research disk space on SCU Design Center's Hadoop cluster for my benchmark testing and examining my journal research paper. I would like to especially thank SCU Design Center System Administrator Chris Tracy for setting up, configuring, and maintaining any necessary Hadoop MapReduce, Spark, and Spark-Bench suite and benchmark tools, and allocating additional disk space sufficient for my research on SCU Design Center's Hadoop cluster whenever needed. Special thanks go to Professor Dr. JoAnne Holliday, Professor Dr. Ahmed Amer, and Professor Dr. Weijia Shang for advising me in the Master of Science degree in Computer Science and Engineering. In addition, I would like to thank the late Professor Dr. Robert Parden for advising me in the Master of Science degree in Engineering Management and Leadership.

I would like to thank the Chair of SCU Computer Engineering department, Professor Dr. Nam Ling for his continuing support, guidance, and advice on the academic process and requirements of SCU graduate programs in Computer Science and Engineering. I also would like to thank the Dean of SCU School of engineering, Dean Professor Dr. Godfrey Mungal for his continuous support and advice, and the Dean's Fellowship for my continuing PhD research at SCU. Finally, I would like to thank all the professors who have

giving me invaluable knowledge in their courses and advices over the decades of my graduate engineering studies at SCU.

Last, but not least, I would like to thank my parents, my sisters and brothers, my other relatives, and my friends, who have supported my graduate engineering studies and academic goal at SCU.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Acknowledgment of Funding and Use of Facilities

The work presented in this thesis was mostly supported by the author’s own federal student loans. The second part of the author’s research during the academic year 2016-2017 was supported in part by a fellowship from the Dean of SCU School of Engineering. The author gratefully acknowledges use of the Hadoop cluster of SCU School of Engineering Design Center.

1.2 Thesis Overview

This thesis addresses the problem of allocating the right amount of task resources for a workload in Hadoop MapReduce and similarly, the right number of executors for a workload in Apache Spark. It relates generally to resource provisioning in software framework and computing systems including but not limited to parallel and distributed processing of Big Data by large computer clusters. More specifically, it relates to methods for determining the number of task resources for each different workload to optimally balance performance and energy efficiency.

Gartner Inc. research firm has forecasted that the rapidly-growing cloud ecosystem will have up to 25 billion IoT sensor devices connected by 2020 [15]. This large number of IoT 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, and other analytics platforms 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 [27]. Gartner estimated in a July 2016 report that Google at the time had 2.5 million servers and counting [13]. 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 [40]. A large portion of datacenter workloads is processed by Hadoop MapReduce and Apache Spark, popular de facto standard frameworks for Big Data processing, which have been adopted by many world's leading cloud computing providers and top Big Data companies, among many other organizations and institutions. As such, we could reduce this datacenter energy expense which is largely incurred for Big Data processing, through more efficient resource provisioning in MapReduce and Spark among other data processing frameworks.

There is a growing amount of research work dedicated to making these Big Data processing frameworks more energy efficient. However, there has been no definite answer to the question of what optimal number of resources should be allocated for a job to get the most efficient performance from MapReduce until now when this proposed Best Trade-off Point method is made available. Hadoop developers and users previously had to rely on popular but inaccurate rules of thumb widely circulated in industry for their MapReduce job execution, leading to significant unintended waste of computing resources and energy. This thesis proposes an innovative method and algorithm for obtaining the best trade-off between performance and computing resources for energy efficiency in any workload running on Hadoop MapReduce and Apache Spark among other applications and systems which rely on a trade-off elbow curve, non-inverted or inverted, for good decision making.

1.3 Thesis Contributions

This thesis makes the following contributions:

We develop the Best Trade-off Point (BToP) method which is applicable to any system relying on a trade-off elbow curve for making good decisions (Fig. 1). It provides a general approach and techniques 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.

We present the BToP method as a job profiling method for optimal resource provisioning for any MapReduce workload by getting runtime samples of the cluster targeted for calibration as reference points for curve-fitting and computation to find the best trade-off point on the runtime elbow curve (Fig. 1).

We then apply the BToP method to Apache Spark running in YARN cluster mode. We show how preview data can be extracted from sample executions of different workloads on Spark YARN cluster computing system targeted for calibration as reference points for curve-fitting and computation to find the best trade-off point on the runtime elbow curve (Fig. 1).

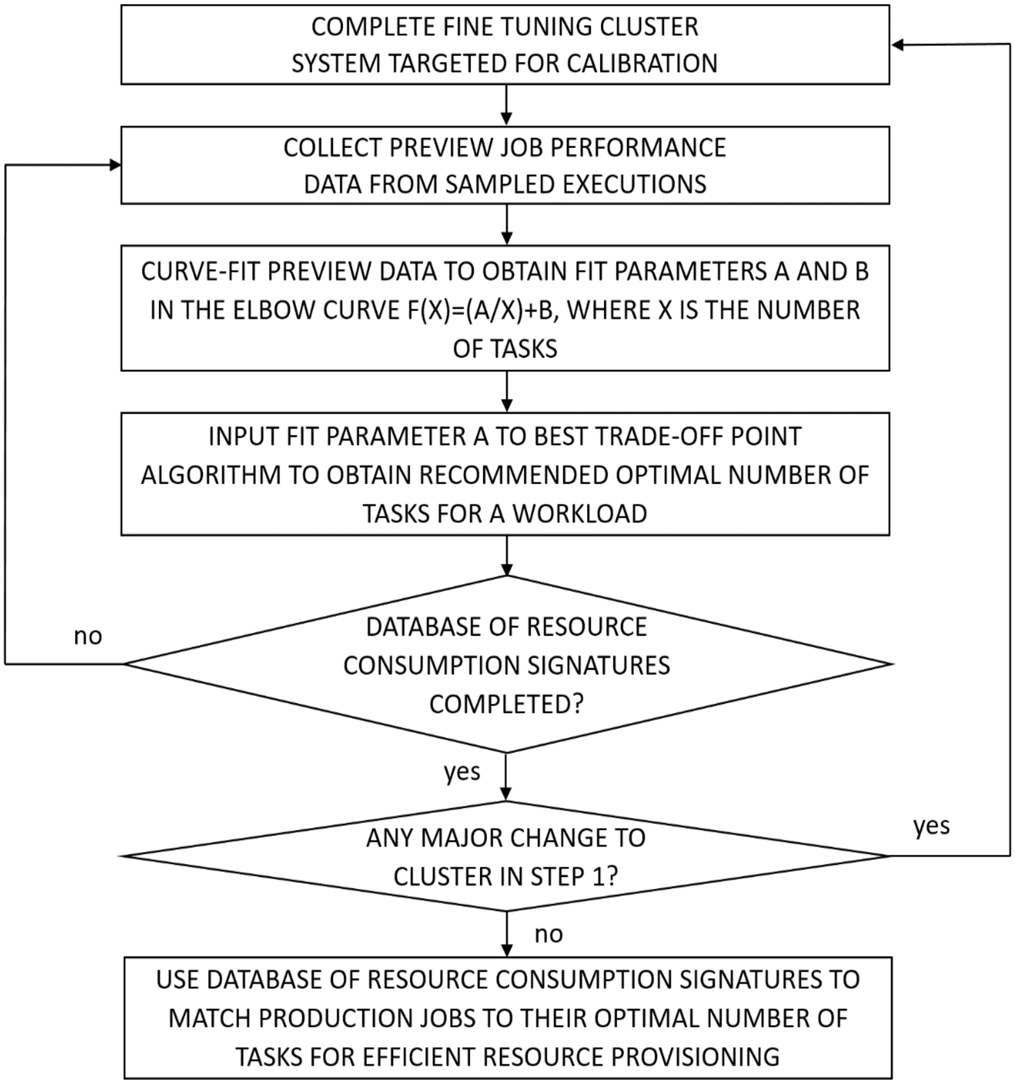

COMPLETE FINE TUNING CLUSTER SYSTEM TARGETED FOR CALIBRATION
COLLECT PREVIEW JOB PERFORMANCE DATA FROM SAMPLED EXECUTIONS
CURVE-FIT PREVIEW DATA TO OBTAIN FIT PARAMETERS A AND B IN THE ELBOW CURVE F(X)=(A/X)+B, WHERE X IS THE NUMBER OF TASKS
INPUT FIT PARAMETER A TO BEST TRADE-OFF POINT ALGORITHM TO OBTAIN RECOMMENDED OPTIMAL NUMBER OF TASKS FOR A WORKLOAD
DATABASE OF RESOURCE CONSUMPTION SIGNATURES COMPLETED?
no
yes
ANY MAJOR CHANGE TO CLUSTER IN STEP 1?
yes
no
USE DATABASE OF RESOURCE CONSUMPTION SIGNATURES TO MATCH PRODUCTION JOBS TO THEIR OPTIMAL NUMBER OF TASKS FOR EFFICIENT RESOURCE PROVISIONING

Fig. 1 Flowchart of Best Trade-off Point Method

We also provide a step-by-step computation process with mathematical formulas for the runtime graph function $f(x) = (a/x) + b$, its first derivative, its second derivative, the Chain Rule, and search conditions for breakpoints and major plateaus to find the optimal number of tasks.

We design an algorithm for best trade-off point to take the single parameter a in the graph function $f(x) = (a/x) + b$ for a workload as input and output the exact recommended optimal number of task resources.

We validate our design and techniques using experiments on a real 24-node homogeneous Hadoop cluster with Teragen and Terasort components of the Terasort benchmark test with 10 GB, 100 GB and 1 TB of data (Fig. 8 in Section 3.5.1).

We verify and compare the results of our algorithm against the numbers of tasks suggested by three currently well-known rules of thumbs widely circulated in industry using the fitted runtime elbow curves. We also provide a numerical example of potential energy savings from the results.

The results of our evaluation show that our approach consistently provides accurate and optimal number of task resources for any MapReduce workload to achieve performance efficiency while the numbers of reduce tasks suggested by the three popular rules of thumb are inaccurate leading to significant unintended waste of computing resources and energy as shown in Fig. 17 in Section 3.6.

We, then, further verify the Best Trade-off method with Apache Spark workloads by executing Spark-Bench Terasort with 10GB, 100GB, and 1TB of data on the same real 24-node homogeneous Hadoop cluster, albeit with a different and newer CDH (Cloudera's Distribution Including Apache Hadoop) version. The Spark-Bench Terasort tests were measured with Spark’s built-in dynamic resource allocation feature first set to enabled and then disabled by manually assigning the numbers of executors.

The results of our Spark-Bench Terasort evaluation show that the optimal numbers of executors recommended by the Best Trade-off Point method are consistently better than the numbers of executors used by Spark’s dynamic resource allocation during the most part of a job execution for large datasets. The overall runtime of Spark using dynamic resource allocation is consistently slower than the duration of the same workload using Spark with the Best Trade-off Point method (Fig. 25 and Table 1 in Section 4.2). In other words, Spark using the BToP method consistently outperforms Spark using the built-in dynamic resource allocation, particularly in production environment where job profiling for behavioral replication will lead to the most optimal resource provisioning.

Our experiment with Spark-Bench Terasort confirms that Spark using 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 its built-in dynamic resource allocation enabled (Tables 1-2 in Section 4.2). These improvements could add up quickly to make a significant impact in performance and cost for numerous jobs with similar profiles in production environment.

Chapter 2

Best Trade-off Point Method (BToP)

The Best Trade of Point (BToP) method provides an innovative approach and algorithm with mathematical formulas for finding an exact optimal number of computing resources for a workload as the best trade-off point between performance and resources on a runtime elbow curve fitted from sampled executions of the target cluster. The proposed 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. This thesis will focus on efficient resource provisioning in the two most prominent cluster-computing frameworks, Hadoop MapReduce and Apache Spark.

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

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

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.

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 task resources.

4. Input the fit parameter *a* to the Best Trade-off Point algorithm to obtain the recommended optimal number of tasks for a workload (Fig. 2-3).

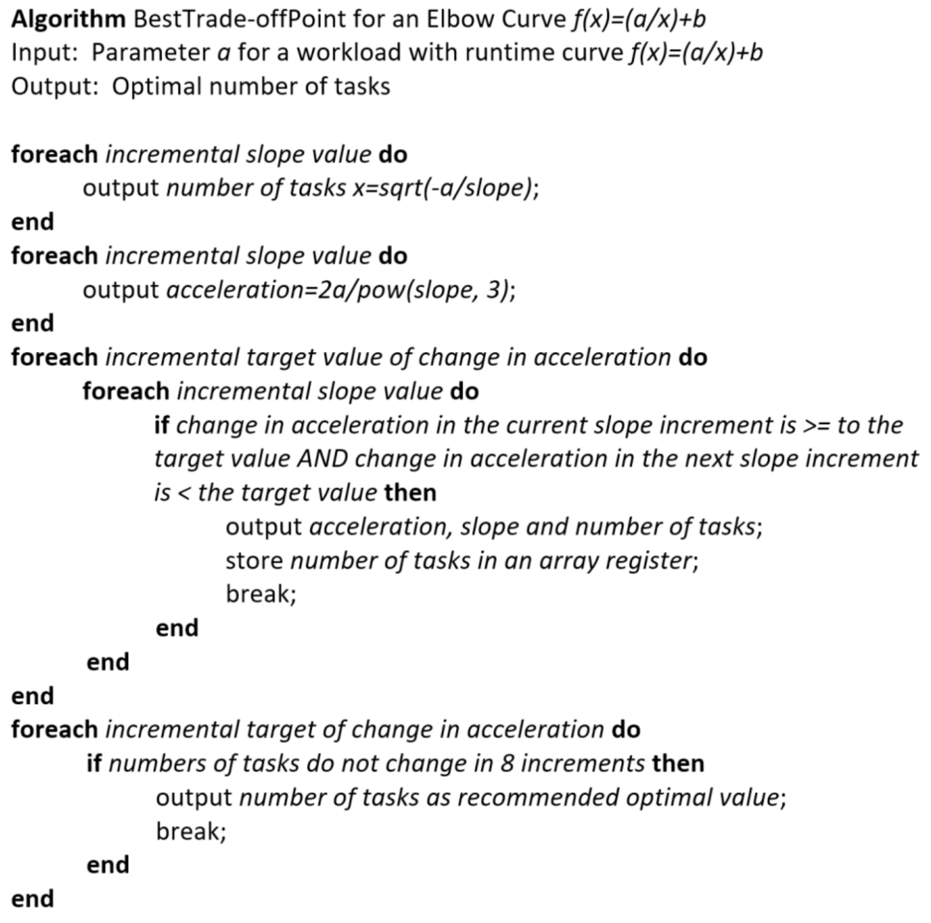
    a. The algorithm computes the number of tasks over a range of slopes from the first derivative of *f(x)=(a/x)+b* and the acceleration over a range of slopes from the second derivative of *f(x)=(a/x)+b* (Fig. 2-3).

b. The algorithm applies the Chain Rule to search for break points and major plateaus on the graphs of acceleration, slope, and task resources over a range of incremental changes in acceleration per slope increment (Fig. 2-3).

c. The algorithm extracts the exact number of tasks at the best trade-off point on the elbow curve and outputs it as recommended optimal number of tasks for a workload (Fig. 2-3).

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

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



```
Algorithm BestTrade-offPoint for an 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 tasks

foreach incremental slope value do
        output number of tasks $x=sqrt(-a/slope)$;
end
foreach incremental slope value do
        output $acceleration=2a/pow(slope, 3)$;
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 >= to the
                target value AND change in acceleration in the next slope increment
                is < the target value then
                        output acceleration, slope and number of tasks;
                        store number of tasks in an array register;
                        break;
                end
        end
end
foreach incremental target of change in acceleration do
        if numbers of tasks do not change in 8 increments then
                output number of tasks as recommended optimal value;
                break;
        end
end
```

Fig. 2 Best Trade-off Point Algorithm for an Elbow Curve (Noninverted)

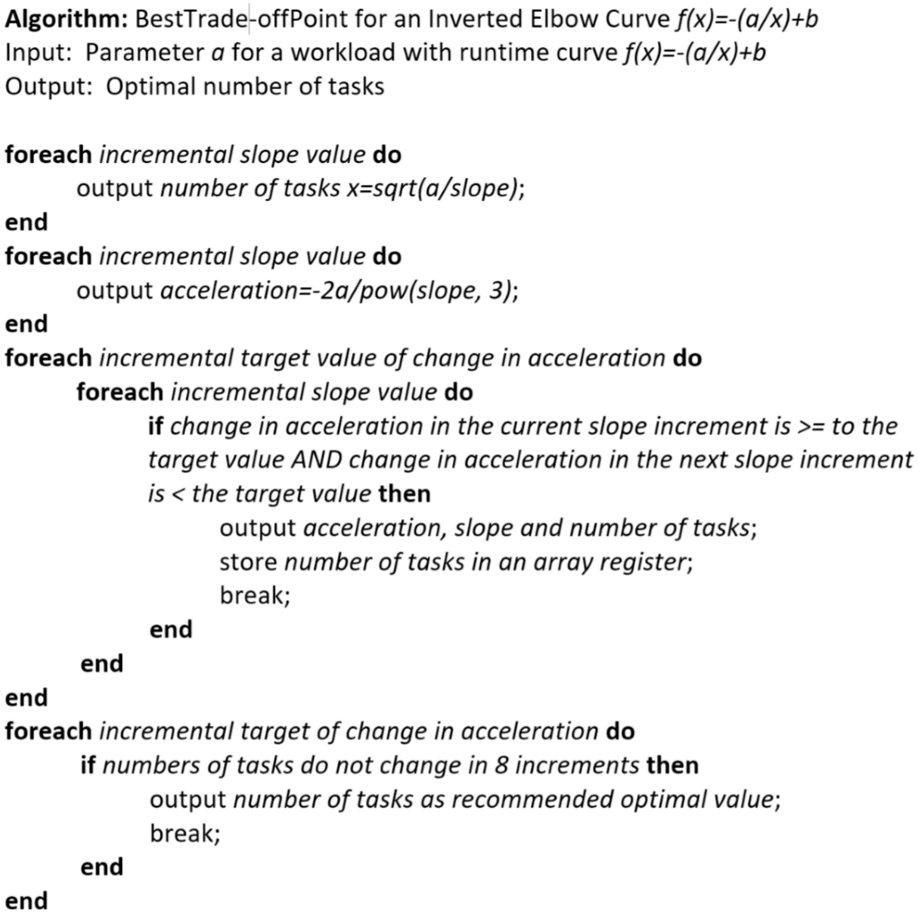
7. Use the database of resource consumption signatures to match dynamically submitted production jobs to their recommended optimal number of tasks for efficient resource provisioning.

These steps for implementing the Best Trade-off Point algorithm for efficient resource provisioning could be also used with any 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 elbow shapes, both inverted or non-inverted, for making intelligent decisions. The isoquant analysis shows various combinations of factors of production that can produce a certain amount of output. In the convex isoquant curve, the factors can be substituted for each other only up to a certain extent. The proposed method can be used to determine the sweet spot on the elbow curve which is the best trade-off point between the factors.

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. 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.

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 after 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 Fig, 3. 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*).

To illustrate another area of application of the proposed BToP method with an inverted elbow curve, we could consider a simplified example of determining the optimal travel speed for achieving the best miles-per-gallon (MPG) in fuel resource consumption on any individual fuel engine vehicle. There have been many suggestions that the travel speed for best MPG is within the range from 50 to 70 miles per hour. But there is yet a viable method for determining the exact optimal travel speed for best MPG, which varies from one car to another even on the same model. The optimal travel speed for best MPG depends on many factors and variables including but not limited to car model, engine size, car condition, engine tuning condition, tires, type of fuel, loads, route and weather condition, and so on, just to name a few. Using the proposed method and algorithm to



```
Algorithm: BestTrade-offPoint 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 tasks

foreach incremental slope value do
    output number of tasks x=sqrt(a/slope);
end
foreach incremental slope value do
    output acceleration=-2a/pow(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 >= to the
        target value AND change in acceleration in the next slope increment
        is < the target value then
            output acceleration, slope and number of tasks;
            store number of tasks in an array register;
            break;
        end
    end
end
foreach incremental target of change in acceleration do
    if numbers of tasks do not change in 8 increments then
        output number of tasks as recommended optimal value;
        break;
    end
end
```

Fig. 3 Best Trade-off Point Algorithm for an Inverted Elbow Curve

determine the Best Trade-off Point on an elbow curve of performance vs. resources, which is an inverted elbow curve of travel speed vs. fuel resources, in this case, the travel speed for best MPG could be accurately and precisely determined for every individual car.

The steps to Implement the Best Trade-off Point algorithm for efficient fuel resource provisioning in a fuel engine vehicle carrying a certain payload on a specific route and weather condition are basically the same as the ones for the application on a computer system. Following is the outline of sequential steps that a fuel engine system would perform to implement an embodiment of the proposed method.

1. Complete the tune-up of the fuel engine vehicle targeted for calibration.

2. Collect enough samples of speed performance vs. fuel resource consumption as preview data to plot the inverted elbow curve (knee curve) by running the car at different travel speeds on a specific route and weather condition and take measurements of the corresponding fuel consumption at those speeds. Each payload on a specific route and weather condition will have its own inverted elbow curve of speed performance vs fuel resources.

3. Curve-fitting the preview data to obtain the fit parameters *a* and *b* in the inverted elbow curve function $f(x)=-(a/x)+b$, where x is the number of fuel resources.

4. Input the fit parameter *a* to the Best-Trade-off-Point algorithm (Fig. 3) which is now modeled with $f(x)=-(a/x)+b$ for an inverted elbow curve, to obtain the recommended optimal number of fuel resources for a payload on a specific route and weather condition.

    a. The algorithm computes the number of fuel resources 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 of $f(x)=-(a/x)+b$.

    b. The algorithm applies the Chain Rule to search for break points and major plateaus on the graphs of acceleration, slope, and fuel resources over a range of incremental changes in acceleration per slope increment.

c. The algorithm extracts the exact number of fuel resources at the best trade-off point on the inverted elbow curve and outputs it as recommended optimal number of fuel resources for a payload on a specific route and weather condition.

5. Repeat steps 2–4 to build a database of resource consumption signatures with different payloads on a specific route and weather condition for subsequent transportation job profiling.

6. If there is any major change to step 1, repeat steps 1–5 to recalibrate the database of resource consumption signatures.

7. Use the database of resource consumption signatures to match transportation jobs to their optimal number of fuel resources for efficient resource provisioning.

The crux of the proposed Best Trade-off Point method is not tied to any specific system. The Best Trade-off Point method provides a general approach and techniques based on an algorithm with mathematical formulas to find the best trade-off point on an elbow curve which is applicable to any system relying on a trade-off curve for making good decisions. As mentioned earlier, there are quite a few of applications and systems which are characterized by an elbow curve. The potential of commercializing the proposed method is huge since it touches every area of decision making process which relies on a trade-off curve for good decision making, including but not limited to efficient resource provisioning. Although the proposed method as presented in this thesis is used to optimize resource provisioning in Hadoop MapReduce and Apache Spark for performance efficiency, which often corresponds to energy efficiency, it is not limited to that software frameworks and hardware computing system. As clearly stated, the proposed method to find the best trade-off point for efficient resource provisioning is applicable whenever there is an elbow curve which is the fundamental trade-off curve. As such, the hardware that the invention could run on depends on the type of application which the Best Trade-off Point method is implemented on. As an example, for applications with runtime vs. resource trade-off curves, the hardware could be any computing systems including multicore systems, computer cluster, grid computers, network routers, and so on. For applications with trade-off curves of horse power, speed, travel distance, payload or weight vs. resource such as fuel or electricity, the hardware could be a car engine, jet engine, boat engine or rocket. Not to mention for business applications with cost vs. quality control or delivery time, the underline hardware then becomes a manufacturing or production system.

Chapter 3

Efficient Resource Provisioning in MapReduce

3.1 Pertinent Research in Performance Efficiency of MapReduce

Several research groups have worked on the performance and energy efficiency of Hadoop MapReduce. Krish et al. [26] present a workflow scheduler for MapReduce framework that profiles the performance and energy characteristics of applications on each hardware sub-cluster in a heterogeneous cluster to improve matching application to resource while ensuring energy efficiency and performance related Service Level Agreement goals. Hartog et al. [17] suggest a MapReduce framework configuration to evaluate node power consumption status and dynamically shift work toward more energy efficient node. Leverich and Kozyrakis [29] propose modifying Hadoop to allow the scaling down of operational clusters by keeping only a small fraction of the nodes running while disabling nodes not in the covering subset to conserve power. Lang and Patel [28] use all the nodes in the Hadoop cluster to run a workload and then power down the entire cluster when there is no work as an all-in-strategy. Kaushik and Bhandarkar [25] place classified data into two logical zones of HDFS, where 26% energy consumption reduction is achieved from cold zone power management, and there is room for further energy saving in the under-utilized hot zone. Lin et al. [30] analyze and derive the job energy consumption from the job completion reliability of the general MapReduce infrastructure based on a Poisson distribution to find way to achieve energy-efficient MapReduce environment. Wang et al. [38] use a genetic algorithm with practical encoding and decoding methods, and specially designed genetic operators to support a new MapReduce energy-efficient task scheduling model. Chen et al. [8] show that for MapReduce workloads, where the work rate is proportional to the amount of resources used, improving the performance as measured by traditional metrics such as job duration is equivalent to improving the performance as measured by lower energy consumed. For most systems, decreasing energy consumption is equivalent to decreasing the finishing time.

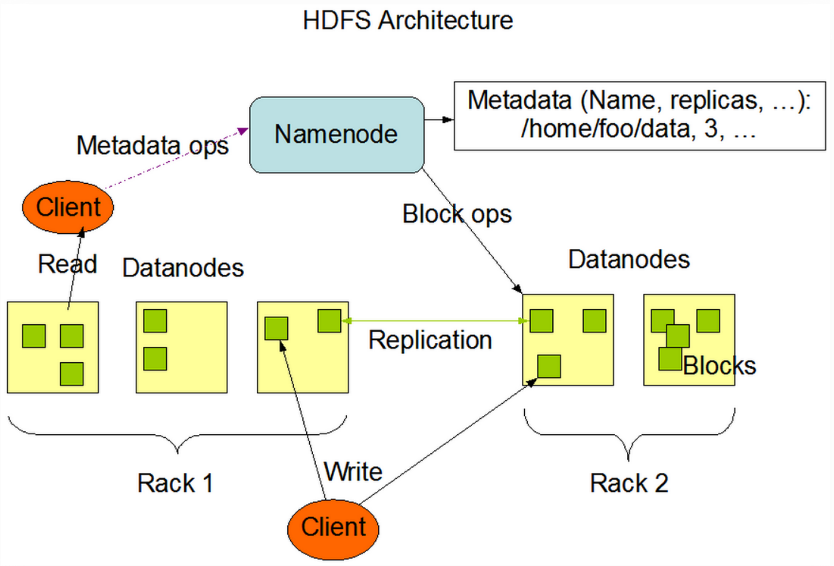
Among the above research work dedicated to improving the energy efficiency of Hadoop MapReduce, we find that Chen et al. [8]'s publication is the most closely related to our work. Chen et al. [8] suggest a way to answer the question of how many machines to allocate to a particular job by comparing energy consumption of different numbers of machines but do not provide a method to find the exact optimal number. A smaller number of machines always consumes less energy, and takes longer to finish a job unless it has far exceeded the resources required for the job. In this thesis, we present a solution for finding the best trade-off point in performance and energy efficiency. We propose a general method, formula, and algorithm for obtaining the exact optimal number of tasks for any workload running on Hadoop MapReduce, to provision for performance efficiency based on the actual preview runtime data of the cluster targeted for calibration.

3.2 Background knowledge on MapReduce

Apache Hadoop [1, 39] is an open source framework for distributed storage and processing of large sets of data on clusters of commodity hardware. Although Hadoop ecosystem includes several software packages such as HBase, Hive, Mahout, Pig, Scoop, Spark, Storm and others, the base Apache Hadoop 2.0 framework comprises only four key modules: (1) Hadoop Common which provides file systems and OS level abstractions, (2) Hadoop Distributed File System (HDFS), (3) Hadoop YARN (Yet Another Resource Negotiator) which manages computing resources in clusters and using them for scheduling of users' apps, and (4) Hadoop MapReduce engine (MR2) which implements MR programming model (Fig. 8). With the addition of YARN in Hadoop 2.0, multiple applications while sharing a common cluster resource management can now be run in parallel by new engines. Hadoop clusters can now be scaled up to a much larger configuration and support iterative processing, graph processing, stream processing, and general cluster computing all at the same time.

3.2.1 Hadoop Distributed File System (HDFS)

HDFS, which is based on Google File System (GFS), a master/slave architecture, supports large-scale data processing workloads and reliable data storage of several TB on clusters of commodity hardware (Fig. 4). It features high scalability, high availability, fault tolerance, flexible access, load balancing, tunable replication, and security. Since HDFS is designed more for batch processing rather than interactive use, it emphasizes more on high throughput of data than low latency of data access. HDFS has a simple coherency model: write-one-and-read-many access model, which supports appends and truncates only with no updates at arbitrary points. It is designed for high portability across heterogeneous hardware and software platforms.



Source: hadoop.apache.org

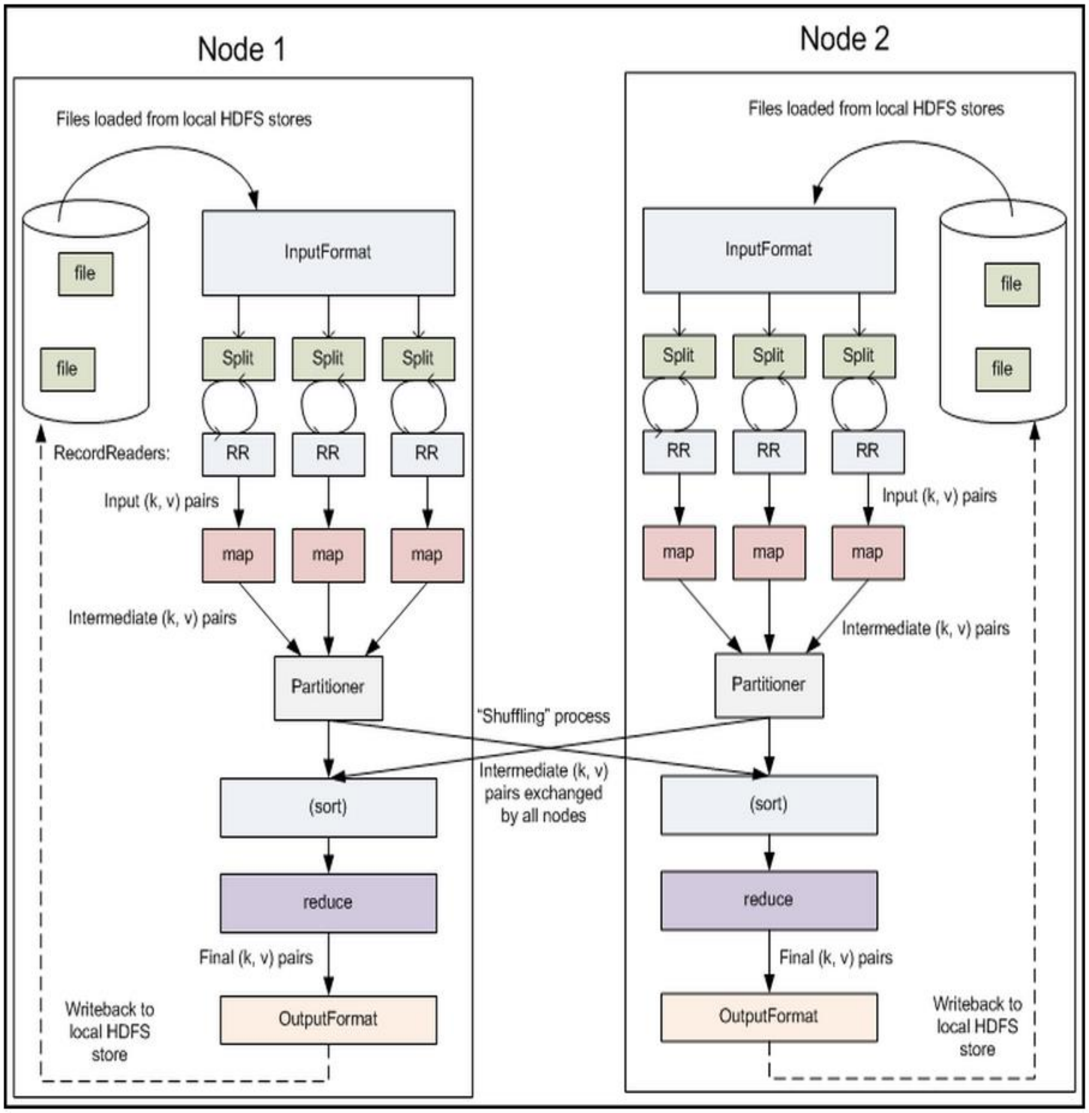
Fig. 4 Architecture of Hadoop Distributed File System (HDFS)

HDFS splits files into default blocks of 64 MB or 128 MB, which are distributed among the nodes to provide a very high aggregate bandwidth across the cluster for compute performance and data protection. There is a single master called NameNode, which coordinates access and metadata as a simple centralized management system. There is no data caching error because the NameNode stores all metadata, which include filenames and locations of each file on DataNode, in memory for fast lookup. The DataNode only stores blocks from files. NameNode makes all decisions regarding block replications. NameNode periodically receives a Heartbeat and a Blockreport from each DataNode. A secondary NameNode, running on a separate machine, periodically merges edit logs with namespace snapshot image stored on disk to prevent the edit log file from growing into a large file. In case of NameNode failure, the saved metadata can rebuild a failed primary NameNode with some data loss since the state of secondary NameNode always lags from the primary NameNode.

HDFS's block replication feature is designed to tolerate frequent component failure and is optimized for huge number of very large files on up to several thousand nodes cluster, which are mostly read and appended. HDFS minimizes global bandwidth consumption and read latency with replica locality. Nodes are chosen based on rack-aware replica placement policy first, and then storage types and policies, to improve data reliability, availability, and network bandwidth utilization.

3.2.2 MapReduce Programming Model

The MapReduce programming model uses parallel and distributed algorithm on a cluster of nodes to process large datasets, unstructured as in a file system or structured as in a database. MapReduce can take advantage of data locality by passing data to each data node within the Hadoop cluster. MapReduce also packages users' MapReduce functions as a Java ARchive (JAR) file and sends it out to each node. The JAR file operates locally on that slice of input on that data node and therefore, reduces the distance over which it must be transmitted. By executing compute at the location of data instead of having data moved to the compute location, traditional network bandwidth bottlenecks could be avoided. Moving computation to data is much more efficient than vice versa, The MapReduce framework provides scalability, security and authentication, resource management, optimized scheduling, flexibility, and high availability for a variety of applications in Big Data including but not limited to machine learning, financial analysis, genetic algorithms, natural language processing, signal processing, and simulation.


Node 1
Node 2
Files loaded from local HDFS stores
Files loaded from local HDFS stores
file
file
InputFormat
Split
Split
Split
RecordReaders:
RR
RR
RR
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Partitioner
"Shuffling" process
Intermediate (k, v) pairs exchanged by all nodes
(sort)
reduce
Final (k, v) pairs
OutputFormat
Writeback to local HDFS store

Source: Yahoo! Developer Network

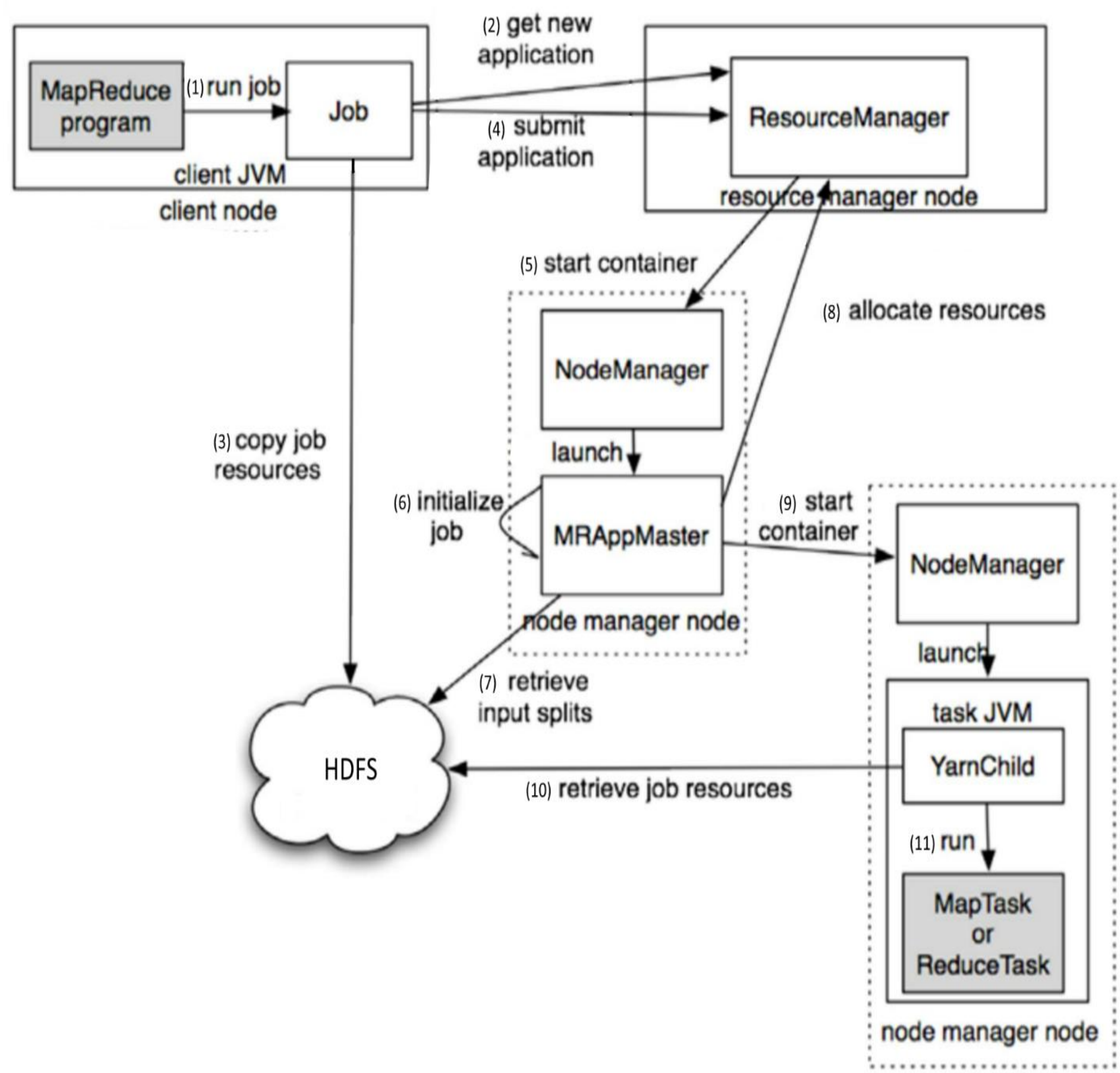
Fig. 5 MapReduce Data Flow

MapReduce consists of three phases, map, shuffle and reduce, where all values are processed independently. The reduce phase cannot start until the map phase is completely finished. At the map phase, map() functions run in parallel, creating different intermediate values from different input datasets: map(input_key, input_value) ->list <intermediate_key, intermediate_value>. At the shuffle phase after partitioning, values are exchanged by a shuffle/combine process which runs on mapper nodes as a mini reduce phase on local map output to save bandwidth before sending data to full reducer. At the reduce phase, reduce() functions, also running in parallel, aggregate all values for a specific key to a single output to generate a new list of reduced output: list<intermediate_key, intermediate_value>->list<output_key, output_value> (Fig. 5).

3.2.3 MapReduce Job Execution on YARN

YARN splits the responsibilities of job tracker and task tracker in MapReduce v.1 into four separate entities in MapReduce v.2: (1) The ResourceManager has a built-in scheduler, which allocates resources across all applications based on the applications' resource requirements. (2) The MR ApplicationMaster, which negotiates appropriate resource containers from the scheduler and tracks their progress, coordinates and manages each and every instance of MapReduce jobs executed on YARN. (3) The NodeManager, which is responsible for containers, monitors each and every node's resource usage (CPU, memory, disk, network bandwidth) within YARN. (4) The Container allocates and represents resources per node available for each specific application (Fig.6). Thus, the tasks running MapReduce job is coordinated by the MR Application Master, which creates a map task object for each split and a number of reduce task objects determined by the *mapreduce.job.reduces* property.

The sequential steps of how Hadoop runs a MapReduce job using YARN are shown in Fig. 6. Step (1), a MapReduce job is submitted to a job client. Step (2), the job client requests for a new application ID from ResourceManager. Step (3), the job client checks HDFS to see whether an output has been created for that input and copy the result from HDFS directly if it exists. Otherwise, the job client copies job resources from HDFS. Step (4), the job is submitted to ResourceManager where a Scheduler allocates resources and an Application Manager monitors progress and status of the job. Step (5), ResourceManager contacts a NodeManager to start a new container and launch a MapReduce AppMaster for the job. Step (6), MR AppMaster creates an object for bookkeeping purpose and task management. Step (7), MR AppMaster retrieves the input splits from HDFS and creates a task for each split. Step (8), MR AppMaster decides how



Source: Hadoop: The Definitive Guide by Tom White (2012)

Fig. 6 MapReduce Job Execution on YARN

to run the MapReduce task. Small jobs can be run on the same JVM on a single node as an Uber task. Large jobs request more resources to be allocated by ResourceManager which gathers information from the heartbeats of NodeManagers to consider data locality in its node allocation. Step (9), MR AppMaster contacts a NodeManager to start a new container for task execution. A YarnChild is launched to run on a separate JVM to isolate user codes from long running system deamons. Step (10), YarnChild retrieves job resources from HDFS. Step (11), YarnChild runs Map task or Reduce task. In every 3 secs, YarnChild sends a progress report to MR AppMaster which aggregates all reports and sends an update directly to the job client. Upon job completion, MR AppMaster and task containers clean up their working states, and terminate themselves to release resources.

3.3. MapReduce Resource Provisioning

In general, allocating a higher number of tasks increases parallelization, framework overhead and load balancing, and minimizes the cost of failures to smaller increments of resources. But too many or too few tasks, whether mappers or reducers, are both detrimental for job performance. When the number of tasks is too large potentially causing resource contention and overall performance degradation, the overhead time spent by all task resources continues to grow while there is no further reduction in job runtime with the gradual increase in number of allocated tasks. When the number of tasks is too little for a workload, the job runtime is extremely high due to resource insufficiency (Fig. 7). Our goal is to find the best trade-off point between runtime and task resources to provision for optimal performance and energy efficiency.

There are some prior works on MapReduce resource provisioning to achieve certain application performance goals and Service Level Objectives (SLOs) which could be referenced when using our method for obtaining optimal task resources for energy efficient computing. Babu [6] suggests different techniques for automatic setting of job configuration parameters for MapReduce programs, including dynamic profiling, but acknowledges that this is an inherently difficult research and engineering challenge when the properties of the actual job being processed, its input data, and resource allocation are not known. Herodotou et al. [18] introduce the Elasticizer system to configure the right cluster size matching a workload's performance needs by using an automated technique based on a mix of job profiling and simulation. Verma et al. [37] generate a set of resource provisioning options to meet given SLOs by applying scaling rules to the job past executions or sampled executions from a given application on the set of small input datasets. Kambatla et al. [22] propose a brute force job provisioning approach by analyzing and comparing the resource consumption of the application at hand with a database of similar resource consumption signatures of other applications to calculate the optimum configuration.

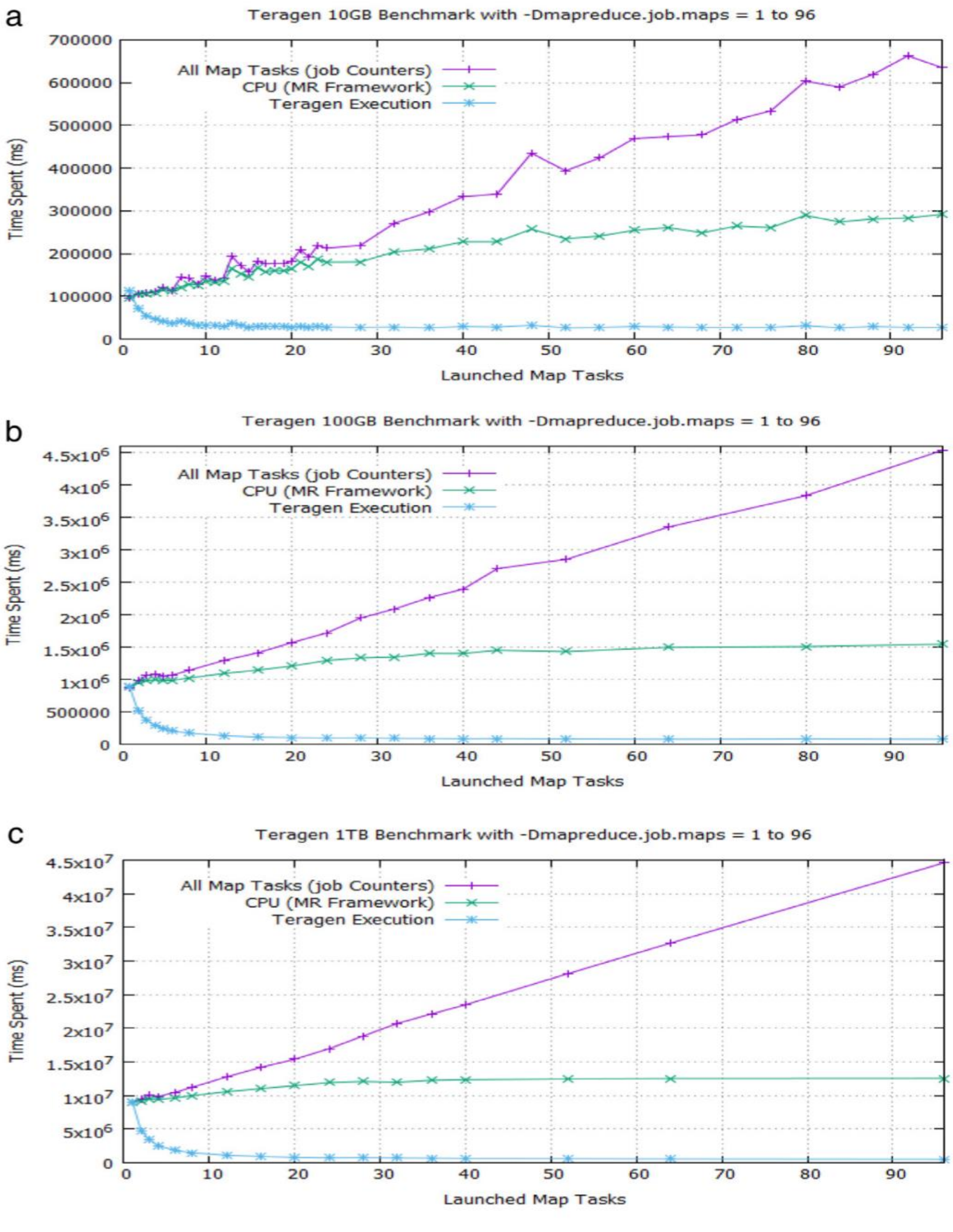


Fig. 7 Graphs of time spent by all map tasks, CPU, and Teragen execution versus number of launched map tasks. The runtime elbow curves of Teragen (a) 10 GB, (b) 100 GB and (c) 1 TB workloads plotted at different y-axis scales all appear to have the best trade-off points for performance efficiency at around 10 map tasks. But that is refuted by our algorithm as a visual misperception of different granularities at low magnification.

For the greater part, these prior research papers on resource provisioning for MapReduce v.1 are still applicable to MapReduce v.2. However, MapReduce v.2 is considerably different than MapReduce v.1, where there are pre-configured static slots for map and reduce tasks, which are inflexible and often leads to an under-utilization of resources. In YARN, the job tracker's role of the previous MapReduce v.1 is now handled by a separate resource manager and history server to improve scalability. The NodeManager in MapReduce v.2, which manages resources and deployment on a node, is now responsible for launching containers. Each container can store a map or reduce task. MapReduce v.2 running on YARN is more scalable with resource utilization configured in terms of physical RAM limit, virtual memory and JVM heap size limit for each task. These improvements allow Hadoop to share resources dynamically between applications in a finer-grained, more practical and scalable resource configuration for better provisioning and cluster utilization. Along the lines proposed by these prior papers for resource provisioning by job profiles, our research paper further provides an innovative method, formula and algorithm to eliminate the guesswork, and accurately identify the optimal numbers of task resources for different workloads to achieve performance efficiency on any specific Hadoop cluster while minimizing any strenuous brute force.

Obtaining the optimal number of mappers and reducers for each job has been a challenge for Hadoop MapReduce users since there are lots of variables involved in balancing computing resources with network transfer bandwidth and disk reads. There are more than 180 parameters specified to control the behavior of a MapReduce job in Hadoop and the settings of more than 25 of these parameters can have significant impact on job performance [6, 22]. However, the optimal number of tasks for a job depends not only on the settings of various parameters and metrics for fine tuning Hadoop cluster performance but also on several other factors including but not limited to the type of application, dataset size and structure, cluster hardware specifications, system setup and configuration, and output buffer size. Therefore, the most practical method to indirectly take all those factors into account is to compute the optimal number of tasks from the actual sampled runtime data of the target cluster.

The number of maps needed for a certain job is usually decided by the number of blocks in the job inputs, which varies with the HDFS block size. The current default HDFS block size is 128 MB, an increase from the previous version, which was 64 MB. In some cases, capitalizing on data locality to enlarge the HDFS block size up to 512 MB to store a large input file can reduce the runtime for I/O bound jobs. On the other hand, when mappers are more CPU bound and less I/O bound, reducing the HDFS block size can improve the utilization of computing resources in the cluster. Hence, the total number of mappers running for a job depends on the number of input splits of the data. According to Hadoop Wiki, the right level of parallelism for maps seems to be around 10–100 maps/node, although it could be taken up to 300 or so for very CPU-light map tasks [16]. Significantly, the number of reducers at the aggregation step is more difficult to estimate since it is not easy to ascertain any spill of intermediate outputs to memory buffer and/or to disk for different workloads. Although there are currently three popular rules of thumb widely circulated in industry for deciding on the optimal number of reducers for a job, none of them provide an accurate and verifiable number of task resources for certain workload as shown in Fig. 17 in Section 3.6.

3.4. Best Trade-off Point (BToP) Algorithm for Optimal Resource Provisioning

We have developed an algorithm (Fig. 2-3) to search for the best trade-off points on the elbow curve of runtime versus number of launched tasks to overcome the uncertainty of all variables involved in finding the optimal number of tasks for a job to run in a specific Hadoop cluster. Before applying the algorithm, Hadoop users should first get some sampled executions from their target production system as reference points sufficient to plot a smooth elbow curve for each workload.

From the shape of the elbow curve of runtime versus task resources, we intuitively recognize its graph function $f(x)=(a/x)+b$, which is confirmed by curve-fitting the preview data to obtain the fit parameters a and b. Using the fit parameter a as input, our program computes the number of tasks over a range of slopes from the first derivative and the acceleration over a range of slopes from the second derivative. Applying the Chain rule to our search algorithm for break points and major plateaus on the graphs of acceleration, slope, and task resources over a range of incremental changes in acceleration per slope increment, our program extracts the exact number of tasks at the best trade-off point on the elbow curve and outputs it as recommended optimal number of tasks for a workload (Fig. 13-14 in Section 3.5.3).

This preview method, as job profiling for optimization of task resource provisioning, should work out well in any production environment where most of the jobs frequently submitted are of the same type of applications combined with different sizes of dataset. Hadoop users only need to calibrate the optimal numbers of tasks for each different workload in their production system once to build up a table of signatures and use them for all equivalent jobs. However, if there are subsequent changes made to the cluster's system architecture, hardware setup, and configuration, a recalibration for a new set of optimal number of task resources might be necessary to maintain accuracy and precision. Once a database of signatures has been established, dynamically submitted jobs with different workloads can be quickly matched to their recommended optimal resource values for allocation using nested for-loops or equivalent structure to find resembling applications and datasets. The performance of task resources should be predictable through the job profiling of the same identical cluster-based system.

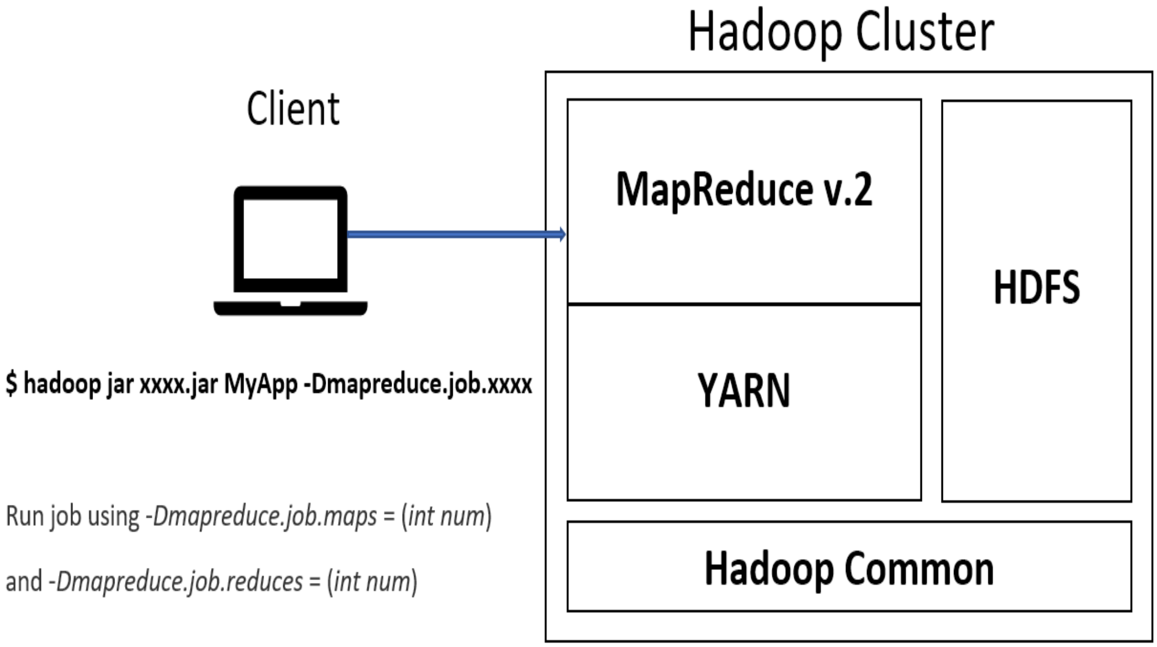
Therefore, it is possible to provide a single and general approach for automatic provisioning based on each specific system and application. However, users must establish a database of resource utilization signatures corresponding to workloads for every different application with various sizes of input datasets in advance. This approach relying on behavior replication is best suitable for production environment with repetitive workloads corresponding to the values of identical characteristics within the range of signatures pre-computed during a preview stage. It may be difficult and far less accurate to generally provision for a class of applications due to the diversified nature of MapReduce applications. Chen and Ganapathi et al. [7], in their development of an empirical workload model using production workload traces from Facebook and Yahoo to generate and replay synthetic workloads, acknowledge that per-workload performance measurements are necessary, and using proxy datasets and map/reduce functions can alter performance behavior considerably. In order to avoid recalibration of their workload model upon any change in the input data, map/reduce function code, or the underlying hardware/software system, Chen and Ganapathi et al. [7] exclude system characteristics and system behavior from the workload description. Their method with replay mechanisms, which yield some useful insights by enabling performance comparisons across various system and workload changes, is in contrast with our general approach, which emphasizes on the accuracy of optimal resource provisioning for each application running on a specific system.

3.5 Design, Analysis, and Implementation of BToP Algorithm

3.5.1 Experimental background

To illustrate our method for obtaining the optimal number of task resources for different workloads, we use the Teragen and Terasort components of the Terasort benchmark test, which is part of the open source Apache Hadoop distribution, to experiment with 10 GB, 100 GB and 1 TB datasets. The benchmark tests are performed on a 24-node homogeneous Hadoop cluster, with two racks of 12 nodes each, running Cloudera CDH-5.2 YARN (MapReduce v.2). The NameNodes are VM (virtual machines) of 4 cores and 24 GB of RAM each running on Intel Xeon E5-2690 physical hosts of 8 cores and 16 threads with 2.9 GHz base frequency and 3.8 GHz max turbo frequency, and Thermal Design Power (TDP) of 135 W. The DataNodes/NodeManagers are physical system running Intel Xeon E3-1240 v.3 CPUs with 3.4 GHz base frequency and 3.8 GHz max turbo frequency, and TDP of 80 W. Each NodeManager has 4 cores, 8 threads, 32 GB of RAM, two 6 TB hard disks and 1Gbit network bandwidth. All nodes are connected to a switch with a backplane speed of 48 Gbps.

To sample executions of the Hadoop cluster under test, we use the -*Dmapreduce.job.maps* = (*int num*) and -*Dmapreduce.job.reduces* = (*int num*) as a hint to the InputFormat to allocate the number of mappers and reducers during command line execution of JAR instead of setting the number of tasks in the code using the JobConf's *conf.setNumMapTasks (int num)* and *conf.setNumReduceTasks (int num).* For Teragen, which uses MapReduce programming engine to break up the data to be sorted using a random sequence, we generate 10 GB, 100 GB and 1 TB of data with -*Dmapreduce.job.maps* set equal to a few reference points between 1 and 96. For Terasort, which uses MapReduce programming engine to sample and sort the data created by Teragen, we sort 10 GB, 100 GB and 1 TB of data with -*Dmapreduce.job.reduces* set equal to a few reference points between 1 and 96 (Fig. 8). We observe MapReduce’s behaviors in terms of total time spent by all map tasks, total time spent by all reduce tasks, CPU time spent by MapReduce framework, and the job execution time to develop a general formula for obtaining the optimal number of tasks for efficient use of available computing resources (Fig. 9, Fig. 7, and Fig. 10).

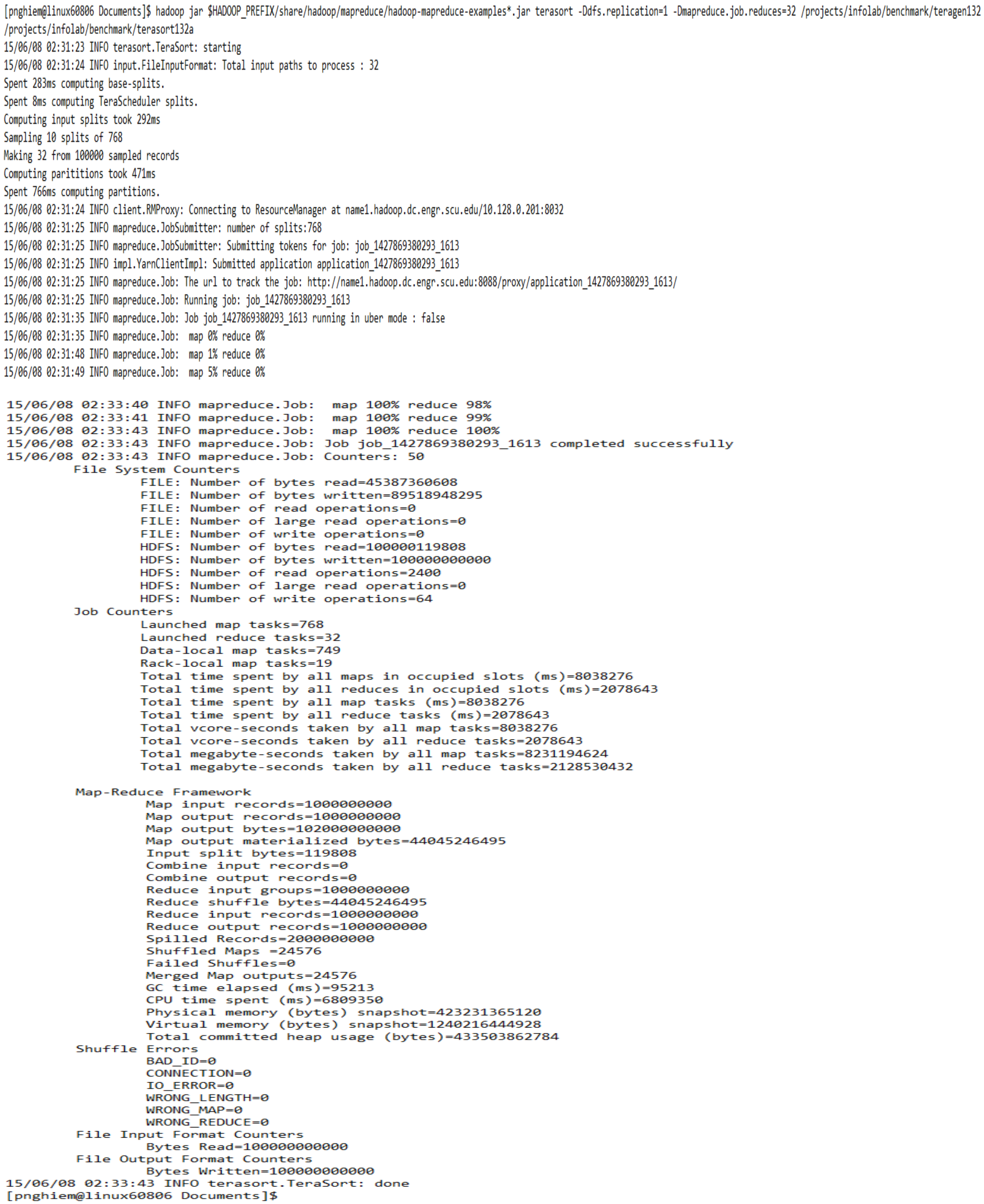


```
$ hadoop jar xxxx.jar MyApp -Dmapreduce.job.xxxx
```

Run job using -*Dmapreduce.job.maps* = (*int num*)

and -*Dmapreduce.job.reduces* = (*int num*)

Fig. 8   Allocating Task Resources to Sample Executions on the Target Hadoop Cluster.



```
[pnghiem@linux60806 Documents]$ hadoop jar $HADOOP_PREFIX/share/hadoop/mapreduce/hadoop-mapreduce-examples*.jar terasort -Ddfs.replication=1 -Dmapreduce.job.reduces=32 /projects/infolab/benchmark/teragen132
/projects/infolab/benchmark/terasort132a
15/06/08 02:31:23 INFO terasort.TeraSort: starting
15/06/08 02:31:24 INFO input.FileInputFormat: Total input paths to process : 32
Spent 283ms computing base-splits.
Spent 8ms computing TeraScheduler splits.
Computing input splits took 292ms
Sampling 10 splits of 768
Making 32 from 100000 sampled records
Computing paritions took 471ms
Spent 766ms computing partitions.
15/06/08 02:31:24 INFO client.RMProxy: Connecting to ResourceManager at name1.hadoop.dc.engr.scu.edu/10.128.0.201:8032
15/06/08 02:31:25 INFO mapreduce.JobSubmitter: number of splits:768
15/06/08 02:31:25 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1427869380293_1613
15/06/08 02:31:25 INFO impl.YarnClientImpl: Submitted application application_1427869380293_1613
15/06/08 02:31:25 INFO mapreduce.Job: The url to track the job: http://name1.hadoop.dc.engr.scu.edu:8088/proxy/application_1427869380293_1613/
15/06/08 02:31:25 INFO mapreduce.Job: Running job: job_1427869380293_1613
15/06/08 02:31:35 INFO mapreduce.Job: Job job_1427869380293_1613 running in uber mode : false
15/06/08 02:31:35 INFO mapreduce.Job:  map 0% reduce 0%
15/06/08 02:31:48 INFO mapreduce.Job:  map 1% reduce 0%
15/06/08 02:31:49 INFO mapreduce.Job:  map 5% reduce 0%

15/06/08 02:33:40 INFO mapreduce.Job:  map 100% reduce 98%
15/06/08 02:33:41 INFO mapreduce.Job:  map 100% reduce 99%
15/06/08 02:33:43 INFO mapreduce.Job:  map 100% reduce 100%
15/06/08 02:33:43 INFO mapreduce.Job: Job job_1427869380293_1613 completed successfully
15/06/08 02:33:43 INFO mapreduce.Job: Counters: 50
        File System Counters
                FILE: Number of bytes read=45387360608
                FILE: Number of bytes written=89518948295
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=100000119808
                HDFS: Number of bytes written=100000000000
                HDFS: Number of read operations=2400
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=64
        Job Counters
                Launched map tasks=768
                Launched reduce tasks=32
                Data-local map tasks=749
                Rack-local map tasks=19
                Total time spent by all maps in occupied slots (ms)=8038276
                Total time spent by all reduces in occupied slots (ms)=2078643
                Total time spent by all map tasks (ms)=8038276
                Total time spent by all reduce tasks (ms)=2078643
                Total vcore-seconds taken by all map tasks=8038276
                Total vcore-seconds taken by all reduce tasks=2078643
                Total megabyte-seconds taken by all map tasks=8231194624
                Total megabyte-seconds taken by all reduce tasks=2128530432

        Map-Reduce Framework
                Map input records=1000000000
                Map output records=1000000000
                Map output bytes=102000000000
                Map output materialized bytes=44045246495
                Input split bytes=119808
                Combine input records=0
                Combine output records=0
                Reduce input groups=1000000000
                Reduce shuffle bytes=44045246495
                Reduce input records=1000000000
                Reduce output records=1000000000
                Spilled Records=2000000000
                Shuffled Maps =24576
                Failed Shuffles=0
                Merged Map outputs=24576
                GC time elapsed (ms)=95213
                CPU time spent (ms)=6809350
                Physical memory (bytes) snapshot=423231365120
                Virtual memory (bytes) snapshot=1240216444928
                Total committed heap usage (bytes)=433503862784
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=100000000000
        File Output Format Counters
                Bytes Written=100000000000
15/06/08 02:33:43 INFO terasort.TeraSort: done
[pnghiem@linux60806 Documents]$
```

Fig. 9 Test Run of Terasort 100GB with –Dmapreduce.job.reduces = 32

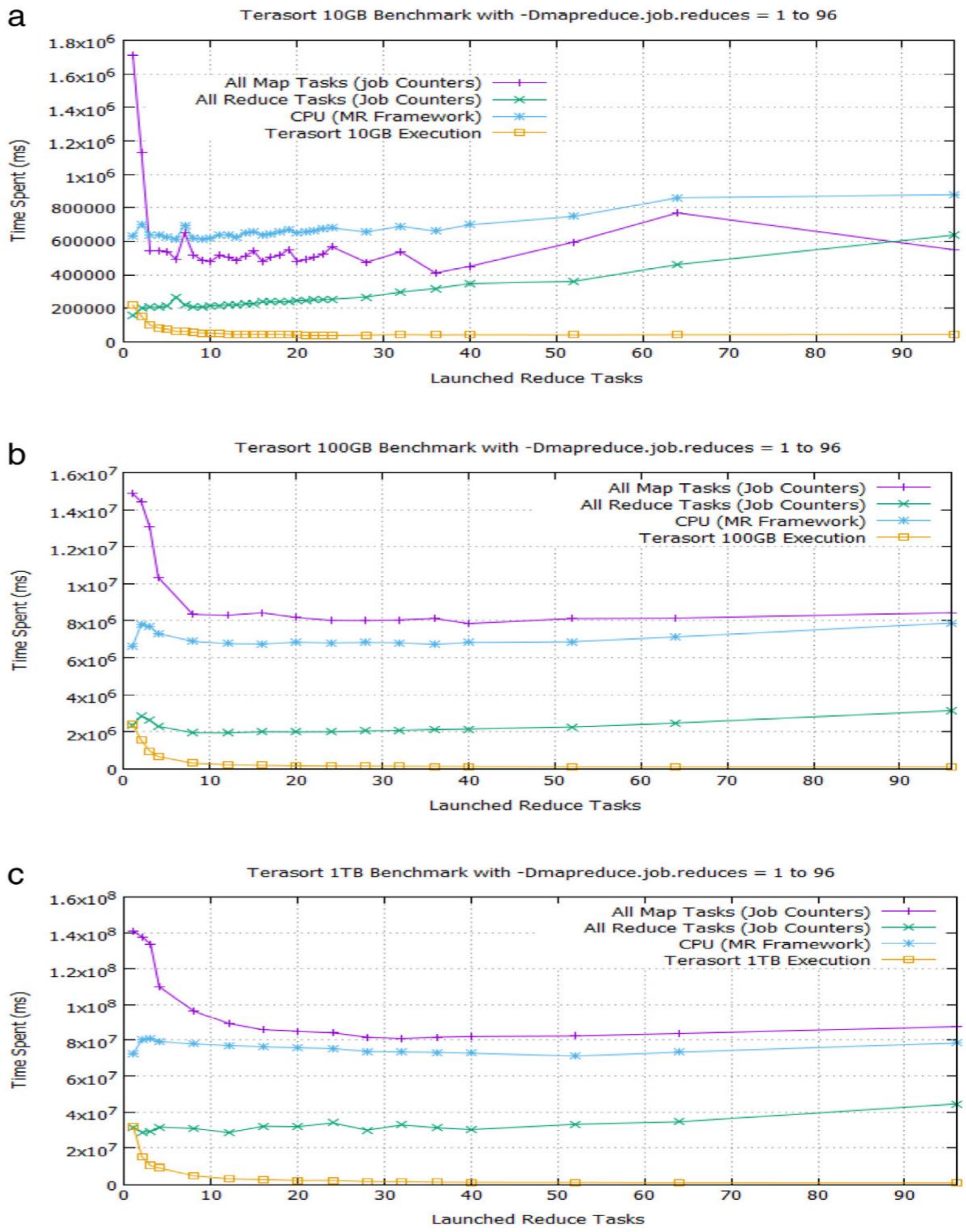


Fig. 10 Graphs of time spent by all map tasks, all reduce tasks, CPU, and Terasort execution versus number of launched reduce tasks. The runtime elbow curves of Terasort (a) 10 GB, (b) 100 GB and (c) 1 TB workloads plotted at different y-axis scales all appear to have the best trade-off points for performance efficiency at around 10 reduce tasks. But that is disproved by our algorithm as a visual misperception of different granularities at low magnification.

3.5.2 Preview data

Although we performed thorough benchmark tests at numerous data points in our experiment, sampling around over a dozen points, which cover the whole elbow curve, will be sufficient to compute the target optimal task resource values. To get a little smoother graph, increase the number of points for the theoretical curve. Since the graphs of both Teragen and Terasort preview data are plotted at different vertical scales, where the 100 GB and 1 TB plots are around 10 to 100 times lower in magnification than the 10 GB plot, respectively (Fig. 7 and Fig. 10), it appears at first glance that there is no further significant improvement in runtime at the bottom of the elbow curves starting from around 10 launched map tasks and up for all three workloads. But that is a visual misperception of different granularities at low magnification since our algorithm shows that the best trade-off points are actually located at higher numbers of tasks, especially for large workloads.

In both component benchmark tests (Fig. 7 and Fig. 10), the CPU time spent by MapReduce framework increases with the number of task resources since there is more framework overhead. There is no plot of CPU time spent on reduce tasks in Teragen since it only breaks up the data to be sorted by Terasort and does not do any aggregation. For Terasort, we are only concerned about the time spent by all reduce tasks. We let mappers be allocated by MapReduce in Terasort based on the number of blocks in the input dataset previously generated by Teragen. The number of mappers for a given workload is driven by the number of input splits, and not by the *-Dmapreduce.job.maps* parameter set at the command line JAR execution. For each input split, a map task is spawned by MapReduce framework. Thus, 80 mappers are spawned from 10 GB / 128 MB = $10*1024$ MB/128 MB = 80 input splits for Terasort 10 GB, and that number increases to 800 and 8192 mappers for Terasort 100 GB and 1 TB, respectively.

As expected, the job execution time increases with larger workload and decreases with a higher number of launched tasks. However, assigning more tasks than necessary for a job will result in waste of computing resources since the reduction in execution time quickly decreases and becomes insignificant after the needed task resource value has been reached.

3.5.3 Process for Ascertaining Optimal Number of Tasks

The best trade-off point on the runtime elbow curve should be the location where no further significant decrease in execution time could be obtained by continuing to increase the number of launched tasks. Since the rate of descending of the execution time 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 has diminished close to almost flat. To find the slope, we take the derivative of the polynomial function

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

where x is the number of launched map tasks and launched reduced tasks for Teragen and Terasort, respectively. 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.

$$f'\ (x) = \lim_{\Delta x \to 0} (f\ (x + \Delta x) - f\ (x))\ /\Delta x. \qquad (2)$$

From (1),

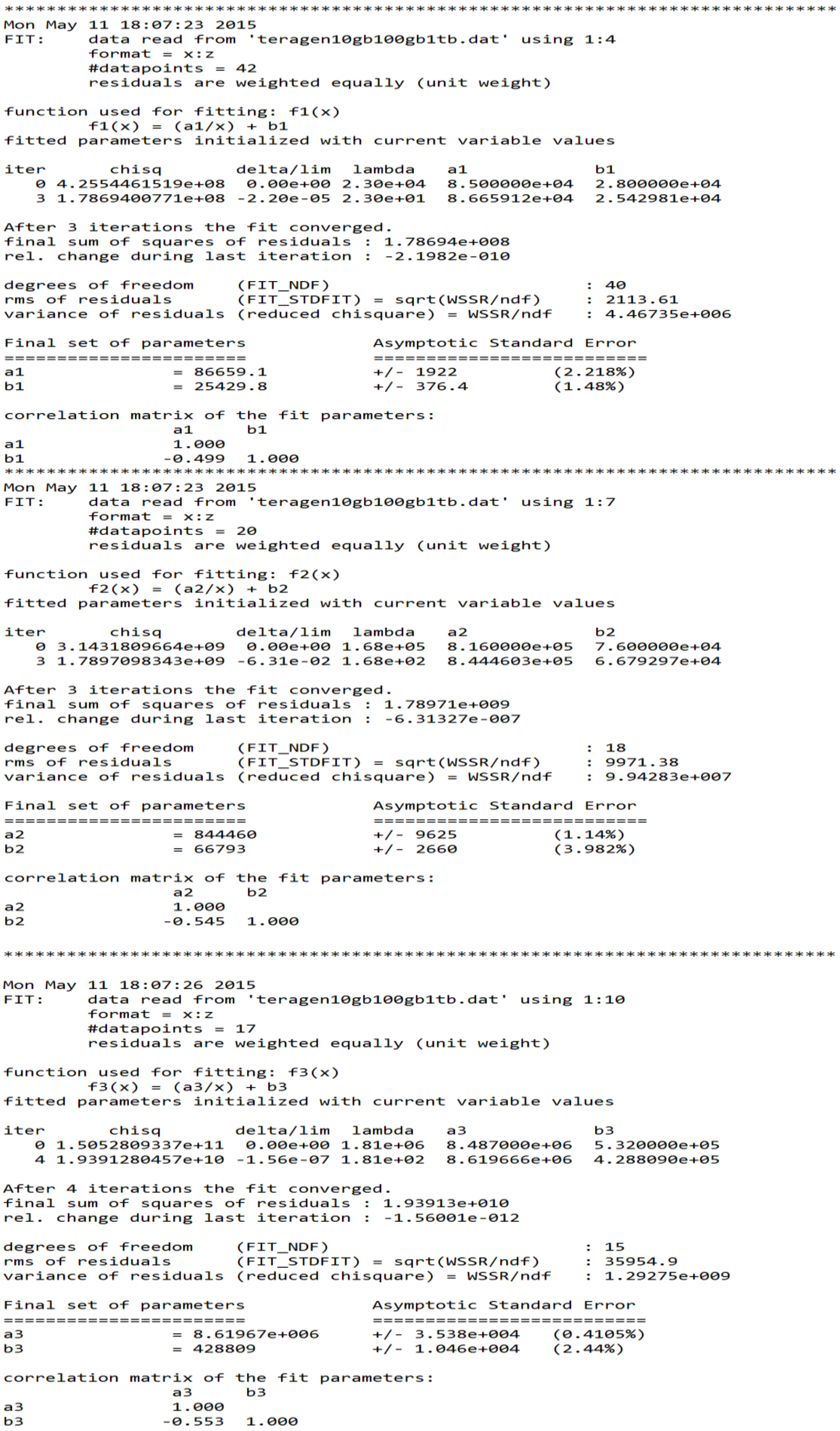
$$\mathrm{f}'(\mathrm{x}) = -\mathrm{a}\mathrm{x}^{-2} \qquad (3)$$

and therefore,

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

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

Using GNUplot to curve-fit the preview data points, we obtain the fit parameters a and b of the graph function $f(x)=(a/x)+b$ (Fig. 11). 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-Neuton algorithm since, in many cases, it could find a solution even if it starts very far off the final minimum. The GNUplot



```
*******************************************************************************
Mon May 11 18:07:23 2015
FIT:    data read from 'teragen10gb100gb1tb.dat' using 1:4
        format = x:z
        #datapoints = 42
        residuals are weighted equally (unit weight)

function used for fitting: f1(x)
        f1(x) = (a1/x) + b1
fitted parameters initialized with current variable values

iter      chisq       delta/lim  lambda   a1            b1
   0 4.2554461519e+08  0.00e+00 2.30e+04  8.500000e+04  2.800000e+04
   3 1.7869400771e+08 -2.20e-05 2.30e+01  8.665912e+04  2.542981e+04

After 3 iterations the fit converged.
final sum of squares of residuals : 1.78694e+008
rel. change during last iteration : -2.1982e-010

degrees of freedom    (FIT_NDF)                        : 40
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 2113.61
variance of residuals (reduced chisquare) = WSSR/ndf   : 4.46735e+006

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a1              = 86659.1          +/- 1922         (2.218%)
b1              = 25429.8          +/- 376.4        (1.48%)

correlation matrix of the fit parameters:
                a1     b1
a1              1.000
b1             -0.499  1.000
*******************************************************************************
Mon May 11 18:07:23 2015
FIT:    data read from 'teragen10gb100gb1tb.dat' using 1:7
        format = x:z
        #datapoints = 20
        residuals are weighted equally (unit weight)

function used for fitting: f2(x)
        f2(x) = (a2/x) + b2
fitted parameters initialized with current variable values

iter      chisq       delta/lim  lambda   a2            b2
   0 3.1431809664e+09  0.00e+00 1.68e+05  8.160000e+05  7.600000e+04
   3 1.7897098343e+09 -6.31e-02 1.68e+02  8.444603e+05  6.679297e+04

After 3 iterations the fit converged.
final sum of squares of residuals : 1.78971e+009
rel. change during last iteration : -6.31327e-007

degrees of freedom    (FIT_NDF)                        : 18
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 9971.38
variance of residuals (reduced chisquare) = WSSR/ndf   : 9.94283e+007

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a2              = 844460           +/- 9625         (1.14%)
b2              = 66793            +/- 2660         (3.982%)

correlation matrix of the fit parameters:
                a2     b2
a2              1.000
b2             -0.545  1.000

*******************************************************************************

Mon May 11 18:07:26 2015
FIT:    data read from 'teragen10gb100gb1tb.dat' using 1:10
        format = x:z
        #datapoints = 17
        residuals are weighted equally (unit weight)

function used for fitting: f3(x)
        f3(x) = (a3/x) + b3
fitted parameters initialized with current variable values

iter      chisq       delta/lim  lambda   a3            b3
   0 1.5052809337e+11  0.00e+00 1.81e+06  8.487000e+06  5.320000e+05
   4 1.9391280457e+10 -1.56e-07 1.81e+02  8.619666e+06  4.288090e+05

After 4 iterations the fit converged.
final sum of squares of residuals : 1.93913e+010
rel. change during last iteration : -1.56001e-012

degrees of freedom    (FIT_NDF)                        : 15
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 35954.9
variance of residuals (reduced chisquare) = WSSR/ndf   : 1.29275e+009

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a3              = 8.61967e+006     +/- 3.538e+004   (0.4105%)
b3              = 428809           +/- 1.046e+004   (2.44%)

correlation matrix of the fit parameters:
                a3     b3
a3              1.000
b3             -0.553  1.000
```

Fig. 11 Fitting Teragen preview data using Levenberg–Marquardt Algorithm (LMA), aka the damped least-squares (DLS) method.

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".

Using the obtained fit parameter *a*, we then plot the three fitted elbow curves of execution time versus launched task resources for Teragen and Terasort 10 GB, 100 GB and 1 TB workloads (Fig. 12).

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 task resources:

$$f''(x)=f'(f'(x)) \tag{5}$$

$$= \lim_{\Delta x \to 0} \frac{[(f\ (x+\Delta x)-f\ (x))\ /\Delta x]-[(f\ (x)-f(x-\Delta x))\ /\Delta x]}{\Delta x} \tag{6}$$

$$= \lim_{\Delta x \to 0} \frac{(f\ (x+\Delta x)-2f\ (x)+f\ (x-\Delta x))}{\Delta x^2} \tag{7}$$

as the second symmetric derivative.

From (2),

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

Our algorithm finds the optimal number of tasks recommended for a workload by locating the best trade-off point at the bottom of the elbow curve where assigning more task resources no longer significantly reduces the job execution time and therefore, reduces the overall system efficiency in resource utilization and energy consumption. Taking the

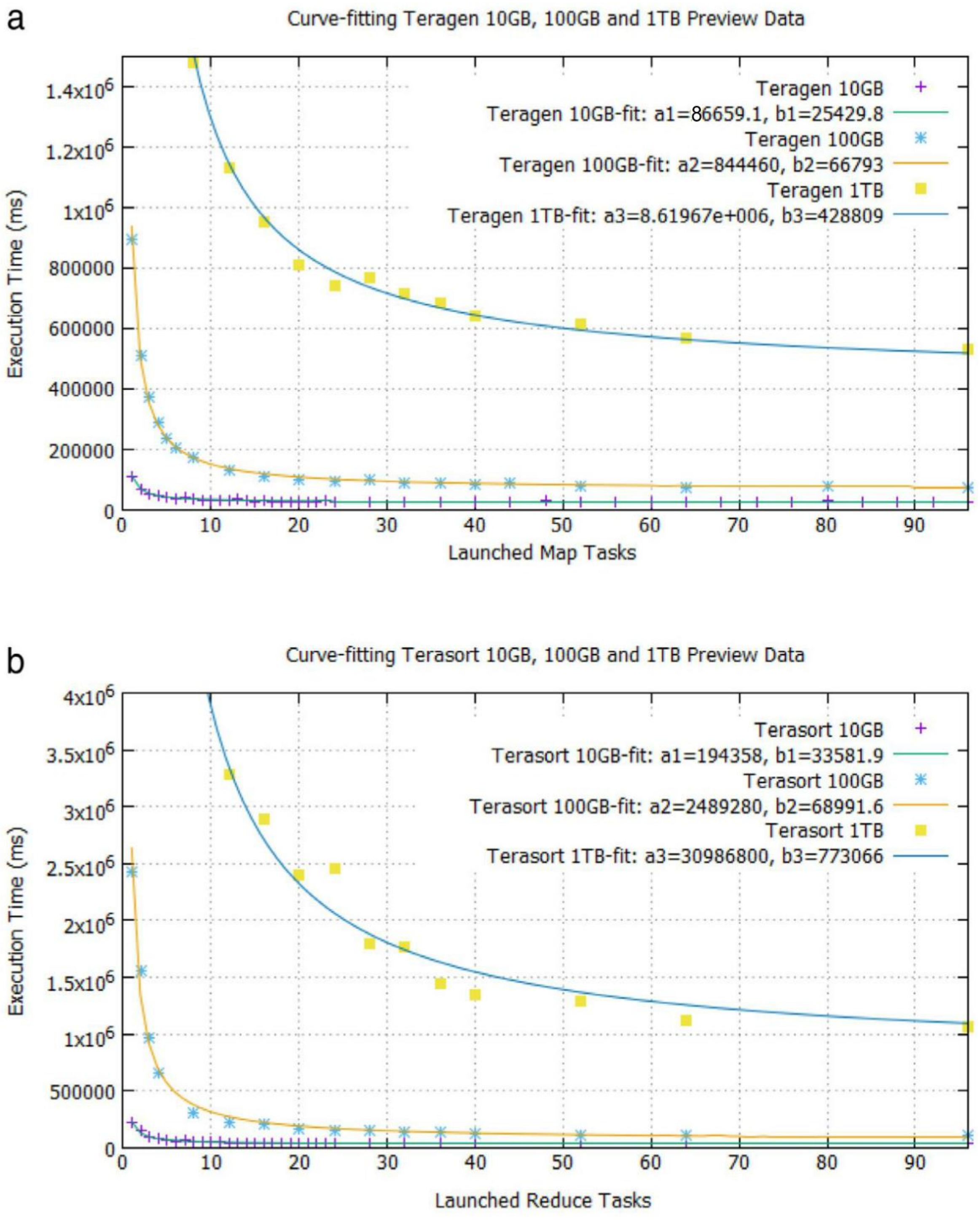


Fig. 12 Fitted runtime elbow curves of (a) Teragen and (b) Terasort 10 GB, 100 GB, and 1 TB workloads versus number of launched map/reduce tasks, and their fit parameters a and b in the graph function $f(x)=(a/x)+b$.

parameter a in $f(x)=(a/x)+b$ as input, our program computes and tabulates the number of tasks over a range of slopes from −0.25 to −39.25 for $x = \sqrt{-a/slope}$, and the acceleration over a range of slopes from −0.25 to −39.25 for $f''(x) = 2a{*}slope^{-3}$.

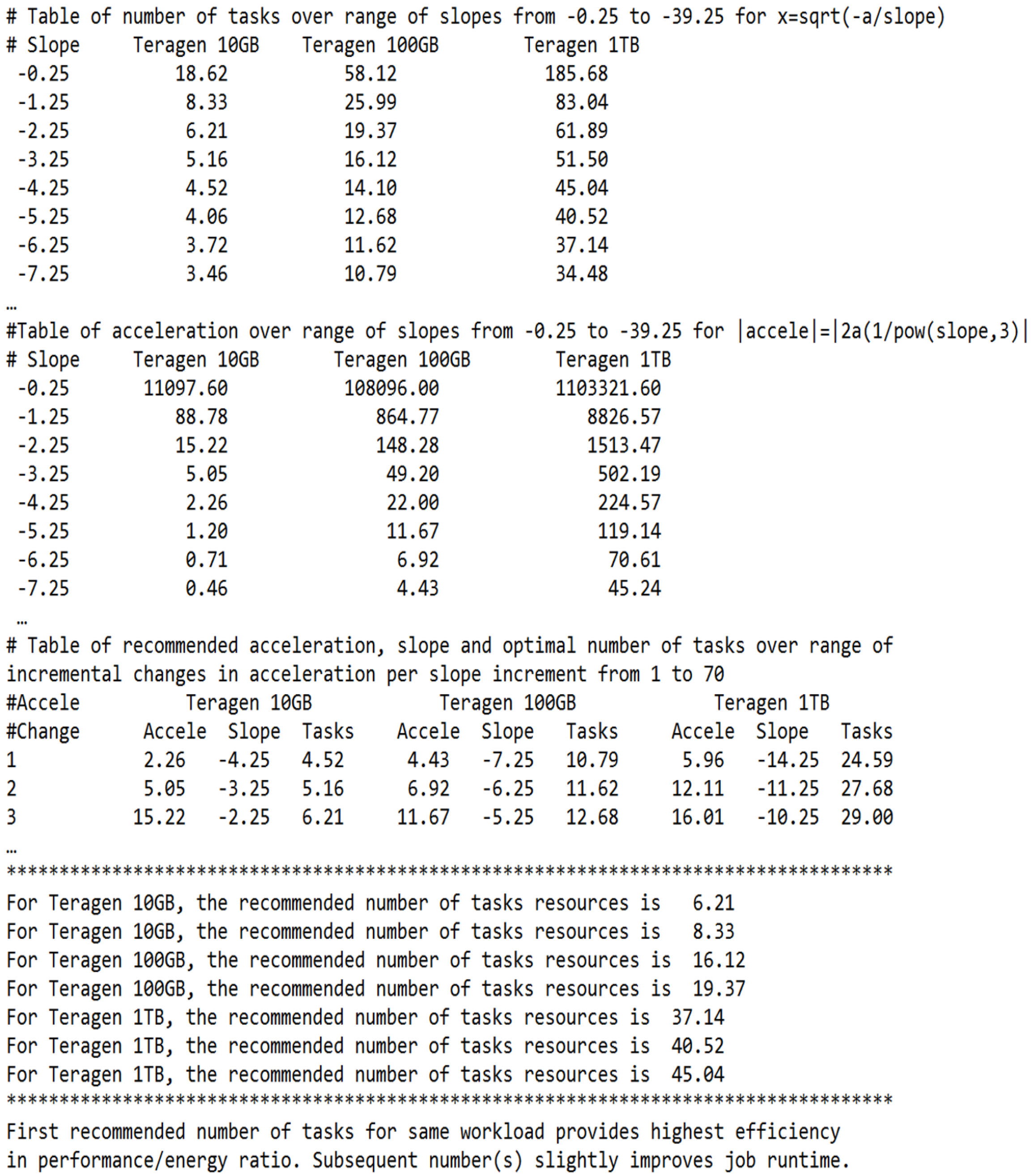
Applying the Chain Rule

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}, \tag{9}$$

the rate of change in acceleration with respect to tasks is

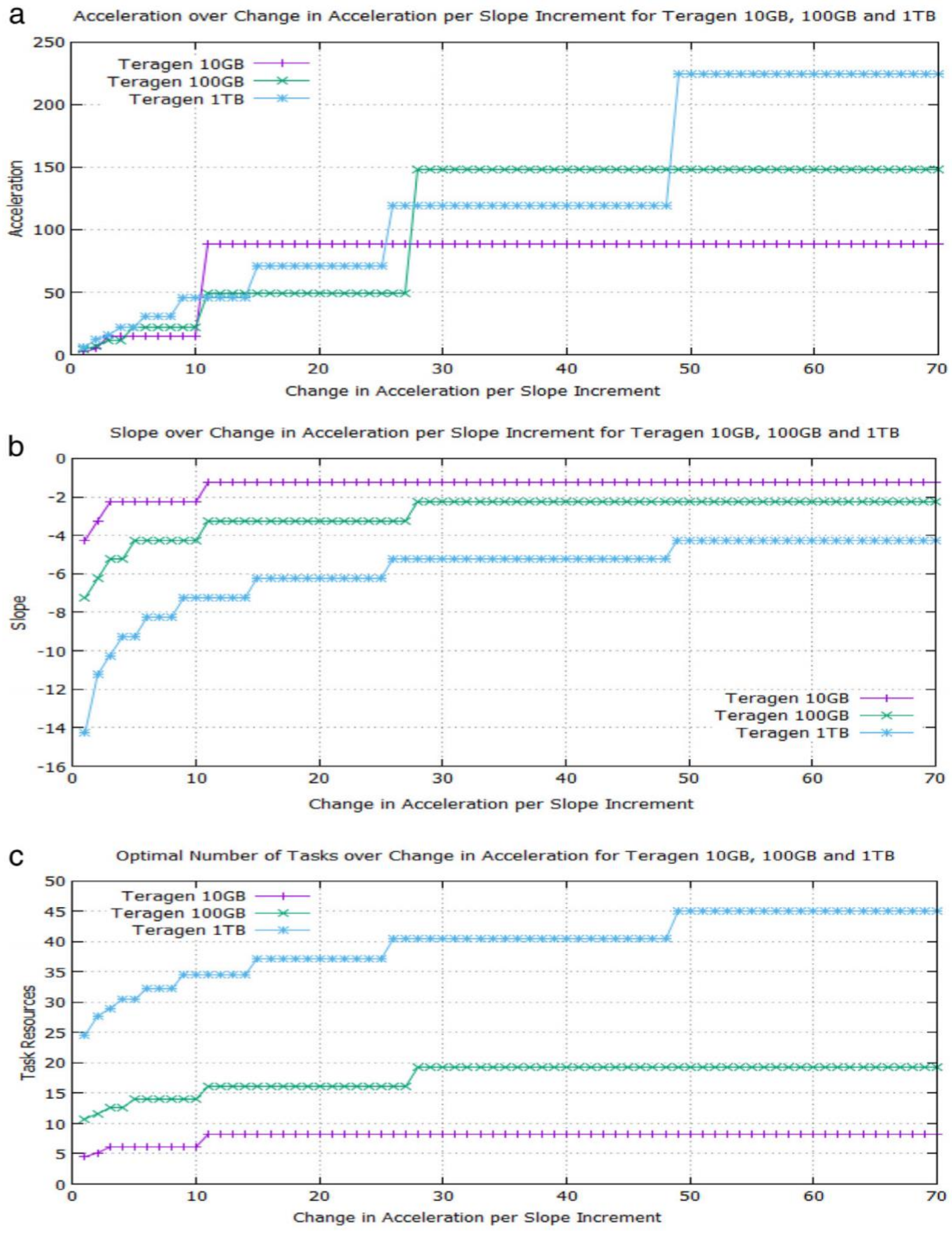
$$\frac{d(acceleration)}{d(tasks)} = \frac{d\,(acceleration)}{d(slope)} \cdot \frac{d\,(slope)}{d(tasks)}. \tag{10}$$

Our algorithm looks for break points on the graphs to compute a table of recommended acceleration, corresponding slope, and optimal number of tasks, 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 (Fig. 2-3, Fig. 13 and Fig. 15). Finally, our algorithm searches for all major plateaus lasting at least eight increments of change in acceleration on the graph of task resources 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 (Fig. 14 and Fig. 16). Our program then outputs the exact optimal numbers of tasks recommended for different workloads (Fig. 13 and Fig. 15). The first recommended number of tasks for the same workload provides the highest efficiency in system performance and energy consumption ratio. The subsequent recommended number(s) of tasks lowers the job runtime a little bit more but at a much less efficient performance/energy ratio. However, increasing the number of tasks beyond the recommended range does not necessarily translate into any further performance gain in execution time.



```
# Table of number of tasks over range of slopes from -0.25 to -39.25 for x=sqrt(-a/slope)
# Slope     Teragen 10GB    Teragen 100GB        Teragen 1TB
 -0.25          18.62           58.12              185.68
 -1.25           8.33           25.99               83.04
 -2.25           6.21           19.37               61.89
 -3.25           5.16           16.12               51.50
 -4.25           4.52           14.10               45.04
 -5.25           4.06           12.68               40.52
 -6.25           3.72           11.62               37.14
 -7.25           3.46           10.79               34.48
…
#Table of acceleration over range of slopes from -0.25 to -39.25 for |accele|=|2a(1/pow(slope,3)|
# Slope     Teragen 10GB       Teragen 100GB        Teragen 1TB
 -0.25       11097.60          108096.00           1103321.60
 -1.25          88.78             864.77              8826.57
 -2.25          15.22             148.28              1513.47
 -3.25           5.05              49.20               502.19
 -4.25           2.26              22.00               224.57
 -5.25           1.20              11.67               119.14
 -6.25           0.71               6.92                70.61
 -7.25           0.46               4.43                45.24
 …
# Table of recommended acceleration, slope and optimal number of tasks over range of
incremental changes in acceleration per slope increment from 1 to 70
#Accele          Teragen 10GB           Teragen 100GB            Teragen 1TB
#Change      Accele  Slope  Tasks    Accele  Slope   Tasks     Accele  Slope   Tasks
1             2.26   -4.25  4.52      4.43   -7.25   10.79      5.96   -14.25  24.59
2             5.05   -3.25  5.16      6.92   -6.25   11.62     12.11   -11.25  27.68
3            15.22   -2.25  6.21     11.67   -5.25   12.68     16.01   -10.25  29.00
…
*************************************************************************************
For Teragen 10GB, the recommended number of tasks resources is   6.21
For Teragen 10GB, the recommended number of tasks resources is   8.33
For Teragen 100GB, the recommended number of tasks resources is  16.12
For Teragen 100GB, the recommended number of tasks resources is  19.37
For Teragen 1TB, the recommended number of tasks resources is  37.14
For Teragen 1TB, the recommended number of tasks resources is  40.52
For Teragen 1TB, the recommended number of tasks resources is  45.04
*************************************************************************************
First recommended number of tasks for same workload provides highest efficiency
in performance/energy ratio. Subsequent number(s) slightly improves job runtime.
```

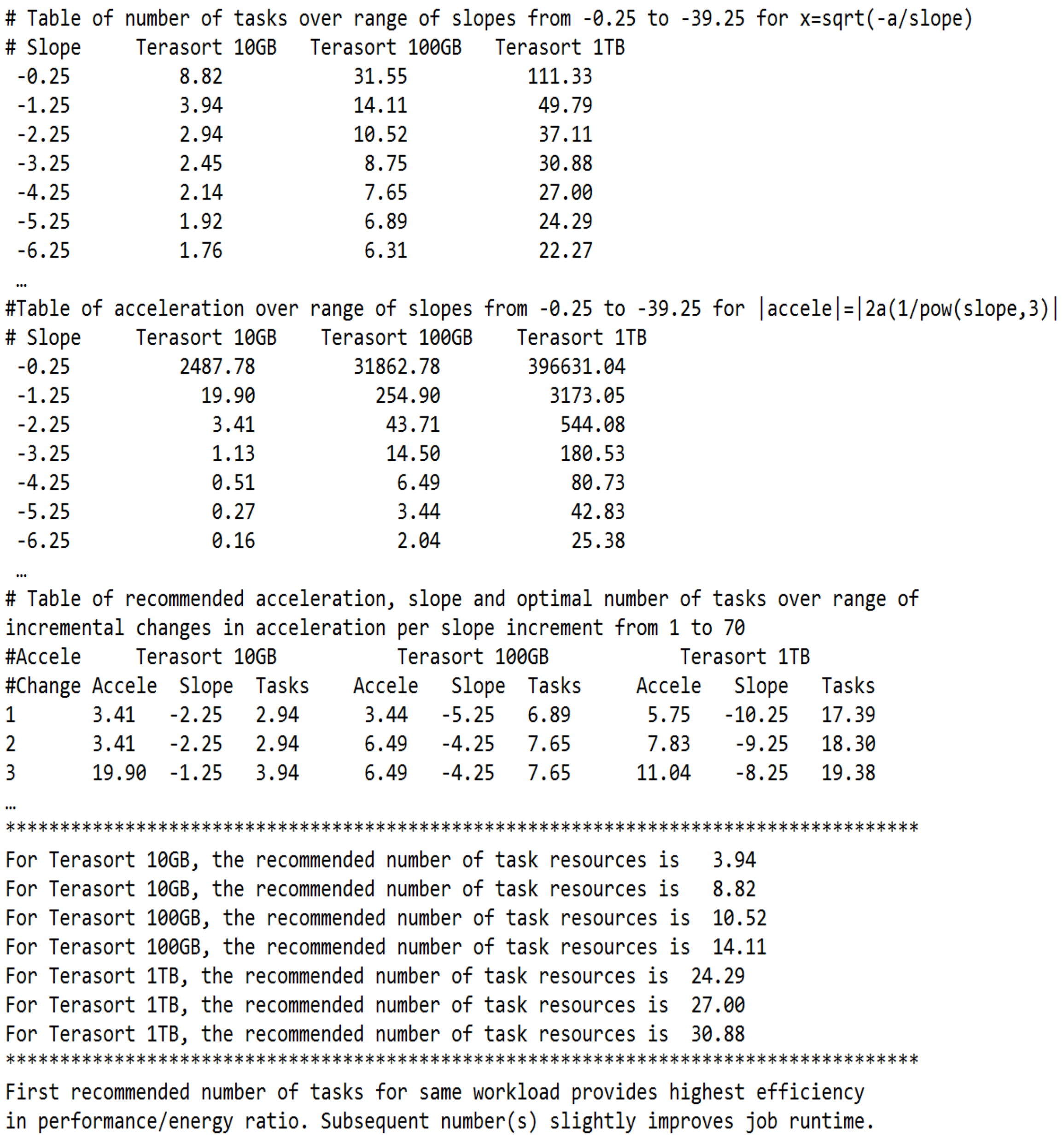
Fig. 13 Applying BToP algorithm to Teragen 10 GB, 100 GB, and 1 TB workloads, our program tabulates the number of tasks over range of slopes, acceleration over range of slopes, and recommended acceleration, slope, and optimal number of tasks over range of incremental changes in acceleration per slope increment, to output the final recommended optimal numbers of tasks for each Teragen workload.


a Acceleration over Change in Acceleration per Slope Increment for Teragen 10GB, 100GB and 1TB

b Slope over Change in Acceleration per Slope Increment for Teragen 10GB, 100GB and 1TB

c Optimal Number of Tasks over Change in Acceleration for Teragen 10GB, 100GB and 1TB

Fig. 14 The optimal number of tasks for Teragen 10 GB, 100 GB, and 1 TB workloads are identified by the major plateaus lasting at least eight 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) task resources versus change in acceleration per slope increment.



```
# Table of number of tasks over range of slopes from -0.25 to -39.25 for x=sqrt(-a/slope)
# Slope     Terasort 10GB   Terasort 100GB   Terasort 1TB
 -0.25          8.82            31.55           111.33
 -1.25          3.94            14.11            49.79
 -2.25          2.94            10.52            37.11
 -3.25          2.45             8.75            30.88
 -4.25          2.14             7.65            27.00
 -5.25          1.92             6.89            24.29
 -6.25          1.76             6.31            22.27
 …
#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          2487.78         31862.78        396631.04
 -1.25            19.90           254.90          3173.05
 -2.25             3.41            43.71           544.08
 -3.25             1.13            14.50           180.53
 -4.25             0.51             6.49            80.73
 -5.25             0.27             3.44            42.83
 -6.25             0.16             2.04            25.38
 …
# Table of recommended acceleration, slope and optimal number of tasks over range of
incremental changes in acceleration per slope increment from 1 to 70
#Accele     Terasort 10GB          Terasort 100GB            Terasort 1TB
#Change Accele  Slope  Tasks    Accele   Slope  Tasks     Accele   Slope    Tasks
1       3.41   -2.25   2.94      3.44   -5.25   6.89       5.75   -10.25    17.39
2       3.41   -2.25   2.94      6.49   -4.25   7.65       7.83    -9.25    18.30
3       19.90  -1.25   3.94      6.49   -4.25   7.65      11.04    -8.25    19.38
…
************************************************************************************
For Terasort 10GB, the recommended number of task resources is   3.94
For Terasort 10GB, the recommended number of task resources is   8.82
For Terasort 100GB, the recommended number of task resources is  10.52
For Terasort 100GB, the recommended number of task resources is  14.11
For Terasort 1TB, the recommended number of task resources is  24.29
For Terasort 1TB, the recommended number of task resources is  27.00
For Terasort 1TB, the recommended number of task resources is  30.88
************************************************************************************
First recommended number of tasks for same workload provides highest efficiency
in performance/energy ratio. Subsequent number(s) slightly improves job runtime.
```

Fig. 15 Applying BToP algorithm to MR Terasort 10 GB, 100 GB, and 1 TB workloads, our program tabulates the number of tasks over range of slopes, acceleration over range of slopes, and recommended acceleration, slope, and optimal number of tasks over range of incremental changes in acceleration per slope increment, to output the final recommended optimal numbers of tasks for each Terasort workload.

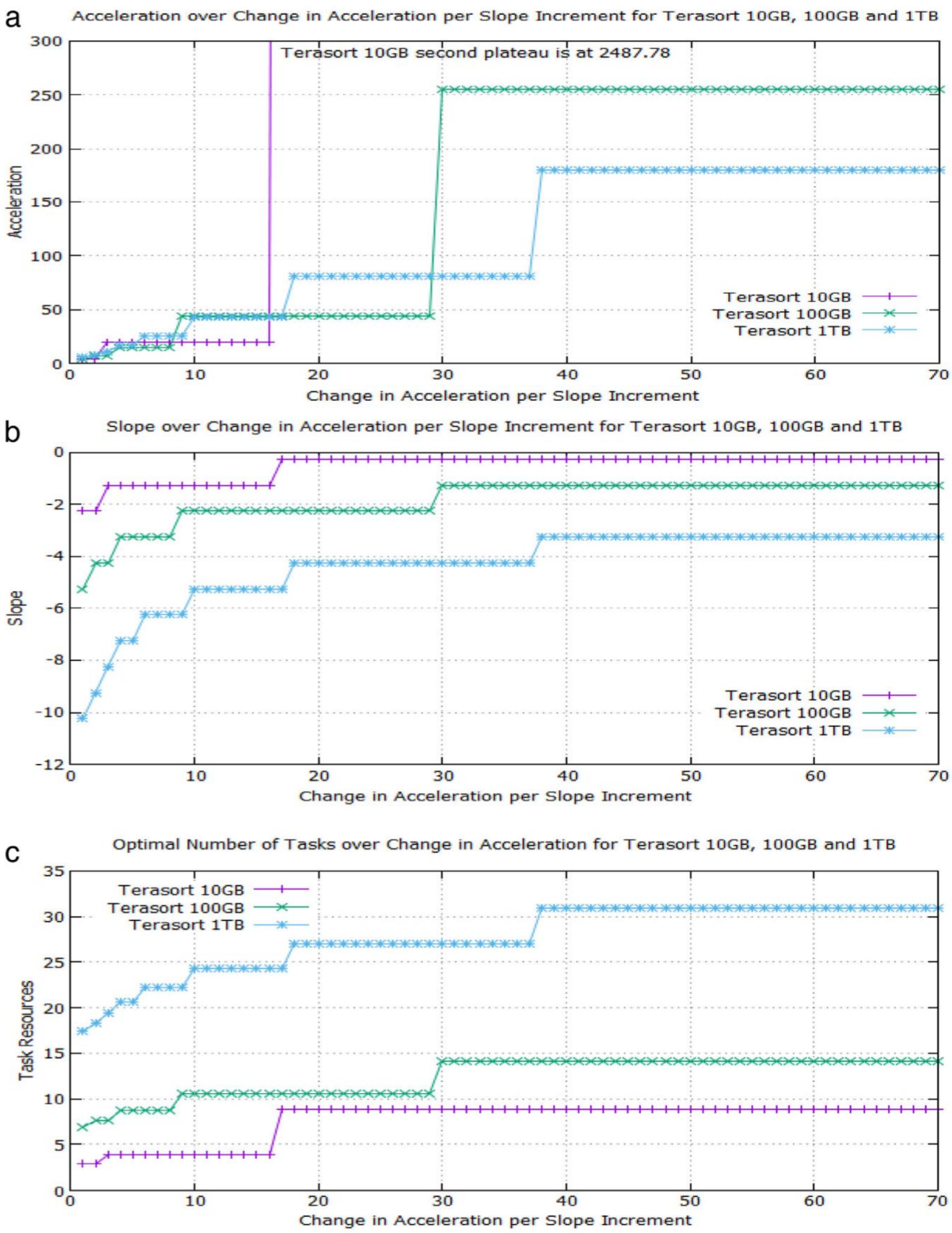


Fig. 16 The optimal numbers of tasks for Terasort 10 GB, 100 GB, and 1 TB workloads are identified by the major plateaus lasting at least eight increments on the graphs. The algorithm searches for break points in the changes in acceleration and outputs: (a) recommended acceleration, (b) corresponding slope and (c) task resources versus change in acceleration per slope increment.

3.6. Comparison of BToP Method to Rules of Thumbs for MapReduce Workload

The recommended optimal resources for Teragen and Terasort 10 GB, 100 GB, and 1 TB in decimal notation generated by our program should be rounded off to integers before use (Fig. 13 and Fig. 15). Their pinpoint accuracy and integrity are verified by the fitted runtime elbow curves generated from their sampled executions (Fig. 12). Comparing the reduce task numbers from our algorithm to those suggested by the three popular rules of thumbs, we notice some major discrepancies throughout the workloads not only between our algorithm and the rules of thumb but also between the rules of thumb themselves (Fig. 17).

From our BToP algorithm, the recommended numbers of reduce tasks were 4–9 for Terasort 10 GB, 11–14 for Terasort 100 GB, and 24–27–31 for Terasort 1 TB (Fig. 15).

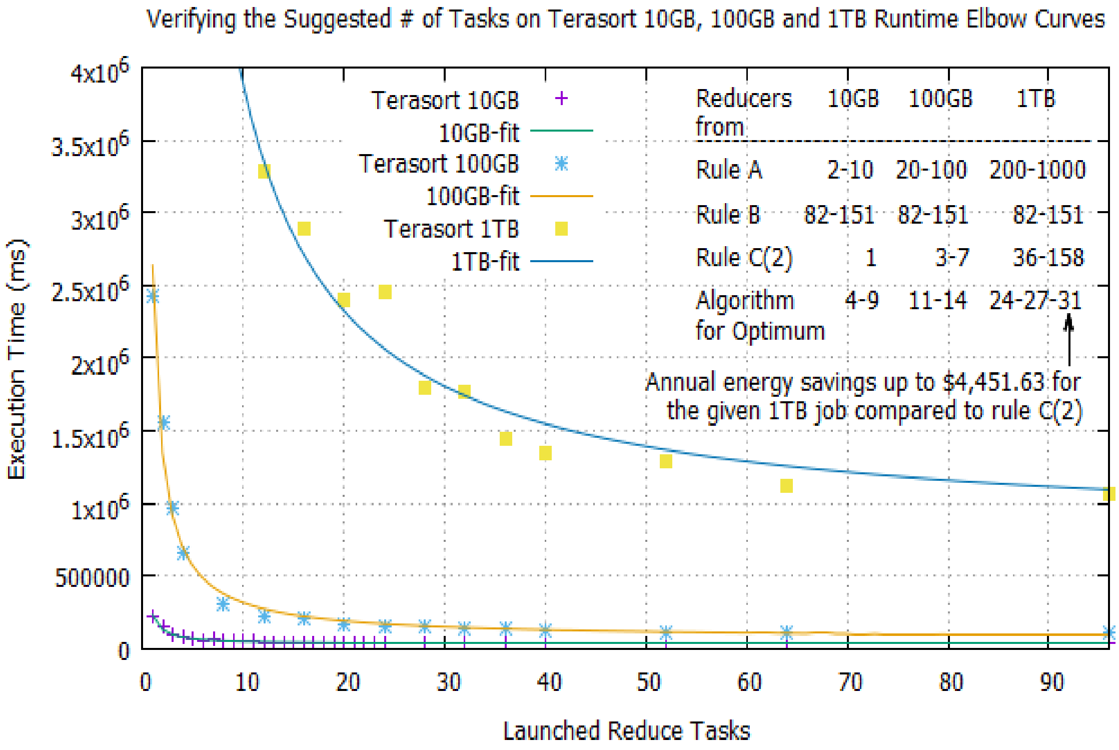


Fig. 17 Fitted elbow curves of Terasort 10 GB, 100 GB, and 1 TB workloads from sampled executions verify the accuracy of our algorithm for optimal resource provisioning in contrast to the unreliable number of reducers calculated from three popular rules of thumb (A, B, and C(2)), which could lead to significant waste of computing resources and energy.

These values are not only optimal but also accurate, as verified by the fitted elbow curves in Fig. 17, since they were derived from the sampled job runtimes of the actual cluster-based system targeted for calibration.

According to rule of thumb (A) [23], where the ideal setting for each reduce task to process should be in a range of 1 GB to 5 GB, the suggested range of reducers are 2–10, 20–100 and 200–1000 for Terasort 10 GB, 100 GB and 1 TB, respectively. Apparently, the suggested range of reducers for Terasort 10 GB is close enough but starting at 2 reducers might be a little weak in performance. The ranges of reducers for Terasort 100 GB and 1 TB are not only wide but also too high causing significant energy waste for no further gain in performance, particularly for 1 TB workload (Fig. 17).

Per Rule of thumb (B) [2], the suggested number of reducers is $0.95*(\text{number of nodes} \; *\text{number of maximum containers per node})= 0.095*(24*3.6)=82$ or $1.75*(\text{number of nodes} * \text{number of maximum containers per node})=1.75*(24*3.6)=151$ for better load balancing. For our cluster node of 4 cores, 2 disks and 32 GB of RAM, the maximum number of containers/node = min ($2*\text{number of CPU cores}, 1.8* \text{number of disks}$, Total available memory/Minimum container size) = min ($2*4, 1.8*2$, (32-6 reserved for system) GB/2 GB) = 3.6, and the recommended minimum container size for total RAM per node above 24 GB is 2048 MB [19]. These suggested numbers of reducers derived solely from the hardware architecture specifications, without taking into consideration the different workloads, are not tailored for performance efficiency since it appears to be based on the misconception that more parallelism is always faster. This rule of thumb suggests an overkill solution for all three Terasort 10 GB, 100 GB, and 1 TB workloads. Using more tasks than necessary equates to overloading the NameNode with unused objects and unnecessarily increasing network transfer as well as framework overhead, needless to say wasting computing resources and energy (Fig. 17).

Under Rule of thumb (C) [16], the ideal number of reducers should be the optimal value that gets them closest to: (1) a multiple of the block size; (2) a task time between 5 and 15 min; (3) creates the fewest files possible. Applying the measurable Rule C(2) of a task time between 300 and 900 s to the benchmark data of our 24-node cluster and their fitted curve functions, the suggested numbers of reducers come out to be 3–7 and 36–158 for Terasort 100 GB and 1 TB, respectively. A value of 1 task is suggested for Terasort 10 GB even though its benchmark task time is below 156 s. None of these values matches the actual optimal range of reducers for Terasort 10 GB, 100 GB and 1 TB workloads. The first value of 36 tasks at the beginning of the range for Terasort 1 TB might be close to the tail end of the actual optimal range of 24–27–31 tasks. But this rule of thumb further suggests an upper range for Terasort 1 TB of up to 158 reducers which is a complete waste of energy with no further improvement in runtime (Fig. 17).

Job runtime is an important metric in MapReduce v.2 since resources are shared by several applications running in parallel on YARN, which allocates maps and reduces as needed by the job dynamically. As shown in equation 11, 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 [8]. Power models based on a linear interpolation of CPU utilization have been shown to be accurate with I/O workloads for this class of server, since network and disk activity contribute negligibly to dynamic power consumption [29, 33].

$$\text{Energy}(N) = [\text{Time}_{\text{run}}(N) * \text{Power}_{\text{active}}(N)] + [\text{Time}_{\text{idle}} * \text{Power}_{\text{idle}}] \qquad (11)$$

To quantify the potential saving in using our algorithm, we compare the highest recommended number of tasks for Terasort 1 TB from our algorithm (31 tasks equivalent to 9 nodes) and the rule of thumb C(2) (158 tasks equivalent to 44 nodes) based on a cluster with a maximum number of containers per node of 3.6 [19]. For an active power consumption per node of 250 W, idle power of 235 W, and an average job arrival time of 2000 s:

$$E(9) = [1{,}773 \text{ s} * (250 \text{ W} * 9)] + [(2{,}000 \text{ s} – 1{,}773 \text{ s}) * 235 \text{ W}]$$

$$= 4{,}042.595 \text{ kj} = 1.123 \text{ kWh per job}$$

$$E(44) = [969 \text{ s} * (250 \text{ W} * 44)] + [(2{,}000 \text{ s} – 969 \text{ s}) * 235 \text{ W}]$$

$$= 10{,}901.285 \text{ kj} = 3.028 \text{ kWh per job}$$

Hence, by provisioning task resources with our algorithm, we reduce the energy consumption by about two-thirds. This translates to (1.905 kWh saved per job) * [((365*24) h/yr / ((2000/3600) h/job)] = 30,038 kWh saved per year. According to the US Department of Energy, the May 2015 average retail price of electricity to commercial customers in California was $0.1482 per kWh. Thus, the annual energy saving amounts to $4,451.63 for the given 1TB compute job [36].

From the table of number of reducers suggested by different methods under assessment for Terasort 10 GB, 100 GB, and 1 TB workloads (Fig. 17), the potential energy savings could be significantly larger if we compared the highest recommended numbers of tasks for Terasort 1 TB from our algorithm (31 tasks) and the rule of thumb A (1000 tasks), or the highest recommended numbers of tasks for Terasort 10 GB and 100 GB from our algorithm (9 tasks and 14 tasks, respectively) and the rule of thumb B (151 tasks for both workloads).

Using only the right number of tasks needed for a job will allow users to allocate the remaining resources for other jobs in a multi-tenant Hadoop YARN cluster running at full or near full capacity and therefore, will increase the overall system throughput. Even when the system is lightly loaded, avoiding allocating more tasks than necessary still certainly results in energy saving. Dialing up the number of tasks allocated for a job within the recommended range, users could get a little bit of extra performance gain. However, the continuing slight reduction in execution time quickly disappears while the power consumption expense increases linearly with the number of tasks launched. As such, we do not recommend allocating more task resources beyond the best trade-off points, which offers rapidly diminishing returns, when it comes to runtime performance and energy efficiency.

Our proposed solution for resource provisioning in MapReduce offers a verifiable working method, formula and algorithm to ascertain the optimal task resource values for performance efficiency. The recommended values will always be accurate since they are derived from actual sampled executions of each specific application and system in use. Hadoop MapReduce users no longer have to rely on inaccurate rules of thumb to guess the required number of tasks for a job. Although our experiment is conducted on a small-scale

24-node Hadoop cluster, our proposed solution should also work for larger workloads running on a much bigger cluster of several thousands of nodes in today’s datacenter. If our proposed method for efficient resource provisioning is adopted and consistently applied to all jobs running on all Hadoop clusters in an organization’s datacenter such as the 42,000 compute nodes running Hadoop clusters in Yahoo datacenter, the amount of aggregate annual energy saving will be very significant, up to several million dollars.

Chapter 4

Efficient Resource Provisioning in Spark

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 [32]. Since then we have further explored some more software modules and data processing engines running on Hadoop ecosystem including Apache Spark for similar or different approach to optimize resource provisioning for performance efficiency. In this second part of the thesis, we address the same question of how to allocate the optimal number of executor resources for a workload in Apache Spark, an emerging general purpose and lightning fast cluster computing system providing rich high-level APIs which define Resilient Distributed Datasets (RDDs) for parallel and distributed computing across cluster nodes.

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, Spark-Bench [5, 31] Terasort tests prove that Spark using BToP method could still outperform Spark with DRA enabled, particularly in production environment where job profiling for behavioral replication will lead to the most optimal resource provisioning.

Our experiment with Spark-Bench Terasort indicates that Spark with BToP method consistently yield better performance than Spark with DRA enabled for all three datasets of 10GB, 100GB, and 1TB. Since the recommended number of executor resources for a workload derived from the BToP method is based on behavioral replication by matching a workload to its corresponding resource consumption signature, it is expected to be more precise for identical repetitive jobs in production environment. On the other hand, Spark's performance optimized by dynamic resource allocation is suitable for all general purposes applications including any workload with erratic and unpredictable behaviors.

In 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. 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. As such, 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 [11], 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 prior research works on Hadoop resource provisioning to accomplish certain application performance goals through dynamic job profiling [6], cluster size elasticizing [18], scaling past execution results to meet given service level objectives [37], and matching an application's resource consumption with a database of similar signatures of other applications [22] could still be referenced when applying the BToP method [32] to 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 its momentum 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 [1], 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.

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. 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 Dynamic Resource Allocation (DRA) mechanism [4]. 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 [4, 10]. 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 using BToP method with DRA disabled or Spark with DRA enabled would be best for certain circumstances, we need to examine Spark's architecture with its Resilient Distributed Dataset (RDD), its distributed execution and DRA mechanism.

4.1 Background Knowledge on Apache Spark

Apache Spark is a general-purpose data processing engine suitable for interactive analytics, machine learning, streaming processing, and data integration. Its cluster-computing framework provides simple APIs centered on Resilient Distributed Datasets (RDDs) functioning as a working set for distributed programming and fault tolerance. Spark can expedite complex analytic applications up to 100 times faster than MapReduce with in-memory processing or up to 10 times faster on disk. Spark job performs multiple operations consecutively in memory and only spills to disk when exceeding memory limitations. Spark is written in Scala to be used with a wide range of programming languages including Java, Python, Scala, SQL, and R, on many different platforms [11, 24].

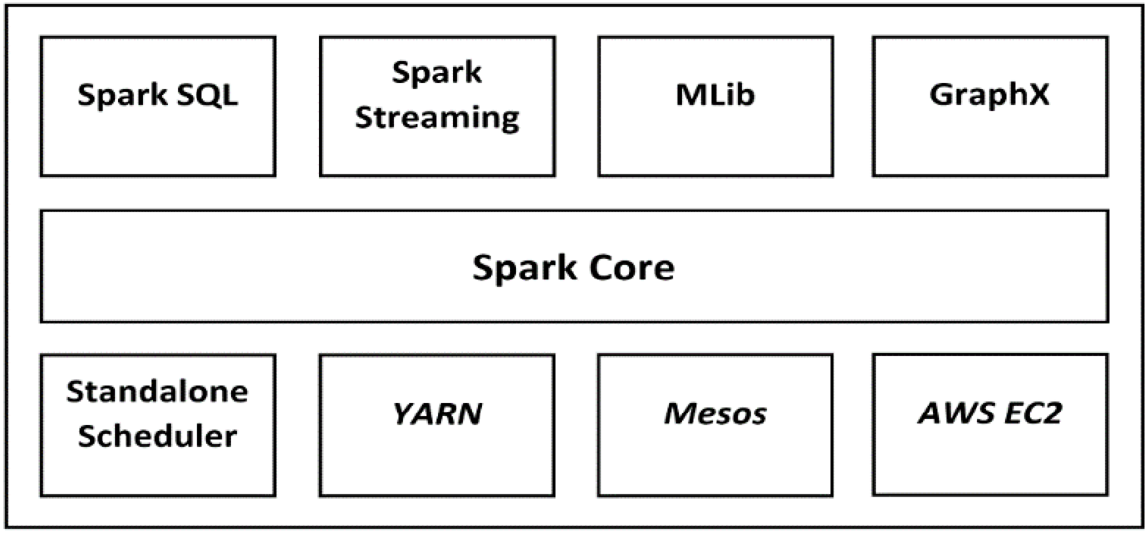


Fig. 18 Apache Spark Echosystem

Apache Spark stack comprises Spark SQL, Spark Streaming, Machine Learning libraries (MLlib), and GraphX which are built on top of Spark core, the base distributed execution engine. Spark core handles the basic functions of task scheduling, memory management, fault recovery, and interacting with storage systems, and is home to the APIs which define RDDs. Spark SQL allows querying data via SQL and HQL, supports many sources of data, such as Hive, parquet, and json, and combines SQL queries with programmatic data manipulations supported by RDDs. Spark Streaming provides API for manipulating data streams and provides some degree of fault tolerance, throughput, and scalability as Spark core. MLlib provides Machine Learning algorithms (classification, regression, clustering, collaborative filtering, generic gradient descent optimization). GraphX provides libraries for manipulating graphs and performing graph-parallel computations and allows the creation of directed graph with arbitrary properties attached to each vertex and edge. Spark is highly scalable and can be used with any storage systems supported by Hadoop APIs including but not limited HDFS, local FS, Google cloud, Cassandra, Hive, MapR FS, and HBase. Spark can run on a variety of clusters including Hadoop YARN, Apache Mesos, and Amazon web services Elastic Compute Cloud (EC2), or in a stand-alone mode (Fig. 18).

4.1.1 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 Nodes, also known as Executors. Each Executor represents a unique resource unit in Spark, which is a combination of certain number of CPU cores and memory as specified through *spark.executor.cores* and *spark.executor.memory*, for requesting resources from the cluster manager (Fig. 19). 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 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 cluster. 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 has sufficient information of how it was derived from other RDDs to recover the lost partition without any costly replication [42, 43].

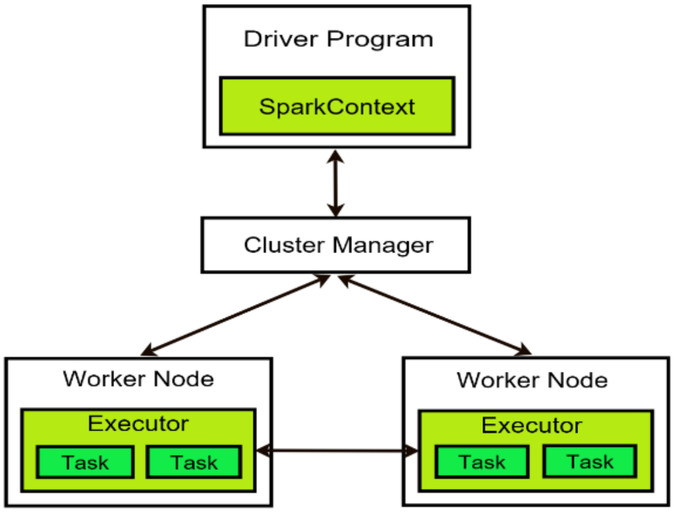


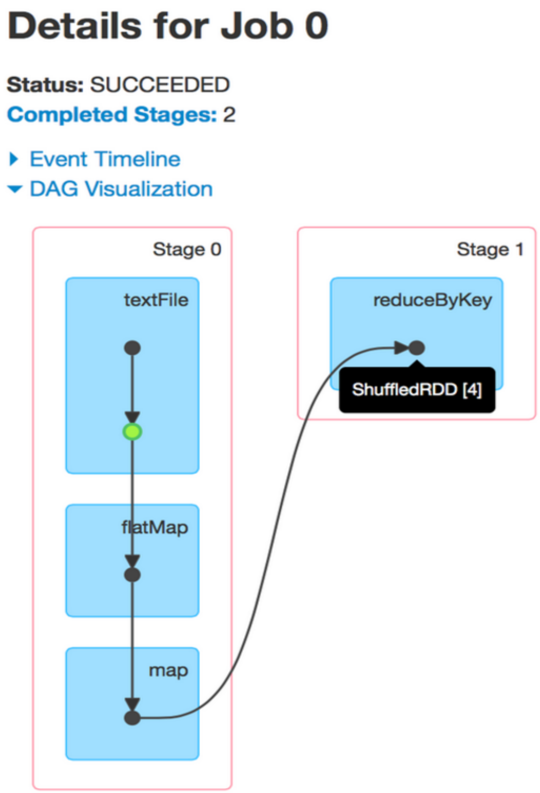
Fig. 19 Spark Architecture in Cluster Mode

Spark programming model is based on two types high order functions which are transformations to create new RDD and actions to compute result 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 later not needed. However, it can also introduce processing bottlenecks causing application stall. For better cluster performance, RDDs persist in memory whenever possible.

Spark jobs often comprises multiple sequential stages. When an action is called on an RDD, a Directed Acyclic Graph (DAG)[1] 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. A stage contains task based on the partition of the input data. 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 task through the cluster manager [12].

For example, Fig. 20 illustrates a DAG visualization of a simple word count job. Once the DAG is build, Spark scheduler creates a physical execution plan. It splits the graph into multiple stages which are created based on the transformations. The narrow transformations are pipelined together into a single stage as stage 0 with three operations. Stage 1 has a single operation for the wide transformation. The number of tasks in stages submitted to the task scheduler depends on the number of partitions present in the *textFile*. A set of tasks is created for each of the partitions which are submitted in parallel provided there are enough core resources. In stage 0, a *textFile* operation is performed to read an input file in HDFS, then a *flatMap* operation is carried out to split each line into words, then a *map* operation is executed to form (word, 1) pairs, then finally a *reduceByKey* operation is performed to sum the counts for each word. The dots in the

[1] A Directed Acyclic Graph (DAG) is consistent with the conditional independence relations in the First Order Markov process which states that the state of X_i depends only on X_{i-1}. For any sequence, $p(X_1=x_1, \ldots, X_n=x_n) = p(X_1=x_1|X_{i-1}=x_{i-1})$. i.e. Once the state is known, the history may be discarded since the future is independent of the past given the present. e.g. The probability of the sequence X1, X2, and X3 is the joint probability $p(X_1=x_1, X_2=x_2, X_3=x_3) = p(X_1=x_1)\ p(X_2=x_2|X_1=x_1)\ p(X_3=x_3|X_1=x_1, X_2=x_2)$ given by the chain rule of probability.



Details for Job 0

Status: SUCCEEDED

Completed Stages: 2

Source: Databricks

Fig. 20 DAG visualization of a simple word count job.

boxes represent RDDs created in the corresponding operations. One of the RDDs denoted in green is cached in the first stage to reduce reading from HDFS. Significantly, Spark optimizes pipeline operations with several operators in the same task to eliminate the need for an extra stage [12].

4.1.2 Distributed Execution in Spark

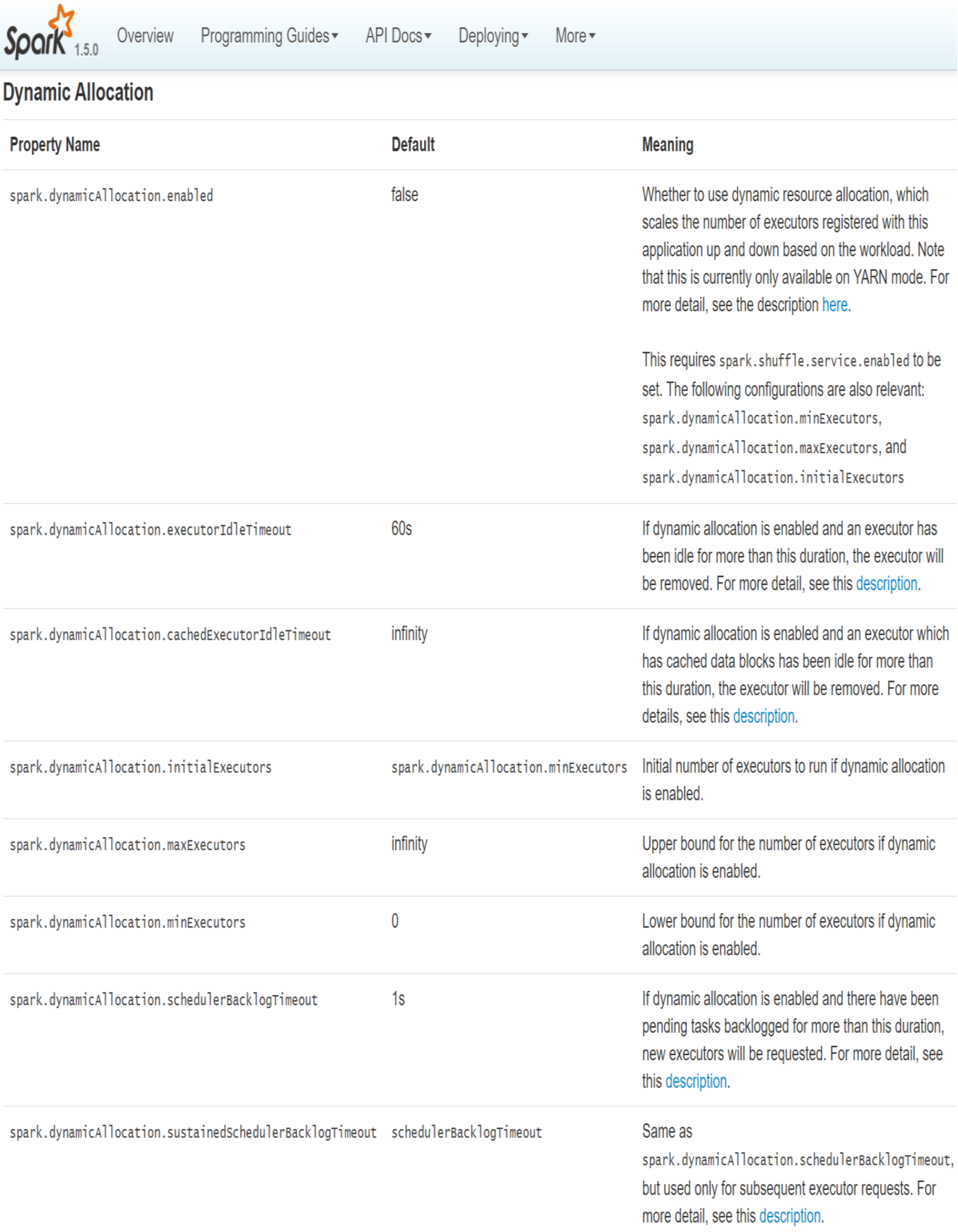
Although Spark starts up in YARN, it does all task scheduling, task forming, and process execution independent of YARN. Spark driver runs inside of YARN's ApplicationMaster, which is responsible for driving the application and requesting resources from YARN. Spark job runs as a Java process on a JVM.

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 requires 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 be later 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 a 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 [24, 41].

4.1.3 Spark's Dynamic Resource Allocation (DRA)

Although sufficient resource pre-allocation 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 (DRA) 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 (Fig. 21) [4].



Dynamic Allocation

Property Name	Default	Meaning
`spark.dynamicAllocation.enabled`	false	Whether to use dynamic resource allocation, which scales the number of executors registered with this application up and down based on the workload. Note that this is currently only available on YARN mode. For more detail, see the description here. This requires `spark.shuffle.service.enabled` to be set. The following configurations are also relevant: `spark.dynamicAllocation.minExecutors`, `spark.dynamicAllocation.maxExecutors`, and `spark.dynamicAllocation.initialExecutors`
`spark.dynamicAllocation.executorIdleTimeout`	60s	If dynamic allocation is enabled and an executor has been idle for more than this duration, the executor will be removed. For more detail, see this description.
`spark.dynamicAllocation.cachedExecutorIdleTimeout`	infinity	If dynamic allocation is enabled and an executor which has cached data blocks has been idle for more than this duration, the executor will be removed. For more details, see this description.
`spark.dynamicAllocation.initialExecutors`	`spark.dynamicAllocation.minExecutors`	Initial number of executors to run if dynamic allocation is enabled.
`spark.dynamicAllocation.maxExecutors`	infinity	Upper bound for the number of executors if dynamic allocation is enabled.
`spark.dynamicAllocation.minExecutors`	0	Lower bound for the number of executors if dynamic allocation is enabled.
`spark.dynamicAllocation.schedulerBacklogTimeout`	1s	If dynamic allocation is enabled and there have been pending tasks backlogged for more than this duration, new executors will be requested. For more detail, see this description.
`spark.dynamicAllocation.sustainedSchedulerBacklogTimeout`	`schedulerBacklogTimeout`	Same as `spark.dynamicAllocation.schedulerBacklogTimeout`, but used only for subsequent executor requests. For more detail, see this description.

Source: Spark.apache.org

Fig. 21 Apache Spark 1.5.0 Dynamic Resource Allocation Properties

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 (Fig. 22).

Spark's *ExecutorAllocationManager* calculates the maximum number of executors it requires through pending and running tasks.

```
private def maxNumExecutorsNeeded(): Int = {
        val numRunningOrPendingTasks = listener.totalPendingTasks + listener.totalRunningTasks
        (numRunningOrPendingTasks + tasksPerExecutor - 1) / tasksPerExecutor
}
```

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 request to cluster manager.

```
// The target number exceeds the number we actually need, so stop adding new
// executors and inform the cluster manager to cancel the extra pending requests
val oldNumExecutorsTarget = numExecutorsTarget
numExecutorsTarget = math.max(maxNeeded, minNumExecutors)
numExecutorsToAdd = 1

// If the new target has not changed, avoid sending a message to the cluster manager
if (numExecutorsTarget < oldNumExecutorsTarget) {
  client.requestTotalExecutors(numExecutorsTarget, localityAwareTasks, hostToLocalTaskCount)
  logDebug(s"Lowering target number of executors to $numExecutorsTarget (previously " +
    s"$oldNumExecutorsTarget) because not all requested executors are actually needed")
}
numExecutorsTarget - oldNumExecutorsTarget
```

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 boost it with the number of executors to add for the first round which is doubled in each subsequent round.

```
val oldNumExecutorsTarget = numExecutorsTarget
// There's no point in wasting time ramping up to the number of executors we already have, so
// make sure our target is at least as much as our current allocation:
numExecutorsTarget = math.max(numExecutorsTarget, executorIds.size)
// Boost our target with the number to add for this round:
numExecutorsTarget += numExecutorsToAdd
// Ensure that our target doesn't exceed what we need at the present moment:
numExecutorsTarget = math.min(numExecutorsTarget, maxNumExecutorsNeeded)
// Ensure that our target fits within configured bounds:
numExecutorsTarget = math.max(math.min(numExecutorsTarget, maxNumExecutors), minNumExecutors)

val delta = numExecutorsTarget - oldNumExecutorsTarget

// If our target has not changed, do not send a message
// to the cluster manager and reset our exponential growth
if (delta == 0) {
  numExecutorsToAdd = 1
  return 0
}

val addRequestAcknowledged = testing ||
  client.requestTotalExecutors(numExecutorsTarget, localityAwareTasks, hostToLocalTaskCount)
if (addRequestAcknowledged) {
  val executorsString = "executor" + { if (delta > 1) "s" else "" }
  logInfo(s"Requesting $delta new $executorsString because tasks are backlogged" +
    s" (new desired total will be $numExecutorsTarget)")
  numExecutorsToAdd = if (delta == numExecutorsToAdd) {
    numExecutorsToAdd * 2
  } else {
    1
  }
  delta
} else {
  logWarning(
    s"Unable to reach the cluster manager to request $numExecutorsTarget total executors!")
  0
}
```

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 (60s for default value).

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 delay later jobs in the queue [3, 4].

4.2 Spark-Bench Terasort Experiment with BToP Method

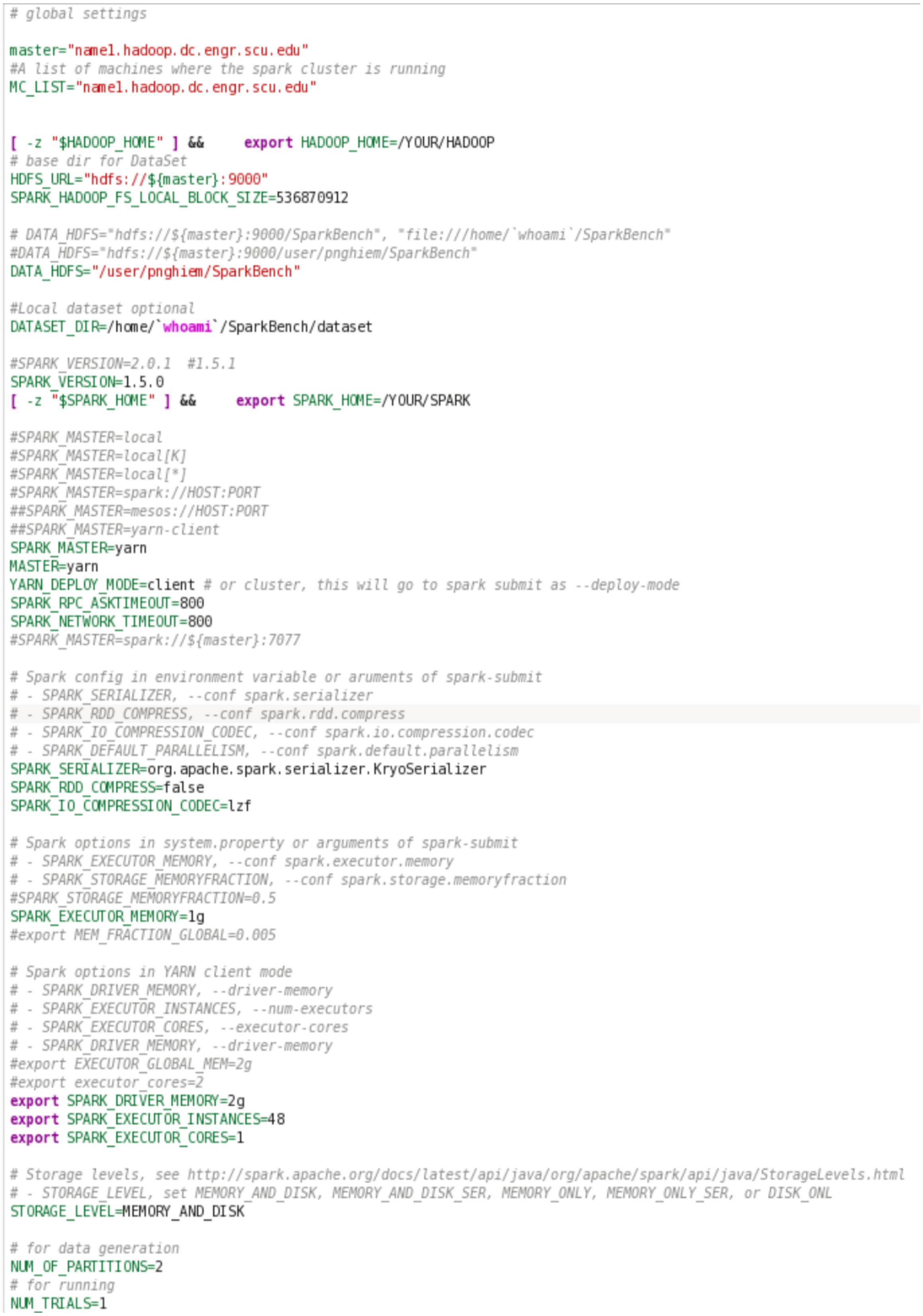
To illustrate our BToP method for obtaining the optimal number of executors for different workloads, we use the Teragen and Terasort components of the Spark-Bench Terasort test to experiment with 10 GB, 100 GB and 1 TB datasets. Spark-Bench, which contains a comprehensive set of benchmark workloads for applications in machine learning, graph computation, SQL queries, and streaming application, is a Spark specific benchmarking suite proposed by M. Li et al. from IBM TJ Watson Research Center as a standard benchmark suite for Apache Spark [31, 5]. The Terasort component among many others not mentioned in Li et al. [31] was later added to the updated Spark-Bench on Github [34].

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 261TB (Raw), 80TB (usable after replication overhead). The NameNodes are VM of 4 cores and 24GB of RAM, each running on Intel Xeon E5-2690 physical hosts of 8 cores and 16 threads, 2.9 GHz base frequency and 3.8GHz max turbo frequency, and TDP of 135 W. The DataNodes are physical system running Intel Xeon E3-1240 v3, 3.4GHz base frequency and 3.8GHz max turbo frequency, and TDP of 80 W. Each node has 4 cores, 8 threads, 32GB of RAM, two 6TB hard disks and 1Gbit network bandwidth. All nodes are connected to a switch with a backplane speed of 48Gbps.

For Spark running in YARN cluster mode, the *--nun-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 --*nun-executors* option is specifically assigned to certain number of executors at runtime, it supersedes the *spark.dynamicAllocation.enabled* configuration which is then disabled for that job submission. For the three 10GB, 100GB, and 1TB 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 elbow 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 (Fig. 22).

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 1TB 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. Apparently, the default value of RPC communication timeout set to 120 seconds is clearly not sufficient for a heavy 1TB workload processed by Spark 1.5.0 with DRA enabled. To get Spark-Bench Terasort to complete the job, we have to increase the timeouts by a large margin in setting the SPARK_RPC_ASKTIMEOUT=800 seconds and adding the umbrella SPARK_NETWORK_TIMEOUT=800 seconds to cover all timeout exceptions in Spark for heavy workloads (Fig. 22).

On the other hand, there is no problem running Spark-Bench Terasort on a 1TB dataset by manually assigning a static --*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



```
# global settings

master="namel.hadoop.dc.engr.scu.edu"
#A list of machines where the spark cluster is running
MC_LIST="namel.hadoop.dc.engr.scu.edu"


[ -z "$HADOOP_HOME" ] &&     export HADOOP_HOME=/YOUR/HADOOP
# base dir for DataSet
HDFS_URL="hdfs://${master}:9000"
SPARK_HADOOP_FS_LOCAL_BLOCK_SIZE=536870912

# DATA_HDFS="hdfs://${master}:9000/SparkBench", "file:///home/`whoami`/SparkBench"
#DATA_HDFS="hdfs://${master}:9000/user/pnghiem/SparkBench"
DATA_HDFS="/user/pnghiem/SparkBench"

#Local dataset optional
DATASET_DIR=/home/`whoami`/SparkBench/dataset

#SPARK_VERSION=2.0.1  #1.5.1
SPARK_VERSION=1.5.0
[ -z "$SPARK_HOME" ] &&     export SPARK_HOME=/YOUR/SPARK

#SPARK_MASTER=local
#SPARK_MASTER=local[K]
#SPARK_MASTER=local[*]
#SPARK_MASTER=spark://HOST:PORT
##SPARK_MASTER=mesos://HOST:PORT
##SPARK_MASTER=yarn-client
SPARK_MASTER=yarn
MASTER=yarn
YARN_DEPLOY_MODE=client # or cluster, this will go to spark submit as --deploy-mode
SPARK_RPC_ASKTIMEOUT=800
SPARK_NETWORK_TIMEOUT=800
#SPARK_MASTER=spark://${master}:7077

# Spark config in environment variable or aruments of spark-submit
# - SPARK_SERIALIZER, --conf spark.serializer
# - SPARK_RDD_COMPRESS, --conf spark.rdd.compress
# - SPARK_IO_COMPRESSION_CODEC, --conf spark.io.compression.codec
# - SPARK_DEFAULT_PARALLELISM, --conf spark.default.parallelism
SPARK_SERIALIZER=org.apache.spark.serializer.KryoSerializer
SPARK_RDD_COMPRESS=false
SPARK_IO_COMPRESSION_CODEC=lzf

# Spark options in system.property or arguments of spark-submit
# - SPARK_EXECUTOR_MEMORY, --conf spark.executor.memory
# - SPARK_STORAGE_MEMORYFRACTION, --conf spark.storage.memoryfraction
#SPARK_STORAGE_MEMORYFRACTION=0.5
SPARK_EXECUTOR_MEMORY=1g
#export MEM_FRACTION_GLOBAL=0.005

# Spark options in YARN client mode
# - SPARK_DRIVER_MEMORY, --driver-memory
# - SPARK_EXECUTOR_INSTANCES, --num-executors
# - SPARK_EXECUTOR_CORES, --executor-cores
# - SPARK_DRIVER_MEMORY, --driver-memory
#export EXECUTOR_GLOBAL_MEM=2g
#export executor_cores=2
export SPARK_DRIVER_MEMORY=2g
export SPARK_EXECUTOR_INSTANCES=48
export SPARK_EXECUTOR_CORES=1

# Storage levels, see http://spark.apache.org/docs/latest/api/java/org/apache/spark/api/java/StorageLevels.html
# - STORAGE_LEVEL, set MEMORY_AND_DISK, MEMORY_AND_DISK_SER, MEMORY_ONLY, MEMORY_ONLY_SER, or DISK_ONL
STORAGE_LEVEL=MEMORY_AND_DISK

# for data generation
NUM_OF_PARTITIONS=2
# for running
NUM_TRIALS=1
```

Fig. 22   SparkBench env.sh settings for the experiment with 48 executors.

inherent timeout errors in processing heavy workloads has been reported by many users, especially for Spark versions older than version 1.6.0.

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 [21]. 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 give 8 threads. Although each node has 32GB of RAM and YARN’s default container allocation size is 1GB, 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 [20].

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 [14]:

Total vCores available>=*spark.executor.cores***spark.executor.instances*+1 core for driver

Total Memory available >= (*spark.executor.memory* + executor memory overhead)
* *spark.executor.instances* + *spark.driver.memory* + driver memory overhead

In our experiment, we set the *spark.executor.memory* = 1g 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 192GB out of 768GB of available RAM. As such, we can optionally increase the heap size up to 3GB without affecting the number of executors per node. In general, a single executor should not have more than 64GB 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 [9]. 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.

We now implement the 7 steps outlined in Chapter 2 for efficient resource provisioning in Spark to apply the BToP method to Spark-Bench Terasort (Fig. 1).

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

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 (Fig. 23).

The preview data of Spark-Bench Terasort indicates that when the allocated number of executors is too little 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 elbow turn on the runtime curve. Although increasing parallelization would normally improve job performance, the growing framework overhead and potential resource contention due to excessive number of executors could eventually degrade the overall job performance. In our experiment, this observed performance degradation of Spark on YARN despite the steady increase in executor resources is more obvious with 10GB and 100GB workloads than with 1TB workload (Fig. 23).

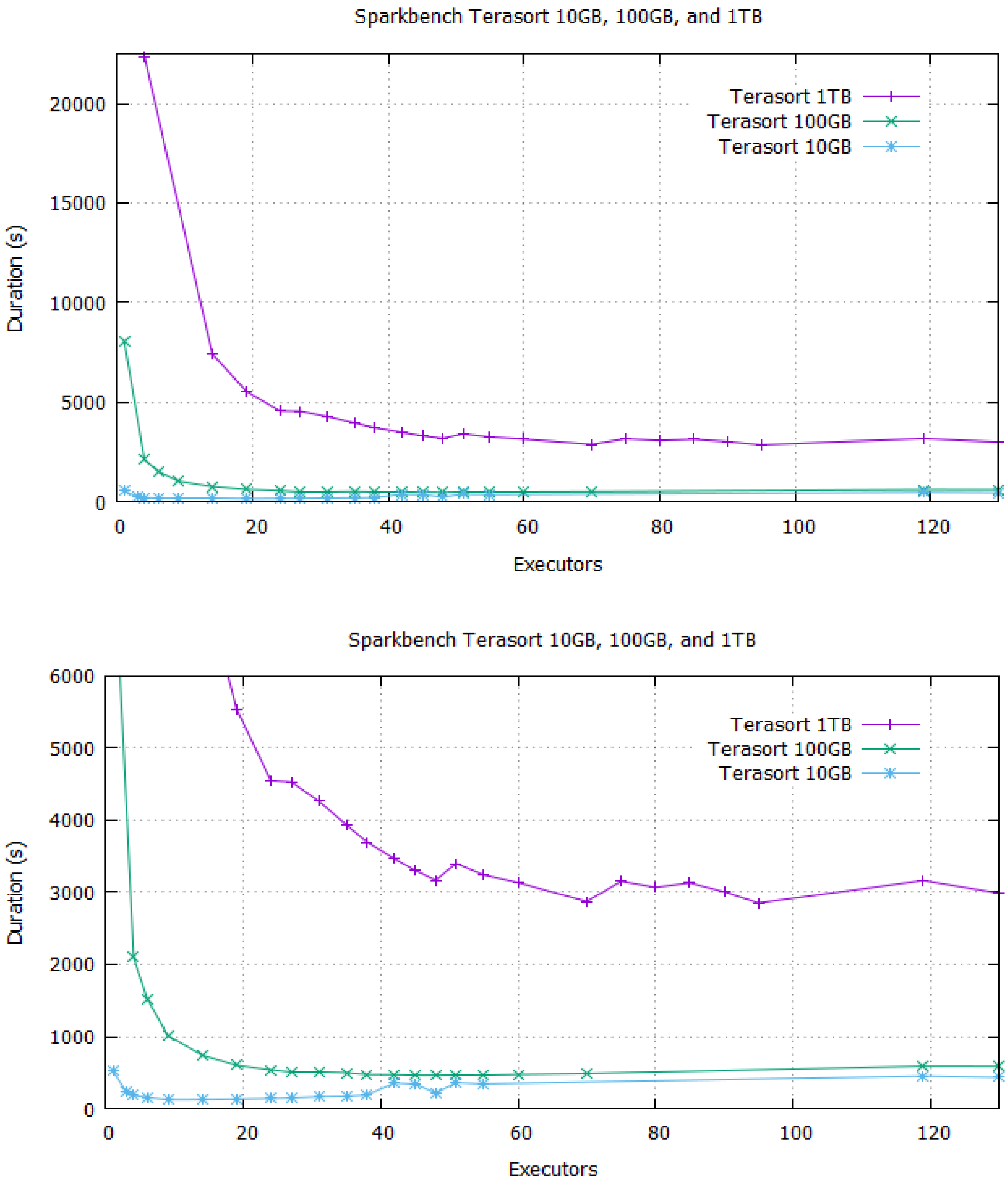
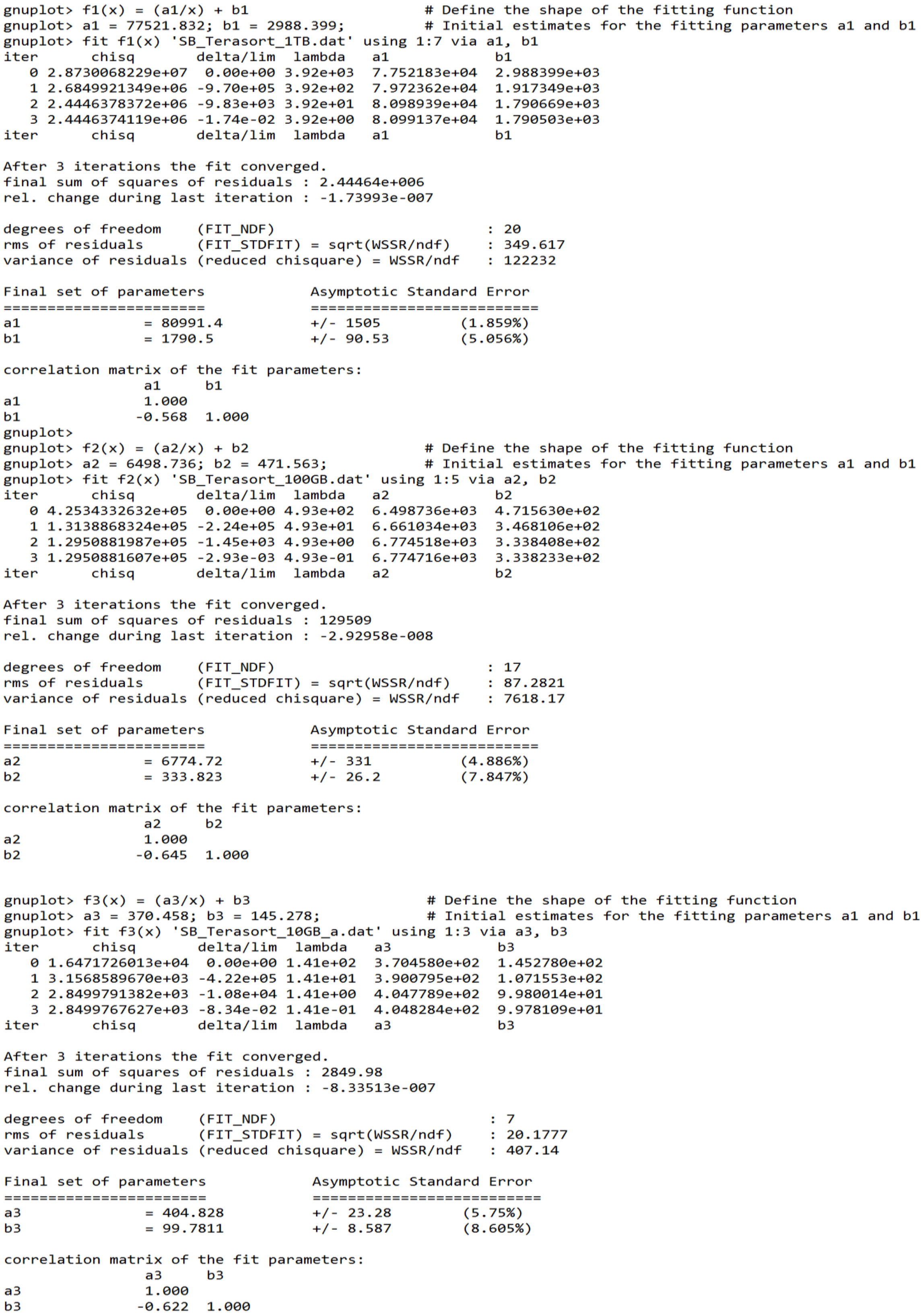


Fig. 23 Plots of Preview data of duration vs. executors of SparkBench Terasort 10GB, 100GB and 1TB of data at different vertical scales.

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 (Fig. 24).

Using GNUplot 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-Neuton 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" (Fig. 24).

With the obtained fit parameter *a*, we then plot the three fitted elbow curves of duration vs. executors for Spark-Bench Terasort 10 GB, 100 GB and 1 TB workloads (Fig. 25). The resulted performances of Spark with DRA enabled for all three workloads, indicate that Spark’s DRA mechanism is 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.



```
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
gnuplot>
gnuplot> f2(x) = (a2/x) + b2                       # Define the shape of the fitting function
gnuplot> a2 = 6498.736; b2 = 471.563;              # Initial estimates for the fitting parameters a1 and b1
gnuplot> fit f2(x) 'SB_Terasort_100GB.dat' using 1:5 via a2, b2
iter      chisq       delta/lim  lambda   a2            b2
   0 4.2534332632e+05  0.00e+00 4.93e+02  6.498736e+03  4.715630e+02
   1 1.3138868324e+05 -2.24e+05 4.93e+01  6.661034e+03  3.468106e+02
   2 1.2950881987e+05 -1.45e+03 4.93e+00  6.774518e+03  3.338408e+02
   3 1.2950881607e+05 -2.93e-03 4.93e-01  6.774716e+03  3.338233e+02
iter      chisq       delta/lim  lambda   a2            b2

After 3 iterations the fit converged.
final sum of squares of residuals : 129509
rel. change during last iteration : -2.92958e-008

degrees of freedom    (FIT_NDF)                        : 17
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 87.2821
variance of residuals (reduced chisquare) = WSSR/ndf   : 7618.17

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a2              = 6774.72          +/- 331          (4.886%)
b2              = 333.823          +/- 26.2         (7.847%)

correlation matrix of the fit parameters:
                a2     b2
a2              1.000
b2             -0.645  1.000


gnuplot> f3(x) = (a3/x) + b3                       # Define the shape of the fitting function
gnuplot> a3 = 370.458; b3 = 145.278;               # Initial estimates for the fitting parameters a1 and b1
gnuplot> fit f3(x) 'SB_Terasort_10GB_a.dat' using 1:3 via a3, b3
iter      chisq       delta/lim  lambda   a3            b3
   0 1.6471726013e+04  0.00e+00 1.41e+02  3.704580e+02  1.452780e+02
   1 3.1568589670e+03 -4.22e+05 1.41e+01  3.900795e+02  1.071553e+02
   2 2.8499791382e+03 -1.08e+04 1.41e+00  4.047789e+02  9.980014e+01
   3 2.8499767627e+03 -8.34e-02 1.41e-01  4.048284e+02  9.978109e+01
iter      chisq       delta/lim  lambda   a3            b3

After 3 iterations the fit converged.
final sum of squares of residuals : 2849.98
rel. change during last iteration : -8.33513e-007

degrees of freedom    (FIT_NDF)                        : 7
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 20.1777
variance of residuals (reduced chisquare) = WSSR/ndf   : 407.14

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a3              = 404.828          +/- 23.28        (5.75%)
b3              = 99.7811          +/- 8.587        (8.605%)

correlation matrix of the fit parameters:
                a3     b3
a3              1.000
b3             -0.622  1.000
```

Fig. 24   Fitting Spark-Bench Terasort preview data points to the function f(x) = (a/x)+b.

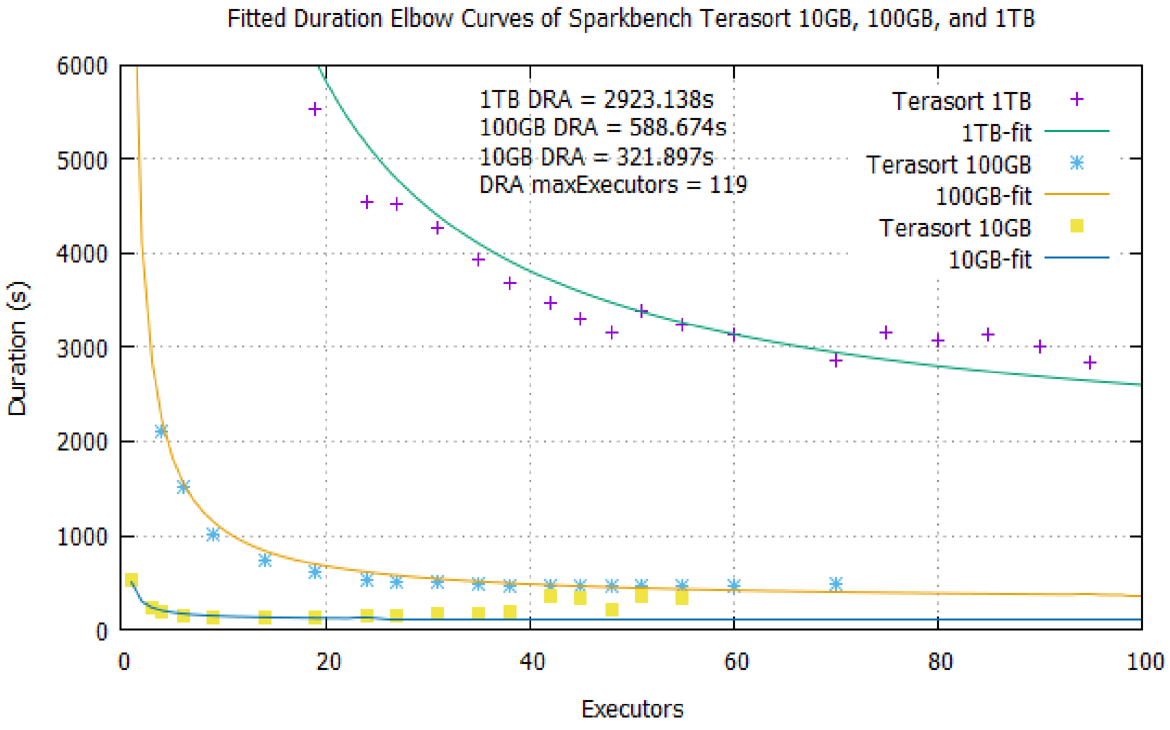


Fig. 25 Fitted runtime vs. executors' elbow curves of SparkBench Terasort 10GB, 100GB, and 1TB of data, and their runtimes with Dynamic Resource Allocation (DRA) enabled.

4. *Input the fit parameter a to the BToP algorithm (Fig. 2) to obtain the recommended optimal number of executors for a workload* (Fig. 26).

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* (Fig. 26).

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* (Fig. 26-27).

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

$$\frac{d(acceleration)}{d(executors)} = \frac{d(acceleration)}{d(slope)} \frac{d(slope)}{d(executors)}$$

```
# 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.
```

Fig. 26 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.

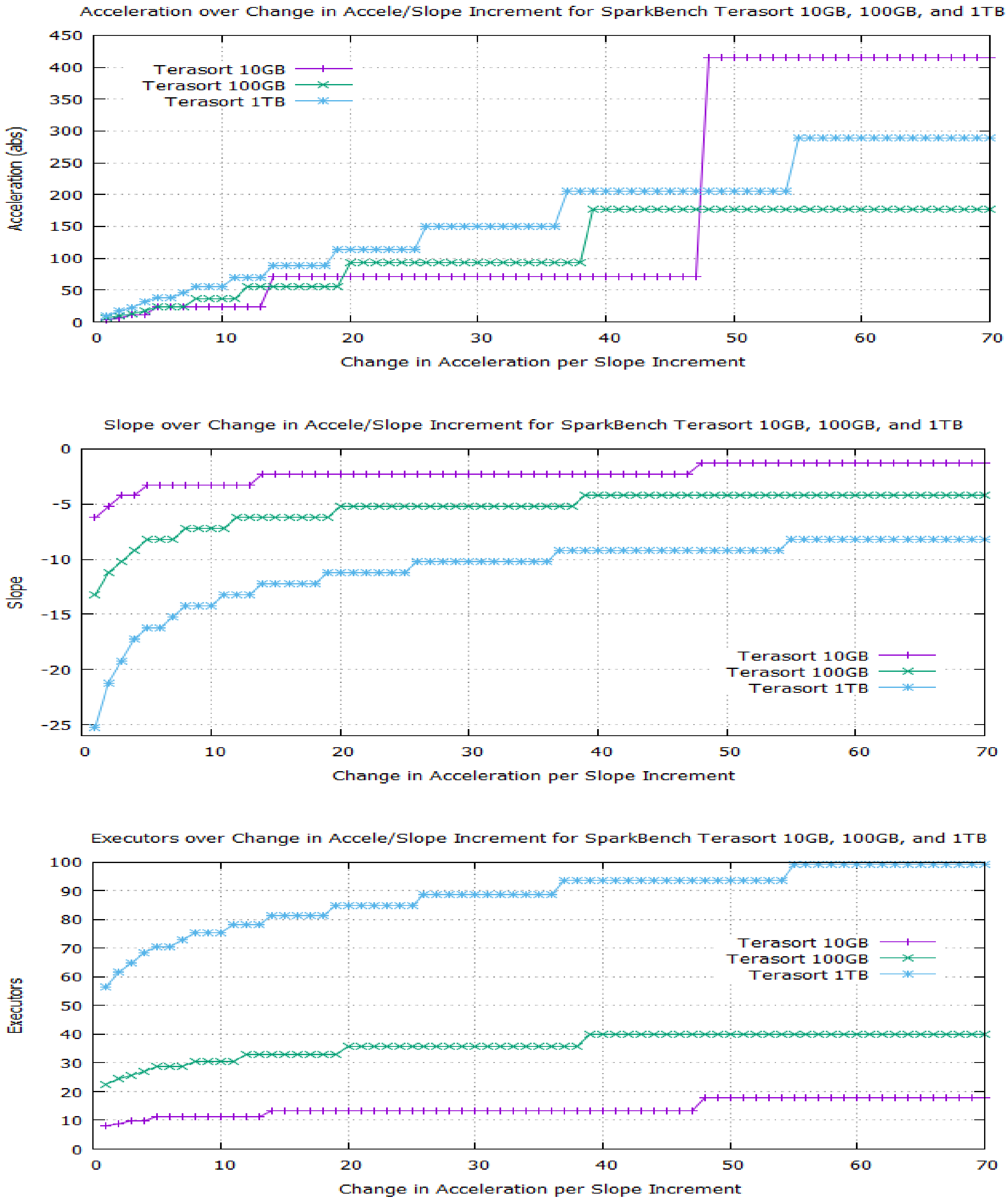


Fig. 27 The optimal numbers of executors for Spark-Bench Terasort 10 GB, 100 GB, and 1 TB workloads are identified by the major plateaus lasting at least seven increments on the graphs. The algorithm searches for break points in the changes in acceleration and outputs: (a) recommended acceleration, (b) corresponding slope and (c) executors versus change in acceleration per slope increment.

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 (Fig. 26-27).

Finally, the algorithm searches for all major plateaus lasting at least seven 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 duration of 7 executors specified for a major plateau could be adjusted by user to further fine tune the search for the first recommended optimal executor resource value.

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 (Fig. 26-27).

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 runtime. The results, as analyzed in section 4.3, prove that Spark using the BToP method could still consistently outperform Spark with DRA enabled in all three tested workloads 10GB, 100GB, and 1TB despite the sophistication and robustness of Spark's DRA mechanism.

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

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

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.

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

4.3.1 Performance Gain

From the graphs of duration vs executors of Spark-Bench Terasort, we notice that the durations of 10GB, 100GB, and 1TB workloads eventually get longer instead of shorter even though the number of executors continue to increase after 20, 60, and 95 executors, 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 the small workloads of 10GB and 100GB than the large 1TB workload due to growing framework overhead and resource contention. As such, for the 10GB 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 100GB 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 1TB 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 (Fig. 23).

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 elbow turn 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 10GB workload and 119-120 executors for the 100GB and 1TB workloads. The Spark-Bench Terasort tests with DRA enabled result in 321.897s for 10GB, 588.674s for 100GB, and 2,923.138s for 1TB (Fig. 25 and Table 1).

Spark-Bench Terasort	**DRA Enabled (s)**	**BToP Method (DRA Disabled)**		**Improved Performance (%)**
		Executors	**Duration (s)**	
10GB	321.897s	11	129.43	59.79
		13	122.06	62.08
		18	114.70	64.37
100GB	588.674s	33	513.50	12.77
		36	498.77	15.27
		40	477.73	18.85
1TB	2,923.138s	85	2,739.04	6.30
		89	2,694.84	7.81
		94	2,644.34	9.54
		99	2,608.56	10.76

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

Running Spark-Bench Terasort with the BToP method (DRA disabled), the recommended optimal numbers of executors are 11-13-18 executors for 10GB, 33-36-40 executors for 100GB, and 85-89-94-99 executors for 1TB. By extracting the corresponding runtimes from the fitted elbow curves of Spark-Bench Terasort, we obtain the durations of Spark with BToP method (DRA disabled) for 10GB, 100GB, and 1TB workloads as shown in Table 1 (Fig. 25).

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 1TB, up to 18.85% for 100GB, and up to 64.37% for 10GB (Table. 1).

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 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.

4.3.2 Energy Saving

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

$$\text{Energy}(N) = [\text{Time}_{\text{run}}(N) * \text{Power}_{\text{active}}(N)] + [\text{Time}_{\text{idle}} * \text{Power}_{\text{Ade}}]$$

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 [8, 32]. 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 [29, 33].

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 1TB, 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 1TB to estimate its energy consumption.

For Spark-Bench Terasort of 1TB, 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:

$$E(15) = [2{,}923.14\text{ s} * (250\text{ W} * 15)] + [(3{,}000\text{ s} - 2{,}923.14\text{ s}) * 235\text{ W}]$$
$$= 10{,}979{,}37.1\text{ j} = 3{,}049.95\text{ Wh per job}$$
$$E(13) = [2{,}608.56\text{ s} * (250\text{ W} * 13)] + [(3{,}000\text{ s} - 2{,}608.56\text{ s}) * 235\text{ W}]$$
$$= 8{,}569{,}808.4\text{ j} = 2{,}380.5\text{ Wh per job}$$

Hence, by provisioning executor resources with the BToP method, we reduce the energy consumption by about 21.95%. This translates to (0.66945 kWh saved per job) * [((365*24) h/yr / ((3000/3600) h/job)] = 7,037.26 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 $1,050.66 for the given 1TB compute job with arrival time of 3000 s [35] (Table 2).

From the output files of Spark-Bench Terasort of 10GB and 100GB, we observe that Spark with DRA enabled frequently ramps up and down within the range of 0 to 80 and 120, respectively, according to the need of each stage of the jobs. But the fluctuating resource utilization rarely reached the maximum available number of executors allocated for DRA in both 10GB and 100GB workloads. As previously rationalized for the 1TB workload, since their estimated energy consumptions would be inconsistent if they

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)**	**PerJob (%)**	**Per Year (kWh)**	**Per Year ($)**
10GB	80	10	398.36	18	3	188.37	52.71	2,207.41	329.57
100GB	120	15	770.60	40	5	330.53	57.11	4,626.02	690.66
1TB	120	15	3,049.95	99	13	2,380.5	21.95	7,037.26	1,050.66

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

were modeled after a moving target, we use their maximum numbers of executors assigned for DRA, 80 executors for 10GB workload and 120 executors for 100GB 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 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 10GB and 100GB 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 10GB compute job and $690.66 for the given 100GB compute job (Table 2).

Chapter 5

Conclusion and Future Work

Our proposed Best Trade-off Point (BToP) method for efficient resource provisioning in Hadoop offers a verifiable working method and algorithm with mathematical formulas to ascertain the optimal number of resources needed for performance efficiency. The recommended resource values will always be accurate since they are derived from actual sampled executions of each specific application and target system in use. Hadoop users no longer have to guess the required number of task resources for a MapReduce job from some unreliable rules of thumb. In addition, Spark users have an option to gain further performance improvement and energy saving for any Spark job running on Hadoop YARN by using the BToP method instead of simply relying on the built-in Dynamic Resource Allocation (DRA) mechanism. Although our experiments are conducted on a small-scale 24-node Hadoop cluster, our proposed BToP method should also work for larger workloads running on much bigger clusters of several thousand nodes in today's datacenters.

Spark-Bench Terasort test confirms that our proposed BToP method for efficient resource provisioning in Apache Spark consistently outperforms Spark with its built-in DRA 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 behaviors, the BToP method might be preferable in production environment 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 brief, our BToP method will work with any fundamental trade-off curves with an elbow shape, non-inverted or inverted, for making good decisions.

References

[1] Apache Hadoop. http://hadoop.apache.org

[2] Apache Hadoop 2.7.1. MapReduce Tutorial. Reducer: How Many Reduces? http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Reducer

[3] Apache Spark. http://spark.apache.org

[4] Apache Spark Dynamic Resource Allocation. http://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation

[5] D. Agrawal, A. Butt, K. Doshi, J.L. Larriba-Pey, M. Li, F.R. Reiss, F. Raab, B. Schiefer, T. Suzumura and Y. Xia. SparkBench–a spark performance testing suite. In *Technology Conference on Performance Evaluation and Benchmarking* (pp. 26-44). Springer, Cham., August 2015.

[6] S. Babu, Towards automatic optimization of MapReduce programs, in: Proceedings of the 1st ACM symposium on Cloud computing (2010)

[7] Y. Chen, A.S. Ganapathi, R. Griffith, R.H. Katz. A methodology for understanding mapreduce performance under diverse workloads. Tech. Rep. UCB/EECS-2010-135 EECS Department, University of California, Berkeley (2010)

[8] Y. Chen, L. Keys, R. Katz. Towards energy efficient mapreduce. EECS Department, Tech. Rep. UCB/EECS-2009-109 University of California, Berkeley (2009)

[9] Cloudera. How-to: Tune Your Apache Spark Jobs (Part 2). http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

[10] Cloudera. Spark Dynamic Allocation. 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

[11] Databricks. https://databricks.com/spark/about

[12] Databricks: Understanding your Apache Spark Application Through Visualization. https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html

[13] Datacenter Knowledge. http://www.datacenterknowledge.com/archives/2017/03/16/google-data-center-faq

[14] DZone/Big Data Zone. Using YARN API to Determine Resources Available for Spark Application Submission: Part II. https://dzone.com/articles/alpine-data-how-to-use-the-yarn-api-to-determine-r

[15] Gartner's Forecast of 25B IoT devices connected by 2020. http://www.gartner.com/newsroom/id/2905717

[16] Hadoop Wiki: HowManyMapsAndReduces. http://wiki.apache.org/hadoop/HowManyMapsAndReduces

[17] J. Hertzog, Z. Fadika, E. Dede, M. Govindaraju. Configuring a MapReduce framework for dynamic and efficient energy adaptation
2012 E 5th Int. Conference on CLOUD, IEEE (2012), pp. 914–92

[18]_H. Herodotou, F. Dong, S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. Proceedings of the 2nd ACM Symposium on Cloud Computing, ACM (2011)

[19] Hortonworks Data Platform. Section 1.11.1. Manually Calculate YARN and MapReduce Memory Configuration Settings. http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.9.1/bk_installing_manually_book/content/rpm-chap1-11.html

[20] Hortonworks. Managing CPU resources in your Hadoop YARN clusters, by Varun Vasudev. https://hortonworks.com/blog/managing-cpu-resources-in-your-hadoop-yarn-clusters/

[21] IBM Hadoop Dev / Tech Tip / Spark / Beginner's Guide: Apache Spark Troubleshooting. https://developer.ibm.com/hadoop/2016/02/16/beginners-guide-apache-spark-troubleshooting/

[22]_K. Kambatla, A. Pathak, H. Pucha.: Towards optimizing hadoop provisioning in the cloud, in: Proc. of the First Workshop on Hot Topics in Cloud Computing, 2009

[23] S. Karanth, Mastering Hadoop. Advanced MapReduce. The Reduce task. 2014, p. 50

[24] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia: Learning spark: lightning-fast big data analysis. " O'Reilly Media, Inc.," 2015.

[25] R.T. Kaushik, M. Bhandarkar. Greenhdfs: towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster, in: Proceedings of the USENIX Annual Technical Conf., 2010, p. 109

[26] K.R. Krish, M.S. Iqbal, M.M. Rafique, A.R. Butt. Towards energy awareness in Hadoop. Proceedings of Fourth International Workshop on Network-Aware Data Management, IEEE Press (2014), pp. 16–22

[27] J. Koomey. Growth in Data Center Electricity Use 2005 to 2010. Analytics Press, Oakland, CA (2011)

[28] W. Lang, J.M. Patel. Energy management for mapreduce clusters. Proc. VLDB Endow., 3 (1–2) (2010), pp. 129–139

[29] J. Leverich, C. Kozyrakis. On the energy (in) efficiency of hadoop clusters. ACM SIGOPS Oper. Syst. Rev., 44 (1) (2010), pp. 61–65

[30] J. Lin, F. Leu, Y. Chen. Analyzing job completion reliability and job energy consumption for a general MapReduce infrastructure. J. High Speed Netw., 19 (3) (2013), pp. 203–214

[31] M. Li, J. Tan, Y. Wang, L. Zhang, V. Salapura. SparkBench: a comprehensive benchmarking suite for in memory data analytic platform Spark. Proceedings of the 12th ACM International Conference on Computing Frontiers, ACM (2015), p. 53

[32] P.P Nghiem and S.M. Figueira. Towards efficient resource provisioning in MapReduce. *Journal of Parallel and Distributed Computing*, *95*, pp.29-41, 2016.

[33] S. Rivoire, P. Ranganathan, C. Kozyrakis. A comparison of high-level full-system power models. HotPower, 8 (2008) 3–3

[34] Sparkbench: Benchmark Suite for Apache Spark.
https://sparktc.github.io/spark-bench/

[35] U.S. Energy Information Administration.
Electric Power Monthly Data for May 2017.
https://www.eia.gov/electricity/monthly/epm_table_grapher.php?t=epmt_5_06_a

[36] U.S. Energy Information Administration.
Electric Power Monthly Data for June 2015
http://www.eia.gov/electricity/monthly/epm_table_grapher.cfm?t=epmt_5_06_a

[37] A. Verma, L. Cherkasova, R.H. Campbell. Resource provisioning framework for mapreduce jobs with performance goals. Middleware 2011, Springer, Berlin, Heidelberg (2011), pp. 165–186

[38] X. Wang, Y. Wang, H. Zhu. Energy-efficient task scheduling model based on MapReduce for cloud computing using genetic algorithm. J. Comput., 7 (12) (2012), pp. 2962–2970

[39] T. White. Hadoop: The Definitive Guide. Yahoo Press (2010)

[40] J. Whitney and P. Delforge. Data center efficiency assessment. *Issue paper on NRDC (The Natural Resource Defense Council), 2014*.

[41] R. Yadav. *Spark Cookbook*. Packt Publishing Ltd., 2015.

[42] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, USENIX Association (2012) pp. 2–2

[43] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, in: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, Vol. 10, June 2010, p. 10