9-2016

Towards efficient resource provisioning in MapReduce

Peter Nghiem
Santa Clara University, pnghiem@scu.edu

Silvia M. Figueira
Santa Clara University, sfigueira@scu.edu

Recommended Citation
Nghiem, P. P., & Figueira, S. M. (2016). Towards efficient resource provisioning in MapReduce. Journal of Parallel and Distributed Computing, 95, 29–41. https://doi.org/10.1016/j.jpdc.2016.04.001

J. Parallel Distrib. Comput. 95 (2016) 29–41

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Towards efficient resource provisioning in MapReduce

Peter P. Nghiem *, Silvia M. Figueira

Department of Computer Engineering, Santa Clara University, 500 El Camino Real, Santa Clara, CA 95053-0207, United States

HIGHLIGHTS

- Overprovisioning could lead to significant waste of computing resources and energy.
- Performance gain decreases quickly beyond the best trade-off point on elbow curve.
- Our algorithm for optimal resource provisioning is better than any rules of thumbs.
- Use dynamic job profiling with table of signatures to match optimal task resources.
- Efficient task provisioning saves energy and resources for jobs in multi-tenancy.

ARTICLE INFO

Article history:
Received 18 September 2015
Received in revised form
16 January 2016
Accepted 1 April 2016
Available online 12 April 2016

Keywords:
Hadoop MapReduce
Spark
Optimal resource provisioning
Energy efficiency
Runtime elbow curve
YARN

ABSTRACT

The paper presents a novel approach and algorithm with mathematical formula for obtaining the exact optimal number of task resources for any workload running on Hadoop MapReduce. In the era of Big Data, energy efficiency has become an important issue for the ubiquitous Hadoop MapReduce framework. However, the question of what is the optimal number of tasks required for a job to get the most efficient performance from MapReduce still has no definite answer. Our algorithm for optimal resource provisioning allows users to identify the best trade-off point between performance and energy efficiency on the runtime elbow curve fitted from sampled executions on the target cluster for subsequent behavioral replication. Our verification and comparison show that the currently well-known rules of thumb for calculating the required number of reduce tasks are inaccurate and could lead to significant waste of computing resources and energy with no further improvement in execution time.

1. Introduction

The energy consumption associated with datacenters has risen quickly, and server energy costs over its useful life can now exceed its original capital expenditure [18]. Google, Microsoft, Amazon, Facebook and Yahoo now have several hundred thousand to over a million servers in their datacenter infrastructure. A large portion of datacenter workloads is processed by Hadoop MapReduce, a popular de facto standard framework for Big Data processing, which has been adopted by these world's leading cloud computing providers and top Big Data companies, among many other organizations and institutions. As such, there has been a growing amount of research work dedicated to making MapReduce more energy efficient. However, the issue of how to decide on the right number of task resources for different workloads still has not been resolved. Therefore, Hadoop developers and users have 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.

Several research groups have worked on the performance and energy efficiency of Hadoop MapReduce. Krish et al. [17] 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. [11] suggest a MapReduce framework configuration to evaluate node power consumption status and dynamically shift work toward more energy efficient node. Leverich and Kozyrakis [20] 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 [19] 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 [16] place classified data into two logical zones of HDFS, where 26% energy consumption reduction is achieved from cold zone power

* Corresponding author.
E-mail addresses: pnghiem@scu.edu (P.P. Nghiem), sfigueira@scu.edu (S.M. Figueira).

http://dx.doi.org/10.1016/j.jpdc.2016.04.001

management, and there is room for further energy saving in the under-utilized hot zone. Lin et al. [21] 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. [26] 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. [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 paper, we present a solution for finding the best trade-off point in performance and energy efficiency. We propose a standard 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.

This paper makes the following contributions:

- We develop 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.
- We 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.
- 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 workload to achieve performance efficiency while the numbers of reduce tasks suggested by the three currently popular rules of thumb are inaccurate leading to significant unintended waste of computing resources and energy as shown in Fig. 9 in Section 6.

The remainder of this paper is organized as follows: Section 2 presents a brief background knowledge on MapReduce. Section 3 introduces MapReduce resource provisioning and related research efforts. Section 4 presents our algorithm for optimal resource provisioning. Section 5 discusses the design, analysis and implementation of our algorithm. Section 6 compares the accuracy and energy efficiency of our algorithm to three popular rules of thumb. Finally, Section 7 concludes our work and proposes future research on other types of applications and parallel processing frameworks running on Hadoop YARN including Apache Spark with its dynamic resource allocation feature.

2. Background knowledge on MapReduce

Apache Hadoop [1,27] 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 three key modules: the Hadoop Common which provides file systems and OS level abstractions, the Hadoop Distributed File System (HDFS), and the Hadoop MapReduce engine with YARN (MR v. 2). 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.

HDFS, which is based on Google File System (GFS), supports large-scale data processing workloads and reliable data storage of several TB on clusters of commodity hardware. It features scalability, high availability, fault tolerance, flexible access, load balancing, tunable replication, and security. 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. 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 with 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.

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. 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.

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>.

YARN splits the responsibilities of job tracker and task tracker in MapReduce v.1 into four separate entities in MapReduce v.2: (1) The Resource Manager has a built-in scheduler, which allocates resources across all applications based on the applications' resource requirements. (2) The MR Application Master, 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 Node Manager, 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. 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.

3. Optimization of task 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. 1). 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 work 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 without knowing the properties of the actual job being processed, its input data, and resource allocation. Herodotou et al. [12] introduce the Elastisizer 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. [25] 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. [14] 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.

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 right 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 [6,14]. 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 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 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 particular job actually 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 [3]. 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. 9 in Section 6.

4. Algorithm for optimal resource provisioning

We have developed an algorithm (Fig. 2) 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 right number of tasks for a job to run in any specific Hadoop cluster. Before applying the algorithm, Hadoop

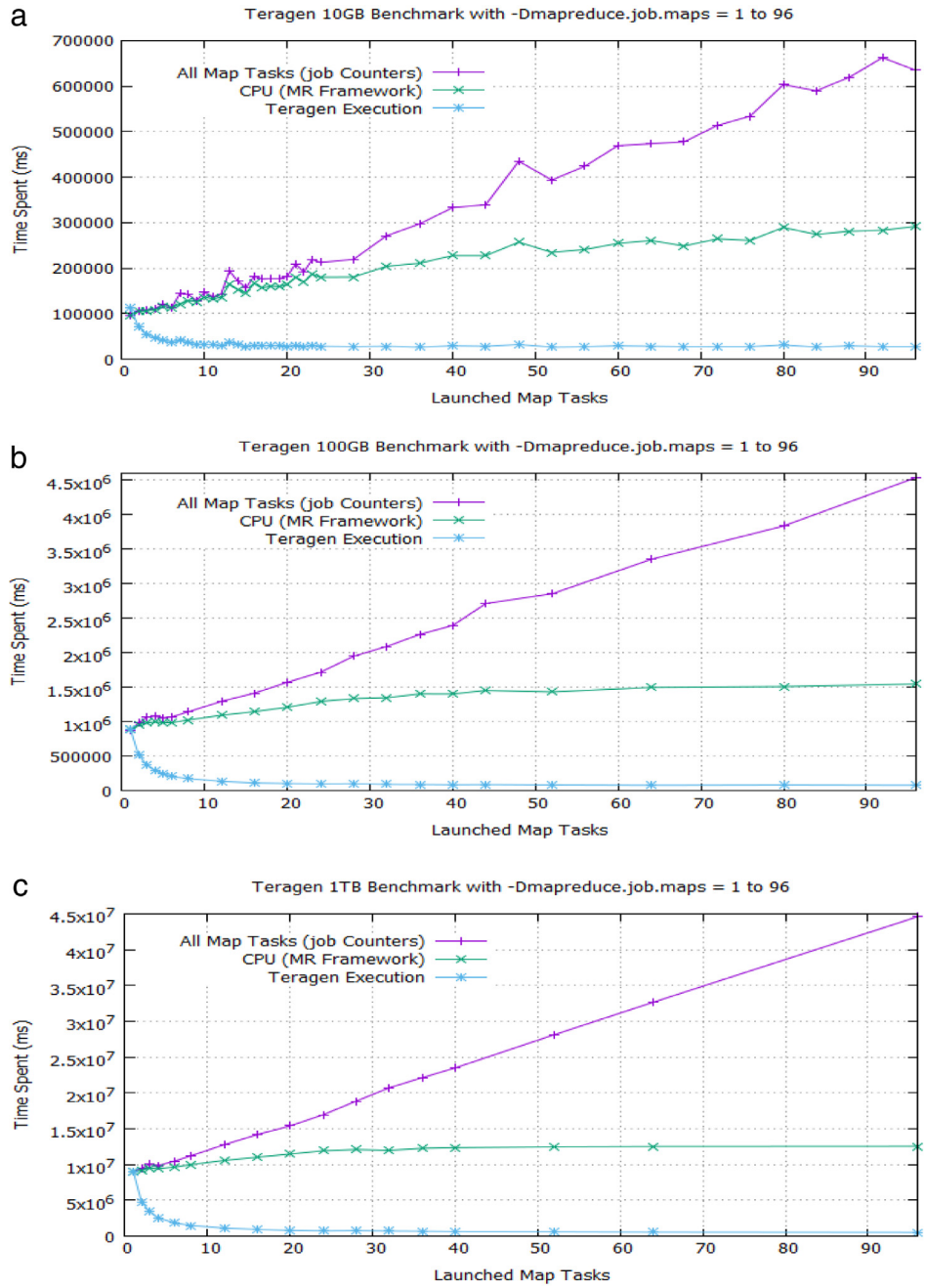

Fig. 1. Graphs of time spent by all map tasks, CPU, and Teragen execution versus number of launched map tasks. The runtime elbow curves of Teragen (a) 10 GB, (b) 100 GB and (c) 1 TB workloads plotted at different y-axis scales all appear to have the best trade-off points for performance efficiency at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

users should first get some sampled executions from their target production system as reference points 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 curve and outputs it as recommended optimal number of tasks for a workload (Figs. 5 and 6 in Section 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 tasks might be necessary to maintain accuracy and precision. Once a database of signatures has been established, dynamically submitted jobs with different workloads could 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

```
Algorithm BestTrade-offPoint
Input:  Parameter a for a job 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(1/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. Algorithm to ascertain the best trade-off point on a runtime elbow curve for optimal resource provisioning.

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 have to 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 preview stage. It will be difficult and far less accurate to generally provision for a class of applications due to the diversified nature of MapReduce applications. Chen 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, [7] exclude system characteristics and system behavior from the workload description. [7]'s 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 particular application running on a specific system.

5. Design, analysis and implementation

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. 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 tasks for efficient use of available computing resources (Figs. 1 and 3).

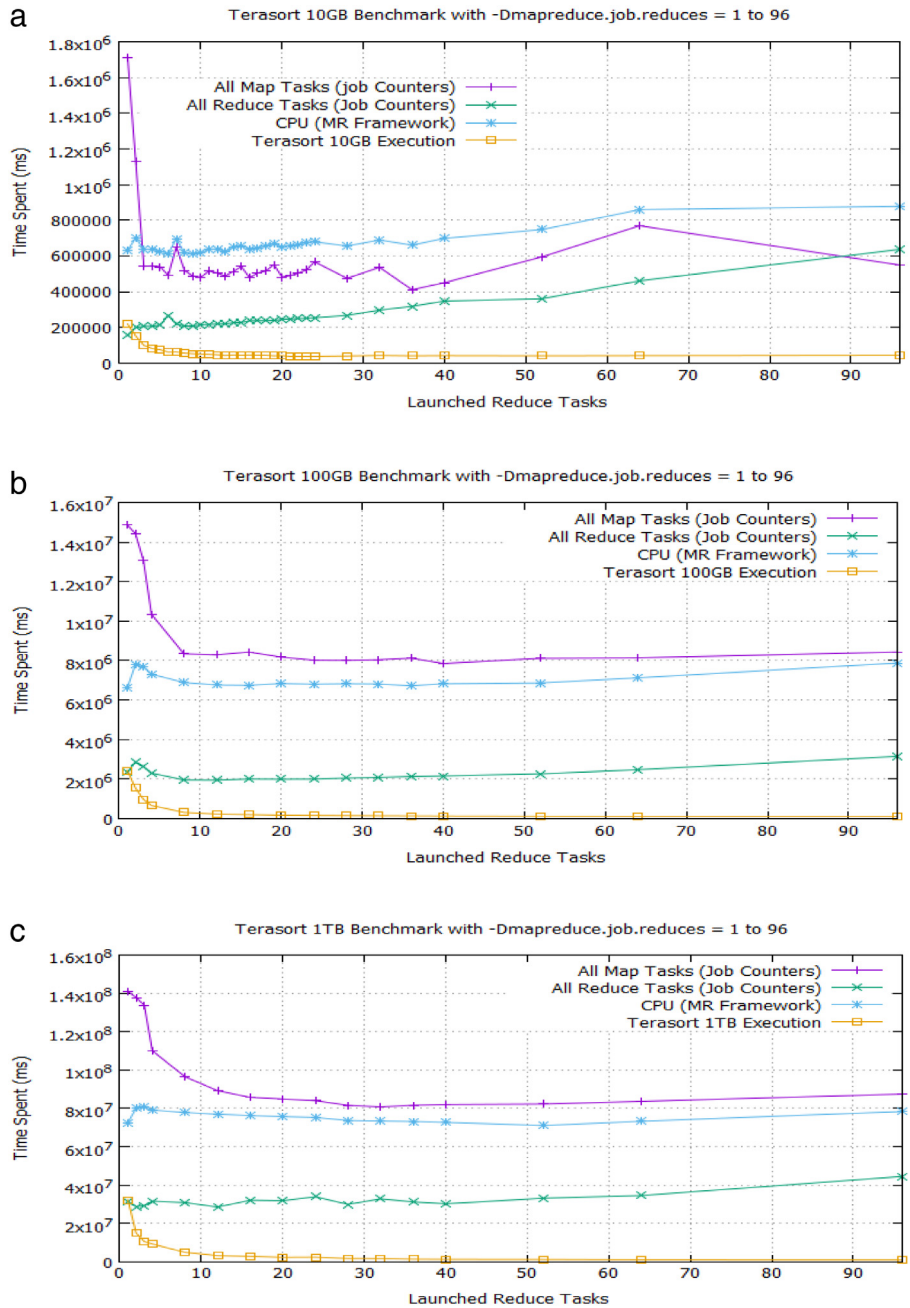

Fig. 3. Graphs of time spent by all map tasks, all reduce tasks, CPU, and Terasort execution versus number of launched reduce tasks. The runtime elbow curves of Terasort (a) 10 GB, (b) 100 GB and (c) 1 TB workloads plotted at different *y*-axis scales all appear to have the best trade-off points for performance efficiency at around 10 reduce tasks. But that is disproved by our algorithm as a visual misperception of different granularities at low magnification.

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 (Figs. 1 and 3), 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 (Figs. 1 and 3), 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

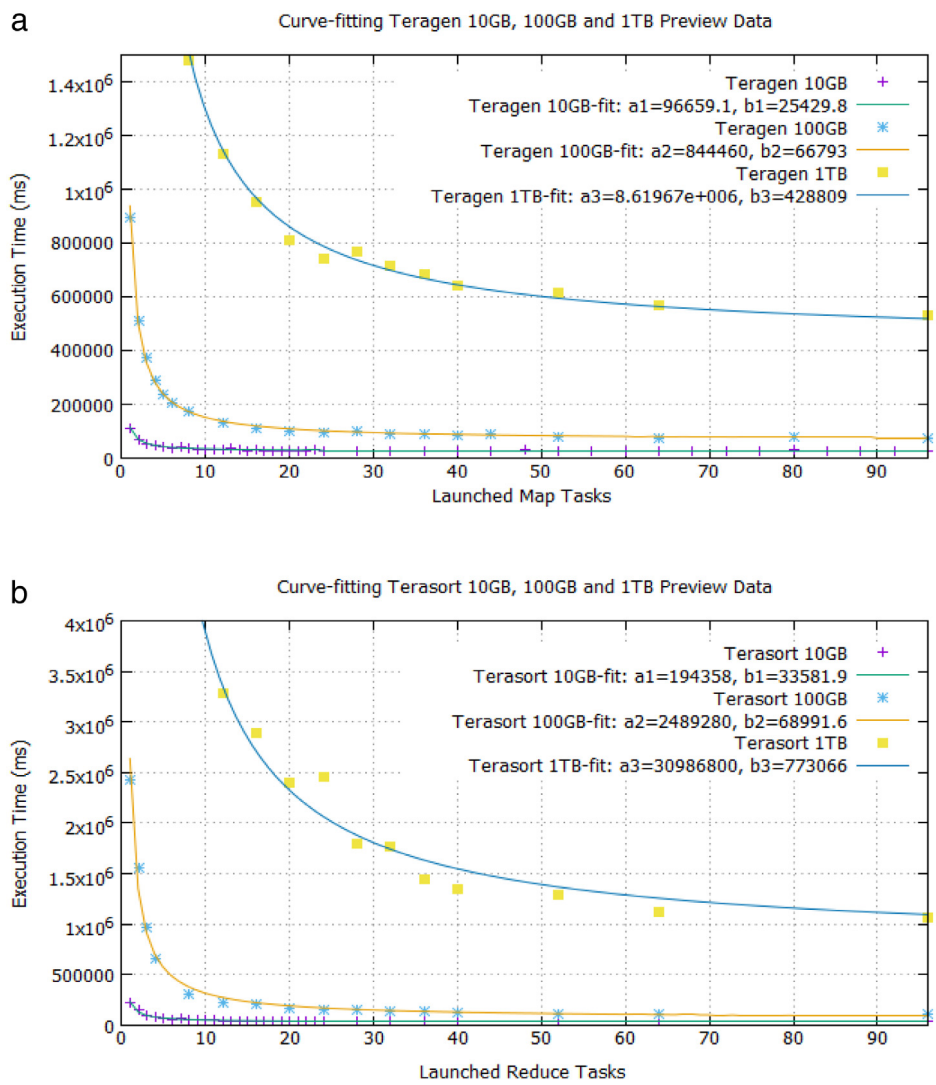

Fig. 4. Fitted runtime elbow curves of (a) Teragen and (b) Terasort 10 GB, 100 GB and 1 TB workloads versus number of launched map/reduce tasks, and their fit parameters a and b in the graph function $f(x) = (a/x) + b$.

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.

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

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

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.

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

From (1),

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

and therefore,

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

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 $f(x) = (a/x) + b$. We then plot the three fitted elbow curves of execution time versus launched tasks for Teragen and Terasort 10 GB, 100 GB and 1 TB workloads (Fig. 4).

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.

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

```
# 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. 5. Applying the algorithm for best trade-off point 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.

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

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

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 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 (Figs. 2, 5 and 6). 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 (Figs. 7 and 8). Our program then outputs the exact optimal numbers of tasks recommended for different workloads (Figs. 5 and 6). 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

```
# 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. 6. Applying the algorithm for best trade-off point to 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.

tasks beyond the recommended range does not necessarily translate into any further performance gain in execution time.

In summary, the sequential steps to implement our method for optimal resource provisioning in a computer system is outlined as follows:

1. Complete the configuration and fine-tuning of the computer system targeted for calibration.
2. Collect preview job performance data from sampled executions on the same target computer system.
3. Curve-fitting the preview data to obtain the fit parameters a and b in the elbow curve function $f(x) = (a/x) + b$, where x is the number of tasks.
4. Input the fit parameter a to our Best-Trade-off-Point algorithm to obtain the recommended optimal number of tasks for a workload.
5. Repeat steps 2–4 to build a database of resource consumption signatures with different workloads for subsequent job profiling.
6. If there is any major change to step 1, repeat steps 2–5 to recalibrate the database of resource consumption signatures.
7. Use the database of resource consumption signatures to match dynamically submitted production jobs to their optimal number of tasks for efficient resource provisioning.

6. Verification and comparison to rules of thumb

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 (Figs. 5 and 6). Their pinpoint accuracy and integrity are verified by the fitted runtime elbow curves generated from their sampled executions (Fig. 4). 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. 9).

From our algorithm for optimum, the recommended numbers of reduce tasks range 4–9 for Terasort 10 GB, 11–14 for Terasort 100 GB, and 24–27–31 for Terasort 1 TB (Fig. 6). These values are not only optimal but also accurate, as verified by the fitted elbow curves in Fig. 9, since they are derived from the sampled job runtimes of the actual cluster-based system targeted for calibration.

According to rule of thumb (A) [15] 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 a bit

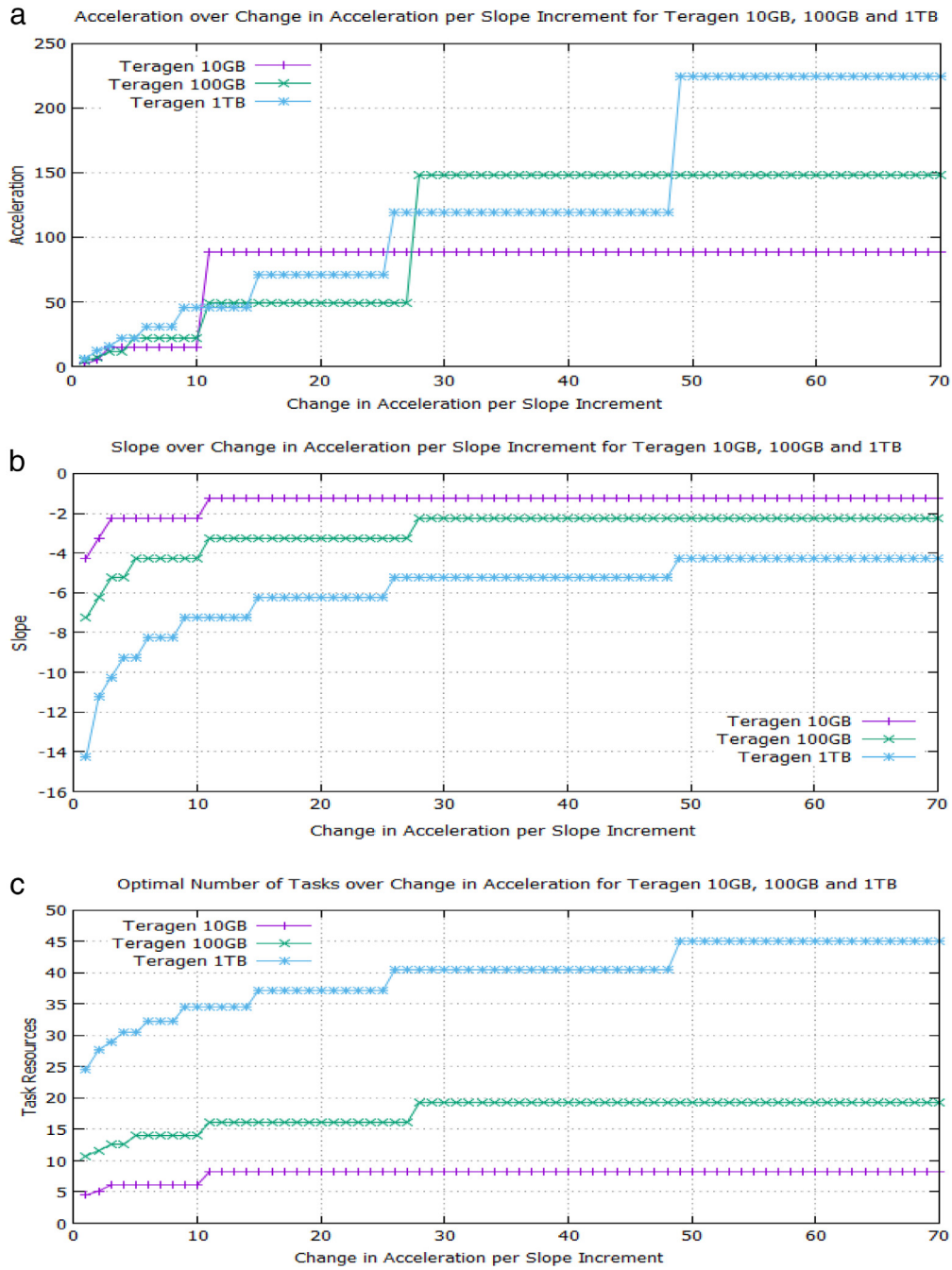

Fig. 7. The optimal number of tasks for Teragen 10 GB, 100 GB and 1 TB workloads are identified by the major plateaus lasting at least eight increments on the graphs. The algorithm searches for break points in the changes in acceleration and outputs: (a) recommended acceleration, (b) corresponding slope and (c) task resources versus change in acceleration per slope increment.

too wide but also too high causing significant energy waste for no further gain in performance, particularly for 1 TB workload (Fig. 9).

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 $= \min(2 * \text{number of CPU cores}, 1.8 * \text{number of disks}, \text{Total available memory/Minimum container size}) = \min(2 * 4, 1.8 * 2, (32\text{-}6 \text{ reserved for system}) \text{ GB}/2 \text{ GB}) = 3.6$, and the recommended minimum container size for total RAM per node above 24 GB is 2048 MB [13]. These suggested numbers of reducers derived solely from the hardware architecture specifications, without taking into consideration the 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. 9).

Under Rule of thumb (C) [3], the ideal 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

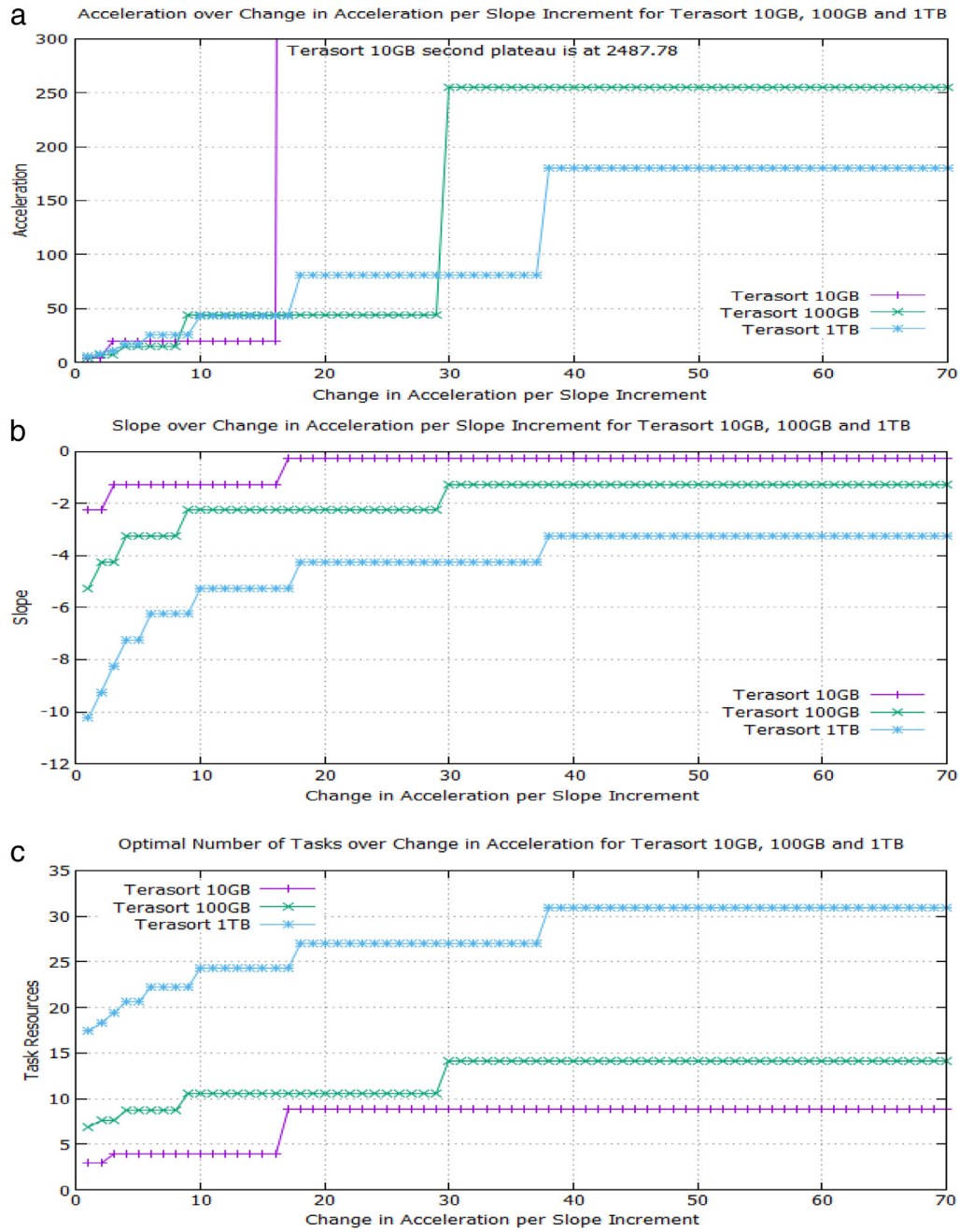

Fig. 8. The optimal numbers of tasks for Terasort 10 GB, 100 GB and 1 TB workloads are identified by the major plateaus lasting at least eight increments on the graphs. The algorithm searches for break points in the changes in acceleration and outputs: (a) recommended acceleration, (b) corresponding slope and (c) task resources versus change in acceleration per slope increment.

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. 9).

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. The energy consumption per job can be computed from the linear sum multiplying job duration by active power and idle duration 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 [20,23].

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

To quantify the potential saving in using our algorithm, we compare the highest recommended numbers 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 [13].

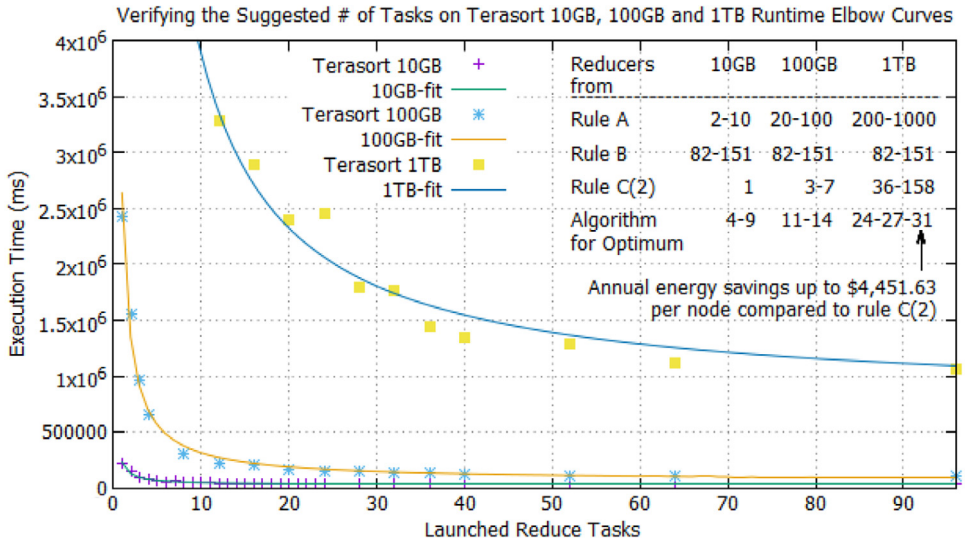

Fig. 9. Fitted elbow curves of Terasort 10 GB, 100 GB and 1 TB workloads from sampled executions verify the accuracy of our algorithm for optimal resource provisioning in contrast to the unreliable number of reducers calculated from three popular rules of thumb (A, B and C(2)), which could lead to significant waste of computing resources and energy.

For an active power consumption per node of 250 W, idle power of 235 W, and an average job arrival time of 2000 s:

$$E(9) = [1773\text{ s} * (250\text{ W} * 9)] + [(2000\text{ s} - 1773\text{ s}) * 235\text{ W}] = 4042.595\text{ kJ} = 1.123\text{ kWh per job}$$

$$E(44) = [969\text{ s} * (250\text{ W} * 44)] + [(2000\text{ s} - 969\text{ s}) * 235\text{ W}] = 10,901.285\text{ kJ} = 3.028\text{ kWh per job.}$$

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, May 2015 average retail price of electricity to commercial customers in California of $0.1482 per kWh, this amounts to an annual saving of $4451.63 per compute node [24].

From the table of number of reducers suggested by different methods under assessment for Terasort 10 GB, 100 GB and 1 TB workloads (Fig. 9), the potential energy savings could be even much larger if we compare 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.

7. Conclusion and future work

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 in Yahoo datacenter, the amount of aggregate annual energy saving will be very significant, up to several million dollars.

Our algorithm for optimum should also work for many other types of parallel processing frameworks running on Hadoop YARN beyond MapReduce such as Apache Spark [4,10,29,28], which has recently gained its momentum of popularity for in-memory processing of Big Data analytic applications with better sorting performance for large clusters. Spark, which can access to HDFS dataset without being tied to the two-stage MapReduce paradigm, also supports running application JARs in HDFS. Our approach and algorithm for optimum could be used to determine the initial number of executors required for a job at the inceptive stage of Spark's dynamic resource allocation, an important feature available in Spark v.1.2 and up. Unlike a MapReduce task resource, which resides in a process and is immediately killed upon its completion, a Spark task, which is actually a thread residing in a process known as executor, is not released until the long running application is finished. As such, it is necessary for Spark to be able to acquire and release resources during runtime through its dynamic allocation. Spark could relinquish executors when they are no longer used and acquire executors when they are needed according to its mechanism to gracefully decommission an executor by preserving its state before its removal using timeout [22,5,9]. Such 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.

Since Spark could process large-scale data up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk, and it can dynamically adjust the number of executors allocated to the application based on the workload, the additional small gain in efficient

performance through a more optimized initial number of executors might not be desirable considering the extra work involved in building an extensive database of signatures for this further optimization. Users will have to weigh the benefits in each specific case to determine applicability and best practices. In general, our algorithm to compute the best trade-off point for optimization of resource provisioning is applicable whenever there is a runtime elbow curve.

References

[1] Apache Hadoop. http://hadoop.apache.org (12/24/15).
[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 (09/09/2015).
[3] Hadoop Wiki: HowManyMapsAndReduces. http://wiki.apache.org/hadoop/HowManyMapsAndReduces (09/09/2015).
[4] Apache Spark. http://spark.apache.org (12/24/15).
[5] Apache Spark Dynamic Resource Allocation. http://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation (12/24/15).
[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 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 (12/24/15).
[10] Databricks. https://databricks.com/spark/about (12/24/15).
[11] J. Hartog, Z. Fadika, E. Dede, M. Govindaraju, Configuring a MapReduce framework for dynamic and efficient energy adaptation, in: 2012 IEEE 5th Int. Conference on CLOUD, IEEE, 2012, pp. 914–921.
[12] H. Herodotou, F. Dong, S. Babu, No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics, in: Proceedings of the 2nd ACM Symposium on Cloud Computing, ACM, 2011.
[13] Hortonworks Data Platform. Section 1.11.1. Manually Calculate YARN and MapReduce Memory Configurating Settings. http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.9.1/bk_installing_manually_book/content/rpm-chap1-11.html (09/09/2015).
[14] 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.
[15] S. Karanth, Mastering Hadoop. Advanced MapReduce. The Reduce task. 2014, p. 50.
[16] 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.
[17] K.R. Krish, M.S. Iqbal, M.M. Rafique, A.R. Butt, Towards energy awareness in Hadoop, in: Proceedings of Fourth International Workshop on Network-Aware Data Management, IEEE Press, 2014, pp. 16–22.
[18] J. Koomey, Growth in Data Center Electricity Use 2005 to 2010, Analytics Press, Oakland, CA, 2011.
[19] W. Lang, J.M. Patel, Energy management for mapreduce clusters, Proc. VLDB Endow. 3 (1–2) (2010) 129–139.
[20] J. Leverich, C. Kozyrakis, On the energy (in) efficiency of hadoop clusters, ACM SIGOPS Oper. Syst. Rev. 44 (1) (2010) 61–65.
[21] 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) 203–214.
[22] M. Li, J. Tan, Y. Wang, L. Zhang, V. Salapura, SparkBench: a comprehensive benchmarking suite for in memory data analytic platform Spark, in: Proceedings of the 12th ACM International Conference on Computing Frontiers, ACM, 2015, p. 53.
[23] S. Rivoire, P. Ranganathan, C. Kozyrakis, A comparison of high-level full-system power models, HotPower 8 (2008) 3–3.
[24] 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 (09/09/2015).
[25] A. Verma, L. Cherkasova, R.H. Campbell, Resource provisioning framework for mapreduce jobs with performance goals, in: Middleware 2011, Springer, Berlin, Heidelberg, 2011, pp. 165–186.
[26] 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) 2962–2970.
[27] T. White, Hadoop: The Definitive Guide, Yahoo Press, 2010.
[28] 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, in: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, USENIX Association, 2012, pp. 2–2.
[29] 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.

Peter P. Nghiem is a seasoned IT and Semiconductors professional with over two decades of experience in the high-tech industry. He has previously held various positions in design engineering, product marketing management and national sales management at AMD, Fujitsu, Raytheon, Toshiba, NEC and Adastra JUMPtec (Kontron). He holds a B.S. with honors in Electrical and Electronics Engineering, from California State University, Sacramento, an M.S. in Engineering Management, and an M.S. in Computer Science and Engineering, from Santa Clara University. He does Ph.D. thesis research in Computer Engineering at Santa Clara University in the area of energy-efficient computing and Big Data.

Silvia M. Figueira received the B.S. and M.S. degrees in Computer Science from the Federal University of Rio de Janeiro (UFRJ), Brazil, and the Ph.D. degree also in Computer Science from the University of California, San Diego. She is an Associate Professor of Computer Engineering at Santa Clara University. Her research is in the area of performance evaluation and prediction, recently with a focus on energy efficiency. She is also the Director of the SCU Frugal Innovation Lab, in which she leads the Mobile Lab and advises students working on mobile applications for under-served communities and emerging markets.