SLA-aware Provisioning and Scheduling of Cloud Resources for Big Data Analytics

Mohammed Alrokayan, Amir Vahid Dastjerdi, and Rajkumar Buyya
Cloud Computing and Distributed Systems (CLOUDS) Laboratory,
Department of Computing and Information Systems,
The University of Melbourne, Parkville, Victoria 3010, Australia
Emails: m.alrokayan@student.unimelb.edu.au, amir.vahid@unimelb.edu.au, rbuyya@unimelb.edu.au

***Abstract*—The stunning growth in data has immensely impacted organizations. Their infrastructure and traditional data management system could not keep up to scale of Big Data. They have to either invest heavily on their infrastructure or move their Big Data analytics to Cloud where they can benefit from both on-demand scalability and contemporary data management techniques. However, to make Cloud hosted Big Data analytics available to wider range of enterprises, we have to carefully capture their preferences in terms of budget and service level objectives. Therefore, this study aims at proposing a SLA and cost-aware resource provisioning and task scheduling approach tailored for Big Data applications in the Cloud. Current approaches assume that data is pre-stored in cluster nodes prior to deployment of Big Data applications. In addition, their focus is purely on task scheduling, and not virtual machine provisioning. We argue that in the Cloud computing context this is not applicable, because the nodes are provisioned dynamically (data cannot be pre-stored) and leaving provisioning to user may lead to under or over provisioning that can both lead to SLA or budget constraint violations. Therefore, in this study we first model the user request, which consist of Big Data analytics jobs with budget and deadline. Then, we model infrastructures as a list of data centers, virtual machines (offered in a pay-as-you-go model), data sources, and network throughputs. After that, to address the aforementioned issues, we propose and compare cost-aware and SLA-based algorithms which provision cloud resources and schedule analytics tasks.**

Keywords*—*Cloud Computing, Big Data Computing, Big Data.

I. Introduction

With prodigious growth in Web and social network data, more than ever before, it is clear that big data is coming. The big data wave is going to surface new opportunities for enterprises although constitutes series of challenges. The key issues is how to ingest Big Data and convert it to information and knowledge that possesses business value. However, this conversion is not economically viable for small to medium enterprises in a traditional infrastructure setting.

Cloud computing is growing rapidly as an extremely successful paradigm offering on-demand infrastructure, platform and software services to end users. Cloud computing, with its on-demand elasticity, and pay-as-you-go model, enables data intensive applications to dynamically provision resources and process large data sets in parallel, which was not economically feasible in a traditional data management systems. To pave the way for organizations to adopt Cloud-hosted Big Data analytics, we have to carefully consider their preferences in terms of budget and service level objectives through provisioning and scheduling phases, which is the focus of this study. There are three main deployment models in cloud computing: Private Cloud, Public Cloud, and Hybrid Cloud [1]. Our focus is on Hybrid Clouds, where applications are deployed and run across private and public Clouds in seamless manner.

There is no single tool that provides a complete solution for Big Data analytics. We use the Lambda architecture [2] to tackle the problem of Big Data computing via three layers, namely batch, serving, and speed layer. In this paper, we restrict our focus to the batch layer that is responsible for running a function on a the whole dataset to build batch views which are indexed and later are utilized by other layers to compute the final query result. Scalability, simplicity, and fault-tolerance are among desired properties of the batch layer. We utilized MapReduce [3] for the batch layer as it possesses the aforementioned properties and is capable of processing large set of data in parallel to overcome disk I/O bottlenecks.

The majority of SMEs are constantly looking for cutting edge technologies and solutions to accomplish objectives of the company more efficiently and at the minimum cost and big data processing via Cloud resources is not an exception. There are many SLA-based Big Data computing and MapReduce scheduling studies [4], [5], [6], [7] in the Cloud context, however they do not provision Cloud resources dynamically. Instead, they have the resources already pre-provisioned (static) on a private Cloud, which form a virtual cluster. We argue that Cloud resources should be provisioned dynamically and on-demand based on the application workload and the size of the data. This introduces new challenges, namely: a) how many and which type of cloud resources to provision; b) which private infrastructure or public cloud provider to select for a given request with budget and deadline constraints; and c) given that data is geographically distributed, which resources should be chosen that minimize the data transfer and processing costs.

Our major contributions are summarized as follows: 1) a model for SLA-based resource provisioning and tasks scheduling for Big Data processing in cloud environments; 2) an SLA-based and cost minimization algorithm to provision cloud resources and schedule MapReduce tasks in the batch layer of Lambda architecture, and a technique for reducing the size of the search space; 3) an approach to enable the algorithm to run in parallel taking the advantage of multi-core system to find an optimal solution; and 4) the design and development of a new extension to CloudSim to evaluate and compare the algorithms.

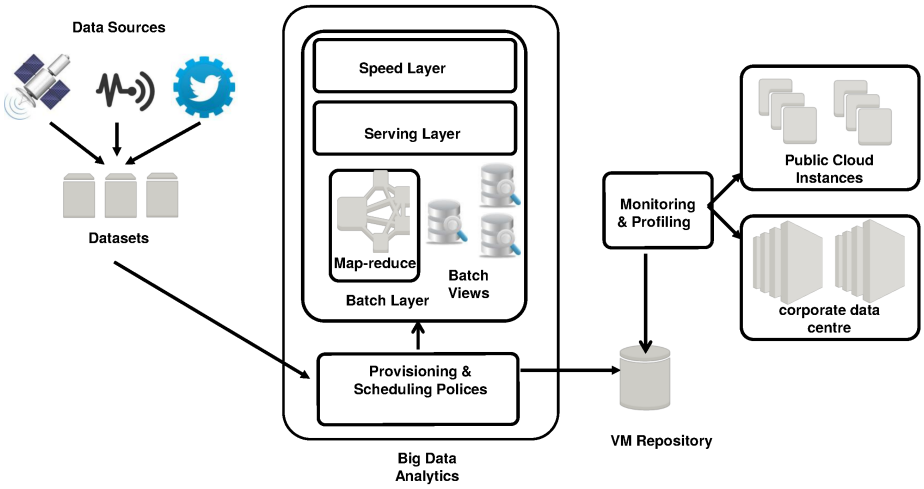

Fig. 1. The Proposed Architecture for Cloud-hosted Big-Data Analytics

The rest of the paper is organized as follows: In the next section, we position our work in the context of Cloud-hosted Big Data analytics while we are describing the proposed architecture. Section III presents the MapReduce on cloud model for the batch layer and discuses formulation of the problem, objectives, and constraints. Next, the proposed algorithm is described in Section IV following by performance evaluation of the algorithms and experiments' results in Section V. We present related works and compare them with our approach in section VI. We conclude the paper in Section VII with ideas on future directions.

II. Architecture

The proposed architecture is depicted in Figure 1. It is based on the Lambda architecture [2] and its main components are explained below:

Data Sources and Datasets- There are more objects than humans connected to the Internet, and their numbers are growing rapidly. These objects are called **Data Sources** and can send what they have sensed from different locations and environments. Datasets are raw data collected (eg. sensors and social media feeds) from data sources and are source of truth.

Batch Layer- When it comes to Big Data analytics, executing a random query on the whole dataset in real time can be computationally expensive. This problem can be alleviated by the use of batch views and pre-computed results to speed up query execution. The role of Batch Layer is to generate batch views from datasets.

Map-Reduce Component- MapReduce On Cloud is an attractive model for enterprises to build batch views due to its flexibility, agility, and low cost. Apache Hadoop is among the most popular implementations of MapRedcue. However, the proposed model in this study is not Hadoop compatible. The reason is Hadoop's schedulers are designed for static cluster of homogeneous machines in a single datacentre, while our model considers heterogeneous virtual machines (VM) across clouds. MapReduce consists of three phases: Map, Shuffle and Sort, and Reduce. MapReduce data is presented in key-value pairs for the mappers in the map phase for processing. Each mapper process a block of key-value pairs, and the size of the block is defined by the user. Mappers emit processed data in key-value pairs, generating intermediate data for the reducers in the reduce phase to aggregate the values. Shuffle and sort phase groups the values with the same key, and sort the keys. Reducers receive intermediate data with a set of values for each key for aggregation, and send the result as batch views. We consider the following characteristics for MapReduce in our architecture : 1) The reduce phase can not start before all map tasks finished. 2) Map tasks are almost homogeneous because they have the same function and almost same data block size. 3) Reduce tasks are heterogeneous because each reduce task process different size of data based on the emitted data (intermediate data) from the map phase. 4) A reduce task can be scheduled in the same map node to reduce the intermediate data transfer time and cost between nodes. 5) Completed map tasks can start sending intermediate data to reduce nodes even before all map tasks finished. This can minimize network I/O bottleneck and to save on execution time as well.

Serving Layer and Speed Layer- The Serving layer typically consists of a database system which consumes batch views and facilitates complex queries. To this extent, we are capable of executing queries on the precomputed views, however to execute an arbitrary query on an evolving dataset in real time, we need the Speed Layer. Resource Scheduling and Provisioning policies for these two layers will be presented in our future works.

Monitoring and Profiling- This component consists of a collection of monitoring services that, together with benchmarking toolkits, extract and analyze performance statistics about public and private Cloud instances for a running or a benchmarking application. Profiling occurs while Data procession is running, or separately for a new version of application or instance to update and improve the collected statistics.

Resource Scheduling and Provisioning Component- In order to enable cloud-hosted MapReduce for application with SLA, we require the Cloud resource provisioning and scheduling component, which is a focus of this study. This component make decisions on both how many and which type of VM are required (provisioning phase) and which task

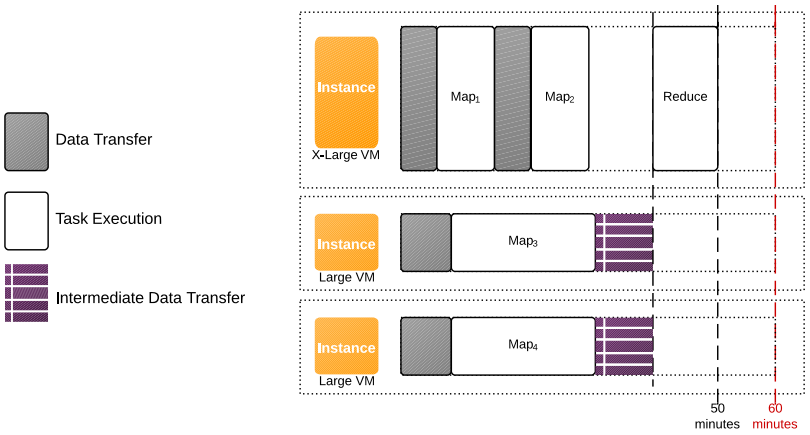

Fig. 2. An example of scheduling four map tasks and one reduce tasks in three virtual machines to achieve the deadline of 60 minutes

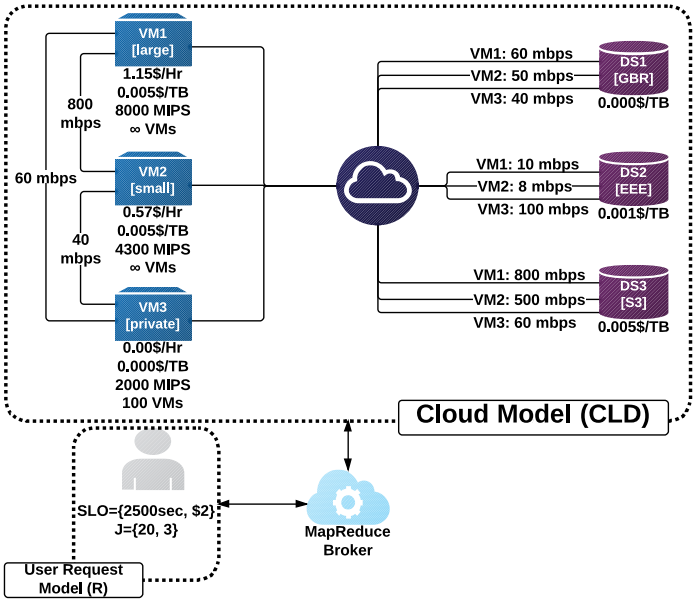

Fig. 3. Cloud (CLD) and User Request (R) Models

is running on which resource (scheduling phase) to meet the deadline for creating batch views as well as minimizing the cost. An efficient scheduling and provisioning component can guarantee higher level of accuracy for Big Data analytics by ensuring that there are always batch views built in time for Serving Layer. Figure 2 demonstrates an example of provisioning three resources (virtual machines) and running a MapReduce job with four map tasks and one reduce task on a Cloud within 50 minutes, where the deadline is 60 minutes.

III. System Model

There are two models in our system: Cloud Provider (CP) and User Request (R) Models. The Cloud model (shown in Equation 1) consist of: a set of data centres (DC), a set of data sources (DS), a matrix (T_{VM}) that shows throughputs between a virtual machine type and other types of all data centres in megabits per second (mbps), and a matrix (T_{DS}) that shows throughputs between a data source and a virtual machine type in megabits per second (mbps). Each DS has a cost for transferring the data from it (CT_{DS}) per terabyte. Each data centre has a set of virtual machine types (VM). Each virtual machine type has: cost of leasing (C_{VM}) per hour, the cost of transferring the data from it (CT_{VM}) per terabyte, the performance of the virtual machine ($MIPS$) in million instructions per second.

$$CP = \{[DS], [DC], [[T_{VM}]], [[T_{DS}]]\} \quad (1)$$
$$DC = \{[VM]\} \quad (2)$$
$$VM = \{C_{VM}, CT_{VM}, MIPS\} \quad (3)$$
$$DS = \{CT_{DS}\} \quad (4)$$

User request (R) model (shown in Equation 5) consists of: SLA objectives (SLO) and MapReduce Job (J). Each SLO includes: Budget (B) and Deadline (D).

$$R = \{SLO, J\} \quad (5)$$
$$SLO = \{B, D\} \quad (6)$$

A MapReduce Job (J) (shown in Equation 7) consists of a DS, a set of map tasks (M_{Task}), and reduce tasks (R_{Task}). Each M_{Task} consists of: Input Data Size ($DSize$), the required million CPU instructions(MI), and the size of the intermediate data ($IDSize_{R_{Task}}$) to each reducer R_{Task}. Each R_{Task} has the required instructions in million instructions (MI). The optimization algorithm receives a J as a part of R.

$$J = \{DS, [M_{Task}], [R_{Task}]\} \quad (7)$$
$$M_{Task} = \{DSize, MI, [IDSize_{R_{Task}}]\} \quad (8)$$
$$R_{Task} = \{MI\} \quad (9)$$

The objective is to satisfy the SLA requirements of the user while minimizing the total cost. The total cost of running the MapReduce job is denoted as TC. Given that total of n machines and m data sources are used, the TC can be computed as shown in Equation 10 where LP is the leasing period for a virtual machine and TD_{VM} and TD_{DS} are total data in terabyte transferred from a machine and a data source respectively. The total execution time for running the MapReduce job is denoted as: ET, which is the total time of executing a task and transferring the data in and out from data centres.

$$TC = \sum_{i=1}^{n} C_{VM_i} * LP_{VM_i} + CT_{VM_i} * TD_{VM_i} + \sum_{j=1}^{m} CT_{DS_j} * TD_{DS_j} \quad (10)$$

As descried in Section III, the SLA requirements consist of Budget (B) and Deadline (D). As a result, algorithms' objective is given as:

$$Min\,(TC)\; Subject \quad to \quad TC < B \quad and \quad ET < D \quad (11)$$

IV. Algorithms

The provisioning and scheduling problem described in the last section is a multidimensional knapsack problem that was shown to be NP-complete. To tackle the problem, one may consider a greedy algorithm [8]. However, it cannot be directly adopted as it is not capable of satisfying the budget constraint. In addition, it is important to emphasize that the optimization algorithms are required to both determine what is the best set of Cloud resources to provision and also how to schedule tasks on those resources. Knowing the the characteristics of the problem, series of algorithms are presented and later in Section V their performances are compared. Algorithms are: List and First Fit sorted by Cost (LFFCost), Backtracking sorted by Cost (BTCost), Branch and Bound sorted by Cost (BBCost), Branch and Bound sorted by Cost and Performance (BBCostPerf), Branch and Bound with Multiple trees sorted by Cost (BBMultiCost), and finally Branch and Bound sorted by cost based on a Pruned Tree (BBPruned), which will be describe in detail.

The LFFCost algorithm [5] is a heuristic that is expected to have lower execution time. However, it can not handle budget and deadline constraints. The rest of the algorithms (BTCost, BBCost, BBCostPerf, BBMultiCost, and BBPruned) are tree-based which can cover all possible solutions. Prior to describing the algorithms we would like to describe two types of trees that are used by the aforementioned algorithms: Standard Tree and Pruned Tree.

A. Standard Trees

Figure 4 illustrates an example of a constructed Standard Tree for two tasks (Task#1 and Task#2) and two VM types (**L** for Large VM and **XL** for X.Large VM). The depth (levels) of the tree is the total number of map and reduce tasks (Task#1 and Task#2 in Figure 4), while the breadth (branches) is the total number of tasks multiplied by the number of VM instances. Therefore, in each level of the tree, each node is a VM type that can be selected for a task execution. The number of branches of all nodes are the same, which is the number of tasks (map and reduce tasks) multiplied by the number of VM instances. The reason why the Standard Tree is constructed in this way is that we need to cover all of the possibilities for scheduling MapReduce tasks. For example, in Figure 4 we have two tasks and two type of VMs. As a result, we have four branches under each node. This can be considered as a disadvantage in the Standard Tree as it grows exponentially as number of tasks and VM instances increases.

Standard Tree is sorted by cost ascending from left to right, so it will start consolidating all MapReduce tasks into the cheapest virtual machine, which is the most left leaf solution. If that solution does not satisfy the deadline constraint; it schedule one of the tasks to the next cheapest virtual machine. However, if the budget is violated the traversing process will stop with no solution found. An example of a solution set/vector is the the third leaf node in Figure 4 (v = $\{L_1, XL_1\}$), which means that the first task will be scheduled in a Large VM L_1, and the second task in an X.Large VM XL_1.

B. Pruned Tree

Figure 5 illustrates an example of a constructed Pruned tree for two tasks (Task#1 and Task#2) and two types of VM (**L** for Large VM and **XL** for X.Large VM). We managed to reduce the size of the tree compared to the Standard Tree. Similar to the Standard Tree, the depth (levels) of the tree is the total number of map and reduce tasks. However, the breadth in Pruned Tree is different than the Standard Tree. As shown in Figure 4 and 5, L2 is eliminated from children of the root node. The reason is that there is no difference in cost and execution time of scheduling the first task in L1 compared to L2. Similarly, XL2 (and generally for root children, all instances of each specific VM Type except one) is eliminated. The rest of the nodes are built from: 1) the set of nodes in the path from the root to the current node, and 2) an extra VM instance from each VM type, where the maximum number of VM instances to be added from each type is the number of MapReduce tasks. As illustrated in Figure 5, node branches are not of the same root branches, not like Standard Tree branches. The $path$ will be selected as an optimal solution once it does not violate the SLA constraints. In summary the solution space of Pruned Tree is considerably smaller in size compared to Standard tree, as out of VM instances with similar performance, we have only kept one and removed the others when it makes no difference in total cost and execution time.

Traversing the Pruned Tree is similar to the Slandered Tree, as nodes are sorted by cost ascending from left to right. Hence, it will start consolidating all MapReduce tasks into the cheapest VM (i.e the most left leaf solution). If that solution does not satisfy the deadline SLA objective; it moves one task to the next cheapest virtual machine, and so on. However, if the budget SLA objective is violated at any time the traversing will stop and return no solution is found. An example of a solution set/vector is the fifth leaf node in Figure 5 (v = $\{XL_1, XL_1\}$) - the first task and the second task will be scheduled in the same X.Large VM XL_1.

C. Algorithms for Provisioning and Scheduling

We propose several algorithms including a modified version of branch and bound algorithm and backtracking for the problem. As described in Section III, algorithms aim at satisfying SLA constraints (budget and deadline) while minimizing the cost. When algorithms employ the standard tree,it takes long time (days) to traverse the tree and find an optimal solution, especially for certain deadlines and low

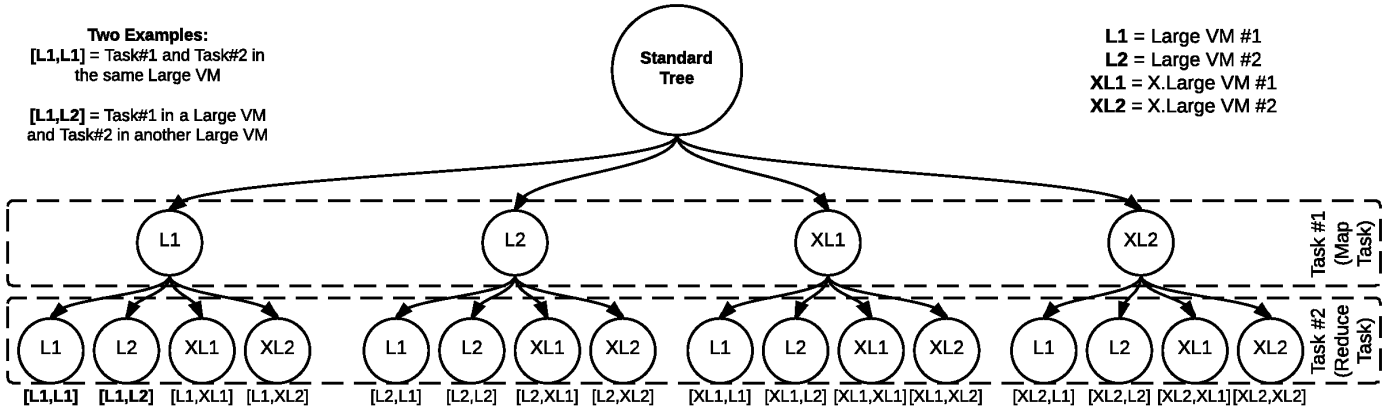

Fig. 4. An Example of a constructed Standard Tree with two tasks (Task#1 and Task#2) and two types of VMs (**L** for Large VM and **XL** for X.Large VM). BTCost, BBCost, BBCostPerf, and BBMultiCost algorithms use this type of tree

budget. Therefore, we have limited the maximum running time for algorithms to three minutes and chosen the best solution (even if it is not the optimal one). For all algorithms, solutions are stored in vectors (v). The vector index is a task and the vector value is a virtual machine.

In the branch and bound (BB) algorithms (BBCost, BBCostPerf, BBMultiCost, and BBPruned), we use two functions to estimate the execution time and cost: 1) $GetT(v)$ function: returns the execution time for a given solution vector. 2) $GetC(v)$ function: returns the cost for a given solution vector.

- **LFFCost** - an implementation of the List and First Fit (LFF) algorithm proposed by Hwang and Kim [5]. Virtual machines (VMs) in LFFCost are sorted by cost. It requires two steps for any algorithm to run MapReduce jobs on Cloud: resource provisioning and tasks scheduling. However, Hwang and Kim skipped the resource provisioning step and jumped into tasks scheduling. Therefore, we feed LFFCost with all possible combinations of VMs to compare it with our proposed algorithm.
- **BTCost** - a backtracking algorithm traversing the whole Standard Tree. It checks the execution time and cost on all leafs, and returns the solution with minimum cost that does not violate the deadline. BTCost is a Depth First Search (DFS) algorithm on the described Standard Tree.

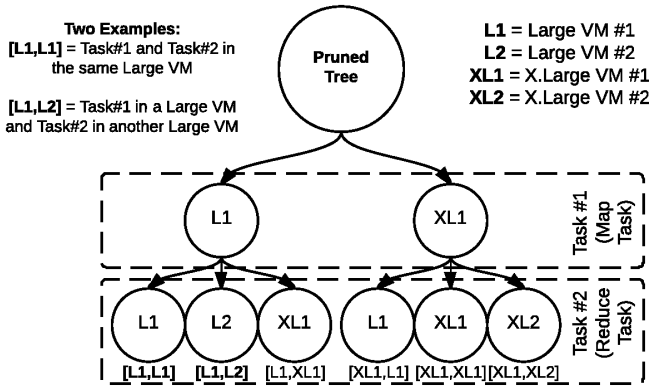

Fig. 5. An example of pruned tree used in BBPruned algorithm with two tasks (Task#1 and Task#2) and two types of VMs (**L** for Large VM and **XL** for X.Large VM)

Algorithm 1: Branch and bound algorithm on Standard Tree sorted by cost

input : Request (R) = $\{SLO, J\}$
output: Scheduling vector v

$v \leftarrow 1$;
$done \leftarrow false$;
while *!done* **do**
  $T \leftarrow$ GetT(v) ;
  $C \leftarrow$ GetC(v) ;
  if $T < D$ & $C < B$ **then**
    if *is in leaf* **then**
      return v ;
    else
      $v \leftarrow$ GoDeep(v) ;
    end
  else
    $v \leftarrow$ GoNextOrBack(v) ;
    if v *has no values* **then**
      $done \leftarrow true$;
    end
  end
end
return no solution is found ;
`GoDeep` (v)
  return $v + 1$;
end
`GoNextOrBack` (v)
  if *last element of the branch* **then**
    return v without the last value of the vector ;
  else
    return v with last value of the vector increased by one ;
  end
end

Algorithm 2: Branch and bound algorithm on a Pruned Tree sorted by cost

```
input : Request (R) = {SLO, J}
output: Scheduling vector v
foreach vm from each type of VM do
    vectors ← vectors+ Search({vm},null) ;
end
return Min(vectors); Search (currentBranch, path)
    foreach vm of currentBranch do
        newPath ← path + vm ;
        T ← GetT(newPath) ;
        C ← GetC(newPath) ;
        if T < D & C < B then
            if is in leaf then
                return newPath ;
            else
                nextBranch ← newPath ;
                Add to nextBranch one VM instance
                from each type ;
                return Search(nextBranch, newPath)
                ;
            end
        end
    end
end
```

- **BBCost** - uses branch and bound to improve the backtracking algorithm (BTCost) by using a validation function on each node (bounding step) to decide whether to go deep on the tree (branching step) or discard the rest of the tree (pruning step). This validation function calculates the execution time ($GetT(v)$) and cost ($GetC(v)$) and compares it with the defined SLA constraints. Algorithm 1 shows the pseudo code of the branch and bound algorithm that is used in BBCost, BBCostPerf, and BBMultiCost. BBCost uses the Standard Tree sorted by cost and initialize the vector v with the value of $v = \{1\}$, which is the cheapest VM. Then the algorithm keeps going deep in the tree unless the solution vector v violates the constraints. After the completion of every major branch (major branches are branches under the root), one of following occurs: 1) If there is a budget violation in any point during the search, the algorithm returns: a) "low budget" if no solution is founded, or b) the solution just before the violation. 2) If the budget is not violated, it goes to the next major branch until a solution which can meet the deadline is found.

- **BBCostPerf** - runs two trees at the same time, one sorted by cost (CostTree) and the other one by performance (PerfTree). CostTree uses BBCost algorithm to find optimal solution, while PerfTree uses an algorithm similar to BBCost but the tree is sorted by the performance of VMs. Our model in Section III shows the measurement of the VM performance is based on Million Instruction Per Second (MIPS). As soon as PerfTree finds a candidate solution, it will be sent to CostTree. CostTree will check its best solution up to this point with the one that has been received by PerfTree. If PerfTree's solution costs less and satisfies both of the budget and deadline objectives, then PerfTree's solution will be taken as the best solution. On the other hand, having another tree sorted by performance (PerfTree) helps us to find out whether there is a feasible solution at all or not (if deadline is violated before finding a solution) earlier than BTCost and BBCost.

- **BBMultiCost** - is an improved version of BBCostPerf. It splits the CostTree into multiple equal parts to search them in parallel along with PerfTree. The number of CostTree trees that runs in parallel depends on the capacity of the machine that runs BBMultiCost algorithm. In general, we based it on the number of cores considering one core for the main application/thread and another core for PerfTree tree. In our case, we leveraged the Hyper-Threaded technology on our machine, which allows us to run two trees in parallel on each core.

- **BBPruned** - described in Algorithm 2 and stands for **B**ranch and **B**ound algorithm for a **Pruned** Tree. It uses branch and bound algorithm on a Pruned Tree to find an optimal solution. BBPruned uses a recursive function ($Search$) to find a solution. The $Search$ function is called on each node except leaf nodes and its inputs are the current branch nodes ($currentBranch$) and the $path$ solution vector. The core functionality of this recursive function is to iterate on each node of $currentBranch$ and return $path$ as an optimal solution if it is a leaf node and it does not violate the constraints, or call $Search$ if it is not a leaf node and it does not violate the constraints. However, the node will simply be skipped if it violates any of constraints. To find a solution with the minimum cost, the Pruned Tree is sorted by cost ascending from left to right, so it will start consolidating all MapReduce tasks into the cheapest virtual machine, which is the most left leaf solution. If that solution does not satisfy the objectives, it evaluates the next cheapest solution.

V. Performance Evaluation

We have extended CloudSim [9] to build a model for a Cloud-hosted Big data analytics environment. For the batch layer, it is worth mentioning that the model is not compatible with Hadoop because Hadoop's scheduler is not designed for Cloud computing and it can not schedule tasks on heterogeneous VMs. VM types that are used in simulation[1] are similar to Amazon AWS VM types and OpenStack private Cloud VM flavours. We performed three sets of experiments as follows:

- **SLA Violation:** Table I listed the SLA violation percentage for the algorithms. The table shows that BTCost has 50% SLA violation for the scheduled MapReduce jobs, which is the highest reported percentage. BBCost and BBCostPerf come next with a

[1]The data, workloads, and algorithms implementations are available at: https://github.com/Cloudslab/CloudSimEx/tree/master/cloudsimex-mapreduce

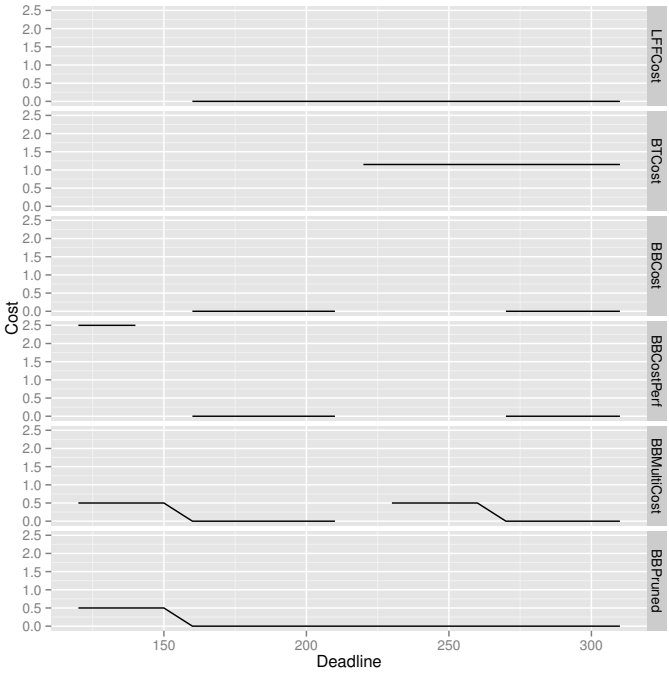

Fig. 6. The cost in dollar for provisioning and scheduling MapReduce jobs on cloud using different algorithms on different deadlines (solid line = SLA satisfied)

TABLE I. SLA VIOLATION PERCENTAGE FOR DIFFERENT ALGORITHMS

Algorithm	SLA Violation Percentage
LFFCost	20%
BTCost	50%
BBCost	45%
BBCostPerf	30%
BBMultiCost	5%
BBPruned	0%

slight decrease in SLA violation percentage. The next algorithm is LFFCost, which violates 20% of the SLA constraints. BBMultiCost shows a good performance with about 5% SLA violation. Finally, this experiment shows BBPruned algorithm causes no SLA violation, and manages to meet MapReduce jobs deadline within the defined budget. That is because BBPruned is capable of traversing the tree faster than the others, and therefore can find a feasible solution within the time limit (three minutes).

- **Cost Minimization:** Figure 6 shows the performance of algorithms and their total cost of provisioning and scheduling with different deadlines. In this experiment the budget is fixed to $2.5 and the deadline varies from 120 seconds to 310 seconds. When an algorithm fails to find a solution within the defined budget and/or deadline, no value for the cost has been reported. We have noticed that BBCost, BBCostPerf, and BBMultiCost manage to find solutions on loose deadlines or tight deadlines. This is because when a deadline is tight, the bounding step prunes most of the tree due to SLA violation, and as the solution space becomes smaller. As a result the algorithms manage to find a solution within the defined period of time. When we have loose deadlines the algorithms manages to find a feasible solution quicker due to the higher probability of spotting a solution at the early stage of search. Figure 6 shows that BBPruned outperforms the other algorithms and minimizes the cost without violating the SLA.

- **Algorithm Running Time:** Figure 7 shows the algorithm running time when deadline varies from 120 seconds to 310 seconds. BBPruned algorithm manages to find solutions in comparability short period of time similar to LFFCost. BTCost (the backtracking algorithm) takes considerably longer time to find a solution on tight deadlines. However, BBCost, BBCostPerf, and BBMultiCost (branch and bound algorithms) do not following a consistent trend. Figure 7 shows the variations for different deadlines- from less than one second to three minutes (which is the maximum allowed running time). We notice that when the deadline gets looser majority of the algorithms managed to find solution relatively faster. The reason behind this variations of running time for the cases of BBCost, BBCostPerf, and BBMultiCost (branch and bound algorithms) is that for some deadline, especially for tight deadlines, the bounding step prunes most of the tree and the algorithms manage to find a solution within the defined period of time. In addition, when we have loose deadlines there is a higher probability of spotting a solution at the early stage of the search. As shown in Figure 7, BBCostPerf algorithm runs faster than BTCost and BBCost because it stops traversing the tree in early stages when PerfTree finds a solution with less cost earlier than CostTree.

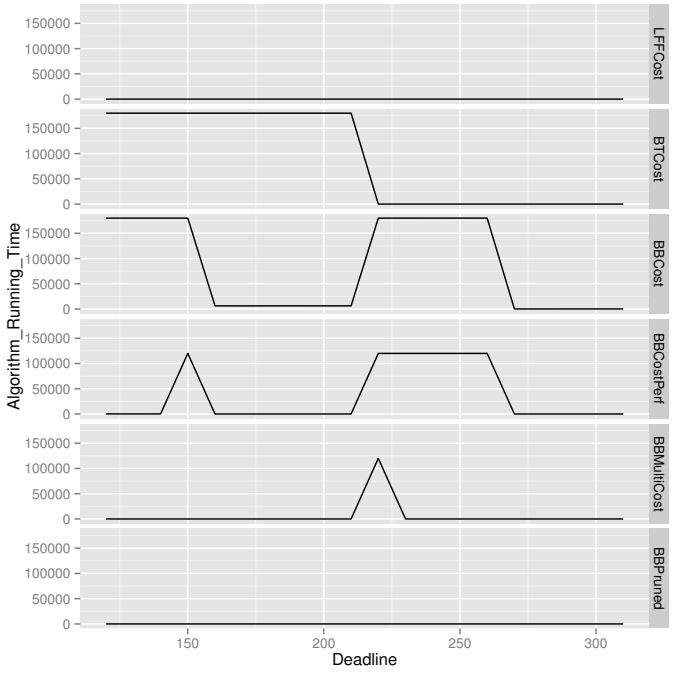

Fig. 7. The running time for the six algorithms in seconds

VI. Related Work

Most existing MapReduce schedulers and frameworks do not consider budget as a constraint [4], [10], [6], [5], [7], [11], although they help in efficiently running a MapReduce job in Cloud environments. In addition, the majority of the implementations are Hadoop-based and use Hadoop default scheduling algorithms to schedule jobs on Clouds [4], [10], [6], [11]. Nevertheless, Hadoop scheduling algorithms were designed for clusters of homogeneous machines, which is not applicable for Cloud heterogeneous resources. However, the proposed model in this paper is designed for provisioning and scheduling MapReduce applications across instances of multiple clouds.

In addition, majority of prior works [4], [10], [6], [5], [7], [11] did not consider the time it takes to upload data to Cloud when modelling the scheduling and provisioning problems. Conversely, we consider the time it takes to transfer the input data to the mappers and intermediate data to the reducers. The transfer time is computed based on the size of data and network throughput.

Lama et al. [4] developed a Hadoop auto-reconfiguration and Cloud resource provisioning system called AROMA. It has two main operations modes: offline and online. The offline mode uses a machine learning techniques to cluster Hadoop jobs to feed the online operations. The online mode operations uses optimization to allocate resources and configure Hadoop. They used Hadoop's default scheduler to run MapReduce jobs on Cloud, which is designed to schedule tasks on Cluster. Hwang et al. [5] proposed a deadline-constrained MapReduce scheduler on Clouds. Their scheduling algorithms skip the resource selection and assume that the user will provide those resources (VMs) to the scheduler. Lee et al. [6] built a system called TARA (Topology-Aware Resource Allocation). TARA does not consider any constraints on budget and deadline. It aims at minimizing the execution time of running MapReduce jobs on Clouds. TARA does not play any role in allocating and provisioning resources on Clouds. Instead, it manages the virtual machine placement. In other words, TARAs goal is limited to VM placement by mapping them to hosts.

We have identified three different MapReduce Cloud infrastructure types in the literature: 1) *MapReduce Cluster on Cloud:* This infrastructure uses virtualization technology to pre-provision a static set of homogeneous virtual machines (VMs), and run MapReduce jobs on all of them. The major contribution in this type of infrastructure is limited to task scheduling, ignoring resource selection and provisioning [6], [7], [11]. 2) *MapReduce Grid on Cloud:* Likewise, in this category virtualization technology is used to pre-provision resources. However, in this category they are shared between users, and they are not destroyed and re-provisioned for each user. The focus here is on fair resource distribution [4], [5]. 3) *MapReduce on Cloud:* The use of a Cloud Infrastructure as a Service (IaaS) to dynamically provision resources is the focus of this category. The contribution here is designing efficient resource provisioning and task scheduling [7].

VII. Conclusions and Future Work

In this paper we discussed the problem of provisioning and scheduling heterogeneous Cloud resources for Big Data analytics and narrowed our focused to Batch layer. We presented an efficient architecture and algorithm (BBPruned) to provision resources and schedule MapReduce tasks in Batch layer for users with SLA constraints (budget and deadline). The proposed algorithm is a modified version of branch and bound algorithm that traverses a customized Pruned Tree, which is smaller than a Standard Tree. The designed algorithm splits the tree into almost equal branches and traverses them in parallel to utilize multi-core systems. Finally, we have shown the efficiency of the proposed algorithm and its ability to generate more prominent solutions faster without violating the SLA constraints. This allows to build batch views in timely manner for Serving layer and bring us one step closer to build a SLA-aware Big Data analytics solution in Cloud.

For future work, we develop algorithms for the other layers and for private clouds with different classes of users to decrease SLA violation for users with higher priorities. In addition, we will investigate a hybrid Cloud scheduling and provisioning strategy that decreases the intermediate data transfer between different datacentres and clouds. This helps to satisfy tighter deadlines and reduce the cost of data transfer.

References

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," *NIST special publication*, vol. 800, p. 145, 2011.

[2] N. Marz and J. Warren, *Big Data Principles and best practices of scalable realtime data systems*. Manning Publications; First edition.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation (OSDI) 2004*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004.

[4] P. Lama and X. Zhou, "AROMA: Automated Resource Allocation and Configuration of MapReduce Environment in the Cloud," in *Proceedings of the 9th International Conference on Autonomic Computing (ICAC '12)*. ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2371536.2371547

[5] E. Hwang and K. H. Kim, "Minimizing Cost of Virtual Machines for Deadline-Constrained MapReduce Applications in the Cloud," in *Proceedings of ACM/IEEE 13th International Conference on Grid Computing (GRID'12)*, 2012.

[6] G. Lee, N. Tolia, P. Ranganathan, and R. H. Katz, "Topology-Aware Resource Allocation for Data-Intensive Workloads," in *Proceedings of the First ACM Asia-Pacific Workshop on Systems (APSys'10)*, 2010. [Online]. Available: http://doi.acm.org.ezp.lib.unimelb.edu.au/10.1145/1851276.1851278

[7] M. Mattess, R. N. Calheiros, and R. Buyya, "Scaling mapreduce applications across hybrid clouds to meet soft deadlines," in *Proceedings of the 27th IEEE International Conference on Advanced Information Networking and Applications (AINA'13)*, 2013.

[8] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of operations research*, vol. 4, no. 3, pp. 233–235, 1979.

[9] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.

[10] K. Kambatla, A. Pathak, and H. Pucha, "Towards optimizing hadoop provisioning in the cloud," in *Proceedings of the First Workshop on Hot Topics in Cloud Computing (HotCloud '09)*, 2009.

[11] A. Verma, L. Cherkasova, and R. Campbell, "Aria: automatic resource inference and allocation for mapreduce environments," in *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC'11)*, 2011.