Budget Driven Scheduling MapReduce Jobs in the Heterogeneous Cloud

Article *in* IEEE Transactions on Cloud Computing · January 2014

CITATIONS
0

READS
156

2 authors, including:

Wei Shi
Carleton University
94 PUBLICATIONS **358** CITATIONS

SEE PROFILE

Budget-Driven Scheduling Algorithms for Batches of MapReduce Jobs in Heterogeneous Clouds

Yang Wang and Wei Shi *IEEE Member*

Abstract—In this paper, we consider task-level scheduling algorithms with respect to budget and deadline constraints for a batch of MapReduce jobs on a set of provisioned heterogeneous (virtual) machines in cloud platforms. The heterogeneity is manifested in the popular "pay-as-you-go" charging model where the service machines with different performance would have different service rates. We organize the batch of jobs as a κ-stage workflow and study two related optimization problems, depending on whether the constraints are on monetary budget or on scheduling length of the workflow. First, given a total monetary budget B, by combining an *in-stage* local greedy algorithm (whose optimality is also proven) and dynamic programming (DP) techniques, we propose a global optimal scheduling algorithm to achieve minimum scheduling length of the workflow within $O(\kappa B^2)$. Although the optimal algorithm is efficient when B is polynomially bounded by the number of tasks in the MapReduce jobs, the quadratic time complexity is still high. To improve the efficiency, we further develop two greedy algorithms, called *Global Greedy Budget* (GGB) and *Gradual Refinement* (GR), each adopting different greedy strategies. In GGB we extend the idea of the local greedy algorithm to the efficient global distribution of the budget with minimum scheduling length as a goal whilst in GR we iteratively apply the DP algorithm to the distribution of exponentially reduced budget so that the solutions are gradually refined. Second, we consider the optimization problem of minimizing cost when the (time) deadline of the computation D is fixed. We convert this problem into the standard *Multiple-Choice Knapsack Problem* via a parallel transformation. Our empirical studies verify the proposed optimal algorithms and show the efficiencies of the greedy algorithms in cost-effectiveness to distribute the budget for performance optimizations of the MapReduce workflows.

Index Terms—MapReduce scheduling, cost and time constraints, optimal greedy algorithm, optimal parallel scheduling algorithm, dynamic programming, Cloud computing

1 Introduction

Due to their abundant on-demand computing resources and elastic billing models, clouds have emerged as a promising platform to address various data processing and task computing problems [1]–[4]. MapReduce [5], characterized by its remarkable simplicity, fault tolerance, and scalability, is becoming a popular programming framework to automatically parallelize large scale data processing as in web indexing, data mining [6], and bioinformatics [7]. MapReduce is extremely powerful and runs fast for various application areas.

Since a cloud supports on-demand "massively parallel" applications with loosely coupled computational tasks, it is amenable to the MapReduce framework and thus suitable for the MapReduce applications in different areas. Therefore, many cloud infrastructure providers (CIPs) have deployed the MapReduce framework on their commercial clouds as one of their infrastructure services (e.g., Amazon Elastic MapReduce (Amazon EMR) [8]). Often, some cloud service providers (CSPs) also offer their own *MapReduce as a Service* (MRaaS) which is typically set up as a kind of Software as a Service (SaaS) on the owned or provisioned MapReduce clusters of cloud instances (e.g., Microsofts Apache Hadoop on Windows Azure Services [9], Hadoop on Google Cloud Platform [10] and Teradata Aster Discovery Platform [11]). Traditionally, these cloud instances are composed of a homogeneous set of commodity hardware multiplexed by virtualization technology. However, with the advance of computing technologies and the ever-growth of diverse requirements of end-users, a heterogeneous set of resources that take advantages of different network accelerators, machine architectures, and storage hierarchies allow clouds to be more beneficial to the deployments of the MapReduce framework for various applications [12], [13].

Clearly, for CSPs to reap the benefits of such a deployment, many challenging problems have to be addressed. However, most current studies focus solely on system issues pertaining to deployment, such as overcoming the limitations of the cloud infrastructure to build-up the framework [14], [15], evaluating the performance harm from running the framework on virtual machines [16], and other issues in fault tolerance [17], reliability [18], data locality [19], etc. We are also aware of some recent research tackling the scheduling problem of MapReduce as well as the heterogeneity in clouds [12], [20]–[25]. Some contributions mainly address the scheduling is-

- *Y. Wang is with the Faculty of Computer Science, University of New Brunswick, Fredericton, Canada, E3B 5A3. E-mail: {ywang8@unb.ca}.*
- *W. Shi is with the Faculty of Business and I.T., University of Ontario Institute of Technology, Ontario, Canada, H1L 7K4. E-mail: {wei.shi@uoit.ca}.*

sues with various concerns placed on dynamic loading [21], energy reduction [23], task-slot assignment [26], and network performance [24] while others optimize the MapReduce framework for heterogeneous Hadoop clusters with respect to data placements [25], resource utilization [27], and performance modelling [28].

To the best of our knowledge, prior work squarely on optimizing the scheduling of a batch of MapReduce jobs with budget constraints at task level in heterogeneous clusters is quite few [29]. In our opinion two major factors that may account for this status quo. First, as mentioned above, the MapReduce service, like other basic database and system services, could be provided as an infrastructure service by CIPs (e.g., Amazon), rather than CSPs. Consequently, it would be charged together with other infrastructure services. Second, some properties of the MapReduce framework (e.g., automatic fault tolerance with speculative execution [12]) make it difficult for CSPs to track job execution in a reasonable way, thus making scheduling very complex.

Since cloud resources are typically provisioned on demand with a "pay-as-you-go" billing model, cloud-based applications are usually budget driven. Consequently, in practice, the effective use of resources to satisfy relevant performance requirements within budget is always a pragmatic concern for CSPs.

In this paper, we investigate the problem of scheduling a batch of MapReduce jobs as a workflow within budget and deadline constraints. This workflow could be an iterative MapReduce job, a set of independent MapReduce jobs, or a collection of jobs related to some high-level applications such as Hadoop Hive [30]. We address *task-level scheduling*, which is fine grained compared to the frequently-discussed job-level scheduling, where the scheduled unit is a job instead of a task. More specifically, we focus on the following two optimization problems (whose solutions are of particular interest to CSPs intending to deploy MRaaS on heterogeneous cloud instances in a cost-effective way):

1) Given a fixed budget B, how to efficiently select the machine from a candidate set for each task so that the total scheduling length of the workflow is minimum without breaking the budget;
2) Given a fixed deadline D, how to efficiently select the machine from a candidate set for each task so that the total monetary cost of the workflow is minimum without missing the deadline;

At first sight, both problems appear to be mirror cases of one another: solving one may be sufficient to solve the other. However, we will show that there are still some asymmetries in their solutions. In this paper, we focus mainly on the first problem, and then briefly discuss the second. To solve the fixed-budget problem, we first design an efficient *in-stage* greedy algorithm for computing the minimum execution time with a given budget for each stage. Based on the structure of this problem and the adopted greedy approach, we then prove the optimality of this algorithm with respect to execution time and budget use. With these results, we develop a dynamic programming algorithm to achieve a global optimal solution with scheduling time of $O(\kappa B^2)$. Although the optimal algorithm is efficient when B is polynomially bounded by the number of tasks in the workflow, the quadratic time complexity is still high. To improve the efficiency, we further develop two greedy algorithms, called *Global Greedy Budget* (GGB) and *Gradual Refinement* (GR), each having different greedy strategies. Specifically, in GGB we extend the idea of the in-stage greedy algorithm to the efficient global distribution of the budget with minimum scheduling length as a goal whilst in GR we iteratively run a DP-based algorithm to distribute exponentially reduced budget in the workflow so that the final scheduling length could be gradually refined. Our evaluations reveal that both the GGB and GR algorithms, each exhibiting a distinct advantage over the other, are very close to the optimal algorithm in terms of scheduling lengths but entail much lower time overhead.

In contrast, a solution to the second problem is relatively straightforward as we can reduce it into the standard *multiple-choice knapsack* (MCKS) problem [31], [32] via a parallel transformation. Our results show that the two problems can be efficiently solved if the total budget B and the deadline D are polynomially bounded by the number of tasks and the number of stages, respectively in the workflow, which is usually the case in practice. Our solutions to these problems facilitate the deployment of the MapReduce framework as a MRaaS for CSPs to match diverse user requirements (again with budget or deadline constraints) in reality.

The rest of this paper is organized as follows: in Section 2, we introduce some background knowledge regarding the MapReduce framework and survey some related work. Section 3 presents our problem formulation. The proposed budget-driven and time-constrained algorithms are discussed in Section 4. We follow with the the results of our empirical studies in Section 5 and conclude the paper in Section 6.

2 Background and Related Work

The MapReduce framework was first advocated by Google in 2004 as a programming model for its internal massive data processing [33]. Since then it has been widely discussed and accepted as the most popular paradigm for data intensive processing in different contexts. Therefore there are many implementations of this framework in both industry and academia (such as Hadoop [34], Dryad [35], Greenplum [36]), each with its own strengths and weaknesses.

Since Hadoop MapReduce is the most popular open source implementation, it has become the *de facto* research prototype on which many studies are conducted. We thus use the terminology of the Hadoop community in the rest of this paper, and focus here mostly on related work built using the Hadoop implementation.

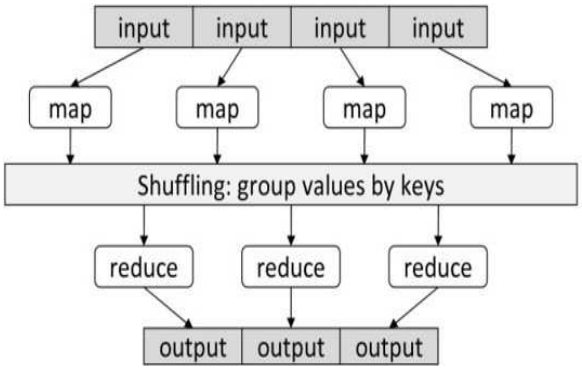

Fig. 1: MapReduce framework.

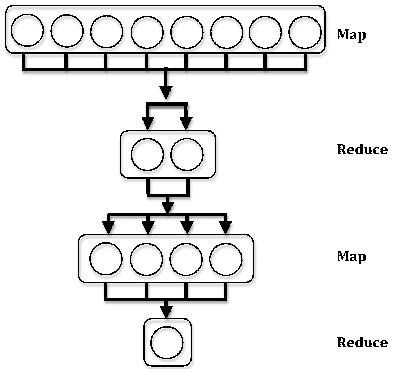

Fig. 2: A 4-stage MapReduce workflow.

From an abstract viewpoint, a MapReduce job essentially consists of two sets of tasks: map tasks and reduce tasks, as shown in Fig. 1. The executions of both sets of tasks are synchronized into a map stage followed by a reduce stage. In the map stage, the entire dataset is partitioned into several smaller chunks in forms of key-value pairs, each chunk being assigned to a map node for partial computation results. The map stage ends up with a set of intermediate key-value pairs on each map node, which are further shuffled based on the intermediate keys into a set of scheduled reduce nodes where the received pairs are aggregated to obtain the final results. For an iterative MapReduce job, the final results could be tentative and further partitioned into a new set of map nodes for the next round of the computation. A batch of MapReduce jobs may have multiple stages of MapReduce computation, each stage running either map or reduce tasks in parallel, with enforced synchronization only between them. Therefore, the executions of the jobs can be viewed as a fork&join workflow characterized by multiple synchronized stages, each consisting of a collection of sequential or parallel map/reduce tasks.

An example of such a workflow is shown in Fig. 2 which is composed of 4 stages, respectively with 8, 2, 4 and 1 (map or reduce) tasks. These tasks are to be scheduled on different nodes for parallel execution. However, in heterogeneous clouds, different nodes may have different performance and/or configuration specifications, and thus may have different service rates. Therefore, because resources are provisioned on-demand in cloud computing, the CSPs are faced with a general practical problem: how are resources to be selected and utilized for each running task in a cost-effective way? This problem is, in particular, directly relevant to CSPs wanting to compute their MapReduce workloads, especially when the computation budget is fixed.

Hadoop MapReduce is made up of an execution runtime and a distributed file system. The execution runtime is responsible for job scheduling and execution. It is composed of one master node called *JobTracker* and multiple slave nodes called *TaskTrackers*. The distributed file system, referred to as *HDFS*, is used to manage task and data across nodes. When the JobTracker receives a submitted job, it first splits the job into a number of map and reduce tasks and then allocates them to the TaskTrackers, as described earlier. As with most distributed systems, the performance of the task scheduler greatly affects the scheduling length of the job, as well as, in our particular case, the budget consumed.

Hadoop MapReduce provides a FIFO-based default scheduler at job level, while at task level, it offers developers a TaskScheduler interface to design their own schedulers. By default, each job will use the whole cluster and execute in order of submission. In order to overcome this inadequate strategy and share fairly the cluster among jobs and users over time, Facebook and Yahoo! leveraged the interface to implement Fair Scheduler [37] and Capacity Scheduler [38], respectively.

Beyond fairness, there exists additional research on the scheduler of Hadoop MapReduce aiming at improving its scheduling policies. For instance, Hadoop adopts *speculative task scheduling* to minimize the slowdown in the synchronization phases caused by straggling tasks in a homogeneous environment [34].

To extend this idea to heterogeneous clusters, Zaharia et al. [12] proposed the LATE algorithm. But this algorithm does not consider the phenomenon of dynamic loading, which is common in practice. This limitation was studied by You et al. [21] who proposed a load-aware scheduler. Zaharia's work on a *delay scheduling* mechanism [19] to improve data locality with relaxed fairness is another example of research on Hadoop's scheduling. There is also, for example, work on *power-aware scheduling* [39], *deadline constraint scheduling* [40], and scheduling based on automatic task slot assignments [41]. While these contributions do address different aspects of MapReduce scheduling, they are mostly centred on system performance and do not consider budget, which is our main focus.

Budget constraints have been considered in studies focusing on scientific workflow scheduling on HPC platforms including the Grid and Cloud [42]–[44]. For example, Yu et al. [42] discussed this problem based on service Grids and presented a QoS-based workflow scheduling method to minimize execution cost and yet meet the time constraints imposed by the user. In the

same vein, Zeng et al. [43] considered the executions of large scale many-task workflows in clouds with budget constraints. They proposed *ScaleStar*, a budget-conscious scheduling algorithm to effectively balance execution time with the monetary costs. Now recall that, in the context of this paper, we view the executions of the jobs as a fork&join workflow characterized by multiple synchronized stages, each consisting of a collection of sequential or parallel map/reduce tasks. From this perspective, the abstracted fork&join workflow can be viewed as a special case of general workflows. However, our focus is on MapReduce scheduling with budget and deadline constraints, rather than on general workflow scheduling. Therefore, the characteristics of MapReduce framework are fully exploited in the designs of the scheduling algorithms.

A more recent work that is highly related to ours is the *Dynamic Priority Task Scheduling* algorithm (DPSS) for heterogeneous Hadoop clusters [29]. Although this algorithm also targets at task-level scheduling with budget optimization as a goal, it is different from ours in two major aspects. First, DPSS is designed to allow capacity distribution across concurrent users to change dynamically based on user preferences. In contrast, our algorithm assume sufficient capacities, each with different prices, for task scheduling and the goal is to minimize the scheduling length (budget) within the given budget (deadline). Second, DPSS optimizes the budget on per-job basis by allowing users to adjust their spending over time whereas our algorithms optimize the scheduling of a batch of jobs as a whole. Therefore, our algorithms and DPSS can complement to each other for different requirements.

3 Problem Formulation

We model a batch of MapReduce job as a multi-stage fork&join workflow that consists of κ stages (called a κ-stage job), each stage j having a collection of independent map or reduce tasks, denoted as $J_j = \{J_{j0}, J_{j1}, ..., J_{jn_j}\}$, where $0 \leq j < \kappa$, and $n_j + 1$ is the size of stage j. In a cloud, each map or reduce task may be associated with a set of machines provided by cloud infrastructure providers to run this task, each with possibly distinct performance and configuration and thus having different charge rates. More specifically, for Task J_{jl}, $0 \leq j < \kappa$ and $0 \leq l \leq n_j$ the available machines and corresponding prices (service rates) are listed in Table 1 where $t^u_{jl}, 1 \leq u \leq m_{jl}$ represents the time to

TABLE 1: Time-price table of task J_{jl}

$$\begin{bmatrix} t^1_{jl} & t^2_{jl} & ... & t^{m_{jl}}_{jl} \\ p^1_{jl} & p^2_{jl} & ... & p^{m_{jl}}_{jl} \end{bmatrix}$$

run task J_{jl} on machine M_u whereas p^u_{jl} represents the corresponding price for using that machine, and m_{jl} is the total number of the machines that can run J_{jl}, the values of these variables could be determined by the VM power and the computational loads of each task. Here, for the sake of simplicity, we assume that there are sufficient resources for each task in the Cloud, which implies that a machine is never competed by more than one tasks, and its allocation is charged based on the actual time it is used based on a fixed service rate. Although there some studies on *dynamic pricing* to maximize revenue [45], *static pricing* is still the dominant strategy today. Therefore, we believe this model is accurate to reflect the on-demand provisioning and billing of the cloud resources.

Without loss of generality, we further assume that times have been sorted in increasing order and prices in decreasing order, and furthermore, that both time and price values are unique in their respective sorted sequence. These assumptions are reasonable since given any two machines with same run time for a task, the expensive one should never be selected. Similarly, given any two machines with same price for a task, the slow machine should never be chosen.

Note that we do not model the communication cost inherent to these problems since, in our particular case, communication between the map/reduce tasks is manipulated by the MapReduce framework via the underlying network file systems, and transparent to the scheduler.[1] For clarity and quick reference, we provide in Table 2 a summary of some symbols frequently used hereafter.

3.1 Budget Constraints

Given budget B_{jl} for task J_{jl}, the shortest time to finish it, denoted as $T_{jl}(B_{jl})$ is defined as

$$T_{jl}(B_{jl}) = t^u_{jl} \qquad p^{u+1}_{jl} < B_{jl} < p^{u-1}_{jl} \tag{1}$$

Obviously, if $B_{jl} < p^{m_{jl}}_{jl}$, $T_{jl}(B_{jl}) = +\infty$.

The time to complete a stage j with budget B_j, denoted as $T_j(B_j)$, is defined as the time consumed when the last task in that stage completes within the given budget:

$$T_j(B_j) = \max_{\sum_{l\in[0,n_j]} B_{jl} \leq B_j} \{T_{jl}(B_{jl})\} \tag{2}$$

For fork&join, one stage cannot start until its immediately preceding stage has terminated. Thus the total makespan within budget B to complete the workflow is defined as the sum of all stages' times. Our goal is to minimize this time within the given budget B.

$$T(B) = \min_{\sum_{j\in[0,\kappa)} B_j \leq B} \sum_{j\in[0,\kappa)} T_j(B_j) \tag{3}$$

3.2 Deadline Constraints

Given deadline D_j for stage j, the minimum cost to finish stage j is

$$C_j(D_j) = \sum_{l\in[0,n_j]} C_{jl}(D_j) \tag{4}$$

1. In practice the tasks in workflow computations usually communicate with each other via the file systems in the Cloud.

TABLE 2: Notation frequently used in model and algorithm descriptions

Symbol	*Meaning*
κ	the number of stages
J_{ji}	the ith task in stage j
J_j	task set in stage j
n_j	the number of tasks in stage j
n	the total number of tasks in the workflow
t^u_{jl}	the time to run task J_{jl} on machine M_u
p^u_{jl}	the cost rate for using M_u
m_{jl}	the total number of the machines that can run J_{jl}
m	the total size of time-price tables of the workflow
B_{jl}	the budget used by J_{jl}
B	the total budget for the MapReduce job
$T_{jl}(B_{jl})$	the shortest time to finish J_{jl} given B_{jl}
$T_j(B_j)$	the shortest time to finish stage j given B_j
$T(B)$	the shortest time to finish job given B
D_j	the deadline to stage j
$C_{jl}(D_j)$	the minimum cost of J_{jl} in stage j within D_j
$C(D)$	the minimum cost to finish job within D

where $C_{jl}(D_j)$ is the minimum cost to finish J_{jl} in stage j within D_j. Note that we require $t^1_{jl} \leq D_j \leq t^{m_{jl}}_{jl}$. Otherwise $C_{jl}(D_j) = +\infty$. Finally, our optimization problem can be written as

$$C(D) = \min_{\sum_{j\in[1,\kappa]} D_j \leq D} \sum_{j\in[0,\kappa)} C_j(D_j) \tag{5}$$

Some readers may question the feasibility of this model since the number of stages and the number of tasks in each stage need to be known a prior to the scheduler. But, in reality, it is entirely possible since a) the number of map tasks for a given job is driven by the number of input splits (which is known to the scheduler) and b) the number of reduce tasks can be preset as with all other parameters (e.g., parameter mapred.reduce.tasks in Hadoop). As for the number of stages, it is not always possible to predefine it for MapReduce workflows. This is the main limitation of our model. But under the default FIFO job scheduler, we can treat a set of independent jobs as a single fork&join workflow. Therefore, we believe our model is still representative of most cases in reality.

4 Budget-Driven Algorithms

In this section, we propose our task-level scheduling algorithms for MapReduce workflows with the goals of optimizing Equations (3) and (5) under respective budget and deadline constraints. We first consider the optimization problem under budget constraint where an in-stage local greedy algorithm is designed and combined with dynamic programming techniques to obtain an optimal global solution. To overcome the inherent complexity of the optimal solution, we also present two efficient greedy algorithms, called Global-Greedy-Budget algorithm (GGB) and Gradual-Refinement algorithm (GR). With these results, we then briefly discuss the second optimization problem with respect to the deadline constraints.

Algorithm 1 In-stage greedy distribution algorithm

1: **procedure** $T_j(n_j, B_j)$ ▷ Dist. B_j among J_j
2: $B'_j = B_j - \sum_{l\in[0,n_j]} p^{m_{jl}}_{jl}$
3: **if** $B'_j < 0$ **then return** $(+\infty)$
4: **end if**
5: ▷ Initialization
6: **for** $J_{jl} \in J_j$ **do** ▷ $O(n_j)$
7: $T_{jl} \leftarrow t^{m_{jl}}_{jl}$ ▷ record exec. time
8: $B_{jl} \leftarrow p^{m_{jl}}_{jl}$ ▷ record budget dist.
9: $M_{jl} \leftarrow m_{jl}$ ▷ record assigned machine.
10: **end for**
11: **while** $B'_j \geq 0$ **do** ▷ $O(\frac{B_j \log n_j}{\min_{0\leq l\leq n_j}\{\delta_{jl}\}})$
12: $jl^* \leftarrow \arg\max_{l\in[0,n_j]}\{T_{jl}\}$ ▷ get the slowest task
13: $u \leftarrow M_{jl^*}$
14: **if** $u = 1$ **then**
15: **return** (T_{jl^*})
16: **end if**
17: ▷ Lookup matrix in Table 1
18: $< p^{u-1}_{jl^*}, p^u_{jl^*} > \leftarrow$ **Lookup**$(J_{jl^*}, u-1, u)$
19: $\delta_{jl^*} \leftarrow p^{u-1}_{jl^*} - p^u_{jl^*}$
20: **if** $B'_j \geq \delta_{jl^*}$ **then** ▷ reduce J_{jl^*}'s time
21: $B'_j \leftarrow B'_j - \delta_{jl^*}$
22: ▷ Update
23: $B_{jl^*} \leftarrow B_{jl^*} + \delta_{jl^*}$
24: $T_{jl^*} \leftarrow t^{u-1}_{jl}$
25: $M_{jl^*} \leftarrow u - 1$
26: **else**
27: **return** (T_{jl^*})
28: **end if**
29: **end while**
30: **end procedure**

4.1 Optimization under Budget Constraints

The proposed algorithm should be able of distributing the budget among the stages, and in each stage distributing the assigned budget to each constituent task in an optimal way. To this end, we design the algorithm in two steps:

1) Given budget B_j for stage j, distribute the budget to all constituent tasks in such a way that $T_j(B_j)$ is minimum (see Equation (2)). Clearly, the computation for each stage is independent of other stages. Therefore such computations can be treated in parallel using κ machines.
2) Given budget B for a workflow and the results in Equation (2), optimize our goal of Equation (3).

4.1.1 *In-Stage Distribution*

To address the first step, we develop an optimal in-stage greedy algorithm to distribute budget B_j between the $n_j + 1$ tasks in such a way that $T_j(B_j)$ is minimized. Based on the structure of this problem, we then prove the optimality of this local algorithm.

The idea of the algorithm is simple. To ensure that all the tasks in stage j have sufficient budget to finish while minimizing $T_j(B_j)$, we first require $B'_j = B_j - \sum_{l\in[0,n_j]} p_{jl}^{m_{jl}} \geq 0$ and then iteratively distribute B'_j in a greedy manner each time to a task whose current execution time determines $T_j(B_j)$ (i.e., the slowest one). This process continues until no sufficient budget is left. Algorithm 1 shows the pseudo code of this algorithm.

In this algorithm, we use three *profile variables* T_{jl}, B_{jl} and M_{jl} for each task J_{jl} to record respectively its execution time, assigned budget, and the selected machine (Lines 6-10). After setting these variables with their initial values, the algorithm enters into its main loop to iteratively update the profile variables associated with the current slowest task (i.e., J_{jl^*}) (Lines 11-30). By searching the time-price table of J_{jl^*} (i.e., Table 1), the *Lookup function* can obtain the costs of the machines indexed by its second and third arguments. Each time, the next faster machine (i.e., $u-1$) is selected when more δ_{jl^*} is paid. The final distribution information is updated in the profile variables (Lines 18-28).

Theorem 4.1: Given budget B_j for stage j having n_j tasks, Algorithm 1 yields the optimal solution to the distribution of the budget B_j to all the n_j tasks in that stage within time $O(\frac{B_j \log n_j}{\min_{0\leq l\leq n_j}\{\delta_{jl}\}} + n_j)$.

Proof: Given budget B_j allocated to stage j, by following the greedy algorithm, we can obtain a solution $\Delta_j = \{b_0, b_1, ..., b_{n_j}\}$ where b_l is the budget allocated to task $J_{jl}, 0 \leq l \leq n_j$. Based on this sequence, we can further compute the corresponding finish time sequence of the tasks as $t_0, t_1, ..., t_{n_j}$. Clearly, there exists $k \in [0, n_j]$ that determines the stage completion time to be $T_j(B_j) = t_k$.

Suppose $\Delta_j^* = \{b_0^*, b_1^*, ..., b_{n_j}^*\}$ is an optimal solution and its corresponding finish time sequence is $t_0^*, t_1^*, ..., t_{n_j}^*$. Given budget B_j, there exists $k' \in [0, n_j]$ satisfying $T_j^*(B_j) = t_{k'}^*$. Obviously, $t_k \geq t_{k'}^*$. In the following we will show that, necessarily, $t_k = t_{k'}^*$.

To this end, we consider two cases:

1) If for $\forall l \in [0, n_j]$, $t_l^* \leq t_l$, then we have $b_l^* \geq b_l$. This is impossible because given $b_l^* \geq b_l$ for $\forall l \in [0, n_j]$, the greedy algorithm would have sufficient budget $\geq b_k^* - b_k$ to further reduce t_k of $task_k$, which is contradictory to $T_j(B_j) = t_k$, unless $b_k^* = b_k$, but in this case, $T_j^*(B_j)$ will be t_k, rather than t_k^*. Thus, case 1 is indeed impossible.
2) Given the result in 1), there must exist $i \in [0, n_j]$ that satisfies $t_i < t_i^*$. This indicates that in the process of the greedy choice, $task_i$ is allocated budget to reduce the execution time at least from t_i^* to t_i and this happens no later than when $T_j(B_j) = t_k$. Therefore, we have $t_i^* \geq t_k \geq t_k^* \geq t_i^*$, then $t_k = t_{k'}^*$.

Overall, $t_k = t_{k'}^*$, that is, the algorithm making the greedy choice at every step does produce an optimal solution.

The (scheduling) time complexity of this algorithm is straightforward. It consists of the overhead in initialization (Lines 6-10) and the main loop to update the profile variables (Lines 11-30). Since the size of J_j is n_j, the initialization overhead is $O(n_j)$. If we adopt some advanced data structure to organize $T_{jl}, 0 \leq l \leq n_j$ for efficient identification of the slowest task, l^* can be obtained within $O(\log n_j)$ (Line 12). On the other hand, there are at most $O(\frac{B_j}{\min_{0\leq l\leq n_j}\{\delta_{jl}\}})$ iterations (Line 11). Overall, we have the time complexity of $O(\frac{B_j \log n_j}{\min_{0\leq l\leq n_j}\{\delta_{jl}\}} + n_j)$. □

Since all the κ stages can be computed in parallel, the total time complexity for the parallel pre-computation is $O(\max_{j\in[0,\kappa)}\{\frac{B_j \log n_j}{\min_{0\leq l\leq n_j}\{\delta_{jl}\}} + n_j\})$.

Given Theorem 4.1, we immediately obtain the following corollary, which is a direct result of the first case in the proof of Theorem 4.1.

Corollary 4.2: Algorithm 1 minimizes the budget to achieve the optimal stage execution time.

To illustrate the algorithm, Fig. 3 shows an example where a stage has four (map/reduce) tasks, each being able to run on 3 or 4 candidate machines with different prices and anticipated performance. For instance, in Fig. 3(a), t_3 is the execution time of $task_0$ running on a certain machine. After paying extra $delta_3$, the task can be shifted to the next faster machine with t_2 as the execution time. Since the stage completion time is determined by the slowest task. The algorithm first invests some budget to $task_2$, allowing it to move to the next faster machine (Fig. 3(a)). Unfortunately, after this investment, $task_2$ is still the slowest one, then the algorithm continues to invest budget to this task so that it can run on the next faster machine (Fig. 3(b)). Eventually $task_2$ is no longer the slowest task and instead $task_3$ is. Consequently, the algorithm starts to allocate budget to $task_3$ (Fig. 3(c)) in order to minimize its execution time. This process can be repeated until no more budget is left over to invest (Fig. 3(d)). At that time, the algorithm completes and the minimum stage execution time under the given budget constraint is computed.

4.1.2 Global Distribution

Now we consider the second step. Given the results of Algorithm 1 for all the κ stages, we try to use a dynamic programming recursion to compute the global optimal result. To this end, we use $T(j, r)$ to represent the minimum total time to complete stages indexed from j to κ when budget r is available, and have the following recursion ($0 < j \leq \kappa, 0 < r \leq B$):

$$T(j,r) = \begin{cases} \min\limits_{0<q\leq r}\{T_j(n_j, q) + T(j+1, r-q)\} & \text{if } j < \kappa \\ T_j(n_j, r) & \text{if } j = \kappa \end{cases} \quad (6)$$

where the optimal solution can be found in $T(1, B)$. The scheduling scheme can be reconstructed from $T(1, B)$ by recursively backtracking the Dynamic Programming (DP) matrix in (6) up to the initial budget distribution at stage κ which can, phase by phase, steer to the final optimal result. To this end, in addition to the time value, we only store the budget q and the index of the previous

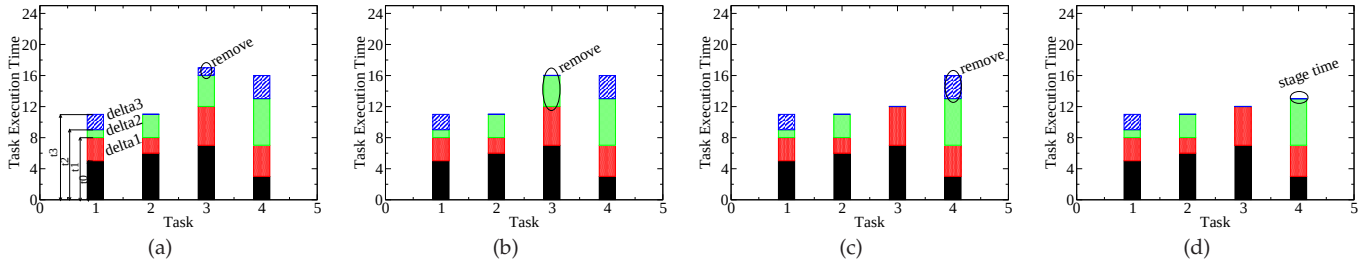

Fig. 3: An illustrative example of the in-stage greeddy algorithm on budget distribution

stage (i.e., $T(j+1, r-q)$) in each cell of the matrix since, given the budget for each stage, we can simply use Algorithm 1 to recompute the budget distribution.

Theorem 4.3: Given budget B for a κ-stage MapReduce job, each stage j having n_j tasks, Recursion (6) yields an optimal solution to the distribution of budget B to all the κ stages with time complexity $O(\kappa B^2)$ when $T_j(n_j, q), 0 < j \leq \kappa, 0 < q \leq B$ is pre-computed. Otherwise, $O(nB^3)$ is required if computed online.

Proof: We prove this by induction on the number of stages (κ). Let the number of stages, $\kappa = 1$. Clearly, given budget r, the optimal solution is obtained by $T_j(n_j, r)$. Suppose there are κ stages. Consider stages j and $j+1$. As an induction hypothesis, let $T(j+1, p)$ be an optimal solution to stages from $j+1$ to κ given budget p. We will show that $T(j, r)$ is an optimal solution to stages from j to κ under budget constraint r. In order to find the optimal distribution of the budget r among $\kappa - j + 1$ stages, we need to consider all possibilities. To this end, we assign q units to the first stage j and the remaining $r-q$ units to the leftover stages from $j+1$ to κ, and allow q to be varied in the range of $(0, r]$. Clearly, the recursion chooses the minimum of all these, thus serving all the stages from j to κ using minimum time.

Finally, at stage 1, since there are no more previous stage, the recursion (6) yields the optimal result $T(1, B)$ for the workflow.

There are $O(\kappa B)$ elements in the DP matrix (6). For each element, the computation complexity is at most $O(B)$ when $T_j(n_j, q), 0 < q \leq r$ have been pre-computed. Therefore, the total time complexity is $O(\kappa B^2)$. Otherwise, it would be written as $B(\sum_{j=1}^{\kappa} \sum_{q=0}^{B} (\frac{q \log n_j}{\min_{0 \leq l \leq n_j} \{\delta_{jl}\}} + n_j))$, which is upper bounded by $O(nB^3)$ given $n = \sum_{j=1}^{\kappa} n_j$. □

4.2 Efficiency Improvements

In the previous subsection, we presented an optimal solution to the distribution of a given budget among different stages to minimize the workflow execution time. The time complexity of the proposed algorithm is pseudo-polynomial and proportional to the square of the budget, which is fairly high. To address this problem, we now propose two heuristic algorithms that are based on different greedy strategies. The first one, called *Global Greedy Budget* (GGB) extends the idea of Algorithm 1 in computing $T_j(n_j, B_j)$ to the whole multi-stage workflow. The second one, called *Gradual Refinement* (GR), is built upon the recursion (6) in an iterative way, each iteration using different budget allocation granularities to compute the DP matrix so as to gradually refine the final results. Each algorithm offers a specific advantage over the other one. However, our empirical studies show that both are very close to the optimal results in terms of scheduling lengths but enjoy much lower time overhead.

4.2.1 Global-Greedy-Budget Algorithm (GGB)

This algorithm applies the idea of Algorithm 1 with some extensions to the selection of candidate tasks for budget assignments across all the stages of the workflow. The pseudo code of GGB is shown in Algorithm 2. Similar to Algorithm 1, we also need to ensure the given budget has a lower bound $\sum_{j \in [1,\kappa]} B_j$ where $B'_j = \sum_{l \in [0,n_j]} p_{jl}^{m_{jl}}$ that guarantees the completion of the workflow (Lines 2-3). We also use the three profile variables T_{jl}, B_{jl} and M_{jl} for each task J_{jl} in stage j to record its execution time, assigned budget, and selected machine (Lines 6-12).

Since in each stage, the slowest task determines the stage completion time, we first need to allocate the budget to the slowest task in each stage. After the slowest task is allocated, the second slowest will become the bottleneck. In our heuristic, we must consider this fact. To this end, we first identify the slowest and the second slowest tasks in each stage j, which are indexed by jl and jl', respectively. Then we gather these index pairs in a set L thereby determining which task in L should be allocated budget (Lines 14-18). To measure the quality of a budget investment, we define a *utility value*, v_{jl}^u, for each given task J_{jl}, which is a value assigned to an investment on the basis of anticipated performance: [2]

$$v_{jl}^u = \alpha \beta_j + (1 - \alpha) \beta'_j \tag{7}$$

where $\beta_j = \frac{t_{jl}^u - t_{jl'}^{u'}}{p_{jl}^{u-1} - p_{jl}^u} \geq 0$, $\beta'_j = \frac{t_{jl}^u - t_{jl}^{u-1}}{p_{jl}^{u-1} - p_{jl}^u} \geq 0$, and α is defined as:

$$\alpha = \begin{cases} 1 & \text{if } \sum_{j=1}^{\kappa} \beta_j > 0 \\ 0 & \text{Otherwise} \end{cases} \tag{8}$$

2. Recall that the sequences of t_{jl}^u and p_{jl}^u are sorted, respectively in Table 1.

Algorithm 2 Global-Greedy-Budget Algorithm (GGB)

1: **procedure** $T(1, B)$ ▷ Dist. B among κ stages
2: $B' = B - \sum_{j\in[1,\kappa]} B'_j$ ▷ $B'_j = \sum_{l\in[0,n_j]} p_{jl}^{m_{jl}}$
3: **if** $B' < 0$ **then return** $(+\infty)$
4: **end if** ▷ No sufficient budget!
5: ▷ Initialization
6: **for** $j \in [1, \kappa]$ **do** ▷ $O(\sum_{j=1}^{\kappa} n_j) = \#\ of\ tasks$
7: **for** $J_{jl} \in J_j$ **do**
8: $T_{jl} \leftarrow t_{jl}^{m_{jl}}$ ▷ record exec. time
9: $B_{jl} \leftarrow p_{jl}^{m_{jl}}$ ▷ record budget dist.
10: $M_{jl} \leftarrow m_{jl}$ ▷ record assigned machine index.
11: **end for**
12: **end for**
13: **while** $B' \geq 0$ **do** ▷ $\leq O(\frac{B}{\min_{1\leq j\leq\kappa, 0\leq l\leq n_j}\{\delta_{jl}\}})$
14: $L \leftarrow \varnothing$
15: **for** $j \in [1, \kappa]$ **do** ▷ $O(\sum_{j=1}^{\kappa} \log n_j)$
16: $< jl, jl' >^* \leftarrow \arg\max_{l\in[0,n_j]}\{T_{jl}(B_{jl})\}$
17: $L \leftarrow L \cup \{< jl, jl' >^*\}$ ▷ $|L| = \kappa$
18: **end for**
19: $V \leftarrow \varnothing$
20: **for** $< jl, jl' > \in L$ **do** ▷ $O(\kappa)$
21: $u \leftarrow M_{jl}$
22: **if** $u > 1$ **then**
23: $< p_{jl}^{u-1}, p_{jl}^{u} > \leftarrow$ **Lookup**$(J_{jl}, u-1, u)$
24: $v_{jl}^{u} \leftarrow \alpha\beta_j + (1-\alpha)\beta'_j$
25: $V \leftarrow V \cup \{v_{jl}^{u}\}$ ▷ $|V| \leq \kappa$
26: **end if**
27: **end for**
28: **while** $V \neq \varnothing$ **do** ▷ $O(\kappa \log \kappa)$
29: ▷ sel. task with max. u.value
30: $jl^* \leftarrow \arg\max_{v_{jl}^{u}\in V}\{v_{jl}^{u}\}$
31: $u \leftarrow M_{jl^*}$ ▷ Lookup matrix in Table 1
32: $\delta_{jl^*} \leftarrow p_{jl^*}^{u-1} - p_{jl^*}^{u}$ ▷ $u > 1$
33: **if** $B' \geq \delta_{jl^*}$ **then** ▷ reduce J_{jl^*}'s time
34: $B' \leftarrow B' - \delta_{jl^*}$
35: $B_{jl^*} \leftarrow B_{jl^*} + \delta_{jl^*}$
36: $T_{jl^*} \leftarrow t_{jl^*}^{u-1}$
37: $M_{jl^*} \leftarrow u - 1$
38: **break** ▷ restart from scratch
39: **else**
40: $V \leftarrow V \setminus \{v_{jl^*}^{u}\}$ ▷ select the next one in V
41: **end if**
42: **end while**
43: **if** $V = \varnothing$ **then**
44: **return** ▷ $B_j = \sum_{l\in[0,n_j]} B_{jl}$
45: **end if**
46: **end while**
47: **end procedure**

β_j represents time saving on per-budget unit when task J_{jl} is moved from machine u to run on the next faster machine $u-1$ in stage j ($\beta_j > 0$) while β'_j is used when there are multiple slowest tasks in stage j ($\beta_j = 0$). α is defined to allow β_j to have a higher priority than β'_j in task selection. Put simply, unless for $\forall j \in [1, \kappa], \beta_j = 0$ in which case β'_j is used, we use the value of $\beta_j, j \in [1, \kappa]$ as the criteria to select the allocated tasks.

In the algorithm, all the values of the tasks in L are collected into a set V (Lines 19-28). We note that the tasks running on machine $u = 1$ in each stage have no definition of this value since they are already running on the fastest machine under the given budget (and thus no further improvement is available).

Given set V, we can iterate over it to select the task in V that has the largest utility value, indexed by jl^*, to be allocated budget for minimizing the stage computation time (Lines 29-30). We fist obtain the machine u to which the selected task is currently mapped and then compute the extra monetary cost δ_{jl^*} if the task is moved from u to the next faster machine $u-1$ (Lines 31-32). If the leftover budget B' is insufficient, the selected task will not be considered and removed from V (Line 40). In the next step, a task in a different stage will be selected for budget allocation (given each stage has at most one task in V). This process will be continued until either the leftover budget B' is sufficient for a selected task or V becomes empty. In the former case, δ_{jl^*} will be deducted from B' and added to the select task. At the same time, other profile information related to this allocation is also updated (Lines 33-37). After this, the algorithm exits from the loop and repeats the computation of L (Line 13) since L has been changed due to this allocation. In the latter case, when V becomes empty, the algorithm returns directly, indicating that the final results of the budget distribution and the associated execution time of each tasks in each stage are available as recorded in the corresponding profile variables.

Theorem 4.4: The time complexity of GGB is not greater than $O(B(n + \kappa \log \kappa))$. In particular, when $n \geq \kappa \log \kappa$, the complexity of GGB is upper bounded by $O(nB)$.

Proof: The time complexity of this algorithm is largely determined by the nested loops (Lines 13-42). Since each allocation of the budget B' is at least $\min_{1\leq j\leq\kappa, 0\leq l\leq n_j}\{\delta_{jl}\}$, the algorithm has at most $O(\frac{B}{\min\{\delta_{jl}\}}), 1 \leq j \leq \kappa, 0 \leq l \leq n_j$ iterations at Line 13. On the other hand, if some advanced data structure such as a *priority queue* is used to optimize the search process, the algorithm can achieve a time complexity of $O(\sum_{j=1}^{\kappa} \log n_j)$ at Line 15 and $O(\kappa \log \kappa)$ at Line 29. Therefore, the overall time complexity can be written as

$$O(n + \frac{B}{\min\{\delta_{jl}\}}(\sum_{j=1}^{\kappa} \log n_j + \kappa \log \kappa)) < O(B(n + \kappa \log \kappa)) \tag{9}$$

where $\delta_{jl} = p_{jl}^{u-1} - p_{jl}^{u}$, $1 \leq j \leq \kappa, 0 \leq l \leq n_j$ and $n = \sum_{j=1}^{\kappa} n_j$ the total number of tasks in the workflow. Here, we leverage the fact that $\log n < n$. Obviously, when $n \geq \kappa \log \kappa$, which is reasonable in multi-stage MapReduce jobs, we obtain a time complexity of $O(nB)$. □

4.2.2 Gradual Refinement Algorithm (GR)

Given the results of the per-stage and global budget distributions, in this subsection we propose the GR algorithm to drastically reduce time complexity in most cases.

Algorithm 3 Gradual Refinement Algorithm (GR)

1: **procedure** $T(1, B)$ ▷ Dist. B among κ stages
2: $B' = B - \sum_{j\in[1,\kappa]} B_j$ ▷ $B_j = \sum_{l\in[0,n_j]} p_{jl}^{m_{jl}}$
3: **if** $B' < 0$ **then return** $+\infty$
4: **end if** ▷ No sufficient budget!
5: ▷ Update Table 1 of each task in each stage
6: ▷ Stages is a global var.
7: **for** $j \in [0, \kappa)$ **do**
8: $TaskTabs \leftarrow$ **Stages.getStage**(j)
9: **for** $l \in [0, n_j]$ **do**
10: $TaskTabs[l]$.**substractPrice**$(p_{jl}^{m_{jl}})$
11: **end for**
12: **end for**
13: $r \leftarrow 10^k$
14: **while** $r \geq 1$ **do**
15: $< C, R > \leftarrow < B'/r, B'\%r >$
16: **for** $b \in [0, C]$ **do**
17: $T[\kappa-1][b] \leftarrow < T_{\kappa-1}(n_{\kappa-1}, b/r), 0 >$
18: **end for**
19: **for** $j \in [\kappa-2, 0]$ **do**
20: **for** $b \in [0, C]$ **do**
21: $T[j][r] \leftarrow < +\infty, 0 >$
22: $q \leftarrow 0$
23: **while** $q \leq b$ **do**
24: $t1 \leftarrow T_j(n_j, q/r)$
25: $t2 \leftarrow T[j+1][b-q]$
26: **if** $T[j][r].tval > t1 + t2$ **then**
27: $T[j][r] \leftarrow < t1 + t2, b - q >$
28: **end if**
29: $q++$
30: **end while**
31: **end for**
32: **end for**
33: $b' \leftarrow$ **constructSchedule**$(0, C, R)$
34: $r \leftarrow r/10$
35: $B' \leftarrow b'$ ▷ $b' = \sum_{i=1}^{\kappa} b_i + R$
36: **end while**
37: **end procedure**

This algorithm consists of two parts. First, we consider the distribution of $B' = B - \sum_{j\in[1,\kappa]}\{B'_j\}$ instead of B in Recursion (6), where $B'_j = \sum_{0\leq l\leq n_j} p_{jl}^{m_{jl}}$ is the lower bound of the budget of stage j. This optimization is simple yet effective to minimize the size of the DP matrix. Second, we optimize the selection of the size of q to iterate over the B in (6). Instead of using a fixed value of 1 as the indivisible cost unit, we can continuously select 10^k, $10^{k-1}, ..., 1$ units as the incremental budget rates in the computation of (6), each being built upon its immediately previous result. In this way, we can progressively approach the optimal result while drastically reducing the time complexity. The details of the algorithm are formally described in Algorithm 3.

After getting the remaining budget B', we update the time-price table (Table 1) of each task by subtracting its minimal service rate from each price (Lines 7-12). This step is necessary as now we are considering the distribution of B' instead of B. It is accomplished by accessing a global variable $Stages$ that stores the information of all the stages. Then the algorithm enters a main loop (Lines 14-37). Each loop leverages Recursion (6) to compute a DP matrix $T[\kappa][C+1], C \leftarrow B'/r$ using a different budget rate r (initialized by 10^k). The distributed and remaining budgets under r are stored in C and R respectively so that they can be used in the current and the next rounds (Line 15). In the computation of Recursion (6), we not only keep the execution time but also store the budget index of the previous step in each cell of the matrix (Lines 17 and 27). This is necessary for reconstructing the schedule, as well as gathering the allocated budget that is not used in the current loop, and prorate to the next round. For example, suppose given $r \leftarrow 10$, we compute $T(4, 70) = \min\{T_4(2, 30) + T(3, 40)\}$. If $T_4(2, 30)$ is allocated 30 units but only uses 27, then 3 units are left for the next round (where $r \leftarrow 1$).

Algorithm 4 Construct scheduler and gather unused budget

1: **procedure** $constructSchedule(i, j, R)$
2: $< t, p > \leftarrow T[i][j]$
3: $b \leftarrow j - p$
4: $TaskTabs \leftarrow$ **Stages.getStage**(i)
5: **if** $i = \kappa - 1$ **then**
6: $b' \leftarrow T_i(n_i, b)$ ▷ return allocated budget
7: **for** $l \in [0, n_j]$ **do**
8: $TaskTabs[l]$.**substractPrice**(b')
9: **end for**
10: $b_i \leftarrow b - b'$
11: $R \leftarrow R + b_i$
12: **return** R
13: **end if**
14: $b' \leftarrow T_i(n_i, b)$
15: **for** $l \in [0, n_j]$ **do**
16: $TaskTabs[l]$.**substractPrice**(b')
17: **end for**
18: $b_i \leftarrow b - b'$
19: $R \leftarrow R + b_i$
20: **return constructSchedule**$(i + 1, p, R)$
21: **end procedure**

Following the computation of the DP matrix $T[\kappa][C+1]$, the loop ends invoking **constructSchedule**$(0, C, R)$ that constructs the allocation schedule based on the current value of r. There are two other purposes for this construction. First, we can gather the allocated budget that is not used in the current loop (stored in b_i for stage i). Second, we can update the time-price tables of each stage to reflect the current optimal distribution, which forms the basis for the next loop. This step makes the algorithm efficient but non-optimal. The details of these steps are shown in Algorithm 4.

In this algorithm, we first compute the budget allocated to the current stage i (Lines 2-3) and then obtain its tasks. The algorithm is recursive from stage 0 down to stage $\kappa - 1$ where it is returned with the total unused budget represented by R. It is worthwhile to point out the update of the time-price tables in Lines 7-9 and Lines 15-17. The newly allocated budget (i.e., b') to each stage is

deducted from each task's machine price (if it is not zero in that stage) so that the current optimal allocation can be taken as the starting point for the next round as previously discussed. Clearly, the **constructSchedule** function walks over all the stages, and in each stage, it modifies all tasks' time-price table. Therefore, the time complexity of this function is $O(\sum_{j=1}^{\kappa}\sum_{l=0}^{n_j}\log m_{jl}) = O(m)$, which is the total size of the time-price tables. (Recall that prices in each table have been sorted.)

However, the time complexity of the GR algorithm as a whole is difficult to analyze since it is very hard, if not impossible, to bound the remaining budget of each stage (i.e., b_i) for each round of distributions, and thus to determine the remaining budget of the current round to be used in the next one. Consequently, here we only roughly estimate complexity. To this end, we denote $B' = \rho_k 10^k + \rho_{k-1}10^{k-1} + \rho_{k-2}10^{k-2} + ... + \rho_0$, and have the following lemma to estimate the remaining budget of each round.

Lemma 4.5: Given $\mu_j = \sum_{i=1}^{j-1}\frac{\gamma_{j-i}}{10^i}$, the remaining budget in the tth round of the while loop is $b'_t \leq \mu_k\mu_{k-1}...\mu_t\rho_k 10^k + \mu_{k-1}...\mu_t\rho_{k-1}10^{k-1} + ... + \mu_t\rho_t 10^t + \rho_{t-1}10^{t-1} + ... + \rho_0$ where $\gamma_i \leq 9, 0 \leq i < k$ are dependent on the allocated budget that is not used by each stage.

Proof: We prove this by induction on the number of rounds (t) of the while loop. Initially ($t = 1$), given $B' = \rho_k 10^k + \rho_{k-1}10^{k-1} + \rho_{k-2}10^{k-2} + ... + \rho_0$, we have $C = B'/r = \rho_k$ and $R = B'\%r = \rho_{k-1}10^{k-1} + \rho_{k-2}10^{k-2} + ... + \rho_0$. According to procedure **constructSchedule**$(0, C, R)$, the allocated budget that is not used is $b'_k = \sum_{i=1}^{\kappa} b_i + R$ where $\sum_{i=1}^{\kappa} b_i \leq \sum_{i=1}^{\rho_k}(10^k - T_i(n_i, 10^k))$ since there are at most $C = \rho_k$ stages allocated.[3] Therefore, $\exists\gamma_{k-1}, ..., \gamma_0$, $\sum_{i=1}^{\rho_k} b_i \leq C\gamma_{k-1}10^{k-1} + C\gamma_{k-2}10^{k-2} + ... + C\gamma_0 = C(\frac{\gamma_{k-1}}{10} + \frac{\gamma_{k-2}}{10^2} + \frac{\gamma_{k-3}}{10^3} + ... + \frac{\gamma_0}{10^k})10^k = C\mu_k 10^k$, then,

$$\begin{aligned} b'_k \leq & C\mu_k 10^k + \rho_{k-1}10^{k-1} + ... + \rho_0 \\ = & \mu_k\rho_k 10^k + \rho_{k-1}10^{k-1} + ... + \rho_0 \end{aligned} \tag{10}$$

Consider rounds t and $t+1$. As an induction hypothesis, let $b'_t \leq \mu_k\mu_{k-1}...\mu_t\rho_k 10^k + \mu_{k-1}...\mu_t\rho_{k-1}10^{k-1} + ... + \mu_t\rho_t 10^t + \rho_{t-1}10^{t-1} + ... + \rho_0$. In the $(t+1)$th round, we have $C = \mu_k\mu_{k-1}...\mu_t\rho_k 10^{k-t+1} + \mu_{k-1}\mu_{k-2}...\mu_t\rho_{k-1}10^{k-t} + ... + \mu_t\rho_t 10 + \rho_{t-1}$ and $R = \rho_{t-2}10^{t-2} + ... + \rho_0$. Since at most C are allocated, we have $\sum_{i=1}^{\rho_k} b_i \leq C\gamma_{t-2}10^{t-2} + C\gamma_{t-3}10^{t-3} + ... + C\gamma_0 = C(\frac{\gamma_{t-2}}{10} + \frac{\gamma_{t-3}}{10^2} + \frac{\gamma_{t-4}}{10^3} + ... + \frac{\gamma_0}{10^{t-1}})10^{t-1} = C\mu_{t-1}10^{t-1}$, then we have

$$\begin{aligned} b'_{t-1} \leq & C\mu_{t-1}10^{t-1} + \rho_{t-2}10^{t-2} + ... + \rho_0 \\ = & (\mu_k\mu_{k-1}...\mu_t\rho_k 10^{k-t+1} + \mu_{k-1}\mu_{k-2}...\mu_t\rho_{k-1}10^{k-t} \\ & + ... + \mu_t\rho_t 10 + \rho_{t-1})\mu_{t-1}10^{t-1} + \rho_{t-2}10^{t-2} + ... + \rho_0 \\ = & \mu_k\mu_{k-1}...\mu_t\mu_{t-1}\rho_k 10^k + \mu_{k-1}...\mu_t\mu_{t-1}\rho_{k-1}10^{k-1} \\ & + ... + \mu_t\mu_{t-1}\rho_t 10^t + \mu_{t-1}\rho_{t-1}10^{t-1} + ... + \rho_{t-2}10^{t-2} \\ & + ... + \rho_0 \end{aligned} \tag{11}$$

3. If $\rho_k > \kappa$, multiple 10^k units could be assigned to the same stage. In this case, we can split the stage into several dummy stages, each being allocated 10^k units. Then, we can follow the same arguments.

Hence, the proof. □

Since $\forall j, \mu_j < 1$, this lemma demonstrates that for GR, the remaining budget in each round of the while loop is nearly exponentially decreased. With this result, we can have the following theorem.

Theorem 4.6: The time complexity of GR is $O(\sum_{t=1}^{\log B}(\kappa C_t^2 + m)$ where $C_t = \mu_k\mu_{k-1}...\mu_{t+1}\rho_k 10^{k-t} + \mu_{k-1}\mu_{k-2}...\mu_{t+1}\rho_{k-1}10^{k-t-1} + ... + \rho_t$.

Proof: There is a total of $\log B$ rounds in the while loop, and in each round t, we need to a) compute the DP matrix with a size of C_t according to Recursion (6) (which has complexity of $O(\kappa C_t^2)$), and then b) count the time used in **constructSchedule**$(0, C, R)$. □

Ideally, if the allocated budget is fully used in each stage (where $\mu_j = 0$), the algorithm is $\kappa\sum_{i=0}^{k}\rho_i^2$, which is the lower bound. But in practice, the actual speed of the algorithm is also determined by the parameter γ sequence, which is different from stage to stage, and from job to job. We will investigate this when discussing our empirical studies.

4.3 Optimization under Deadline Constraints

In this section, we discuss task-level scheduling for MapReduce workflows with the goal of optimizing Equation (5), which pertains to deadline constraints. Since most of the techniques we presented earlier can be applied to this problem, the discussion is brief. We partition the total deadline D into κ parts, denoted by $D_0, D_1, ..., D_{\kappa-1}$ such that $\sum_{0\leq j<\kappa} D_j \leq D$. For a given deadline D_j for stage j, we must ensure that all tasks of this stage can be finished within D_j. Thus, in order to minimize cost, we need to select, for each task, the machine on which the execution time of this task is closest to D_j. Formally $C_{jl}(D_j) = p_{jl}^u$, $t_{jl}^{u-1} < D_j < t_{jl}^{u+1}$. (Obviously, $C_{jl}(D_j)$ is the minimum cost to finish stage j within D_j. If stage j cannot be finished within D_j, $C_{jl}(D_j) = +\infty$.) We then can compute Equation (4).

By following the same approach as in Section 4.1, we can derive the optimal solution. However, this strategy is not efficient since allocation to each stage, as well as optimal distribution within each stage, cannot be computed in a simple way.

Alternatively, we can transform this problem into the standard MCKS problem by constructing κ classes in parallel, each corresponding to a stage of the workflow. The class j consists of a set of tuples (D_{ji}, C_{ji}) where $1 \leq i \leq \sum_{l\in[0,n_j]} p_{jl}^{m_{jl}}$, representing the total minimum cost C_{ji} for stage j under the given D_{ji}. These pairs are computed as follows:

1) for each task J_{jl} in stage j, gather its execution time on the candidate machines and put into set S;
2) sort S in ascending order;
3) for each element t_i in S, $D_{ji} \leftarrow t_i$ and then compute $C_{jl}(D_{ji})$ for each task l in stage j. (This step can be further parallelized based on t_i.)

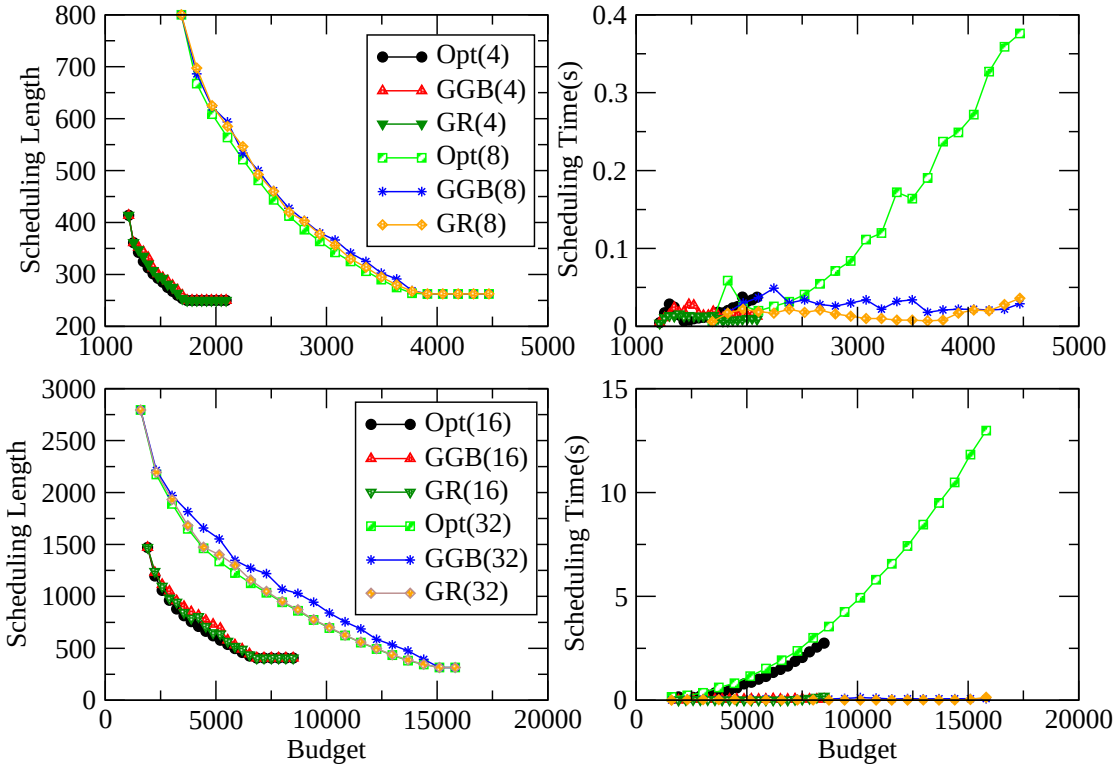

Fig. 4: Impact of time-price table (TP) size on the scheduling length and the scheduling time (Stage:8, Task: ≤ 20/each stage, and the numbers in the brackets represent the different TP table sizes)

The aim of the problem then becomes to pick up exactly one tuple from each class in order to minimize the total cost value of this pick, subject to the deadline constraint, which is a standard multiple-choice knapsack problem equivalent to Equation (5). To optimize the computation, we can remove the tuple (D_{ji}, C_{ji}) from the class if $C_{ji} = +\infty$.

5 Empirical Studies

To verify and evaluate the proposed algorithms and study their performance behaviours in reality, we developed a *Budget Distribution Solver* (BDS) in Java (Java 1.6) that efficiently implements the algorithms for the specified scheduling problem in Hadoop. Since the monetary cost is our primary interest, in BSD we did not consider some properties and features of the network platforms. Rather, we focus on the factors closely related to our research goal. In particular, how efficient the algorithms (i.e., scheduling time) are in minimizing the scheduling lengths of the workflow subject to different budget constraints is our major concern. Moreover, since the remaining budget after workflow scheduling always reflects the profit that the MapReduce providers could make, we also compare it between the algorithms.

The BDS accepts as an input a batch of MapReduce jobs that are organized as a multi-stage fork&join workflow by the scheduler at run-time. Each task of the job is associated with a time-price table, which is pre-defined by the cloud providers. As a consequence, the BDS can be configured with several parameters, including those described time-price tables, the number of tasks in each stage and the total number of stages in the workflow. Since there is no model available to these parameters, we assume that they are automatically generated in a uniform distribution, whose results can form a baseline for further studies. In particular, the task execution time and the corresponding prices are assumed to be varied in the ranges of [1, 12.5*table_size] and [1, 10*table_size], respectively. The rationale behind this assumption is twofold. First of all, the distribution of these parameters do not have any impact on the results of our scheduling algorithms. Second, big table size usually manifest the heterogeneity of the cluster, which implies a broad range of task execution time. Again, the table sizes are determined by the available resources and specified by the cloud providers in advance.

Intuitively, with the table size being increased, the scheduler has more choices to select the candidate machines to execute a task. On the other hand, in each experiment we allow the budget resources to be increased from its lower bound to upper bound and thereby comparing the scheduling lengths and the scheduling time of the proposed algorithms with respect to different configuration parameters. Here, the lower and upper

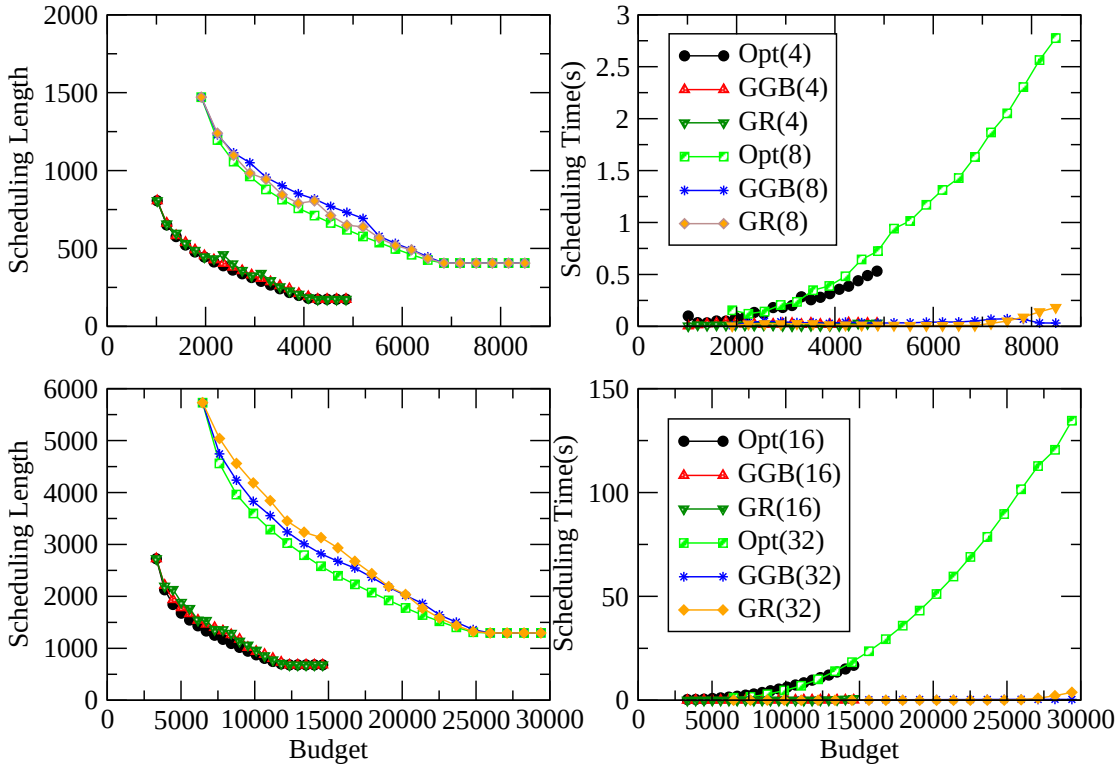

Fig. 5: Impact of the number of stages on the total scheduling length and scheduling time (Task: ≤ 20, Table Size ≤ 16, and the numbers in the brackets represent the different number of stages)

bounds are defined to be the minimal and maximal budget resources, respectively, that can be used to complete the workflow.

All the experiments are conducted by comparing the proposed GGB and GR algorithms with the optimal algorithm and the numerical scheduling time results (average over five trials except for Fig. 6) are obtained from running the scheduling algorithms on a Ubuntu 12.04 platform having a hardware configuration with a total of 8 processors (a quadcore CPU with each core having 2 hardware threads) activated, each with 1600 cpu MHz and 8192K cache.

5.1 Impact of Time-Price Table Size

We first evaluate the impact of the time-price table sizes on the total scheduling length of the workflow with respect to different budget constraints. To this end, we fix an 8-stage workflow with at most 20 tasks in each stage, and the size of the time-price table associated with each task is varied by 4, 8, 16 and 32.

The results of algorithm GGB and GR compared to the optimal algorithm are shown in Fig. 4. With the budget increasing, for all sizes of the tables, the scheduling lengths are super-linearly decreased. These results are interesting and hard to make from the analysis of the algorithm. We attribute these results to the fact that the opportunities of reducing the execution time of each stage are super-linearly increased with the budget growth, especially for those large size workflows. This phenomenon implies that the ratio *performance/cost* is increased if cloud users are willing to pay more for MapReduce computation.

This figure also provides evidence that the performances of both GGB and GR are very close to the optimal algorithm, but their scheduling times are relatively stable and significantly less than that of the optimal algorithm, which is quadratic in its time complexity. These results not only demonstrate compared to the optimal, how GGB and GR are resilient against the changes of the table size, a desired feature in practice, but also show how effective and efficient the proposed GGB and GR algorithms are to achieve the best performance for MapReduce workflows subject to different budget constraints.

5.2 Impact of Workflow Size

In this set of experiments, we evaluate the performance changes with respect to different workflow sizes when the budget resources for each workflow are increased from the lower bound to the upper bound as we defined before. To this end, we fix the maximum number of tasks in the MapReduce workflow to 20 in each stage,

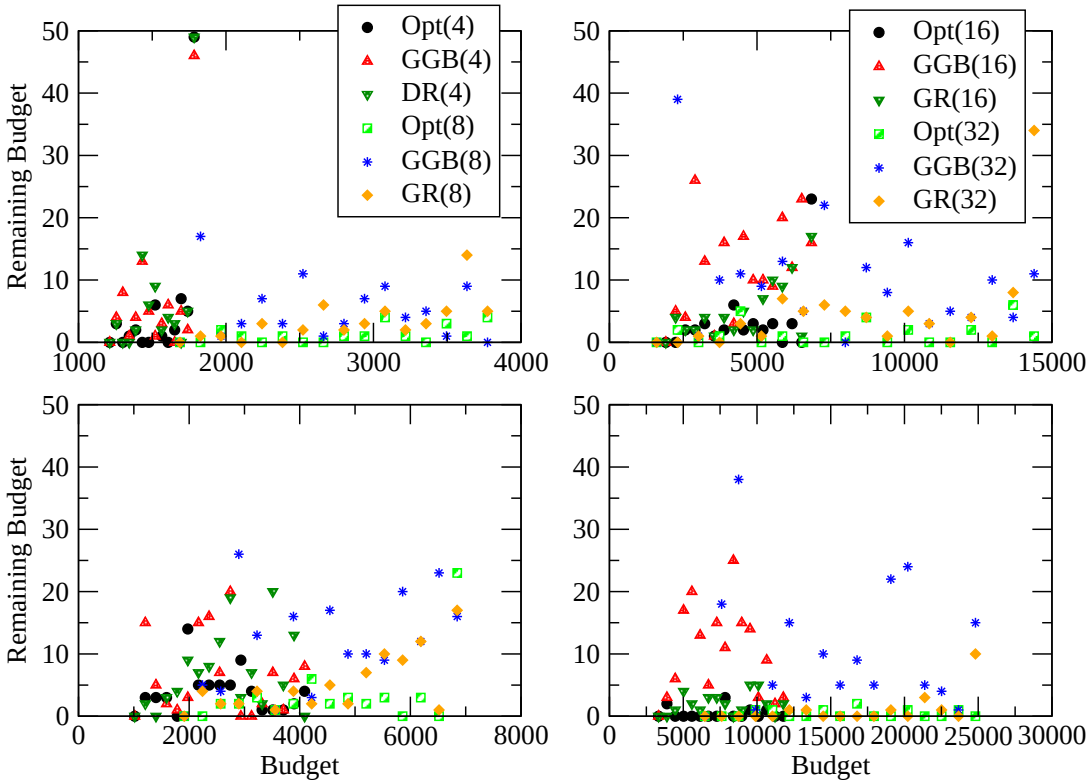

Fig. 6: Comparison of remaining budget between the optimal algorithm, GGB and GR. The top two sub-graphs show the case that the number of tasks ≤ 20, stages is 8, and the time-price table sizes are varied from 4 to 32 (shown in the brackets). By contrast, the bottom two sub-graphs are the cases that the number of stages are changed from 4 to 32 (shown in the brackets) while the table size is fixed to 16 and the number of task is ≤ 20.

and each task is associated with a time-price table with a maximum size of 16. We vary the number of stages from $4, 8, 16$ to 32, and observe the performance and scheduling time changes in Fig. 5. From this figure, we can see that all the algorithms exhibit the same performance patterns with those we observed when the impact of the table size is considered. These results are expected as both the number of stages and the size of tables are linearly correlated with the total workloads in the computation. This observation can be also made when the number of tasks in each stage is changed.

5.3 GGB vs. GR

It is interesting to compare GGB and GR. Although their overall performances are close to each other, there are still some performance gaps between them, which are different from case to case. Given the different table sizes, we can see that GR is constantly better or at least not worse than GGB in terms of the scheduling length and the scheduling time (Fig. 4). However, we can not observe similar phenomenon in the case when the number of stages is varied where neither algorithm is constantly better than the other (Fig. 5). The reasons behind is complicated, and mostly due to the different algorithm behaviours. For example, as the budget resources approach to the upper bound, the execution time of GR could be increased as more budget resources could be left for computing the DP matrix in each iteration.

In addition to the scheduling length and the execution time, another interesting feature of the algorithms is the remaining budget after the allocation of the computation. The remaining budget is important as it could be an indicator of the profit that the MapReduce providers can earn. Fig. 6 compares the remaining budget between the optimal algorithm and the two proposed greedy algorithms. In most cases, the optimal algorithm has the minimum remaining budget, which means it fully utilizes the resources to achieve the best performance while for the greedy algorithms, the minimum remaining budget only indicates that they once made unwise allocations during the scheduling process. By comparing GGB and GR, one can easily see GGB in most cases has more remaining budget not used in the computation than GR, which is also consistent with the observation that GR has better performance than GGB.

In summary, based on our experiments, GR is preferable as in most cases it is better than GGB. However, when the budget is not a concern for the computation,

using GGB may be a better choice because while offering a performance that is very close to GR, it always leaves more remaining budget after computation, which could be viewed as the profits of the cloud service providers.

5.4 Optimality Verification

We verify the optimality of the GR algorithm when r is initially set to one by using the same set of workflows as in the previous experiments. To this end, by following the principle of Recursion (6), we design another dynamic programming algorithm as a per-stage distribution algorithm which is shown in Recursion (12) where $T_i[j,b]$ represents the minimum time to complete jobs indexed from j to n_j given budget b, in which $0 \leq i < \kappa$, $0 \leq l \leq n_i$, and $0 < b \leq B_i$.

$$\begin{cases} T_i[j,b] = \min\limits_{0<q\leq b}\{\max\{T_{i\_j}[q], T_i[j+1, b-q]\}\} \\ T_i[n_i, B_{i\_n_i}] = T_{i\_n_i}(B_{i\_n_i}) \qquad\qquad B_{i\_n_i} \leq B_i \end{cases} \tag{12}$$

The optimal solution to stage i can be found in $T_i[0, B_i]$. Given the proof of Recursion (6), the correctness of this algorithm is easy to follow. We combine this algorithm with Recursion (6) to achieve the global results of the workloads in our first experiment and compare these results with our current ones. The comparison confirms the optimality of the greedy algorithm (Algorithm 1 in Section 4.1.1).

6 Conclusions

In this paper, we studied two practical constraints on budget and deadline for the scheduling of a batch of MapReduce jobs as a workflow on a set of (virtual) machines in the Cloud. First, we focused on the scheduling-length optimization under budget constraints. We designed a global optimal algorithm by combining dynamic programming techniques with a local greedy algorithm for budget distribution on per-stage basis, which was also shown to be optimal.

Then, with this result, we designed two heuristic algorithms, GGB and GR, which are based on different greedy strategies to reduce the time complexity in minimizing the scheduling lengths of the workflows without breaking the budget. Our empirical studies reveal that both the GGB and GR algorithms, each exhibiting a distinct advantage over the other, are very close to the optimal algorithm in terms of the scheduling lengths but entail much lower time overhead.

Finally, we briefly discussed the scheduling algorithm under deadline constraints where we convert the problem into the standard MCKS problem via a parallel transformation.

Admittedly, our model for the budget-driven scheduling of the MapReduce workflows is relatively simple, which might not fully reflect some advanced features in reality such as the speculative task scheduling, redundant computing for fault tolerance, dynamic pricing [46] and so on. However, it at least makes a reasonable use case as a baseline to demonstrate how cost-effective scheduling of the MapReduce workflows could be available in Cloud computing. The advanced features in the scheduling with respect to the budget constraints will be considered in our future work.

Clearly, the full infrastructure required to manage, schedule a batch of MapReduce jobs using the proposed algorithms in Hadoop would be a substantial implementation project. Our current focus was on providing simulation-based evidences to illustrate the performance advantages of the proposed algorithms. Implementing the full system in a real cloud computing construct (Eg. Amazon) will also be tackled as future work.

Acknowledgments

The authors also would like to thank anonymous reviewers who gave valuable suggestion that has helped to improve the quality of the manuscript.

References

[1] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Good, "On the use of cloud computing for scientific workflows," in *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, Dec. 2008, pp. 640 –645.

[2] G. Juve and E. Deelman, "Scientific workflows and clouds," *Crossroads*, vol. 16, no. 3, pp. 14–18, Mar. 2010.

[3] G. Juve, E. Deelman, G. B. Berriman, B. P. Berman, and P. Maechling, "An evaluation of the cost and performance of scientific workflows on amazon ec2," *J. Grid Comput.*, vol. 10, no. 1, pp. 5–21, Mar. 2012.

[4] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: the montage example," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08, Piscataway, NJ, USA, 2008, pp. 50:1–50:12.

[5] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[6] S. Papadimitriou and J. Sun, "Disco: Distributed co-clustering with map-reduce: A case study towards petabyte-scale end-to-end mining," in *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, ser. ICDM '08, 2008, pp. 512–521.

[7] Q. Zou, X.-B. Li, W.-R. Jiang, Z.-Y. Lin, G.-L. Li, and K. Chen, "Survey of mapreduce frame operation in bioinformatics," *Briefings in Bioinformatics*, pp. 1–11, 2013.

[8] Amazon Elastic MapReduce, http://aws.amazon.com/elasticmapreduce/ [Online; accessed Jan-11-2014].

[9] Microsofts Apache Hadoop on Windows Azure Services Preview, http://searchcloudapplications.techtarget.com/tip/The-battle-for-cloud-services-Microsoft-vs-Amazon [Online; accessed Jan-11-2014].

[10] Hadoop on Google Cloud Platform, https://cloud.google.com/solutions/hadoop/ [Online; accessed Jan-11-2014].

[11] Teradata Aster Discovery Platform, http://www.asterdata.com/product/discovery-platform.php [Online; accessed Jan-11-2014].

[12] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855744

[13] S. Crago, K. Dunn, P. Eads, L. Hochstein, D.-I. Kang, M. Kang, D. Modium, K. Singh, J. Suh, and J. Walters, "Heterogeneous cloud computing," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, 2011, pp. 378–385.

[14] H. Liu and D. Orban, "Cloud mapreduce: A mapreduce implementation on top of a cloud operating system," in *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, May 2011, pp. 464–474.

[15] S. Ibrahim, H. Jin, B. Cheng, H. Cao, S. Wu, and L. Qi, "Cloudlet: towards mapreduce implementation on virtual machines," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*, ser. HPDC '09, 2009, pp. 65–66.
[16] S. Ibrahim, H. Jin, L. Lu, L. Qi, S. Wu, and X. Shi, "Evaluating mapreduce on virtual machines: The hadoop case," in *Proceedings of the 1st International Conference on Cloud Computing*, ser. CloudCom '09, Jun. 2009, pp. 519–528.
[17] M. Correia, P. Costa, M. Pasin, A. Bessani, F. Ramos, and P. Verissimo, "On the feasibility of byzantine fault-tolerant mapreduce in clouds-of-clouds," in *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, 2012, pp. 448–453.
[18] F. Marozzo, D. Talia, and P. Trunfio, "Enabling reliable mapreduce applications in dynamic cloud infrastructures," *ERCIM News*, vol. 2010, no. 83, pp. 44–45, 2010.
[19] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10, 2010, pp. 265–278.
[20] H. Chang, M. Kodialam, R. Kompella, T. V. Lakshman, M. Lee, and S. Mukherjee, "Scheduling in mapreduce-like systems for fast completion time," in *INFOCOM, 2011 Proceedings IEEE*, 2011, pp. 3074–3082.
[21] H.-H. You, C.-C. Yang, and J.-L. Huang, "A load-aware scheduler for mapreduce framework in heterogeneous cloud environments," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11, 2011, pp. 127–132.
[22] B. Thirumala Rao and L. S. S. Reddy, "Survey on Improved Scheduling in Hadoop MapReduce in Cloud Environments," *International Journal of Computer Applications*, vol. 34, no. 9, pp. 29–33, Nov. 2011.
[23] Y. Li, H. Zhang, and K. H. Kim, "A power-aware scheduling of mapreduce applications in the cloud," in *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, 2011, pp. 613–620.
[24] P. Kondikoppa, C.-H. Chiu, C. Cui, L. Xue, and S.-J. Park, "Network-aware scheduling of mapreduce framework ondistributed clusters over high speed networks," in *Proceedings of the 2012 workshop on Cloud services, federation, and the 8th open cirrus summit*, ser. FederatedClouds '12, 2012, pp. 39–44.
[25] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin, "Improving mapreduce performance through data placement in heterogeneous hadoop clusters," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, 2010, pp. 1–9.
[26] K. Wang, B. Tan, J. Shi, and B. Yang, "Automatic task slots assignment in hadoop mapreduce," in *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*, ser. ASBD '11. New York, NY, USA: ACM, 2011, pp. 24–29. [Online]. Available: http://doi.acm.org/10.1145/2377978.2377982
[27] Z. Guo and G. Fox, "Improving mapreduce performance in heterogeneous network environments and resource utilization," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, ser. CCGRID '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 714–716. [Online]. Available: http://dx.doi.org/10.1109/CCGrid.2012.12
[28] Z. Zhang, L. Cherkasova, and B. T. Loo, "Performance modeling of mapreduce jobs in heterogeneous cloud environments," in *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, ser. CLOUD '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 839–846. [Online]. Available: http://dx.doi.org/10.1109/CLOUD.2013.107
[29] T. Sandholm and K. Lai, "Dynamic proportional share scheduling in hadoop," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, E. Frachtenberg and U. Schwiegelshohn, Eds. Springer Berlin Heidelberg, 2010, vol. 6253, pp. 110–131.
[30] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using hadoop," in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, 2010, pp. 996–1005.
[31] P. Sinha and A. A. Zoltners, "The multiple-choice knapsack problem," *Operations Research*, vol. 27, no. 3, pp. pp. 503–515, 1979.
[32] D. Pisinger, "A minimal algorithm for the multiple-choice knapsack problem." *European Journal of Operational Research*, vol. 83, pp. 394–410, 1994.
[33] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04, 2004, pp. 10–10.
[34] Apache Software Foundation. Hadoop, http://hadoop.apache.org/core [Online; accessed Jan-11-2014].
[35] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07, 2007, pp. 59–72.
[36] Greenplum HD, http://www.greenplum.com [Online; accessed Jan-11-2014].
[37] Hadoop FairScheduler http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html [Online; accessed Jan-11-2014].
[38] Hadoop CapacityScheduler http://hadoop.apache.org/docs/r0.19.1/capacity_scheduler.html [Online; accessed Jan-11-2014].
[39] Y. Li, H. Zhang, and K. H. Kim, "A power-aware scheduling of mapreduce applications in the cloud," in *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, 2011, pp. 613–620.
[40] K. Kc and K. Anyanwu, "Scheduling hadoop jobs to meet deadlines," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, 2010, pp. 388–392.
[41] K. Wang, B. Tan, J. Shi, and B. Yang, "Automatic task slots assignment in hadoop mapreduce," in *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*, ser. ASBD '11, 2011, pp. 24–29.
[42] J. Yu and R. Buyya, "Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms," *Sci. Program.*, vol. 14, no. 3,4, pp. 217–230, Dec. 2006.
[43] L. Zeng, B. Veeravalli, and X. Li, "Scalestar: Budget conscious scheduling precedence-constrained many-task workflow applications in cloud," in *Proceedings of the 2012 IEEE 26th International Conference on Advanced Information Networking and Applications*, ser. AINA '12, 2012, pp. 534–541.
[44] E. Caron, F. Desprez, A. Muresan, and F. Suter, "Budget constrained resource allocation for non-deterministic workflows on an iaas cloud," in *Proceedings of the 12th international conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ser. ICA3PP'12, 2012, pp. 186–201.
[45] H. Xu and B. Li, "Dynamic cloud pricing for revenue maximization," *IEEE Transactions on Cloud Computing*, vol. 1, no. 2, pp. 158–171, 2013.
[46] Amazon EC2, Spot Instances, http://aws.amazon.com/ec2/spot-instances [Online; accessed March-29-2014].

Yang Wang received the BS degree in applied mathematics from Ocean University of China in 1989, and the MS and PhD degrees in computer science from Carleton University and University of Alberta, Canada, in 2001 and 2008, respectively. He is currently working with IBM Center for Advanced Studies (CAS Atlantic), University of New Brunswick, Fredericton, Canada. His research interests include big data analytics in clouds and exploiting heterogeneous multicore processors to accelerate the execution of the Java Virtual Machine.

Wei Shi is an assistant professor at the University of Ontario Institute of Technology. She is also an adjunct professor at Carleton University. She holds a Bachelor of Computer Engineering from Harbin Institute of Technology in China and received her Ph.D. of Computer Science degree from Carleton University in Ottawa, Canada. Prior to her academic career, as a software developer and project manager, she was closely involved in the design and development of a large-scale Electronic Information system for the distribution of welfare benefits in China, as well as of a World Wide Web Information Filtering System for Chinas National Information Security Centre.