One-Restart Algorithm for Scheduling and Offloading in a Hybrid Cloud

Jaya Prakash Champati and Ben Liang
Department of Electrical and Computer Engineering, University of Toronto
{champati,liang}@comm.utoronto.ca

***Abstract*—The hybrid cloud architecture utilizes both privately owned cloud servers and rented instances from public cloud providers, to offer flexible services that are particularly suited to enterprise computing. The task scheduler at a hybrid cloud decides both the selection of tasks to be offloaded to the public cloud and the scheduling of the remaining tasks on the processors at the private cloud. In this work, we consider the problem of minimizing a weighted sum of the makespan at the private cloud and the offloading cost to the public cloud. In contrast to prior works, we do not assume that the task processing times are known a priori. We show that the original problem can be solved by the same algorithms designed toward minimizing the maximum between the makespan and the weighted offloading cost, only with doubling of the competitive ratio. Furthermore, the latter problem can be equivalently transformed into a makespan minimization problem with unrelated processors. In the case where all tasks arrive at time zero, we propose a Greedy-One-Restart (GOR) algorithm based on online estimation of the unknown processing times, and one-time cancellation and rescheduling of tasks that turn out to require long processing times. We derive its competitive ratio and show that it is upper bounded on the order of the square root of the number of private processors, which is a substantial improvement over the best known algorithms in the literature. We present also a tight constant competitive ratio for the special two-processor case. In the case where tasks arrive dynamically with unknown arrival times, we extend GOR to Dynamic-GOR (DGOR) and find its competitive ratio. Further simulation results demonstrate that GOR and DGOR are favorable also in terms of average performance, in comparison with the well-known list scheduling algorithm and idealized offline algorithms.**

I. Introduction

Cloud computing has emerged as a vital technology used by many enterprises and individuals. The cloud infrastructure can be classified into three types: the *private* cloud is owned by an enterprise and is operated solely to serve its own purposes; in contrast, the *public* cloud offers services that are open for public usage; finally, the *hybrid* cloud supports the usage of both private and public cloud services.

The hybrid cloud is attractive to many medium and large enterprises [1]. They already have significant investment in their own private data centers. Together with the additional options provided by computing instances from larger public cloud centers, they offer flexibility, efficiency and expediency over solely using either the private or public cloud alone. It has been predicted that 61% of the enterprises will use a hybrid cloud platform by the end of 2016, and this percentage is expected to increase to 90% [2], [3]. Major vendors such as Amazon, VMware, IBM, and Rackspace are already providing services to build a hybrid cloud.

978-1-4673-7113-1/15/$31.00 ©2015 IEEE

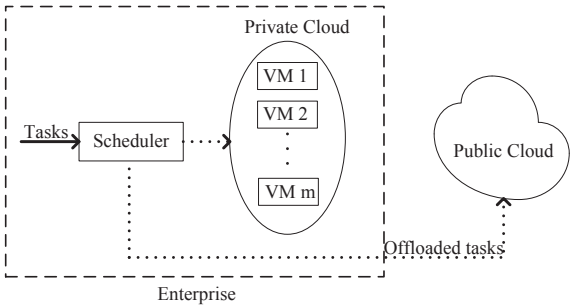

Fig. 1. Hybrid cloud abstract model

Joint scheduling and offloading of tasks submitted by users to a hybrid cloud is an important and non-trivial problem. We consider the hybrid cloud model presented in Figure 1. The private cloud consists of m identical processors, or virtual machines, to serve the incoming tasks. Further, the tasks can also be offloaded to a more powerful public cloud with purchased computing instances. Therefore, the scheduler in the enterprise has two important decisions to make: 1) which tasks are to be offloaded to the public cloud, and 2) how to efficiently schedule the remaining tasks on processors at the private cloud. In this work we address these two questions jointly.

We consider *makespan* as the efficiency measure for scheduling tasks. A scheduler may reduce makespan by offloading the tasks to the public cloud. However, each task offloaded to the public cloud incurs a cost, which may include multiple factors such as network bandwidth usage and the price paid for the computing instances. Since the objectives of makespan and offloading cost are conflicting, in this work we study the joint optimization problem toward minimizing a weighted sum of the makespan and the offloading cost. For tractable analysis, we assume that the makespan of processing offloaded tasks at the public cloud is smaller than the makespan at the private cloud and hence can be ignored, so that the offloading cost is the only limiting factor in using the public cloud. This is a mild assumption when there are many powerful computing instances available at the public cloud.

The problem of minimizing the weighted sum of makespan and the offloading cost, which is denoted by $\mathcal{P}_{sum}$, is known to be NP-hard in general, and approximation algorithms exist [4].

However, to the best of our knowledge, all previous studies assume that the processing times of the tasks are known a priori [5]–[7]. This assumption is impractical, since the scheduler generally does not have information about the task processing time until a task is executed to completion [8]. Therefore, we study the problem without this assumption, which is challenging as it requires online estimation of the processing times.

Our general approach to solve this problem is as follows. We first consider a problem of minimizing the maximum of the makespan on the m private processors and the weighted offloading cost to the public cloud, which is denoted by $\mathcal{P}_{max}$. We observe that $\mathcal{P}_{max}$ is equivalent to a problem of scheduling independent tasks on $m+1$ parallel processors with the objective of minimizing the makespan, where the task processing time on the $(m+1)$-th processor, which represents the public cloud, is equal to its weighted offloading cost. We then show that any θ-competitive algorithm for $\mathcal{P}_{max}$ results in a 2θ-competitive algorithm for $\mathcal{P}_{sum}$. This key property motivates us to first focus on the makespan minimization problem.

The makespan minimization problem is also known to be NP-hard [9]. Further, we show that any algorithm with a predetermined scheduling order to solve $\mathcal{P}_{max}$ has a competitive ratio of at least $\frac{n}{m}-1$, where n is the number of tasks. The $O(n)$ factor in the competitive ratio is due to the makespan penalty incurred by unknowingly scheduling tasks with very large processing times into the private cloud. Therefore, the key to solving $\mathcal{P}_{max}$ is to identify such tasks and offload them to the public cloud, provided that their offloading cost is not too high. This forms the basis of the proposed Greedy-One-Restart (GOR) algorithm, which identifies those tasks, cancels them, and reschedules them. To identify those tasks, GOR uses an *estimation factor* η and the known cost of offloading the tasks. A salient feature of GOR is that its competitive ratio is a function of η.

The main contributions of our work are the following:

- When all tasks arrive at the scheduler at time zero, we propose the GOR algorithm and show that its competitive ratio for $\mathcal{P}_{max}$ is a convex function of the estimation factor η, given by
$$f(\eta) = 1 + \frac{(2m+1)}{m}\eta + \frac{m}{\eta}.$$
Its competitive ratio for $\mathcal{P}_{sum}$ is $2f(\eta)$.
- We show that, for $m \geq 2$, the minimum of $f(\eta)$ is $1+2\sqrt{2m+1}$ for $\eta = \frac{m}{\sqrt{2m+1}}$. Hence, GOR is an $O(\sqrt{m})$-competitive algorithm for both $\mathcal{P}_{max}$ and $\mathcal{P}_{sum}$, which is an improvement over the previously known $O(\log n)$-competitive algorithm given in [10] for $\mathcal{P}_{max}$.
- For the case $m = 1$, we derive an even lower competitive ratio of 4 for GOR. We show that this competitive ratio is tight. We further note that when $m = 1$, $\mathcal{P}_{max}$ is equivalent to minimizing makespan on two unrelated processors. While there exists offline algorithms [11] and online algorithms [10] for this problem, to the best of our knowledge, GOR is the first algorithm to solve the semi-online version, where the processing times on one processor are known and those on the other processor are unknown.
- We further consider the weighted sum minimization problem where tasks arrive dynamically in time and their arrival times are unknown a priori, which is denoted by $\mathcal{P}^r_{sum}$. Adopting a general approach in [10], we extend GOR to Dynamic-GOR (DGOR) to accommodate this case and show that it has $4f(\eta)$ competitive ratio.
- Our simulation results suggest that, in terms of average performance, GOR provides $30-40\%$ improvement over the celebrated *list scheduling* algorithm [12], while DGOR provides $50-90\%$ improvement. They remain competitive compared with idealized offline algorithms.

The rest of the paper is organized as follows. In Section II, we present the related work. The system model is given in Section III. In Section IV, we provide some preliminary analysis essential to later derivations. The GOR algorithm is presented in Section V, and its competitive ratio is derived in Section VI. In Section VII, we consider the special case of $m = 1$. In Section VIII we present DGOR for dynamic tasks with unknown arrival times. We present simulation results in Section IX and conclude in Section X.

II. Related Work

In this section we present prior works related to the weighted sum minimization problem $\mathcal{P}_{sum}$ and the min-max/makespan-minimization problem $\mathcal{P}_{max}$.

A. Minimizing Makespan Plus Weighted Offloading Cost

The general problem of makespan plus weighted penalty is of practical interest in operations research [4], mainly due to its applicability to highly loaded make-to-order production systems. In such a system, accepting all the tasks may result in a delay in the completion orders, so the production firm may reject some tasks at penalty and aim to minimize the makespan plus penalty.

The problem was first studied in [5], under the assumption that processing times are known a priori. The authors considered two cases, all tasks available at time zero and tasks arriving dynamically in time. They proposed a $(2-\frac{1}{m})$-approximation algorithm for the former case, and a Rejection-Total-Penalty (RTP) algorithm that has $\frac{\sqrt{5}+3}{2}$ competitive ratio for the latter case. Further, they showed that this is the best competitive ratio any algorithm can achieve. The authors in [6] proposed a $\frac{1+\sqrt{3}}{2}$-competitive algorithm for the problem with the assumption that all the tasks have unit processing time. The authors in [7] studied the problem with $m = 2$. They proposed $\frac{3}{2}$-competitive algorithms for two variants of the problem.

We emphasize that all these previous works assumed that the task processing times are known a priori, while in our work the processing time of a task is not known until the completion of its execution.

B. Minimizing Makespan on Parallel Processors

In the *offline* setting, where the task processing times are known a priori and all tasks are available at time zero, the problem of scheduling independent tasks on parallel processors to minimize the makespan has been well studied in the literature [13] [9] [14]. In contrast, in the *online* setting, works are sparse. The celebrated *list scheduling* [12] is a greedy algorithm that selects a task from the given set in an arbitrary order and assigns it to whichever processor that becomes idle first. It does not require a priori knowledge of the processing times of the tasks. For m identical parallel processors, it has $(2 - \frac{1}{m})$ competitive ratio. For the case where the processors are unrelated, i.e., the processing times of a task on different processors are independent, an $O(\log n)$-competitive algorithm was proposed in [10].

Our min-max/makespan-minimization problem $\mathcal{P}_{max}$ is a special case of minimizing the makespan on $m + 1$ unrelated parallel processors. To the best of our knowledge, the $O(\log n)$-competitive algorithm of [10] is the only one that solves our problem with a provable competitive ratio. In our work, the proposed GOR algorithm is $O(\sqrt{m})$-competitive. This is a significant improvement noting the fact that, especially in the enterprise cloud environment, the number of tasks n is generally much larger than the number of processors m.

C. Other Hybrid Cloud Models

In the literature, the hybrid cloud architecture has also been studied under various system models [1], [15]–[19]. In these works, the makespan of the tasks was not considered in the design objective. Even though some of them considered the delay in the processing of individual tasks [15] [16], we note that such delay values do not provide a means to compute the makespan since tasks are processed in parallel. Therefore, none of the solutions provided in these works are applicable to our problem.

III. System Model

We consider a hybrid cloud system model as illustrated in Figure 1. The private cloud of the enterprise consists of m identical processors or virtual machines, indexed by $i \in \mathcal{Q} = \{1, \ldots, m\}$. The enterprise also has access to a more powerful public cloud. The tasks are submitted by different users to a scheduler. The scheduler may choose to schedule a task on one of the processors or offload it to the public cloud.

Initially, we focus on the case where all n tasks are available at time zero. In Section VIII, we will extend this to the case of dynamic task arrivals with unknown arrival times. The tasks are independent and non-preemtible. Let $\mathcal{T} = \{1, \ldots, n\}$ be the set of task indices. The processing time of task $j \in \mathcal{T}$ on processor $i \in \mathcal{Q}$ is given by u_j and is unknown. We assume that there are many powerful computing instances available at the public cloud, so that the makespan of processing offloaded tasks can be ignored. However, when a task j is offloaded to the public cloud it incurs a cost $\hat{a}_j$. It may be viewed as an aggregation of various penalties, e.g., transmission energy loss, network bandwidth usage, and price paid for purchased computing instances. We assume that this cost is known for each task before it is processed.

A natural objective for the enterprise is to minimize the makespan of the tasks scheduled on processors 1 to m. The makespan can be reduced by offloading tasks to the public cloud, but offloading a task incurs some cost. Hence, we consider the makespan and the offloading cost jointly, by combining them using a weight parameter w, so that we may tune it to emphasize one objective over the other.

Let $\mathbf{s}$ denote a schedule and $\mathcal{S}$ denote the set of all possible schedules. The schedule $\mathbf{s}$ decides whether to offload a task to the public cloud or process it on one of the processors in $\mathcal{Q}$. Let $\mathcal{T}_i(\mathbf{s})$ be the set of tasks scheduled on processor $i \in \mathcal{Q}$ under schedule $\mathbf{s}$. Given the set of tasks at time 0, the makespan of a schedule $\mathbf{s}$ on processors 1 to m is defined as the time when the processing of the last task from $\cup_{i\in\mathcal{Q}}\mathcal{T}_i(\mathbf{s})$ is completed. It equals $\max_{i\in\mathcal{Q}}\{C_i(\mathbf{s})\}$, where $C_i(\mathbf{s})$ is the completion time of the last task assigned to processor i. Note that the schedule does not know $C_i(\mathbf{s})$ a priori, since u_j are unknown. Let $\mathcal{T}_{m+1}(\mathbf{s})$ denote the set of tasks offloaded to the public cloud under schedule $\mathbf{s}$. The offloading cost of the tasks is given by $\Gamma(\mathbf{s}) = \sum_{j\in\mathcal{T}_{m+1}(\mathbf{s})} \hat{a}_j$. We define $\Upsilon(\mathbf{s}) \triangleq \max_{i\in\mathcal{Q}}\{C_i(\mathbf{s})\} + w\Gamma(\mathbf{s})$ as the *total cost* of schedule $\mathbf{s}$. We are interested in the following sum cost minimization problem $\mathcal{P}_{sum}$:

$$\underset{\mathbf{s}\in\mathcal{S}}{\text{minimize}} \quad \Upsilon(\mathbf{s}).$$

In the offline setting, all parameter values of the tasks are known at time 0. In this case, let $\bar{\mathbf{s}}^*$ denote an optimal schedule. It is known that the offline version of $\mathcal{P}_{sum}$ is NP-hard [5]. In practice, however, the processing time required for a task generally is unknown without first processing it. Therefore, we are interested in the *semi-online* setting, where u_j, $\forall j$, are not known a priori but $\hat{a}_j$, $\forall j$, are known.

The efficacy of an online algorithm is often measured by its competitive ratio in comparison with the optimal offline algorithm. We use the same measure for semi-online algorithm as well. Let P be a problem instance of $\mathcal{P}_{sum}$, $\mathbf{s}(P)$ be the schedule given by an online algorithm and $\bar{\mathbf{s}}^*(P)$ be the schedule given by an optimal offline algorithm. The online algorithm is said to have a competitive ratio θ if and only if

$$\max_{\forall P} \frac{\Upsilon(\mathbf{s}(P))}{\Upsilon(\bar{\mathbf{s}}^*(P))} \leq \theta.$$

Furthermore, θ is said to be tight for the online algorithm if $\exists P$ such that $\Upsilon(\mathbf{s}(P)) = \theta\Upsilon(\bar{\mathbf{s}}^*(P))$.

For convenience of notation, we define $a_j \triangleq w\hat{a}_j, \forall j$.

IV. Preliminary Analysis

We note that the authors in [5] have proposed a $(2 - \frac{1}{m})$-approximation algorithm to solve $\mathcal{P}_{sum}$ in the offline setting. However, to the best of our knowledge, the online or semi-online version of this problem has not been studied under any scenario.

In this work, instead of solving $\mathcal{P}_{sum}$ directly, we first minimize the maximum of the makespan on the m processors

and the weighted offloading cost to the public cloud. This problem is denoted by $\mathcal{P}_{max}$:

$$\underset{\mathbf{s}\in\mathcal{S}}{\text{minimize}} \quad C_{max}(\mathbf{s}),$$

where $C_{max}(\mathbf{s}) \triangleq \max\{\max_{i\in\mathcal{Q}}\{C_i(\mathbf{s})\}, w\Gamma(\mathbf{s})\}$. In the offline setting, let $\mathbf{s}^*$ denote the optimal schedule for $\mathcal{P}_{max}$ and C^*_{max} denote the optimal objective value.

Further, consider a *hypothetical* processor $m+1$ on which the processing time of a task j is $a_j = w\hat{a}_j$. Let $C_{m+1}(\mathbf{s})$ denote the completion time on processor $m+1$. It is given by

$$C_{m+1}(\mathbf{s}) = \sum_{j\in\mathcal{T}_{m+1}(\mathbf{s})} a_j = w\Gamma(\mathbf{s}).$$

Now, $C_{max}(\mathbf{s})$ can be written as $C_{max}(\mathbf{s}) = \max_{i\in\mathcal{Q}\cup\{m+1\}}\{C_i(\mathbf{s})\}$. In other words, $C_{max}(\mathbf{s})$ can be viewed as the makespan of the tasks from $\mathcal{T}$ when they are scheduled on $m+1$ processors, where the processing time of task j on processor $i \in \mathcal{Q}$ is u_j, and its processing time on processor $m+1$ is a_j. This leads to the following conclusion:

Proposition 1. *$\mathcal{P}_{max}$ is equivalent to the problem of minimizing the makespan of the tasks from $\mathcal{T}$ when they are scheduled on $m+1$ processors, where the processing time of task j on processor $i \in \mathcal{Q}$ is u_j and its processing time on processor $m+1$ is a_j.*

Note that, if $a_j = u_j, \forall j$, then $\mathcal{P}_{max}$ is equivalent to minimizing makespan on $m+1$ identical processors, which is an NP-hard problem [9]. Therefore, $\mathcal{P}_{max}$ is NP-hard.

In the following proposition we establish a relation between the problems $\mathcal{P}_{sum}$ and $\mathcal{P}_{max}$.

Proposition 2. *Any θ-competitive algorithm for $\mathcal{P}_{max}$ is a 2θ-competitive algorithm for $\mathcal{P}_{sum}$.*

Proof. The proof is given in the Appendix. □

We therefore conclude that an effective solution to $\mathcal{P}_{max}$ is suitable for $\mathcal{P}_{sum}$ as well. Hence, we next focus on designing an algorithm for $\mathcal{P}_{max}$ with a provable competitive ratio. We note that list scheduling [12] has $(2-\frac{1}{m+1})$ competitive ratio, for scheduling on $m+1$ *identical* parallel processors. In $\mathcal{P}_{max}$ we have $m+1$ parallel processors with m of them identical and the $(m+1)$th processor unrelated to the other processors, i.e., a_j on processor $m+1$ is independent of u_j on processors 1 through m. Therefore, the $(2-\frac{1}{m+1})$ competitive ratio is not applicable when list scheduling is used to solve $\mathcal{P}_{max}$. In fact, in the following theorem we show that list scheduling, or any other simple deterministic semi-online algorithm, has at least $O(n)$ competitive ratio.

Theorem 1. *The competitive ratio of any semi-online algorithm with a pre-determined scheduling order is at least $\frac{n}{m}-1$.*

Proof. We consider a family of problem instances where $u_j = 1, \forall j \in \{1,\ldots,n-m\}$, $u_j = n^2, \forall j \in \{n-m+1,\ldots,n\}$, and $a_j = \frac{n^2}{n-m}, \forall j$. Since u_j are unknown, any algorithm using some pre-determined order to schedule the tasks can only use the knowledge of a_j. However, since all a_j are equal, the tasks cannot be differentiated by such an algorithm. Therefore, this may lead it to schedule the m tasks with processing time n^2 on processors 1 through m, with one task on each processor and all the other tasks on processor $m+1$. This will result in a makespan of n^2. However, the optimal makespan is $\frac{mn^2}{n-m}$, which is achieved by executing tasks $\{n-m+1,\ldots,n\}$ on processor m and performing simple list scheduling for the other tasks on processors 1 through m. This results in a competitive ratio of $\frac{n}{m}-1$. □

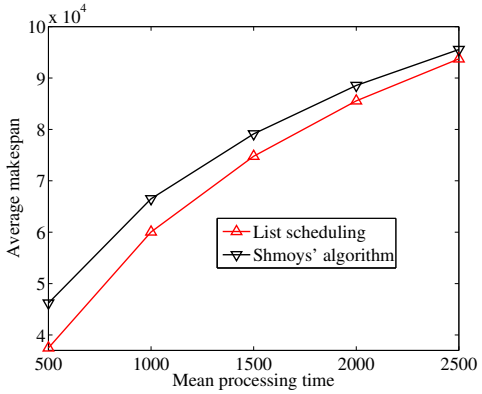

Fig. 2. Comparison of Shmoys' algorithm and list scheduling for varying mean processing time

V. Greedy-One-Restart Algorithm

In this section we first present our design considerations for the proposed algorithm, then describe the algorithm details, and finally provide a sample case study.

A. Design Considerations

The problem $\mathcal{P}_{max}$ is related to minimizing makespan on unrelated parallel processors where the processing times of tasks on *none* of the processors are known. For this fully online version of the problem, Shmoys et. al. in [10] have proposed an $O(\log n)$-competitive algorithm. We will call it Shmoys' algorithm. It estimates the processing times of the tasks and then uses an offline algorithm to schedule them. Tasks that are not completed in the estimated time are cancelled and rescheduled using the same offline algorithm. The procedure is repeated until all tasks are executed to completion.

In Figure 2, we present the average makespan achieved by Shmoys' algorithm and list scheduling, to solve $\mathcal{P}_{max}$ for the case $m = 1$, where u_j and a_j are generated from an exponential distribution with mean 1500. The average is computed over 5000 problem instances. The $\frac{3}{2}$-approximation algorithm given by Potts in [11] is used as the offline component in Shmoys' algorithm.

We observe that, despite its higher than $O(n)$ competitive ratio, list scheduling gives a shorter average makespan than Shmoys' algorithm. We conjecture that this is due to Shmoys' algorithm using crude estimates of the unknown processing times, which results in multiple restarts of some tasks. Although restarting the tasks paves the way to obtain an $O(\log n)$

competitive ratio, it penalizes the makespan on average, as the time already spent in processing a cancelled task is wasted. This motivates us to combine the virtues of both algorithms in a new semi-online design.

Neither list scheduling nor Shmoys' algorithm utilizes the known processing times on processor $m+1$. In contrast, we design the GOR algorithm to judicially utilize the known processing times, while allowing at most *one restart* for any task. The idea behind one restart is that, the tasks that are scheduled on processors 1 to m and have large u_j compared with a_j are identified, cancelled, and rescheduled, so that they may be scheduled on processor $m+1$ in the new schedule. Cancelling a task with large u_j on some processor in $\mathcal{Q}$ may allow some tasks that have smaller u_j values to be scheduled on that processor. At the same time, we avoid the wastage of time in cancelling a task more than once. A more detailed description of the GOR algorithm is given below.

B. Algorithm Description

The GOR algorithm initially estimates that the processing time of any task j on a processor in $\mathcal{Q}$ is ηa_j, where η is an estimation factor. We form a list according to the ascending order of a_j. Tasks from the start of the list are executed one by one on processor $m+1$. From the end of the list, they are scheduled on the processors in $\mathcal{Q}$ using list scheduling. A task j that is scheduled on any processor in $\mathcal{Q}$ and is not finished within the estimated processing time ηa_j is cancelled and set aside. In this work we consider $\eta \geq 1$ so that a task is not cancelled until at least it is processed for a duration equivalent to its weighted offloading cost.

After going through all tasks in the above iteration, those that are cancelled are again sorted and a list is formed in the ascending order of a_j. In the next iteration, the list is scheduled using the same procedure as above, but this time we do not cancel any task, except the last one. Note that in both iterations the last task is scheduled on both processor $m+1$ and some processor in $\mathcal{Q}$. In such a case we cancel the task on one processor if it is either *finished or cancelled* on another processor first.

The details of the algorithm are presented in Algorithm 1. We note that GOR runs in $O(n \log n)$ time due to the need for sorting n tasks. We use $\mathbf{s}^{\text{GOR}}$ to denote the resultant schedule.

Algorithm 1: Greedy-One-Restart (GOR) Algorithm

$l = 1, \mathcal{T}^{(l)} = \mathcal{T}$
while $l \leq 2$ **do**
  $j_1 = 1$, $j_0 = |\mathcal{T}^{(l)}| + 1$
  Sort $\mathcal{T}^{(l)}$ in the ascending order of a_j. WLOG, re-index tasks such that $a_1 \leq a_2 \leq \ldots \leq a_{|\mathcal{T}^{(l)}|}$.
  Start processing task j_1 on processor $m+1$
  for $k = 1$ to $\min\{m, |\mathcal{T}^{(l)}|\}$ **do**
    $j_0 = j_0 - 1$
    Start processing task j_0 on processor k.
    if $l = 1$ **then**
      Cancel task j_0 if its execution time exceeds ηa_{j_0} and include it in $\mathcal{T}^{(l+1)}$
    end if
  end for
  while $j_0 \neq j_1$ **do**
    Wait until next event E occurs
    if $E =$ a task is cancelled or finished on some processor $i \in Q$ **then**
      $j_0 = j_0 - 1$
      Start processing task j_0 on processor i.
      if $l = 1$ **then**
        Cancel task j_0 if its execution time exceeds ηa_{j_0} and include it in $\mathcal{T}^{(l+1)}$
      end if
    else if $E =$ a task is finished on processor $m+1$ **then**
      $j_1 = j_1 + 1$
      Start processing task j_1 on processor $m+1$
    end if
  end while
  $q^{(l)} = j_0 = j_1$
  Task $q^{(l)}$ is scheduled both on processor $m+1$ and some processor $\hat{i} \in \mathcal{Q}$. If task $q^{(l)}$ is finished on processor $m+1$ first, cancel its execution on processor $\hat{i}$. Similarly, if task $q^{(l)}$ is finished or cancelled on processor $\hat{i}$ first, cancel its execution on processor $m+1$.
  $l = l + 1$
end while

C. Case Study

We demonstrate the working of GOR using the family of problem instances given in the proof of Theorem 1, i.e., $u_j = 1, \forall j \in \{1, \ldots, n-m\}$, $u_j = n^2, \forall j \in \{n-m+1, \ldots, n\}$ and $a_j = \frac{n^2}{n-m}, \forall j$. For now we simply use $\eta = 1$. Later, we will study in detail how to choose η. As a_j are the same for all tasks, GOR cannot differentiate the tasks. Therefore, in the first iteration, the schedule given by GOR is equivalent to list scheduling with the exception that the last task is scheduled both on processor $m+1$ and one of the processors in $\mathcal{Q}$. Another exception is that, in the first iteration, the processing time of any task before cancellation on processors 1 through m is $\frac{n^2}{n-m}$ (since $\eta = 1$). Therefore, any task $j \in \{n-m+1, \ldots, n\}$ scheduled on a processor in $\mathcal{Q}$ will be cancelled after being processed for duration $\frac{n^2}{n-m}$. Also, any task $j \in \{1, \ldots, n-m\}$ will be finished in the first iteration since it will be cancelled if scheduled on a processor in $\mathcal{Q}$, as $\frac{n^2}{n-m} > 1 = u_j, \forall j \in \{1, \ldots, n-m\}$. In the second iteration any cancelled task $j \in \{n-m+1, \ldots, n\}$ will be finished on processor $m+1$.

Next, to illustrate how restarting tasks with large processing times improves the worst case bound, we find a simple upper bound for the makespan achieved by GOR for the above family of problem instances. In the worst case, GOR may schedule tasks $n-m+1$ through n on some processor $i \in \mathcal{Q}$. Each of them will be cancelled in the first iteration after being

processed for duration $\frac{n^2}{n-m}$. Therefore, $\frac{mn^2}{n-m}$ time is elapsed on processor i. In this duration, m tasks will be finished on processor $m+1$. Now, the rest of the $n-2m$ tasks can be executed in at most $\frac{n-2m}{m-1}+1$ time. This is because the processing time of those tasks on processors 1 through m is 1. In the second iteration all the tasks $n-m+1$ through n will be scheduled on processor $m+1$. Therefore, the duration of execution of these tasks will be $\frac{mn^2}{n-m}$.

From the above analysis, a simple upper bound for $C_{max}(\mathbf{s})$ is

$$\begin{aligned} C_{max}(\mathbf{s}^{\text{GOR}}) &\leq \frac{mn^2}{n-m} + \frac{n-2m}{m-1} + 1 + \frac{mn^2}{n-m} \\ \Rightarrow \frac{C_{max}(\mathbf{s}^{\text{GOR}})}{C^*_{max}} &\leq \frac{n-m}{mn^2}\left(\frac{2mn^2}{n-m} + \frac{n-m}{m-1}\right) \\ &\leq 2 + \frac{(1-m/n)^2}{m(m-1)}. \end{aligned}$$

Therefore, the competitive ratio of GOR for this family of problem instances is $O(1)$, which is a huge improvement over the competitive ratio $\frac{n}{m}-1$ as shown in the proof of Theorem 1 for algorithms with pre-determined scheduling order.

VI. Competitive Ratio Analysis

In this section, we first derive a competitive ratio for GOR as a function of the estimation factor η. We then find η that minimizes the competitive ratio.

A. Competitive Ratio for General η

We refer to the time to process the set of tasks $\mathcal{T}^{(l)}$ in iteration l as the *schedule length* of this iteration, denoted by $C^{(l)}_{max}$. In the following lemma we give a bound for $C^{(1)}_{max}$.

Lemma 1.

$$C^{(1)}_{max} \leq \left(\frac{2m+1}{m}\right)\eta C^*_{max}.$$

Proof. The proof is given in the Appendix. □

We note that, a task j scheduled in the second iteration of GOR should have been scheduled on some processor $i \in \mathcal{Q}$ in the first iteration and was cancelled as $u_j \geq \eta a_j$. Therefore, for task j scheduled in the second iteration we know some information about its processing time u_j. This insight forms the basis for deriving a bound for $C^{(2)}_{max}$, which is stated in the following lemma.

Lemma 2.

$$C^{(2)}_{max} \leq \left(1 + \frac{m}{\eta}\right) C^*_{max}$$

Proof. The proof is given in the Appendix. □

As a direct consequence of Lemmas 1 and 2, since $C_{max}(\mathbf{s}^{\text{GOR}}) = C^{(1)}_{max} + C^{(2)}_{max}$, in the following theorem we give a competitive ratio for GOR.

Theorem 2.

$$\frac{C_{max}(\mathbf{s}^{GOR})}{C^*_{max}} \leq f(\eta),$$

where

$$f(\eta) = 1 + \frac{(2m+1)}{m}\eta + \frac{m}{\eta}.$$

Remark: We note that the proofs of Lemmas 1 and 2 does not require the tasks to be ordered in the ascending order of their costs. Therefore, GOR without the sorting step will still have $O(\sqrt{m})$ competitive ratio. However, we observed that for problem instances generated randomly from typical distributions, sorting helps in reducing the total cost on average.

B. Minimizing the Competitive Ratio

One interesting feature of GOR is that the competitive ratio of the algorithm can be tuned by choosing an appropriate value for η. By using a large η, in the first iteration, we allow the tasks to run for a longer duration before cancellation and the worst case bound for $C^{(1)}_{max}$ increases. On the other hand, using a small η value results in aggressive cancellation of the tasks in the first iteration and the worst case bound for $C^{(2)}_{max}$ increases. Next, we find the optimal $\eta^* \geq 1$ that minimizes the competitive ratio $f(\eta)$.

Note that $f(\eta)$ is a convex function as its second derivative $f''(\eta) = \frac{2m}{\eta^3}$ is positive for $\eta \geq 1$. Therefore, we find $\eta*$ by equating the derivative $f'(\eta)$ to zero.

$$\begin{aligned} f'(\eta) &= \frac{2m+1}{m} - \frac{m}{\eta^2} = 0 \\ \Rightarrow \eta &= \frac{m}{\sqrt{2m+1}}. \end{aligned}$$

We note that the solution $\eta = \frac{m}{\sqrt{2m+1}}$ is undesirable for the case $m = 1$, since $\eta = \frac{1}{\sqrt{3}} < 1$. Therefore, for the case $m = 1$ we proceed as follows. Observe that for $m = 1$, $f(\eta) = \frac{3}{2} - \frac{1}{\eta^2}$ which is positive for $\eta \geq 1$. This implies $f(\eta)$ is a non-decreasing function for $m = 1$. Therefore, the minimum occurs at $\eta = 1$. We summarize the solution below.

$$\eta^* = \begin{cases} 1 & m = 1 \\ \frac{m}{\sqrt{2m+1}} & m \geq 2. \end{cases}$$

This leads to our main theorem below.

Theorem 3. *GOR is $O(\sqrt{m})$-competitive for $\mathcal{P}_{max}$ and $\mathcal{P}_{sum}$.*

Proof. Substituting $\eta = \frac{m}{\sqrt{2m+1}}$ for $m \geq 2$ in Theorem 2, we have

$$\frac{C_{max}(\mathbf{s}^{GOR})}{C^*_{max}} \leq 1 + 2\sqrt{2m+1}\ .$$

Therefore, GOR is $O(\sqrt{m})$-competitive for $\mathcal{P}_{max}$. It has the same competitive order for $\mathcal{P}_{sum}$ by Proposition 2. □

VII. Tight Competitive Ratio for $m = 1$

For $m = 1$, we have $\eta^* = 1$ and $f(1) = 5$. In the following theorem, we observe that an even lower competitive ratio of 4 can be proved. Further, the competitive ratio is tight.

Theorem 4. *For $m = 1$ and choosing $\eta = 1$, GOR is 4-competitive for $\mathcal{P}_{max}$. This competitive ratio is tight.*

Proof. The proof is given in the Appendix. □

The significance of Theorem 4 is the following. When $m = 1$, $\mathcal{P}_{max}$ is a semi-online version of the problem of minimizing the makespan on two unrelated parallel processors. For the offline version of the problem, where the processing times of the tasks on both the processors are known, the authors in [11] gave a $\frac{3}{2}$-approximation algorithm. Further, $\frac{3}{2}$ is the lower bound on the approximation ratio that any polynomial-time algorithm can achieve, unless P=NP [20]. On the other extreme, in the fully online version of the problem, where the processing times of the tasks on neither processors are known, an $O(\log n)$-competitive algorithm was given in [10]. To the best of our knowledge, GOR is the first constant-competitive algorithm to solve the semi-online version of the problem, where the processing times of the tasks on one processor are known and those on the other processor are unknown. We observe substantial improvement in the achievable competitive ratio when the processing times on one processor become available. Further, the competitive ratio 4 compares well with the lower bound $\frac{3}{2}$ in the offline case.

VIII. GOR for Dynamic Task Arrivals

So far we have assumed that all n tasks are available to be scheduled at time zero. In practice, the tasks may arrive dynamically in time, and their arrival times may not be known a priori. We consider $\mathcal{P}_{sum}$ and $\mathcal{P}_{max}$ under such dynamic task arrivals, and relabel them as $\mathcal{P}^r_{sum}$ and $\mathcal{P}^r_{max}$, respectively.

Given a θ-competitive algorithm for $\mathcal{P}_{max}$, we may adopt the general approach proposed in [10] to extend the algorithm to one that has a 2θ competitive ratio for $\mathcal{P}^r_{max}$. The extended GOR algorithm, termed Dynamic-GOR (DGOR), is described as follows. Without loss of generality, suppose there is at least one task available at time 0. Let $\mathcal{T}(0)$ be the set of tasks available at time 0. Schedule $\mathcal{T}(0)$ using GOR. Accumulate the tasks that arrive while waiting for the time when the last task scheduled on processors 1 to m from $\mathcal{T}(0)$ is finished. Then schedule the accumulated tasks using GOR. Again, accumulate tasks and repeat the above procedure until no more tasks are available to be scheduled.

Furthermore, we note that Proposition 2 holds for problems $\mathcal{P}^r_{sum}$ and $\mathcal{P}^r_{max}$ as well. Therefore, such an extended algorithm has a 4θ competitive ratio for $\mathcal{P}^r_{sum}$. Since GOR is $f(\eta)$-competitive for $\mathcal{P}_{max}$, DGOR is $2f(\eta)$-competitive for $\mathcal{P}^r_{max}$ and $4f(\eta)$-competitive for $\mathcal{P}^r_{sum}$. Therefore, it is $O(\sqrt{m})$-competitive for both $\mathcal{P}^r_{max}$ and $\mathcal{P}^r_{sum}$.

IX. Average Performance of GOR

In addition to the proven competitive ratios presented in the previous sections, we next study the average performance of GOR and DGOR over randomly generated problem instances. We provide comparison with the celebrated list scheduling algorithm [12]. Further, in order to establish a benchmark, we also present results for the two algorithms from [5] as described in Section II, which require a priori task processing times.

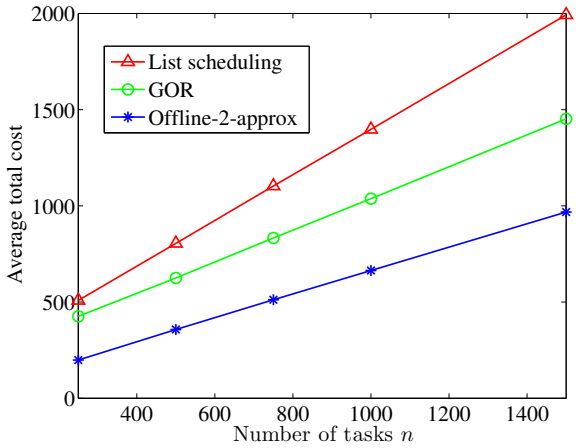

Fig. 3. Effect of varying number of tasks. All tasks at time zero.

A. All Tasks Available at Time Zero

In the following simulation results, both the processing times u_j and offloading cost a_j are chosen from an exponential distribution, with default means 60 sec and $\frac{60}{m}$ sec, respectively. Note that, even though we have chosen parameter values from an exponential distribution, similar results are observed when other distributions are used. The other default parameters are as follows: number of tasks $n = 1000$, number of processors $m = 50$, and weight factor $w = 1$. We generate over 5000 problem instances for each data point and compare the average total cost achieved by different algorithms.

Figures 3, 4, 5, and 6 present the average total cost with varying number of tasks, mean processing time, number of processors, and weight factor, respectively. We label the $(2-\frac{1}{m})$-approximation offline algorithm from [5] as "Offline-2-approx." It can be observed that GOR outperforms list scheduling and provides a reduction of $30-40\%$ in the average total cost. Another important observation is that, as the number of tasks increases, the gap between GOR and list scheduling increases. This suggests that GOR will perform far better than list scheduling in an enterprise where the number of tasks is large.

At the same time, we observe that GOR incurs about 50% higher cost than the best known offline algorithm. It is impractical to know the exact processing time of tasks before they are processed. However, this result suggests an interesting future research direction on how to use some available statistical information about the task processing times to further improve the average performance of GOR.

B. Dynamic Task Arrivals

We next simulate dynamic task arrivals and observe the average performance of DGOR. We compare it with list scheduling and the RTP algorithm in [5]. The tasks are generated with inter arrival times chosen from an exponential distribution with mean 100 ms. All the other parameter values are the same as described in the previous subsection. We simulate for over 10^5 task arrivals for each data point.

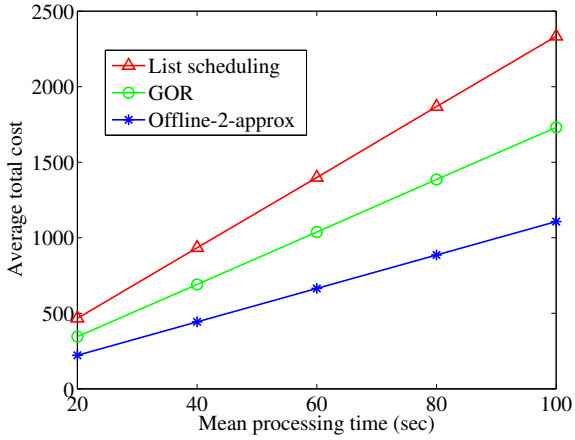

Fig. 4. Effect of varying mean processing time. All tasks at time zero.

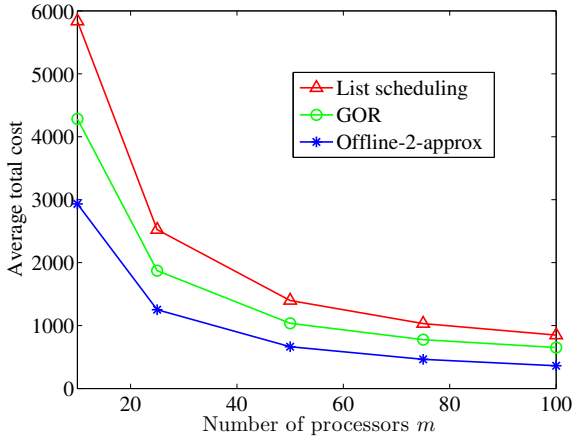

Fig. 5. Effect of varying number of processors. All tasks at time zero.

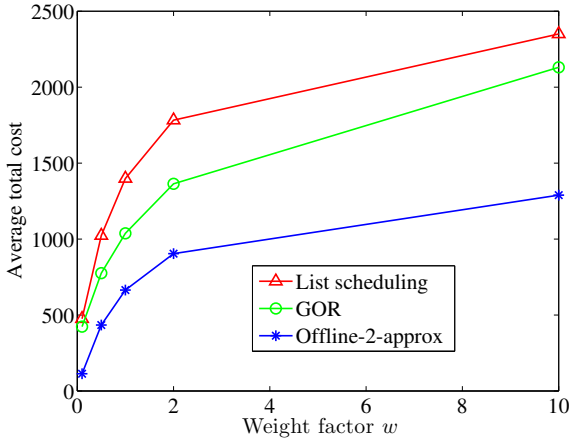

Fig. 6. Effect of varying weight factor. All tasks at time zero.

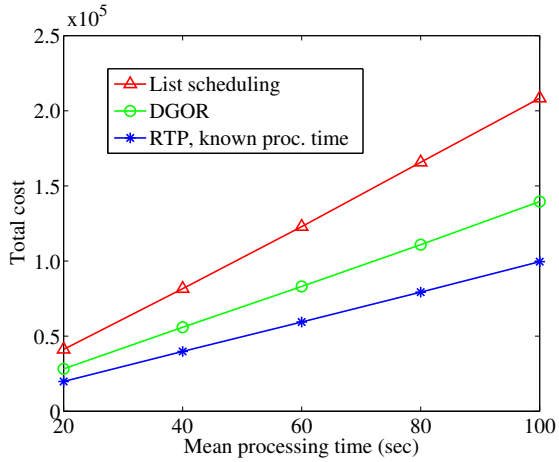

Fig. 7. Effect of varying mean processing time. Dynamic task arrival.

In Figures 7, 8, and 9, we present the total cost of the algorithms with varying mean processing time, number of processors, and weight factor, respectively. We observe that DGOR provides a reduction of $50-90\%$ in the total cost compared with list scheduling. Furthermore, its performance is competitive with the RTP algorithm, even though RTP requires that task processing times are known a priori.

X. Conclusion

We have studied joint scheduling and offloading in a hybrid cloud. We have formulated an optimization problem to minimize the weighted sum of the makespan on m identical processors and the offloading cost to the public cloud. We propose a GOR algorithm to solve this problem under the challenging yet practical semi-online setting where the task processing times are not known a priori. We show that GOR is $O(\sqrt{m})$-competitive, which is a significant improvement over previously known algorithms.

Further analytical improvement indicates that GOR has a tight competitive ratio of 4 for minimizing the makespan on two unrelated processors. We have also extended GOR to DGOR to accommodate dynamic task arrivals. We show that DGOR has the same $O(\sqrt{m})$ scaling of competitive ratio. Our simulation results further demonstrate that both GOR and DGOR provide substantial savings in average performance over list scheduling, and they are competitive with algorithms where the task processing times are assumed to be known a priori.

XI. Acknowledgement

This work has been supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada.

XII. Appendix

A. Proof of Proposition 2

Let $\mathbf{s}'$ be the computed schedule of a θ-competitive algorithm for solving $\mathcal{P}_{max}$. We have the following inequalities.

$$\begin{aligned}\Upsilon(\mathbf{s}') &= \max_{i\in\mathcal{Q}}\{C_i(\mathbf{s}')\} + w\Gamma(\mathbf{s}') \\ &\le 2\max\{\max_{i\in\mathcal{Q}}\{C_i(\mathbf{s}')\}, w\Gamma(\mathbf{s}')\} \\ &\le 2\theta\max\{\max_{i\in\mathcal{Q}}\{C_i(\mathbf{s}^*)\}, w\Gamma(\mathbf{s}^*)\}\end{aligned}$$

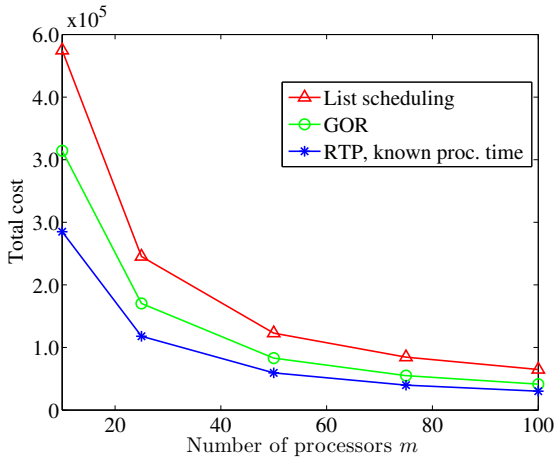

Fig. 8. Effect of varying number of processors. Dynamic task arrival.

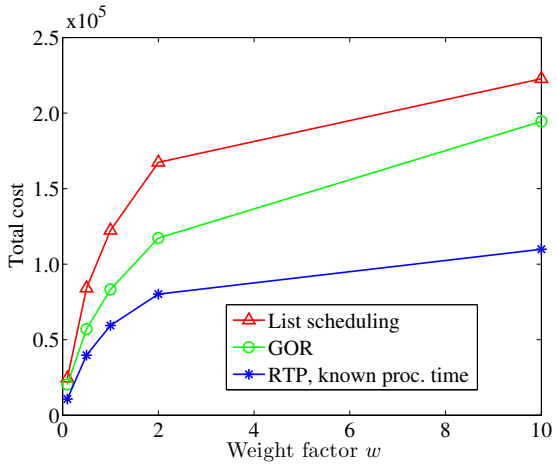

Fig. 9. Effect of varying weight factor. Dynamic task arrival.

$$\begin{aligned}
&\leq 2\theta \max\{\max_{i\in\mathcal{Q}}\{C_i(\bar{\mathbf{s}}^*)\}, w\Gamma(\bar{\mathbf{s}}^*)\} \\
&\leq 2\theta[\max_{i\in\mathcal{Q}}\{C_i(\bar{\mathbf{s}}^*)\} + w\Gamma(\bar{\mathbf{s}}^*)] \\
&= 2\theta\Upsilon(\bar{\mathbf{s}}^*).
\end{aligned}$$

Hence the result.

B. Proof of Lemma 1

Let $C_i^{(l)}$ and $\mathcal{T}_i^{(l)}$ denote the schedule length and the set of tasks scheduled, respectively, on processor i in iteration l. Let $\gamma_j = \min(u_j, \eta a_j)$. Noting that $\eta \geq 1$, we have

$$\begin{aligned}
C^*_{max} &\geq \frac{1}{m+1}\sum_{j=1}^{n}\min(u_j, a_j) \\
&= \frac{1}{\eta(m+1)}\sum_{j=1}^{n}\min(\eta u_j, \eta a_j) \\
&\geq \frac{1}{\eta(m+1)}\sum_{j=1}^{n}\gamma_j. \qquad (1)
\end{aligned}$$

Also $\forall j$,

$$\begin{aligned}
C^*_{max} &\geq \min(u_j, a_j) \\
&\geq \frac{1}{\eta}\min(\eta u_j, \eta a_j) \\
&\geq \frac{1}{\eta}\gamma_j. \qquad (2)
\end{aligned}$$

We also note that, in the first iteration of GOR, a task j scheduled on processor $i \in \mathcal{Q}$ is processed for γ_j duration. We now consider the following cases.

Case 1: $C^{(1)}_{max} = C^{(1)}_{m+1}$. For this case, from Line 27 of Algorithm 1, task $q^{(1)}$ should have been scheduled on some processor $\hat{i} \in \mathcal{Q}$, but its processing was completed first on processor $m+1$. In other words, processing $q^{(l)}$ on processor $\hat{i}$ to completion would have increased the schedule length. Therefore, we have

$$C^{(1)}_{max} \leq \sum_{j\in\mathcal{T}^{(1)}_{\hat{i}}}\gamma_j + \gamma_{q^{(1)}}. \qquad (3)$$

Also, at time $C^{(1)}_{max} - \gamma_{q^{(l)}}$, all the processors $i \in \mathcal{Q}\backslash\{\hat{i}\}$ should have been busy executing some task, otherwise GOR would have scheduled task $q^{(l)}$ on processor $i \in \mathcal{Q}\backslash\{\hat{i}\}$ which is idle at that time. Therefore,

$$C^{(1)}_{max} - \gamma_{q^{(l)}} \leq \sum_{j\in\mathcal{T}^{(1)}_{i}}\gamma_j + \gamma_{q^{(1)}},\ \forall i \in \mathcal{Q}\backslash\{\hat{i}\}. \qquad (4)$$

From (3) and (4) we have

$$\begin{aligned}
&C^{(1)}_{max} - \gamma_{q^{(1)}} \leq \sum_{j\in\mathcal{T}^{(1)}_{i}}\gamma_j,\ \forall i \in \mathcal{Q} \qquad (5) \\
\Rightarrow\ &C^{(1)}_{max} - \gamma_{q^{(1)}} \leq \frac{1}{m}\sum_{j\in\cup_{i\in\mathcal{Q}}\mathcal{T}^{(1)}_{i}}\gamma_j \\
\Rightarrow\ &C^{(1)}_{max} \leq \frac{\eta(m+1)}{m}C^*_{max} + \gamma_{q^{(1)}} \\
\Rightarrow\ &C^{(1)}_{max} \leq \left(\frac{2m+1}{m}\right)\eta C^*_{max}.
\end{aligned}$$

In the third inequality above, we have used (1) and in the last inequality we have used (2).

Case 2: $C^{(1)}_{max} \neq C^{(1)}_{m+1}$. Let $C^{(1)}_{max} = C^{(1)}_{\hat{i}}$ for some $\hat{i} \in \mathcal{Q}$. In this case task $q^{(1)}$ should have been scheduled both on processor $\hat{i}$ and processor $m+1$ but either was cancelled or was completed processing on processor $\hat{i}$ first. Again, at time $C^{(1)}_{max} - \gamma_{q^{(1)}}$ any processor $i \in \mathcal{Q}\backslash\{\hat{i}\}$ should be busy executing some task. Otherwise, scheduling task $q^{(1)}$ on some processor $i \in \mathcal{Q}\backslash\{\hat{i}\}$ that is idle by that time will result in a smaller schedule length and, GOR would have done so. Therefore,

$$C^{(1)}_{max} - \gamma_{q^{(1)}} \leq \sum_{j\in\mathcal{T}^{(1)}_{i}}\gamma_j,\ \forall i \in \mathcal{Q}.$$

The result follows as the above inequality is the same as (5).

C. Proof of Lemma 2

Let $C^{(2)*}_{max}$ denote the optimal schedule length of the tasks from $\mathcal{T}^{(2)}$ with the assumption that the processing time of task $j \in \mathcal{T}^{(2)}$ on processor $i \in \mathcal{Q}$ is ηa_j and on processor $m+1$ is a_j. Also, let $C^{(2)*}_{i,max}$ denote the corresponding schedule length on processor i. We have $u_j > \eta a_j, \forall j \in \mathcal{T}^{(2)}$. Therefore, $C^{(2)*}_{max} \leq C^*_{max}$. We know that

$$\frac{1}{\eta}\sum_{i=1}^{m} C^{(2)*}_{i,max} + C^{(2)*}_{m+1,max} = \sum_{j=1}^{|\mathcal{T}^{(2)}|} a_j$$

$$\Rightarrow \left\{1 + \frac{m}{\eta}\right\} C^{(2)*}_{max} \geq \sum_{j=1}^{|\mathcal{T}^{(2)}|} a_j$$

$$\Rightarrow \left\{1 + \frac{m}{\eta}\right\} C^{*}_{max} \geq \sum_{j=1}^{|\mathcal{T}^{(2)}|} a_j. \quad (6)$$

Now, $C^{(2)}_{max}$ cannot be greater than the schedule length of tasks from $\mathcal{T}^{(2)}$ when all of them are processed on processor $m+1$. This implies $C^{(2)}_{max} \leq \sum_{j=1}^{|\mathcal{T}^{(2)}|} a_j$. Therefore, the result follows from (6).

D. Proof of Theorem 4

Since $\eta = 1$, in the first iteration the processing time of task j scheduled on processor 1 is $\gamma_j = \min(u_j, a_j)$.

$$C^{(1)}_{max} \leq \sum_{j=1}^{n} \min(u_j, a_j),$$

since the schedule given by GOR in first iteration is always better than scheduling all tasks on processor 1. Substituting $m = 1$ and $\eta = 1$ in (1) we get

$$C^{*}_{max} \geq \frac{1}{2}\sum_{j=1}^{n} \min(u_j, a_j).$$

Therefore, $C^{(1)}_{max} \leq 2C^*_{max}$. Substituting $m = 1$ and $\eta = 1$ in Lemma 2, we get $C^{(2)}_{max} \leq 2C^*_{max}$. Therefore, $C_{max} = C^{(1)}_{max} + C^{(2)}_{max} \leq 4C^*_{max}$.

To show that the competitive ratio is tight we provide the following problem instance for which the competitive ratio is achieved by GOR. Consider $n = 8$ and the task processing times are as follows:

$$a_j = \begin{cases} 10 - \delta & j = 1, 2, 3, 4 \\ 10 & j = 5, 6, 7, 8, \end{cases}$$

$$u_j = \begin{cases} \delta & j = 1, 2, 3, 4 \\ 40 & j = 5 \\ 10 + \delta & j = 6, 7, 8, \end{cases}$$

where δ is a positive real number close to 0. GOR lists the tasks in the ascending order of a_j. The tasks 1 through 4 are scheduled on processor 2 and tasks 5 through 8 are scheduled on processor 1 in the first iteration. Since $u_j > a_j, \forall j \in \{5, 6, 7, 8\}$, all the tasks scheduled on processor 1 will be cancelled. Therefore, the schedule length in the first iteration is $40 - 4\delta$. Tasks 5 through 8 have same a_j. Therefore, in the second iteration GOR cannot differentiate the tasks and may schedule task 5 on processor 1 and tasks 6 through 8 on processor 2. In this case the schedule length in the second iteration is 40. This results in a makespan $C_{max}(\mathbf{s}^{\text{GOR}}) = 80 - 4\delta$.

The optimal schedule $\mathbf{s}^*$ is the following. Schedule tasks $1, 2, 3, 4, 6, 7$ on processor 1 and tasks $5, 8$ on processor 2. The optimal maskespan is $C^*_{max} = 20 + 6\delta$. Since δ can be chosen arbitrarily close to 0, the competitive ratio 4 is achieved.

REFERENCES

[1] T. Guo, U. Sharma, P. Shenoy, T. Wood, and S. Sahu, "Cost-aware cloud bursting for enterprise applications," *ACM Trans. Internet Tech.*, vol. 13, no. 3, pp. 10:1–10:24, May 2014.

[2] IBM, http://www.ibm.com/cloud-computing/us/en/private-cloud.html.

[3] VMware, http://www.vmware.com/ca/en/cloud-computing/hybrid-cloud.

[4] D. Shabtay, N. Gaspar, and M. Kaspi, "A survey on offline scheduling with rejection," *J. Scheduling*, vol. 16, no. 1, pp. 3–28, 2013.

[5] Y. Bartal, S. Leonardi, A. Marchetti-Spaccamela, J. Sgall, and L. Stougie, "Multiprocessor scheduling with rejection," *SIAM J. Discrete Math.*, vol. 13, no. 1, pp. 64–78, 2000.

[6] C. Miao and Y. Zhang, "On-line scheduling with rejection on identical parallel machines," *J. Systems Science & Complexity*, vol. 19, no. 3, pp. 431–435, 2006.

[7] X. Min, Y. Wang, J. Liu, and M. Jiang, "Semi-online scheduling on two identical machines with rejection," *J. Comb. Optim.*, vol. 26, no. 3, pp. 472–479, 2013.

[8] A. Fiat and G. Woeginger, Eds., *Online Algorithms: the State of the Art*, ser. Lecture notes in computer science. New York: Springer, 1998.

[9] M. Drozdowski, *Scheduling for Parallel Processing*. Springer Publishing Company, 2009.

[10] D. B. Shmoys, J. Wein, and D. P. Williamson, "Scheduling parallel machines on-line," *SIAM J. Comput.*, vol. 24, no. 6, pp. 1313–1331, Dec. 1995.

[11] C. N. Potts, "Analysis of a linear programming heuristic for scheduling unrelated parallel machines," *Discrete Applied Mathematics*, vol. 10, no. 2, pp. 155–164, 1985.

[12] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, pp. 416–429, 1969.

[13] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, "Optimization and approximation in deterministic sequencing and scheduling: a survey," *Annals of discrete mathematics*, vol. 5, no. 2, pp. 287–326, 1979.

[14] D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms*, 1st ed. New York, NY, USA: Cambridge University Press, 2011.

[15] X. Qiu, W. L. Yeow, C. Wu, and F. C. M. Lau, "Cost-minimizing preemptive scheduling of mapreduce workloads on hybrid clouds," in *Proc. IEEE IWQoS*, 2013, pp. 213–313.

[16] S. Li, Y. Zhou, L. Jiao, X. Yan, X. Wang, and M. R. Lyu, "Delay-aware cost optimization for dynamic resource provisioning in hybrid clouds," in *Proc. IEEE International Conference on Web Services, ICWS*, 2014, pp. 169–176.

[17] R. Van Den Bossche, K. Vanmechelen, and J. Broeckhove, "Online cost-efficient scheduling of deadline-constrained workloads on hybrid clouds," *Future Gener. Comput. Syst.*, vol. 29, no. 4, pp. 973–985, Jun. 2013.

[18] M. Rahman, X. Li, and H. N. Palit, "Hybrid heuristic for scheduling data analytics workflow applications in hybrid cloud environment," in *Proc. IEEE IPDPS Workshops*, 2011, pp. 966–974.

[19] M. Shifrin, R. Atar, and I. Cidon, "Optimal scheduling in the hybrid-cloud," in *Proc. IFIP/IEEE International Symposium on Integrated Network Management*, 2013, pp. 51–59.

[20] J. K. Lenstra, D. B. Shmoys, and E. Tardos, "Approximation algorithms for scheduling unrelated parallel machines," *Math. Program.*, vol. 46, no. 3, pp. 259–271, Feb. 1990.