On efficient resource use for scientific workflows in clouds

Khaled Almi’ani\textsuperscript{a,b,*}, Young Choon Lee\textsuperscript{c}, Bernard Mans\textsuperscript{c}

\textsuperscript{a}Al-Hussein Bin Talal University, M\textsuperscript{a}t\textsuperscript{a}in, Jordan
\textsuperscript{b}Princess Sumaya University for Technology, Amman, Jordan
\textsuperscript{c}Macquarie University, Sydney, NSW, Australia

A R T I C L E   I N F O

Article history:
Received 1 June 2018
Revised 3 September 2018
Accepted 4 October 2018
Available online 9 October 2018

Keywords:
Cloud computing
Fair share
Scheduling
Max-min fair share algorithm

A B S T R A C T

The abundance of cloud resources has enabled not only web applications, but also scientific applications to easily scale to meet their objectives, such as performance and costs. However, due to the complex and large-scale nature of scientific workflows, the decision on such scaling (resource management) is much complicated often resulting in inefficient use of resources. In this paper, we present RDAS+ as a resource demand aware scheduling algorithm to optimize resource efficiency for the execution of scientific workflows in clouds. RDAS+ maximizes resource utilization by allocating the minimum number of resources (virtual machines or VMs in clouds) with little sacrifice of completion time (makespan). This optimization eventually leads to cost efficiency for pay-per-use cloud resources. RDAS+ consists of partitioning, resource allocation and task scheduling steps to realize such optimization. We have evaluated RDAS+ using five types of real-world scientific workflows in comparison with three existing algorithms. Our experimental results confirm our claims on achieving resource efficiency. In particular, the average rate of cost savings (32\%) outweighs makespan increase (11\%). Although these two performance metrics are incompatible, the trade-off RDAS+ optimizes shows significant benefit particularly in clouds.

© 2018 Elsevier B.V. All rights reserved.

1. Introduction

Due to their abundant resources that can be elastically provisioned with pay-as-you-go pricing model, clouds have emerged as cost-efficient platforms for many applications including large-scale science and engineering applications [1]. Many of these applications consist of a number of processes/tasks forming a workflow (Directed-Acyclic Graph or DAG, see Fig. 1). The tasks are connected by directed edges that represent the data dependency between tasks (precedence constraints). Each task performs its computation on either initial data submitted with the workflow or data submitted to it by parent tasks. Typically, tasks of a given workflow are scheduled and executed in a distributed manner across multiple processing elements without violating precedence constraints.

The scheduling of these workflows in clouds concerns pay-per-use resources (virtual machines or VMs), which are rented from cloud providers such as Amazon EC2 and Google Cloud. One of the major challenges in executing scientific workflows on clouds is to determine the “right” number of cloud resources to rent in terms of both cost and performance. Over-renting results in increasing the cost by having unnecessary VMs, while under-renting results in degrading the performance. Furthermore, heterogeneity in both resources and applications increases the complexity of this scheduling problem. For instance, Amazon EC2 and Google Cloud Platform provide several types of VMs with different capacities and capabilities, and tasks of scientific workflows often have different resource requirements in terms of both computation and communication capacity.

This scheduling problem and its variations have been extensively studied, e.g., [2–8]. A majority of previous studies focus on either the minimization of makespan or the minimization of cost. In many cases, studies with the former objective implicitly aim to also reduce cost. However, the minimization of makespan often results from the use of more VMs that makes little cost reduction or even increase of cost. Besides, VMs in clouds are often charged based on the number of machine hours; hence, unless the reduction of makespan affects a fewer number of machine hours, the actual cost remains unchanged. In the meantime, studies with the latter cost optimization often assume no strict makespan constraint resulting in less attractive schedule in terms of makespan. Although there have been some works on the optimization of both makespan and cost, they tend to focus on one objective and making the other one as some form of constraint.

In this paper, we address the problem of workflow scheduling in clouds with the objective of optimizing resource efficiency. In particular, we aim to minimize cost and makespan by improving
resource utilization, without enforcing any constraints on cost and makespan. To this end, we present the Resource Demand Aware Scheduling plus (RDAS+) algorithm consisting of three steps: partitioning, resource allocation and task scheduling. For a given workflow, RDAS+ aims to maximize resource utilization with the minimal increase of makespan. The former objective is handled by partitioning and resource allocation steps while the latter objective is dealt with by the task scheduling step. In particular, the maximization of resource utilization is achieved by determining the minimum number of resources and reducing resource idle time between tasks executions. Makespan increase is minimized by ensuring the actual start time of task is as close as the earliest start time.

This work significantly extends our earlier and preliminary work [9]. RDAS+ ensures resource allocation to a group of tasks (task partition or simply partition) to be envy-free and Pareto-efficient. For a given partition, the number of resources allocated is referred to as utility. An allocation is envy-free if any partition cannot achieve better utility using any other partition allocation. A Pareto-efficient allocation means that the utility of any partition cannot be increased without reducing the utility of some other partition(s). To increase the utility of any partition, we need to allocate more resources to this partition. Thus, if we can remove a resource from a partition retaining its utility, this partition is over-utilizing resources. We establish such an allocation, by employing the well-known weighted version of the max-min fairness allocation policy [10]. This policy works by maximizing the minimum allocation received by any partition. By establishing envy-free Pareto-efficient allocation, we aim to reduce makespan and avoid using unnecessary resources to reduce cost.

Our experiments with five types of real-world scientific workflow show that RDAS+ achieves efficient schedules using significantly fewer numbers of resources compared to three existing algorithms, PBWS [11], IC-PCP [12] and HEFT [8]. Our experimental results confirm our claims on achieving resource efficiency. In particular, the average rate of cost savings (32%) outweighs makespan increase (11%); this is achieved with an average reduction of 39% of idle time and that of 17% of resource count (#VMs).

The rest of the paper is organized as follows. In Section 2, we discuss the related work. The problem is formally described in Section 3. The Resource Demand Aware Scheduling plus algorithm is presented in Section 4. Section 5 presents experiment results, and the paper is concluded in Section 6.

2. Related work

The problem of workflow scheduling has been extensively studied in the literature. Many of previous studies have attempted to minimize either cost or execution time (e.g., [2,4,12]) while there are several works [11,13,14] that concern the optimization of both criteria. Wu et al. [2] proposed a greedy approach that iteratively reduces the cost of the initial schedule by reallocating tasks. Similarly, Arabnejad et al. [4] proposed the HBCS algorithm, which improves the obtained schedule by allocating tasks based on the left budget. Sakellariou et al. [15] present two scheduling heuristics, which aim to satisfy the budget constraint by iteratively modifying the schedule to adjust its cost and to minimize the execution time by adjusting/increasing the cost of schedule (budget), respectively. Our work differs from these works in that the cost is aimed to be minimized by improving resource efficiency, without the introduction of any cost/budget constraint.

Several studies [12,16–19] have addressed the problem of workflow scheduling with the aim of cost minimization under the presence of a deadline constraint. Abrishami et al. [12] proposed IC-PCP, which aims to minimize cost within a pre-determined deadline. It begins by assigning the exit task for which the latest finish time is set to be the deadline. Then, it determines the partial critical path for the exit task. Once the deadline for each task in this partial critical path is determined, the IC-PCP algorithm proceeds to determine the cheapest resources that can execute these tasks on time. The algorithm repeats to determine the partial critical path for each assigned task until all tasks are assigned to resources. Malawski et al. [18] model this problem as a Mixed Integer Program, and solve it for small instances using a mathematical programming language. Metaheuristics including genetic algorithms (GAs) [17] and particle swarm optimization [16,19] are also a popular stream of approach. These previous works remain single-objective optimization techniques. The introduction of the deadline and cost constraints reduces the solution space and therefore limits the optimization to one of the objectives.

The bi-criteria scheduling problem addressed by Durillo et al. [13], Prodan et al. [14] and our previous work in [11] shares some similarities with the problem addressed in this paper. Durillo et al. [13] propose a Pareto-based approach, which presents the user with a number of solutions. The number of solutions that this approach generates is pre-determined by the user. Then, the user manually selects the best solution that meets their objective. This method is an extension of the well-known HEFT algorithm [8]. The
main factor in determining this approach’s efficacy is the number of solutions. Prodan et al. [14], proposed the DCA algorithm that requires the user to specify their interest in the objectives, in other words, the user has to specify the primary and secondary objectives (time, cost). This algorithm consists of two phases of scheduling, the primary and secondary scheduling phases for optimizing each of the two goals in turn.

Almi’ani et al. [11] present the PBWS algorithm, which enables to trade between two objectives of cost minimization and makespan minimization. The algorithm uses a slack parameter $\beta$ to determine the priority ratio between two objectives, i.e., $\beta \in [0, 1]$. The lower the $\beta$ value, the higher the priority on makespan. The cost minimization is prioritized when $\beta$ values are close to 1.

Although works in [11,13,14] are closely related to our RDAS+ algorithm, they tend to prioritize the minimization of makespan. More importantly, the fair share allocation in this paper is what makes RDAS+ different from these works in the sense that cost and execution time minimization are achieved by improving resource utilization with the use of fewer resources.

3. Problem statement

An input of our problem consists of workflow $(G)$ and resources sets $(R)$. A workflow is a directed acyclic graph (DAG) $G = \langle V, E, >$, where $V$ is a set of tasks, and $E$ is a set of edges. Each edge connects two tasks, and the direction of each edge represents the data dependency (precedence constraints). Additionally, we are provided with multiple resource sets $R = R_1, R_2, \ldots, R_m$. Capacity is the main distinguisher among the resource sets. Each set of resources $R_i \in R$ contains a limited number of identical resources in terms of computation and communication capacities (memory and storage space). We order $R$ based on the capacity of the resources, and thus resources belonging to set $R_i$ have higher capacity than resources belonging to set $R_j (j > i)$. Each processor core is responsible for executing one or more task(s) in a sequential order, and each task is only assigned to a single processor core. We assume individual tasks are sequential without task-level parallelism as in workflow applications used in this study. A task $v_i$ must receive all the data generated by its parents (or predecessors), before it starts its execution. For each task, the last parent that delivers its data is termed the Most Influential Parent (MIP). We denote the communication cost between any two tasks as $v_i$ and $v_j$ by $c(i, j)$. The completion time of a workflow is defined as makespan or schedule length.

TheEarliest Start Time (EST) and Earliest Finish Time (EFT) for task $v_i \in V$ is defined as:

\[
EST(v_i) = \begin{cases} 
0 & v_i \text{ is entry task} \\
EFT(MIP(v_i)) + c(MIP(v_i), i) & \text{otherwise} 
\end{cases}
\]

\[
EFT(v_i) = EST(v_i) + c_i
\]

where $c_i$ refers to the execution time for task $v_i$. The actual start time (AST) and actual finish time (AFT) can be different from EST and EFT, and mainly depend on the available number of processor cores. The execution time of tasks that belong to the critical path (CP) in the workflow has a major influence on the makespan of the obtained schedule. This path starts at one of the entry tasks and ends at the exit task, where this path has the highest cost (communication and computation). The exit task is the last task to be executed in the workflow. The execution time of a task $v_i \in V$ is decided based on the capacity of the processor core allocated to that task. Allocating a processor core with high capacity to a task results in low execution time compared to allocating a processor core with low capacity. Without loss of generality, we assume that resources of type $R_i \in R$ $(i > 1)$ are $i$ times faster and more expensive compared to $R_1 \in R$.

4. The resource demand aware scheduling plus algorithm

RDAS+ consists of three steps: partitioning (Algorithm 1), resource allocation (Algorithms 2 and 3) and task scheduling.
(Algorithm 4). In particular, we begin by giving an overview of RDAS+ and describe those three steps in detail.

4.1. Overview

The RDAS+ algorithm starts by partitioning the workflow into a number of task groups (workflow partitions or simply partitions) considering the structure and size of workflow. We treat each partition as an execution agent. The main idea of this step is to transform the problem of partitioning the workflow into the well-known bin-packing problem [20]. In this context, partitions represent bins and tasks represent items to be packed to bins. The objective of the partitioning step is to determine the minimum number of partitions (bins) required to host tasks (items).

The resource allocation step then divides VMs between partitions. For each partition, we calculate demand and weight: demand is the number of resources it requires to achieve the lowest possible execution time, and weight is a priority value. In particular, weight values are calculated based on dependency relationships (dictated by data dependencies) between partitions. We use weight to determine each partition’s share of resources (utility).

In the task scheduling step, we schedule tasks on VMs (determined in the resource allocation step) in a greedy manner such that the actual finish time (AFT) of task is minimized. For a given partition, its tasks are sorted by EST in ascending order and scheduled in that order. Note that AFT can be different from the earliest finish time (ETF) if AFT of another task scheduled on the same VM is later than ETF of the task being scheduled.

4.2. Partitioning

The partitioning step (Algorithm 1) starts by determining the maximum number of partitions (P) considering the number of tasks per partition and the structure of workflow; hence, partitions are balanced in terms of their weights and demands. In particular, for a given workflow, we traverse CP until we find the first task that has more than one predecessor tasks (line 6). We refer to this task as the bottleneck task. Partitions are to be rooted at this bottleneck level since the location of this task and its execution time impact the makespan significantly. Now, considering the number of VMs available from resource set R1 and the number of predecessor tasks of the bottleneck task (bp), we set the maximum number of partitions (mp) to |R1| (line 8) if |R1| < |bp|, otherwise |bp| (line 9).

The capacity (C) for each partition is determined by (1) the execution time of CP (CP length or |CP|), (2) the total execution time of all tasks in a workflow (τw), and (3) the maximum number of partitions (mp). If the product of CP length and mp is greater than τw (i.e., |CP| × mp > τw), we use |CP| as the capacity for each partition (line 12). Otherwise, we set the capacity to a fraction of mp (τw/mp, line 13).

We now assign tasks to partitions in a way that the number of partitions is minimized and the total execution time of tasks assigned to each partition is within the capacity, C. This assignment problem can be reduced to the well-known bin-packing problem [20]. We employ the well-known First-Fit strategy [20]. The First-Fit and Best-Fit decreasing strategy guaranteed an absolute performance ratio of 1.5 which is the best absolute performance ratio for the bin-packing problem, unless P=NP [21].

CP tasks are assigned to the first partition as a pre-processing step (line 17). Then, we order tasks in bp based on EST (line 16). In each iteration, we identify task pt ∈ bp with the smallest EST and all of its predecessor tasks. Each predecessor task must be connected with pt through a valid path. We refer to these identified tasks as U (line 19). Then, starting from the task located at the highest level in U, we assign each task to the first partition, such that this assignment does not violate the capacity constraint for this partition (if statement, line 24). Once this assignment is performed, we update the capacity of the partition by subtracting the execution time of the assigned task from this partition capacity. Once a task is assigned to a partition, it will be removed from consideration in subsequent iterations. The process of partitioning stops when all tasks are assigned to partitions.

To clarify the partitioning step, consider the example shown in Fig. 2. For simplicity, assume that the execution cost of each task is 10 m, and the communication cost is negligible and the number of available resources from resource set type R1 is 10. CP tasks are t2, t7, t10, t15, t19, t20 and t21 and CP length is 70, |CP| = 70. The maximum number of partitions is 4, mp = 4 and the total execution time of all tasks is 210 min. We first assign CP tasks to the first partition p1. This partition will not accept any more tasks since its capacity after this addition is zero. Next, we order tasks t15, t17 and t18 based on EST. The ordered list becomes t17, t18 and t15. We assign task t17 and all of its predecessor tasks (t13, t14, t19, t10 and t12) to partition p2 since the sum of their execution times is within the capacity of this partition. The rest of tasks (t3, t4, t5, t6, t7, t11, t16, and t1) are assigned to partition p3. Note that only the first three partitions are actually used while the maximum number of partitions was initially four.

4.3. Resource allocation

Decisions on allocating VMs to partitions are made based primarily on the demand and weight of partition (Algorithms 2 and 3). As it is often the case that the number of VMs is limited and possibly smaller than the number of tasks (e.g., the initial resource limit, 20 instances, in Amazon EC2 or the number of resources in a private cloud), the fair allocation of resources between partitions is a key decision criterion. For a particular partition, the fairness in the context of our resource allocation refers to the amount of resources proportional to its demand rather than simply an equal amount of resources among all partitions.

In this step, our objective is to divide R1 resources between the partitions such that the resultant allocation is envy-free Pareto-efficient. Other VMs with higher capacity are used as a replacement for R1 resource. An allocation is envy-free (equitable) if and only if no partition prefers any other partition share of resource to its own [22]. In other words, no partition can achieve higher utility if it is given any other partition share of resources. An allocation is Pareto-efficient if the resources cannot be divided between the tasks in any other way such that at least one partition can achieve better utility, without reducing the utility of any other partition [22]. Given n number of resources (R1), the utility of a partition (Pi) is defined as

$$u(P_i, n) = \frac{n}{W_i}, n \leq d_i$$

where \(w_i\) and \(d_i\) are the weight and demand of partition \(P_i\) respectively. The weight of partition \(P_i\) is defined as the number of partitions that depend on the output data of \(P_i\). The demand of a partition refers to the number of VMs required by the partition. It is determined based on the maximum number of tasks in the partition that can run in (partially) parallel.

Algorithm 2 shows detailed steps of resource allocation. We start by calculating the demand of the partitions (d) (line 5). Then, we determine the weight for the partitions (w) (the for statement in line 6). We normalize the weight of the partitions, to achieve
Algorithm 2. Resource Allocation.

1: procedure RAlocation(P, R)
2: \hspace{1em}▷ Input: P(Paritions), R (Resources sets)
3: \hspace{1em}⇒ output:S = s_1, ..., s_m
4: \hspace{1em}⇒ s_i, VMs allocated to partition P_i
5: \hspace{1em}d ← calculate partitions demand
6: \hspace{1em}for P_i ∈ P do
7: \hspace{2em}w_i ← # of partitions depend on P_i
8: \hspace{1em}end for
9: \hspace{1em}w ← normalize w
10: \hspace{1em}a_1 ← number of VMs from R_1
11: \hspace{1em}round ← \sum_{i=1}^{P_i} w_i
12: \hspace{1em}while a_1 ≥ round and round > 0 do
13: \hspace{2em}for P_i ∈ P do
14: \hspace{3em}r_i ← P_i allocated VMs
15: \hspace{3em}if r_i < d_i then
16: \hspace{4em}r_i ← r_i + \min\{w_i, d_i - r_i\}
17: \hspace{4em}a_i ← a_i + \min\{w_i, d_i - r_i\}
18: \hspace{3em}round ← round - \min\{w_i, d_i - r_i\}
19: \hspace{3em}n ← HC\text{apacity}(P, VMs_1) // Algo. 3
20: \hspace{3em}⇒ VMs_1 ≤ r_i \#used VMs from R_1
21: \hspace{3em}if n > 0 then
22: \hspace{4em}a_i ← a_i + n
23: \hspace{3em}end if
24: \hspace{3em}end if
25: \hspace{2em}end for
26: \hspace{1em}round ← \sum_{i=1}^{P_i} \min\{w_i, d_i - r_i\}
27: \hspace{1em}end while
28: \hspace{1em}return round fractional r
29: end procedure

Fig. 2. Example to illustrate the partitioning step.

the smallest weight equal to one (line 9). For instance, if we have only two partitions with weights two and four, the weight of these partitions become one and two. Additionally, if any partition has zero weight, its weight becomes one. The only partition that may have zero weight is the partition that contains CP.

In each iteration, a partition is allocated number of resources equal to the minimum of either the partition normalized weight \(w_i\) or the difference between its demand and the number of VMs already allocated to this partition \(d_i - r_i\) (lines 16 and 17). The later allocation only occurs when the number of VMs required by the partition to achieve full utility is lower than its normalized weight. The full utility is achieved when the number of resources allocated to a partition is equal to its demand \((r_i = d_i)\). Partitions that obtain full utility will be removed from consideration in consecutive iterations. This allocation process stops when either all of the partitions achieve the full utility or the number of available VMs \(a_1\) is not enough to at least satisfy a single round of allocation (while loop line 12).

Afterwards, a partition attempts to replace some of its allocated VMs with higher capacity VM (line 19 and Algorithm 3). If there are enough high capacity VMs available, the replacement takes place and the replaced VMs (from \(R_1\)) are added to \(a_1\) (lines 21 and 22). We also identify the type of resource to be allocated to each task. In particular, we identify the maximum number of tasks (not labeled as high capacity VM tasks) with overlapping execution time (7). We order these tasks based on EST. Then, starting from resource set \(R_1(i = 2)\), we determine the maximum number of tasks \((e T)\) that can be assigned to a single resource from this high capacity resource (while loop, line 12). We have to ensure that this assignment does not result in increasing the expected finish time of these tasks (lines 11 and 21 in Algorithm 3).

As VMs are non-divisible entities, if the final allocation results in allocating a fractional number of VMs to a partition, we round down, e.g., 2.5 VMs is rounded down to 2 VMs. Resource allocation with a fractional number of VMs happens because of the weight value of the partition.

The resource allocation algorithm results in envy-free Pareto efficient allocation of the resources among the partitions. An allocation is envy-free if there is no partition envy other partition share of the resources [22]. In other words, a partition cannot achieve better utility if it has been assigned to a different share of resources. Initially, all partitions have zero utility, and at the end of each iteration all partitions with unsatisfied demand has the same utility. This is established since in each round the additional share of resources assigned to each partition improves the utility of all partitions by a constant factor. Partitions that achieve their demands are removed for consideration in subsequent step, since they have achieved the full utility. Thus, at the end of each iteration, the allocation is envy-free, as unsatisfied partitions have the same utility, and the demand of any satisfied partition is less than the minimum demand of any unsatisfied partition. Therefore,
Algorithm 3. Replacement with High Capacity VMs.
1: procedure HCAPACITY($P$, $VMs_1$)
2: > Input: $P$ ∈ $P$, VMs$_1$ #used VMs from $R_1$
3: > output: # of replaced VMs
4: $T$ ← maxoverlappingTasks($P_i$)
5: $T$ tasks are not labeled as high capacity VM tasks
6: $i$ ← 2
7: while $i$ ≤ |$R_i$| do
8: if |$R_i$| > 0 then
9: for $v_i, v_j$ ∈ $T$ do
10: $L$ ← Add($v_i, v_j$)
11: cond ← $ex(L, R_i)$ ≤ $ex(L, R_i)$
12: while $|L|$ ≤ VMs$_1$ and cond do
13: $te$ ← find task in $T \setminus L$
14: $te$ use VM from $R_1$
15: if $te$ is null then
16: tag $T$ tasks as high capacity VM tasks
17: reduce $R_i$ size by 1
18: return $L$
19: end if
20: $L$ ← Add($te$)
21: cond ← $ex(L, R_i)$ ≤ $ex(L, R_i)$
22: end while
23: Undo last addition to $L$
24: if $|L|$ ≥ 2 then
25: tag $T$ tasks as high capacity VM tasks
26: reduce $R_i$ size by 1
27: return $L$
28: end if
29: end for
30: end if
31: end while
32: $i$ ← $i$ + 1
33: return 0
34: end procedure

at the end of this algorithm, the resultant resource allocation between the partitions is envy-free. Similarly, we can show that the resultant allocation is Pareto efficient, since we cannot increase the utility of any partition without reducing the utility of at least one other partition.

4.4. Task scheduling

In this step (Algorithm 4), we schedule tasks based on AFT. For each partition, the order in which tasks are scheduled is based on their level in the partition—from the entry level. We start by scheduling the tasks located at the lowest level in the partition (for loop line 10). Then, for each task located at the current considered level, we determine the VM that results in the earliest AFT for these tasks (FindVM procedure). Once we identify this VM, we assign it to execute the current considered task.

To clarify the processes of the resource allocation and task scheduling steps, consider the example shown in Fig. 3, where the resource allocation process is presented in Table 1. For simplicity, assume that the computation time for each task is 15 min. In this example, assume that the number of available VMs from $R_1$ is seven. Clearly, in this example, to achieve full utility partition $p_1$ requires a single VM ($d_1 = 1$), $p_2$ requires one VM ($d_2 = 1$), $p_3$ requires three VMs ($d_3 = 3$), and $p_4$ requires three VMs ($d_4 = 3$). For each partition, the demand (number of VMs required to achieve full utility) equals the number of tasks with overlapping execution time (EST and EFT). In this example, partitions one and two depend on partition four ($w_4 = 2$). Also, partitions one and three depend on partition two ($w_2 = 2$). Partition one's data is required by partitions two and three ($w_1 = 3$), and all partitions depend on partition three ($w_3 = 3$).

We start by normalizing the weight, to achieve the smallest weight equal to one ($w_1 = 1, w_2 = 1, w_3 = 1.5, w_4 = 1$). In the first round, each of the partitions one, two, and four will be assigned a single VM, and partition three will be assigned 1.5 VMs. In this case, partitions one and two achieve full utility, since the number of allocated VMs is equal to their demand ($r = d$). In the second round, partition three will be allocated another 1.5 VMs, and partition four will be allocated one VM. Now, all available VMs are allocated, and the only partition that did not achieve full utility is partition four. As we mentioned, we treat the VMs as indivisible objectives. Thus, if the result of the final allocation is a fractional number, we round this number to the lowest closest integer (2.5 VMs is rounded to 2 VMs). In this example, both partitions three and four have the same demand. However, more partitions depend on partition three, and thus this partition must be prioritized.

4.5. Computational complexity

The calculation of EST and the EFT and the identification of CP takes at most $O(|V|)$ to converge. The partitioning step, the for loop (line 18) is executed $|b_{pr}|$ times. The process of grouping tasks requires $O(|V||P|)$ iterations, where |P| is the number of partitions. Overall, the partitioning algorithm requires time $T_p$:

$$T_p \leq |b_{pr}| \times O(|V||P|) = O(|b_{pr}| |V||P|)$$

The number of iterations in the resource allocation step depends on the available number of resources and the weight and demand of partitions. In the worst case, when the number of partitions is
equal to one and the number of available resources is enough to satisfy the demand, the resource allocation step takes at most \( d_i \) iterations to converge. In each iteration of this step, the worst case running time for the VMs replacement process is \( O(|V_i| \times |R_i|) \), where \( |V_i| \) is the number of tasks assigned to \( |R_i| \) VMs and \( |R_i| \) is the number of resource sets with computational capacity higher than the capacity of resources in \( R_i \). The calculation of AST takes \( O(V^2r) \), where \( r \) is the number of VMs used. The resource allocation and the task scheduling steps require time \( T_i \):

\[
T_i \leq |d_i| \times O(|V_i||R_i|) + O(V^2r) = O(V^2r)
\]

Overall, the worst case running time of the RDAS+ algorithm is:

\[
T_{RDAS+} \leq O(V^2r) + O(|V||P|)
\]

5. Evaluation

In this section, we present our evaluation results in terms of four performance metrics:

- **Makespan**, 
- **Cost** (calculated based on the amount of resource/VM usage), 
- **The number of used VMs**, 
- **Idle time** (total amount of paid unused time)

We have used five types of real scientific workflows: LIGO, Montage, Epigenomics, SIPHT and CyberShake (Fig. 1). These workflows are obtained from the Pegasus workflow repository (https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator). There are a total of 300 workflows used in our experiments. We focus on workflows with 1000 tasks or above, i.e., 1000 – 6000 tasks in our experiments. As scientific workflows are repeatedly run in practice, their characteristics, including task execution times and amounts of (intermediate) data to be communicated, can be estimated by profiling. We used three resource sets \( R = R_1, R_2 \) and \( R_3 \). \( R_1 \) VMs are three times faster and more expensive compared to \( R_1 \) VMs, and \( R_3 \) VMs are twice faster and more expensive compared to \( R_1 \) VMs. We also assume that each VM consists of 8 vCPUs with identical processing power. The actual hourly rates used in our experiments are \( \hat{C} \) 10 ($0.1), \hat{C} 20 ($0.2) and \( \hat{C} 30 ($0.3) for \( R_1, R_2 \) and \( R_3 \), respectively. The inter-node bandwidth is assumed to be 1Gbps. The results in this section is an average of 10 different runs, and 95% confidence interval is shown.

To benchmark the performance of RDAS+, we compare its performance against PBWS [11], IC-PCP [12] and HEFT [8]. In particular, we have conducted a wide range of experiments in a simulated environment. In IC-PCP, we are given a pre-determined deadline \( D \), and the objective is to determine the cheapest schedule that satisfies this deadline without the upper bound of the number of VMs. In HEFT, the number of VMs is bounded by the resource limit of each resource type in this work, and the objective is to minimize makespan. This algorithm works in a greedy manner, where in each scheduling step, the task with the highest priority is assigned to the resource with which the task can finish the earliest. In PBWS, the optimization of schedule is determined by the value of slack parameter \( \beta \). The value of \( \beta \), \( \beta \in [0,1] \), trades between makespan and cost. \( \beta = 0 \) enforces that each task is executed by its EST to minimize makespan while \( \beta = 1 \) tends to make tasks in each partition are executed by a single VM to minimize cost. For the fairness of the comparison, we start by running PBWS for \( \beta \) values of 0 and 1. Then we use the makespan of schedule obtained by PBWS as an input to IC-PCP (D). Additionally, we use number of used VMs (from each resource set) by PBWS as inputs to RDAS+ and HEFT.

5.1. Evaluation results

We discuss below our evaluation results for each of the input scenarios (Figs. 4–8). Table 2 summarizes the performance of the algorithms with concrete values.

5.1.1. LIGO

Fig. 4 shows that increasing the number of available VMs results in reducing the makespan obtained by RDAS+. Actually, at a low value for \( \beta = 0 \), RDAS+ outperforms the other algorithms in terms of makespan. In addition, we can see that in terms of idle time, the gap between RDAS+ and the other algorithms is significant. In LIGO experiments, due to the structure of the workflows, demand and weight values of different partitions are very similar. This results in increasing the quality of the schedule obtained by RDAS+, since partitions are more likely to finish the execution relatively at the same time. By having such a structure, RDAS+ also increases the utilization of the resources, since the behavior of execution across the partitions will be similar.

5.1.2. Montage

From the results (Fig. 5) we can see that RDAS+ constantly outperforms the other algorithms in terms of cost and idle time. To understand this behavior, let us consider the main mechanism in each algorithm. In PBWS, the main idea is to use the slack parameter to control the overall execution time. This is clearly shown...
in the results, since reducing the value of this parameter results in increasing the number of available VMs, and this reduces the makespan. However, increasing the number of VMs without quantifying the benefits of such addition (in terms of VMs) can result in wasting the available resources. This is established, since eventually the data dependency between the tasks plays a major role in determining the upper bound on the number of VMs that we can actually use. IC-PCP determines its scheduling strategy based on the pre-determined deadline. Toward this end, this algorithm might result in very high scheduling cost, since bounding the execution deadline is expected to increase the number of required VM. In HEFT, by scheduling the tasks on the VMs that results in lowering the execution time, this algorithm is expected to achieve low makespan, if the number of available VMs is relatively high. This algorithm does not consider the structure of the workflow during the establishment of the schedule, and this is expected to increase the cost associated with total idle time. In RDAS+, the idea of allocating VMs to the partitions based on their demand and
weight, results in reduction in the overall idle time. This is due to such assignment resulting in using the resources efficiently. No partition will be allocated more than its requirement. In addition, such assignment attempts to quantify the relationship between the makespan and the cost. Thus, at any stage, if allocating more VMs to a particular partition(s) does not result in reducing makespan, this allocation will be avoided. Overall, these are the main factors behind the performance behavior observed.

5.1.3. Epigenomics

Fig. 6 shows the results for the Epigenomics workflow experiments. We observe that RDAS+ outperforms other algorithms in terms of makespan and idle time. We can also see from the results that at low value for $\beta = 0$, IC-PCP outperforms RDAS+ in terms of cost. This behavior is related to the structure of the workflow, and the main mechanism each algorithm uses. This workflow has a well-defined and balanced structure, where most tasks (except the entry task) are involved in a single data-dependency relation-
ship. Thus, we can expect that RDAS+ will result in partitions with relatively same weight and demand. The balanced structure of the partitions and the low degree for the nodes results in increased efficiency of the schedule obtained by RDAS+, since this increases the probability of allocating most of the available VMs. In addition, as we can see from the results, IC-PCP uses a significantly lower number of VMs, and this is the main factor behind its performance in terms of cost. However, as shown in the figures (at $\beta = 0$), the gap between RDAS+ and IC-PCP is relatively small, and this is due to the efficient utilization of the available VMs by RDAS+.

5.1.4. SIPHT

The results for the SIPHT workflow experiments are shown in Fig. 7. Here, RDAS+ significantly outperforms other algorithms in terms of cost and idle time. Further, we can see that decreasing the value of $\beta$ increases the performance of RDAS+ in terms of makespan. This workflow is unstructured, compared to other workflows. Tasks belonging to the same level are involved in different numbers of dependency relationships. In RDAS+, this introduces a dependency relationship between the number of available VMs, and the makespan obtained. In this workflow, RDAS+ requires a large number of VMs in order to achieve competitive makespan, since partitions are expected to have a large number of parallel tasks. In this direction, RDAS+ will use a small percentage of the available VMs to maintain the Pareto-efficient and envy-free properties. However, RDAS+ efficiently utilizes the used resources, and this can be shown by taking into account the number of VMs used by RDAS+ and the performance gap between the algorithms in terms of makespan.

Table 2

<table>
<thead>
<tr>
<th>Workflow</th>
<th>Algo.</th>
<th>Makespan (s)</th>
<th>Cost (cent)</th>
<th>Idle time (s)</th>
<th># of VMs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\beta = 0$</td>
<td>$\beta = 1$</td>
<td>$\beta = 0$</td>
<td>$\beta = 1$</td>
<td>$\beta = 0$</td>
</tr>
<tr>
<td>LIGO</td>
<td>RDAS+</td>
<td>991</td>
<td>1531</td>
<td>98</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>PBWS</td>
<td>1395</td>
<td>1424</td>
<td>181</td>
<td>172</td>
</tr>
<tr>
<td></td>
<td>IC-PCP</td>
<td>1321</td>
<td>1399</td>
<td>244</td>
<td>242</td>
</tr>
<tr>
<td>Montage</td>
<td>RDAS+</td>
<td>259</td>
<td>397</td>
<td>242</td>
<td>203</td>
</tr>
<tr>
<td></td>
<td>PBWS</td>
<td>356</td>
<td>386</td>
<td>252</td>
<td>222</td>
</tr>
<tr>
<td></td>
<td>IC-PCP</td>
<td>230</td>
<td>323</td>
<td>364</td>
<td>432</td>
</tr>
<tr>
<td>Epigenomics</td>
<td>RDAS+</td>
<td>206</td>
<td>280</td>
<td>453</td>
<td>381</td>
</tr>
<tr>
<td></td>
<td>PBWS</td>
<td>19,093</td>
<td>19,174</td>
<td>132</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>IC-PCP</td>
<td>17,777</td>
<td>13,079</td>
<td>69</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>HEFT</td>
<td>17,075</td>
<td>17,257</td>
<td>107</td>
<td>100</td>
</tr>
<tr>
<td>SIPHT</td>
<td>RDAS+</td>
<td>4239</td>
<td>6791</td>
<td>103</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>PBWS</td>
<td>5285</td>
<td>8127</td>
<td>177</td>
<td>294</td>
</tr>
<tr>
<td></td>
<td>IC-PCP</td>
<td>5175</td>
<td>5288</td>
<td>321</td>
<td>307</td>
</tr>
<tr>
<td></td>
<td>HEFT</td>
<td>3649</td>
<td>3649</td>
<td>178</td>
<td>153</td>
</tr>
<tr>
<td>CyberShake</td>
<td>RDAS+</td>
<td>291</td>
<td>1465</td>
<td>399</td>
<td>408</td>
</tr>
<tr>
<td></td>
<td>PBWS</td>
<td>184</td>
<td>1133</td>
<td>510</td>
<td>459</td>
</tr>
<tr>
<td></td>
<td>IC-PCP</td>
<td>135</td>
<td>621</td>
<td>344</td>
<td>169</td>
</tr>
<tr>
<td></td>
<td>HEFT</td>
<td>249</td>
<td>1042</td>
<td>640</td>
<td>379</td>
</tr>
</tbody>
</table>
5.15. Cybershake

In the CyberShake workflow experiments (Fig. 8), the results show that RDAS+ is outperformed by all the other algorithms in terms of makespan. This is also related to the structure of the workflow. In this workflow, the length of the critical path is the shortest, compared to the other algorithms. This, together with the very wide structure of the workflow, results in the creation of a very dominant relationship between the number of available VMs, and the expected behavior for RDAS+. In this workflow, if the number of available VMs is relatively small, the algorithm shifts its behavior towards reducing the number of obtained partitions. As a result, the number of VMs assigned to each partition is small; and, this reduces the benefits of RDAS+.

6. Conclusion

In this paper, we have presented RDAS+, which optimizes resource efficiency of scientific workflow execution in clouds. RDAS+ enables efficient resource use in clouds by reducing the number of VMs allocated and idle time between task executions leading to significant cost savings with relatively small degree of Makespan increase. In particular, the optimization of resource use implicitly helps performance, i.e., makespan. As cost is an important consideration when running scientific workflows in clouds, the capacity of RDAS+ in delivering cost efficient schedules is of great practical importance. Besides, the degree of cost efficiency improvement is even higher as the scale of scientific workflows continues to increase, e.g., a 6.0° Montage workflow with over 8.5k tasks and 35 GB of data footprint.

References


Khaled Almi’ani is an Associate Professor at the Department of Computer Science, Princess Sumaya University for Technology, Jordan. He is currently on a sabbatical from Al-Hussein Bin Talal University, Jordan. His research interests lie in algorithms for distributed systems, network optimization, and transportation network modeling. He received his Ph.D. in Information Technology from the University of Sydney in 2010.

Young Choon Lee is currently a senior lecturer at Department of Computing, Macquarie University. His research is centered around distributed systems with particular interests of scheduling, resource management and sustainability.

Bernard Mans is Professor at the Department of Computing at Macquarie University, Sydney, Australia, which he joined in 1997. His research interests center on algorithms and graphs for distributed and mobile computing, and in particular on wireless networks. In 2003, he was the HITACHI-INRIA Chair at INRIA, France. He received his Ph.D. in Computer Science from University Pierre et Marie Curie, Paris 6, while at INRIA France, in 1992.