Optimal Time-Cost Tradeoff of Parallel Service Workflow in Federated Heterogeneous Clouds

Gueyoung Jung and Hyunjoo Kim
Xerox Research Center Webster
New York, USA
{gueyoung.jung, hyunjoo.kim}@xerox.com

***Abstract*—Federated cloud enables a workflow to be deployed in multiple private and public clouds. By facilitating external cloud-based services to execute sub-tasks of the workflow, service workflow owners can reduce the cost of executing the workflow, while meeting a performance requirement, since those cloud-based services can be more cost efficient and have better performance than internal ones. However, due to inter-dependencies between sub-tasks, the complexity of the workflow, and the heterogeneity of clouds, it is a challenge to achieve the optimal tradeoff between cost and performance. This paper presents a novel workflow scheduler designed to achieve the optimal *end-to-end* execution time and cost when deploying such complex workflows in heterogeneous computing nodes in clouds. Specifically, our scheduling algorithm addresses the tradeoff between the execution cost, the computing time, and the data transfer delay between sub-tasks. Our scheduler can handle complex workflows that contain *recursively paralleled* sub-flows caused by branch and merging sub-tasks. Experiments indicate that our scheduler can efficiently compute the near optimal deployment compared with greedy and evolutionary algorithms for both *end-to-end* execution time and corresponding cost.**

***Keywords*-optimization; tradeoff; parallel workflow; cloud**

I. Introduction

Federated cloud (or often called hybrid cloud) has been increasingly popular in the last few years by allowing enterprises to flexibly select the mix of different cloud-based services for various performance requirements [1], [2], [3], [4]. By outsourcing such cloud-based services, enterprises can deploy their workflows with high service availability and cost-efficiency while maintaining reasonable execution time. We consider the workflow as business process that consists of many sub-tasks. Each sub-task of the workflow can be instantiated by a cloud-based service that can be provided by many external cloud providers (i.e., 3^{rd} party cloud-based services) or developed internally [5], [6].

Many open source and commercial Business Process Management Systems (BPMS) [7], [8], [9] have been developed for flexible service composition over service-oriented architectures. Those systems can seamlessly compose executable workflows using existing external services. However, they have focused on functional features to deploy workflows by providing a list of available service applications and indicating inputs and outputs required to compose workflows, rather than non-functional features such as cost and performance.

We have also been developing a BPMS that is designed to automatically compose executable workflows, deploy them into clouds, and manage the cloud-based workflows [10], [11]. It allows users to design an abstract workflow (i.e., business process) using a graphical interface and standard Business Process Modeling Notation (BPMN [1]). It finds all service applications and computing nodes in clouds for each sub-task and then, our workflow scheduler identifies an optimal executable workflow among all possible combinations of those service applications and computing nodes. It executes the workflow using an existing business process engine [7] when a request is invoked. Finally, it keeps monitoring the workflow process and reporting when any exception occurs. Please refer to [10], [11] for more detail.

In this paper, we focus on the approach to identify the optimal workflow deployment. Specifically, we consider three challenges in the context of non-functional features on heterogeneous clouds. First, the workflow scheduler typically has to explore a huge search space of possible deployment combinations (i.e., a combinatorial problem). Each sub-task of a workflow can be instantiated by a number of alternative services and each service can be hosted in a number of alternative computing nodes in clouds that provide different performance and cost offerings. Users have to determine which clouds and service applications would be integrated into their workflows to optimize the performance and cost.

Second, we consider the potential tradeoffs between service computation time, execution cost, and data transfer delay. Even though we select services, each of which has the fastest computation time among alternatives, the end-to-end workflow execution time may not be optimal if those services are geographically distributed and data transfer delay between services is significant. Also the service execution cost is likely to be in opposite side to service computation time (i.e., the highly capable service is typically more expensive than less capable service). Therefore, these three factors need to be balanced unless one of them needs to be considered significantly than others as a preference.

Finally, a workflow often has multiple branch and merging sub-tasks, and a branch is either OR-branch (i.e., if-then-else) or AND-branch (i.e., parallel processing). This paper focuses on parallel sub-flows induced by AND-branch, where parallel sub-flows run simultaneously after a branch sub-task,

[1]Business Process Model and Notation 2.0, available at http://www.omg.org/spec/BPMN/2.0

978-0-7695-5025-1/13 $26.00 © 2013 IEEE
DOI 10.1109/ICWS.2013.73

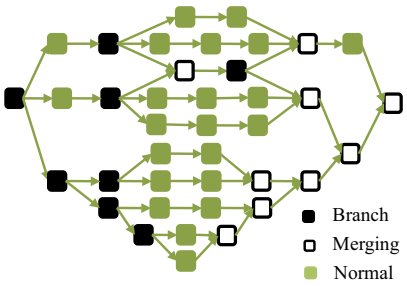

Fig. 1. A recursively parallel workflow (a business process)

and synchronously end at a merging sub-task. Each sub-flow can also have *parallel sub-flows recursively*. As mentioned in some prior work such as [6], BPMSs should deal with various workflow structures such as *loop*, *scope*, and *if-then-else*. However, to the best of our knowledge, none has precisely addressed the recursive parallel sub-flows in BPMS. Meanwhile, we keep improving our BPMS by integrating such workflow structures. Figure 1 shows an example of the recursively parallel workflow that is the abstract for one of our complex business processes to be deployed into clouds. This workflow has been used in our experiments.

Parallel sub-flows make the search problem further complicated since the scheduler must consider *critical sub-flows*, which are slower than other sub-flows, and the *slack time* caused by different velocity of sub-flows to reach a merging sub-task from a branch. When dealing with parallel sub-flows, the scheduler must choose service applications and computing nodes in a way of minimizing the slack time by rebalancing execution time and cost. For example, if a critical sub-flow dominates execution time in parallel sub-flows, the scheduler selects cheaper (and may be slower) computing nodes instead of expensive and fast nodes.

In this paper, we precisely address these challenges by providing the near optimal deployment of complex workflows on cloud-based services that have heterogeneous performance and cost. Specifically, we have developed a workflow scheduler, which can find the near optimal workflow deployment minimizing both *end-to-end* workflow execution time and its cost for given time and cost weights (i.e., $pareto$ optimization format [12]). To do this, we cast the search problem into the shortest path problem in a directed graph, which has AND-branch, OR-branch, and merging vertices. Vertices have positive values, which reflect computation time and cost of each instantiation of a sub-task, and edges also have positive values, which reflect data transfer delay between sub-tasks. Then, the goal is to find an optimal path, which has the minimum accumulated execution time and cost between source and sink. We have extended A* graph search algorithm [13] to speed up the search in a potentially large search space.

II. Problem Description

We first define execution time and cost considered in our workflow scheduler. The end-to-end execution time of a workflow consists of the sum of computation times in computing nodes, each of which runs a sub-task of the workflow, and the sum of data transfer delays between two consecutive computing nodes. For cost, we can include various cost factors such as resource usage based on pay-as-you-go in public clouds, software license, and cloud service usage per transaction. The overall cost to run the workflow is the sum of all such cost. We assume that computation time, data transfer delay, and cost are different in computing nodes (i.e., heterogeneity).

We define the *deployment* of sub-task into a computing node as allocating the sub-task to the cloud-based service that is run in the computing node, rather than allocating (or placing) the cloud-based service into the computing node. This is because we assume that the location of the cloud-based service (especially, 3^{rd} party service) is not changed unless the service provider changes the location.

In our search problem, since we have two different units, time and cost, we normalize them to compare different deployments. Hence, we define a linear utility function as follows,

$$u = \theta_t u_t + \theta_c u_c = \theta_t \frac{t - t^{min}}{t^{max} - t^{min}} + \theta_c \frac{c - c^{min}}{c^{max} - c^{min}} \quad (1)$$

where $\theta_t + \theta_c = 1$. The overall utility u consists of the execution time utility u_t and the cost utility u_c with weight factors θ_t and θ_c, respectively. Given the upper-bound execution time t^{max} and the upper-bound cost c^{max}, we normalize measured execution time t and cost c. Note that lower-bound time and cost, t^{min} and c^{min}, can be ignored in Equation 1 unless they are given in SLA.

When we denote a set of services by $\mathcal{A} = \{a_1, a_2, \ldots, a_l\}$ and a set of computing nodes by $\mathcal{R} = \{r_1, r_2, \ldots, r_m\}$, a service a_j is deployed to at least one computing node, and a computing node r_i can host at least one service. If a_j is deployed to r_i, it is represented as $r_i.a_j$. We define an abstract workflow W^{abt} which consists of a set of sub-tasks $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$. Each sub-task $s_k \in \mathcal{S}$ can be instantiated with at least one service from $\mathcal{A}$. If s_k is instantiated as $r_i.a_j$, we briefly denote it by $d_{ij}(s_k)$. We also define $D(s_k)$ as a set of all possible instantiations for s_k. Therefore, W^{abt} can be deployed into clouds as a workflow W_h chosen from all possible workflow deployments $\mathcal{W} = \{W_1, W_2, \ldots, W_o\}$. Each workflow $W_h \in \mathcal{W}$ can be obtained by replacing its sub-task $s_k \in \mathcal{S}$ with a $d_{ij}(s_k) \in D(s_k)$.

Given service applications $\mathcal{A}$, computing nodes $\mathcal{R}$, an abstract workflow W^{abt} and its sub-tasks $\mathcal{S}$, the problem is to identify an optimal workflow deployment W^*, which minimizes the total utility value u in Equation 1, from a set of all possible workflow deployments $\mathcal{W}$.

To calculate execution time in a sequential sub-flow (see Figure 2), we can choose the best instantiation $d_{ij}(s_k)^*$, which has the minimum computation time for s_k by simply comparing all possible instantiations in $D(s_k)$. We can also obtain the minimum data transfer delay between two consecutive sub-tasks by comparing all possible combinations between $D(s_k)$ and $D(s_{k+1})$. Then, the sum of those values is the fastest end-to-end execution time of the sequential sub-flow. The least cost of the sequential sub-flow can be obtained similarly. The least

accumulated computation time from s_1 to $s_{n'}$ of a sequential sub-flow is formulated as follows,

$$t^{c*}|_1^{n'} = \sum_{k=1}^{n'} min\{t_k^c \mid \forall d_{ij}(s_k) \in D(s_k)\} \quad (2)$$

where t_k^c is a computation time for $d_{ij}(s_k)$. The least accumulated data transfer delay from s_1 to $s_{n'}$ is formulated as follows,

$$t^{d*}|_1^{n'} = \sum_{k=1}^{n'-1} min\{t_k^d \mid \forall d_{ij}(s_k) \in D(s_k), \forall d_{i'j'}(s_{k+1}) \in D(s_{k+1})\} \quad (3)$$

where t_k^d is a transfer delay between $d_{ij}(s_k)$ and $d_{i'j'}(s_{k+1})$. Then, the optimal end-to-end execution time of the workflow up to $s_{n'}$ can be defined as

$$t^*|_1^{n'} = t^{c*}|_1^{n'} + t^{d*}|_1^{n'}. \quad (4)$$

The optimal cost of the workflow up to $s_{n'}$ can be formalized as follows,

$$c^*|_1^{n'} = \sum_{k=1}^{n'} min\{c_k \mid \forall d_{ij}(s_k) \in D(s_k)\}. \quad (5)$$

where c_k is the cost for $d_{ij}(s_k)$.

In parallel sub-flows (see Figure 3), computing the optimal execution time is more complicated than in sequential sub-flow, while computing the optimal cost is identical as shown in Equation 5. We have to consider time margin (*slack time*) between the slowest sub-flow and relatively faster sub-flows. Even though a fast sub-flow is complete and reaches the merging sub-task, the merging sub-task has to wait until other slower sub-flows finish their sub-tasks. Hence, the execution time of the parallel sub-flows is dominated by the slowest sub-flow. We denote the execution time of each parallel sub-flow by $w_p.t^*|_1^{n'}$ computed with Equation 4. Then, the execution time of parallel sub-flows can be defined as,

$$t^*|_1^{n'} = max\{w_p.t^*|_1^{n'} \mid \forall w_p \in \bar{\bar{W}}(s_b)\} \quad (6)$$

, where $\bar{\bar{W}}(s_b)$ is a set of parallel sub-flows starting from the branch sub-task s_b, and w_p is a single parallel sub-flow in $\bar{\bar{W}}(s_b)$. The parallel sub-flows should be synchronized at the merging sub-task (i.e., the workflow can proceed when all parallel sub-flows are merged to a sub-task). Note that s_b is considered as a starting sub-task of $AND-branch$, which triggers all sub-flows simultaneously. For OR-branch, since only one of parallel sub-flows will be selected to proceed at runtime, the scheduler does not know which sub-flow will be executed in advance. Therefore, we can consider 1) a conservative strategy that computes the execution time of parallel sub-flows with the slowest sub-flow using Equation 6, or 2) an aggressive strategy that computes it with the fastest sub-flow, or 3) taking a median value of execution time of all sub-flows. In our current prototype, we use the conservative strategy. However, if we know the selection probability for each sub-flow, we would elaborate it with a better approximation. This will be considered in the future work.

To find an optimal workflow deployment, the workflow scheduler should explore the search space of possible workflow deployments, calculate utility function and then, compare them. It will be expensive to use such brute-force, which investigates all possible workflow deployments. Hence, we develop a heuristic approach to find a near optimal workflow deployment in an efficient way.

III. Workflow Scheduler

A. Execution Time Estimation

To evaluate workflow deployments, the scheduler has to estimate computation time and data transfer delay accurately. There are many existing estimation techniques such as queueing models [14], response surface model [4], kalman filter [15], and auto-regressive moving average (ARMA) filter [16], [17]. While our approach is not limited to use any specific technique, we use ARMA filter in our current prototype (see [17] for detail) for estimating computation time and data transfer delay. We refer to static pricing scheme for estimating cost that is used by most cloud providers.

B. Scheduling Algorithm

We cast this optimization problem into *a weighted shortest path problem* in a directed graph $G =< V, E >$, where a vertex $v \in V$ represents *a possible instantiation for each sub-task* of given abstract workflow, and an edge $e \in E$ represents a path between two consecutive vertices. The heuristic algorithm explores the abstract workflow from source to sink by first expanding each sub-task s_k to a set of vertices (possible instantiations) and then, selecting the best vertex v^{best}. While the algorithm visits each v, its utility u is calculated with Equation 1 using its computation time t^c, data transfer delay t^d, and cost c. Then, the utility is cumulated while exploring the workflow. When the algorithm reaches sink, the accumulated utility will be close to minimal if the algorithm successfully finds the optimal path. Note that we assume the source is a user location, where a request is invoked. If the workflow is a closed one (i.e., the final result goes back to the user), the sink is the user location as well.

1) Greedy vs. Dynamic Algorithms: A greedy algorithm can be applied because it is typically faster than dynamic algorithms. However, it does not guarantee to find a global optimal path. As an example, in Figure 2, the greedy algorithm will select the instantiation $d_{1,3}(s_1)$ for sub-task s_1 if it has the best utility among all nine instantiations. Note that $d_{1,3}(s_1)$ represents service application a_3 running on computing node r_1 for sub-task s_1. The algorithm will further select the best instantiations for the rest of sub-tasks as shown in Figure 2. However, when data transfer delay is considered, and if the data transfer delay between $d_{4,12}(s_2)$ and $d_{7,21}(s_3)$ is very high, the alternative path (dashed arrows) can be a better choice since $d_{6,16}(s_2)$ and $d_{6,25}(s_3)$ are in the same node r_6.

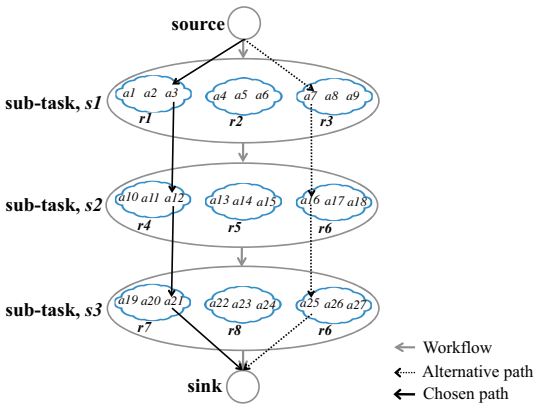

Fig. 2. A simple sequential workflow

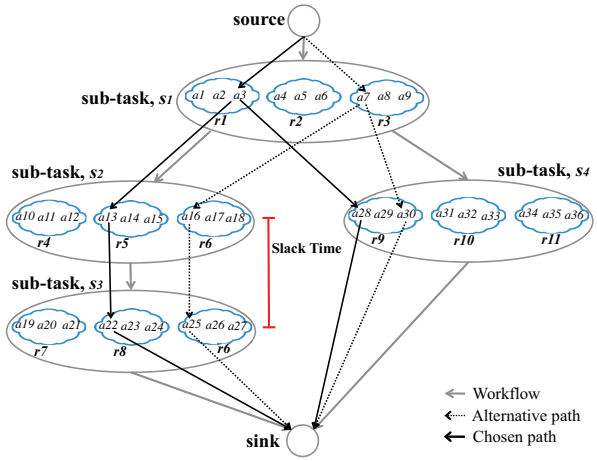

Fig. 3. A simple workflow that has two parallel sub-flows

In the parallel workflow, the greedy algorithm can be even worse due to the slack time caused by a critical sub-flow (i.e., the slowest sub-flow). Figure 3 shows a simple workflow that contains two parallel sub-flows, w_1 and w_2, represented as $source \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow sink$ and $source \rightarrow s_1 \rightarrow s_4 \rightarrow sink$, respectively, where $source$ is a branch, and $sink$ is a merging sub-task of the workflow. As described in Equation 6, the optimal execution time is determined by the critical sub-flow. Let us assume that w_1 is the critical sub-flow. Then, w_2 has a slack time at $sink$, which means w_2 should wait until w_1 is complete and reaches $sink$. The larger the slack time is, the farther it is away from the optimal path. To minimize the slack time, the algorithm should be able to select an alternative path as shown in Figure 3, which may be *slower enough* but cheaper. Note that "slower enough" means new w_2 is slightly faster than w_1 or ideally same.

To address this problem, we have designed a dynamic algorithm that efficiently revisits alternatives to find better paths. Our algorithm extends the A* algorithm, which is one of dynamic path-finding algorithms introduced in [13]. This algorithm uses a heuristic function h to estimate the optimal utility from the current node to $sink$, and a function g to calculate the accumulated utility from $source$ to the current

Algorithm 1 Searching optimal time-cost tradeoff

Input: W^{abt}, $\mathcal{S}$, $\mathcal{A}$, $\mathcal{R}$, θ_t, θ_c, t^{max}, c^{max}
Output: W^*, t^*, c^*
$v^{best} \leftarrow \emptyset$; $v_0.\mathcal{D} \leftarrow v_0.\mathcal{D} \cup$ '$source$';
$Q \leftarrow Q \cup v_0$;
$\mathcal{T} \leftarrow \emptyset$;
for each $s_k \in \mathcal{S}$ **do**
  $W^{abt}.\mathcal{D}(s_k) \leftarrow$ `SetInstantiations`$(s_k, \mathcal{A}, \mathcal{R})$;
end for
`ComputeAdmissibleTimeAndCost`$(W^{abt}.s_1, \mathcal{T})$;
while $\exists d' \in v^{best}.\mathcal{D}$ s.t. $d' \neq$ '$sink$' **do**
  $v^{best} \leftarrow Q$.`removeLast()`;
  $\mathcal{V} \leftarrow$ `Expand`$(v^{best}, W^{abt}, \mathcal{T}, \theta_t, \theta_c, t^{max}, c^{max})$;
  for each $v \in \mathcal{V}$ **do**
    if $\exists v' \in Q$ *s.t.* $v' = v$ **then**
      if $v'.u^* > v.u^*$ **then** Q.`remove`(v'); **end if**
      if $v'.u^* \leq v.u^*$ **then** *continue*; **end if**
    end if
    $Q \leftarrow Q \cup v$;
  end for
  `Sort`(Q) by $(v.u^*)$;
end while
$W^* \leftarrow$ `GetOptimalWorkflow`(v^{best});
return $W^*, v^{best}.t^*, v^{best}.c^*$;

node. Therefore, the optimal utility u^* is defined as $g + h$. It is a challenge to choose a good h. To guarantee an optimal utility, h should not overestimate the utility to reach the $sink$, otherwise, the algorithm may lose a chance to find u^*.

2) Heuristic Search Algorithm: The overall procedure of our heuristic approach is described in Algorithm 1. It takes the abstract workflow W^{abt} with sub-tasks $\mathcal{S}$, a set of applications $\mathcal{A}$, a set of computing nodes $\mathcal{R}$, and parameters for the utility function (Equation 1) as inputs. Then, the algorithm outputs the optimal workflow deployment W^* with the optimal end-to-end execution time t^* and cost c^*. Each vertex v has a data structure $v.\mathcal{D}$ that maintains instantiation(s) for each sub-task. Q is a descending queue that stores all vertices sorted by utility estimate $v.u^*$, and $\mathcal{T}$ is a tree structure for sub-flow traces maintaining all parallel sub-flows.

The algorithm first obtains a set of candidate instantiations $W^{abt}.\mathcal{D}(s_k)$ for each sub-task s_k as shown in the first for loop in Algorithm 1. It maps service applications in $\mathcal{A}$ into available computing nodes profiled in $\mathcal{R}$ for each sub-task s_k. Then, the algorithm calls `ComputeAdmissibleTimeAndCost(.)`, where heuristic time and cost from s_k to $sink$ are calculated (details in Algorithm 2).

In the $while$ loop, the algorithm explores candidate paths by expanding v (by calling `Expand(.)`) until all instantiations in $v^{best}.\mathcal{D}$ reach $sink$ (details in Algorithm 3), where v^{best} is selected from Q as the next starting point of search. If an expaned vertex has been explored already, the algorithm chooses the vertex with smaller utility. Finally, in `GetOptimalWorkflow`(v^{best}), the algorithm tracks back to $source$ while mapping chosen instantiations to sub-tasks.

Computing admissible time and cost: Algorithm 2 computes heuristic time and cost from each s_k to $sink$ as "admissible time to go" $\hat{t}|_k^n$ and "admissible cost to go" $\hat{c}|_k^n$,

Algorithm 2 Computing admissible time and cost

Input: s_k, $\mathcal{T}$
if $s_k =$ 'end' **then**
  w.push(s_k); $\mathcal{T}$.addSubFlow(w);
  $return$;
end if
$S \leftarrow s_k$.nextSubTasks();
if $|S| = 1$ **then**
  ComputeAdmissibleTimeAndCost($s_{k+1} \in S, \mathcal{T}$);
  $s_k.\hat{t}|_k^n \leftarrow s_k.t^*|_k^k + s_{k+1}.\hat{t}|_{k+1}^n$;
  $s_k.\hat{c}|_k^n \leftarrow s_k.c^*|_k^k + s_{k+1}.\hat{c}|_{k+1}^n$;
end if
if $|S| > 1$ **then**
  for each $s_{k+1} \in S$ **do**
    ComputeAdmissibleTimeAndCost($s_{k+1}, \mathcal{T}$);
  end for
  $s_k.\hat{t}|_k^n \leftarrow s_k.t^*|_k^k + \mathcal{T}$.MaxTimeOfSubFlows($s_k$);
  $s_k.\hat{c}|_k^n \leftarrow s_k.c^*|_k^k + \mathcal{T}$.UnionCostOfSubFlows($s_k$);
end if
if $|s_k$.PreviousSubTasks$()| > 1$ and $|S| = 1$ **then**
  $w \leftarrow \mathcal{T}$.getLastSubFlow();
  w.push(s_k);
end if
if $|S| > 1$ **then**
  $\mathcal{T}$.pushBranchToSubFlows(s_k);
end if

respectively. Then, the algorithm assigns the heuristic time and cost to s_k while recursively parsing W^{abt}. For the heuristic time and cost to be admissible (i.e., not overestimated), they can be computed by assuming the best condition that a single instantiation $d_{i,j}(s_k)$ has both the minimum computation time and the minimum cost among all candidates, and there is no data transfer delay between two instantiations.

When the algorithm recursively returns from $sink$, it records all branch sub-tasks and corresponding merging sub-tasks in a sub-flow w and stores w in $\mathcal{T}$. Meanwhile, it computes the local best computation time $s_k.t^*|_k^k$ using Equation 2 and the local best cost $s_k.c^*|_k^k$ using Equation 5.

Computing $s_k.\hat{t}|_k^n$ and $s_k.\hat{c}|_k^n$ at a branch sub-task is complicated because the sub-tree starting from the branch sub-task can also have other branch sub-tasks and merging sub-tasks (i.e., its sub-trees). To compute $s_k.\hat{t}|_k^n$, the algorithm takes the maximum heuristic time among sub-flows as described in Equation 6 using $\mathcal{T}$.MaxTimeOfSubFlows(s_k). To compute $s_k.\hat{c}|_k^n$, the algorithm considers the overlapped sub-trees caused by common merging sub-tasks, which leads to redundant cost additions. Hence, it first obtains the union of sub-flows and then, computes the cost for the union of sub-flows using $\mathcal{T}$.UnionCostOfSubFlows(s_k).

Expanding to the next candidate paths: Algorithm 3 describes how to expand the current path by generating the next vertices from the chosen instantiation $d^{exp} \in v.\mathcal{D}$, and compute each expanded path's utility estimate $v^{nt}.u^*$. The algorithm first selects d^{exp} which has the minimum transfer delay $d^{exp}.t^d$ from its previous instantiation and the minimum computation time $d^{exp}.t^c$ among non-merging instantiations in $v.\mathcal{D}$. Each instantiation $d \in v.\mathcal{D}$ has been chosen *from each parallel sub-flow* (Note that $\|v.\mathcal{D}\| = 1$ in sequential

Algorithm 3 Expanding to the next candidates

Input: v, W^{abt}, $\mathcal{T}$, θ_t, θ_c, t^{max}, c^{max}
Output: $\mathcal{V}^{nt}$
$d^{exp} \leftarrow \emptyset$; $t \leftarrow \infty$;
for each $d \in v.\mathcal{D}$ **do**
  if $|d$.preSubTasks$()| = 1$ and $d.t^c + d.t^d < t$ **then**
    $d^{exp} \leftarrow d$; $t \leftarrow d.t^c + d.t^d$;
  end if
end for
$S^{nt} \leftarrow d^{exp}$.nextSubTasks();
$\mathcal{D}^{nt} \leftarrow \emptyset$;
for each $s^{nt} \in S^{nt}$ **do**
  if $\exists d(s^{nt}) \in W^{abt}.\mathcal{D}(s^{nt})$ s.t. $d(s^{nt}) \in v.\mathcal{D}$ **then**
    $continue$;
  end if
  $\mathcal{D}^{nt} \leftarrow \mathcal{D}^{nt} \cup W^{abt}.\mathcal{D}(s^{nt})$;
end for
if $|v.\mathcal{D}| > 1$ **then** $\mathcal{D}^{nt} \leftarrow \mathcal{D}^{nt} \cup v.\mathcal{D}$; **end if**
$\mathcal{C} \leftarrow$ getCombinationsOfInstantiations($\mathcal{D}^{nt}$);
for each $D \in \mathcal{C}$ **do**
  $v^{nt}.t^*|_1^{n'} \leftarrow \max\{(v.t^*|_1^{n'-1} + d.t^c + d.t^d) \mid \forall d \in D\}$;
  $v^{nt}.c^*|_1^{n'} \leftarrow v.c^*|_1^{n'-1} + \sum_{d \in D} d.c$;
  $v^{nt}.\hat{t}|_{n'}^n \leftarrow \mathcal{T}$.MaxTimeOfSubFlows($S^{nt}$);
  $v^{nt}.\hat{c}|_{n'}^n \leftarrow \mathcal{T}$.UnionCostOfSubFlows($S^{nt}$);
  $v^{nt}.t^* = v^{nt}.t^*|_1^{n'} + v^{nt}.\hat{t}|_{n'}^n$;
  $v^{nt}.c^* = v^{nt}.c^*|_1^{n'} + v^{nt}.\hat{c}|_{n'}^n$;
  $v^{nt}.u^* \leftarrow$ compUtility($v^{nt}.t^*, v^{nt}.c^*, \theta_t, \theta_c, t^{max}, c^{max}$);
  $v^{nt}.\mathcal{D} \leftarrow D$; $v^{nt}.parent \leftarrow v$;
  $V^{nt} \leftarrow V^{nt} \cup v^{nt}$;
end for

sub-flow). This allows the algorithm to be more biased to cost than time when it computes $v^{nt}.u^*$ because the algorithm can make a *cost-biased decision* by expanding from the fastest sub-flow to minimize the slack time. Note that this cost-biased decision is not applied for sequential sub-flows. Then, the algorithm generates the next candidate vertices having a set of instantiations $v^{nt}.\mathcal{D}$. To obtain $v^{nt}.\mathcal{D}$, it generates all candidate instantiations into $\mathcal{D}^{nt}$ that are grouped by the next sub-tasks, and computes all combinations of instantiations using getCombinationsOfInstantiations($\mathcal{D}^{nt}$), where each instantiation d in a combination D is selected from each parallel sub-task.

$v^{nt}.u^*$ is computed with time and cost estimates (i.e., $v^{nt}.t^*$ and $v^{nt}.c^*$, respectively). To compute $v^{nt}.t^*$, the algorithm derives the maximum accumulated time $v^{nt}.t^*|_1^{n'}$ among sub-flows up to the current $d \in D$ and the maximum heuristic time $v^{nt}.\hat{t}|_{n'}^n$ among sub-flows, each of which is from a next sub-task to $sink$. Similarly, to compute $v^{nt}.c^*$, it derives the accumulated cost $v^{nt}.c^*|_1^{n'}$ up to d and the union of heuristic cost $v^{nt}.\hat{c}|_{n'}^n$ among sub-flows. Then, compUtility(.) computes $v^{nt}.u^*$ with $v^{nt}.t^*$ and $v^{nt}.c^*$.

Reducing the search space: The running time of the algorithm depends on the number of vertices (or instantiations) to be explored. The number of candidate vertices can exponentially increase, especially due to branch sub-tasks in the function getCombinationsOfInstantiations($\mathcal{D}^{nt}$) in Algorithm 3. We have developed a couple of techniques to reduce the number of candidate vertices from newly generated

candidate set before combining candidate vertices, while little impacting on the optimality.

First, the algorithm sorts instantiations in $W^{abt}.\mathcal{D}(s^{nt})$ by time estimate and cost estimate, separately and then, removes the instantiations, which have higher time estimate (and cost estimate) than given t^{max} and c^{max} constraints. Second, the algorithm figures out potential critical sub-flows in $\mathcal{T}$ by sorting sub-flows by time estimate and then, from *critical sub-flows*, it removes a certain portion of instantiations, which are relatively slow. For *non-critical sub-flows* in $\mathcal{T}$, it removes a certain portion of instantiations that are relatively expensive. The portion for pruning depends on the distribution of time (and cost) estimates. If the range of time (and cost) estimates is small (i.e., similar each other), the pruning rate increases since the selection may not significantly impact on optimality. Additionally, the portion rate increases as the search gets close to the $sink$ since the impact on optimality decreases.

IV. Experimental Evaluation

A. Evaluation Scenarios

To evaluate our approach, we have used one of our complex business processes (i.e., an abstract workflow designed for the integration of health-care and human resource business domains [2]) shown in Figure 1 in Section I. The abstract workflow consists of 39 sub-tasks, 8 branches and 9 merging sub-tasks. Then, we have simulated its deployment into a cloud environment for evaluating the optimality and the scalability of our algorithm while increasing the number of candidate sub-task instantiations.

In the real world, some sub-tasks should be instantiated with specific service applications and computing nodes in clouds (e.g., a search service using search engine and a data-intensive analytics requiring a specialized platform). However, in our experiments, we have assumed that a sub-task can be instantiated with any combination of computing nodes and services. We have assigned computation time, transfer delay, and cost for each instantiation by randomly selecting values in ranges around measured values. We consider two scenarios. First, we assume a heterogeneous setup, where computation time, transfer delay, and cost are widely different among instantiations. Second, we assume a homogeneous setup, where the variations of those values are small.

We have compared our heuristic algorithm, referred to as Adaptive A* (AA*), with four other algorithms in the context of optimality and scalability.

- **Exhaustive Search (ES):** This brute-force approach exhaustively explores all possible workflow deployments and selects the best one. The result has been used as the *ground truth* for the optimal utility value.
- **Greedy Search (GS):** This algorithm explores the graph along the *local optimal* paths without revisiting alternative paths to reduce slack times or data transfer delays.
- **Genetic Algorithm (GA):** This heuristic algorithm is considered in our comparison since it has been applied to the similar problem in many prior work (e.g., [18], [19], [5], [6]) with one we tackle in this paper. We have compared to the most recent work [6] that has integrated various techniques to improve the accuracy and scalability of GA. The overall approach is 1) evaluating the current set of possible workflow deployment candidates (i.e., population) based on the utility function; 2) reproducing a new population evolved from the current population using genetic operators such as the crossover of sub-trees caused by branches, random mutation, and elitism. The genetic operators select workflow deployments following a probability based on their evaluated values (i.e., fitness). This procedure is iterated until the utility value does not further improve in subsequent iterations. The size of population is an important parameter for the optimality, but there is a tradeoff between the optimality and the runtime [18]. Large population takes longer to converge but much higher quality. We have set two different population sizes for GA. GA-L has a large population size (i.e., 2000), and GA-S has a small size (i.e., 200).

[2]We intentionally hide the name of services due to confidentiality. Some portion of the workflow has been demonstrated in IEEE Service Cup 2012. Please refer to our prior paper [11].

B. HW and SW Setups

We have deployed our workflow scheduler into one of our production servers that has 16 cores with 3.1GHz CPU and 32GB memory. The current prototype has been developed with Java 1.6 and run on Fedora version 12. We have collected computation times and transfer delays of service applications used in our business process from Rackspace (one of public cloud providers). We have run three different types of virtual machines (VMs). The lowest-end VM ($0.06/hr) is configured with 1 vCPU with 1GB memory. The mid-end VM ($0.48/hr) has 4 vCPUs with 8GB memory, and the highest-end VM ($1.20/hr) has 8 vCPUs with 30GB memory. For the heterogeneous setup, the cost range is [0.06, 1.20], and the time range is $[x/4, 4x]$, where x is a measured time (i.e., computation time or transfer delay) for a service application in the mid-end VM. For the homogeneous setup, the cost range is [0.44, 0.52], and the time range is $[(9/10)x, (11/10)x]$.

C. Experimental Results

1) Optimality: We first show the optimality of our algorithm by comparing its quality with those of other three algorithms, where the quality indicates the normalized difference between the utility value computed by each algorithm and the ground truth computed by ES. These utility values have been computed in both heterogeneous and homogeneous setups separately, and in each setup, we have run algorithms with three different utility function parameters (i.e., time and cost weight ratios with θ_t and θ_c) – 0.5:0.5, 0.9:0.1, and 0.1:0.9. We have run each algorithm 30 times and obtained median, but we have run GS and ES just one time.

As shown in Figure 4, AA* outperforms other algorithms in all time:cost weight ratios. The optimality of GA-L has been slightly better than AA* in a few of runs. However,

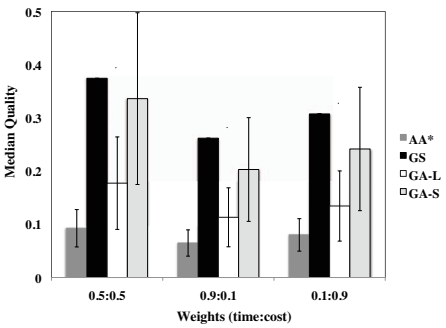

Fig. 4. Optimality comparison in heterogeneous setup (lower is better)

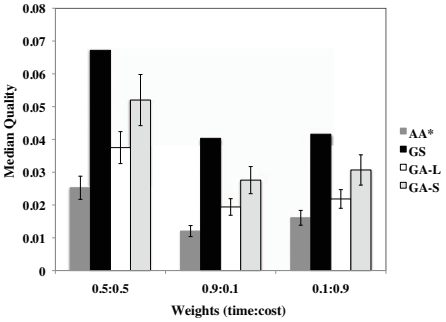

Fig. 5. Optimality comparison in homogeneous setup (lower is better)

its variance much higher than AA* and in many runs, our algorithm has better optimality. This is because the optimality of GA algorithm mainly depends on how well candidates in population are selected. If the algorithm selects the majority of candidates beyond the range of high quality utility values, it is likely to converge a value located outside of the range. AA* does not depend on such probability, but the utility value distribution of candidate instantiations. The optimality of GA-L is usually better than GA-S simply because GA-L starts with larger population size so that it has more chance to converge into better results. The optimality of AA* is almost similar across different time:cost ratios, while it is slightly better in biased time:cost ratios (i.e., 0.9:0.1 and 0.1:0.9) in other algorithms. In homogeneous setup, the optimality is much better than heterogenous setup as shown in Figure 5. This is because the range of utility values obtained from different workflow deployments is much smaller than heterogenous setup. This leads to smaller variances of all algorithms as well. In this setup, AA* still outperforms other algorithms.

2) Scalability: We have compared the scalability among algorithms by increasing the size of graph and measuring the runtime of each algorithm. Here, the size of graph is the sum of all possible instantiations of sub-tasks in the given workflow. In heterogeneous setup, as shown in Figure 6, AA* is scalable and close to GA-S and GS that have linear scalability. Especially, it is very close in the biased ratios (i.e., Figure 6 (b) and (c)). GA-S can converge very quickly because of its small population size, and GS is very scalable since it does not revisit other paths by ignoring minimal slack times and data transfer delays that our approach has considered. These features cause the low optimality as shown previously, although GA-S and GS is scalable. Meanwhile, GA-L shows the worst scalability. This is because it has a large population size so that it takes a long time to converge. As shown in Figure 6 (a), AA* is relatively less scalable in 0.5:0.5. This case makes AA* exploring more candidate paths than other two cases since utility values among instantiations can be similar by considering time and cost equally. This fact can seriously impacts on the scalability in homogeneous setup. However, AA* aggressively prunes out candidate instantiates based on their distribution as described in Section III-B. Figure 7 shows AA* has a reasonable scalability in this case as well.

V. Related Work

Similar to our approach, [20], [21] have dealt with parallel sub-flows using graph search techniques, but they have just attempted to discover semantic functional matching of user requests from web service repository when building workflows. Our approach has focused on non-functional aspects such as execution time, cost, and their tradeoff.

Meanwhile, [5] and [6] have integrated a cost function when scheduling business processes, but they have not precisely addressed the tradeoff between the end-to-end execution time and the accumulative service costs in a complex business processes that has a number of parallel sub-flows. By considering a critical path that represents the slow path in parallel sub-flows, our workflow scheduler can precisely address the tradeoff while deploying business processes in federated cloud.

Scheduling workflow for its optimal performance is known as an NP-complete problem. Many heuristic algorithms have been introduced to approximate the optimal performance. A* Prune algorithm [22] has been proposed to find K shortest paths in a given graph, subject to multiple constraints. Basically, the algorithm expands alternative paths from source to sink while pruning paths that do not meet constraints. However, this algorithm did not consider critical paths in parallel sub-flows that can impact on the optimal end-to-end execution time. Additionally, the goal of this algorithm is to obtain K paths that meet given constraints, while our algorithm aims to address the optimal time-cost tradeoff for given time and cost preferences. Our algorithm can return a best-effort optimal path even if it cannot meet constraints.

Another type of heuristic algorithm is based on the evolutionary approach to an optimal solution. It includes Simulated Annealing [23], Genetic Algorithm [19], [5], [6], Particle Swarm Optimization [24], [25], Ant Colony Optimization [26], and Artificial Bee Colony [27]. Specific approaches for these algorithms are different, but the overall approach is to randomly modify a current good solution and create a new better solution iteratively until it finds a reasonable solution. One of advantages of this approach is that it makes few or no assumptions about the problem. Hence, it can be applied to various problems including the optimal time-cost tradeoff. However, such heuristic algorithms may not guarantee a reasonably good solution in a tight time bound for complex workflows with parallel sub-flows.

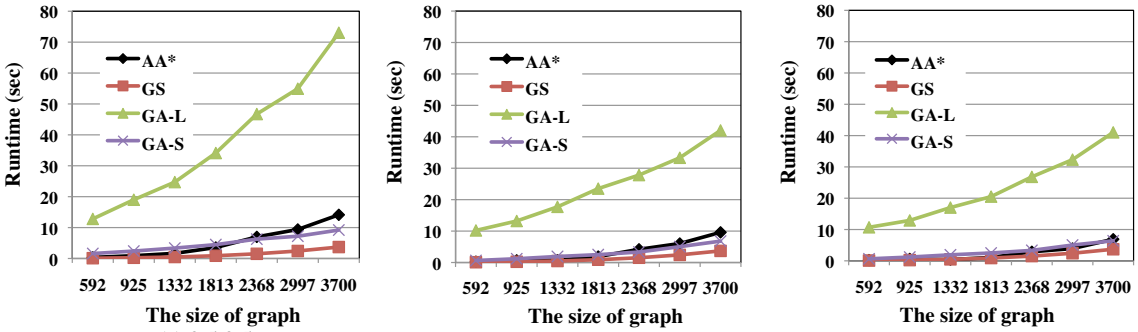

Fig. 6. Scalability comparison in heterogeneous setup with different time and cost weights

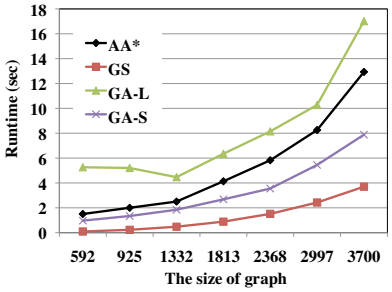

Fig. 7. Scalability comparison in homogeneous setup with 0.5:0.5 ratio

VI. Conclusion

This paper has tackled three specific challenges to achieve the optimal deployment of parallel service workflow. The explosion of available cloud services makes the selection of appropriate cloud services to be a non-trivial problem. Due to the tradeoff between service computation time, the execution cost, and the data transfer delay, searching an optimal balance is difficult. The complexity of workflows makes the search problem even more difficult. In this paper, we have described a workflow scheduler designed to address the tradeoff for recursively parallel workflows in an efficient way. Our experimental simulation has shown that our approach is scalable while providing a reasonable optimality.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A Berkeley view of cloud computing," in *Tech. Report University of California Berkeley*, 2009.

[2] H. Kim, S. Chaudhari, M. Parashar, and C. Marty, "Online risk analytics on the cloud," in *Proc. of IEEE/ACM Int. Symp. on Cluster Computing and the Grid*, 2009, pp. 484–489.

[3] H. Zhang, G. Jiang, K. Yoshihira, H. Chen, and A. Saxena, "Intelligent workload factoring for a hybrid cloud computing model," in *Proc. Congress on Services*, 2009, pp. 701–708.

[4] S. Kailasam, N. Gnanasambandam, J. Dharanipragada, and N. Sharma, "Optimizing service level agreements for autonomic cloud bursting schedulers," in *Parallel Processiong Workshop*, 2010, pp. 285–294.

[5] J. Yu, M. Kirley, and R. Buyya, "Multi-objective planning for workflow execution on grids," in *Proc. of Int. Conf. on Grid*, 2007, pp. 10–17.

[6] E. Juhnke, T. Dornemann, D. Bock, and B. Freisleben, "Multi-objective scheduling of BPEL workflows in geographically distributed clouds," in *Proc. of IEEE Int. Conf. on Cloud Computing*, 2011, pp. 412 –419.

[7] "JBoss jBPM," http://www.jboss.org/jbpm.

[8] "Activiti BPM platform," http://www.activiti.org/.

[9] "Oracle BPM suite 11g: BPM without barrier, 2010," http://www.oracle.com/us/technologies/bpm/bpm-without-barriers-wp-190949.pdf.

[10] H. Liu, Y. Charif, G. Jung, A. Quiroz, F. Goetz, and N. Sharma, "Towards simplifying and automating business process lifecycle management in hybrid clouds," in *Proc. of the IEEE International Conference on Web Services*, 2012, pp. 592–599.

[11] Y. Charif, H. Liu, A. Quiroz, and X. Liu, "Automating reusable workflow development from design to instantiation," in *Proc. of World Congress on Services*, 2012, pp. 369–376.

[12] D. Fudenberg and J. Tirole, *Game Theory, Chapter 1* MIT Press, 1983.

[13] R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality of A*," *Journal of ACM*, vol. 32, no. 3, pp. 505–536, 1985.

[14] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *Transactions on Autonomous and Adaptive Systems*, vol. 3, no. 1, pp. 1–39, 2008.

[15] D. Kusic, J. Kephart, J. Hanson, N. Kandasamy, and G. Jiang, "Power and performance management of virtualized computing environments via lookahead control," in *ICAC*, 2008, pp. 3–12.

[16] G. Box, G. Jenkins, and G. Reinsel, *Time Series Analysis: Forecasting and Control*, 3rd ed. Prentice Hall, 1994.

[17] G. Jung, M. Hiltunen, K. Joshi, R. Schlichting, and C. Pu, "Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures," in *ICDCS*, 2010, pp. 62–73.

[18] F. G. Lobo and C. F. Lima, "A review of adaptive population sizing schemes in Genetic Algorithms," in *Proc. of Workshop on Genetic and Evolutionary Computation*, 2005, pp. 228–234.

[19] J. Yu and R. Buyya, "Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms," *Sci. Program*, vol. 14, no. 3,4, pp. 217–230, 2006.

[20] M. Shiaa, J. Fladmark, and B. Thiell, "Incremental graph-based approach to automatic service composition," in *Proc. of IEEE Int. Conf. on Services Computing*, 2008, pp. 397–404.

[21] Y. Yan, B. Xu, and Z. Gu, "Automatic service composition using and/or graph," in *Proc. of the* 5^{th} *IEEE Int. Conf. on Enterprise Computing, E-Commerce and E-Services*, 2009, pp. 335–338.

[22] G. Liu and K. Ramakrishnan, "A*Prune: an algorithm for finding K shortest paths subject to multiple constraints," in *Proc. of the* 12^{th} *IEEE INFOCOM*, vol. 2, 2001, pp. 743–749.

[23] A. YarKhan and J. Dongarra, "Experiments with scheduling using simulated annealing in a grid environment," in *Proc. of the* 3^{rd} *Int. Workshop on Grid Computing*, 2002, pp. 232–242.

[24] S. Pandey, L. Wu, S. Guru, and R. Buyya, "A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments," in *Proc. of the* 24^{th} *IEEE Int. Conf. on Advanced Information Networking and Applications*, 2010, pp. 400–407.

[25] Q. Tao, H. Chang, Y. Yi, C. Gu, and Y. Yu, "QoS constrained grid workflow scheduling optimization based on a novel PSO algorithm," in *Int. Conf. on Grid and Cooperative Computing*, 2009, pp. 153–159.

[26] M. Dorigo, M. Birattari, C. Blum, M. Clerc, T. Sttzle, and A. Winfield, "Ant colony optimization and swarm intelligence," in *ANTS*, 2008.

[27] A. Banharnsakun, B. Sirinaovakul, and T. Achalakul, "Job shop scheduling with the best-so-far ABC," *Engineering Applications of Artificial Intelligence*, vol. 25, no. 3, pp. 583–593, 2012.