A Survey of Modern Scientific Workflow Scheduling Algorithms and Systems in the Era of Big Data

Junwen Liu*, Shiyong Lu*
Department of Computer Science
*Wayne State University**
Detroit, Michigan, USA
*Email: {junwen, shiyong}@wayne.edu**

Dunren Che†
School of Computing
Southern Illinois University†
Carbondale, Illinois, USA
Email: dche@cs.siu.edu†

***Abstract*—This paper provides a survey of the state-of-the-art workflow scheduling algorithms with the assumption of cloud computing being used as the underlying compute infrastructure in support of large-scale scientific workflows involving big data. The survey also reviews a few selected representative scientific workflow systems in light of usability, performance, popularity, and other prominent features. In contrast to existing related surveys, which most try to be comprehensive in coverage and inevitably fall short in the depth of their coverage on workflow scheduling, this survey puts an emphasis on the two dominant factors in workflow scheduling, the makespan and the monetary cost of workflow execution, resulted in a useful taxonomy of workflow scheduling algorithms as an additional contribution. This survey tries to maintain a good balance between width and depth in its coverage – after a broad review, it spotlights on selected top ten representative scheduling algorithms and top five workflow management systems leveraging cloud infrastructure with an emphasis on support for big data scientific workflows.**

***Keywords*-Workflow; Workflow Scheduling; Workflow Systems; Cloud; Constraints; Optimization.**

1. Introduction

In the last two decades, scientific workflow becomes the mainstream for empowering scientists to accelerate scientific discoveries in all fields of science. Traditionally, scientific workflows are described as directed-acyclic graphs (DAGs), in which nodes represent computational tasks and edges represent data dependencies among tasks. Many scientific experiments leverage scientific workflows to organize complex computations to process and analyze data. As more data is collected by various scientific experiments, scientific workflows become more and more data and computation intensive. To meet such needs, a number of workflow management systems have been developed over the past decade, including Pegasus [24], DATAVIEW [25], Kepler [26], Taverna [27], Swift [28], etc. They are extensively used by various research communities, covering astronomy, bioinformatics, ecology, computational engineering, etc. In order to facilitate users to design, execute and monitor scientific workflows throughout the whole workflow life-cycle, most *Scientific Workflow Management Systems (SWFMSs)* take a layered architecture that consists of four or more layers, e.g., a presentation layer, a workflow management layer, a task management layer, and an infrastructure layer, etc.

In order to handle the increasingly data-intensive scientific applications, various compute resources that facilitate disparate types of parallelism have been incorporated in scientific workflow execution environments, such as clusters, grids and clouds. Cloud computing is the recent trend in distributed scalable computing. It fulfills the vision to deliver on-demand, reliable, and scalable services over the Internet, with easy access to virtually infinite compute, storage and networking resources. Users can outsource complex tasks to very large data centers operated by numerous cloud providers and access their resources through services based on a convenient pay-as-you-go pricing model.

In this exciting era of big data, intelligent connectivity is quickly embracing the new advancement of IoT, Edge Computing and 5G networking. Meanwhile, providing efficient workflow scheduling and execution by novel scheduling algorithms in workflow systems remain an important and challenging issue. As large amounts of data are collected and need to be processed in real time in numerous scientific fields, scientists realize their increasing need for more advanced SWFMSs to accelerate their scientific discoveries under both budget and deadline constraints.

In this survey, we focus on reviewing the state-of-the-art workflow scheduling algorithms that were designed with particular consideration of supporting big data and leveraging cloud computing infrastructure for expedited execution of large-scale scientific workflows. In addition, the survey reviews five representative scientific workflow systems in light of prominent features, usability, performance, and popularity. Different from existing related surveys, which typically aim at making comprehensive reviews on SWFMSs, inevitably lacking the desired focus and depth in the core workflow scheduling algorithms, this survey particularly put its emphasis on workflow scheduling algorithms leveraging the cloud infrastructures to achieve two dominant objectives, minimizing both the makespan and the monetary cost of workflow execution. A secondary focus is put on reviewing representative workflow management systems. Therefore the survey achieves an enjoyable balance between breadth

Table I: A summary of scheduling algorithms and systems.

Category 1	**IC-PCP [1]**, **LPOD [2]**, **iCATS [3]**, WRPS [4], Heuristic GA [5], CGA [6], BORRIS [7], CEGA [8]
Category 2	**BAGS [9]**, **Scheduling-first [10]**, PCP [11], BARENTS [12]
Category 3	**SHEFT [13]**, **MOHEFT [14]**, HEFT [15], CPOP [15], N-DNSGA-II [16], EMS-C [17], MOELS [18]
Category 4	**DPDS [19]**, **SPSS [19]**, **HPSO [20]**, DBCS [21], BDAS [22], DBWS [23]
Systems	**Pegasus [24]**,**DATAVIEW [25]**, **Kepler [26]**, **Taverna [27]**, **Swift [28]**, Galaxy [29], VisTrails [30], TimeStudio [31], KNIME [32], Pipeline Pilot [33], ClowdFlows [34], TextFlows [35], VIEW [36], U-Compare [37], SecDATAVIEW [38]

Table II: Comparison on representative scheduling algorithms.

Algorithms	Constraints	Optimization Objectives	Types	D/S	Time Complexity
IC-PCP	Deadline	Monetary cost	Path-based	Static	$O(n^2)$
LPOD	Deadline	Monetary cost	Path-based	Static	$O(n^2 + k^n)$
iCATS	Deadline	Monetary cost	Workflow-based	Static	$O(K \times NlogN)$
BAGS	Budget	Makespan	BoT-based	Dynamic	$O(n \times O(milp))$
Scheduling-first	Budget	Average makespan	Task-based	Static	$\mathbf{O}(\Sigma_i N_i \times k)$
SHEFT	No Constraints	Monetary cost, makespan	Task-based	Static	$O(n^2 + n \times k)$
MOHEFT	No Constraints	Monetary cost, makespan	Task-based	Static	$O(e \times k \times K)$
DPDS	Deadline, Budget	# of workflows	Task-based	Dynamic	$O(n)$
SPSS	Deadline, Budget	# of workflows	Task-based	Static	$O(n^2)$
HPSO	Deadline, Budget	Monetary cost, makespan	Workflow-based	Static	$O(K \times m \times NlogN)$

Table III: Comparison on representative workflow systems.

Systems	Domains	Execution Envs	SaaS	UI	API	Language support	Open Source
Pegasus	Scientific computing	local, cluster, grid, clouds	No	Cmd	Yes	Java, Python, Perl	https://pegasus.isi.edu/downloads/
DATAVIEW	Big data analytics	local, clouds	Yes	Web	Yes	Java, Python	https://github.com/shiyonglu/DATAVIEW
Kepler	Big data analytics	local, clusters, web services	No	Desktop	Yes	Java, R	https://kepler-project.org/
Taverna	Bioinformatics	local, web services	No	Desktop, Cmd	Yes	Scufl2	https://taverna.incubator.apache.org/
Swift	Scientific computing	local, cluster, grid, clouds	No	Cmd	Yes	Swift	https://github.com/swift-lang

and depth in its review of both the workflow scheduling algorithms and SWFMSs.

In summary, this survey makes the following contributions:

1) We provide an overview on workflow scheduling, especially with regard to the parallelization techniques used and the challenges regarding workflow scheduling in the cloud (The review can be potentially useful as a quick and concentrated introduction to the state-of-the-art workflow scheduling algorithms for those relevant readers and researchers).
2) We propose a new practical taxonomy of workflow scheduling algorithms in the cloud and identify four categories of scheduling algorithms by considering two dominant factors, makespan and monetary cost of workflow execution (see Table I). A survey with in-depth comparison of selected representative scheduling algorithms is made, as summarized in Table II.
3) We provide a condensed survey on five representative scientific workflow systems regarding their respective prominent features, scientific impact, literature citations and open source accessibility. We compare them in terms of their targeted application domains, execution environments, and other features such as third-party API support and programmatic language support. A side-by-side comparison is made in table III.

2. Workflow Scheduling: Problem, Challenges, and Taxonomy

This section briefly describes the problem, challenges, and taxonomy of workflow scheduling.

A. Workflow Scheduling

Workflow scheduling is the procedure of mapping workflow tasks to computing resources (e.g. VMs) needed for

the tasks' execution. The goal of scheduling is to get an efficient scheduling plan (SP) that optimizes certain objectives, such as minimizing makespan and/or monetary cost. Workflow scheduling algorithms can be categorized into three types: *static*, $dynamic$ and $hybrid$, of which *static* algorithms generate schedules statically – i.e., before workflow execution starts; *dynamic* algorithms flexibly intermingle the scheduling and execution steps; and *hybrid* algorithms typically construct a (preliminary) schedule first, then start its execution, and continue to adapt/optimize (parts of) the schedule during execution based on newly available dynamics of the actual execution of the workflow.

B. Challenges of workflow scheduling in clouds

We identify three main challenges for running workflows in the cloud: 1) **Workflow scheduling strategy:** since we are using a utility based pricing model, the overall cost and makespan are two conflicting factors when running workflows in the cloud, a good scheduling strategy should achieve a good trade-off between these two factors. Pursuing a better (or perfect) trade-off by spending a minimum scheduling time will remain to be a big challenge. 2) **Resource provisioning and de-provisioning strategy:** a good provisioning and de-provisioning strategy is needed to decide how many and what types of VMs need to be provisioned and de-provisioned at any specific time. 3) **Dynamic cloud computing environments:** the cloud is a dynamic computing environment in which the performance of VMs and networks can vary from time to time. A good workflow scheduling algorithm must appropriately address such dynamics of the underlying cloud infrastructure during workflow execution.

C. A taxonomy of workflow scheduling algorithms in clouds

In the following, based on the two dominant factors, time and monetary cost, we differentiate four scenarios of workflow scheduling and accordingly classify existing cloud-based workflow scheduling algorithms into four categories:

- ***Category 1****: Given a deadline, minimize the monetary cost.* Usually the scheduling algorithms in this category focus on how to distribute the time constraints of a given deadline among the various workflow tasks, so that when each task completes within its sub-deadline, the whole workflow is guaranteed to finish within the given overall deadline. The scheduling goal is more focused on minimizing the monetary cost of executing the workflow while satisfying the given deadline.
- ***Category 2****: Given a budget, minimize the makespan.* The scheduling algorithms in this category focus on how to allocate the overall budget to sub-workflows or tasks so that if each sub-workflow or task can be scheduled on resources within its allocated sub-budget, the whole workflow can complete within the overall budget. The scheduling goal is more focused on minimizing the makespan of the workflow's execution while satisfying the given budget.
- ***Category 3****: Given no constraints, minimize the makespan and/or the monetary cost.* Essentially, this is a mono-objective or multi-objective optimization problem. For the later, one approach is to use a traditional multi-objective optimization technique to solve this problem. A second approach is to convert the problem into a mono-objective optimization problem by aggregating two objectives into one.
- ***Category 4****: Given both deadline and budget constraints, optimize some other performance metrics.* With both deadline and budget constraints given, the scheduling algorithms in this category focus on optimizing one or several performance metric(s), such as makespan, monetary cost, or both, or success rate (the chance that the actual schedule will succeed within the constraints).

3. Workflow scheduling algorithms in the cloud

In this section, we select several representative scheduling algorithms from each category per their performance and impact. Based on the scheduling approaches, we introduce four types of scheduling algorithms:

- **Task-based** (scheduling task-by-task): In the literature, they are also called *list-based scheduling*. In these algorithms, tasks are ordered based on some priority ranking and then a scheduling decision is made for each task in that order.
- **Path-based** (scheduling path-by-path): In these algorithms, a workflow is partitioned into paths based on some criteria and then a scheduling decision is made for each path.
- **BoT-based** (scheduling BoT-by-BoT): In these algorithms, a workflow is partitioned into BoTs (Bag of Tasks) such that each BoT is a set of tasks that have no data dependencies among them, and a scheduling decision is made for each BoT.
- **Workflow-based** (scheduling workflow-by-workflow): In these algorithms, all tasks in a workflow are simultaneously scheduled and then the schedule for each task is improved using some procedure to improve the quality of the global workflow schedule.

There exists two types of billing models that are applicable to workflow scheduling in the cloud:

1) **Fine-grained billing model**: a VM instance is charged precisely based on the real usage time of the instance and the unit price, usually $/minute, of the VM.
2) **Coarse-grained billing model**: A VM instance is charged based on the number of billing cycles (τ minutes/cycle) leased for the instance and the per-cycle price of the VM. Partial usage of a billing cycle is charged at the price of a full billing cycle.

We will describe each algorithm in terms of its category, type, constraints, optimization objectives, assumptions of the cloud computing model (VM types, network, storage, and billing), and the main idea of the algorithm. A comparison of these algorithms is summarized in Table II. We also provide a video presentation for each algorithm at https://www.youtube.com/user/shiyonglu.

A. Category 1: given a deadline, minimize the monetary cost

1) **IC-PCP:** IC-PCP [1] is a typical path-based static workflow scheduling algorithm of Category 1. It assumes a homogeneous network in its cloud computing model, so that the bandwidth between two arbitrary virtual machines is similar. On the other hand, the computing environment is heterogeneous - every VM might be of different type, with different computation speed, size of memory and IO performance. It assumes a coarse-grained billing model. The performance metric for this algorithm is the total cost of the whole workflow execution. The time complexity of this algorithm is $O(n^2)$, where n is the number of tasks.

The main idea of IC-PCP is to identify the so called *partial critical path* p of a node t_i and then schedule all the tasks on p to one VM instance. Initially, the EST (Earliest Start Time) and LFT (Latest Finish time) (also called the sub-deadline) for each task are estimated as follows:

$$EST(t_i) = \max_{t_p \in t_i's\ parents} \{EST(t_p) + MET(t_p) + TT(e_{p,i})\} \quad (1)$$

$$LFT(t_i) = \min_{t_c \in t_i's\ children} \{LFT(t_c) - MET(t_c) - TT(e_{i,c})\} \quad (2)$$

where $MET(t_i)$ is the execution time of t_i on the fastest machine and $TT(e_{p,i})$ is the data transfer time from parent task t_p to child task t_i. The algorithm identifies the partial critical path of a task t_i as follows: the critical parent of t_i is the parent t_p such that 1) it is unscheduled, and 2) it has the latest data arrival time, that is, the largest $EFT(t_p) + TT(e_{p,i})$; the partial critical path of t_i is identified recursively (backward) via the critical parent t_p of t_i, the critical parent of t_p, and so on. Once a partial critical path p is identified, it is then scheduled on the cheapest virtual machine instance such that each task in p will satisfy its sub-deadline. After p is scheduled, the EST, EFT (Earliest Finish Time) and LFT of each task in p is adjusted, and the EST of each descendant of p and the LFT of each ancestor of p are adjusted accordingly. The procedure of identifying and scheduling a partial critical path is invoked from the exit task node t_{exit} backward recursively. As a result, the scheduling of each partial critical path improves the estimate of $ESTs$ and $LFTs$ of the remaining unscheduled tasks, and thus improves the performance of the schedule of the remaining tasks.

2) **LPOD:** LPOD [2] is another typical path-based static workflow scheduling algorithm of Category 1. It assumes a homogeneous network environment and a heterogeneous cloud computing environment. It assumes a coarse-grained billing model. The performance metric adopted by LPOD is the C score, which ranges from 0 to 1 and combines two performance factors (one for monetary cost and one for makespan) into one metric. If the makespan is smaller than the deadline δ, a reward between 0.0 to 0.5 is added to C, otherwise a penalty between 0.0 to 0.5 is subtracted from C. The definition of C is as follow:

$$C = \begin{cases} 0.5 + 0.5 * \frac{maxcost - cost}{maxcost - mincost}, & if\ makespan \leq \delta \\ 0.5 - 0.5 * \frac{makespan - \delta}{maxmakespan - \delta}, & otherwise \end{cases} \quad (3)$$

where $maxcost$ ($mincost$) is the monetary cost of running the whole workflow in one instance of the most (least) expensive VM type and $maxmakespan$ is the makespan of running the whole workflow in one instance of the slowest VM type. The time complexity of LPOD is $O(n^2 + k^n)$, where n is the number of tasks in the workflow and k is the number of VM types.

The main idea of LPOD is as follows. First, calculate the priority rank for each task t_i, which measures the longest path from t_i to the exit node t_{exit}, and is defined as follow:

$$Pri(t_i) = \begin{cases} 0, & if\ t_i = t_{exit} \\ \overline{ET}(t_i) + \max_{t_j \in t_i's\ children} (\overline{DTT}(D_{i,j}) + Pri(t_j)), & otherwise \end{cases} \quad (4)$$

where t_j is the child task of t_i, $\overline{ET}(t_i)$ denotes the execution time of task t_i and $\overline{DTT}(D_{i,j})$ denotes the data transfer time between task t_i and t_j in the workflow. Second, identify the unvisited task t_i that has the highest priority rank and then recursively select its critical child - the unvisited child that has the highest priority rank, to form a partial critical path. This procedure is repeated until all tasks are visited. As a result, the whole workflow is partitioned into a list of partial paths PP. Third, for each path $P_i \in PP$, schedule as many tasks as possible in the prefix of P_i to existing VM instances, and then use a dynamic programming technique to schedule the remaining tasks in P_i to new VM instances, with the goal of minimizing the monetary cost while satisfying each task's sub-deadline. After each P_i is scheduled, the EST, EFT and LFT for each task in P_i will be adjusted, then the EST for each descendant of P_i and the LFT for each ancestor of P_i will also be adjusted. In contrast to IC-PCP, which always assigns a path to one VM instance, LPOD can assign a path to one or more VM instance(s), resulting in a more optimized solution.

3) **iCATS:** iCATS [3] is a workflow-based static workflow scheduling algorithm of Category 1. It assumes a homogeneous network environment where data transfer rates among all virtual machine types are fixed, and a heterogeneous cloud computing environment where different VM types can be utilized for computations. It assumes a coarse grained billing model. The performance metric/fitness function for this algorithm is the C score, which combines

monetary cost and makespan into one metric, as defined in equation (3). The time complexity of this algorithm is $O(K \times NlogN)$, where K is number of iterations, N is the number of solutions that is kept in the comprehensive elite.

The main idea of iCATS is to adopt an improved version of Cultural Algorithm (CA) [39] as a knowledge intensive evolutionary search process, to synthesize a globally optimal solution called *the comprehensive elite*, and to exploit multiple knowledge sources to collaborate in the generation of a new population of individuals called *the population space*. In contrast to traditional evolutionary algorithms, which have no implicit or explicit mechanism for storing and transferring the knowledge from one generation to another, CA provides an explicit mechanism called *the Belief Space*, for storing, evolving and transferring the knowledge. It also supports dual inheritance in which *the Population Space* and *the Belief Space* are updated at each step based upon feedback from each other. iCAT first constructs a *Belief Space* by ranking the current population set based on their fitness values, and select the best solutions and update *the Normative Knowledge* (lower bound and upper bound of fitted VM instance numbers for each task). Second, it will construct *the comprehensive elite* by ranking the normalized frequency of each VM for specific task (similar to a voting process), thus the VM with the highest normalized frequency will be selected and assigned to the corresponding task in *the comprehensive elite*. Third, it generates a new population set by searching within *the Normative Knowledge* range in the *Belief Space* and apply crossover and mutation to randomly selected individuals in the population set. Finally, the *comprehensive elite* is added to the next generation. The above steps are repeated until a termination condition is met. The best solution as the final schedule is returned by iCATS.

B. Category 2: given a budget, minimize the makespan

1) **BAGS:** BAGS [9] is a typical BoT-based dynamic workflow scheduling algorithm of Category 2. BAGS assumes a homogeneous network model, a heterogeneous cloud computing model and a centralized data storage model with no monetary cost associated with data transfer. It uses a coarse-grained billing model with a billing cycle of τ minutes. A provisioning delay is considered for each VM instance. The performance metrics for this algorithm are cost/budget and makespan. The first metric tells the degree of budget constraint satisfaction, with $<= 1$ being full satisfaction. The later is the main goal for optimization. The time complexity of this algorithm is $O(n \times O(milp))$, where n is the number of tasks and $O(milp)$ is the complexity of mixed integer linear programming, which is highly dependent on the input of fine-grained constraints.

The main idea of BAGS is as follows. First, the workflow is partitioned into three sets of BoTs: BoT_{hom}, a set of bags of homogeneous tasks; BoT_{het}, a set of bags of heterogeneous tasks; and BoT_{sin}, a set of bags containing a single task. Second, the overall budget is distributed over all BoTs as sub-budgets as follows: initially, pick the same VM type for all tasks and that is as fast as possible such that the total cost of running the workflow on the VM will not exceed the given budget β, and then if possible, upgrade each BoT to the next fastest VM type within the total budget β; the monetary cost of running each BoT on the corresponding VM type is assigned as the sub-budget for that BoT; the remaining spare budget is then proportionally allocated to each BoT as new VM provisioning budget so that when necessary a new VM instance can be provisioned . Third, a ready queue is used to place the tasks that are ready to execute. During each scheduling cycle, for the first task t in the queue, the bag bot that t belongs to is identified, and then check whether there exists a resource provisioning plan for bot. If not, then a similar budget redistribution procedure will be applied, and a resource provisioning plan will be generated for bot using a MILP (Mixed Integer Linear Programming) solver based on the sub-budget for bot. Then all tasks in bot will be scheduled based on the generated resource provisioning plan. In particular, t will be scheduled on an idle VM instance either from the bot specific idle VM pool or from the general idle VM pool when possible, otherwise, a new VM instance will be provisioned according to the resource provisioning plan for bot, and t will be scheduled on this newly provisioned VM instance. If there is no idle VM instance and the spare budget is not sufficient for provisioning a new VM instance, then t is put back to the queue for the next cycle of scheduling. Both resource provisioning and scheduling are done at run time.

2) **The Scheduling-first algorithm:** The scheduling-first algorithm [10] is another typical task-based static workflow scheduling algorithm of Category 2. It assumes 1) each workflow/job with a priority, 2) every task inherits the same priority from its workflow, 3) a cloud storage model that stores intermediate data products in a centralized storage and 4) a heterogeneous computing model in which each VM type has a price and provisioning delay. The goal is to minimize the weighted average *turnaround* time (the weight is the priority of tasks), which is subject to the constraint that the cost of all VM instances added up is smaller than the budget set for any particular time point. The weighted average makespan, $\Sigma_j makespan_j * priority_j / \Sigma_j priority_j$ is used as the performance metric, in which $makespan_j$ and $priority_j$ are the makespan and priority of workflow w_j, respectively. This algorithm is designed for service providers that need to schedule a stream of workflows on an online cloud service under a budget in the form of $/hour instead of $/workflow. This form of budget restricts the types and numbers of VM instances that can be allocated at each time point. The time complexity of this algorithm is $O(\Sigma_i N_i \times k)$, where N_i is the number of tasks in workflow W_i, and k is the number of VM types.

The main idea of the scheduling-first algorithm is as

follows. First, the overall budget B is proportionately distributed over workflows based on their priorities, resulting in a sub-budget B_j for each workflow W_j. Second, schedule each workflow W_j based on its sub-budget B_j. More specifically, for each ready task t_i of W_j, schedule t_i on the fastest VM instance without exceeding B_j and then adjust B_j. After scheduling W_j, any remaining budget in B_j is returned back to B. Finally, any remaining budget in B will be used to schedule the ready and wait for running tasks with the highest priority on the fastest VM instance without exceeding B.

C. Category 3: given no constraints, minimize both the makespan and/or monetary cost

1) **SHEFT:** The SHEFT [13] algorithm is a simple but effective task-based static algorithm of category 3, which is an improved version of the HEFT algorithm [15] by considering the new opportunity of allocating VMs from the cloud. It assumes 1) a heterogeneous network model between VMs in different resource clusters, 2) a homogeneous network model between VMs in the same resource cluster, so the bandwidth between two arbitrary virtual machines in the same resource cluster is similar, and 3) a heterogeneous computing environment with different VM types that can be utilized for task execution. SHEFT has no specific billing model since this algorithm does not consider the monetary cost. The performance metric for this algorithm is the *makespan* of the whole workflow execution. The SHEFT algorithm has an $O(n^2+n\times k)$ time complexity for n tasks and k VM types.

The main idea of SHEFT is to schedule first the task that can potentially affect the makespan the most, and aims to assign each task to a VM instance so that the task can complete as soon as possible. First, calculate the priority rank for each task t_i, which measures the longest path from t_i to the exit node t_{exit}, as defined in equation 4. Second, let PL be a queue that contains all the tasks of the workflow in descending order of their priority ranks and R be the current VM pool, which is set to be an empty set initially. Each task t_i in PL is then scheduled in that order by assigning t_i to a VM instance that results in the least EFT: pick up a VM instance r from R such that $EFT(t_i, r)$ is minimized, and then select a VM type V from the cloud such that $EFT(t_i, V)$ is minimized; if $EFT(t_i, r) < EFT(t_i, V)$, then t_i will be scheduled on the existing VM instance r, otherwise, a new VM instance v of type V will be provisioned for t_i to be scheduled on. After that, v will be added to R. Finally, the de-provisioning time for each VM instance in R can be calculated as the EFT of the last task in each VM. In contrast to HEFT, which only considers a fixed resource pool R, SHEFT considers both R and new VM instances from the cloud, as a result, the schedule produced by SHEFT is always better than or equal to the schedule that is produced by HEFT.

2) **MOHEFT:** MOHEFT [14] is a task-based static workflow scheduling algorithm of Category 3, aiming to minimize both the makespan and the monetary cost. It assumes a homogeneous cloud network model so that the bandwidth between two arbitrary virtual machines is similar, and a heterogeneous computing environment that every VM might be of different type. It assumes a fine-grained billing model. The performance metric for this algorithm is called *crowding distance* that calculates the hypervolume that enclosed the maximum of ROI (region of interest). The time complexity is $O(e\times k\times K)$ for e edges, k VM instances and K iterations.

The main idea of MOHEFT is that the algorithm returns a set of trade-off solutions, called the *Pareto front*, such that each solution in this set cannot be further improved in any of the considered objectives without degrading another objective. A user can then select one of the solutions based on some ad-hoc criteria. MOHEFT extends the classic HEFT algorithm [15] (which aims to minimize only the makepan on a fixed set of resources) in three ways: 1) to achieve a trade-off between makespan and monetary cost; 2) to return the Pareto front; and 3) to support cloud computing environments. Initially, the Pareto front is initialized to N empty solutions, $S_1, \cdots, S_N$, where N is the size of the Pareto front, a parameter of the MOHEFT algorithm. The tasks of the workflow are ordered into a list L in decreasing order of priority rank (aka B-rank) - which measures the longest distance from a task t_i to the exit node t_{exit}. For each $t_i \in L$ in that order, it considers to schedule t_i on each VM instance (v_1, $\cdots$, v_k), thus extending each S_N with k solutions, resulting in S', a set of $N*k$ intermediate solutions. Then the Pareto front S is reassigned with the top N solutions in S' based on the metric of crowding distance [14]. To support cloud computing environments, the algorithm considers the number of resources as $k = m * I$ where m is the maximum number of VM instances that a customer can acquire for each VM type and I is the number of VM types offered by the cloud provider. The algorithm then invalidate all intermediate solutions in S' in which the number of VM instances for some VM type exceeds m. This procedure is repeated until each $t_i \in L$ is considered, and then the final Pareto front is returned.

D. Category 4: Given both deadline and budget constraints, optimize the success rate or other metrics

1) **DPDS:** DPDS [19] is a task-based dynamic workflow scheduling algorithm of category 4. The goal of DPDS is by a given budget and deadline, to schedule as many workflows as possible. DPDS assumes that all the virtual machine are the same, the transfer time from one task to another is always fixed, and every workflow has a different priority in that the workflows comes earlier receive higher priorities than the ones come afterwards. The performance metric of this algorithm is an exponential scoring function defined

as $Score(e) = \sum_{\omega \in Completed(e)} 2^{-Priority(\omega)}$, which gives out a value ranging from 0 to 1 and combines: 1) the amount of completed workflows within the budget before the deadline, and 2) the priority (or how big these workflows are) into one metric. The time complexity of this algorithm is $O(n)$, where n is the number of tasks in the ensemble.

The main idea of DPDS is to separate the procedure of resource provisioning and de-provisioning ($Algorithm1$ [19]) from the procedure of dynamic workflow scheduling ($Algorithm2$ [19]) and run them in parallel. Initially, N_{VM} VM instances will be provisioned, where N_{VM} is the maximum number of VM instances that can be leased up to the deadline D without exceeding the budget B. Then, $Algorithm1$ will be executed periodically to either terminate existing VMs or provision new VMs based on deadline and budget constraints as well as VM utilization. More specifically, DPDS will terminate some VM instances whenever the remaining budget cannot sustain all existing VMs or the deadline is exceeded. The number of VMs that will be terminated is determined by the remaining budget. Without exceeding the budget, in each cycle of execution, DPDS will allocate a new VM whenever the current VM pool is overutilized ($> u_h$), or terminate half of the idle VM instances whenever the current VM pool is underutilized ($< u_l$). In the meanwhile, $Algorithm2$ is executed to schedule and execute workflow tasks in the workflow ensemble based on their priorities. A priority queue is used to maintain the ready tasks from different workflows based on their priorities. A task becomes ready when all its parents complete their executions. Each task in the priority queue is then scheduled on an idle VM instance until either there is no more idle VM or the priority queue is empty. Next, it waits for one task to complete its execution and return its VM instance to the idle VM pool, and put new ready children tasks to the priority queue. The same scheduling procedure is repeated until the deadline is exceeded or all workflows complete. The budget constraint is checked only by $Algorithm1$ (not by $Algorithm2$), while both algorithms check the deadline constraint.

2) **SPSS:** SPSS [19] is a task-based workflow static scheduling algorithm of category 4. By given budget and deadline, the goal of SPSS is to schedule as many workflows as possible. It has the same assumption of cloud network and computing environment as DPDS. A workflow that comes earlier in the priority queue has a higher priority than those come afterwards. The performance metric of this algorithm is the same exponential scoring function used for DSPS. The time complexity of this algorithm is $O(n^2)$, where n is the number of tasks in the ensemble.

The main idea of SPSS is as follow: workflows are scheduled based on priorities, one after another until all workflows are scheduled or the budget is exceeded. For each workflow w, first, DPSS distributes the deadline d over tasks in w by calculating the so called $slacktime$ by $ST(w) = d - CP(w)$, where $CP(w)$ is the execution time of the longest path in w, then distribute $ST(w)$ over w level by level. In particular, the slack time for each level l, $ST(l)$, is calculated as follows:

$$ST(l) = ST(w)\left[\alpha * \frac{N(l)}{N(w)} + (1-\alpha) * \frac{R(l)}{R(w)}\right] \quad (5)$$

where $N(w)$ is the number of tasks in workflow w, $N(l)$ is the number of tasks in level l, $R(w)$ is the total runtime of all tasks in workflow w, $R(l)$ is the total runtime of all tasks in level l, and α is a parameter between 0 and 1 that causes more slack time to be given to levels with more tasks (large α) or to levels with more runtime (small α). Second, the sub-deadline for each task t, $DL(t)$ is calculated based on the level l of t and $ST(l)$. Finally, SPSS will schedule every task according to its deadline in topological order. SPSS will schedule a task t on an existing VM instance that minimizes the cost while meeting the sub-deadline $DL(t)$; otherwise, schedule t on a new VM instance.

3) **HPSO:** HPSO [20] is also a typical workflow-based workflow scheduling algorithm of Category 4. The HPSO algorithm is based upon the multi-objective PSO and BDHEFT algorithms to solve a multi-objective workflow scheduling problem in the cloud. It adopts the Pareto optimal set, which is considered to be a set of potential solutions that are optimal for multiple objectives. Any solution in the Pareto optimal set is not dominated by other solutions in the set. HPSO assumes a heterogeneous computing environment and all VMs are in same physical region such that costs of data storage and data transmission can be assumed zero. It also assumes the bandwidths between arbitrary VMs are equal, and the time to transmit data between different VMs is considered. It assumes a coarse-grained billing model. The performance metrics for this algorithm include a 1) *fitness function*, $fitness = \alpha * time + (1-\alpha) * cost$, where α is the cost-time balance factor in the range of [0,1], to evaluate feasible solutions ***iff*** budget and monetary constraints are met; 2) *Generational Distance (GD)* as convergence metric, to evaluate the quality against true front $P*$, the true front $P*$ is obtained by merging solutions of algorithm over 20 runs; and 3) *Spacing metric* to evaluate diversity among solutions. Small values of *GD* and *Spacing metrics* are desirable in evolutionary algorithm. The time complexity of the HPSO algorithm is $O(K \times m \times NlogN)$, where K is number of iterations, m is number of objectives, N is number of solutions kept in the Pareto front.

The main idea of HPSO is as follow: First, it initializes the population consisting of N swarm particles (solutions) by randomly assigning the workflow tasks over the available cloud resources, and insert the result of the BDHEFT algorithm as one swarm particle. Second, based on the fitness function, solutions are evaluated and sorted on the basis of non-dominance order in elite archive for each solution as $A_{(i)}$. HPSO introduces an extra diversity parameter $I(y)$ to

order solutions that are with same dominance values. Third, through binary tournament selection in the elite archive, it initializes the global best position $gbest_{(i)}$ for $particle_i$, then updates the velocity and position for $particle_i$, and mutates the particle position in the adaptive possibility margin (the possibility margin turns to decrease as the number of iterations grows). Lastly, the particles in the population are evaluated to update the elite archive with the best N non-dominant solutions. The above procedure is repeated K iterations, in which K is defined by the user.

4. Representative Scientific workflow management systems

A scientific workflow management system provides a platform for domain scientists to compose and execute scientific workflows, which are pipelined series of computational and/or data processing tasks designed to solve complex computation-intensive scientific problems. Scientists can remotely collaborate on complex scientific projects based on scientific workflow platforms through GUI or command line (CMD) tools.

In this section, we provide a survey on five representative workflow management systems. They were selected due to their respective outstanding features. For example, Pegasus contributed to LIGO (Laser Interferometer Gravitational wave Observatory) that successfully helped detect the gravitational wave - a discovery that won the Noble prize; DATAVIEW manifested the notion of Workflow-as-a-Service (a special kind of SaaS) that allows users to utilize the system through the DATAVIEW website without the need to download and install the system; Kepler and Taverna both provide highly intuitive client-side UIs that ease workflow construction and execution, while Swift comes with a scripting based language tool that allows users to use C-like syntax to enact rapid applications of workflows involving big data. All these five selected systems are open-source and can be freely downloaded from their respective project websites (URLs are provided in table III for readers' convenience). We particularly address them in terms of their targeted application domains, execution environments, and other features such as third-party API support, programmatic language support, etc.

A. Pegasus

The Pegasus [24] workflow management system encompasses a set of technologies that facilitate scientific workflow application execution. Pegasus was designed to manage workflow execution on potentially distributed data and compute resources, in close collaboration with domain scientists. Pegasus workflows are based on Directed Acyclic Graphs (DAG), a model that has been commonly assumed by various scientific workflow management systems. Pegasus allows a node in a workflow DAG be a sub-DAG, which facilitates composition of very large workflows in the scale of millions of task nodes. In Pegasus, tasks exchange data between machines in the form of files, and workflow execution can be arranged to take place in a local or remote cluster, or in a grid or cloud. User interaction with Pegasus is through either command line commands or API interfaces. Pegasus provides programmatic API in python, Java and Perl for workflow generation in the form of DAX (or DAG in XML). The system also keeps variety of catalogs in order to support workflow optimization.

Pegasus is open-source. It has contributed to the LIGO software infrastructure and executes the main analysis pipelines of LIGO to detect the gravitational wave. Pegasus uses HTCondor as its workflow engine and scheduler, and can be setup on distributed or cloud environments. In Pegasus, graph transformations and optimizations are performed during *mapping* when a workflow is mapped onto a distributed environment before its *execution*. Optimizations is also performed during run-time by interleaving *mapping* and just-in-time planning. In order to improve reliability of workflow execution, during run-time, Pegasus performs actions such as job retry and failed workflow recovery .

B. DATAVIEW

DATAVIEW [25] is a generic scientific workflow management system. The applications of DATAVIEW range from machine learning, medical image analysis, bioinformatics, to automotive data analysis, etc. DATAVIEW is also based on DAG and adopts a layered architecture design that includes a presentation layer, a workflow management layer, a task management layer, and an infrastructure layer. DATAVIEW features a user-friendly Web portal for workflow creation and execution, and workflow execution can be flexibly arranged to run locally or on a cloud platform such as AWS. DATAVIEW adopts a master-slave deployment architecture and supports fast provisioning of virtual machines through VM images created and saved on AWS. The DATAVIEW VM images include the DATAVIEW kernel that schedules and executes workflows. With a developer-friendly Java API, DATAVIEW supports programmatic workflow development through Java and Python. DATAVIEW seamlessly integrates Dropbox as optional storage capacity for feeding workflow input and storing workflow output products.

DATAVIEW is open-source. In addition to local installation, it can be used as a SaaS (Software-as-a-Service) from www.dataview.org without download and installation of the system. In DATAVIEW, web-based GUI allows users to compose and edit workflows in an appealing visual style, e.g., by dragging and dropping task components and data elements onto the design panel and connecting them through edges as executable workflows. Its workflow engine manages the workflow schedulers, workflow specification mappers, dataflow storage, provenance data, compute resources, runtime monitor and analysis tools, etc. Workflow specifications are written in JSON-based SWL (Scientific Workflow

Language). Its elastic *Cloud Resource Management* module dynamically provisions and de-provisions virtual machines throughout workflow execution, based on user specified preferences. DATAVIEW features an open extensible architecture for its workflow engine, which consists of a set of workflow planners and a set of workflow executors. A developer can easily choose any existing or to develop their own custom workflow planners and executors.

C. **Kepler**

Kepler [26] is a community-driven workflow system that supports scientific workflow applications, and help scientists, analysts and programmers to create and analyze scientific data such as sensor data, medical images and simulations, etc. Kepler provides a Java-based component assembly framework with a graphical user interface to support the assembly of concurrent task components. The key underlying principle of Kepler is to utilize well-defined models of computation to govern the interactions between task components in a workflow during execution. Using Kepler's graphical desktop GUI, scientists can create executable scientific workflows by simply dragging and dropping task components. Kepler supports workflow execution on a local machine, a cluster or through web services. Kepler is capable to invoke remote Restful web APIs and broadcasts the response through its output port. Java and R are supported in Kepler for programmatic workflow application development.

Kepler is open-source. It can perform type checking at both design-time (static) and run-time (dynamic) on workflows and data. Kepler adopted the "one thread for each task" strategy, in which tasks are run as local Java threads by default, while distributed execution threads are provided via Grids and web services. The Web service support in Kepler allows users to take a WSDL (Web Service Description Language) description and the name of a web service to customize a scientific workflow. The Grid support in Kepler consists of certificate-based authentication, job submission, third-party data transfer, and SRB (Storage Resource Broker), etc. Kepler also supports execution of MapReduce tasks on the Hadoop Master-slave architecture, and the tasks can be executed in batch mode using Kepler's background execution.

D. **Taverna**

Taverna [27] is a tool suite written in Java, and can help scientists in diverse domains, including biology, chemistry, medicine, etc., to create and execute scientific workflows. Taverna supports workflow execution locally or remotely via WSDL-style web services or RESTful APIs. Taverna system includes a workbench application that provides a GUI interface for composition of workflows, and a Taverna Server that executes remote workflows. Besides desktop GUI support, Taverna also provides a command-line tool for executing workflows from a terminal. Workflows in Taverna are written in an XML-based language called *Scufl2* (Simple conceptual unified flow language). Taverna supports user-interaction with a running workflow within a web browser and has built-in support for myExperiment so that users can browse the myExperiment website within the Taverna Workbench. Users can access the full myExperiment search options and publish their workflows on myExperiment for others to use.

Taverna is open-source. In Taverna, a workflow consists of three main types of entities: *processors*, *data links*, and *coordination constraints*. *processors* take input data and produce a set of output data; *data links* mediate the data flow between a data source and a data sink; *coordination constraints* bind two processors and control their execution to ensure their executions are in a certain order. Workflows can be executed in the *Scufl workbench* using its enactor panel, which allows users to specify their input data for a workflow and launch a local instance of the *Freefluo* enactment engine. The *Freefluo* engine is not tied to any workflow language nor to any execution architecture, thus in effect is decoupled from both the textual form of a workflow specification and the details of a service invocation.

E. **Swift**

Swift [28] represents an interesting and distinct category of workflow management and is reviewed below in comparison with other systems presented above. By its nature, Swift is both a general-purpose programming language and a scripting language for distributed parallel scripting. It is used for composing integrated parallel applications/workflows that can be executed on multicore processors, clusters, grids, or clouds. In Swift, the scripts express the execution of constituent programs that consume and produce file-resident datasets. Swift is a compiled language that uses C-like syntax and supports local clusters, grids, HPCs, and clouds. It explicitly declares files and other command-line arguments as the inputs to each program invocation. A focal point in Swift's design is that it provides a simple set of language constructs that regularize and abstract the notions of processes and external data for distributed parallel execution of large application programs.

Swift is open-source. Workflow execution is implicitly parallel and location-independent in Swift. As the number of processing units available on the shared resources varies with time, Swift can exploit the maximal concurrency permitted by data dependencies within the script and the resources available. Swift can use whatever resources available or economical at that moment when the user needs to run a swift application, without the need to continuously reprogram the execution scripts. The implicit parallelism achieved through Swift functions is no necessarily executed in the source-code order but rather based on their input data's availability. Applications should not assume that they will be executed on a particular host, or in any particular order with respect

to other application invocations in a script, or whether their working directories will be cleaned up after execution.

5. Conclusions and future work

In this survey, we provided an in-depth review on the state-of-the-art workflow scheduling algorithms in the cloud and existing representative workflow management systems with an emphasis on big data support. We started from a general introduction to the workflow scheduling problem and the challenges of workflow scheduling in the cloud, and made an insightful presentation of 10 state-of-the-art workflow scheduling algorithms and 5 representative workflow management systems. We proposed a new taxonomy for workflow scheduling algorithms in the cloud based on two dominant factors, makespan and monetary cost of workflow execution, and group workflow scheduling algorithms into four categories. We elaborated on each of the selected scheduling algorithms with comparisons in terms of their constraints, optimization objectives, types, time complexity etc. We also surveyed 5 representative workflow management systems regarding their scientific impact, novel features, UI accessibility, etc. As future work, we plan to introduce AI techniques in workflow management systems with the hope to bring mutual improvement between workflow management and machine learning/deep-learning.

Acknowledgement

This work is partially supported by National Science Foundation under grants CNS-1747095 and OAC-1738929.

References

[1] S. Abrishami, M. Naghibzadeh *et al.*, "Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds," *FGCS*, vol. 29, no. 1, pp. 158–169, 2013.

[2] C. Bai, S. Lu *et al.*, "LPOD: A local path based optimized scheduling algorithm for deadline-constrained big data workflows in the cloud," in *BidData*. IEEE, 2019, pp. 35–44.

[3] S. Z. M. Mojab, M. Ebrahimi *et al.*, "iCATS: Scheduling big data workflows in the cloud using cultural algorithms," in *IEEE BigData*, 2019, pp. 99–106.

[4] M. A. Rodriguez and R. Buyya, "A responsive Knapsack-based algorithm for resource provisioning and scheduling of scientific workflows in clouds," in *ICPP*. IEEE, 2015, pp. 839–848.

[5] A. Verma and S. Kaushal, "Deadline constraint heuristic-based genetic algorithm for workflow scheduling in cloud," *IJGUC*, vol. 5, no. 2, pp. 96–106, 2014.

[6] L. Liu, M. Zhang *et al.*, "Deadline-constrained coevolutionary genetic algorithm for scientific workflow scheduling in cloud computing," *CCPE*, vol. 29, no. 5, p. e3942, 2017.

[7] M. Ebrahimi *et al.*, "Scheduling big data workflows in the cloud under deadline constraints," in *BigData*. IEEE, 2018, pp. 33–40.

[8] J. Meena, M. Kumar *et al.*, "Cost effective genetic algorithm for workflow scheduling in cloud under deadline constraint," *IEEE Access*, vol. 4, pp. 5065–5082, 2016.

[9] M. A. Rodriguez and R. Buyya, "Budget-driven scheduling of scientific workflows in IaaS clouds with fine-grained billing periods," *ACM TAAS*, vol. 12, no. 2, pp. 1–22, 2017.

[10] M. Mao and M. Humphrey, "Scaling and scheduling to maximize application performance within budget constraints in cloud workflows," in *IPDPS*. IEEE, 2013, pp. 67–78.

[11] C. Q. Wu, X. Lin *et al.*, "End-to-end delay minimization for scientific workflows in clouds under budget constraint," *TCC*, vol. 3, no. 2, pp. 169–181, 2014.

[12] A. Mohan *et al.*, "Scheduling big data workflows in the cloud under budget constraints," in *BigData*. IEEE, 2016, pp. 2775–2784.

[13] C. Lin and S. Lu, "Scheduling scientific workflows elastically for cloud computing," in *CLOUD*. IEEE, 2011, pp. 746–747.

[14] J. J. Durillo and R. Prodan, "Multi-objective workflow scheduling in amazon EC2," *Cluster computing*, vol. 17, no. 2, pp. 169–189, 2014.

[15] H. Topcuoglu, S. Hariri *et al.*, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *TPDS*, vol. 13, no. 3, pp. 260–274, 2002.

[16] G. Ismayilov and H. R. Topcuoglu, "Neural network based multi-objective evolutionary algorithm for dynamic workflow scheduling in cloud computing," *FGCS*, vol. 102, pp. 307–322, 2020.

[17] Z. Zhu, G. Zhang *et al.*, "Evolutionary multi-objective workflow scheduling in cloud," *TPDS*, vol. 27, no. 5, pp. 1344–1357, 2015.

[18] Q. Wu *et al.*, "MOELS: Multiobjective evolutionary list scheduling for cloud workflows," *T-ASE*, vol. 17, no. 1, pp. 166–176, 2019.

[19] M. Maciej, G. Juve *et al.*, "Cost-and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds," in *SC*, 2012.

[20] A. Verma and S. Kaushal, "A hybrid multi-objective particle swarm optimization for scientific workflow scheduling," *Parallel Computing*, vol. 62, pp. 1–19, 2017.

[21] H. Arabnejad, J. G. Barbosa *et al.*, "Low-time complexity budget–deadline constrained workflow scheduling on heterogeneous resources," *FGCS*, vol. 55, pp. 29–40, 2016.

[22] V. Arabnejad *et al.*, "Budget and deadline aware e-science workflow scheduling in clouds," *TPDS*, vol. 30, no. 1, pp. 29–44, 2018.

[23] M. Ghasemzadeh *et al.*, "Deadline-budget constrained scheduling algorithm for scientific workflows in a cloud environment," in *OPODIS 2016*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[24] E. Deelman, K. Vahi *et al.*, "Pegasus, a workflow management system for science automation," *FGCS*, vol. 46, pp. 17–35, 2015.

[25] A. Kashlev *et al.*, "Big data workflows: A reference architecture and the DATAVIEW system," *STBD*, vol. 4, no. 1, pp. 1–19, 2017.

[26] B. Ludäscher, I. Altintas *et al.*, "Scientific workflow management and the Kepler system," *CCPE*, vol. 18, no. 10, pp. 1039–1065, 2006.

[27] T. Oinn, M. Addis *et al.*, "Taverna: A tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004.

[28] M. Wilde *et al.*, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011.

[29] J. Goecks *et al.*, "Galaxy: A comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome biology*, vol. 11, no. 8, p. R86, 2010.

[30] J. Freire, D. Koop *et al.*, "Reproducibility using vistrails," *Implementing Reproducible Research*, vol. 33, 2014.

[31] P. Nyström *et al.*, "The TimeStudio Project: An open source scientific workflow system for the behavioral and brain sciences," *Behavior research methods*, vol. 48, no. 2, pp. 542–552, 2016.

[32] M. R. Berthold, N. Cebron *et al.*, "KNIME-the Konstanz information miner: version 2.0 and beyond," *AcM SIGKDD explorations Newsletter*, vol. 11, no. 1, pp. 26–31, 2009.

[33] W. A. Warr, "Scientific workflow systems: Pipeline Pilot and KNIME," *J Comput Aided Mol Des*, vol. 26, no. 7, pp. 801–804, 2012.

[34] J. Kranjc, R. Orač *et al.*, "ClowdFlows: Online workflows for distributed big data mining," *FGCS*, vol. 68, pp. 38–58, 2017.

[35] M. Perovšek, J. Kranjc *et al.*, "TextFlows: A visual programming platform for text mining and natural language processing," *Science of Computer Programming*, vol. 121, pp. 128–152, 2016.

[36] C. Lin, S. Lu *et al.*, "A reference architecture for scientific workflow management systems and the VIEW SOA solution," *TSC*, vol. 2, no. 1, pp. 79–92, 2009.

[37] Y. Kano *et al.*, "U-Compare: A modular NLP workflow construction and evaluation system," *J. Res. Dev*, vol. 55, no. 3, pp. 11–1, 2011.

[38] S. Mofrad, I. Ahmed *et al.*, "SecDATAVIEW: A secure big data workflow management system for heterogeneous computing environments," in *ACSAC*, 2019, pp. 390–403.

[39] R. G. Reynolds *et al.*, "Cultural algorithms: modeling of how cultures learn to solve problems," in *ICTAI*. IEEE, 2004, pp. 166–172.