A Generic Efficient Workflow Executor for the Optimizations of Run-time Execution of Workflow Schedules

***Abstract*—While much research has been done on workflow scheduling to improve the efficiency of workflow execution under various constraints, the question remains open whether it is possible to design and develop a generic workflow executor that can efficiently execute a schedule generated by an arbitrary workflow scheduler (planner). In this paper, in the context of the DATAVIEW scientific workflow management system, we design a workflow engine architecture that separates a workflow planner from a workflow executor. While the workflow planner focuses on generating a near-optimal workflow schedule according to a given performance metric under some given constraints, a workflow executor serves as the run-time system for the orchestration of workflow task execution that is performed by individual task executors at various worker nodes. We implement this architecture, design the necessary distributed algorithms for the workflow executor and task executors with optimizations in data movement, task movement and communication among different subsystems. The new implementation under the designed architecture, algorithms and optimization strategies results in a new version of DATAVIEW.**

***Keywords*-Workflow Engine; Workflow Execution; Parallel Processing; Data Movement; Optimization;**

1. INTRODUCTION

Recently, more and more complex analysis problems are addressed by using scientific workflow, which has been widely recognized to be a non-trivial computing paradigm [1] in numerous domains such as astronomy [2], biology [3] [4], earthquake science [5], physics [6], patient diagnosis [7], and others [8]. Workflows enable scientists to address problems in a modular manner, which provide the capability for different researchers working on one problem but focus on their own domains [9].

Workflow design and workflow execution are two key fundamental research problems in the field of scientific workflow management systems [10] [11]. While much research has been done on workflow scheduling to improve the efficiency of workflow execution under various constraints [12], the question remains open whether it is possible to design and develop a generic workflow executor that can efficiently execute a schedule generated by an arbitrary workflow scheduler (planner). The answer to this question is important because 1) such separation will allow the independent and parallel optimizations of workflow planners and workflow executors, and 2) the possibility of the development of a generic and efficient workflow executor that can be coupled with an arbitrary workflow planner for the efficient execution of workflow schedules. The first objective of this research is to design a workflow engine architecture and then focuses on the design and development of a generic and efficient workflow executor for the optimizations of run-time execution of workflow schedules.

A workflow is usually represented by a directed acyclic graph (DAG), in which tasks are indicated by nodes and data flows are indicated by edges [13]. Automatically executing a workflow in the cloud is not an easy task while considering the dynamic VMs provisioning policy, the tasks parallelism execution on different VM instances and the parallel data movements from one VM instance to another VM instance. The generic nature of the proposed workflow executor means that the workflow executor can take as input an arbitrary workflow schedule that is produced by an arbitrary workflow planner and then execute the workflow schedule in a parallel and distributed manner, making best efforts in various optimizations and management of low level workflow execution details, including task movement, data movement, synchronization and communication among the workflow executor and various task executors residing at different worker nodes. Therefore, a second objective of this research is to design a parallel and distributed algorithm to prescribe the coordinated execution of a workflow schedule by various task executors under the orchestration of the workflow executor. This algorithm must be correct and efficient.

As scientific workflow management systems are increasingly used in big data analysis, leading to the so called big data workflow management systems [14]. Data movement optimization become increasingly important in the efficient distributed execution of scientific workflows. In particular, we need to consider the following principles: 1) If possible, we should perform multiple data movement processes in parallel; 2) If possible, we should initiate a data movement process as early as possible; 3) If possible, we should use in-memory data movement without persisting the data as a file; 4) If the data is larger than the size of the main memory, we might need to use files for data movement; 5) We should initiate the execution of a workflow task as early as all input data is available; 6) If possible, we should allow the parallelism between workflow task execution and data movement. A third objective is to apply these principles in the design and implementation off our proposed workflow executor and task executors. In summary, the contributions

of this paper are:

1) We propose algorithms for workflow executor $Workflow Executor\_Beta$, on which parallel tasks are submitted. And its associated task executors $TaskExecutor\_Beta$, on which task are executed.
2) We propose an adaptive data movement strategy that alleviates the data movement overheads.
3) We conducted systematic experiments on several real workflows including diagnosis recommendation workflow [7], Map-Reduce workflow [15] and distributed K-means workflow to corroborate the validity of our proposed architecture. The results also ameliorate overheads on the execution times of various workflows.

This paper is structured as follows. Section 2 introduces the architecture of workflow engine in the DATAVIEW system. Section 3 explains workflow executor $Workflow Executor\_Beta$ and its associated task executors $TaskExecutor\_Beta$ in algorithms manner. The experimental results are presented and discussed in Section 4. Some other centralized workflow systems and serverless workflow systems are introduced as related work in Section 5. Finally, Section 6 concludes this paper.

2. The Architecture for Workflow Engine

In an early paper [14], a reference architecture of the big data workflow management system was proposed and is reflected in our previous implementation of DATAVIEW, on which a new workflow engine architecture is designed and proposed. Figure 1 shows the three layer components including workflow planners, workflow executors and task executors in the generic workflow engine.

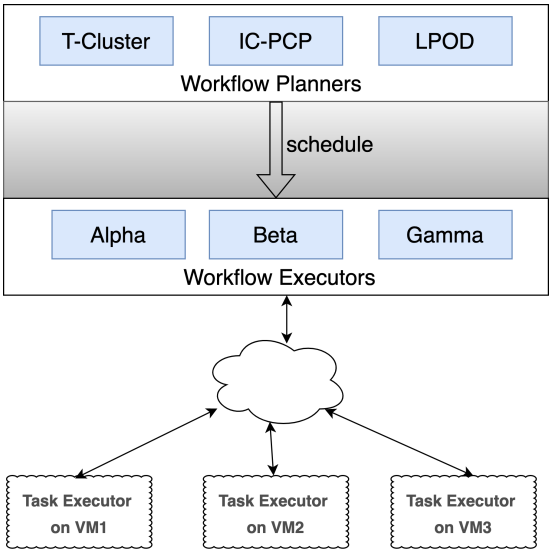


Figure 1: Workflow engine architecture in cloud context.

Generally, a workflow executor will be running in a master node and task executors are executed in worker nodes represented by Virtual Machines (VMs). Workflow planners and executors are independent, but task executors will bind with a specific workflow executor. A task executor executed on a VM can be deployed in a static or dynamic manner through code transfer mechanism from a workflow executor or a VM launching with pre-defined image.

Specially, a workflow planner such as T-Cluster [16], IC-PCP [17] or LPOD [18] takes a workflow specific information from the workflow design panel and a given performance metric under some constrains and generates a workflow schedule. Then a workflow executor takes the schedule and orchestrates the execution according to the schedule including provisioning, de-provisioning VMs and invoking the task execution on a task executor at the scheduled time point.

3. The Algorithms for Workflow Executor and Task Executors

The workflow executor is an essential component of the workflow engine module, which dispatches workflow tasks to the best fit VM instances based on the workflow schedule sch generated by a workflow planner and manages the life cycle of each VM instance. A recently implemented new version of workflow executor, named $Workflow Executor\_Beta$, and its corresponding task executors, named $TaskExecutor\_Beta$, in DATAVIEW are described in this section.

This implementation provides a centralized workflow executor on the master node to manage each workflow task execution state and assign tasks to worker nodes. Before delving in the implementation of $Workflow Executor\_Beta$, we introduce few key terminologies which related to our implementation as follow. First, a *task schedule* in our implementation refers to an abstract object that hosts a task of workflow, which includes the task ID, the task name, all direct parent tasks, lists of incoming/outgoing data channels, and the IP of the designated VM of this task. A *local schedule* is created per each provisioned VM and hosts all the tasks (or task schedules) assigned to run on the VM. Finally, a *global schedule* is a collective objects that hosts all local schedules. In other words, the global schedule of a workflow is the corresponding implementation object of the (logical) schedule sch which is generated by the workflow planner in DATAVIEW for a given workflow.

The details of $Workflow Executor\_Beta$ is shown in Algorithm1. It takes workflow global schedule $gsch$ as input, which comprises of a list of local schedules: $[lsch_1, \cdots, lsch_n]$. Each local schedule $lsch$ is to be realized by a LocalScheduleRun thread object, called $lschr$, spawned to check the task status, and each task schedule $tsch$ is to be realized by a TaskRun object, called tr, spawned to submit each task to the corresponding VM instance.

First (line 2), each thread $lschr_i$ is created for each local schedule; then one task schedule object is retrieved based on the index of $lschr_i$, on which a TaskRun object is created for each local task (included in the local schedule) and is initialized with the number of unready input ports data of each task (lines $3-7$). Then all threads are executed in parallel to execute the ready tasks (lines $9-35$). All tasks in each thread (corresponding to a local schedule and a VM) are topologically sorted and stored in variable ts_i (line 10). Each task in ts_i needs be checked on its readiness to run, and this is done by checking the number of unready input ports data of the task. If there exits any unready input ports data, the current thread will be put on waiting (lines $15-19$). Otherwise, the task is ready to be submitted to its corresponding VM instance, since its all input ports data are ready and the execution code is ready (lines $12-13$). Those tasks connected with t_{entry} should be updated its number of unready input ports data after finishing moving its input data (line 14). An socket connection is created between $Workflow Executor\_Beta$ and $TaskExecutor\_Beta$ and obtain the object input/output streams to the $TaskExecutor\_Beta$ (lines $20-21$). Line 22 invokes $TaskExecutor\_Beta$ to execute task t_j on its assigned VM instance $M(t_j)$ (M is the mapping function between task and its assigned VM instance) by writing the task specification information to $TaskExecutor\_Beta$. During the execution period of task t_j, several lists of task IDs representing data movement from input ports to output ports are read from $TaskExecutor\_Beta$ (line 24). In the For loop, each TaskRun is retrieved based on a task ID. Then the number of unready input ports data of each task represented in TaskRun object tr_k is decreasing by 1 (line 27). Once the child task in tr_k is ready, the thread containing the parent task run of tr_k is waken up through the $signal()$ method (lines $28-32$), which is lock-protected.

The associated task executor $TaskExecutor\_Beta$ running on each VM instance is responsible for a task's execution and data movement between VMs. The whole process of $TaskExecutor\_Beta$ is shown in Algorithm2. Within the do-while loop, an output stream and an input stream object to workflow executor $Workflow Executor\_Beta$ are created first (lines $2-3$). Then the specification information of a task is read from the input stream object (line 4), from which the task name is further retrieved (line 5). After that, a new task instance of t_j is created and executed (line 6). Once the task is finished, the execution time of the task is recorded in the task specification object (line 7). The $NotationThread$ algorithm is called by persistently checking the existing elements in a queue and writing to $Workflow Executor\_Beta$, until the success status is reached (line 8). The output data generated in each output port of this task is moved to the VMs that the direct children tasks of t_j are going to be executed on in a parallel manner (line 11). In the meantime, each task ID indicating a child task of t_j with one input data being ready is added into queue q and the data transfer time between the pair of parent-child tasks is stored in the task specification information (lines $12-13$). If no data movements happens, those task IDs

Algorithm 1: WorkflowExecutor_Beta

Input : GlobalSchedule $gsch = [lsch_1, \cdots, lsch_n]$
Output: Exit code

1 **foreach** $lsch_i \in gsch$ **do**
2 create a LocalScheduleRun $lschr_i$ for $lsch_i$;
3 **foreach** $j \in lsch_i.length()$ **do**
4 Taskschedule $tsch \leftarrow lsch_i[j]$;
5 create TaskRun tr_j;
6 $tr_j.numOfIncomingDataUnready \leftarrow$ $tsch.getIncomingDataChannels().size()$;
7 **end**
8 **end**
9 **forall** *LocalScheduleRun* $lschr_i$ ***in parallel*** **do**
10 $ts_i \leftarrow$ all TaskRuns in $lschr_i$ in order ;
11 **foreach** *TaskRun* $tr_j \in ts_i$ **do**
12 send all input datas of task t_j in tr_j to its corresponding $VM, M(t_j)$;
13 send the code of t_j to its corresponding $VM, M(t_j)$;
14 update $numOfIncomingDataUnready$ of tr_j if task t_j connects t_{entry};
15 **while** $tr_j.numOfIncomingDataUnready \neq 0$ **do**
16 $lschr_i.mLock.lock()$;
17 $lschr_i.mReadyCon.await()$;
18 $lschr_i.mLock.unlock()$;
19 **end**
20 $out \leftarrow$ an object output stream to the $TaskExexcutor\_Beta$;
21 $in \leftarrow$ an object input stream to the $TaskExexcutor\_Beta$;
22 invoke the TaskExecutor_Beta at $VM, M(t_j)$ to execute the task t_j by $out.write(taskSpec)$;
23 **while** *t_j's execution status is not successful* **do**
24 $list \leftarrow in.read()$;
25 **foreach** *taskID* $str \in list$ **do**
26 get taskRun tr_k from str;
27 $tr_k.numOfIncomingDataUnready \leftarrow tr_k.numOfIncomingDataUnready - 1$;
28 **if** $tr_k.numOfIncomingDataUnready == 0$ **then**
29 $tr_k.ownerLocalScheduleRun.mLock$
30 .lock();
31 $tr_k.ownerLocalScheduleRun.mReadyCon$
32 .signal();
33 $tr_k.ownerLocalScheduleRun.mLock$
34 .unlock();
35 **end**
36 **end**
37 **end**
38 **end**
39 **end**
40 return SUCCESS;

Algorithm 2: TaskExecutor_Beta

```
1  do
2      out ← an Object Output Stream to the
         WorkflowExecutor_Beta;
3      in ← an Object Input Stream from the
         WorkflowExecutor_Beta;
4      taskSpec ← in.read();
5      task t'_j s taskName ← taskSpec.get(taskName);
6      instantiate and execute task t_j;
7      taskSpec.put("execTime", ET(t_j, M(t_j)));
8      NotationThread(out, q).start();
9      forall task t_i ∈ all children tasks of t_j in parallel do
10         if M(t_i) ≠ M(t_j) then
11             move output data of t_j to VM, M(t_i);
12             q.add(taskID);
13             taskSpec.put("dataTransferTime",
                 DTT(D_{j,i}, M(t_j), M(t_i)));
14         end
15         else
16             arrayList.add(taskID);
17         end
18     end
19     if !arrayList.isEmpty() then
20         q.add(arrayList);
21     end
22     q.add(taskSpec);
23 while True;
```

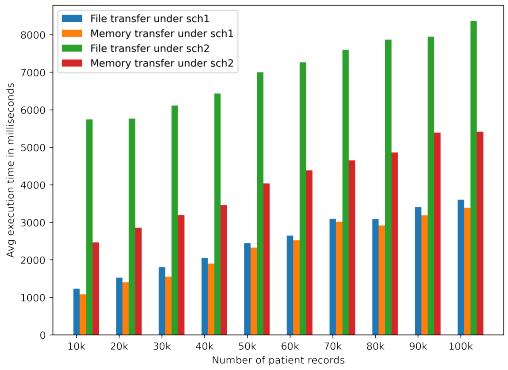
representing children tasks with ready inputs data are added into the $arrayList$ array firstly (lines $15-17$). If $arrayList$ is not empty, it will be added into queue q. At last, the task specification information including the task's execution time on the VM type and the data transfer time between the pair of parent-child tasks is added into q, which is signaled as the task execution succeeded status. The do-while loop continues to run until the instance is deprovisioned. While the tasks in a workflow have different start times, this arrangement assures each task get instantly executed once it gets ready.

4. Evaluation

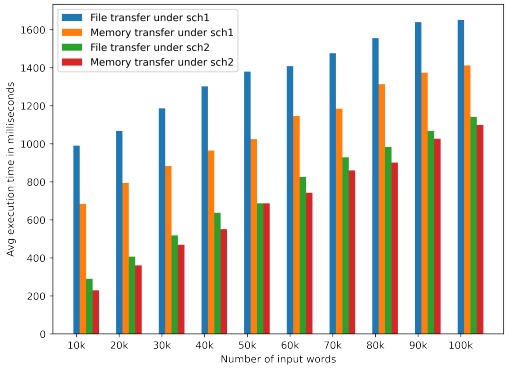
This section presents the evaluation experimental results of DATAVIEW with $Workflow Executor\_Beta$ and its associated $TaskExecutor\_Beta$ by using three real-world workflows application.

A. Experiment Setup

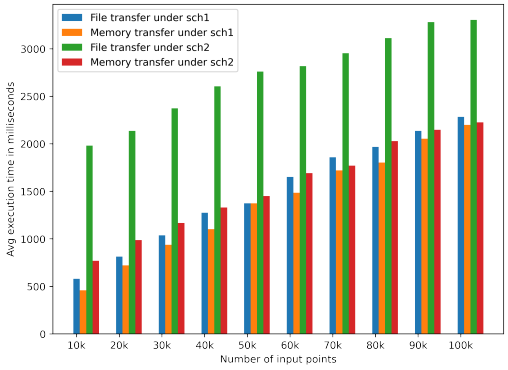
We have implemented three real-world workflows based on the DATAVIEW API. Several workflow scheduling algorithms have been integrated into DATAVIEW system, which can be used to generate a workflow schedule sch, on which each concrete workflow can be assigned to scheduled VM instances. In order to support the parallel data movement in $TaskExecutor\_Beta$, t2.xlarge instance type on AWS EC2 is selected as the provisioned VM type, which has 4 vCPU and 16GM memory and moderate network performance. All experiments were conducted with 2GB JVM heap memory space. In order to demonstrate the efficiency of direct memory transfer strategy, two workflow schedules on which all workflow tasks are assigned to one VM named $sch1$ and every task is assigned to each VM instance named $sch2$ are applied. The execution time is used as a criterion to demonstrate the reported results, which is calculated based on the provenance information collected from the master node.



(a) Diagnosis Recommendation Workflow.



(b) Word Count Workflow.



(c) Distributed K-means Workflow.

Figure 2: Execution time of running different workflows in different configurations with different input datasets.

B. Diagnosis Recommendation Workflow

The diagnosis recommendation workflow [7] having a pipeline structure uses semi-supervised clustering model to assign unlabeled instances, on which the diagnosis label is recommended by applying a certain minimum threshold to a new patient. We synthetically created $10,000 - 100,000$ records with a step size of $10,000$. In the first experiment, all experiments were conducted under four settings: (a) schedule $sch1$ with intermediate data transferred with file, (b) schedule $sch1$ with intermediate data transferred with memory, (c) schedule $sch2$ with intermediate data transferred with file, (d) schedule $sch2$ with intermediate data transferred with memory. Figure 2a shows that, the execution time increase as the patient records increasing. This is because when the patients size increases, it takes more time for training and testing the diagnosis model and bigger size intermediate data are transferred if necessary. The experimental results show that case (b) reduces 8% average overhead on the same dataset compared with cast (a) and case (d) reduces 42% overhead on the same dataset compared with case (c). The results depict that the direct memory transfer strategy is more efficient than file transfer policy.

C. Word Count Workflow

The word count workflow is a well-known application example with the Map-Reduce [19] structure. We created a concrete workflow containing 10 tasks including two Split tasks, two Map tasks, three Shuffle tasks and three Reduce tasks. In the experiments, we randomly generated $100,000 - 1,000,000$ words as the workflow input. Figure 2b compares the execution time under different workflow schedules and data movement strategies. The execution time under $sch2$ is shorter than $sch1$ under same data movement strategy because of the parallel task execution in cloud. We noticed that the file transfer strategy impose $1.28x$ and $1.1x$ overhead under each schedule respectively.

D. The Distributed K-means Workflow

We also designed a distributed K-means workflow in DATAVIEW by predefining the number of iteration loops. During each iteration, three tasks DisCalculateCentroidStep1 task, DisCalculateCentroid task and DisReassignCluster task are executed. The DisKMeansInitialization task is executed only once to split the input data to several clusters by a user-defined input. In the experiment, we randomly generated $100,000 - 1,000,000$ points with an x and a y coordinate. Figure 2c gives the execution time under different cases with iteration number equaling to 3. Since we do not increase the parallel level of task DisCalculateCentroid and task DisReassignCluster, the execution time under $sch2$ is longer than $sch1$ under same data movement strategy. The results demonstrates that the file transfer strategy impose $1.13x$ and $1.3x$ overhead under each schedule.

5. Related Work

Workflow engine plays an import role to execute a workflow in a target computing context successfully. The optimization is a multi-dimensional problem, which includes workflow scheduling optimization, data movement optimization and execution engine selection and computing environment selection. Representative workflow engines which support various forms of parallelism are provided as follow.

Swift [20] implements its own scripting language, which supports parallel computing essentially. A concrete workflow designed by Swift can be modeled as a set of program invocations with their available inputs. Since Swift scripts are oblivious to computing resources, a workflow designed by Swift can be executed on local computer, cluster, grids or clouds. Tasks in a workflow are assigned to computing resource reactively at run time period, once their input data are available.

Pegasus [21] is implemented on the top of Condor's Directed Acyclic Graph Manager (DAGMan) [22], which specifies a workflow as a DAG. DAGMan submits ready tasks in a DAG one by one to HTCondor [23] by maintaining a task queue, on which a task parallelism execution manner is supported. A workflow designed by Pegasus is translated to an executable DAGMan workflow with optimizing some workflow structure by clustering some short-running time tasks. Different from the Swift, the intermediate files generated by Pegasus are location specific.

Kepler [24] supports the assembly of concurrent tasks in workflows through a graphical interface. The workflows constructed in Kepler can be executed on local computer, grids and clouds, which adopt the "one thread per task" execution strategy. Well-defined computation models have been developed and utilized in Kepler to govern the interactions between tasks during the workflow execution. Kepler supports customization on workflows through WSDL (Web Service Description Language) in the cloud, which also supports MapReduce tasks on the Hadoop master-slave architecture.

Hadoop [25] is implemented for distributed processing of large data sets across cluster of computer by using predefined Map-Reduced models, which is friendly to IO-intensive workflows. However, the Hadoop cannot execute customized workflow structure, which limit the usability for scientific domain researchers. Benefit from the HDFS system in the cluster environment, Hadoop can handle the intermediate data efficiently. Moving the execution environment from cluster to cloud will delegate the performance.

Compared with Hadoop, Spark [26] was implemented to have a in-memory storage of intermediate data by using the Resilient Distributed Datasets (RDDs), which makes data partitioned and processed in a parallel manner. Executing a workflow with Spark engine cannot benefit from RDDs since each task in a workflow is treated as a blackbox. What's

more, a workflow schedule cannot directly applied since the input format is in the Spark RDDs.

Recently, microservices has attracted more attention in academic and industry ear because of its scalability and flexibility [27]. A serverless infrastructure representing by Google Cloud Functions [28] or AWS Lambda [29] has emerged recently, which has been applied to execute a scientific workflow [30]. However, this kind of architecture relies on intermediate data storage compared with service based architecture, such as DATAVIEW [14].

6. Conclusions and future work

In this paper, we present the $Workflow Executor\_Beta$ and its associated $TaskExecutor\_Beta$ in the DATAVIEW workflow system. Several detailed algorithms depict the task submission, task execution and successful execution status transfer between them. Our experimental results show the usability and efficiency of our system with incorporating the direct memory transfer strategy for the intermediate data generated during the task execution process. Currently, SWfMSs provides a centralized workflow engine on the master node to manage the task execution status and assign task to the worker node, which brings some communication overhead. In the future, some optimizations can be done to alleviate this communication overhead by assigning sub-workflow to worker node directly.

Acknowledgement

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

References

[1] L.-J. L. Zhang, "Quality-driven service and workflow management," *IEEE Transactions on Services Computing*, vol. 4, no. 02, pp. 84–84, 2011.

[2] J.-S. Vöckler, G. Juve, E. Deelman, M. Rynge, and B. Berriman, "Experiences using cloud computing for a scientific workflow application," in *Proceedings of the 2nd international workshop on Scientific cloud computing*. ACM, 2011, pp. 15–24.

[3] M. Abouelhoda, S. A. Issa, and M. Ghanem, "Tavaxy: Integrating taverna and galaxy workflows with cloud computing support," *BMC bioinformatics*, vol. 13, no. 1, pp. 1–19, 2012.

[4] V. C. Emeakaroha, M. Maurer, P. Stern, P. P. Łabaj, I. Brandic, and D. P. Kreil, "Managing and optimizing bioinformatics workflows for data analysis in clouds," *Journal of grid computing*, vol. 11, no. 3, pp. 407–428, 2013.

[5] M. Atkinson, C. S. Liew, M. Galea, P. Martin, A. Krause, A. Mouat, O. Corcho, and D. Snelling, "Data-intensive architecture for scientific knowledge discovery," *Distributed and Parallel Databases*, vol. 30, no. 5-6, pp. 307–324, 2012.

[6] D. A. Brown, P. R. Brady, A. Dietz, J. Cao, B. Johnson, and J. McNabb, "A case study on the use of workflow technologies for scientific analysis: Gravitational wave data analysis," in *Workflows for e-Science*. Springer, 2007, pp. 39–59.

[7] I. Ahmed, S. Lu, C. Bai, and F. A. Bhuyan, "Diagnosis recommendation using machine learning scientific workflows," in *2018 IEEE International Congress on Big Data (BigData Congress)*. IEEE, 2018, pp. 82–90.

[8] F. Bhuyan, S. Lu, I. Ahmed, and J. Zhang, "Predicting efficacy of therapeutic services for autism spectrum disorder using scientific workflows," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 3847–3856.

[9] E. Deelman, K. Vahi, M. Rynge, R. Mayani, R. F. da Silva, G. Papadimitriou, and M. Livny, "The evolution of the pegasus workflow management software," *Computing in Science & Engineering*, vol. 21, no. 4, pp. 22–36, 2019.

[10] G. Singh, C. Kesselman, and E. Deelman, "Optimizing grid-based workflow execution," *Journal of Grid Computing*, vol. 3, no. 3, pp. 201–219, 2005.

[11] K. Lee, N. W. Paton, R. Sakellariou, E. Deelman, A. A. Fernandes, and G. Mehta, "Adaptive workflow processing and execution in pegasus," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 16, pp. 1965–1981, 2009.

[12] J. Liu, S. Lu, and D. Che, "A survey of modern scientific workflow scheduling algorithms and systems in the era of big data," in *2020 IEEE International Conference on Services Computing (SCC)*. IEEE, 2020, pp. 132–141.

[13] J. Yu and R. Buyya, "A taxonomy of scientific workflow systems for grid computing," *ACM Sigmod Record*, vol. 34, no. 3, pp. 44–49, 2005.

[14] A. Kashlev, S. Lu, and A. Mohan, "Big data workflows: a reference architecture and the DATAVIEW system," *Services Transactions on Big Data (STBD)*, vol. 4, no. 1, pp. 1–19, 2017.

[15] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," 2004.

[16] A. Mohan, M. Ebrahimi, S. Lu, and A. Kotov, "A nosql data model for scalable big data workflow execution," in *2016 IEEE International Congress on Big Data (BigData Congress)*. IEEE, 2016, pp. 52–59.

[17] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 158–169, 2013.

[18] C. Bai, S. Lu, I. Ahmed, D. Che, and A. Mohan, "Lpod: A local path based optimized scheduling algorithm for deadline-constrained big data workflows in the cloud," in *2019 IEEE International Congress on Big Data (BigDataCongress)*. IEEE, 2019, pp. 35–44.

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

[20] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011.

[21] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. Da Silva, M. Livny *et al.*, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[22] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger, "Workflow management in condor," in *Workflows for e-Science*. Springer, 2007, pp. 357–375.

[23] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor-a hunter of idle workstations," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 1987.

[24] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the kepler system," *Concurrency and computation: Practice and experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[25] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 2010, pp. 1–10.

[26] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

[27] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Computing Surveys (CSUR)*, 2021.

[28] "Google cloud function." [Online]. Available: https://cloud.google.com/functions

[29] "Amazon lambda function." [Online]. Available: https://aws.amazon.com/lambda/

[30] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions," *Future Generation Computer Systems*, vol. 110, pp. 502–514, 2020.