Simple yet Efficient Deployment of Scientific Applications in the Cloud

Leyi Sun, Yifan Zhuo, and Olivier Marin
New York University Shanghai, Shanghai, China
emails: [ls4571, yz4044, ogm2]@nyu.edu

***Abstract*—Scientific applications can benefit greatly from getting deployed on a cloud computing platform, but such deployments require skills and expertise that are beyond the reach of many scientists. We address this issue with a framework that simplifies the process of writing cloud-ready scientific applications, and that automates their deployment and execution on top of cloud infrastructures. This paper presents (1) our domain-specific language whose syntax is simple to learn and use, and (2) our compiler that exploits potential data parallelism opportunities and handles load balancing automatically for the users. Our framework prototype demonstrates the feasibility of our approach, and our scalability analysis looks promising.**

***Index Terms*—cloud computing, scientific workflows, middleware.**

I. Introduction

Among the scientific community, how many can write a scientific application that is ready for deployment on a cloud? We base this paper on the claim that writing a cloud-ready scientific application requires skill and expertise beyond the reach of a significant proportion of potential users. Our aim is to endow the scientific community at large with a framework that makes short work of writing and deploying a scientific application on the cloud.

Scientific applications and cloud computing platforms are an obvious match. Scientific applications generally manipulate extremely large datasets, and their workflows can reach high levels of complexity. By providing on-demand provisioning and elasticity, cloud computing platforms offer invaluable flexibility and scalability to accommodate for any size of dataflow. But in and of itself, cloud computing does not simplify the modelling and the deployment of a scientific workflow.

Modelling and deployment are the two major obstacles to designing a language or tool that decreases the complexity of writing cloud-ready applications. The former consists in defining the right model to make the process of writing accessible to a large audience. And the latter consists in hiding the intricacies of workflow distribution and aggregation away from the user. Both challenges are tightly coupled. Domain Specific Language (DSL) frameworks, and in particular Scientific Workflow Management Systems (SWfMSs), address this modelling and deployment issue by combining a workflow model with a system-level implementation to support scientific applications. They allow scientists to break up an abstract workflow into distinct yet interdependent tasks, and execute them on computing resources in a simpler way.

But the learning curve of SWfMSs and scientific DSLs remains steep. Many of them ask users to convert their workflows to Directed Acyclic Graphs (DAGs). Though they provide languages for such conversion, their usage requires a strong understanding of their complex syntax rules. Moreover, to take better advantage of cloud elasticity, these solutions often ask users to provide explicit information about the deployment: for example, how many replicas of given tasks can be run in parallel, or how to distribute and aggregate the data. In our opinion, it is much preferable to delegate the management of this information to parallel and distributed computing experts.

Considering these challenges, we propose a scientific DSL framework that automatically assists users with the deployment and execution of their scientific workflows on top of cloud computing infrastructures. Similarly to existing frameworks, our DSL also helps domain experts write their scientific applications into DAGs but with a simpler syntax. Our framework combines a middleware with the DSL to (1) exploit data parallelism opportunities in the workflow, and (2) scale the workflow according to the availability of cloud resources. At the deployment stage, our framework schedules tasks with a focus on load-balancing, and preserves dependencies as specified by the users to ensure the correct execution of tasks.

This paper is organized as follows. We motivates this work further in Section II with the help of a simple example. We introduce our approach and detail the design of our middleware's components in Section III. In Section IV, we present a performance evaluation of the prototype we implemented as a proof of concept. We discuss the relative simplicity of its usage in Section V, and draw a comparison with related work in Section VI. Finally, we conclude and discuss the perspectives our solution offers in Section VII.

II. A motivational example: Word Count

Let's use a simple example to illustrate the potential complexity of writing a cloud-ready scientific application. The Word Count problem is very common in the literature: given a string, a solution must identify each unique word and count its number of occurrences. As simple as it sounds, it is essential for many applications such as search engines. One way to solve the Word Count problem is through Google's MapReduce [1]. The algorithm processes the input string with 2 phases, map and reduce. Figure 1(1) shows the workflow of the two phases: node A collects text data and feeds it to

the application; node B runs the `map` task to single out every word and associate it with an occurrence value of 1; and node C executes the `reduce` task which identifies duplicates and combines their associated occurrence values into a running total. Note that this is a "linear" workflow such that the reduce phase requires the output from the map phase as its input.

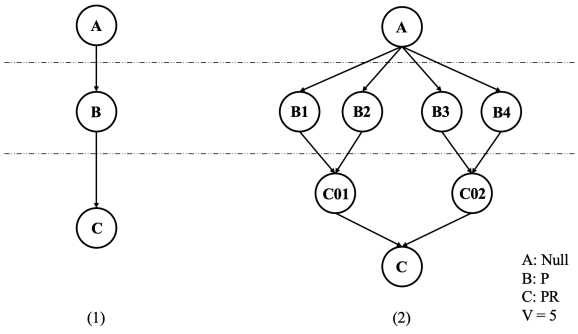

Fig. 1. Example of a workflow expansion: Word Count

Introducing data parallelism for both the `map` and `reduce` functions improves the performance of this algorithm at runtime. Multiple replicas of B and C can handle different pieces of data, provided node A now splits its text output into multiple disjoint subsets. `reduce` can also run in parallel, so we can build a data aggregation tree for this phase. Figure 1(2) shows the same workflow after parallelization. A scientist must address several issues to achieve this result. First of all, they must decide when it is adequate to create task replicas for parallel executions, and they must build a corresponding data aggregation tree. They must then plan the replication itself by creating an adequate number of task replicas, and finally schedule their executions: the goal is to achieve a high resource utilization rate on the available resources. They must also coordinate the deployment of the different replicas on available computing nodes, and link the nodes correctly to preserve dependencies. At this point, we hope our esteemed readers will agree that these issues are far from trivial.

In this paper, we propose *ODESSA*[1], a solution that answers all of these questions for the users. Our simple scripting language, *SWIG*[2], helps users describe the workflow of their application. Our compiler parses user-written scripts to generate a parallelization plan. Using this plan as input, our deployment engine distributes the execution efficiently on the cloud resources provisioned by the user. In the following sections, we use Word Count to illustrate how our solution operates.

III. *ODESSA*: A SIMPLER WAY OF WRITING CLOUD-READY SCIENTIFIC APPLICATIONS

This section presents the three components of our framework for facilitating the deployment of scientific applications in the cloud. These three components consist of: a scripting language for scientific workflows, a compiler that analyzes and the parallelizes these workflows, and a deployment architecture that runs the expanded workflows as processes in the cloud.

A. SWIG : *Easy conversion of workflows into graphs*

A Direct Acyclic Graph (DAG) is a common way of representing a workflow: each vertex denotes a computational task, and each edge indicates data dependencies between two tasks. In our opinion, such a graphic representation is very user-friendly. Therefore we designed *SWIG*: a scripting language for writing scientific workflows as DAGs. A *SWIG* script first lists the definitions of all the application tasks, and ends with the list of edges.

A task definition begins with the keyword `Task`, followed by the name of the task. Curly braces delimit the internal description of the task. The `run` keyword immediately precedes, and thus indicates the name of the executable associated with the task to run. Users can use their favorite common programming language to write their executables. Our prototype only supports Python at the moment, but extending it to additional languages would be trivial.

Users can provide indications about a task via optional annotations. `@ parallel` indicates a task that allows data parallelism, while `@ reduce` designates a task which performs reduction/data aggregation. Combining both annotations paves the way for building a data aggregation tree in the later graph expansion stage (see Subsection III-B). The line break between the executable declaration and the annotations is compulsory, but our compiler will skip indentations; we inserted them for the sake of readability.

```
Task [name] {
    run [name of the task executable]
    [@ parallel | @ reduce]*
}
```

Defining an edge only takes one line: the keyword `Edge` precedes the expression of a link $u \rightarrow v$, indicating that v is the successor of u in the DAG.

```
Edge [task name 1] -> [task name 2]
```

Aside from syntactic correctness, there are two rules for a *SWIG* script to be valid. Rule #1: there can only be one root task in the corresponding DAG, and it cannot be annotated as `@reduce`. This first rule simplifies the planification of the deployment, and prevents users from performing data reduction on an input flow that isn't parallel. Rule #2: the successor of a `@parallel` task must also have at least one annotation. This second rule also prevents inconsistencies in the parallelism of the DAG.

SWIG does not require the user to specify any further detail about parallelism or deployment. In the associated DAG, each node executes exactly one function. Our compiler's graph expansion algorithm automatically sets up a deployment according to available cloud resources (see Subsection III-C). Hence, the *SWIG* script given below ends up describing both potential workflows in Figure 1.

[1] *ODESSA* stands for *Odessa is a Deployment Engine for Scalable Scientific Applications*

[2] *SWIG: Scientific Workflows Into Graphs*. In *ODESSA*, a complicated job should always start with a good *SWIG* of vodka.

```
Task A {
    run split_data.py
}

Task B {
    run map.py
    @parallel
}

Task C {
    run reduce.py
    @parallel @reduce
}

Edge A -> B
Edge B -> C
```

B. Parsing and graph expansion

A user feeds both their task executables and *SWIG* script as input for *ODESSA*. In the first step, *ODESSA*'s compiler parses the script, and generates the code for deploying the workflow on the cloud.

Our compiler follows the recursion-descent approach. It parses the *SWIG* script to identify keywords. Upon encountering a `Task` keyword, it stores all the information contained between the corresponding curly braces in a *tasks list* entry for the creation of future task replicas. Besides the *tasks list*, the compiler also maintains a mapping between every task and its set of predecessors and successors. Upon encountering an `Edge` keyword, it updates this mapping so as to generate the necessary links during the deployment phase.

Our compiler also identifies errors in *SWIG* scripts such as missing task executables and undefined task names, and raises exceptions to help users spot and correct mistakes.

At this point, our compiler has listed all the tasks and mapped all the dependencies defined in the *SWIG* script. It can now correlate the corresponding DAG with the task annotations it collected to identify exploitable opportunities for parallel execution. Our graph expansion algorithm creates replicas of the tasks according to the annotations, and thus allows parallelism within suitable tasks.

The algorithm first uses the *SWIG* annotations to classify tasks into 4 types. Null (N) nodes cannot be executed in parallel, and they are not reduce jobs. Parallel (P) nodes can run as parallel replicas on different chunks of their input data. Reduce (R) nodes perform data reducing tasks. And Parallel Reduce (PR) nodes perform data reducing jobs that can be executed in parallel.

The expansion algorithm then computes the number of replicas it assigns to each task. Of course, it handles different types of nodes in different ways.

1) Parallel Nodes P: To exploit opportunities for parallelism, our algorithm creates replicas for P and PR nodes. Figure 2 shows how it handles a P node: case (1) if its predecessor is a P node, case (2) if it isn't.

Let V be the total number of cloud Virtual Machines (VMs) provisioned by the user in the cloud. Given a P task v, let the set S of siblings of v be

$$S = \{x | x \text{ and } v \text{ has the same depth}\}$$

Since nodes that are siblings in the DAG share the same depth, they can run concurrently or in parallel. There are two kinds of siblings: siblings associated with a P task, siblings associated with a non-P task. For a given v, assume there are m P-task siblings and n non-P-task siblings. Our algorithm

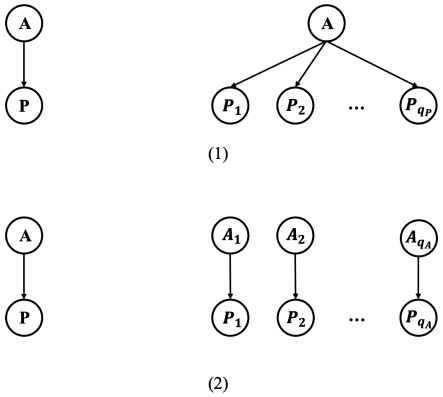

Fig. 2. Expanding a P node

assigns one VM to each non-P task node (n VMs in total) and the same number of VMs to all sibling P tasks ($m+1$ P tasks in total). The base number q of replicas created (and the other P-task siblings) is:

$$q = \frac{(V-n)\cdot\alpha}{m+1}$$

The factor α ($\alpha \in [0,1]$) saves a provision of idle VMs, so that successors of v and its siblings can start earlier.

The number of replicas created for a P node also depends on the types of their predecessors. If v has a P predecessor u, then we set q_v to be the same as q_u. Creating another number of replicas would require data splitting or aggregating and might affect the performance. Therefore,

$$q_v = \begin{cases} 2^{\frac{\log q}{\log 2}-1} & \text{if } v \text{ has no } P \text{ predecessor and } 2^{\frac{\log q}{\log 2}} > V, \\ 2^{\frac{\log q}{\log 2}} & \text{if } v \text{ has no } P \text{ predecessor and } 2^{\frac{\log q}{\log 2}} \leq V, \\ q_u & \text{if } v \text{ has a } P \text{ predecessor } u \end{cases} \tag{1}$$

Note that we round q to the nearest power of 2 in the first two cases for building potential binary aggregation trees as the successors.

2) Null Nodes N and Reduce Nodes R: Rule #2 of *SWIG* ensures that a P node cannot precede an N node. Our compiler will raise an error if this case occurs in the DAG.

For an R node v, if v has a P predecessor u that has q_u replicas, then the algorithm will create q_u new edges:

$$(u_i, v) \text{ for } i \text{ in } q_u$$

For other types of predecessors (N, R, PR), the original directed edge remains the same.

3) Parallel Reduce Nodes PR: Since a PR node is also a P node, the algorithm will create replicas as well. The algorithm rounds q to the closest powers of two minus one for the PR node for building a binary aggregation tree.

$$q_v = 2^{\frac{\log q}{\log 2}} - 1$$

The number of layers of this aggregation tree is $\log_2(q_v + 1)$.

The node D in Figure 3 shows the expansion of a PR node to a data aggregation tree. Note that, although q_D can be as large as 7 and the tree can have 3 layers, it would make no sense if the top layer has more than 2 nodes. Since its predecessor C has 4 replicas and D performs a reduction job, each D node in the top layer should process the data from at least 2 replicas of C. To simplify the algorithm, all aggregation trees are binary trees, so each node will reduce outputs from two of its predecessors.

Figures 1 and 3 show two different workflow expansions from our algorithm. The DAG in Figure 1 is the Word Count workflow given $V = 5$, whereas the DAG in Figure 3 is a more complex example with $V = 10$.

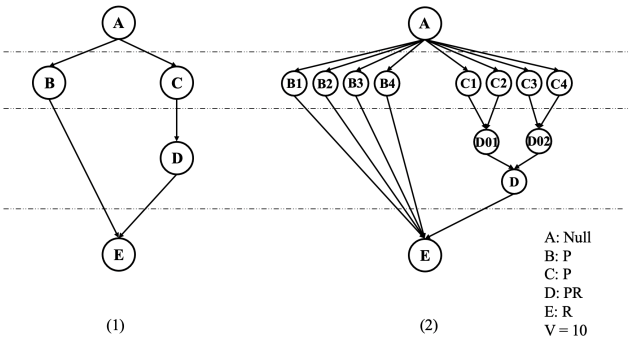

Fig. 3. Another example of a workflow expansion

C. Deployment architecture

ODESSA relies on a master-worker architecture to schedule and deploy scientific workflows in the cloud. A single master and multiple workers run on the provisioned cloud VMs. The master picks the next runnable task, dispatches its execution on a worker VM, and awaits task completion feedback from all workers. Workers receive the dispatch from the master, fetch the input data for execution, run the task executable, store the generated output locally, and report to the master after the completion of a task. We assume a failure-free environment; fault tolerance is out of scope for this paper. We do ensure some reliability for master-worker communications by implementing them on top of TCP connections.

The master controls the execution of the entire workflow. It maintains three data structures for this purpose: (1) a model of the expanded DAG, that combines the list of tasks and the mapping of edges produced by our graph expansion algorithm; (2) a table that lists the location of every piece of data among the cloud VMs; and (3) another table that lists the status of every worker VM (*idle* or *running*) and the tasks they are currently executing.

The master uses all of this information to schedule *ready* tasks in the expanded DAG onto worker VMs for execution. A task is *ready* when all its predecessors are complete. Our definition of a ready task creates many synchronization points in the workflow, so load-balancing is crucial for a scalable deployment. For this purpose, the master's scheduling algorithm takes a random walk on a randomly-generated directed graph. We adopted this strategy from [2], because it distributes tasks uniformly among the VMs. An important benefit of this strategy is its low computational cost: it runs locally on the master, and doesn't require any monitoring of the status of the VMs.

The master continuously runs two threads for controlling and monitoring the execution of the workflow: one for scheduling the tasks to worker VMs (*assign*), and one for receiving feedback from the workers (*report*).

The *assign* thread runs a loop that ends when all tasks in the DAG are `complete`. Upon each iteration, the master executes the following steps. (1) Among all the ready tasks, randomly pick one: t. (2) Run the Random Walk scheduling algorithm. The algorithm returns a machine m. (3) Refer to the DAG to get the predecessors of t, $\{p_i\}$, and the number n of t's successors. (4) Refer to the data location table to get the required filenames and the locations according to $\{p_i\}$. (5) Send m the following information: the executables of t, the value of n, and the filename-IP mapping. (6) Set t's `state` and `machineID` to `running` and m in the DAG. (7) Set m's `state` and increment its `running_tasks` in the VM status table.

The *report* thread also loops until the completion of all tasks. Upon each iteration, the master executes the following steps. (1) Listen and receive data from the workers. The data obtained from a worker includes the names and the locations of the files that it just generated when executing a task. (2) Update the data location table with the new files. (3) Set t's `state` to be `completed`. Decrement m's `running_tasks` by 1. (4) Update the status of the VMs according to the Random Walk scheduling algorithm.

Workers are the VMs that execute tasks. Unlike the master node, these worker nodes are not in charge of planning and tracking the execution. Upon receiving a dispatch to run a task from the master, a worker communicates with other worker nodes that execute the predecessor tasks, fetches the desired data, runs the tasks by itself, splits the generated output into multiple files according to the number of successors, and stores the files locally.

In order to manage their behaviors, we implement the following functions in each worker:

`recv_msg`. This function opens up a TCP socket and listens to connections from the master as well as the peer workers. A worker will receive three types of messages and behave accordingly. When it receives a dispatch message from the master indicating that it must run a task, it adds the task to the waiting queue. When it receives a message from a peer worker asking for data, it puts the corresponding file in the sending queue (see `send_msg` below). Finally, when it receives data from a peer worker, it stores the data locally.

`send_msg`. Every worker maintains a sending queue of outgoing messages. A worker sends three types of messages: (1) notification of a completed task to the master, (2) data request to a peer worker, and (3) emission of requested data to a peer worker. Each of these three events leads to a new job in the sending queue. This function constantly checks the send

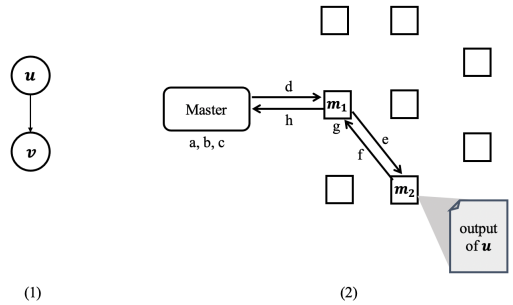

Fig. 4. A Summary of Our Framework's Architecture

jobs in the queue, connects to the destinations, and transfers the messages.

`read`. Input data may originate from multiple replicas of a predecessor task that is parallelizable. For example, in the pre-expansion DAG of Figure 3, E's task executable only expects 2 inputs, one from B and one from D. However, E has 5 predecessors after the expansion. Thus the data from B_1 to B_4 must be merged as if generated by a single B task. This function carries out this operation, processing fetched data and merging it according to the task names of the predecessors.

`write`. By default, a task executable generates only one output right after the execution. However, if the task has a P/PR node as its successor, it might have to divide and name its output in order for the successor replicas to fetch the data. Thus, `write` splits the output after executing the task executable according to how many successors this task has.

`task_handler`. This function is a forever-running thread that constantly pops a task from the task waiting queue, waits until input data exists locally, calls `read` to merge the data, executes this task, and calls `write` to split the generated output.

With the above functions, three threads run `recv_msg`, `send_msg`, and `task_handler`, respectively, controlling the behavior of each worker VM.

Figure 4 summarizes the whole master-worker architecture. Assume the ready task v has only one predecessor u. u is executed on one VM m_2, so $m2$ stores the output of u locally. The master schedules v to another VM m_1, so m_1 needs the output of u as v's input. The round rectangle represents the master, whereas the squares represent the workers. The main actions and interactions between the master and the worker are as follows: (a) the master randomly picks a ready task, v; (b) the master schedules v to m_1; (c) the master obtains the information to execute v; (d) the master send the information to m_1; (e) m_1 asks for the output of u from m_2; (f) m_2 sends m_1 the data; (g) m_1 executes v; (h) m_1 reports to the master after v is done.

IV. Performance Evaluation

We implemented a prototype of *ODESSA* and deployed it on top of the Aliyun[3] Elastic Computing Service (ECS) as a proof of concept. This section presents a performance

[3]https://www.aliyun.com

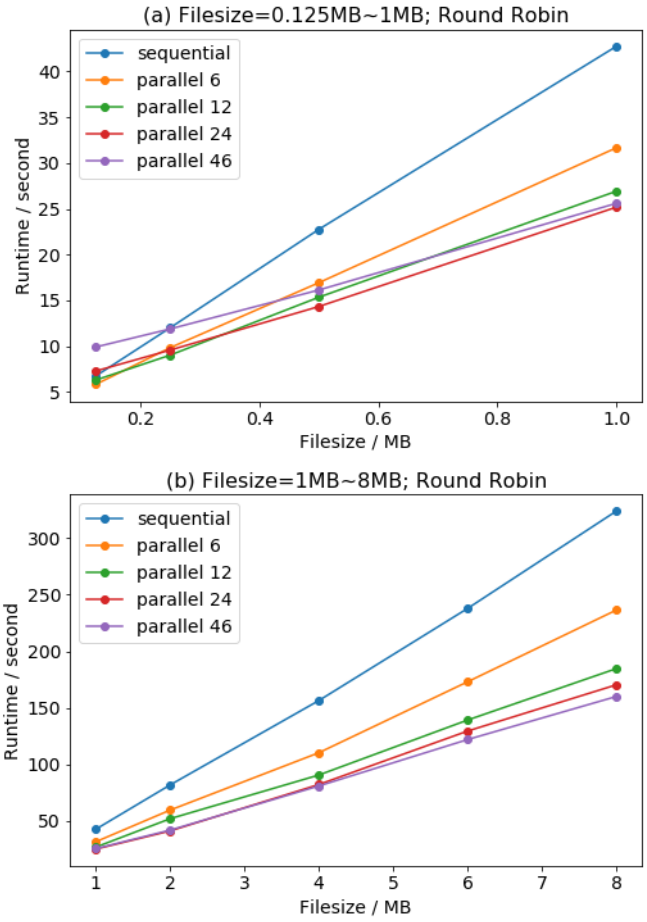

Fig. 5. Runtime Comparison

evaluation when solving the Word Count problem described in Section II. Each virtual machine (VM) sports 2 vCPU and 4 GiB of storage (ecs.t6-c2m1.large). Each measurement is an average of 13 runs, with the maximum and minimum values removed to eliminate potential outliers.

Our first experiment aims to demonstrate the scalability of our framework. We test our implementation with increasing input file sizes: 0.125, 0.25, 0.5, 1, 2, 4, 6, and 8 MB. The baseline of the experiment ('sequential') executes all three tasks sequentially on three different worker VMs, without introducing `@parallel` nor `@reduce`. We compare the baseline results with executions of the same workflow over 6, 12, 24, and 46 worker VMs: thus with different numbers of map and reduce tasks automatically assigned by our Graph Expansion algorithm. In this setup, our framework uses a round robin strategy to schedule tasks on VMs; the point is to limit the influence of load balancing on our results, and hence to focus on the benefits of our graph expansion algorithm. Figure 5 plots the time (in seconds) it takes to complete an experimental run as a function of the data input size. Our framework performs well: distributing the computation over an increasing number of VMs does improve the performance for large input sizes. The performance improvement is linear, which means our deployment is likely to scale up well. As

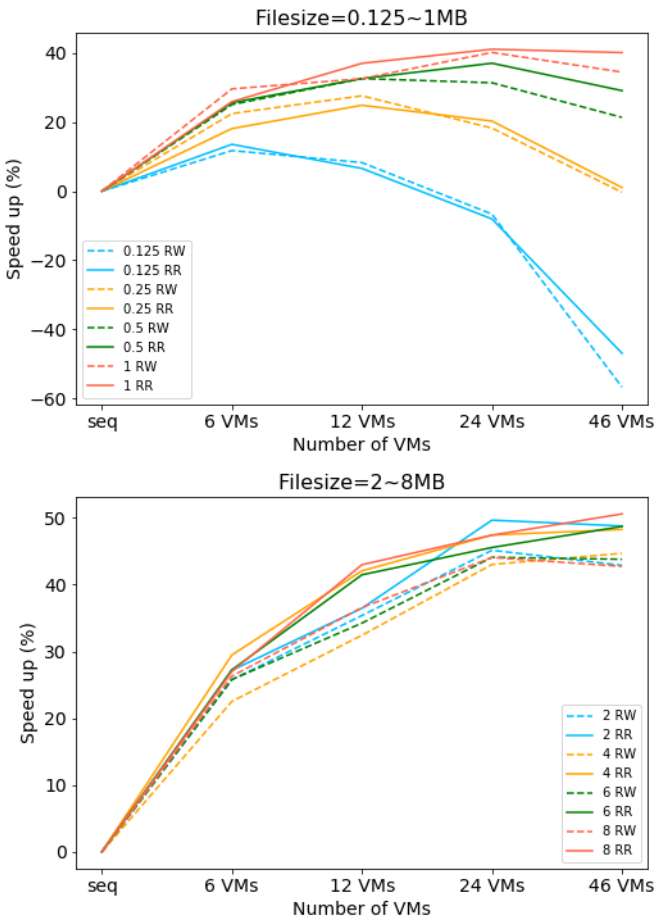

Fig. 6. Scheduling Algorithms Comparison

expected, however, the overhead introduced by scaling and transferring data between 24 and 46 VMs overburdens the computation times for smaller inputs. This is not a concern in our opinion, because the logical aim of cloud deployment is to handle ever larger datasets.

The goal of our second experiment is to assess our scheduling strategy based on the Random Walk. We set the value of $\alpha = 0.7$, and implement an alternative round robin strategy as a baseline for comparison. Figure 6 shows the resulting speedups for both scheduling algorithms, as we run our application on an increasing number of VMs, for increasing input sizes. This figure gives a basic idea of how many VMs the user should provision in order to improve runtime with respect to different file sizes. While the overhead becomes prohibitive for larger scale provisioning (24 and 46 VMs) for small input sizes, the performance speedup quickly increases with the input size. Also, note that both scheduling algorithms affect the runtime. Although the comparative results are extremely close, round robin generally performs better. It assigns each task with the same depth to a different VM, whereas random walk may assign multiple map/reduce tasks on the same VM, thus creating potential bottlenecks. A secondary reason for this outcome may be the homogeneity of our cloud environment setting: all VMs have the same configuration. When deployed on heterogeneous VMs, random walk might produce better results than round robin.

V. A DISCUSSION ABOUT EASE OF USE AND SIMPLICITY

In this section, we argue that *ODESSA* facilitates the process of writing cloud-ready scientific applications and of deploying them on a cloud. For this purpose, let us first define two subjective measures of the effort and knowledge required by a user. The former, *writing simplicity*, applies to describing application tasks and their relative dependencies. The latter, *ease of deployment*, focuses on spotting opportunities for parallelization and on applying them at runtime in order to improve performance.

In terms of *writing simplicity*, *SWIG* defines a workflow as a set of computation tasks and a set of directed edges to show data dependencies. No matter how complex a task, *SWIG* defines it in two to three lines. The user associates every task with (1) a distinct name, and (2) an executable file. In our opinion, figuring out these elements is trivial and straightforward. Defining edges is even simpler: the keyword `Edge` and its association with an arrow shows the direction of the data flow. We also opted for letting users write task executables in a common programming language, instead of creating a specialized language for this purpose. Our rationale is that many domain experts have some experience in computer programming: spending extra time to learn a new syntax for writing task jobs only burdens users.

SWIG uses annotations in the task definitions to tackle *ease of deployment*. Thus, the user points out potential parallelism opportunities for the runtime system. The `@parallel` annotation indicates that a task can be replicated to handle different pieces of data in parallel. Conversely, the `@reduce` annotation indicates a task that operates on and merges the results of multiple predecessor tasks. We believe that even the least deployment-savvy users know best how their application operates, so it makes sense to let them handle these two annotations. In return, *ODESSA* creates and deploys processes, and connects them with communication links along these guidelines to ensure the correctness of the data dependencies. Our graph expansion algorithm and deployment platform determine the number of replicas for each task process according to the availability of resources. We estimate this is a strong argument for *ease of deployment*. It saves considerable time and effort for the users: they can skip the tedious chore of dimensioning their workflow description to their cloud setup by defining similar parallel task multiple times. An added benefit is that it helps avoid mistakes when defining redundant tasks and edges.

We conclude this analysis with a comparison between *SWIG* and Pegasus. Consider the diamond workflow in Figure 7. Task *preprocess* splits the input `f.a` into two outputs `f.b1` and `f.b2`; task *findrange* turns each `f.b` into an `f.c`, and task *analyze* combines `f.c1` and `f.c2` to produce `f.d`. Listing 1 shows how to write this diamond workflow with *SWIG*, while Listing 2 does the same but with Pegasus's API[4].

[4]https://pegasus.isi.edu/documentation/user-guide/creating-workflows.html

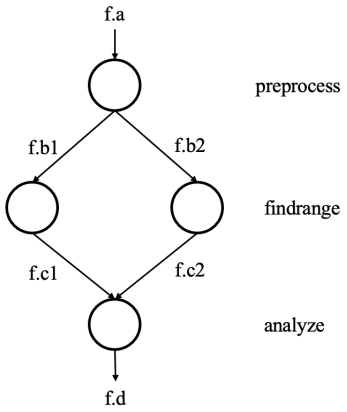

Fig. 7. The Diamond Workflow

Since *SWIG* doesn't support metadata injection and email notifications, we removed the corresponding calls in Listing 2. With Pegasus, the user must manually define *job_findrange_1* and *job_findrange_2* to run two findrange jobs in parallel. The user must also explicitly designate files for data transfers (fa, fb1, fb2, fc1, fc2, fd), and then assign them as job input/output. *ODESSA* handles this all by itself: given an appropriate number of resources (e.g. 4 worker nodes), it will automatically modify the original workflow by generating two replicas of task *job_findrange* and feeding them the output of *job_preprocess*. We acknowledge that while the `add_args` API may add more complexity to the workflow creation, it does give the user more flexibility to execute task programs. Our approach prioritizes simplicity over flexibility in this case.

Listing 1. Writing the Diamond Workflow with *SWIG*

```
Task job_preprocess {
    run preprocess
}

Task job_findrange {
    run find_range
        @parallel
}

Task job_analyze {
    run analyze
}
Edge job_preprocess -> job_findrange
Edge job_findrange -> job_analyze
```

Listing 2. Writing the Diamond Workflow with Pegasus

```
fa = File("f.a")
wf = Workflow("diamond")

fb1 = File("f.b1")
fb2 = File("f.b2")
job_preprocess = Job("preprocess")\
                 .add_args("-a", "preprocess", "-T",\
                 "3", "-i", fa, "-o", fb1, fb2)\
                 .add_inputs(fa) .add_outputs(fb1, fb2)

fc1 = File("f.c1")
job_findrange_1 = Job("findrange")\
                  .add_args("-a", "findrange", "-T",\
                  "3", "-i", fb1, "-o", fc1)\
                  .add_inputs(fb1) .add_outputs(fc1)

fc2 = File("f.c2")
job_findrange_2 = Job("findrange")\
                  .add_args("-a", "findrange", "-T",\
                  "3", "-i", fb2, "-o", fc2)\
                  .add_inputs(fb2) .add_outputs(fc2)

fd = File("f.d").add_metadata(final_output="true")
job_analyze = Job("analyze")\
              .add_args("-a", "analyze", "-T", "3",\
              "-i", fc1, fc2, "-o", fd)\
              .add_inputs(fc1, fc2) .add_outputs(fd)

wf.add_jobs(job_preprocess, job_findrange_1,\
job_findrange_2, job_analyze)
wf.write()
```

VI. Related work

Many solutions aim to assist the development and the deployment of scientific workflows. In this section, we compare and discuss the solutions we feel are most relevant, along four metrics: writing simplicity, ease of deployment, applicability, and performance/scalability.

Whether represented as a DAG or a script, a workflow can be seen as a set of tasks connected with data dependencies. So our definition for *writing simplicity* (see Section V) applies to all solutions equally. MapReduce [1] and Fly [3] seem the simplest. However, their simplicity leads to strong limitations in terms of applicability: it restricts the kinds of workflows they can support. Pegasus [4] and WorkflowDSL [5] allow users to implement the tasks by themselves and call a set of specialized APIs to define the tasks and dependencies. Snakemake [6] relies on its own language to implement different tasks and data transfer ports. Other solutions are more complex for different reasons: Swift [7] needs different built-in file types for transferring the data, and Dryad's [8] rules to create data transfer channels are far from straightforward. Besides normal tasks, AiiDA [9] aims to capture full provenance, even when the workflow terminates before its completion, so users need to implement failure/abort situations. AFCL [10] is the most complicated of all: users need to contend with asynchronous/synchronous task invocations, and complex data distribution rules to define a task.

Besides how much users need to know and understand about data parallelism, *ease of deployment* also includes how much effort users need to spend on deploying the tasks over distributed resources. Swift, Skitter [11], and AFCL offer the easiest deployments: users only need to specify whether a task can be executed in parallel, and the framework takes care of everything. Pegasus, MapReduce, WorkflowDSL, Fly, and Snakemake require more skill: they let their users spot parallelism opportunities, determine the degree of parallelism, and build the input DAG/script accordingly. Dryad and AiiDA are the hardest to deploy with. Dryad lets users specify which tasks get executed within the same process or machine. AiiDA executes tasks on local machines by default, so deploying tasks on remote resources requires many complex extra steps.

Workflows are built for different purposes and structures. The existing solutions may not be able to execute all kinds of workflows. The *applicability* metric evaluates each solution's

TABLE I
A SUBJECTIVE COMPARISON OF RELATED SOLUTIONS

Solution	Writing Simplicity	Ease of Deployment	Applicability
Pegasus	++	++	+++
Swift	+++	+	+++
MapReduce	+	++	++
Dryad	+++	+++	+++
WorkflowDSL	++	++	+++
Skitter	+++	+	+++
Fly	+	++	++
Snakemake	++	++	+++
AFCL	++++	+	+++
AiiDA	+++	+++	+++

capacity to execute various types of scientific workflows. MapReduce and Fly execute scientific workflows that follow a particular two-phase structure. The first phase speeds up via parallelization, and the second phase aggregates the results of the first phase. All the remaining solutions can run an unlimited range of scientific workflows.

All of the solutions we consider here are scalable. Different solutions adopt different techniques to achieve scalability and enhance performance. Pegasus implements task clustering and Swift uses a monitoring scheduling algorithm to enforce load balancing. Dryad enhances its performance by (1) asking the developers to specify whether some tasks can be grouped to avoid the data transfer cost and (2) refining the input workflow during runtime. MapReduce and AiiDA are scalable because the number of machines for task execution can scale on demand. Skitter's scalability is attributed to automatic task replication to achieve data parallelism. Fly and AFCL scale the workflow according to the user's indication of whether there is parallelization in some parts of the workflow. Snakemake is scalable on a single machine because of its optimization algorithm that determines how many processes should run a task. Finally, WorkflowDSL achieves scalability because the engine it relies on supports auto-scaling.

Table I sums up our subjective assessment of the related work with respect to our first three metrics. Since a workflow is composed of computation tasks and data dependencies, we believe a DAG is the simplest option. Indeed, most of the solutions assessed in this section expect a DAG as input. Upon executing scientific workflows, there is usually a trade-off between the writing simplicity and the ease of deployment. While *SWIG* may not be the simplest among the related work, it draws inspiration from Dryad and Skitter to maximize the ease of deployment. There is also a balance between writing simplicity and applicability. The simplest languages lack applicability for workflows with complex data distribution. *ODESSA* is our attempt to achieve a healthy balance between all of these aspects.

VII. CONCLUSION

This paper introduces *ODESSA*, a scalable scientific framework[5] that helps domain scientists deploy and execute data-intensive workflows on the cloud. Our main contributions are the following: a simple DSL that helps scientists write cloud-ready applications as DAG workflows; a graph expansion algorithm that automates the cloud scale-up of the input workflow and the exploitation of data parallelism opportunities; and a master-worker implementation that assures the correctness of the deployment and achieves its scalability. Our experiments, executing the Word Count problem with a prototype of *ODESSA*, prove its scalability both in terms of input size and in terms of network distribution.

There are several perspectives to pursue in the wake of this paper. One of them aims to improve *ODESSA*'s provisioning of idle nodes. By default, we set parameter α to 0.7, but its optimal value may vary with the behavior of the application and of the network. We intend to study means of determining and adopting the best value for α at runtime. Another track is the design of an input/output analysis tool to improve the partitioning and sharding of application data. A final perspective is the integration of fault-tolerance mechanisms to *ODESSA* so as to recover from VM crashes and data losses.

[5]Our prototype implementation of *ODESSA* can be found at https://github.com/VeraLeyiSun/odessa-framework.git

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI'04: 6th Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004, pp. 137–150.

[2] M. Randles, D. Lamb, and A. Taleb-Bendiab, "A comparative study into distributed load balancing algorithms for cloud computing," in *2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*, 2010, pp. 551–556.

[3] G. Cordasco, M. D'Auria, A. Negro, V. Scarano, and C. Spagnuolo, "Toward a domain-specific language for scientific workflow-based applications on multicloud system," *Concurrency and Computation: Practice and Experience*, 2020.

[4] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, p. 219–237, 2005.

[5] T. Fernando, N. Gureev, M. Matskin, M. Zwick, and T. Natschläger, "Workflowdsl: Scalable workflow execution with provenance for data analysis applications," in *42nd IEEE Computer Software and Applications Conference (COMPSAC)*, vol. 01, 2018, pp. 774–779.

[6] J. Köster and S. Rahmann, "Building and documenting workflows with python-based snakemake," in *German Conference on Bioinformatics, Jena, Germany, 2012*, 2012, pp. 49–56.

[7] 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.

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM European Conf. on Computer Systems 2007*, vol. 41, no. 3, New York, NY, USA, 2007, p. 59–72.

[9] M. Uhrin, S. P. Huber, J. Yu, N. Marzari, and G. Pizzi, "Workflows in aiida: Engineering a high-throughput, event-based engine for robust and modular computational workflows," *Computational Materials Science*, vol. 187, 2021.

[10] S. Ristov, S. Pedratscher, and T. Fahringer, "Afcl: An abstract function choreography language for serverless workflow specification," *Future Generation Computer Systems*, vol. 114, pp. 368–382, 2021.

[11] M. Saey, J. D. Koster, and W. D. Meuter, "Skitter: A dsl for distributed reactive workflows," in *5th Int. Workshop on Reactive and Event-Based Languages and Systems*, 2018, p. 41–50.