Deploying High Throughput Scientific Workflows on Container Schedulers with Makeflow and Mesos

Chao Zheng, Ben Tovar and Douglas Thain
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46530, USA
Email: czheng2,btovar,dthain@nd.edu

***Abstract*—Workflows are a widely used abstraction for describing large scientific applications and running them on distributed systems. However, most workflow systems have been silent on the question of what execution environment each task in the workflow is expected to run in. Consequently, a workflow may run successfully in the environment it was created, but fail on other platforms due to the differences in execution environment. Container-based schedulers have recently arisen as a potential solution to this problem, adopting containers to distribute computing resources and deliver well-defined execution environments to applications. In this paper, we consider how to connect workflow system to container schedulers with minimal performance loss and higher system efficiency. As an example of current technology, we use Makeflow and Mesos. We present five design challenges, and address them by using four configurations that connecting workflow system to container scheduler from different level of the infrastructure. In order to take full advantage of the resource sharing schema of Mesos, we enable the resource monitor of Makeflow to dynamically update the task resource requirement. We explore the performance of a large bioinformatics workflow, and observe that using Makeflow, Work Queue and the Resource monitor together not only increase the transfer throughput but also achieves highest resource usage rate.**

***Keywords*-Workflow; Container; Scheduling**

I. Introduction

Workflows are a widely used abstraction in scientific computing. A workflow is often described by a directed acyclic graph (DAG), consisting of sequential tasks that are joined together by the data that they create and consume. The DAG as a whole can be treated as a single highly parallel program. There exist a variety of workflow systems that share common principles but are fall into several distinct communities and use cases, including HPC machines, commercial clouds, and grid computing systems [1], [2], [3], [4], [5], [6], [7], [8].

Most scientific workflows focuse on data interchange among tasks, while not specifying what **environment** each task of the workflow is expected to run in. A given task may depend on a particular operating system, software installation, and available data to run. Normally, end users will build workflows locally and test on a particular environment, and then migrate workflows to another cluster only to find that some tasks may fail because of incompatibilities with the cluster environment, which results in arduous debugging.

A number of container-based scheduling platforms like Amazon ECS [9], Google Kubernetes [10], and Apache Mesos [11] have emerged. Through implementation details vary, they share a same basic idea: containers can be used to encapsulate the different environments and resources needed by tasks. As one example, Mesos allows users to deploy customized environments on a cluster by using Mesos containers or Docker containers. It employs a two level scheduling model, which enables users to develop their customized schedulers for their applications and allows master to assign resource offers to proper applications in needs. By adopting Mesos, end users can ship a relatively small container that acts like a complete operating system with lower overhead, and cloud vendors are able to deliver computing resources to applications in a fine-grained manner.

Even though container schedulers have been widely adopted to run commercial workflows, there are few use cases about using them for large-scale scientific computations. Scientific workflows share many features with commercial workflows, but also go beyond them. Therefore, existing settings of workflow systems may be inadequate for running large scientific workflows on container scheduler. **In this paper, we explore the possibility of running scientific workflows on the container-based scheduling platform with lower performance loss and higher system efficiency**. As an example, we use specific technologies of the Makeflow [5] workflow engine, the Work Queue [12] execution system, and the Mesos [13] container-based scheduling system. We highlight several challenges encountered, and illustrate the general principles that can be applied across technologies.

Specifically, we consider four different configurations of running workflows on Mesos. (1) Launching workflows directly on Mesos from Makeflow, which is straightforward but has inefficient resource usage and low transfer throughput. (2) Running Work Queue workers on the Mesos cluster, and making these workers to execute tasks for Makeflow, which reduce the data transfer overheads by reusing existing TCP connections and relief the disk pressure by deleting the intermediate data. The other two configurations rely on the resource monitor to make the best use of Mesos' fine-grained resource sharing feature. (3) Running Makeflow directly on Mesos with the resource monitor enabled. (4) Launching Makeflow with Work Queue on Mesos with the resource monitor enabled. By using resource monitor, Makeflow can dynamically updates the resource requirement of each task during the runtime, which enable task to send precise resource demands to Mesos master.

To evaluate these configurations, we launch a large bioinformatics workflow that consists of many short and few long tasks. We choose this workflow because it contains a highly parallel phase, which has 5079 concurrent tasks and transfers 10148 small files. Based on the benchmark results, we notice that Work Queue can help to avoid resource starvation with fewer users involvements and increase the average transfer throughput. And resource monitor can help to increase the resource usage of Makeflow, which largely reduces the overall execution time when the amount of available resources is fixed.

II. Background

A. Scientifc Workflows

A workflow is often represented as a Directed Acyclic Graph (DAG) where the nodes of the graph are tasks to execute, and the edges of the graph represented dependencies between tasks. The dependency is usually in the form of a file, created by one task and consumed by another.

In this paper, we use Makeflow to generate the DAG for representing the structure of workflows. Makeflow is a workflow description tool similar to classic Make build tool [14]. Each task in the workflow is defined as a rule, annotated with the output files it generates and the input files it requires. For example, consider a task, the result file is *output.1*, inputs are *target.csfasta, seq.1.csfasta* and the executable file *rmapper*. The command is *./rmapper seq.1.csfasta target.csfasta* > *output.1*. In the Makeflow file, one would write it as following:

```
output.1: seq.target subseq.1 rmapper
    ./rmapper subseq.1 seq.target > output.1
```

Makeflow can launch the same workflow across many batch systems, like HTCondor [15], Sun Grid Engine (SGE), Torque, SLURM, Amazon EC2 and Work Queue. Regardless of the system in use, Makeflow expects them to work as follows. For each task in the workflow, an independent sandbox is created, the inputs are transferred into the sandbox, the task is executed and the outputs are moved out from the sandbox. This semantics make sure that files other than the input dependencies would not presented in the execution environment, and Makeflow dispatching tasks as their dependencies are presented.

B. Container Schedulers

Recently, container-based scheduling platforms have been widely adopted. Though the implementation details of different platforms may vary, they all use containers to encapsulate computing resources and deliver user-defined execution environment. Containers are implemented by mounting filesystems on top of an existing operating system kernel, which largely eliminating the overheads found in traditional virtual machines. Recently various container runtimes have been considerable developed in the Linux community by combining the `cgroups` resource control framework and the `unionfs` filesystem management to provide comprehensive isolation.

In this paper, we use Mesos as an example of cluster container scheduler. As one of the most popular container-based resource management systems, Mesos deliver resource offers to tasks in need by using containers. Some container schedulers apply monolithic resource scheduling architecture, like Kubsernetes and Docker Swarm. Differently, Mesos adopts a two-level scheduling model that enable each application to have its own customized scheduler. This scheduler receives resource offers from Mesos master and try to find a matched task for the offer. If a task is found, it will claim the offer and ask Mesos to launch the task on the agent provide this offer. Since a long task can occupied many offers, Mesos encourage frameworks to use short tasks. This may be a limitation for running scientific workflows on Mesos, because a scientific workflow normally contains heterogeneous tasks that can be either short or long.

C. Work Queue

Work Queue is a master-worker execution engine for distributed systems. The system includes a master process on user side, and workers running on cloud that has an amount specified by users. As an execution engine, Work Queue can deploy workers across multiple clusters, clouds and grid infrastructures. The master enable user to create tasks with a command line to execute, a number of input files, and a set of outputs. After tasks are created, the master would assign them to available workers.

The worker run tasks as follows. A cache directory is created by the worker. Input files transferred from the master are stored in this directory. For each task, the worker create a sandbox, links the inputs, executes the task, copies the outputs to the cache directory, and then the outputs will be transferred back to the master. This mechanism not only allows each task having an independent namespace but also enables different tasks sharing data through the cache directory. Thus multiple tasks can be executed concurrently by a single worker on a multicore machine.

D. Resource Monitoring

Users seldom have precise knowledge of the resources (e.g., cores, memory, or disk) needed to execute a task. More often, users are able to provide the size of a computational node where a task is known to execute to completion as a ballpark figure.

Makeflow can be directed to automatically manage the computational resources assigned to tasks, refining the resources allocated per task in order to minimize waste of maximize throughput. This management is built as resource feedback loop that considers the measurement of resources used per task, the allocation and enforcement of resources, and automatic retries on resource exhaustion.

Initially, tasks are run using a maximum allowable resource sizes (i.e., the ballpark figure provided). As tasks are completed, their real usage is measured and recorded, and allocations are computed for newly created tasks in order to minimize waste or maximize throughput. Some tasks may exhaust these computed allocations, and are retried using the maximum allowable sizes. As we present the fully details in [16], a small number of retries leads to substantial increases in throughput and decreases in resource waste.

III. Challenges With Mesos

Even though features of scientific workflows from different domains can vary largely, there are several common ones. (1) Each task of workflows is tightly coupled, often having intricate dependencies on other tasks. For example, the input of one task may be produced by other tasks. Thus, we usually divide a large workflow into several phases and execute them sequentially. (2) Scientific workflows often consist of heterogeneous tasks that can be either short or long. Therefore, running scientific workflows requires a more sophisticated resource management strategy that can handle both short and long tasks at the same time. (3) Large number of tasks are running concurrently and massive amount of intermediate data is generated and consumed during the lifecycle of workflows. This feature results in high I/O throughput and requires the underlying systems to have large storage space. (4) Resource demand of each task is unclear before execution. Many large scientific workflows are long-term, and require large amount of computing resources. However, the resource demands are often coarse-granularity, which results in low resource usage rate. These features make scientific workflows different from commercial workflows, which raise challenges for launching them on platforms that originally developed for commercial environment. Followings are five design challenges we encountered when try to connect Makeflow to Mesos.

First, a simple implementation of Mesos scheduler is not capable of synchronizing the status of workflow between Makeflow and Mesos. Due to the dependencies among different phases of the workflows, some phases can not start until they have collected inputs produced by other phases. Thus a well-designed workflow scheduler must be aware of the completion of tasks on Mesos side, and inform Makeflow to dispatch tasks for next phase.

Second, if large amount of tasks are started concurrently, all tasks will try to fetch inputs at the same time, which may cause network congestion on the client side. Most scientific workflows consists of many small tasks that can be launched concurrently, and for each task running on a Mesos agent, the agent-internal fetcher process will try to download the task input files based on the URIs provided by the end user. Even though all files requested by a single task are fetched sequentially to reduce the risk of bandwidth issues, multiple fetch operations can still be invoked simultaneously due to launching multiple tasks concurrently. Normally, all the input files of a workflow are located in a same host, which requires the host to handle hundreds of fetching requests at the same time. Thus the fetching procedure can become the bottleneck of the whole system.

Third, workflows would suffer from starvation due to long tasks. As mentioned in the original Mesos paper [13], Mesos assumes that most tasks are short. And running workflows that have long tasks can hurt other workflows running on the same Mesos cluster, because long task might occupy multiple slots that can be assigned to more short tasks. Some scientific workflows consist of fairly long task that will hold large amount of computing resources for a long time. This phenomenon may causes other workflows to suffer from resource starvation and greatly decreases the fairness of the cloud.

Fourth, delayed garbage collection can cause disks be filled up fast. By default, Mesos executors do not send results back to user after the tasks complete. It temporarily stores results in executors' sandboxes and marks them as garbage, which will be removed in the future. The default delay time for garbage collection is 1 week, and the minimum delay time can be set is 1 day. However, this is still too long for the cluster has multiple scientific workflows running on it. In our experience, 5 runs of SHRiMP workflows on Mesos cluster that has 26 nodes can produce 83GB to 119GB of intermediate data on each node in just one day. Normally, there are far more than 5 workflows being launched on the cluster every day, which can generate massive amount of data that fill disks to full quickly. Even though the lifetime of a sandbox can be set to shorter than 1 day by setting a threshold of maximum disk usage, one should remember that Mesos may serve other workflows that prefer to keep results on the cluster at the same time. Thus simply shorten the period of time between garbage collection cycles may cause malfunction of other workflows.

Fifth, without providing a precise resource requirement for each task, we can not take full advantage of the fine-grained resource sharing feature of Mesos. Often, the resource requirement of each task of a scientific workflow does not change during task lifetime, but it is unknown before runtime. One of the common used resource schedulers in HPC center is the monolithic coarse-grained scheduler, which has one central resource manager that is responsible for allocating resources for all applications. The resource allocation policy of a coarse-grained scheduler is mainly based on the task priority. That is to say, instead of assigning proper amount of resources to each task before running, it changes task priority at runtime, and preempt lower priority tasks in favor of higher priority ones. Generally speaking, a coarse-grained scheduler put priority on providing high computing performance to workflows instead of optimizing resource sharing across them in a fine-grained manner, which might lead to inefficient resource use. One of the key features of Mesos is the two level scheduling model, which require users to provide resource demand for each task before execution. For running a scientific workflows, an experienced user may be able to provide a coarse-grained resource usage prediction, but it is far from optimal for Mesos master to assign resource offers to proper tasks.

IV. Proposed Solutions

To cope with above challenges, we consider four configurations for launching scientific workflows on Mesos with Makeflow and Work Queue. In order to take full advantage of Mesos fine-grained resource sharing property, we also try to use the resource monitor of Makeflow, which keeps monitoring the resource consumption of each task and provide a real-time resource usage evaluation.

A. *Makeflow and Mesos*

The simplest set up is directly connecting Makeflow to Mesos. As shown in figure 1, we use a customized scheduler to connect Makeflow to Mesos. Mesos will launch each task with an independent executor on available agent. Two daemon threads are spawned by Mesos scheduler, one is responsible for polling the file that has information of ready tasks, and another one start an HTTP server for handling file fetching requests from executors. Since some of the tasks share same inputs, to relief the network pressure, we enable Mesos to cache inputs on agent nodes. But it is important to note that the current version of Mesos can not refresh the cache entry and it only uses URIs to decide whether the files have been cached. Thus, if the contents of input files have changed, tasks can fail due to the usage of stale input data. We implement a batch job system specially for Mesos that compatible to Makeflow. One would launch Makeflow with Mesos by specifying the batch mode as Mesos. For example, by executing this command:

```
makeflow -T mesos --mesos-master=localhost:5050
```

user start the Makeflow with underlying batch system set as Mesos. Then, Makeflow will try to connect to the Mesos cluster with master located at the local host and listening on port 5050.

The details of how Makeflow work with Mesos are shown in figure 1: (1) Makeflow writes the information of ready tasks to a file. (2) The task information monitor get information of ready tasks from the file. (3) Mesos scheduler get resource requests of each task. (4) Mesos agents send resource offer to Mesos master. (5) Mesos master advertise resource offers to schedulers. (6) Scheduler matches offers with proper tasks, and launch an executor on agents that provides offers. (7) Before running tasks, executors retrieve inputs from client or fetch from cache directory. (8) After tasks have completed, executors send output URIs to scheduler. (9) Scheduler download outputs from executors sandbox directory and ask executors to delete the outputs. (10) Scheduler write information of finished tasks to file. (9) Makeflow keeps checking whether there are new finished tasks. If yes, Makeflow mark the tasks as completed.

This configuration is straightforward and resolves some of the design challenges mentioned above. It uses one daemon thread with scheduler and one sub-process with Makeflow to synchronize the Workflow status between Makeflow and Mesos. To resolve the bandwidth issue, we implement a HTTP server with a thread pool on scheduler side, which handles each fetching request by an independent thread. The HTTP server limits the number of threads to 30 and maintains a request queue to cache the incoming requests when there is no available threads. To avoid resource starvation, there are two options, one is not launching long tasks on Mesos, but running tasks on local machine by using Makeflow LOCAL mode, another way is preserved certain amount of resources for short tasks, which requires user to extend the default resource allocator. To prevent disks from being filled up quickly, an customized executor is required, which can delete intermediate results after the scheduler has retrieved them. To match the task with resource offers, user can classify tasks into various categories, and specify the resource requirement of tasks in the same category.

Even though this configuration resolves the design challenges to some degree, it has two non-negligible drawbacks. First, in order to avoid the resource starvation, users may try not to execute long-running tasks on Mesos, or reserve resources for short tasks. The first method requires users to classify tasks as short or long. If a user failed to specify long tasks, other workflows will still suffer from resource starvation. And implementing a customized resource allocator requires quite a bit of work. Second, to use this configuration, end users must specify the resource requirement of each task, which may be crude and imprecise. A conservative user may provide a conservative resource evaluation, which leads to longer execution time. Conversely, bold users always raise greedy resource claims that can cause inefficient resource usage.

Without knowing the features of each task, there is no effective way to address the first problem. Thus we suggest that user without full knowledge of workflows should not adopt this configuration. To solve the second problem, we use the resource monitor provided with Makeflow to evaluate the resource consumption of workflow and provide a prediction.

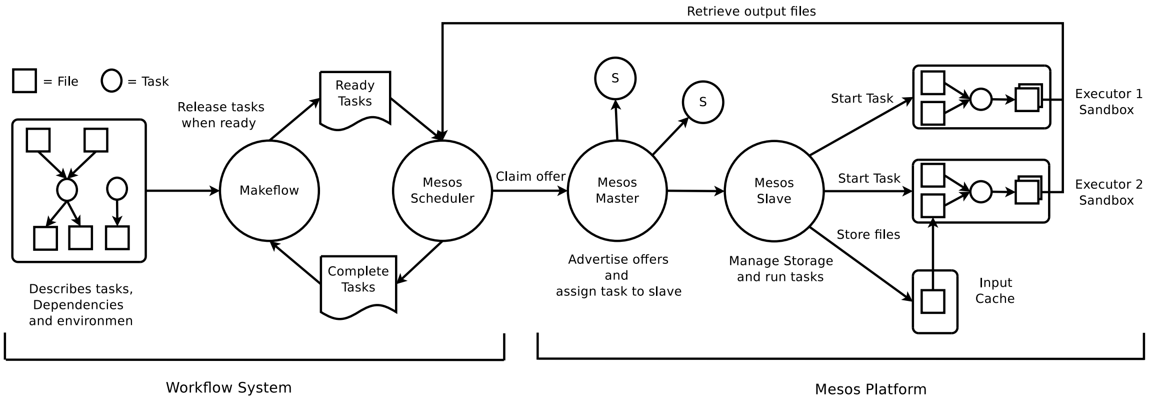

Figure 1: Makeflow on Mesos

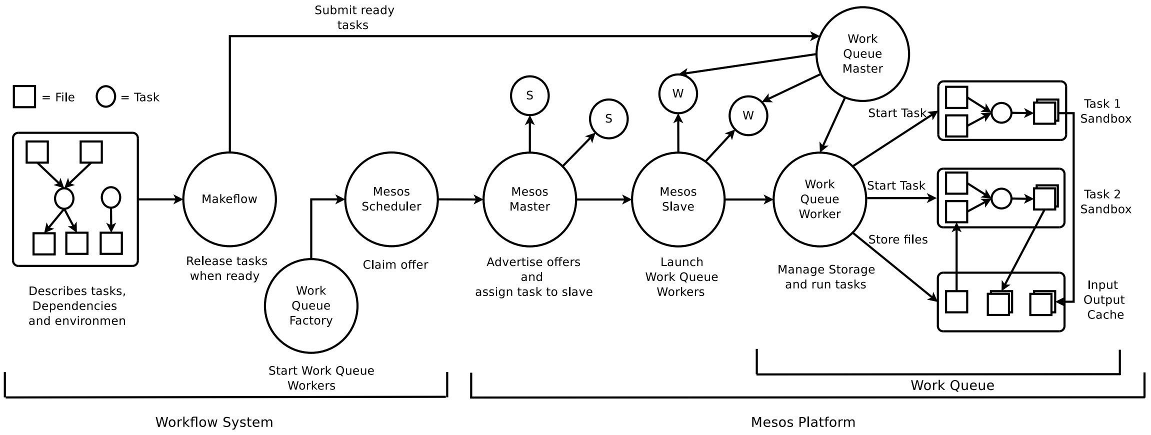

Figure 2: Makeflow and Work Queue on Mesos

B. Makeflow, Work Queue and Mesos

An alternative approach is to use Work Queue as the workflow execution layer between Makeflow and Mesos, the system structure is shown in figure 2. The main idea is that using Work Queue factory to set up a master for the workflow and have workers launched on Mesos that work for the master. The Work Queue factory is a tool for fast starting workers on different batch systems. It shares the same batch job interface with Makeflow, thus we can reuse the existing batch job system developed for Mesos. To launch Work Queue on Mesos and connect Makeflow to Work Queue, one would use following commands:

```
makeflow -T wq -N shrimp_workflow

work_queue_factory -M shrimp_workflow -T mesos
                   --mesos-master=localhost:5050
```

First, we start Makeflow and specify the underlying batch system as wq (i.e. Work Queue). We also define a project name for this workflow. This project name will be used later to match Work Queue to Makeflow through a catalog server. Second, we run work_queue_factory and specify the underlying system as Mesos. Then, Work Queue will try to access the Mesos cluster through Mesos master that is located at the local host and listens on port 5050.

The details of how this configuration work are shown as follows: (1) After starting Makeflow, Work Queue factory create a Work Queue master. (2) The Work Queue master is linked to the Makeflow through our catalog server. (3) Work Queue master writes information of the ready workers to a file. (4) Worker monitor keeps polling the text file, get information of ready workers. (5) Mesos scheduler submits tasks of launching workers for Mesos master. (6) Mesos agents send resource offers to Mesos master. (7) Mesos master advertise resource offers to the scheduler. (8) The scheduler matches resource offers to proper workers, and then launches workers on the agent provided offers. Each worker is treated as a task and run in an executor. (9) Work Queue workers work for the master, execute tasks for the workflow, and Work Queue master informs the Makeflow the completion of tasks. (10) After the workflow has completed, workers are deactivated and Mesos scheduler writes the information of the workers to a file. (11) Worker status monitor keeps checking the list of deactivated workers and informs the factory to remove the idle workers

This approach not only overcomes the design challenges but also addresses the weaknesses of the first configuration. To resolve the resource starvation problem, Mesos can limit the amount of resources available to each workflow by starting workers with certain amount of resources, and assign spare resources to workflows in need. Since Work Queue extends the semantics of Makeflow, Mesos is transparent to Makeflow and Work Queue can delete all intermediate data for Makeflow, which does not require sophisticated executors.

But this approach also presents new problems: how many resources does each worker require? If the resource requirement of each task is unknown, how many tasks a worker can run at the same time? Ignoring these two problems can cause improper task arrangement. Imagining two bad use cases: The number of tasks per worker is too small, which can lead to inefficient resource usage and reduced throughput. The number of tasks per worker is too large, which can causes resource contention and reduced performance.

C. Enabling Resource Monitoring

Both configurations mentioned above are using coarse-grained resource predictions to match tasks to resource offers, which requires user to have full knowledge of the workflow. While, the resource prediction provided by user are often far from optimal, thus it is nearly impossible to ensure full use of resources.

Therefore, we enable Makeflow's resource monitor to measure the resource usage at the runtime and update the resource requirements of tasks dynamically. The system works as usual except that each makeflow task will be run with a resource monitor thread. At the beginning, we launch several tasks with the maximum allowable resource size or with 1 task per worker for collecting resource consumption data. Based on the sample of resource usage, we apply the Slow-peaks model [16] to predict the exact resource requirements of tasks. If Work Queue is used, the accurate resource requirements can help us to assign max number of tasks to each worker without resource waste and exhaustion.

A sample Makeflow file with rules for resource monitor is shown as following:

```
.MAKEFLOW CATEGORY local_split

subseq.1 .. subseq.5079: seq.inp splitreads.py
    LOCAL python splitreads.py 5079 seq.inp

.MAKEFLOW CATEGORY remote_map
.MAKEFLOW MODE MIN_WASTE

output.1: seq.target subseq.1 rmapper
    ./rmapper subseq.1 seq.target > output.1
    ...
output.5079: seq.target subseq.5079 rmapper
    ./rmapper subseq.1 seq.target > output.1

.MAKEFLOW CATEGORY local_combine

output: output.1 .. output.5079 combine.sh
    LOCAL ./combine.sh
```

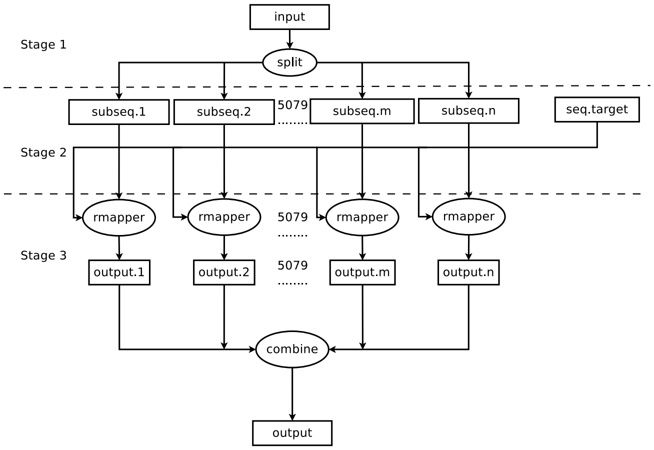

Figure 3: Shrimp Workflow

We classify tasks into three phases, (1) *local_split*, which contains one task that splits the input sequence into 5079 sub-sequences. This phase is run locally without resource reinforcement. (2) *remote_map*, which consists of 5079 tasks with each aligning one of the sub-sequences to part of the target sequence. We run this phase on Mesos with the MIN_WASTE mode enabled, which will start a reinforcement loop that uses a small number (100 by default) of tasks to collect resource consumption data and update the resource requirements at the runtime. If new resource requirements do not suit the needs of tasks, the reinforcement loop will be activated again. (3) *local_combine*, which combines all the results of the second phases and generates the final output.

V. Experimental Evaluation

A. Experimental Setup

To evaluate both of the configurations, we implemented a batch job system for mesos, connected it to Makeflow and Work Queue and provided mesos mode as options within Makeflow and Work Queue. we explore the performance of running a large bioinformatics workflow with each configuration on a Mesos cluster. The cluster contains twenty-four 8 core Intel Xeon E5620 CPUs each with 32 GB RAM, 12 2TB disks, 1GB Ethernet, running Red Hat Enterprise Linux 6.8 with Linux kernel 2.6.32-642.6.1.el6.x86_64 and Mesos 0.26.

Generally, there are three categories of scientific workflows, workflow that has mainly of long tasks, one that contains many short tasks that can be launched simultaneously, and one that includes both types of tasks. To emphasizes all possible problems can be caused by ignoring the feature of Mesos' default resource allocation and garbage collection policies, we chose SHRiMP workflow, which consists of 5081 tasks in total. SHRiMP is a genomic software package for aligning genomic reads against a target genome.

The structure of the workflow is shown in Figure 3, which contains 3 stages. Stage 1 has one long-running task that sub-samples a 313 MB input file into 5079 parts. Stage 2 compares each sub-sequence with part of the target genome that is 361 MB and generate outputs of approximately 650 KB each. The optimal task execution time of task in this stage should be no more than 300 seconds. Stage 3 combines all outputs and produces a single output file. Stage 2 is highly parallel that consists of large amount of short-running tasks. The size of intermediate results generate by each task of stage 2 is approximate to 362 MB, that is 1.8 TB in total. Thus without proper garbage collecting policy, the disk will be filled up quickly.

In order to see how Work Queue and the resource monitor affect the performance of the workflow, we consider four configurations: (1) Makeflow and Mesos (2) Makeflow, the resource monitor and Mesos (3) Makeflow, Work Queue and Mesos (4) Makeflow, Work Queue, the resource monitor and Mesos. And we expect the fourth configuration to achieve best performance. To emphasizes the problem of inefficient resource usage due to the coarse-grained estimation of resource consumption. We assume that a greedy user requires 4 cpus, 5120 MB of memory and 5120 MB disk for each task, which is superfluous. For the same reason, when using Work Queue, we assign 4 cpus, 5120 MB of memory and 5120 MB disk to each worker. We run each configuration 5 times and get approximate result for each run, thus we select one of the five result sets to compare the performance of different configuration.

B. Results and Analysis

Table I gives the key details of the total runtime, the average execution time of each task, average transfer rate between the Makeflow and the Mesos agent, and the average CPU usage rate. To emphasizes the influence of different data transfer methods on the workflow, we include the data transfer time into the task execution time. Figure 4 presents the task execution time histogram and transfer throughput histogram with rows represent configuration one to four respectively. The first column gives a histogram of individual task execution time of the 5079 tasks of stage 2, and the second column gives the histogram of transfer rate of individual file between Makeflow and the Mesos agent. In Figure 5, we show the histograms of the number of cores being allocated and the number of cores being used. The first column gives the CPU usage during the whole lifecycle of the workflows. The second column shows the CPU usage in the first 30 minutes of the workflows.

Running Makeflow directly on Mesos has the longest overall execution time (11.17 hours) and a low average CPU usage rate (0.500). As shown in the first column of the first row in figure 5, across the whole life cycle of the workflow, only half of the allocated cores are actually in use. The average task execution time is 408 seconds, which is relative long compares to the configurations that use Work Queue. This is mainly due to the overheads of repeatedly setting up TCP connections. After Mesos master assign a task to an executor, the fetcher process of the executor will try to fetch each input file of the task with an independent HTTP request. And after the task is complete, the scheduler will retrieve each output file with an HTTP get request. Since there are large amount of small files (around 10150 and each one is 70 bytes to 1 MB) transferred during the lifecycle of the workflow, there are massive amount of TCP connections set up. This can also explain why the average transfer rate (43.11 MB/S) is relative low.

As hypothesized, by using resource monitor, we increase the resource usage rates from 0.5000 to 0.976, which help us to achieve a better performance (6.7 hours). Be more specifically, as shown in the second column of the second row of figure 5, the number of cores in use is gradually increased during the first 20 minutes, and finally approximate to the number of allocated cores. Therefore, even though the average task execution time is still long, we achieve 1.67× overall performance improvement. By using Work Queue, we achieve faster and more compact task execution times. This is due to the fact that during the lifecycle of a Work Queue worker, a single TCP connection is responsible for handling all the data transfers between this worker and the Work Queue master, which reduces the overheads of setting up TCP connections repeatedly for each task and increase the transfer throughput of the whole system. Even though this configuration achieve a better overall performance, it still pays a penalty due to the waste of resources. To further improve the performance, we using Work Queue and the resource monitor together. As expected, we gain better performance in every aspect, which lead to the fastest overall execution time (5.37 hours). The only downside of adopting resource monitor is that a small file contains resource consumption is generated for each task running on the mesos agent. Then, these small files are retrieved by the main process of makeflow, which causes the slight decrease of the average transfer rate.

As can be seen, to take full advantage of a container scheduling platforms, the selection of a strategy for managing data transfers and claiming resource requirements has a significant impact to the overall performance, primarily due to the non-trivial expense of transferring large amount of small files and the difficulty of precisely evaluating the resource requirements of each tasks.

VI. Related Work

There exist various of workflow systems that share similar principles while addressing the needs of different user communities and use cases. Examples include DAGMan [2], Galaxy [3], Kepler [4], Pegasus [6], Swift [17], and Taverna [8]. The general principles and considerations presented in this paper could be applied to all of these systems.

With the growing adoption of container-based computing, researchers have started to consider harnessing container

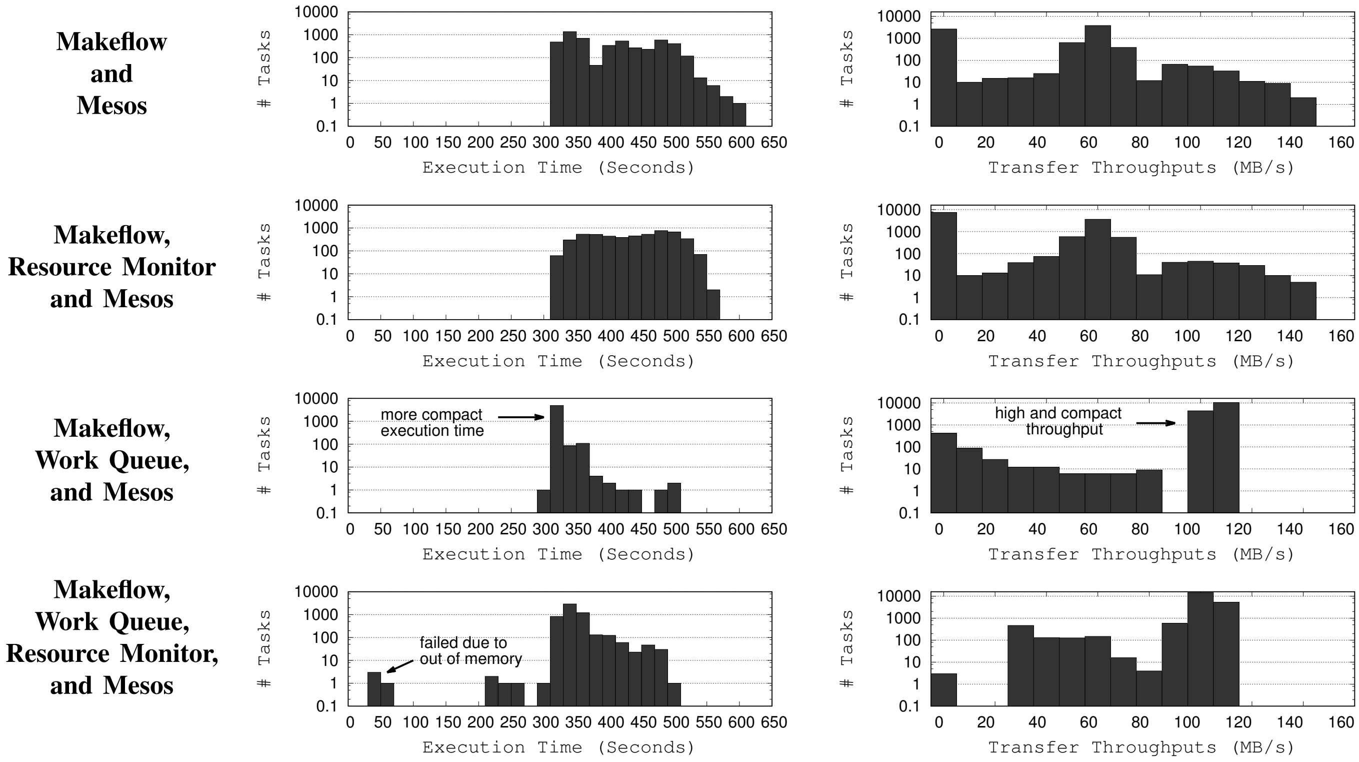

Figure 4: Task Execution Time and Transfer Rate

In each row of the table, task execution time histogram and transfer throughput histogram of each configuration are presented.

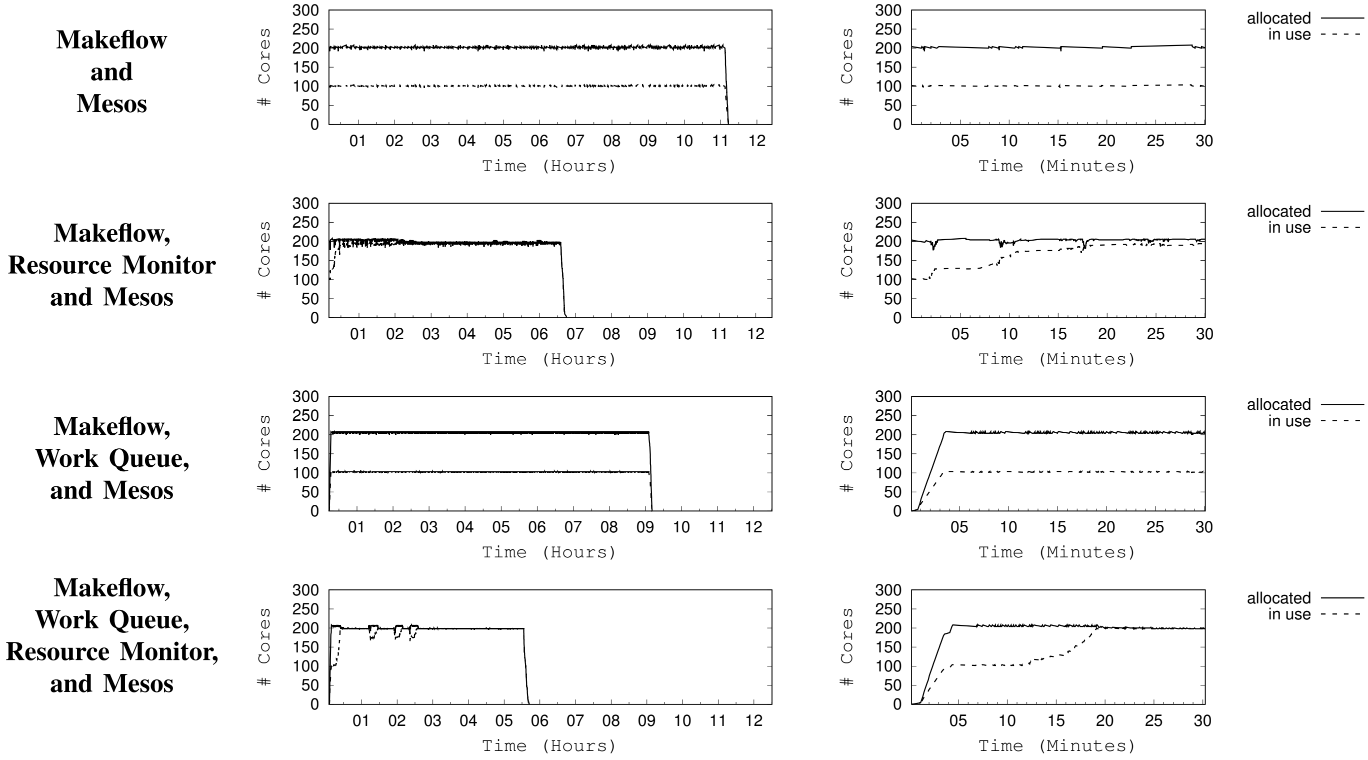

Figure 5: CPU Usage Rate

Each row shows the complete CPU usage timeline, and the CPU usage timeline of the first 30 minutes are presented. The total number of available cores is 208. The solid line depict the number of allocated cores, and the dashed line depict the number of cores in use.

	Makeflow	Makeflow Resource Monitor	Makeflow Work Queue	Makeflow Work Queue Resource Monitor
Total Exec Time (Hours)	11.17	6.7	8.97	5.37
Average Task Exec Time (Seconds)	408	445	327	355
Average Transfer Rate (MB/S)	43.11	26.88	106.87	104.66
Average CPU Usage (#used/#allocated)	0.500	0.976	0.501	0.975

Table I: Performance Summary of Each Configuration

runtimes for large-scale scientific computation. Many studies have been done for measuring the performance of container runtimes in cloud computing environment [18], [19], [20], [21]. Some other projects focus on adopting container to reduce the overheads of deploying customized computing environment. Skyport [22] as an extension of AWS/Shock, uses Docker container runtimes to automatically deploy isolated execution environment with low overhead. And in our previous work [20], we integrated Docker container runtime into Makeflow and Work Queue, which enable the tasks of a given workflow to have isolated environments without loss of performance. In this paper, we exploit the possibilities of connecting workflow system to a container orchestration platform, which not only speed up the deployment of execution environment but also improve the efficiency of resource usage of the whole cloud.

There existing a large variety of resource scheduling platforms. One popular model is the **Centralized coarse-grained scheduler**. This kind of scheduler usually have a central resource manager, which is responsible for assigning resource to all applications, well-known systems like Borg [23], TORQUE resource manager [24], old Hadoop scheduler and SGE are all fall in this category. In our previous work [5], we have enabled Makeflow to run on SGE through the batch system interface. One drawback of this implementation is that when cooperating with SGE, Makeflow does not know the resource state of the whole cluster, and can only submit tasks to the queuing system that may lead to Head-of-line blocking. In this paper, we try to connect Makeflow to Mesos, which adopts the **Two-level resource scheduling model**. By using Mesos, frameworks can have their own schedulers to claim customized resource demands.

Other well known two-level schedulers include HTCondor [15] and YARN [25]. These systems enable applications to have their own task assignment semantics, which is more flexible and allow different workloads to share resource together.

VII. Conclusion and Future Work

By adopting container-scheduler platform, users can set up customized environment for distributed applications with shorter deployment time and less overheads compare to using platforms that adopt virtual machine technologies. At the mean time, some of them, like Mesos, employ the two-level resource scheduling model, which enables various frameworks to share resource with each other. In this paper, we exploit the possibility to run large scientific workflows on Mesos. We list five design challenges and try to resolve them by employing four configurations that use Makeflow and Work Queue. We also use resource monitor to evaluate the resource consumption of workflow and update the resource requirement at the runtime, which makes good use of Mesos fine-grained resource management service. To benchmark the performance of the four configurations, we run a large bioinformatics workflow with them. The experimental results show that with resource monitor enabled, we achieves up to 2× performance and resource usage improvement, even though the resource requirements given by users are inaccurate. Therefore, we conclude that users without full knowledge of workflows are recommended to use tools, like resource monitor, to evaluate and update the resource requirements at the runtime, and by adopting an existing execution engine, like Work Queue, we reduce the risk of I/O and bandwidth issues with more system scalability and less labor.

We notices two opportunities to further improve the system performance and stability. First, there exists a relationship between the intervals of resource reinforcement loop, maximum number of running tasks and overall execution time. we intend to construct a model to achieve better settings for these three parameters. Second, one of the performance bottleneck of the current system is the limited bandwidth between Makeflow and Mesos. In the future, we plan to set up redundant services on Makeflow side by using software like ZooKeeper [26], which can helps us to expand the bandwidth and improve the system availability

References

[1] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: distributed data-parallel programs from sequential building blocks,” *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 59–72, 2007.

[2] J. Frey, “Condor dagman: Handling inter-job dependencies,” *University of Wisconsin, Dept. of Computer Science, Tech. Rep*, 2002.

[3] B. Giardine, C. Riemer, R. C. Hardison, R. Burhans, L. Elnitski, P. Shah, Y. Zhang, D. Blankenberg, I. Albert, J. Taylor *et al.*, “Galaxy: a platform for interactive large-scale genome analysis,” *Genome research*, vol. 15, no. 10, pp. 1451–1455, 2005.

[4] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*. IEEE, 2004, pp. 423–424.

[5] M. Albrecht, P. Donnelly, P. Bui, and D. Thain, "Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids," in *Workshop on Scalable Workflow Enactment Engines and Technologies (SWEET) at ACM SIGMOD*, 2012.

[6] 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*, 2014.

[7] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *Grid Computing Environments Workshop*, 2008, pp. 1–10.

[8] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher *et al.*, "The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud," *Nucleic acids research*, p. gkt328, 2013.

[9] "Amazon ec2 container service." Amazon Web Services, Inc, 2016, available at https://aws.amazon.com/ecs/.

[10] "Production-grade container orchestration." The Linux Foundation, 2016, available at https://kubernetes.io/.

[11] "Program against your datacenter like its a single pool of resources." The Apache Software Foundation, 2016, available at http://mesos.apache.org/.

[12] P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain, "Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications," in *Workshop on Python for High Performance and Scientific Computing (PyHPC) at the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing)* , 2011.

[13] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, 2011, pp. 22–22.

[14] S. I. Feldman, "Makea program for maintaining computer programs," *Software: Practice and experience*, vol. 9, no. 4, pp. 255–265, 1979.

[15] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience," *Concurrency and computation: practice and experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[16] B. Tovar, "Resource monitor and visualizer," 2016, available at http://ccl.cse.nd.edu/software/resource_monitor/.

[17] Y. Zhao, J. Dobson, L. Moreau, I. Foster, and M. Wilde, "A notation and system for expressing and executing cleanly typed workflows on messy scientific data," in *SIGMOD*, 2005.

[18] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," *technology*, vol. 28, p. 32, 2014.

[19] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2013, pp. 233–240.

[20] C. Zheng and D. Thain, "Integrating containers into workflows: A case study using makeflow, work queue, and docker," in *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*. ACM, 2015, pp. 31–38.

[21] J. P. Walters, V. Chaudhary, M. Cha, S. Guercio Jr, and S. Gallo, "A comparison of virtualization technologies for hpc," in *22nd International Conference on Advanced Information Networking and Applications (aina 2008)*. IEEE, 2008, pp. 861–868.

[22] W. Gerlach, W. Tang, K. Keegan, T. Harrison, A. Wilke, J. Bischof, M. D'Souza, S. Devoid, D. Murphy-Olson, N. Desai *et al.*, "Skyport: container-based execution environment management for multi-cloud scientific workflows," in *Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds*. IEEE Press, 2014, pp. 25–32.

[23] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 18.

[24] G. Staples, "Torque resource manager," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006, p. 8.

[25] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[26] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX Annual Technical Conference*, vol. 8, 2010, p. 9.