Flexible Container-Based Computing Platform on Cloud for Scientific Workflows

Kai Liu
SOKENDAI (The Graduate University for Advanced Studies)
Kanagawa, Japan
liukai@nii.ac.jp

Kento Aida
National Institute of Informatics
Tokyo, Japan
aida@nii.ac.jp

Shigetoshi Yokoyama
National Institute of Informatics
Tokyo, Japan
yoko@nii.ac.jp

Yoshinobu Masatani
National Institute of Informatics
Tokyo, Japan
ym@nii.ac.jp

Abstract—**Cloud computing is expected to be a promising solution for scientific computing. In this paper, we propose a flexible container-based computing platform to run scientific workflows on cloud. We integrate Galaxy, a popular biology workflow system, with four famous container cluster systems. Preliminary evaluation shows that container cluster systems introduce negligible performance overhead for data intensive scientific workflows, meanwhile, they are able to solve tool installation problem, guarantee reproducibility and improve resource utilization. Moreover, we implement four ways of using Docker, the most popular container tool, for our platform. Docker in Docker and Sibling Docker, which run everything within containers, both help scientists easily deploy our platform on any clouds in a few minutes.**

I. INTRODUCTION

With the development of new technologies, such as next generation sequencing (NGS) for genome research, the scientific data amount is growing rapidly. Cloud platforms are expected to be promising solutions for conducting scientific analysis, as they are able to provide powerful, elastic and economical storage and computation ability [1]. Thanks to the flexibility of cloud platforms, scientists are able to build their own data analysis platform on demand. Moreover, the pay-as-you-go pricing model of cloud platforms is very attractive for scientists with limited research funding. Nowadays, researchers from various scientific domains are interested in utilizing cloud platforms like Amazon EC2 [2], for running scientific workflows [3].

Scientific workflow is a data processing pipeline using multiple scientific tools. These tools developed by different programming languages rely on plenty of libraries, binaries and configuration files, causing serious tool installation problem. And the situation will be more disappointing when the tool is lack of document or multiple tools require conflicting dependencies. In addition, it is quite challenging to reproduce the whole tool execution environment if we take OS, library and tool version into consideration. Some researches [4] [5] [6] have found that the container technology, such as Docker [7], was an elegant solution for the dependency hell problem of scientific tools. By packaging all dependencies into Docker images, the tool installation procedure will be less painful. At the same time, it guarantees great reproducibility.

A lot of scientific workflow systems, such as Galaxy [8] and Nextflow [9], are proposed to help scientists to define, submit, monitor and manage scientific workflows. Thanks to these systems, scientists are able to focus on their research issues without worrying about the details of workflow execution mechanisms. However, the tool installation problem is even worse for scientific workflow systems because scientists need to run different workflows which include plenty of scientific tools. Galaxy has already supported Docker to solve this problem. However, it only runs containers on a single computing node, which obviously limits the computing ability. In order to take full advantage of available computing resources, we should be able to run containerized scientific tools in a cluster of multiple nodes in a scalable manner.

A container cluster system, such as Kubernetes [10], helps users to run containers on a cluster of multiple nodes. However, most of existing scientific workflow systems do not work with container cluster systems. A few workflow systems working with container cluster system are dependent on their own proprietary execution engines [4] [5], while others run tasks on traditional cluster systems which do not support the container technology [8] [9].

In this paper, we propose to utilize a container cluster system as the execution engine for a scientific workflow system, and it brings lots of benefits. First, it perfectly solves the scientific tool installation problem. We package scientific tools into Docker images, then we are able to run these tools on any Docker enabled hosts within containers after downloading corresponding images. Second, it guarantees a high level of scientific reproducibility because Docker images ensure exactly the same tool execution environment. Third, container-based method improves resource utilization. Containerized tools can run on any hosts, while without containers, a task will only be scheduled on certain nodes on which the specific tools have been installed. In addition, the container cluster system can be used to run other tasks like MapReduce or web application at the same time, which will greatly improve the overall resource utilization of the whole cluster.

In order to study the feasibility of utilizing a container cluster system as the execution engine for a scientific workflow system, we developed the prototype platform, which integrated Galaxy [8], a popular workflow system in the biology com-

978-1-5090-3951-7/16 $31.00 © 2016 IEEE
DOI 10.1109/ICCCRI.2016.17

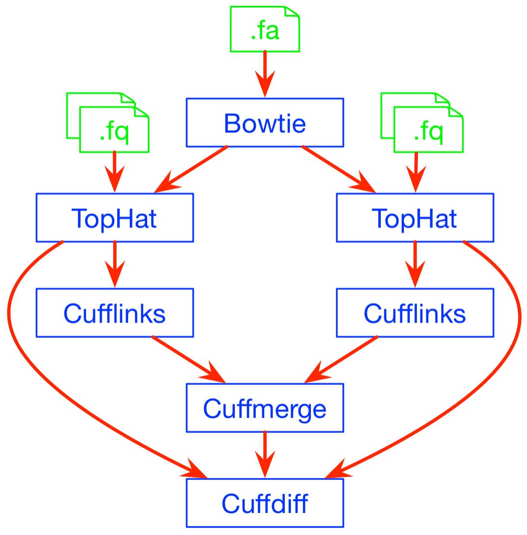

Fig. 1. RNA-Seq Analysis Workflow

munity, with four container cluster systems—Docker Swarm [11], Kubernetes [10] Mesos/Aurora [12] [13] and HTCondor [14]. Our preliminary evaluation results show that container cluster systems introduce negligible performance overhead for data intensive workflows, making it a promising execution engine for scientific workflows considering its great advantages mentioned above.

We compared four different implementations of using Docker for our platform, which are Docker in Docker, Sibling Docker, Tool in Docker and Without Docker. Both Docker in Docker and Sibling Docker methods run everything within containers and they help users easily deploy our platform on any clouds in a few minutes. We find that Docker in Docker and Sibling Docker introduce negligible performance overhead and they perform almost the same. Docker in Docker, which encapsulates an additional Docker within cluster slave container, has better portability than Sibling Docker. For Sibling Docker, it simplifies container management such as logging, monitoring and debugging since it only introduces a single layer of Docker virtualization. Therefore, a tradeoff must be made between better portability and easier management.

The rest of this paper is organized as follows. Section II provides background and related work of this paper. We introduce the architecture of the container-based scientific computing platform in section III. Section IV shows the preliminary performance evaluation of our platform. Finally, we conclude the work of this paper in section V.

II. Background and Related Work

A. *Scientific Workflow System*

Scientific workflow is a data processing pipeline using multiple scientific tools and it is usually represented as a directed acyclic graph (DAG), where the nodes denote data processing tasks and the edges represent data flow. Figure 1 shows an example of workflow used in genome sequencing, RNA-Seq, including scientific tools, TopHat and Cufflinks [15]. Scientific tools composing the scientific workflow are developed using different programming languages [16], and they rely on plenty of libraries, binaries and configuration files. Therefore, installing and configuring a scientific tool is nontrivial for scientists since they are not computer experts.

In various scientific communities like biology and astronomy, there are plenty of scientific workflow systems, which aim to help scientists to define, submit, monitor and manage scientific workflows. For example, Galaxy [8] is a popular web-based workflow system for genomic research. AWE/Shock [17], which consists of an object storage system and a proprietary execution engine, is also a workflow system focusing on running bioinformatics pipelines. Using Nextflow [9] programming model, scientists are able to easily create parallel scientific pipelines. For Pegasus [18], it has been used for more than a decade by scientists from multiple domains.

Scientific workflow systems usually provide a graph editor or a programming script for users to define and submit their workflows. As for execution of workflows, most workflow systems are able to submit tasks to conventional cluster resource management systems. For instance, HTCondor [14] can be used as the execution engine for Galaxy, Nextflow and Pegasus. Other workflow systems like AWE/Shock come with their own proprietary execution engines.

B. *Container Cluster System*

Compared to virtual machine technology, container technology is a more lightweight virtualization technology which is also able to isolate applications. Multiple containers running on the same host share the same kernel with the host while coexisting virtual machines run their independent kernel. Therefore, the container technology is able to provide near native performance [19], higher density and faster start/stop time. Modern linux kernel features, such as Namespaces [20] and Cgroups [21], are important building blocks for container technology. Namespaces are the key to isolate containers with the host and other containers, while Cgroups can limit the usage of host resources like CPU and memory for containers. Although container technology has existed for over a decade, only recently have it gained increasing attraction with the popularity of a container tool called Docker.

Docker [7], the most popular container tool, transcends container pioneers like Linux-VServer [22] and LXC [23]. In fact, Docker has become the de facto standard container tool widely utilized in the industry. By using Docker, we can easily create, stop, restart and delete containers. This great user friendliness contributes a lot for the success of Docker. The ability to package an application and its dependencies into a lightweight Docker image is another advanced feature of Docker. We can download Docker images and run the application in any Docker enabled hosts without worrying about laborious installation steps. This helps us a lot to solve the tool installation and cloud deployment problem. Docker image can be specified using Dockerfile, which is a text file with all commands for building the image. Therefore, Dockerfile can be easily maintained through version control tool like git.

Using Docker, we can easily run containers on a single node, but if we want to run containers on a cluster of multiple nodes, we need to use container cluster systems, such as Docker Swarm [11], Kubernetes [10] and Mesos/Aurora [12] [13]. Docker Swarm is a native cluster system developed by Docker company and Kubernetes is an open source version of Google's internal container cluster system called Borg [24]. For Mesos/Aurora, Mesos is a cluster resource manager while Aurora is a job scheduler. These container cluster systems are quite different with each other, but they all have a master/slave architecture. The master node schedules containers to slave nodes and slave nodes are in charge of executing containers. Traditional cluster systems like HTCondor [14] also support the container technology now, therefore, we can categorize it into the container cluster system. Since we can run any applications within containers and isolate them with each other, container cluster systems can be used as the execution engine for any kind of workload. For example, Google runs everything including web search, Gmail and Google Docs on Borg for almost a decade. Therefore, container cluster systems can also be used for running scientific workflows.

C. Related Work

In order to improve the scalability, SAASFEE [16] utilizes Hadoop YARN [25] as the execution engine for Galaxy [8] and Pegasus [18]. On one hand, the idea of utilizing general purpose cluster system to run scientific workflow is similar to our proposal. On the other hand, the ability of supporting multiple workflow systems is quite impressive, which is also one of the ultimate goal of our platform. Although Hadoop YARN also supports Docker now, SAASFEE does not introduce container technology to provide tool execution environment, making the system less flexible.

Skyport [4] is an extension of AWE/Shock [17] workflow system and it utilizes Docker to solve the tool installation problem. Skyport is the earliest one to use container technology for a scientific workflow system, however, it only enable Docker support for its own proprietary execution engine, which means other workflow systems hardly benefit from it. In addition, general container cluster systems like Kubernetes [10], which are widely used and actively developed in the growing container community, can at least promise higher reliability with faster bug fixing than proprietary ones.

Zheng [5] examines four different methods of integrating Docker with Makeflow [26] workflow system and Work Queue [27] execution engine. Therefore, their work is quite similar to ours. However, like Skyport, they only add container support for Work Queue, which is their proprietary workflow execution engine. In addition, they do not consider the deployment problem of workflow system and execution engine, thus their methods of using Docker do not include Docker in Docker and Sibling Docker.

The authors are developing Virtual Cloud Provider(VCP) [6] to help users to easily configure customized computing environments on clouds. In VCP, workflows submitted from Galaxy are executed trough Mesos/Aurora [12] [13] framework. VCP is the first one to utilize general container cluster system as the execution engine for scientific workflow system. However, the current VCP only works with Mesos/Aurora framework, while there are a lot of other options, e.g. Kubernetes. Our goal is to extend the current VCP implementation for supporting multiple container cluster systems. VCP utilizes nested Docker mechanism to provide an overlay cloud, while this paper proposes a new method called Sibling Docker which simplifies the architecture. In addition, VCP runs Network File System (NFS) on hosts, but our platform runs NFS within containers, helping us truly containerize everything.

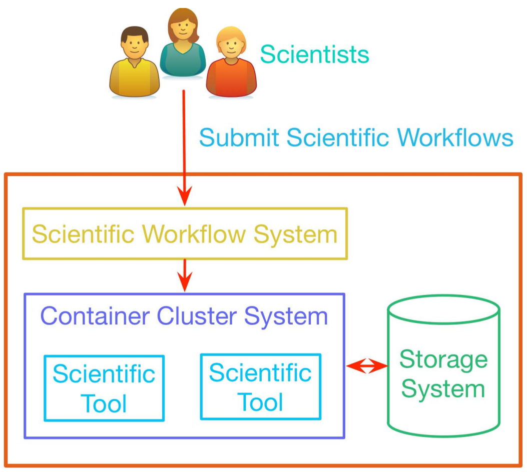

Fig. 2. Scientific Computing Platform

III. PLATFORM ARCHITECTURE

A. Scientific Computing Platform

The scientific computing platform discussed in this paper consists of a scientific workflow system, a container cluster system, a storage system and various scientific tools as illustrated in Figure 2. Scientists use the scientific workflow system to submit, monitor and manage their workflows. Scientific tools composing the workflow will be executed within containers and the container cluster system is in charge of scheduling and executing these containers. All input and output data are stored in the storage system.

The proposed architecture utilizes a container cluster system as the execution engine for scientific workflows. Existing system like Skyport tightly couples the workflow system and the execution engine, while our platform decouples the workflow system and the execution engines. Decoupling the workflow system and the execution engines greatly improves the flexibility of the scientific computing platform; that is, users are able to run their workflows through the familiar workflow system with suitable container cluster systems.

In theory, our proposed platform is capable to run almost any kind of scientific workflows except for tools with specific hardware requirements(e.g., high RAM). Moreover, unlike

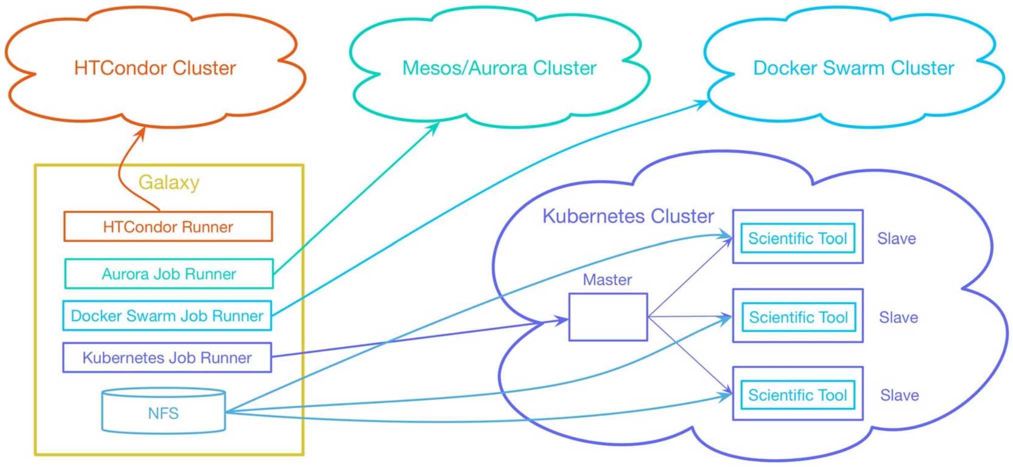

Fig. 3. Scientific Computing Platform for Galaxy Workflow

other workflow platforms, no scientific tools or their dependencies need to be preinstalled on our platform. Scientific tools run within containers and they can be scheduled on any nodes of the computing cluster. Without container, a task can run on specific nodes on which the required tools have been installed.

Deploying or reproducing a scientific computing platform on clouds is quite challenging due to the system complexity, especially for scientists who are lack of computer skills. To solve the cloud deployment problem, the proposed architecture runs the whole platform within Docker containers. The Container based method enables users to easily deploy the scientific computing platform on clouds in few minutes. Thanks to the great portability of container technology, the platform can be created on any clouds without the vendor lock-in problem. Scientists can also build the scientific computing platform on their local host, providing a consistent environment for testing, which will benefit a lot for their research.

Actually, there are many scientific workflow systems which are popular in different scientific communities. And we also have a lot of options for container cluster systems. For the storage system, shared file system, object storage system or other storage systems can also be candidate. Ultimately, we plan to provide support for various choices, then scientists will have full flexibility to select and combine their preferred components to build a customized scientific computing platform on clouds. Selection of the suitable container cluster system may significantly affect the performance of user applications. In this paper, we investigate the application performance on different container cluster systems.

B. *Scientific Computing Platform for Galaxy Workflow*

As a case study, we select Galaxy, a widely used biology workflow system, to build a prototype platform as shown in Figure 3. Galaxy provides a web interface for scientists to define their workflows. Four widely used container cluster systems act as the execution engine. All scientific tools run within Docker containers. Currently, we use NFS as the storage system, and we plan to try other storage systems to improve the scalability as our future work.

To study the feasibility of utilizing the container cluster system as the execution engine for the scientific workflow system, we integrate Galaxy with three general purpose container cluster systems, Docker Swarm, Kubernetes and Mesos/Aurora. In addition, we also support HTCondor, a conventional cluster system widely used in the HPC community. HTCondor supports Docker recently.

We implemented four job runners as the interfaces to container cluster systems in Galaxy as illustrated in Figure 3. The job runners are in charge of submitting, monitoring and deleting jobs on their corresponding container cluster systems. The ability to submit jobs to different container cluster systems improves the flexibility of our platform because scientists will be able to make decisions according to their preference, e.g., performance or familiarity. In addition, the comparison of these container cluster systems is also valuable for promoting the usage of the container cluster system for running scientific workflows. Note that different container cluster systems provide different user interfaces for creating and managing containers. Although we have implemented interfaces for four container cluster systems in Galaxy, it will cause problems when we extends workflow systems like Nextflow. Developing unified interfaces for multiple container cluster systems is one of solutions for this problem, and we leave it for future work.

C. *Implementation Settings for Docker*

We implemented two options to run everything within containers, Docker in Docker and Sibling Docker. We also implemented two options to run cluster slaves without containers, Tool in Docker and Without Docker, to investigate flexibility and overhead of containerizing everything.

1) Docker in Docker: For Docker in Docker, the cluster slave container runs directly on the host while scientific tool containers run inside the slave container. As shown in Figure 4, the Docker daemon running on the host starts the cluster slave

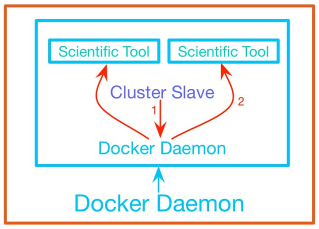

Fig. 4. Docker in Docker

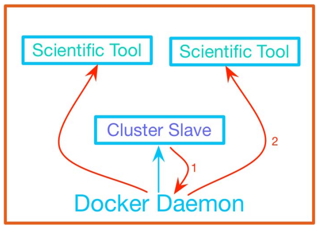

Fig. 5. Sibling Docker

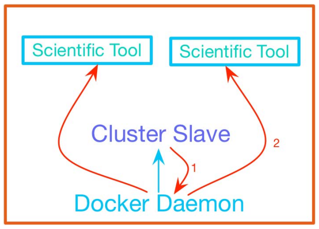

Fig. 6. Tool in Docker

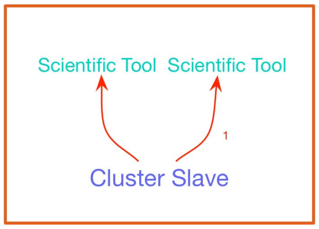

Fig. 7. Without Docker

container while the Docker daemon running within the slave container is responsible for managing scientific tool containers. The cluster slave requests the inner Docker daemon to start, monitor and delete scientific tool containers. We do not limit the resource usage, e.g., CPU and memory, for the slave container, which means the slave container is able to consume host resources as much as they need. Docker in Docker mechanism perfectly solves the deployment problem for both cluster slave and scientific tools by introducing two layer of Dockerization. Docker in Docker is easier to implement than Sibling Docker since it is natively supported by Docker, especially Cgroups is automatically mounted into container after Docker 1.8.0. In fact, Docker in Docker method is initially proposed in the Virtual Cloud Provider(VCP) project [6] and it is an intuitive approach for our platform to run everything within containers.

2) Sibling Docker: Sibling Docker, just like the name suggests, cluster slave container and scientific tool containers run side by side on the host, in other word, both slave container and scientific tool containers run on the same host. In Figure 5, there is only one Docker daemon run on the host and it takes charge of managing both the slave container and scientific tool containers. By mapping the Docker Remote API endpoint, which is a Unix socket or a TCP socket, into the slave container using Docker volume or environment variable, all container management requests from the cluster slave are sent to the outer docker daemon. Therefore, scientific tool containers are started by the outer Docker daemon and the scientific tool containers directly run on the host. Like Docker in Docker, we set no resource limitation for the slave container, thus all resources of the host are “visible” for the cluster slave and scientific tool containers are able to consume all host resources. Sibling Docker works as same as Docker in Docker from the users perspective because they both run cluster slave and scientific tools within containers. In addition, sibling Docker only introduces one layer of the Docker virtualization, thus it simplifies container management, such as logging, monitoring and debugging. However, for Sibling Docker, Docker client within cluster slave container need to communicate with Docker daemon on the host using the same Docker API. Therefore, Sibling Docker can only be deployed on hosts which has installed specific version of Docker, making Sibling Docker method is less portable than Docker in Docker method.

3) Tool in Docker: We name it Tool in Docker when only scientific tools run within Docker containers while the cluster slave runs directly on the host. From Figure 6, we can see that Docker daemon on the host will instantiate tool containers in response to the requests from the cluster slave. For Tool in Docker method, we do not have to worry about the tool installation problem because all containers run within Docker containers based on corresponding Docker images, in which scientific tools and their dependencies have already been packaged. This greatly improves the platform flexibility because we can run almost any scientific tools on any nodes without installing any tools on these nodes. However, we have to manually install and configure a container cluster system, which is usually quite challenging. According to our experiment experiences, installing container cluster systems, including Docker Swarm, Kubernetes, Mesos/Aurora and HTCondor, are difficult due to their plenty of dependencies, multiple system components, complex configuration files and inaccurate documents. In addition, this method causes a lot of trouble when we want to deploy or reproduce the platform on clouds. In summary, running scientific tools within containers benefits a lot but just containerizing scientific tools is not enough for solving issues discussed in this paper.

4) Without Docker: For comparison, we also implemented another architecture called Without Docker, which is depicted in Figure 7. For this method, both the cluster slave and scientific tools directly run on the host, and the cluster slave executes scientific tools without containers. Docker Swarm and Kubernetes are container native and they can only run applications within containers, so we just implemented Without Docker for Mesos/Aurora and HTCondor. Without the help of Docker, we need to install both the cluster system and scientific tools manually. Note that the platform should be able to run various scientific tools, making the tool installation problem even worse because some tools will require conflicting dependencies and sometimes we need to test different versions for the same tool. Although we just test a few scientific tools in this paper, we also experienced some tool installation problems such as missing dependencies, wrong dependencies and lack of permissions. Therefore, Docker in Docker and Sibling Docker methods, which run both cluster slave and scientific tools within containers, are much more flexible than Without Docker method.

IV. EVALUATION

We conducted the preliminary performance evaluation on the prototype platform. We compared performance of four

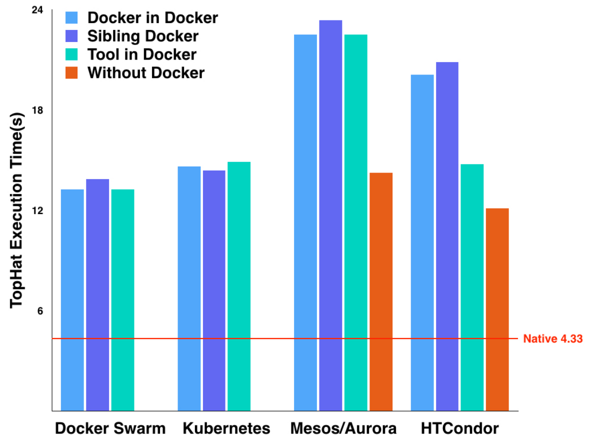

Fig. 8. Execution Time of TopHat with Small Input

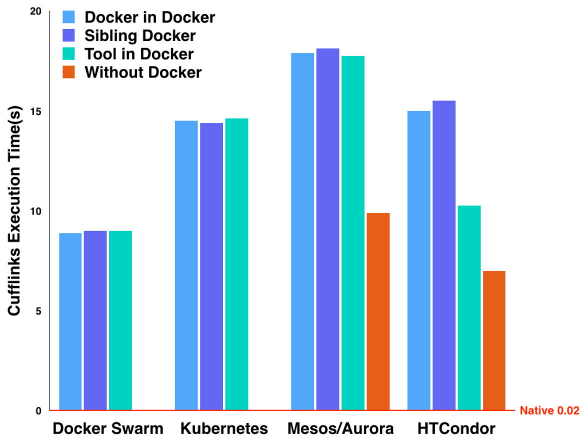

Fig. 9. Execution Time of Cufflinks with Small Input

container cluster systems and four different implementation settings for Docker discussed in Section 3.3.

A. Testbed Setup

For four container cluster systems, Mesos/Aurora and HT-Condor have four implementation settings for Docker, while Docker Swarm and Kubernetes do not support Without Docker method. In total, there are 14 cluster settings.

We deployed our platform on Amazon EC2, the most popular IaaS cloud platform, to run all tests. For each cluster setting, we used five m1.xlarge instances with four 2.0 GHz Intel Xeon E5-2650 processors, 15GB of RAM and 200GB of SSD. We ran Galaxy on one node and used other four nodes for container cluster system with one master node and three slave nodes.

For all 14 cluster settings, Galaxy run within containers, while just Docker in Docker and Sibling Docker methods run container cluster system within containers. Galaxy container, as well as the cluster master and slave containers run with "–net=host" option to share the same network stack with the host node, which means they have the same IP addresses of the hosts and share port numbers with the hosts. This setting enables these containers to communicate each other although they run on different nodes.

TABLE I
SOLFTWARE SETTING

Host OS	Ubuntu 14.04 (64bit)
Kernel	3.19.0-49-generic
Docker	1.10.1
Container OS	Ubuntu 14.04 (64bit)
Galaxy	15.10
Docker Swarm	1.1.0
Kubernetes	1.0.7
Mesos	0.26.0
Aurora	0.11.0
HTCondor	8.4.4
TopHat	2.0.9
Cufflinks	2.2.1

Table I shows the software setting used in our platform. Except for host OS, Kernel and Docker, all other software together with their dependencies are packaged into Docker images. We also packaged host OS together with Docker into virtual machine image, called Amazon Machine Image (AMI) for Amazon EC2, to launch instances. Therefore, our platform guarantees greate reproducibility. Although we can encapsulate the whole platform using virtual machine image to achieve the same goal, container technology is much more portable. Based on the same Docker image, Docker container can run almost anywhere, including a local machine, a private datacenter or a public cloud. Therefore, our platform can be easily deployed on any other clouds. While for virtual machine image, it is highly coupled with specific cloud provider.

Given the TCP endpoint(IP and port) of the Docker daemon, we can start containers on each node remotely. Therefore, for Docker in Docker and Sibling Docker, the deployment can be finished in a few minutes just by running a shell script to start galaxy, cluster master and cluster slave containers. In contrast, it took considerable time for us to deploy clusters with Tool in Docker and Without Docker methods because it requires laborious and error prone tasks.

We selected a simple RNA-Seq workflow, which is used for analyzing the sequence of RNA in a cell, to evaluate our platform. We downloaded all required Docker images before running tests and we allocated the same amount of CPU and Memory for each container for all cluster settings.

B. TopHat and Cufflinks with Small Input

We evaluated the execution time of TopHat and Cufflinks with small datasets in order to investigate the performance overhead on our platform. We used official test datasets, which are less than 50KB, as the input data. We ran each tool for 10 times to measure the average execution time. In addition, we ran two tools directly on the EC2 node to measure the native execution time. Figure 8 and Figure 9 show the average tool execution time of TopHat and Cufflinks grouped by four container cluster systems. Four implementation settings of

Docker are distinguished by different colors. And we draw a horizontal line to represent the native execution time.

Unsurprisingly, 14 cluster settings introduce 7-19 seconds of overhead compared to the native execution time. The tool execution time is calculated from task submission to its completion, therefore, the overhead of executing tools on container cluster systems should include data transfer time, container scheduling time, container initiation time as well as the overhead of containerization. Although the results indicate relatively large overhead with small input data size, the overhead is negligible in the real world setting with much larger data size. We will see results in the next section.

Among four container cluster systems, Docker Swarm shows the best performance. In fact, the architecture of Docker Swarm is much simpler than other three competitors, which should benefit a lot for its high performance. In a Docker Swarm cluster, Swarm agents just register slave nodes, Swarm manager is responsible for scheduling containers to slaves nodes. After selecting suitable slave nodes, Swarm manager start containers remotely using Docker Remote API. While for Kubernetes, HTCondor and Mesos/Aurora, their cluster slaves is in charge of starting containers, and both their cluster master and cluster slaves consist of multiple components, which will probably introduce additional communication overhead. Kubernetes introduces an additional layer of abstraction above container called pod. For Mesos/Aurora, it has a two layer architecture, Mesos is a cluster resource manager while Aurora is a job scheduler.

Kubernetes performs worse than Docker Swarm, especially for Cufflinks. Pod, which is a group of containers, is the smallest deployable unit in Kubernetes. Thus for Kubernetes, we ran a scientific tool within a pod other than a container. Kubernetes runs an additional container called "pause" to handle networking for each pod. The "pause" container only consumes a small amount of resources, but it likely affects the overall performance of short running tasks. With default network setting, Kubernetes will create an independent network namespace for each pod, therefore, a pod will has its own IP address. As shown in Figure 10, the performance improved a lot when we ran the cufflinks pod with "host" mode, which eliminates network overhead by sharing the same network namespace with the host node. The default network setting of Kubernetes is designed for web application to enable pod to pod communication, while this feature is unnecessary for scientific tools like Cufflinks. Therefore, we adopt "host" mode for Kubernetes in this evaluation.

Among four implementation settings of Docker, Without Docker of Mesos/Aurora and HTCondor outperforms other methods since it introduces no Dockerization overhead. However, the overhead is acceptable since Docker in Docker and Sibling Docker methods bring dramatic flexibility.

For all container cluster systems, Docker in Docker and Sibling Docker present nearly the same performance, which indicates Sibling Docker is a promising alternative method for our platform to run everything within Docker containers. We can observe the same results in the next section.

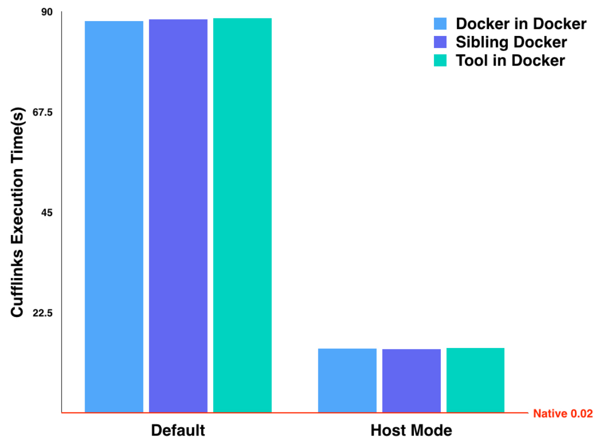

Fig. 10. Execution Time of Cufflinks on Kubernetes

C. *TopHat and Cufflinks with Large Input*

In order to evaluate the performance of our platform for long running tasks that we see in the real world, we ran TopHat and Cufflinks with different size of large input datasets. We got different size of read files by cutting the original datasets as the input for TopHat. Different size of TopHat output datasets were used as the input datasets for Cufflinks. In this evaluation, we focus on the performance of Docker in Docker and Sibling Docker methods which fulfill our requirements of containerizing everything. For comparison, we also ran two tools directly on the EC2 node as the native execution time. For each input dataset, we ran the tool for 10 times to get the average execution time. Figure 11 and Figure 12 show the average tool execution time of TopHat and Cufflinks grouped by four container cluster systems. Results of Docker in Docker, Sibling Docker and Native settings are distinguished by different colors.

The evaluation results are concluded as follows. First, compared to the native execution time, the overhead observed on four container cluster systems is negligible. Therefore, container cluster systems are promising execution engine for running scientific workflows considering their great flexibility. Second, the performances of running tools on four container cluster systems are almost the same. Thus for long running scientific workflows, users can ignore the performance differences when choosing container cluster systems. Third, Docker in Docker and Sibling Docker show almost the same performance, making Sibling Docker a potential alternative for our platform as it makes the container management easier. However, we have to slightly sacrifice portability if we choose Sibling Docker as we discussed before.

Current evaluation only ran a single container each time, which shows the basic performance of our platform, however, the performance may different when we run a large number of containers simultaneously. In the next step, we will conduct detailed performance evaluation by running concurrent containers.

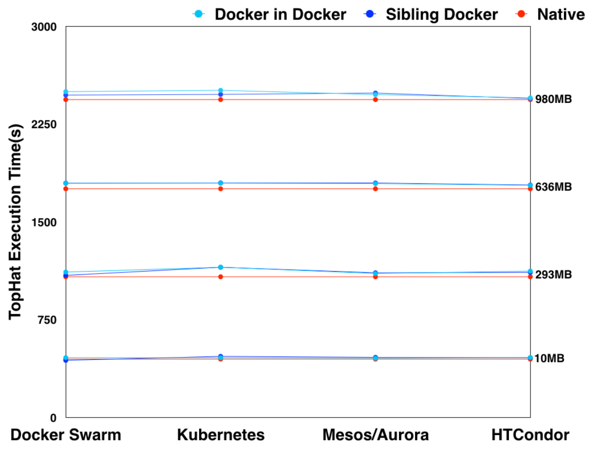

Fig. 11. Execution Time of TopHat with Large Input

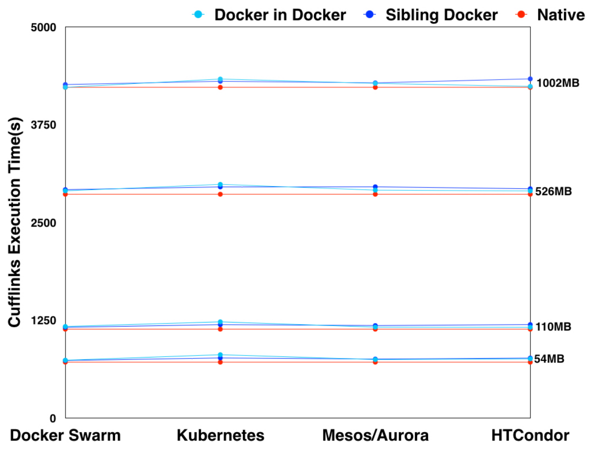

Fig. 12. Execution Time of Cufflinks with Large Input

V. CONCLUSION

Container technology, which gains increasing popularity in recent years, is an ideal technology for solving the dependency hell problem of scientific computing platform. And as we have shown, container cluster systems like Docker Swarm, are promising execution engines for scientific workflow systems for their native performance and dramatic flexibility. Moreover, by utilizing container technology, we are able to easily deploy our scientific computing platform on any cloud in a few minutes. According to our experimental results, Sibling Docker and Docker in Docker perform almost the same. Thanks to its full encapsulation, Docker in Docker presents better portability than Sibling Docker. While for Sibling Docker, which only introduces a single layer of Docker virtualization, is able to simplify container management. Therefore, a tradeoff must be made between better portability and easier management.

ACKNOWLEDGMENT

This work was supported in part by CREST, JST.

REFERENCES

[1] V. Marx, "Biology: The big challenges of big data," *Nature*, vol. 498, no. 7453, pp. 255–260, 2013.
[2] Amazon EC2. [Online]. Available: https://aws.amazon.com/ec2
[3] E. Deelman, K. Vahi, M. Rynge *et al.*, "Pegasus in the Cloud: Science Automation through Workflow Technologies," *IEEE Internet Computing*, vol. 20, no. 1, pp. 70–76, 2016.
[4] W. Gerlach, W. Tang, K. Keegan *et al.*, "Skyport: Container-based Execution Environment Management for Multi-cloud Scientific Workflows," in *5th International Workshop on Data-Intensive Computing in the Clouds*, Nov. 2014.
[5] 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*, Jun. 2015.
[6] S. Yokoyama, Y. Masatani, T. Ohta *et al.*, "Reproducible Scientific Computing Environment with Overlay Cloud Architecture," in *9th IEEE International Conference on Cloud Computing*, Jun. 2016.
[7] Docker. [Online]. Available: https://www.docker.com
[8] J. Goecks, A. Nekrutenko, J. Taylor *et al.*, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome Biology*, vol. 11, no. 8, pp. 1–13, 2010.
[9] Nextflow. [Online]. Available: http://www.nextflow.io
[10] Kubernetes. [Online]. Available: http://kubernetes.io
[11] Docker Swarm. [Online]. Available: https://docs.docker.com/swarm
[12] B. Hindman, A. Konwinski, M. Zaharia *et al.*, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, Mar. 2011.
[13] Aurora. [Online]. Available: http://aurora.apache.org
[14] 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.
[15] C. Trapnell, A. Roberts, L. Goff *et al.*, "Differential gene and transcript expression analysis of RNA-seq experiments with TopHat and Cufflinks," *Nature protocols*, vol. 7, no. 3, pp. 562–578, 2012.
[16] M. Bux, J. Brandt, C. Lipka *et al.*, "SAASFEE: Scalable Scientific Workflow Execution Engine," *Proceedings of the 41st International Conference on Very Large Data Bases*, vol. 8, no. 12, pp. 1892–1895, 2015.
[17] W. Tang, J. Wilkening, N. Desai *et al.*, "A Scalable Data Analysis Platform for Metagenomics," in *IEEE International Conference on Big Data*, Oct. 2013.
[18] E. Deelman, K. Vahi, G. Juve *et al.*, "Pegasus, a Workflow Management System for Science Automation," *Future Generation Computer Systems*, vol. 46, no. C, pp. 17 – 35, 2015.
[19] W. Felter, A. Ferreira, R. Rajamony *et al.*, "An Updated Performance Comparison of Virtual Machines and Linux Containers," in *IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2015.
[20] E. W. Biederman and L. Networx, "Multiple Instances of the Global Linux Namespaces," in *Proceedings of Linux Symposium*, Jul. 2006.
[21] P. B. Menage, "Adding Generic Process Containers to the Linux Kernel," in *Proceedings of Linux Symposium*, Jun. 2007.
[22] S. Soltesz, H. Pötzl, M. E. Fiuczynski *et al.*, "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors," in *ACM SIGOPS Operating Systems Review*, Mar. 2007.
[23] LXC. [Online]. Available: https://linuxcontainers.org
[24] A. Verma, L. Pedrosa, M. Korupolu *et al.*, "Large-scale Cluster Management at Google with Borg," in *Proceedings of the Tenth European Conference on Computer Systems*, Apr. 2015.
[25] V. K. Vavilapalli, A. C. Murthy, and C. Douglas, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, Oct. 2013.
[26] M. Albrecht, P. Donnelly, P. Bui *et al.*, "Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids," in *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, May. 2012.
[27] P. Bui, D. Rajan, B. Abdul-wahid *et al.*, "Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications," in *Workshop on Python for High Performance and Scientific Computing*, Nov. 2011.