Efficient Integration of Containers into Scientific Workflows

Kyle M.D. Sweeney
Department of Computer Science and Engineering
Notre Dame, IN
ksweene3@nd.edu

Douglas Thain
Department of Computer Science and Engineering
Notre Dame, IN
dthain@nd.edu

ABSTRACT

Containers offer a powerful way to create portability for scientific applications. However yet incorporating them into workflows requires careful consideration, as straightforward approaches can increase network usage and runtime. We identified three issues in this process: container composition, containerizing workers or jobs, and container image translation. To tackle composition, we define data into three types: OS data, Read-Only, and Working data, and define dynamic and static composition. Using the static composition (creating a single container for each job) leads to massive waste in sending duplicate data over the network. Dynamic composition (sending the data types separately) enables caching on worker nodes. To answer running workers or jobs inside a container, we looked at the costs of running inside of a container. Finally, when using different types of container technologies simultaneously, we found it's better to convert to the target image types before sending the container images, instead of repeating the same conversion at the job nodes, leading to more wasted time.[1]

ACM Reference format:
Kyle M.D. Sweeney and Douglas Thain. 2018. Efficient Integration of Containers into Scientific Workflows. In *Proceedings of 9th Workshop on Scientific Cloud Computing, Tempe, AZ, USA, June 11, 2018 (ScienceCloud'18),* 6 pages.
https://doi.org/10.1145/3217880.3217887

1 INTRODUCTION

Containers, such as Docker [9], Singularity [7], and Charliecloud [10], offer an efficient alternative to virtual machines in High Performance Computing (HPC) centers as a method of bringing one's own software dependencies instead of relying on the HPC centers' software. At first, integration of containers into a workflow might seem straight forward. The obvious solution would be to treat a container as a bucket: simply deposit all data, libraries, executables, etc, into a single image for the whole workflow. While easy, this method is inefficient as a lot of irrelevant data for a given job would be transferred to a node for executing the job. Another method would be to simply create a new image for each job which perfectly encapsulates all of the data necessary for that given job, but this too creates inefficiencies if there is data overlap between jobs, leading to sending the same data everywhere many times. To solve this, we need to look at several key aspects of how one can properly integrate containers into a workflow.

We examined three key questions:

- How do we compose data and images into a container?
- Do we run workers or jobs inside of the container?
- When using more than one container technology, when do we transform our images?

To execute a job, a workflow composes, or brings together, the necessary inputs for execution. Thus, the first question is how to compose different kinds of data together when starting a container, and when to compose. We identify three kinds of image types: 1) the container's OS data holding the actual OS for the image, the executables, and libraries and other software necessary for the job, 2) Read-only data which is the job specific scientific data operated on by the job, and 3) the Read-Write data which is generated by the job. If we compose too early, then we send too much replicated data across the network.

Next, we consider the question of running a job inside of a container, or running the worker node accepting jobs inside of a container. If a container image is sent to the worker node for the node itself to run inside of, and if jobs need different images, this limits the ability of the scheduler to assign jobs in the most efficient manner. Similarly, what are the costs of running inside of a container, and can these be amortized over many different jobs?

Finally, a user might desire to run their workflows using resources from different centers, e.g EC2 instances, their local HPC, and a guest allocation at a National HPC center. Each of these resources might have different container technologies on them, and the straightforward approach of creating a custom image for each one creates more hassle for the user. Thus, how do we decide when to convert images from one virtualization type to another? What are the benefits of converting first and then sending jobs, versus converting before running the jobs themselves?

We ran several tests to examine these questions. To test container composition, we ran two workflows: an artificial randomized workflow and a bioinformatics workflow. The artificial workflow took randomized data and passed it around to workers, printing out the data to a file. Both workflows demonstrated that a dynamic approach enables workers to cache the input files across different jobs, leading to less data sent around the network and a faster runtime. To test worker vs job containerization, we examined the startup and teardown costs of a Singularity container, finding that while small, the startup cost can be significant if the runtime of the job is similarly small, thus possibly a benefit to having the worker in a container. We also measured the read overhead of containers by nesting containers, finding that for larger files, the overhead becomes significant, pointing towards having each job in a container if the workflow doesn't need every job to be in a container.

[1]This work was supported in part by National Science Foundation grant OAC-1642409

ScienceCloud'18, June 11, 2018, Tempe, AZ, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5863-7/18/06…$15.00
https://doi.org/10.1145/3217880.3217887

Finally, we measured transforming different types of containers into a common image format, i.e a tar'd file system tree, and how long it took to start containers based off of transformed images. After looking at the top 75 images from Docker and Singularity, we found that conversion of images took on average between 22s to 199s to convert images, and between 5.6s to 67s to then start a container. We then ran a bioinformatics workflow and found that pre-converting the image led to shorter per-job runtime than delaying conversion of images to just before executing a job. Given these results, it was better to apply a dynamic-approach and run each job in a container by itself, and pre-convert their images if using resources with different container technologies.

2 WORKFLOWS AND CONTAINERS

A workflow [12] [8] is a method for breaking up a large scientific problem into smaller chunks which often have dependencies on other jobs, and if their dependencies are fulfilled, can be ran in parallel, thus speeding up scientific work. Containers can be especially useful for workflows as encapsulating the software stack can help with reproducibility as well as portability, two desirable qualities for any scientific work. Ideally, the container should have a minimal impact on the execution time of the workflow, and not alter the actual function of the workflow itself.

The most straightforward approach would be to run the entire workflow in a container: that is, place all data, binaries, libraries, configuration files etc in a single container image and then distribute that image to each worker node that is executing a job inside of the workflow. However, this would lead to lesser performance than native speed, as each worker node would receive not only the data it needed for a single job, but also all of the data for every other single job in the workflow. This is a massive amount of data being passed around, thus increasing the runtime for the job. Additionally, if this image has to be converted from one form to another, since each worker has a different container technology it is supporting, then the entire image, including all of the unnecessary data, has to be copied into some other form, further increasing the runtime of the job and thus entire workflow. We approach this problem step by step in the following sections to fully examine it and offer possible solutions for it.

3 CONTAINER COMPOSITION

When building containers, we first have to consider what kinds of data are composed together when creating our container. We classify container data types into three categories: OS data, Read Only data, and Work data. Similar to how a Harvard cpu architecture separates out instruction memory and working memory since they have different access patterns, the three categories of container data have different kinds of access patterns themselves.

OS data is composed of all the data needed for executing the job. This includes binaries, libraries, and configuration files. Because this data pertains to the execution of programs, it's best to be thought of as read-only data. Additionally, the individual files are typically going to be small, on the order of kilobytes and megabytes, not gigabytes, and stored in very deep file system structures.

Read Only data is the scientific data which the jobs use. This data could be job specific, or universal for all jobs in our workflow, but the key characteristic of it is that it is large and only read. This data is on the orders of gigabytes and larger. Additionally, this data is more likely to be stored in a flat file system structure, not buried deep in directories upon directories.

Work data are the files generated and modified by the jobs themselves. This data is read-write data, and ranges in size from kilobytes to gigabytes on up, depending on the job. It is generally stored flat in the file system tree as well, not buried in directories.

When wanting to work on a job, the first two must be sent to the job before the job can be executed, and a location for the third linked in. Given a workflow, we can imagine the individual jobs, in the workflow as needing to be done inside of a container. The question of composition then becomes tied to the question of image distribution. Here, we describe two different approaches to looking at composing containers: the static and dynamic approaches. To fully understand both methods, it's helpful to imagine the workflow as a directed graph. In the graph there are layers through which jobs and data are passed, and as the workflow proceeds, we move through the different layers.

In the static approach, we first compose our images together as early in the workflow graph as possible, sending out copies of this composed image to every stage that's needed, thus when finally at the necessary stages, there is minimal composition needed to start the container. This approach has the benefit of being simple: if we have a job, simply wrap up everything needed for that job into the same image and then simply send that image to the node which will work on the job, and execute. Additionally, this provides security, as we will not share any data between different jobs: every job has it's own unique container. In the dynamic approach, we delay composing the container until as late as possible, preferably only when the job is at the node and all data needed for a job is at the worker to run. The benefit to this approach is cache-ability. If multiple jobs need the same input data, the worker which runs both jobs only needs to be sent the data once, thus minimizing the amount of data that is spread throughout the network, and increasing the throughput and speed of working the workflow.

To test these approaches, we conducted two tests: an artifical workflow and a BLAST workflow [1]. We wanted to measure how long the different approaches would take, and how much data over the network both transfer. In the artificial workflow, we ran it in four configurations: static and dynamic, both with enabled caching and not. In the dynamic methods, a single image was passed to each worker, and the data transferred separately. In the stacked methods, the images were large enough to store all the random data, assuming a naive approach, and sent out. All cases had an independent run script. As can be seen in 3, caching greatly reduces the amount of data sent, especially seen in the dynamic approach, and thus leading to a shorter runtime. The BLAST workflow did a similar approach, this time images only big enough for each job's data, and with caching enabled. Again, the dynamic approach transferred less data and ran in a shorter runtime, as seen in figure 4.

4 CONTAINERIZE TASKS OR WORKERS?

Another question when discussing how to integrate containers into a workflow is whether or not to place the worker nodes themselves inside a container, or put each job inside of a container. Containers

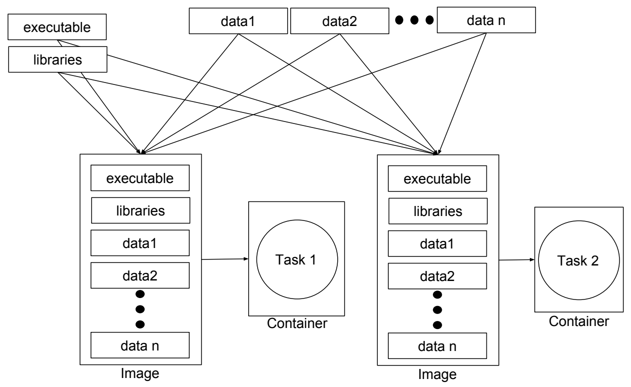

Figure 1: Static Composition

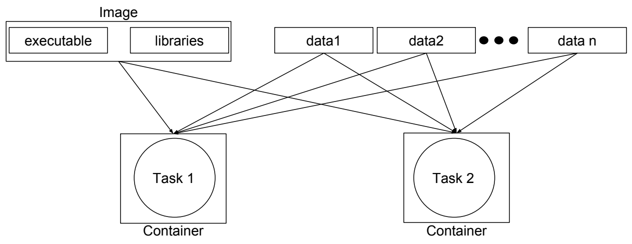

Figure 2: Dynamic Composition

Here we show both the static and dynamic compositions. A static composition involves packaging everything that a job needs to execute into one package, and delivering that package to the job. Included is the executable, the libraries, and all data input necessary. A dynamic composition keeps only one copy of all necessary input files, and passes them as references.

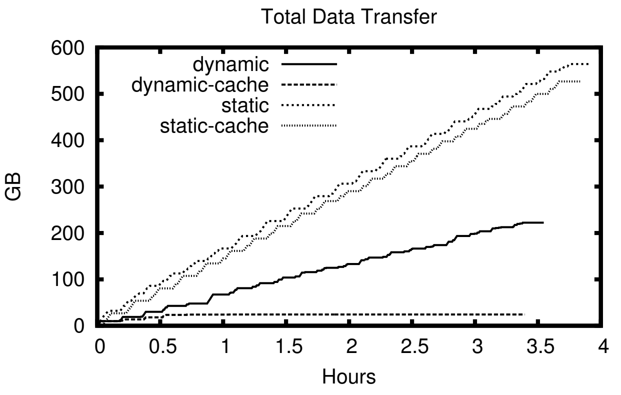

Figure 3: Randomized Workflow

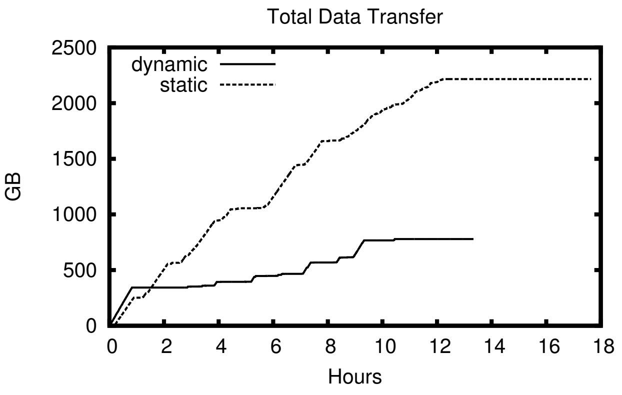

Figure 4: BLAST Workflow

In these two workflows, we are testing dynamic vs static configurations. The vertical shows how many bytes are sent while the bottom represents time ticks created by standardized timestamps from Makeflow [3], our workflow system, for each configuration ran. As we can see, for both tests the dynamic configuration with caching not only sends less data, but also results in shorter runtime than the static configuration.

have some inherent costs associated with them, such as start up and tear down times and read-write overheads.

As discussed by Zheng and Thain [13], two major configurations that one can run a container in are either every job runs in a container, or the worker running the jobs is inside of a container. In the first approach, seen in figure 5, every job runs in its own container, which ensures job isolation between the different jobs being managed by the worker. Thus, jobs have more security, knowing that they can't interfere in the operation of the other jobs. This approach is also helpful when one looks at the read-write overhead associated with containers. Although containers are extremely light-weight in resource usage, as compared to virtual machines, they still suffer some overhead. For example, when testing Singularity, we nested it several times over and attempted to read and write different sizes of files. As can be see in figure 8, for small files, the read performance of Singularity is very close to that of native performance, but as files increase in size, the performance gap between native reading vs even a single layer for Singularity is significant. When the workers themselves are not wrapped inside of a container, this gives more freedom to decide if having a container is truly necessary, especially in cases where the job is simply to split the data up into many smaller files and not do hard computation on them, since those jobs can often be done by a command line tool, such as awk or grep, etc. However, when looking at a per-job level, the startup and teardown costs have to be paid for every job which runs inside of a container, which can build up. As seen in figure 7, looking at Singularity's startup and tear down costs, there is a real cost in starting up containers. The cost might be small at first, but if the

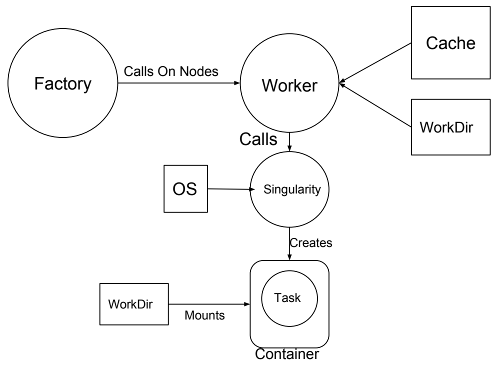

Figure 5: Container in Worker
The factory calls different workers, each with their own cache and different jobs. The jobs are wrapped in a container made from the pulled in OS image, and the working directory is mapped into the container.

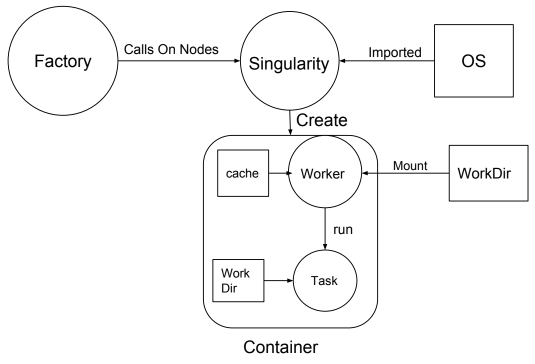

Figure 6: Worker in Container
The factory calls many different workers, each with their own cache, and different jobs. The image is pulled in to create the container, and a working directory is mounted into the image for cache space and creating per-job working directories. This design has the worker itself wrapped in a container, amortizing the cost of running a container across all jobs.

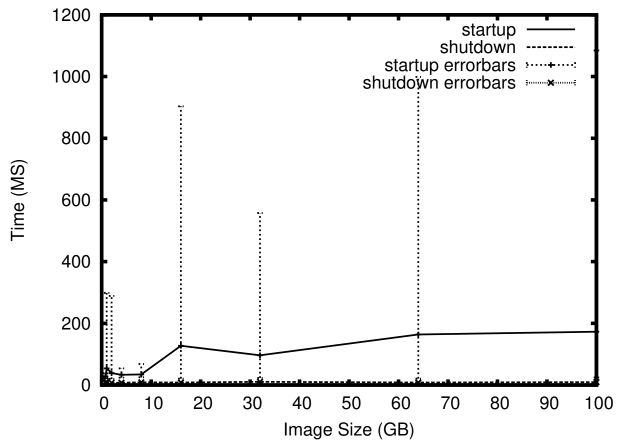

Figure 7: Singularity Startup/Shutdown
Here, we are plotting the average startup and teardown times in milliseconds of different sized Singularity containers measured in GB. The error bars are the extrema averaged data.

jobs themselves are small, and there are a large number of these small jobs, then the small cost of starting up and tearing down a container can add up.

In the other setup, the workers themselves are wrapped in a container, as seen in figure 6. In this configuration, the OS is automatically shared between the different jobs, eliminating the need for transferring it for every job, and the startup+teardown costs of the container are amortized across all jobs ran by that worker.

At first it might seem obvious that the benefits of simply running every job in a container by itself might outweigh any benefits of running a worker inside of a container (caching the images has the same effect as running the same image for all jobs, being able to customize a container for each job, etc), things become more

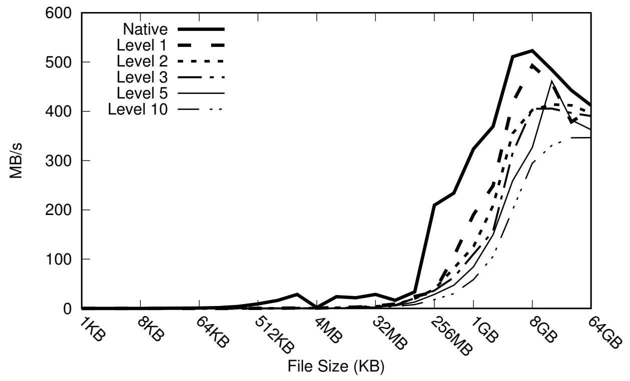

Figure 8: Read Performance Overhead
Here we plot the size of the files on a logarithmic X-axis and the read speed on the Y measured in MB/s.

complicated when looking at transforming containers between different container technologies.

5 CONTAINER TRANSFORMATION

Three widely used container technologies are Docker, Singularity, and Charliecloud. Each container technology has their own advantages and disadvantages e.g Docker's wide-spread usage but requiring a root-daemon. When considering integrating containers into a workflow, which container technologies are available becomes a pressing issue. If a workflow is being ran using resources at three different sites, each of the different sites might use different technologies, meaning transformation of the given containers needs to happen, adding more steps to the workflow itself. There are three configurations for when to convert images: convert images before running the workflow, convert just before running the jobs, and converting and running workers inside of the containers.

Table 1: Time to convert between container types

Type	Mean	stdev	Num
Docker Tar to flat-tarball	36.22 s	35.80 s	73
Singularity to tarball	199.34 s	299.29 s	75
Charliecloud Docker to Tarball	22.26 s	15.75 s	74

Table 2: Time to import and run *echo hello*

Type	Mean	stdev	Num
Docker -> Docker	16.10 s	9.42 s	69
Docker -> Singularity	5.56 s	5.01 s	71
Docker -> Charliecloud	5.77 s	4.74 s	70
Singularity -> Docker	56.12 s	61.99 s	50
Singularity -> Singularity	66.84 s	80.11 s	75
Singularity -> Charliecloud	58.32 s	74.66 s	75

In the first case, the user will have more files to keep track of, but can amortize the transformation costs across the jobs which use it. In the second case, it might be worth it if the transformation cost is small compared to the job length and there are not that many jobs. In the last case, the user still has to deal with many files, but now the conversion and startup costs are amortized across jobs.

We tested converting between different container technologies, importing, and running *echo hello* inside of the container. Each container was converted into a tar-ball containing a filesystem tree. We ran our tests on a 6-core virtual machine using sudo permissions, to ensure a clean translation and avoid file permission issues. We took 75 of the most downloaded Docker containers from Dockerhub, and the first 75 downloadable images from Singularity hub online. The results for converting from native types to a usable tarball are shown in table 1. Flattening from Docker involved simply copying out the layers using tar, excluding caches, Docker metadata, and /dev files, where as Singularity was simply to unsquash and tar up. Each conversion requires a decent amount of time, on the order of seconds to minutes. Importing and running times are shown in table 2. It takes different amounts of time for the different container technologies to import and run them, ranging from about 5.6 seconds to 67 seconds on average. Additionally, some technologies such as Singularity and Charliecloud, have their own ways of importing Docker images which can be faster than converting to flat-tarball and running from that. Not all conversions were fully successful, with number of succeeded shown in the *Num* column.

6 EXAMPLE WORKFLOW

To tie all of this together, we ran an example workflow which implements a parallel Burroughs-Wheeler Alignment and Genome Analysis Toolkit (BWA-GATK), A bioinformatics toolkit for genetic analysis, which is a complicated workflow with 530 jobs [2]. Our setup was to establish 10 workers each only using 1 core on our local cluster and run the workflow in two different configurations: one where the Docker image was converted into a Singularity image before handing it off as an input to the individual jobs, and the other where the Docker image was given to the jobs and have the jobs themselves convert before running. Figure 9 shows that in the pre-converted image case, our jobs had a bimodal distribution,

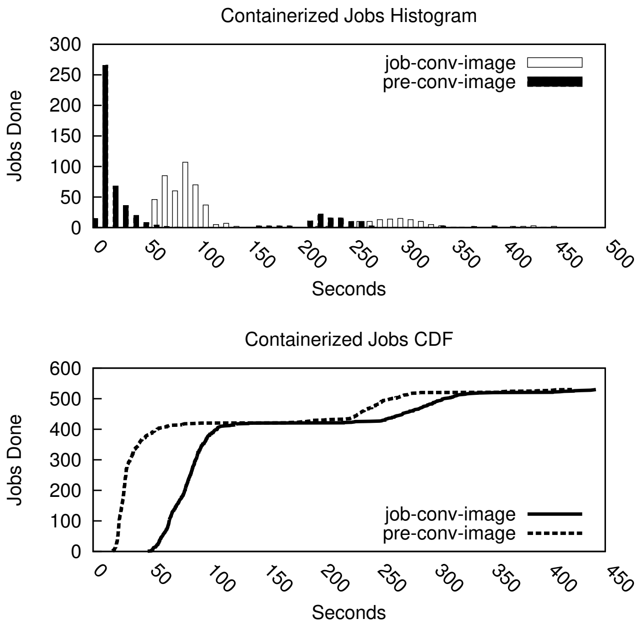

Figure 9: Runtimes of Individual Jobs

with pre-converted image jobs running mostly between 20-70s and 230-280s, with our convert-at-jobs jobs running between 60-110s and 260-320s, due to the 1min delay for converting the images. This implies that it's better to transform images once, before needing to send them to the jobs.

7 CONCLUSION

Taking all of the previous sections combined, the best strategy for introducing containers into a workflow would be to take a dynamic-approach and enable the batch system to cache as many parts of the workflow as possible, combining together at the very end, while transforming images once, and not spending repeat operations to tie them together.

8 RELATED WORK

Other work also is concerned with integrating containers into workflows, such as *Skyport* [4]integrating Docker containers into their AWE/Shock workflow system. Their example deployment also has "workunits" running inside Docker containers, and mounting their working directories into the container, and is called by the workers. Similarly, Higgens et al [6] also discussed the pros and cons of running Docker containers per node or per-job, but not integrated in a workflow system. Another issue that comes up with container integration is transferring images, and image data build-up using certain systems, e.g Docker. For example, *Slacker* [5] uses virtual machine image management technology to deal with this issue. Ramon-Cortes et al [11] created a system for enabling users to transparently deploy containers on Docker or HPC clusters, parallelizing their code.

REFERENCES

[1] 2017. BLAST Workflow Example. (2017). https://github.com/cooperative-computing-lab/makeflow-examples/tree/master/blast

[2] 2017. BWA-GATK Workflow Example. (2017). https://github.com/cooperative-computing-lab/makeflow-examples/tree/master/bwa-gatk

[3] Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain. 2012. 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*.

[4] Wolfgang Gerlach, Wei Tang, Kevin Keegan, Travis Harrison, Andreas Wilke, Jared Bischof, Mark D'Souza, Scott Devoid, Daniel Murphy-Olson, Narayan Desai, et al. 2014. 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, 25–32.

[5] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers.. In *FAST*. 181–195.

[6] Joshua Higgins, Violeta Holmes, and Colin Venters. 2015. Orchestrating Docker Containers in the HPC Environment. In *International Conference on High Performance Computing*. Springer, 506–513.

[7] Gregory M. Kurtzer. 2016. Singularity 2.1.2 - Linux application and environment containers for science. (Aug. 2016). https://doi.org/10.5281/zenodo.60736

[8] Chee Sun Liew, Malcolm P. Atkinson, Michelle Galea, Tan Fong Ang, Paul Martin, and Jano I. Van Hemert. 2016. Scientific Workflows: Moving Across Paradigms. *ACM Comput. Surv.* 49, 4, Article 66 (Dec. 2016), 39 pages. https://doi.org/10.1145/3012429

[9] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014). http://dl.acm.org/citation.cfm?id=2600239.2600241

[10] Reid Priedhorsky and Tim Randles. 2017. Charliecloud: Unprivileged Containers for User-defined Software Stacks in HPC. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 36, 10 pages. https://doi.org/10.1145/3126908.3126925

[11] Cristian Ramon-Cortes, Albert Serven, Jorge Ejarque, Daniele Lezzi, and Rosa M. Badia. 2018. Transparent Orchestration of Task-based Parallel Applications in Containers Platforms. *Journal of Grid Computing* 16, 1 (01 Mar 2018), 137–160. https://doi.org/10.1007/s10723-017-9425-z

[12] Yong Zhao, Jed Dobson, Ian Foster, Luc Moreau, and Michael Wilde. 2005. A Notation and System for Expressing and Executing Cleanly Typed Workflows on Messy Scientific Data. *SIGMOD Rec.* 34, 3 (Sept. 2005), 37–43. https://doi.org/10.1145/1084805.1084813

[13] Charles Zheng and Douglas Thain. 2015. Integrating Containers into Workflows: A Case Study Using Makeflow, Work Queue, and Docker. In *Workshop on Virtualization Technologies in Distributed Computing (VTDC)*.