Integrating Containers into Workflows: A Case Study Using Makeflow, Work Queue, and Docker

Chao Zheng and Douglas Thain
Department of Computer Science and Engineering, University of Notre Dame
{czheng2,dthain@nd.edu}

ABSTRACT

Workflows are a widely used abstraction for representing large scientific applications and executing them on distributed systems such as clusters, clouds, and grids. However, workflow systems have been largely silent on the question of precisely what environment each task in the workflow is expected to run in. As a result, a workflow may run correctly in the environment in which it was designed, but when moved to another machine, is highly likely to fail due to differences in the operating system, installed applications, available data, and so forth. Lightweight container technology has recently arisen as a potential solution to this problem, by providing a well-defined execution environments at the operating system level. In this paper, we consider how to best integrate container technology into an existing workflow system, using Makeflow, Work Queue, and Docker as examples of current technology. A brief performance study of Docker shows very little overhead in CPU and I/O performance, but significant costs in creating and deleting containers. Taking this into account, we describe four different methods of connecting containers to different points of the infrastructure, and explain several methods of managing the container images that must be distributed to executing tasks. We explore the performance of a large bioinformatics workload on a Docker-enabled cluster, and observe the best configuration to be locally-managed containers that are shared between multiple tasks.

1. INTRODUCTION

Workflows are a widely used abstraction for representing large scientific applications and executing them on distributed systems such as clusters, clouds, and grids. Broadly speaking, a workflow is a graph of sequential tasks that are joined together by the files that they create and consume. The graph as a whole can be considered a single program with a high degree of concurrency. Today, there exist a variety of workflow systems that share common principles but are tailored to serve distinct communities and use cases including HPC machines, commercial clouds, and wide area distributed systems [16, 13, 14, 8, 7, 10, 12, 21].

Generally speaking, workflow systems have focused on data interchange between tasks, while being largely silent on the question of precisely what **environment** each task is expected to run. That is, a given task depends upon the presence of a particular operating system, software installation, network configuration, and so forth to run. Often, an end user will construct a workflow on a particular system, and then move it to another system only to discover that the workflow structure works correctly, but the individual tasks fail due to incompatibilities with the new system, resulting in laborious troubleshooting.

Virtual machines (VM) have long been advocated as a solution to the problem of provisioning a consistent environment. While this solution is effective, it imposes significant costs in CPU and I/O performance, as well as moving and instantiating virtual machine images [20, 15]. Lightweight container technology has recently arisen as an alternative to complete virtualization. The basic technology has been present in Linux distributions for many years, and has recently been standardized and adopted in the form of tools such as Docker [3] and Rocket [1]. Instead of starting a complete operating system on top of a host system, a container shares a kernel with the host system, which largely eliminates overheads, but maintains isolation between applications. This makes it possible to ship a relatively small container that acts like a complete operating system but encapsulates only those files necessary to run an application.

This paper considers how to best integrate container technology into workflow systems. As a running example, we consider the specific technologies of the Makeflow [7] workflow system, the Work Queue [9] execution system, and the Docker container system, but the general principles apply across technologies. The design considerations can be broken down into three categories: how to express the desired environment for a workflow and its tasks, how and where to instantiate containers for each task, and how to manage the movement and transformation of images.

Specifically, we examine four different methods of integrating Docker with Makeflow and Work Queue. (1) Wrapping each task at the workflow level, which is simple but inefficient due to lack of information about image state. (2) Running each worker inside a container, which eliminates startup overheads, but exposes more of the system to container overheads. (3) Running each task inside a container, which limits the scope of the container, but increases startup/shutdown events. (4) Running multiple compatible

VTDC'15, June 15, 2015, Portland, Oregon, USA.
Copyright © 2015 ACM 978-1-4503-3573-7/15/06 ...$15.00.
DOI: http://dx.doi.org/10.1145/2755979.2755984.

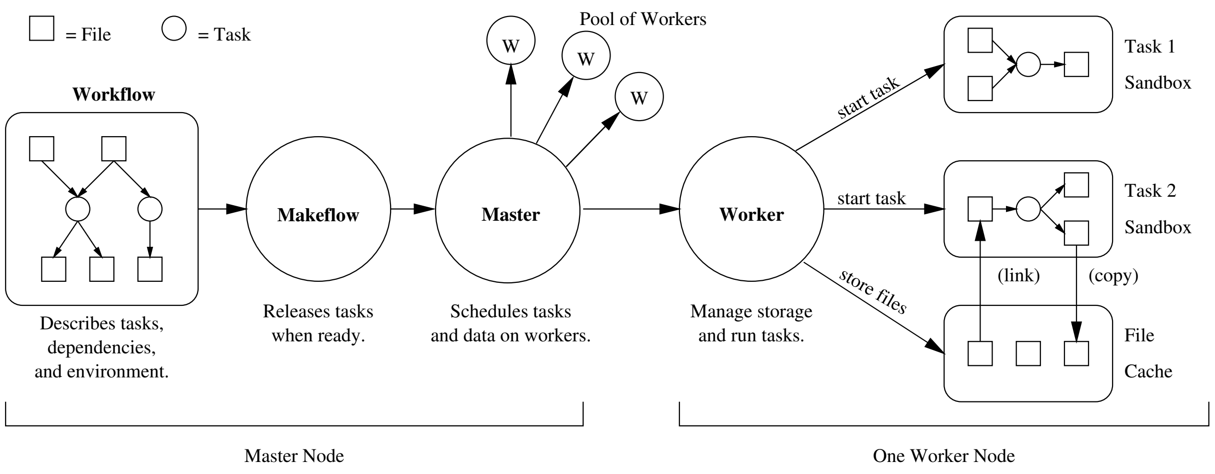


Figure 1: System Architecture Before Containers

tasks inside one container, which minimizes startup/shutdown events, but decreases isolation.

To evaluate these configurations, we execute a bioinformatics workflow which consists of a large number of relatively short tasks, which emphasizes the efficiency of container startup and shutdown. The final configuration achieves performance very close to that of a system without containers, if one is willing to sacrifice isolation between tasks. We observe that this technique requires that the execution system must be container-aware, rather than simply wrapping each task with the desired container operations.

2. BACKGROUND

Figure 1 shows the architecture of Makeflow and Work Queue before introducing container technology. While our discussion focuses on these technologies, similar comments apply to other workflow systems.

Makeflow is a command line workflow engine used to execute data-intensive scientific applications on a variety of distributed execution systems. The end user expresses a workflow using a syntax similar to that of classic Make. Each task in the Makefile is defined as a command line to be executed, prefixed by the output files that it produces and the input files that it requires. For example, to produce the file `index` from `seq.fasta` by running the command `bwa -index seq.fasta`, one would write the following rule:

```
index : seq.fasta bwa
      ./bwa -index seq.fasta
```

The semantics of Makeflow are as follows. For each task to be executed, a fresh sandbox directory will be created, the input files will be copied into the sandbox, the command line run, and then the output files moved out of the sandbox. No other files apart from those mentioned as input dependencies are guaranteed to be present, and so the end user must be careful to mention *all* of the files needed by the command. Note that, in the rule above, the application itself (`bwa`) is mentioned as an input dependency. (In this way, Makeflow is more strict than traditional Make.)

Given this generic statement of a workflow, Makeflow can execute the same workflow across a variety of execution systems, including batch systems such as Condor, Torque, SGE, and Work Queue, which is described below. Regardless of the system in use, Makeflow is responsible for releasing and tracking events in a transaction log.

Work Queue is a lightweight user-level execution engine for distributed systems. The system consists of a master library and a large number of worker processes that can be deployed across multiple cluster, cloud, and grid infrastructures. The master exports an API that allows the user to define tasks consisting of a command line to execute, a set of input files, and a set of expected output files. The master schedules tasks to run on remote workers.

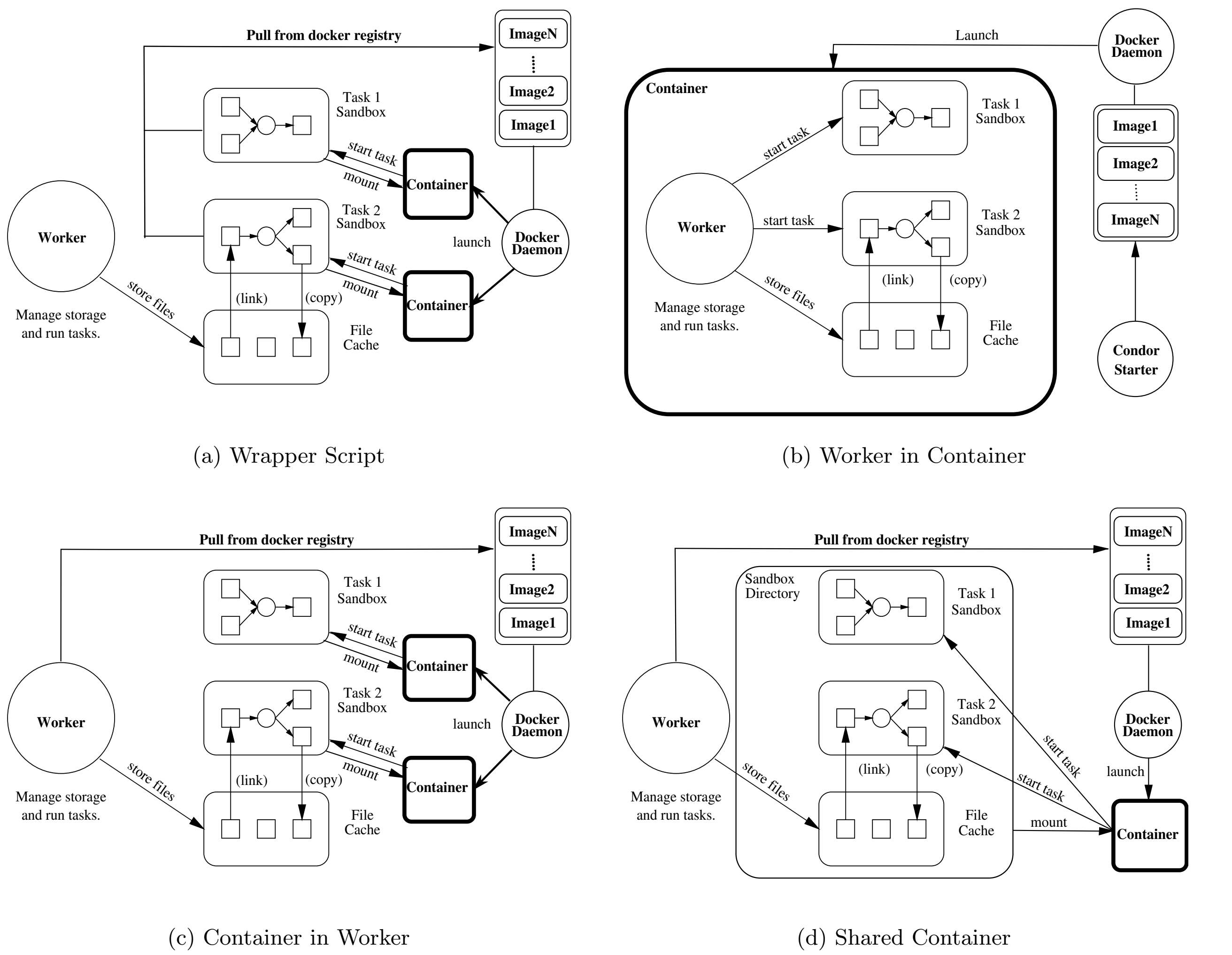
The worker executes tasks as follows. As input files are transmitted from the master, they are stored in a cache directory. For each task to be run, the worker creates a temporary sandbox directory, links in the input files, runs the command, and then copies the output files back to the cache directory, where they await return to the master. In this way, each task is given a fresh namespace that does not interfere with other tasks. On a multicore machine, multiple tasks may safely execute simultaneously. (As long as each stays in its current working directory.)

Work Queue can be used by itself to build dynamic submit-wait applications, but it is also available as an execution option for Makeflow, such that each rule in the workflow is submitted as a task for Work Queue. The implementation of Work Queue is designed to provide precisely the execution semantics expected by Makeflow. Because both Makeflow and Work Queue require all files to be explicitly declared, the combination can run without a shared filesystem.

3. EXECUTION ENVIRONMENTS

As described so far, Makeflow and Work Queue accurately capture the data dependencies of a workload, but say nothing about the **environment** in which a task should executed. Including the program as an input dependency is a good first start, but does not capture all of the shared libraries, script interpreters, configuration files, and other items on which it may depend. Ideally, the end user will provision worker nodes that have exactly the same operating system, installed applications, and so forth.

Unfortunately, this is easier said than done. Our experience is that end users construct complex workflows in an



(a) Wrapper Script

(b) Worker in Container

(c) Container in Worker

(d) Shared Container

Figure 2: Container Management Options

environment not of their own creation, such as a professionally managed HPC center. In this situation, the user's environment is a mix of a standard operating system, patches and adjustments made by the staff, locally installed applications, and items from the user's home directory. When moving the application to another site, it is not at all obvious what components should be included with the workflow. Should the user copy the executables, the libraries, the Perl or Python interpreters, or perhaps even the entire home directory? It is difficult for the expert – much less the end user – to know when to stop. (In previous work, we demonstrated how automated means [19] can be used to observe and capture dependencies, but this results in workflows that are even more complex and difficult to modify.)

An alternate approach is to **define the environment explicitly** when the workflow is created. Rather than accept the current environment as the default, the user would be required to explicitly name an image that contains the operating system, applications, and all other items needed by the application. This image could be constructed by hand by the user, but is more likely provided by administrators. Once explicitly named, the image can easily be transported along with the workflow. A single image might apply to all tasks in the workflow, or might vary between tasks.

Lightweight containers are a promising operating-system virtualization technology that could be used to deliver such execution environments. Unlike virtual machines, containers are implemented by mounting filesystems on top of an existing operating system kernel, largely eliminating the overheads found in traditional virtual machines. While the basic technology behind containers has been available for decades, the concept has recently seen considerable development in the Linux community, combining the `cgroups` resource control framework and the `unionfs` filesystem management to provide comprehensive isolation. Image management facilities such as Docker [3] and Rocket [1] now make it relatively easy to create, share, and deploy container images by name. For example, this command:

```
docker run ubuntu bwa –index seq.fasta
```

results in the `docker` command line tool contacting the `dockerd` server on the same machine to download the image named `ubuntu`, create a running container, and then execute the command `bwa -index seq.fasta` within that container.

However, the combination of these technologies is not as simple as putting `docker run ubuntu` in front of every command, because the workflow data dependencies must be connected to the executing image. To address this problem, we must consider two different design questions. The first is **container management**: namely, which component of the system is responsible for configuring, deploying, and tearing down containers. The second is **image management**: namely, how the container images must be moved to the (possibly thousands) of workers that comprise the system in an efficient way. The remainder of this paper considers each of these problems in detail.

4. CONTAINER MANAGEMENT

First, we consider four strategies for assigning the responsibility of creating and tearing down containers within the workflow system. Each of these methods has been implemented and is evaluated below.

Base-Architecture. (Figure 1) The basic architecture of Makeflow and Work Queue effectively uses a directory as a placeholder for a container. For each task to be executed, the input files are linked from the cache directory of the worker into the task sandbox directory, the task is run within the directory, and then the output files are copied back into the worker's cache for later use. This can be thought of as a (very) lightweight container inasmuch as each task has an assigned namespace, assuming each task is well behaved and stays within its current working directory.

This method has the advantage that it is simple, requires no special privileges, and imposes no overhead on the execution of the application. Of course, the only environment that can be provided to the application is the operating system in which the worker runs.

Wrapper-Script. (Figure 2(a)) The simplest step up from the base architecture is to use a wrapper script to provision a container for each task. A small script can be written which will contact the local `dockerd`, pull the desired image to the execution host, run the desired task in the container, then tear down the image. To simplify access to the task's files, the task sandbox directory is mounted into the container as the task's working directory, such that neither the task nor the worker must change their behavior.

This method has the advantage that no change is necessary to either Makeflow or Work Queue, and can be applied transparently by the end user. Each task will be isolated from all other concurrent tasks. If necessary, different tasks could execute in different environments. (In the common case that all tasks require the same environment, the `-wrapper` command-line option to Makeflow can be used to easily apply a single wrapper globally without modifying the workflow itself.) However, as we show below, this method imposes a container startup and shutdown cost on each task, and does not give the distributed system visibility into the location and movement of the (possibly large) images.

Worker-in-Container. (Figure 2(b)) An alternative approach is to simply take the entire worker itself and place it into a container for the entire duration of the run. This requires a small modification to the provisioning of the worker itself, to pull the image and create the container. In fact, the same wrapper script as above can be used, if the worker itself is provisioned by an underlying system manager.

This approach succeeds in delivering the desired execution environment to each task and avoids paying the startup and shutdown costs for each task. In the common case where all tasks in the workflow require the same environment, one can easily imagine provisioning a number of workers in bulk before the workflow execution. However, it does not provide isolation between tasks, which all execute in a sandbox directory in the same container, so we must assume the same degree of trust as in the base case. More subtly, the worker itself must pay the costs of executing within the container, such that there may be a penalty applied to the network communication between master and worker, as well as in the management of the local cache directory. Finally, this configuration relies on the `aufs` filesystem, which is reported to cause non-negligible overheads when using a container with multiple layers and deep-nested filesystem structure.

Containers-in-Worker (Figure 2(c)) Another approach is to modify the worker code itself to run each task within a specified container. The effect of this is very similar to that of the wrapper script, but with one important difference: the worker itself now has some knowledge of the state of the local `dockerd` and can execute more efficiently. Rather than attempting to pull and transform images for each container invocation, the worker can prepare the image once, and then instantiate multiple containers from the same image.

This approach allows each task to provision a distinct container and allows the worker itself to avoid container overheads, while still providing isolation between tasks. Further the container image itself can becomes an input dependency of the task, which allows for the scheduler to explicitly take advantage of this information. For example, tasks can be scheduled preferentially to nodes where the image has already been transferred and cached.

Shared-Container. (Figure 2(d)) Finally, we may attempt to combine the benefits of multiple approaches by allowing tasks to share the same container where possible. In the Shared-Container approach, the worker is again responsible for creating and deleting containers, but will only create one container for each desired environment. Containers are not removed when tasks complete, but remain in place for the next (or concurrent) task to be placed inside.

This approach allows each task to run in different environments, where needed, avoids the overhead of multiple container creation, but does not provide isolation between tasks beyond the base case. As with the previous case, it also gives the scheduler visibility into image locations.

5. IMAGE MANAGEMENT

If we consider a large scale workflow application running on thousands of workers, the cost of moving the environment to each node can become a significant component of the cost, depending on the form in which it is transferred and the source of the transfer. There are three commonly used forms for communicating an executable environment in Docker:

A **dockerfile** describes the entire procedure by which an image is built, starting with the base operating system image, adding software packages, and running arbitrary commands until the desired result is achieved. Building an image from a dockerfile is comparable to installing a new machine from scratch and could take anywhere from minutes to hours, depending on the number of layers involved. However, a dockerfile is quite small (1KB or less) and is easily transferred across the system.

A **binary image** is the result of executing a dockerfile, and is the executable form of a container ready for activation, with all files laid out into a binary filesystem image in a form that can be directly mounted and used. The image is a binary object specific to the kernel filesystem in use and may not be appropriate for portability or long term preservation.

A **tarball** is a more portable form of a binary container, in that all the layers of the filesystem have been collapsed into a tree of files and directories encoded in the standard `tar` format. A tarball is much more suitable for sharing and preservation, but must be unpacked and encoded back into a binary image before it can be executed directly by Docker. A tarball can be used directly by other technologies, such as a `chroot` based sandbox.

By default, Docker encourages end users to create containers by pulling binary images from the global Docker Hub. This would be the result of using the Wrapper-Script method with reference to an image name. This method is quite useful for sharing frequent-used container images over limited number of computing nodes. But, when employing thousands of workers for a single workflow, this could result in extraordinary loads on the public network. Further, using the central hub may be inappropriate for images with security, copyright, or privacy concerns.

Another approach would be using the workflow system to distribute the dockerfile itself, relying on the workers to construct the desired images at runtime. This would dramatically reduce the network traffic from workers pulling images, but would result in all workers duplicating the same effort for minutes to hours to generate the same resulting image. The effort would be better expended in one location before allocating workers.

In the context of a large workflow on thousands of nodes, the most appropriate solutions seems to be for the workflow manager itself to build the desired environment image on the submit machine, either by executing a dockerfile or by pulling an image from a repository. Once generated, that image can be exported from Docker as a portable tarball and then included as an input dependency for each task. In this way, the Work Queue scheduler has sufficient knowledge to distribute the environment using the existing file transfer mechanisms on the private network of the system, has much larger network bandwidth than the public network. Furthermore, the container images can be cached on the individual workers for later use.

Since there is currently a known bug in Docker that results in a failure when containers are built from the same tarball simultaneously. In the examples that follow, we rely on the global Docker hub for images management, where they are pulled and installed.

6. EVALUATION

We evaluated each of the configurations above by implementing them as options within Makeflow and Work Queue, then observed the performance of a large bioinformatics workflow on a Docker-enabled cluster. The cluster consisted of twenty-four 8-core Intel Xeon E5620 CPUs each with 32GB RAM, 12 2TB disks, 1Gb Ethernet, running Red Hat Enterprise Linux 6.5 with Linux kernel 2.6.32-504.3.3.el6.x86_64 and Docker 1.4.1.

Before evaluating the workflow as a whole, we performed some basic micro-benchmarks on a single machine to evaluate the low level performance of the container technology. We employed `sysbench` to measure CPU performance and memory bandwidth, `netperf` to evaluate the performance of

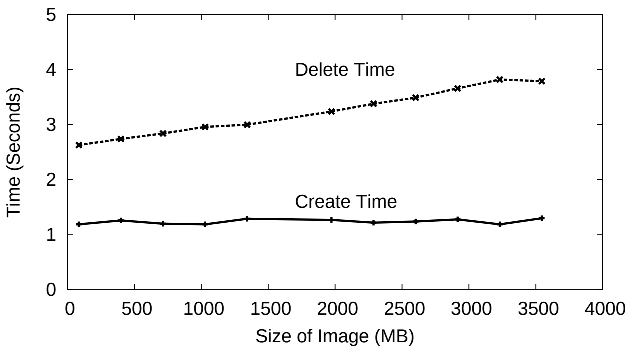


Figure 3: Container Creation and Deletion

TCP throughput both on and off the machine and `bonnie++` to evaluate local disk performance. We expected (and confirmed) that the CPU, memory, and network benchmarks would all result in virtually indistinguishable performance between the host and container, since the container mechanism primarily affects the namespace of kernel objects and not the direct access to machine resources.

However, based on previous reports, we expected to see that the I/O performance within the container would take a significant penalty due to the use of `aufs`. For example, Felter [11] mention that `aufs` should be avoided due to the overhead of metadata lookup in each layer of the union filesystem. What we observed was more difficult to characterize: generally, reads from `aufs` would see performance similar to that of the host, while writes to `aufs` would be sometimes be *faster* than the host, as changes were simply queued up until a later `docker commit` which would flush changes out. The overall effect was very inconsistent performance, sometimes better and sometimes worse than the host filesystem.

That said, `aufs` performance is less relevant in this setting because three of our configurations relies on mounting a sandbox directory from the host in order to access workflow data. The performance of this mount was indistinguishable from the host. The Worker-in-Container relies on `aufs`, but the size of the input file and output file for each task are around 10MB to 20MB. To access small files like this, the difference of I/O performance is negligible.

Where overheads were more clear was in the cost of creating and deleting containers at runtime. To evaluate this, we created a base operating system image, and then progressively increased the size of the image by adding files. We then measured the minimal container startup time by executing `docker run debian /bin/ls` and then the time to delete the image after the command completed. Figure 3 shows the average of ten startups and deletes at each size. We were surprised to see that the startup time was essentially constant with respect to the image size, but the deletion time increased linearly with the image size. These overheads have a significant effect on workflow executions, depending on the container management strategy chosen.

To evaluate the overall system performance, we executed a large bioinformatics workload in each of the five container management configurations discussed above.

Commonly, there are three kinds of workflow, workflow consists mainly of long-running tasks, one containes many

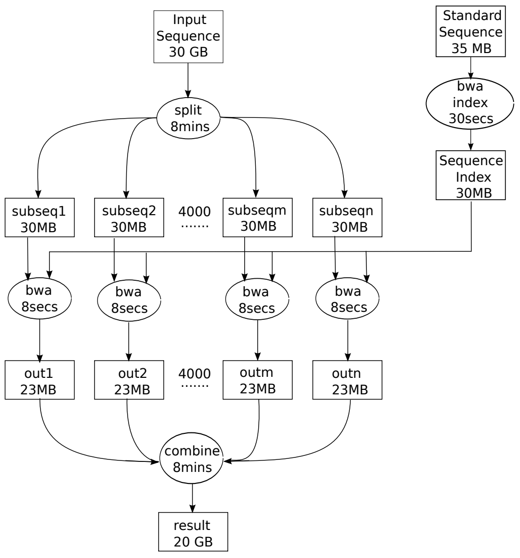


Figure 4: BWA Workflow

short-running tasks and the one include both type of tasks. For long-running tasks, in comparison to the overall execution time, environment setting up time is negligible. While the workflow has many short tasks gains concrete benefits from shorter environment setting up time.

In order to present the maximum level of speedup gain from applying lightweight container technology, The selected workload is a parallelization of the widely used Burrows-Wheeler Alignment (BWA) tool, which consists of 4082 short-running tasks. BWA is a genomic search tool which looks for instances of query strings within a large reference database by applying the applies the Burrows-Wheeler Transform (BWT) algorithm. The original tool is a standalone sequential program which accepts a list of queries and a reference database.

The structure of the workflow is shown in Figure 4. An initial task subsamples a 30GB input file into 4000 parts, each of which is then fed into a separate instance of BWA which searches for the queries in a shared reference database of 30MB, producing results of approximately 23 MB each, which are then joined into a single ouptut file which must be brought back to the master.

The workload was run on a cluster, each task a single-core process, such that up to 192 tasks (or containers) would run simultaneously. Workers were deployed onto the cluster in each of the five configurations described earlier, with Makeflow running on the head node to coordinate the computation.

Figure 5 shows the results of each run, with each configuration in a row. The first column gives the key details of the total runtime, the average execution time of each task, and the average transfer performance between the master and the worker. The second column gives a histogram of individual task execution times for the 4000 tasks of the workflow, while the third column gives a histogram of transfer rate of individual file between the master and the worker to support each task.

As expected, the Base-Architecture has the fastest overall execution time (25 min) and a compact distribution of task execution times. The simplest extension, the Wrapper-Script, has the worst performance of the five (38 min), primarily because each task must independently pull an image, execute the task, and then clean up the image when done. The variation in task execution times is also much higher, due to the interference between tasks managing images and doing work. Worker-in-Container achieves faster and more compact task execution times but still pays a penalty due to overhead applied to the worker itself. Containers-in-Worker shows a slightly worse performance because each task must create and delete a container, but the worker handles the image management. Finally, the Shared-Container case achieves performance very close to that of the Base-Architecture, because the containers are created once per worker, and then shared among up to eight tasks simultaneously.

We included the file transfer histograms because we expected to see some variation in transfer performance, particularly in the case of the Worker-in-Container. However, we do not see any differences significant enough to draw conclusions.

As can be seen the selection of a strategy for managing containers within a workflow has a significant impact upon the bottom line, primarily due to the non-trivial expense of booting and removing containers. A tradeoff must be made between achieving complete isolation and maximum performance.

7. RELATED WORK

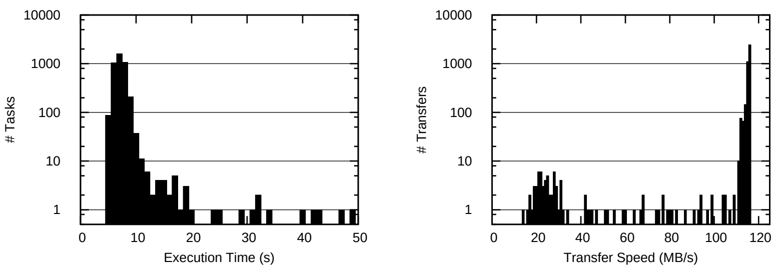
There exist a large variety of workflow systems that share similar principles while addressing the needs of different user communities and use cases. Examples include DAGMan [13], Galaxy [14], Kepler [8] Pegasus [10], Swift [12], and Taverna [21]. The general principles and tradeoffs considered in this paper could be applied to all of these systems.

Combining container technology with distributed system is not a new idea. Core OS [2] as a lightweight Linux distribution aims at providing infrastructure to clustered deployments, running all services and applications inside containers, which facilitate the security, reliability and scalability of clusters. Other two popular platforms are Snappy ubuntu [6] and RedHat Project Atomic [5]. Both platforms minimize operating system level contents and automate the process of deploying containers across multiple hosts. Google also annouce the Kubernetes project [4], which enable the user to manage a cluster of Linux containers as a single system. These projects focuse on how to adopt container technology on system level or how to coordinate containers accross cluster environment. In our papers, we show the possibilities to integrate container technology into existing cloud computing framework, which exploit different ways of applying container on framework level.

There are many ways to improve the performance of High Performance Computing system whose main enabling technology is virtualization. By using hardware with incorporate with virtualization technology, which reduces the latency caused by virtualization [18]. Another way is to enable the hypervisor to deal with the guest processing directly, which eliminating the overheads for latency sensitive tasks [17]. By reducing the memory footprint in virtualized large-scale parallel system, we can enhance the system performance, reliability and power [22]. All these strategies keep using the traditional virtual machine technology. We try to exploit the possibilities to integrate Lightweight container technology to cloud computing framework, which achieves isolation

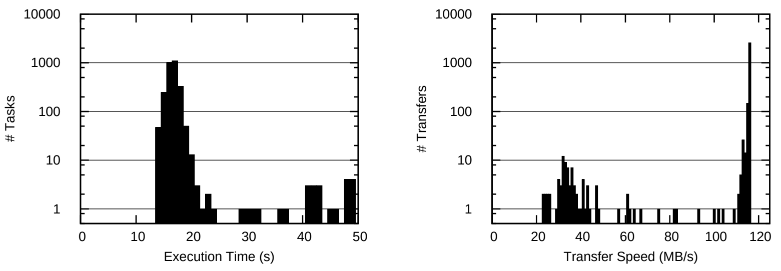
Basic Structure

Total exc time: 25mins
Task time: (8.28 +/- 2.52)secs
Transfer rate: (115 +/- 11)MB/s



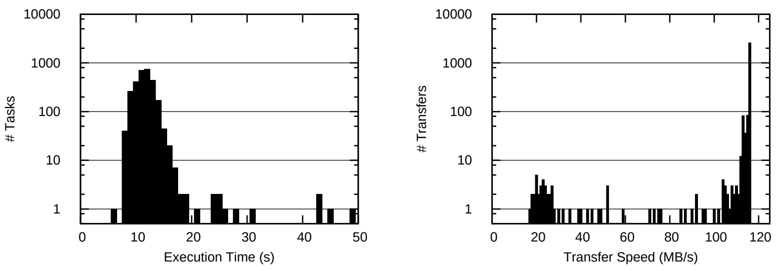
Wrapper Script

Total exc time: 38mins
Task time: (18.93 +/- 12.07)secs
Transfer rate: (114 +/- 13)MB/s



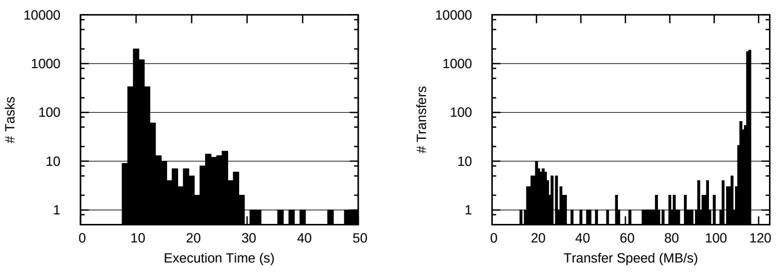
Worker in Container

Total exc time: 30mins
Task time: (10.72 +/- 4.79)secs
Transfer rate: (113 +/- 13)MB/s



Container in Worker

Total exc time: 33mins
Task time: (11.89 +/- 3.17)secs
Transfer rate: (114 +/- 13)MB/s



Shared Container

Total exc time: 26mins
Task time: (8.37 +/- 2.54)secs
Transfer rate: (115 +/- 11)MB/s

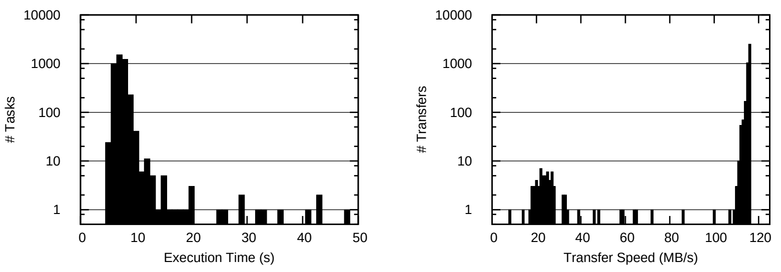


Figure 5: Workflow Performance for Each Configuration

In each row of the table, details of the configuration, task execution time histogram and task transfer rate histogram are presented. (1) for configuration details, total execution time, average execution time +/- standard deviation, average transfer rate +/- standard deviation are given (2) for task execution time histogram, The frequencies of tasks execution time in different time intervals are presented. The x-axis is the time intervals and the y-axis is the number of tasks in certain time interval. (3) for task transfer rate histogram, we show the frequencies of tasks' file transfer rate in different time intervals. The x-axis is the time lapse and the y-axis show the number of tasks.

with much less overheads compare to traditional virtualization technologies on cloud computing environment.

8. CONCLUSIONS

With the advent of container technology, many of the benefits of traditional virtualization can be achieved at significantly lower cost. However, as we have shown, the costs of instantiating and managing a large number of containers can have a significant impact on workflows in a distributed environment. We have demonstrated several techniques for managing containers within a production workflow system and evaluated the tradeoffs between performance, isolation, and consistency. The results show that for large workloads including thousands of tasks, launching and removing large amount of containers cause notable overheads. The overheads can be eliminated by sharing containers across multiple tasks in the cost of losing isolation for each task. More broadly, an important lesson is that the management of containers is best done explicitly by the distributed system, rather than simply leaving it to the user to invoke a container from within each task. In future, we intend to develop an advanced container management mechanism, which aims to control the number of private and shared containers on the cloud. We expect this mechanism can largely benefit different workflows that have specified bias for isolation and performance.

Acknowledgements

This work was supported by National Science Foundation grant OCI-1148330 and the Department of Energy via subcontract to the University of Wisconsin 421K072.

9. REFERENCES

[1] CoreOS is building a container runtime, Rocket. `https://coreos.com/blog/rocket/`, 2015.

[2] CoreOS is Linux for Massive Server Deployments. `https://coreos.com/`, 2015.

[3] Docker project official website. `https://www.docker.com/`, 2015.

[4] Kubernetes by Google. `http://kubernetes.io/`, 2015.

[5] Project Atomic. `http://www.projectatomic.io/`, 2015.

[6] Snappy Ubuntu Core. `http://developer.ubuntu.com/en/snappy/`, 2015.

[7] 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.

[8] 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*, pages 423–424. IEEE, 2004.

[9] 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.

[10] 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.

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

[12] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. Ieee, 2008.

[13] J. Frey. Condor dagman: Handling inter-job dependencies, 2002.

[14] 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*, 15(10):1451–1455, 2005.

[15] N. Huber, M. von Quast, M. Hauck, and S. Kounev. Evaluating and modeling virtualization performance overhead for cloud environments. In *International Conference on Cloud Computing and Services Science (CLOSER 2011)*, pages 563–573, Noordwijkerhout, The Netherlands, May 2011.

[16] 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*, 41(3):59–72, 2007.

[17] A. Nordal, Å. Kvalnes, and D. Johansen. Paravirtualizing tcp. In *Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing Date*, pages 3–10. ACM, 2012.

[18] H. N. Palit, X. Li, S. Lu, L. C. Larsen, and J. A. Setia. Evaluating hardware-assisted virtualization for deploying hpc-as-a-service. In *Proceedings of the 7th international workshop on Virtualization technologies in distributed computing*, pages 11–20. ACM, 2013.

[19] C. Robinson and D. Thain. Automated Packaging of Bioinformatics Workflows for Portability and Durability Using Makeflow. In *WORKS13*, 2013.

[20] M. Rosenblum and T. Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, 2005.

[21] 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*, page gkt328, 2013.

[22] L. Xia and P. A. Dinda. A case for tracking and exploiting inter-node and intra-node memory content sharing in virtualized large-scale parallel systems. In *Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing Date*, pages 11–18. ACM, 2012.