

# Towards Integration of Containers into Scientific Workflows: A Preliminary Experience with Cached Dependencies

Bruno da Silva Alves(✉) and Andrea Schwertner Charão(✉)

Universidade Federal de Santa Maria, Santa Maria, Brazil
{bdalves,andrea}@inf.ufsm.br

**Abstract.** A scientific workflow consists of a large number of tasks that are typically performed on distributed systems. For each node in the distributed computing system, an entire environment must be configured to be able to run a set of tasks, which depend on libraries, binaries, etc. Containers can provide lightweight virtualization capable of isolating the application and its dependencies, ensuring flexibility to run the same environment on different hosts. In this work, we investigate the integration of containers into scientific workflows, by combining Docker and Makeflow technologies. We focus on performance issues and propose a cache system adapted to containers, considering that data transfers can represent a potential bottleneck on iterative executions typically found in scientific workflows. We use Docker Volumes to circumvent the volatility of containers, allowing files stored in the cache to be available when a new container is instantiated. Our experimental results from the execution of two real-world bioinformatics workflows (Blast and Hecil) show that using our cache system effectively decreases execution times. We also show that the number of workers per host impacts the workflow execution time in different ways.

**Keywords:** Scientific workflows · Containers · Distributed systems

## 1 Introduction

Research advances usually rely on experimentation, which allows scientists to support or refute previously formulated hypotheses. Scientists from various fields use software tools to process the data collected from experiments. Also, they usually establish a workflow in which tasks are performed over data. In general, a workflow describes how processes must be coordinated to achieve a common goal [3], and the development of a workflow can help scientists to automate the experimentation process.

Each task in a scientific workflow reads some input data, perform analyses and computations, then generates output data. The output generated by a task can be used as an input for another, generating a graph. However, workflow tasks not only depend on input data, they also require implicit dependencies such as

© Springer Nature Switzerland AG 2020
O. Gervasi et al. (Eds.): ICCSA 2020, LNCS 12253, pp. 736–744, 2020.
https://doi.org/10.1007/978-3-030-58814-4_61

source code files, binary files, and libraries [13]. Configuring an environment to carry out a workflow can be a challenging task involving specialized technical knowledge that many scientists do not have.

To provide preconfigured execution environments for scientific workflows, some authors explore the advantages of lightweight container technologies [13, 15]. Containers are an OS-level alternative to virtual machines, capable of isolating the execution of multiple processes in a single environment. Containers allow the same environment to be virtualized over different hardware, providing flexibility and scalability to the system. Once the entire environment is encapsulated by a container image, scientists can easily replicate experiments in the same environment. Despite its advantages, integrating containers into scientific workflows give rise to performance and design issues that must be carefully examined.

In this work, we explore the Docker container technology integrated to the Makeflow workflow system and its Work Queue component. We propose a cache system using Docker Volume, integrated to Work Queue, as a solution to circumvent the volatility of containers, so to avoid potential bottlenecks when new containers are instantiated. The rest of this paper is organized as follows: Section 2 describes the related work on integrating scientific workflows and containers; Section 3 provides a brief background on Makeflow and Docker container composition; Section 4 describes our integrated architecture, the cache problem and our proposed solution; Section 5 present our experiments and discusses the results, followed by the conclusion in Sect. 6.

## 2 Related Work

Previous research on containers has shown that they provide a low overhead solution for virtualized systems. Using containers, we may expect close to native CPU, memory, and disk performance [8,14]. While there are still concerns about container security [5], they proved to be a more flexible virtualization solution than virtual machines [7,12]. The virtualization offered by containers allows a container image to encapsulate all the necessary dependencies for the task to be run. Many authors choose Docker as the container engine as it comprises facilitating tools (Docker Swarm, Docker Volumes, overlay network tool) and a large container image repository, known as Docker Hub[1].

Some authors have already investigated the integration of containers and scientific workflows [13,15]. This integration must take into account aspects such as technologies for managing containers on the distributed system, overhead of transferring container images over the network, and the reuse of sent files (cache). Zheng and Thain [15] explore Docker along with Makeflow and Work Queue, in a set of experiments comparing methods of connecting containers to the infrastructure. They take into account the cost of creating and deleting containers, considering a real-world bioinformatics workflow named BWA [9]. Sweeney and Thai [13] focus on issues impacting execution times and network usage when incorporating containers into workflows. They explore strategies

[1] Docker Hub: https://hub.docker.com/.

for container composition, containerizing workflow tasks, and container image translation. To explore container composition, they identify different types of data and software dependencies that may be satisfied when building containers.

Our work goes further in the track open by these previous efforts. As in [15], we also explore Docker, Makeflow and Work Queue, but propose changes to the Work Queue communication protocol and explore deeply Docker and two of its facilities: Docker Swarm and Docker Volumes. We focus on the performance of each task and how different strategies can decrease execution time for another two real-world bioinformatics workflows. As in [13], we take into account dependencies for container composition, but we do not handle container translation as we only focus on Docker containers. On another hand, we analyze the impact of cached dependencies, which is not addressed in previous work.

## 3 Background

### 3.1 Makeflow: A Scientific Workflow Engine

Scientific workflow systems provide an environment for workflows to be described and subsequently run on distributed resources. Kepler [2], Taverna [11] and Triana [10] are among the main systems that emerged. However, we choose the Makeflow engine [1] for the following reasons: ability to execute large-scale distributed computations, fault tolerance, availability of a repository with examples of scientific workflows, ease of use and compatibility with a system to run on multiple hosts. Makeflow has a simple way of describing a scientific workflow: it uses a file with syntax similar to Unix Makefiles. A task is described using the following parameters: input files, command to perform the task and output files. In addition to these parameters, it is possible to add minimum requirements for each task: available disk space, RAM and processing cores. Only the workers with minimum resources will execute the workflow tasks.

Makeflow interoperates with another tool called Work Queue [1], which is responsible for reading the tasks defined by Makeflow, distributing (task scheduling) them among the workers, send the input files and finally retrieve the output files generated by the tasks. The order of execution is important, as there are tasks that depend on the output of others. The worker nodes must then connect to the Work Queue server informing that they are able to process tasks. When a worker start its communication, the number of available processing cores, the amount of available RAM and the available disk space is sent to the Work Queue server.

### 3.2 Task Dependencies and Container Composition

The process of analyzing and defining the dependencies of each task is important to determine the files, libraries and binaries that must be present in the container image that will be used to run these tasks. We classify dependencies into two broad types: **Input/Output Dependencies:** The set of input and output data

files that each task receives and generates. This type of dependency must be addressed during the workflow execution, because new files are generated as tasks are completed. **Implicit Dependencies:** Scripts, binaries, configuration files and libraries that must be present in the OS. These files are more difficult to identify, as they are strongly related to the file system and can be placed at different locations. Implicit Dependencies are known before the execution and they are static (do not change at any phase). Those files must be present at the exactly same location on the file system to ensure the correct execution.

The first step to create Docker container images is to choose a base image. This image is a file that contains what is necessary for the execution of an operational environment. The Docker Hub provides several base images to choose from, such as Ubuntu, CentOS, Debian, etc. The base image must be chosen according to the compatibility of the packages, tools and software with the OS. From the base image, we must install all software dependencies for tasks to be performed.

The workflow execution is usually an iterative process, as scientists can add or remove tasks to the flow as the work evolves. When a new task is added to the flow, all the implicit dependencies also need to be present at the container image. As Docker images are organized by layers, so a new layer (with the new task and its dependencies) can be added on top of the previous image.

## 4 Integrated Architecture

### 4.1 System Components

Many workflows require a distributed system, so containers must be able to interconnect and communicate over the network. For this reason, we adopted Docker Swarm technology as a way to connect and manage the containers in each host. Figure 1 shows the system layout used in this work, where the Docker overlay network connects the containers that are running on hosts A and B. The manager node is located at host A, where there is also a local image repository, which is a container that acts as a local repository of all previously created images. Before the workflow execution, all hosts must download the base images from this repository. This procedure allows container images to be updated on worker nodes efficiently, as the hosts download only the layers that are not yet present in their file system.

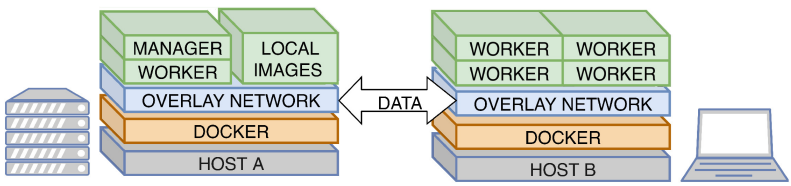


**Fig. 1.** System layout.

### 4.2 The Cache Problem

Each worker, before running, creates a sandbox (a directory on the file system) where the input and output are placed, the sandbox acts as a controlled environment where the worker can process the tasks, without worrying about changing or deleting important files from other areas. Then, when the workflow comes to an end and there are no more tasks to be processed, each worker erases its sandbox, leaving the file system unchanged. However, this approach is only valid when the worker nodes run directly on the host operating system, where changes to the file system can crash the entire machine and several processes can be stopped. When containers are used, they act as a large sandbox, where changes can be made without harming anything outside the container.

In addition to this, the entire flow must be performed several times with minor changes. In this way, when sending a file from one host to the other, it is advantageous that this file could be cached for the next iteration. Work Queue allows files to be cached using the following strategy: the manager node sends a file to the worker and records in a hash table that the node has received it. The worker node, in turn, stores this file inside its sandbox directory and keeps it until it is disconnected by the manager. When a new task uses the same previous file, the manager knows that this file has already been sent and does not sent it again. At the end of the workflow, the worker deletes all files in its sandbox, except those marked to be cached. However, this strategy cannot be applied over containers, because they are units made to be volatile and which can be replaced by other containers when an execution fails. Thus, at the end of the workflow, all the containers would be finished and the files marked to be in the cache would be lost.

### 4.3 Docker Volume as a Cache for Work Queue

To solve the cache problem, we analyzed the source code of Work Queue and Makeflow in order to determine how the cache process was done, and then modify it as follows, so the cache could work in the context of containers. The Work Queue cache system works in a way that the Work Queue server stores a list of all files that are being kept in cache by workers. The manager stores a hash table, which associates a worker identifier (key) with the files the worker has in cache (values). Each worker, when started, receives a unique identifier. To circumvent the problem of container volatility, we set up a persistent Docker Volume stored on the host's native system. Even if the container is turned off, the data stored on these volumes remains accessible. The advantage of a Docker Volume is that multiple containers can access the same volume, sharing data.

The main integration problem of this approach is that the worker node is not aware of its cached files, then it was necessary to add a message to the Work Queue communication protocol for the worker nodes to inform the manager that they already have files in their cache. Since the data in Docker Volume is shared between the containers on the same host, the file cannot be linked to the identifier of only one container, but it has to be linked to all containers ids that have access to the volume.

With this new approach, the worker node, encapsulated by a container, starts its execution and scans the files located at the Docker volume, transfer them to its sandbox directory, and then send a message to the manager containing all the hashes of the cached files. The manager node receives this information and stores the hashes of the files in the hash table, based on the identifier of each worker node. Therefore, when the workflow is re-executed, and all worker nodes (containers) are created again, the files present in the Docker volume will not be sent again, guaranteeing a cache system.

## 5 Experiments and Evaluation

The experiments described in this section aim to evaluate the impact of the proposed cache system on two real-world scientific workflows, with different number of tasks and dependencies. For each workflow, we also compare two strategies for mapping containers over the hosts: 1) one container per thread of the host; 2) one container per host, giving this container access to all the host's resources.

We selected two contrasting scientific workflows, Blast [6] and Hecil [4], from the Makeflow repository[2]. Blast has a set of 15 tasks: 3 of them run locally in the manager node and 12 tasks run distributed over the worker nodes. These tasks execute the same program (a binary named blastall) for several different entries, i.e., their software dependencies are the same. Hecil comprises 112 tasks that must be executed on the worker nodes. Hecil tasks are heterogeneous: they include Python scripts, Perl code and binary programs. Such heterogeneity characterizes an interesting scenario to stress the dependency management for executing the workflow over containers.

We set up our experimental infrastructure based on the architecture shown in Fig. 1. Host A has an Intel (R) Xeon (R) processor with 4 cores and 2 threads per core, 12 GB of RAM. Host B has an Intel (R) Core (TM) i53230M (Intel Core IvyBridge processor) with 2 cores and 2 threads per core, 8 GB of RAM. Using 11 workers, we can test the strategy 1, where 7 worker containers (plus one that should act as manager) run in Host A and another 4 workers in Host B. Using 2 workers on each host, we can test strategy 2. In total, we carried out the following set of test cases: Default Work Queue with 2 and 11 workers; modified Work Queue with 2 and 11 workers; modified Work Queue with 2 and 11 workers with clean cache.

Figure 2 shows the average execution times for Blast and Hecil workflows. Each reported execution time is the average of 10 executions. We can see that the longest execution time for Blast was obtained with 11 workers and the Default Work Queue. Despite having more workers, this case performed slowly because sending all the input files to all containers generates a bottleneck in which workers were delayed and parallel execution was hampered. The modified Work Queue leaded to the best execution times, because the files were already in the Docker Volume of each host. The cases that used the modified Work Queue and the

[2] Makeflow Repository: http://github.com/cooperative-computing-lab/makeflow-examples.

empty cache obtained better times than the executions with the Default Work Queue, since dependencies were transferred in parallel, once the data sent to the Docker Volume could be accessed by all running containers in its host.

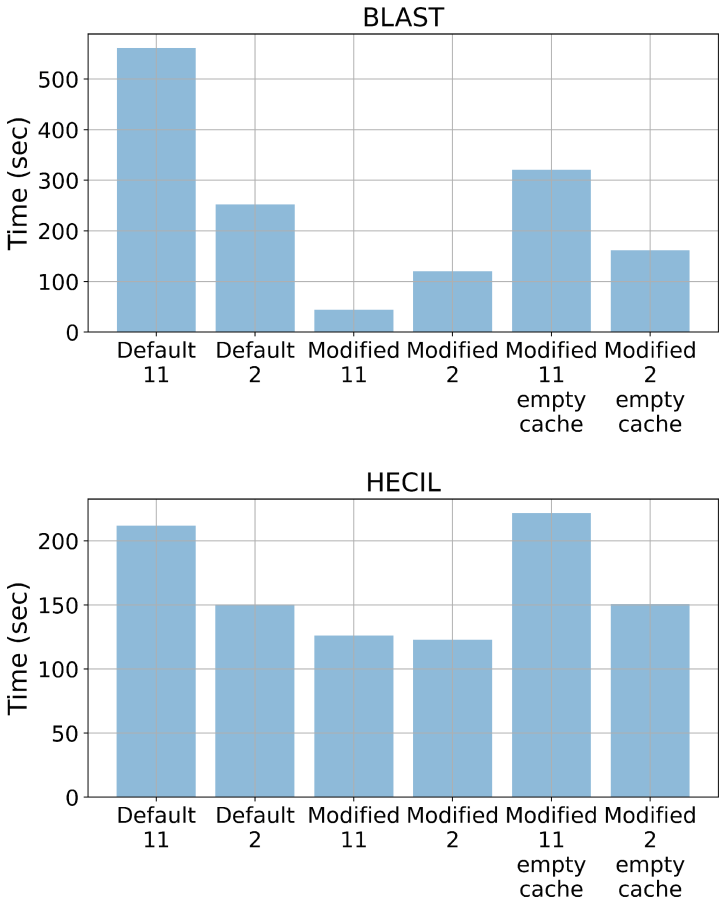


**Fig. 2.** Execution times for Blast and Hecil on multiple test cases.

In other hand, the Hecil workflow obtained the shorted times with 2 workers. This happens because this workflow comprises tasks where thread-level parallelism is better used than host-level parallelism. These results suggest that the mapping strategy should be chosen carefully. Choosing between container-per-host or container-per-threads is a decision that depends on each workflow and will impact its performance. However, we observed that using the cache was beneficial for both workflows and that using Docker Volumes decreases the network traffic, since the dependencies are sent to each host, and not to each container within the hosts.

## 6 Conclusion

Lightweight Docker containers are an effective solution to provide preconfigured execution environments for real-world workflows. In this work, we have explored Docker and Makeflow technologies to execute two bioinformatics workflows: Blast and Hecil. We have proposed a cache system leveraging Docker Volume and Makeflow's Work Queue. Our experiments have shown that using the cache was beneficial for both workflows. Also, we have shown the performance impact of container-per-host and container-per-threads strategies for mapping containers over the hosts dedicated to executing the workflows.

As future work, we intend to develop an adaptive system, where the strategies of instantiating one container per host or containers per threads could be evaluated as new rounds of the workflow are executed. We also intend to expand our experiments to other real-world workflows that may be described with the Makeflow language.

**Acknowledgements.** This research has been partially supported by the GREENCLOUD project (http://www.inf.ufrgs.br/greencloud/) (#16/2551-0000 488-9), from FAPERGS and CNPq Brazil, program PRONEX 12/2014.

## References

1. Albrecht, M., Donnelly, P., Bui, P., Thain, D.: 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, SWEET 2012, New York, NY, USA, pp. 1–13. ACM (2012)
2. Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., Mock, S.: Kepler: an extensible system for design and execution of scientific workflows. In: Proceedings of 2004 16th International Conference on Scientific and Statistical Database Management, pp. 423–424, June 2004
3. Barker, A., van Hemert, J.: Scientific workflow: a survey and research directions. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 746–753. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68111-3_78
4. Choudhury, O., Chakrabarty, A., Emrich, S.J.: HECIL: a hybrid error correction algorithm for long reads with iterative learning. Sci. Rep. **8**(9936), 1–9 (2018)
5. Combe, T., Martin, A., Di Pietro, R.: To Docker or not to Docker: a security perspective. IEEE Cloud Comput. **3**(5), 54–62 (2016)
6. Donkor, E.S., Dayie, N.T.K.D., Adiku, T.K.: Basic local alignment search tool. J. Mol. Biol. **3**(215), 403–410 (1990)
7. Dua, R., Raja, A.R., Kakadia, D.: Virtualization vs containerization to support PaaS. In: 2014 IEEE International Conference on Cloud Engineering, pp. 610–614, March 2014
8. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and Linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 171–172, March 2015

9. Li, H., Durbin, R.: Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics **25**(14), 1754–1760 (2009)
10. Majithia, S., Shields, M., Taylor, I., Wang, I.: Triana: a graphical web service composition and execution toolkit. In: Proceedings of 2004 IEEE International Conference on Web Services, pp. 514–521, July 2004
11. Oinn, T., et al.: Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics **20**(17), 3045–3054 (2004)
12. Preeth, E.N., Mulerickal, F.J.P., Paul, B., Sastri, Y.: Evaluation of Docker containers based on hardware utilization. In: 2015 International Conference on Control Communication Computing India (ICCC), pp. 697–700, November 2015
13. Sweeney, K.M.D., Thain, D.: Efficient integration of containers into scientific workflows. In: Proceedings of the 9th Workshop on Scientific Cloud Computing, ScienceCloud 2018, New York, NY, USA pp. 7:1–7:6. ACM (2018)
14. Xavier, M.G., Neves, M.V., Rossi, F.D., Ferreto, T.C., Lange, T., De Rose, C.A.F.: Performance evaluation of container-based virtualization for high performance computing environments. In: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 233–240, February 2013
15. Zheng, C., Thain, D.: 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, VTDC 2015, New York, NY, USA, pp. 31–38. ACM (2015)