Containerization for HPC in the Cloud: Docker vs Singularity

A Comparative Performance Benchmark

Lorenz Gerber

Lorenz Gerber
VT 2018
Degree Project: Bachelor of Science in Computing Science, 15 hp
Supervisor: Marie Nordström, assoc. prof.
Examiner: Eddie Wadbro, asst. prof.
Bachelor of Science in Computing Science, 180 hp

Abstract

In recent years, ‘containerization’, also known as ‘light-weight virtualization’, has arguably been one of the dominating buzzwords, both in industry and academic research. While scalability has been the main driving force in industry, reproducibility is often stated as the main benefit of containers in scientific High Performance Computing (HPC). Docker is currently the most widely deployed framework, ideally suited for micro-service architecture. Singularity is a framework that aims to provide mobility of compute with a focus on bare-metal HPC cluster systems.

In earlier studies the performance of different light-weight virtualization frameworks when deployed on bare metal HPC systems has been evaluated and Singularity was mostly found to be superior to others in it’s application niche. However, performing HPC applications in the cloud has recently become a viable alternative to dedicated bare-metal HPC clusters. This results in running light-weight virtualization frameworks on virtual machines.

The aim of this study was to benchmark the performance of Singularity when run on virtual machines in the cloud compared to a setup using ’Docker’. Adhering to earlier studies, performance metrics for compute, I/O, memory, and network throughput/latency were established.

Nomenclature

Benchmark Tools

HPL	Compute benchmark used on High Performance Computing Clusters
IOzone	File and file system Input/output performance benchmark
LINPACK	Compute benchmark by solving linear algebra problems
netperf	network bandwith and latency benchmarking tool
STREAM	Memory bandwith benchmark

General

Bare-Metal	Physical hardware, in contrast to virtualized
cli	command line interface
Docker	Containerization Framework
ext3	journaled filesystem
ext4	journaled filesystem, succsessor to ext3
FLOPS	Floating point operations per second
HPC	High Performance Computing
IOPS	Input/output operations per second
LXC	Containerization Framework
MBps	Megabytes per second
MPI	Message Passing Interface, a common parallel computing framework defintion
MVAPICH	an implementation of the MPI definition
OS	operating system
overlay2	Docker storage driver
PaaS	Platform as a Service
Singularity	Containerization Framework
SquashFS	compressed read-only file system

VCPU	Virtual central processing unit, number of available cores to a virtual machine
VM	Virtual Machine

Contents

1 Introduction

Recent work from several areas of applied computer science suggests that the convergence of Cloud HPC and containerization is an on-going development [15, 22, 24, 25]. Several reasons for this development have been suggested and bioinformatics is often stated as a prime example: Containerization is since some time an interest in scientific computing to increase the reproducibility of complicated computational workflows as found in genomics [5, 21]. Further, genome sequencing applications have been developed at universities and research institutes where dedicated HPC clusters were available. However, when genome sequencing became more and more a standard medical analysis technique, a large number of companies started to offer it as a commercial service. Suddenly, the need for transferring typical HPC cluster compute worfklows to public cloud platforms arose [15, 25].

1.1 Moving High Performance Computing to the Cloud

Conventional HPC is today mostly synonymous for large linux cluster system. Such systems superseded in the 90's mainframe and traditional supercomputers. Being constituted by a large number of networked commodity machines run with open-source operating system, these systems quickly became the platform of choice for universities and research institutes. Today most scientific applications are optimized to run in such environments. While such systems are affordable for universities, they require a considerable overhead in highly trained staff to manage them. Access to such systems is shared and usually restricted to researchers of the institute.

When platform-as-a-service (PaaS) become popular, scientific research quickly became interested and started to experiment with cluster setups within the cloud [15].

1.1.1 Dedicated HPC Cluster vs Cloud HPC

Both PaaS server farms and dedicated HPC clusters are run on 'cheap' commodity hardware. However there the similarities usually end. A dedicated HPC cluster is a highly performance tuned setup, with specialized network designs for highest throughput and minimal latency between all members of the cluster. More relevant, dedicated HPC clusters run the operating system on 'bare-metal' hardware. Often the operating system is customized to make full use of the available hardware. PaaS server farms however, usually run hypervisors as host for various types of guest virtual machines.

The overhead of virtual machines compared to bare-metal is of course a serious performance disadvantage. A major challenge and aim of research into cloud HPC is to design systems that can take advantage of the almost unlimited availability of nodes in PaaS server farms. Adding additional identical nodes to improve performance is the typical definition of scale-

out scalability. The challenge in scaling-out for HPC in the cloud is that the architecture of most HPC applications is based on clusters where all potentially available nodes are known ahead of execution.

The most obvious solution is to use cloud configuration and orchestration software to spin up ad-hoc clusters that work much in the same way as dedicated HPC clusters. Another approach is to redesign HPC applications for more flexible execution environments that can scale-out at run-time. According to published work and offered commercial services, both approaches seem to be viable, possibly depending on the type of application. It should be pointed out here, that when considering various cloud HPC solutions, the cost factor becomes an important factor.

However, it seems a trend that newly developed cloud HPC applications do not adhere to the traditional HPC architecture anymore and rather try to make use of the services, such as autoscaling available in PaaS. Another reason for this is also that in many scientific computing fields, such as bioinformatics, the reproducibilty of large complex and heterogeneous compute workflows has become an issue that increased the use of containerization, so called light-weight virtualization.

1.2 Lightweight Virtualization

Lightweight virtualization or containerization has been experimented with for a long time [13]. As shown in Figure 1, the main aim of containerization is, similar as with full system virtualization, the isolation of multiple systems or applications that run on the same physical machine. Containerization however, does only virtualize the operating system while full virtualization regards a whole machine. Practically this means that containers run all on the same kernel while virtual machines include their own kernels. A consequence of this is that container frameworks and their containers are bound to one hardware architecture and operating system. Most of the currently used container frameworks such as Docker, Singularity or LXC are based on the Linux kernel.

In a nutshell, the main advantage of containerization compared to full virtualization is the significantly smaller process overhead during run-time and the required time to deploy new instances on a running host. Especially the latter is a major concern for auto-scaling of large commercial web services.

Regarding Figure 1 it should be noted that ‘Container Engine’ is a typical term from the Docker framework. For the end-user, the other framework of interest here, Singularity, can be understood in the same way for usage, however internally, there are some significant differences. Some details will be discussed in the sections dedicated to the respective container framework.

1.2.1 General Work flow of Containerization

The workflow that a user will apply is mostly identical between Docker and Singularity. Moreover, the developers of Singularity have included a certain compatibility layer to the Docker eco-system.

Figure 2 shows a basic containerization workflow and the related operations. Specific con-

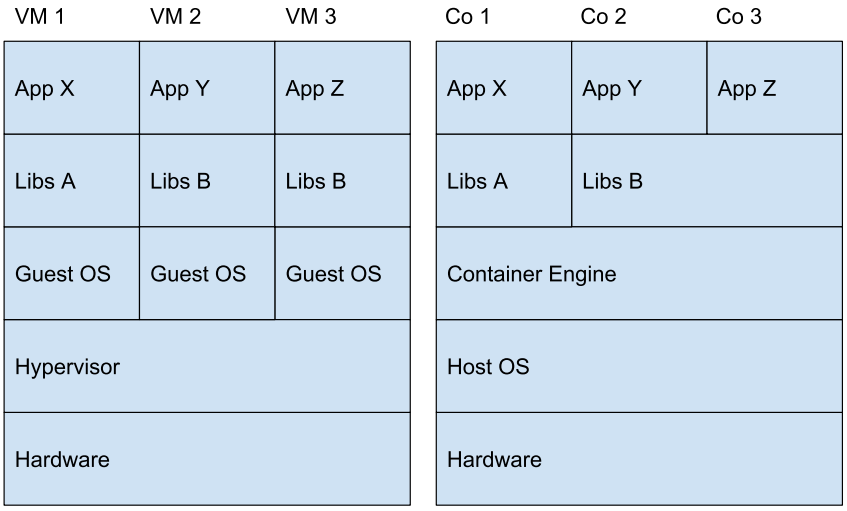

Figure 1: *The left block scheme shows full virtualization, in this case with a cloud typical type 1 'hypervisor' that runs directly on the hardware. The hypervisor coordinates the Guest OS with each their stack of libraries to run the respective applications. In the right block scheme, a host OS (typically Linux) runs on the hardware. The container engine then coordinates the environments including the libraries for each application running in separate containers. Figure reproduced from [14].*

figuration files, here called RECIPE, are used to build the image which contains all libraries and application binaries to be run as a container. The configuration files usually start out by defining a base image of some Linux distribution that shall be used. Then additional sources can be loaded, configured, and compiled during the build process. Both Docker and Singularity offer then a Registry Service where IMAGES can be stored or obtained. Invoking the 'run' command on a container starts a new CONTAINER. Multiple CONTAINER instances can be started from the same IMAGE. Generally, a CONTAINER will be running until all operations defined in the IMAGE are finished. CONTAINERS can also run interactive, for example with a shell on the console. The 'exec' command allows to invoke operations within a running container.

1.2.2 Docker

While various light-weight virtualization tools have been developed and experimented with for a long time, Docker was the one that after it's inception 2013 quickly become the synonym for containerization [13]. Docker uses the Linux Kernel feaures 'cgroups' for resource management and 'namespaces' for isolation of the containers' running processes. The recipe build instructions for Docker images are called 'Dockerfiles'. Most such build instructions start from a 'base image' that, as the name suggests, includes the basic libraries and binaries of a specific Linux distribution. Such 'official' base images are published by the respective organizations, such as 'Canonical' (Ubuntu) or 'Debian' through the public Docker registry 'Docker Hub'.

The main Docker architecture is composed of a persistent process, `dockerd` (Docker daemon) that manages the containers. It can be accessed through a REST API by the Docker client, using a command line interface. An often cited and important detail of the Docker

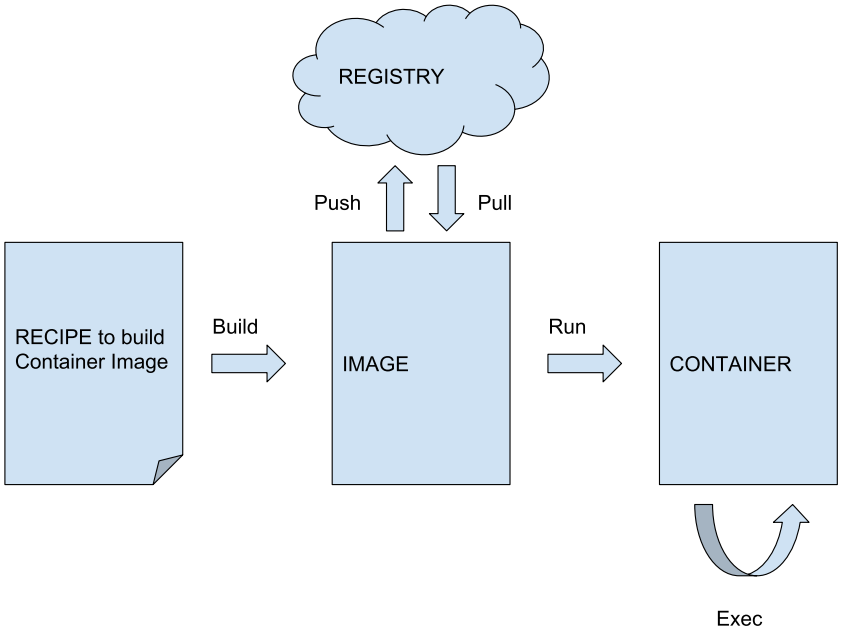

Figure 2: *In a generic container workflow, a RECIPE file is used to build an IMAGE. These can be pushed-to or pulled-from a REGISTRY which could be local or also public in the cloud. By invoking the 'run' command on an IMAGE, a CONTAINER instance is started. Multiple CONTAINER instances of the same image can be started. The 'exec' command allows to execute any command of choice within a running container.*

daemon is that it runs with root privilieges. This is said to be the main reason why Docker was hardly ever deployed on dedicated HPC cluster systems [10].

Docker has also become the base for further developments: While the framework itself offers some support for multi-container orchestration (Docker Swarm), 'Kubernetes' a framework developed on top of Docker by Google has during the last years become the de-facto standard for container orchestration at larger scale [2].

1.2.3 Singularity

Singularity was developed with the aim to increase 'mobility of compute' in scientific applications [10]. The architecture of Singularity is such that common users can safely run their containers on a HPC cluster system without the possibility for root privilege escalation. Internally, Singularity uses a simpler approach than Docker without daemon process. Images are executed with user privileges using the UNIX SUID functionality [20]. Singularity, makes use of Linux kernel 'namespace' feature to isolate processes. However, it does not implement 'cgroups' or any other resource management functionality. As mentioned earlier, Singularity allows using Docker images both directly for execution or also as base image in Singularity build instructions ('Singularity Recipe'). Similar to Docker Hub, a public registry for Singularity Images exists [19].

1.3 The Benchmarking Study

For obvious reasons, in HPC performance is the most relevant criteria. Hence, since the beginning of cloud HPC, there have always been a large number of research groups concerned with evaluating various setups for performance. This can be done either by benchmarking actual applications of interest by running them on the various infrastructures, or the more synthetic and theoretical approach where standardized benchmark tools are used to obtain more widely comparable metrics. Arango et al. have recently published benchmarking results based on such standardized tools [1]. Their interest was to compare various containerization frameworks run on a dedicated HPC cluster. Such data is highly relevant, because of the urgent need for improved reproducibility in scientific computing which can be obtained by containerization. In their work, they benchmarked LXC [4], Docker [6] and Singularity [8] against a bare-metal reference. In many cases, their findings showed a performance advantage of Singularity over the other two frameworks. In some cases Singularity and in a few cases also LXC outperformed even the bare-metal configuration. Arango et al. [1] concluded, based on performance but also general handling considerations that Singularity is ideally suited for developing containerized HPC applications. LXC, often outperformed Docker. But in comparison to Singularity and Docker, it is rather implemented as a software framework and not a tool to be used interactively. Hence Arango et al. did not consider it a viable alternative [1].

The obvious question after reading the mentioned work was whether similar results would be obtained when running the same benchmark using nested virtualization as it is common in cloud HPC: Container run in virtual machines. In a literature search, no similar benchmark data on running container frameworks in virtual machines could be found, which was the starting point for the present work.

2 Methods

2.1 Experimental Setup

All experiments were run on the Microsoft Azure platform, which uses the Azure Hypervisor, a customized version of 'Hyper-V' from Microsoft.

Virtual machines were spawned from an Azure provided Ubuntu 16.04LTS image. The chosen VM type was DS1_v2 (General Purpose, 1 VCPU, 3.5 GB RAM, SSDs with estimated performance of 120 IOPS and max 25 MBps transfer on the OS disc and 5000 IOPS/200 MBps on the data disc. For all except the network benchmark, only one single VM was needed at the time. To guarantee that the results would be comparable, Docker and Singularity were installed in the same VM. Deployment of the VM was automated with a shell script that makes use of the 'azure CLI' extension. Configuration of virtual machines during deployment was automated using 'cloud-init' configuration files [3]. Deployment and configuration scripts can be found in Appendices A and B.

Singularity was built and installed from source code according to descriptions provided in the Singularity user guide [18].

2.2 Container on VM

All containers were built from base images of the same Linux distro and version as the virtual machine, Ubuntu 16.04LTS. When nothing else mentioned, Docker images were also used to build and run Singularity containers. For this, either Docker images were used directly through the command line 'docker://dockerimage' syntax or Docker images were used as base in Singularity recipe files. In either case, Docker images where pushed to Docker Hub (`https://hub.docker.com`) from where they could be accessed.

2.3 Benchmarking Tools

The selection of benchmark tools was adapted from Arango et al. [1]. However, neither the 'hello-world' nor 'GPU' benchmark were considered relevant, hence they were excluded. For network performance, the 'Hewlett-Packard netperf' benchmark was used instead of the Ohio State University MVAPICH Micro-benchmarks.

2.3.1 CPU Performance

Compute performance was measured using a LINPACK benchmark [7]. As proposed by Arango et al. [1] the single core version linpackc.new was used. It was compiled using gcc

5.4.0 in Ubuntu 16.04LTS which was chosen both as VM host in Azure and as base image for Container. Compiler optimization level 3 was chosen. The LINPACK benchmark first generates a dense matrix system filled with random values from -1 to 1 that is then solved by LU decomposition, using routines DGEFA and DGESL. The LU decomposition provides a metric for floating-point instructions per second (FLOPS). This particular version allows choosing the array size which in this case was set to 200×200. The benchmark was setup to run for at least 10 CPU seconds.

The Dockerfile contained instructions to download and compile sourcecode for the LINPACK benchmark during the container build process. After building, the image was run to execute the benchmark and pipe the benchmark results into a text file.

2.3.2 Disk I/O performance

Disk I/O performance was estimated using the 'Iozone Filesystem Benchmark' (`www.iozone.org`). Here, Docker, respectively Singularity images were built from Docker Ubuntu 16.04LT base images. The 'IOzone' source code was downloaded and the tool compiled during the image build process. Then scripts to run 10 replicated experiments with either binding to the local file system or using the container internal file system were used to run the actual benchmark. For the VM reference benchmark, 'IOzone' source code was downloaded into an Ubuntu 16.04LT based Azure VM, compiled and run 10 times accordingly. From the available metrics the four most basic were chosen for comparison: 'read', 'write', 'random read', and 'random write', all measured as data throughput per time unit.

2.3.3 Memory Performance

The 'STREAM' benchmark from McCalpin was used to measure memory performance [12, 11]. A Dockerfile, respectively Singularity recipe was built using a Docker Ubuntu 16.04LT base image. The source of the STREAM tool was downloaded and compiled during the image build process. In contrast to Arango et al. [1], the tool was compiled without OpenMP flag, as the the virtual machine used only had one core. Then, console script files were used to batch run 10 replicate invocation of the STREAM benchmark program which consists of four sub-tests [12]:

- Copy - measures transfer rates in the absence of arithmetic.
- Scale - adds a simple arithmetic operation.
- Sum - adds a third operand to allow multiple load/store ports on vector machines to be tested.
- Triad - allows chained/overlapped/fused multiply/add operations.

2.3.4 Network bandwith and latency performance

To measure network performance Arango et al. [1] used the benchmark MVAPICH OSU Micro-Benchmarks suite. A tool used in MPI cluster network benchmarking. To reduce the number of non-related network performance related effects, it was decided use a benchmark tool that did not require setting up an MPI cluster in the cloud. Here, the 'netperf' benchmarking tool was used [9].

Hewlet-Packard ‘netperf’

For the ‘netperf’ test, two Docker images were built, both from Ubuntu 16.04LT base image and with instructions to download and compile netperf source code. One image was configured to invoke on container instantiation ‘netserver’, the remote daemon to communicate with the local ‘netperf’ program.

To run the actual benchmark, two virtual machines were started using the deployment script with the cloud-init configuration description (see appendix A and B). These scripts are Azure Cloud specific and automatically deploy two machines into the same sub net.

For the virtual machine reference benchmark, ‘netserver’ service was started on ‘vm 1’ and ‘netperf’ on ‘vm 2’, indicating the ip address of ‘vm 1’ as -H option. Below the used command line instruction to start ‘netperf’ is shown followed by explanation to the chosen options:

```
netperf -l 10 -i 10 -I 95,1 -c -j -H netperfClient -t \
    OMNI -- -D -T tcp -O THROUGHPUT,THROUGHPUT_UNITS, \
    THROUGHPUT_CONFID,MEAN_LATENCY,MAX_LATENCY,MIN_LATENCY, \
    STDDEV_LATENCY,LOCAL_CPU_UTIL,LOCAL_CPU_CONFID
```

- `-l 10` run each test for at least 10 seconds
- `-i 10` number of iterations for calculating the confidence interval
- `-I 95,1` Confidence level and confidence interval to be used throughout the test: 95% confidence level and 1% (+-0.5%) interval
- `-c` calculate CPU utilization
- `-j` calculate additional timing information (needed for min/max latency)
- `-H` address of host running netserver
- `-t OMNI` name of test
- `--` separator between global and test specific options
- `-D` set TCP_NO_DELAY
- `-T tcp` explicitly set the protocol
- `-O` which metrics to output
- `THROUGHPUT,THROUGPUT_UNITS,THROUGHPUT_CONFID` bandwith with unit and confidence interval
- `MEAN_LATENCY,MAX_LATENCY,MIN_LATENCY,STDDEV_LATENCY` latency metrics in microseconds
- `LOCAL_CPU_UTIL,LOCAL_CPU_CONFID` percentage of local (netperf) cpu utilizaton with confidence interval

With Docker, the benchmark was run both using the host network and when creating a virtual network that connects two Docker daemons. To create such a network, Docker was used in 'Swarm mode' which allows to coordinate multiple Docker daemons by defining a master slave hierarchy. For the actual network, Docker 'overlay' network driver was setup according to instructions (`https://docs.docker.com/network/network-tutorial-overlay/`). In brief, the 'Swarm' was created by calling 'swarm init' on the master machine and 'swarm join' from the client. Then an overlay network was created on the master machine. Now a Docker container attaching to the freshly created overlay network was started into an interactive shell on the master . Then the container running 'netserver', and the 'netperf' daemon was started in the client, also attaching to the 'overlay' network. This container was started as dameon using the Docker `run` '-d' flag. Now 'netperf' could be run within the 'overlay' network, indicating the host name of the Docker container running the 'netserver' daemon.

For the Singularity benchmark, the above mentioned Docker images were pushed to the Docker hub (`https://hub.docker.com`), so that they could be used directly with a Singularity exec command and the 'docker://dockerImage' syntax. Singularity by default uses the host network. The Docker host network experiment was run accordingly (using 'docker run' with 'netperf' as application to execute).

3 Results

All benchmark tools except ‘netperf’ were batch invoked multiple times to obtain statistics. These results were imported to R language where box plots were generated [17]. Each value in the box plot represents one benchmark tool invocation. The ‘netperf’ tool had run-times of over two minutes and calculated statistics during the run. It did not allow to extract the single data points. Hence, to underline the difference in the data preparation procedure, it was decided to present the results of ‘netperf’ in a table.

3.1 LINPACK Compute Benchmark Results

Figure 3 shows the results of running LINPACK on a 200×200 random matrix. Experiments were run on a virtual machine as reference and using nested virtualization with the container frameworks Docker or Singularity within a virtual machine.

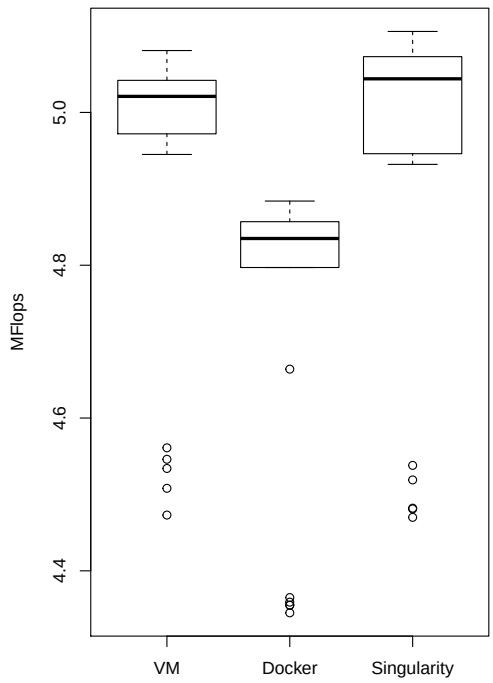

Figure 3: *The LINPACK benchmark was used to measure single core compute performance. For each setup, 20 replicate runs were performed.*

3.2 IOzone Disk and Filesystem IO Benchmark Results

The results of the iozone benchmark are shown in Figure 4 for ‘Read/Write’ and in Figure 5 for ‘Random Read/Write’. For both container frameworks, the benchmark was run once with binding to the virtual machines file system and once when running within the container image. For the latter, default settings of the respective container framework were used: ‘overlay2’ storage driver with ‘ext4’ backing file system for Docker and ‘ext3’ for Singularity. In fact, ‘SquashFS’ is the default file system for Singularity, however read-only. Singularity containers need explicitly to be created as writable with ‘ext3’.

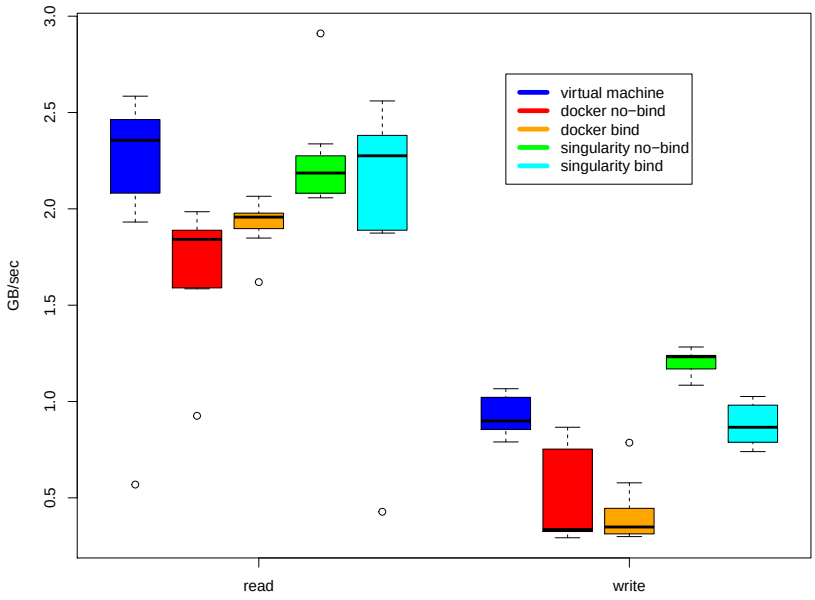

Figure 4: *The ‘IOzone’ results for sequential ‘read’ and ‘write’ operations. Here, ‘write’ creates a new file. Experiments were run using the container internal file system (no-bind) and by attaching the container to the external file system (bind). Write operations are in general slower as they include creation of meta data.*

‘Read’ measures the throughput (MBps) for sequential reading of a file. ‘Write’ measures the sequential writing of a new file to the file system. This includes creation of all meta data. ‘Random read/write’ benchmarks the respective operations on an existing file at random positions.

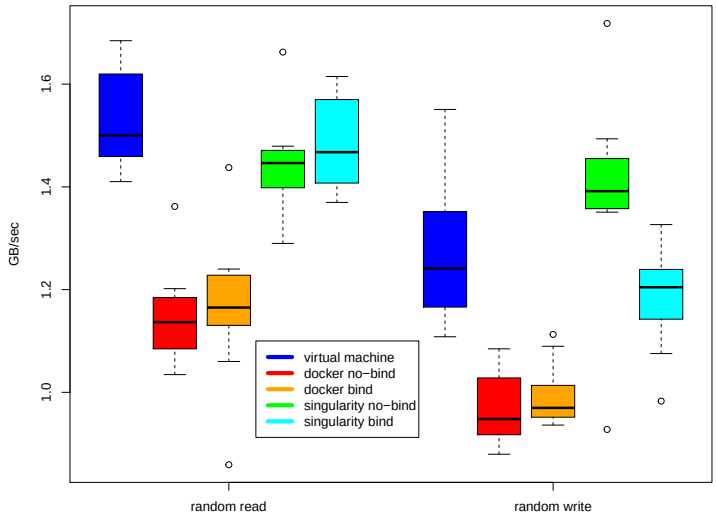

Figure 5: *The 'IOzone' results for random 'read' and 'write' operations on existing files. Also here, experiments with using the containers' internal file system (no-bind) and the external file-system (bind) were run.*

3.3 STREAM Memory Bandwidth Benchmark Results

Figure 6 shows the results from running the 'STREAM' memory bandwidth benchmark. Each individual setup was invoked ten times for statistical replication. In contrast to Arango et al. [1], the stream tool was compiled for single core setup without using the OpenMP flag.

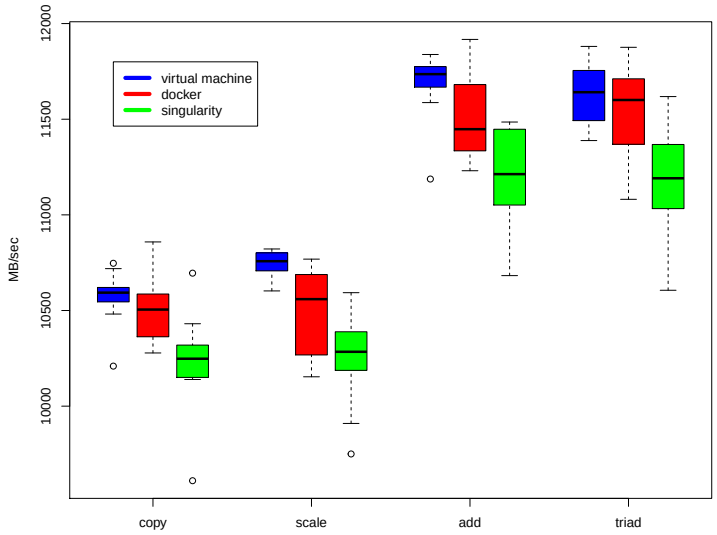

Figure 6: *Results from the STREAM memory bandwidth benchmark. In accordance with the provisioned virtual machine type, the tool was compiled for single core use.*

3.4 'netperf' Network Bandwith and Latency Results

Table 1 shows the results for running the 'netperf' network benchmark tool in various configurations. 'Docker host' and 'Singularity host' setup's use the underlying network provided by the virtual machine setup in Azure cloud. In contrast, 'Docker overlay' uses a container internal network driver that creates a virtual network between various Docker daemons. All statistics shown in Table 1 were calculated directly within the 'netperf' tool. In the 'netperf' documentation is a note that absolute values for CPU usage should not be trusted when benchmarking on virtual machines. Here, as all experiments used the same underlying virtual machine setup, they were reported anyway as their relative values should give some hints about the individual experimental setups.

Table 1 *Results from network benchmarking using the 'netperf'. All statistics were calculated directly within the 'netperf' tool. The tool was configured to apply a 95% confidence level where relevant. According to the 'netperf' documentation, absolut CPU usage values can not be trusted when run on virtual machines.*

	VM	Docker host	Docker overlay	Singularity host
throughput (MBps)	674.82	674.77	600.23	674.87
throughput +/- 0.5%				
@ 95% conf,	0.04%	0.04%	0.15%	0.02%
latency mean (μs)	193.42	193.31	216.94	193.3
latency min (μs)	2	2	3	2
latency max (μs)	19'541	19'579	34'997	21'064
latency stddev (μs)	1460.71	1466.23	1994.29	1472.5
local CPU use (%)	2.71	3.04	9.62	12.11
local CPU +/- 0.5%				
@ 95% conf, width (%)	26.88	21.02	12.80	243.76

4 Discussion

Initially some decisions about the general setup of the benchmark study are discussed. Then each benchmark will be treated separate, in comparison to the findings of Arango et al. [1].

4.1 Cloud vs. Local Virtual Machine

During planning of the experiments both using virtual machines in a public cloud such as 'Microsoft Azure' or 'Amazon AWS' and running a virtual machine on a local physical machine was considered. The advantage of a virtual machines in the public cloud is the simplicity of deployment. Further, running the experiments on a public cloud platform is also highly relevant as running HPC computations in the public cloud is the ultimate goal to which this study shall contribute answers to a number of related questions. However, in benchmarking, full control of the benchmark environment is desirable to produce general valid and comparable results. Although maybe a rather rare event, virtual machine live migration triggered by the platform provider would definitively have an impact on the benchmark. An even more basic problem is that each start of an instance can result in slightly different underlying hardware, which will have an impact on the benchmarks. Here, it was decided to accept the uncertainties and variability connected to the use of a public cloud service because it seemed more relevant to the real application. Cloud platforms run exclusively with 'native' highly optimized hypervisors while in a local installation a generic 'hosted' hypervisor such as VirtualBox would have been used.

4.1.1 Minimizing External Effects

All experiments related to one type of benchmark were run on the same instance of virtual machine and in a time span as short as possible to minimize the risk of potential VM live migration events. Further, it was assumed that a consistent shift of all benchmark values in two replicates was the effect of obtaining different hardware from one VM instance deployment to the next.

4.2 Synthetic Benchmarks

4.2.1 LINPACK Compute Benchmark

Arango et al [1] used a HPC specific Benchmark: HPL [16] while they proposed to use LINPACK for single core, non distributed performance. In their HPL benchmark, Singularity outperformed the bare-metal run while Docker performed slightly worse than bare-metal. Figure 3 in Section 3.1 shows the results for the LINPACK benchmark which resulted in

similar ranking as for Arango et al. [1]: Singularity was on par with the virtual engine, however with a larger variability in the results. LINPACK on Docker was slower than the virtual machine run to a similar degree as Docker on bare-metal vs bare-metal reported by Arango et al [1]. The variability of the results was similar for Docker and the virtual machine runs. It seems that for pure compute performance, without including GPU specific benchmarks, there is little difference between the container frameworks run on a virtual machine. Hence, the cloud HPC possibility of scaling up by choosing more powerful virtual machine instances is also a viable solution when running container frameworks. The slightly lower compute performance of Docker, could be based on the larger overhead for running the Docker daemon.

4.2.2 IOzone Disk and File-System IO Benchmark

For the disk and file system IO benchmark, Docker performed significantly worse than the bare virtual machine and Singularity. This was true both for 'bind' and 'no-bind' as well as 'read' and 'write'. Comparing the two Docker experiments between each other, binding to the local file system performed always at least as good as when running inside the container. Often the 'bind' setup outperformed the 'no-bind'. Using Docker with 'bind' resulted always in less variation between replicate runs. Singularity without 'bind' performed better than the virtual machine alone for 'writes' and similar when binding to the local disk. For 'reads', Singularity was almost on par with the virtual machine, but faster than Docker for both 'bind' and 'no-bind'. Arango et al [1]. used the same setup except that they also benchmarked 'no-bind' and 'bind' for bare-metal which represented either local disc or a network file system (NFS) mount. Then, according to their description the 'bind' mount for the container frameworks was also an NFS mount. That does not seem the most relevant comparison as one would like to know the difference between container file system which is 'nested' on the local disc compared to the local disc files system. Therefore, here the virtual machine was only benchmarked in respect to the local disc. The approximate magnitude in difference between 'write' and 'read' performance was the same for the current study as for Arango et al [1].

For the random read and write benchmarks, an almost identical result to Arango et al. [1] was found. Also with nested virtualization, Singularity outperformed the bare virtual machine for no-bind random writes. When binding to the local file system, performance with Singularity was in accordance with the bare virtual machine which suggests that there is hardly any management overhead involved. This is however not the case for Docker: both 'bind' and 'no-bind' configuration, 'random read' or '-write' is significantly slower than comparable bare virtual machines runs. For all tested setups, the difference between 'random read' and 'random write' is as expected generally smaller than for sequential 'read' and 'write'.

After performing all benchmarks, it was realized that they were run on the 'OS' disc (30 GB, ext4). According to the Azure console, this disc performs with about 120 IOPS and 25 MBps rather in the range of a traditional HDD while the data disc is indicated with SSD conform values (5000 IOPS/200 MBps). When this was found, for a quick comparison, 'IOzone' was run once on the 'HDD like' OS disc and once on the 'SSD like' mounted data disc (1023 GB, ext4). Surprisingly, there was no difference found in performance. This should however be investigated further as it does not really confirm the indicated performance in the azure portal.

In respect to the found performance values using containerization, it should be mentioned that Docker specifically but also Singularity offers a range of different storage drivers. For the non-bind setup, the storage driver is the key determining performance factor and provides a large range for optimization. Here the default driver 'overlay2', a union file system was used in Docker. Depending on the applied containerization strategy, either non-bind or bind performance is more important. When single steps of a computational workflow run in their own individual container and data is exchanged externally, bind performance is important. This seems to be a popular strategy with Docker containers while with Singularity often a whole workflow with several steps is hosted in the same container. For this case either no-bind or bind setup can be used. When IO performance becomes the bottleneck in cloud HPC and needs to be improved upon, parallel file systems, common on dedicated HPC clusters can also be deployed. As example on the Azure platform, support for Lustre, GlusterFS, or BeeGFS is provided.

Generally, it was found that the present simple benchmark does not exhaustively assesses the IO performance of nested containerization systems, but it can serve as a good starting point for further optimization. Generally, it seems more straight forward to obtain good IO performance with Singularity.

4.2.3 STREAM Memory Bandwidth Benchmark

In accordance with Arango et al. [1] neither Docker nor Singularity reached the same performance as running in the bare virtual machine. However, the ranking for the container frameworks was just the opposite to what Arange et al. found: Here, Docker outperformed Singularity clearly. STREAM measures sustainable memory bandwith using vector calculations with vector sizes larger than cache memory on most systems. Together with LINPACK it should be relevant for many real-world applications that are based on matrix calculations. Essentially, the STREAM benchmark can be understood as a correction to pure compute based benchmarks, such as LINPACK for the effect of the 'von Neumann bottleneck'. Memory performance however is probably the parameter with the least possibility for improvement by a container user. A similar study, that evaluated full virtualization systems (KVM, VMware), much larger differences were found for different hypervisors [23]. While VMware performed on par with bare metal, KVM suffered a much larger relative overhead when compared to the overhead of container framework versus virtual machine. This specific observation should remind, that potentially also the combination of hypervisor and container system could have an influence on performance, potentially not only in the STREAM benchmark.

4.2.4 'netperf' Network Bandwidth and Latency

As mentioned earlier, the network performance was here measured with 'netperf' instead of the 'OSU MVAPICH Microbenchmark'. Arango et al. [1] set up Docker in the swarm mode using the 'overlay' network driver. Additional to this, Docker was in this case also benchmarked using 'host' mode to connect directly to the host network. This setup seems a more relevant comparison to Singularity which also connects directly to the host network. In the current study, bandwidth and latency measurements for virtual machine, 'Docker host' and 'Singularity host' yielded almost identical results. The only difference found was that measured CPU usage for Singularity fluctuated heavily between re-runs of the

whole benchmark. Also the measured CPU usage values for Singularity was significantly higher than those measured for the virtual machine and 'Docker host' benchmark runs. It should be noted that the 'netperf' documentation warns for using the CPU measurements when running on a virtual machine. Here, the values were reported anyway, as the virtual machine setup did not change from benchmark to benchmark. Therefore, the metric should at least contain some relative information.

The Docker overlay network setup performed worse compared to all other measurements. The reduction in throughput from about 670 MBps to 600 MBps and an increase in latency by 1 micro second is not dramatic. However Docker showed in overlay mode a consistent increase in CPU usage.

Arango et al.'s MVAPICH measurements show a decrease in bandwidth for Singularity and even more for Docker in 'overlay mode'. This seems reasonable compared to running on native hardware [1]. However, there seems to be a mistake in Figure 9 of Arango et al. [1] which suggests that Docker and even more Singularity achieves lower latency values than bare metal while they state the opposite in the text.

4.3 Benchmark Summary

Most of the obtained results where similar to those from Arango et al. [1]. A general difference to Arango et al's study was however that container frameworks seldom outperformed the reference values which in this case were virtual machine- respectively bare-metal-measurements for Arango et al. [1]. From all the benchmarked disciplines, it seems that disk/file-system IO and networking have the best possibilities for improvement as they offer a considerable range of configuration possibilities within the frameworks.

5 Conclusions

Looking at all the benchmarks and summarizing which framework scored more often as winner, 'Signularity' is the clear winner and would need to be recommended as containerization framework also for cloud HPC. This is fully in-line with the recommendations of Arango et al. [1]. However, when taking a closer look at their actual justifications, many reasons apply almost exclusively to typical dedicated HPC cluster where the safety aspect is of large importance. Arango et al. [1] reason for example that Docker network performance could not be improved as running it in host mode would jeopardize user privilege related precautions. This is however mostly irrelevant for cloud HPC as the whole system is deployed by the user himself. User privilege concerns in cloud HPC are handled at a much higher level through the user access management system of the cloud provider.

Still, the absence of user privilege problems in cloud HPC compared to dedicated HPC does not automatically changes the results. While Singularity in many cases performed on par with the virtual machine, Docker usually underperformed, probably due to the larger overhead of running a more complicated container manager (Docker daemon) compared to a more slim container run-time of Singularity that more over dispenses with the overhead of resource management (cgroups).

Most likely, the answer to: "Docker or Singularity in Cloud HPC", is closer to: "it depends", than a simple choice based on synthetic benchmarks. Probably at least equal important to measurable synthetic performance based on compute, IO, memory and network is the general architecture of the HPC application. Traditional MPI HPC cluster setup seem seldom the chosen architecture when cloud HPC applications are designed freshly instead of transferred from existing HPC setups. Here, Docker profits from its ecosystem and the large number of contributors. Besides the Docker 'Swarm mode', there are several other frameworks, including 'Google Kubernetes', that can be used for large scale container orchestration which is seen by many as the architecture of choice for future cloud HPC systems. Singularity however does not offer any dedicated container orchestration tool at the moment.

If one or some of the benchmarked sub groups should be chosen for further optimization, the author would select disc I/O and network. During the present study, they were found to offer most parameters that potentially have a significant influence on the performance. These parameters specify on one side the container framework internals and on the other side they are available for more in-depth configuration of the PaaS platform. Hence, the latter could be specific to the provider of the PaaS.

References

[1] Carlos Arango, Rémy Dernat, and John Sanabria. Performance evaluation of container-based virtualization for high performance computing environments. *arXiv preprint arXiv:1709.10140*, 2017.

[2] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[3] Canonical. cloud-init: The standard for customizing cloud instances. `https://cloud-init.io/`, 2018. accessed: 2018-05-05.

[4] Canonical. Linux containers: Infrastructure for container projects. `https://linuxcontainers.org/`, 2018. accessed: 2018-05-22.

[5] Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. Nextflow enables reproducible computational workflows. *Nature biotechnology*, 35(4):316–319, 2017.

[6] Docker. Docker: Build, ship and run any app, anywhere. `https://docker.com/`, 2018. accessed: 2018-05-22.

[7] Jack J Dongarra, James R Bunch, Cleve B Moler, and Gilbert W Stewart. *LINPACK users' guide*. Siam, 1979.

[8] Kurtzer M. G. Singularity. `https://singularity.lbl.gov`, 2018. accessed: 2018-05-22.

[9] Hewlett-Packard. Netperf. `https://github.com/HewlettPackard/netperf`, 2018. accessed: 2018-05-22.

[10] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459, 2017.

[11] J. D. McCalpin. Stream: Sustainable memory bandwith in high performance computers. `https://github.com/HewlettPackard/netperf`, 2018. accessed: 2018-05-22.

[12] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[13] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[14] Adrian Mouat. *Using Docker: Developing and deploying software with containers*. O'Reilly Media, Inc., 2015.

[15] Marco AS Netto, Rodrigo N Calheiros, Eduardo R Rodrigues, Renato LF Cunha, and Rajkumar Buyya. Hpc cloud for scientific and business applications: Taxonomy, vision, and research challenges. *ACM Computing Surveys (CSUR)*, 51(1):8, 2018.

[16] Antoine Petitet, R Clint Whaley, Jack J Dongarra, and Andy Cleary. Hpl-a portable implementation of the high-performance linpack benchmark for distributed-memory computers. innovative computing laboratory, september 2000. `http://www.netlib.org/benchmark/hpl/`. accessed: 2018-05-22.

[17] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2018.

[18] Singularityware. Singularity user guide. `https://singularity.lbl.gov/user-guide`, 2018. accessed: 2018-05-05.

[19] Vanessa V Sochat, Cameron J Prybol, and Gregory M Kurtzer. Enhancing reproducibility in scientific computing: Metrics and registry for singularity containers. *PloS one*, 12(11):e0188511, 2017.

[20] W Richard Stevens and Stephen A Rago. *Advanced programming in the UNIX environment*. Addison-Wesley, 2013.

[21] John Vivian, Arjun Arkal Rao, Frank Austin Nothaft, Christopher Ketchum, Joel Armstrong, Adam Novak, Jacob Pfeil, Jake Narkizian, Alden D Deran, Audrey Musselman-Brown, et al. Toil enables reproducible, open source, big biomedical data analyses. *Nature biotechnology*, 35(4):314, 2017.

[22] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240. IEEE, 2013.

[23] Naveed Yaqub. Comparison of virtualization performance: Vmware and kvm. Master's thesis, Oslo University College, 2012.

[24] Andrew J Younge, Kevin Pedretti, Ryan E Grant, and Ron Brightwell. A tale of two systems: Using containers to deploy hpc applications on supercomputers and clouds. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 74–81. IEEE, 2017.

[25] Yong Zhao, Xubo Fei, Ioan Raicu, and Shiyong Lu. Opportunities and challenges in running scientific workflows on the cloud. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on*, pages 455–462. IEEE, 2011.

A Azure VM Deployment Script

```
#!/bin/bash

# This script needs Microsoft Azure Cli installed. The sed -i
# commands are formatted to work on default OSX installed sed.

# Positional cli args:
# 1: Username
# 2: Hostname

cp cloud-init-form.txt cloud-init-temp.txt
PUBLIC_KEY=`cat ~/.ssh/id_rsa.pub`

# set public key
sed 's%PUBLIC_KEY%'"$PUBLIC_KEY"'%' \
    <cloud-init-form.txt >cloud-init-temp.txt

# set user and hostname from pos args
sed -i '' 's/USERNAME/'"${1?"Username missing. \
    Usage: deploy_vim USERNAME HOSTNAME"}"'/g' cloud-init-temp.txt
sed -i '' 's/HOSTNAME/'"${2?"Hostname missing. \
    Usage: deploy_vim USERNAME HOSTNAME"}"'/g' cloud-init-temp.txt


# Create resource group if it does not exist yet
if [ `az group exists -n containerBenchGroup` = false ]
then
    echo "Creating resource group..."
    az group create --name containerBenchGroup --location eastus
    echo "...done."
fi

# Launch VM into resource group
echo "Trying to deploy Virtual Machine..."
az vm create \
    --resource-group containerBenchGroup \
    --name "$2" \
    --image canonical:UbuntuServer:16.04-LTS:latest \
    --generate-ssh-keys \
    --custom-data cloud-init-temp.txt
```

```
echo "Removing temp files..."
rm ./cloud-init-temp.txt
echo "...done."

echo "NOTE: Compiling and installation of Singularity"
echo "will take some minutes from now and might not"
echo "be available directly after immediate login."
```

B Azure Cloud-init configuration

```
#cloud-config

hostname: HOSTNAME

users:
  - default
  - name: USERNAME
    groups: sudo
    shell: /bin/bash
    sudo: ['ALL=(ALL) NOPASSWD:ALL']
    ssh-authorized-keys:
      PUBLIC_KEY

runcmd:
  - 'sudo apt-get update'
  - 'sudo apt-get -y install apt-transport-https ca-certificates curl \
    software-properties-common'
  - 'curl -fsSL "https://download.docker.com/linux/ubuntu/gpg" |  \
    sudo apt-key add -'
  - 'sudo add-apt-repository "deb [arch=amd64] \
    https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"'
  - 'sudo apt-get update'
  - 'sudo apt-get -y install docker-ce'
  - 'sudo usermod -aG docker USERNAME'
  - 'sudo apt-get -y install python dh-autoreconf build-essential \
    libarchive-dev'
  - 'cd /home/USERNAME'
  - 'git clone https://github.com/singularityware/singularity.git'
  - 'cd singularity'
  - './autogen.sh'
  - './configure --prefix=/usr/local --sysconfdir=/etc'
  - 'make'
  - 'sudo make install'
  - 'cd /home/USERNAME && mkdir git && cd git'
  - 'git clone https://github.com/lorenzgerber/containers-benchs.git'
```