Socker: A Wrapper for Running Docker Containers on SLURM *

Abdulrahman Azab
University Center for
Information Technology
University of Oslo
Oslo, Norway
azab@usit.uio.no

ABSTRACT

Linux containers, with the build-once-run-anywhere approach, are becoming popular among scientific communities, such as life sciences, for software packaging and sharing. Docker is the most popular and user friendly platform for running and managing Linux containers. This is proven by the fact that vast majority of containerized tools are packaged as Docker images. A demanding functionality is to enable running Docker containers inside HPC job scripts for researchers to make use of the flexibility offered by containers in their real-life computational and data intensive jobs. The main two questions before implementing such functionality are: how to securely run Docker containers within cluster jobs? and how to limit the resource usage of a Docker job to the borders defined by the HPC queuing system? This position paper presents Socker, a wrapper for running Docker containers on SLURM. Socker enforces running containers within SLURM jobs as the submitting user, as well as enforcing the inclusion of containers in the cgroups assigned by SLURM to the parent jobs. The implementation of socker is currently under testing on Abel, the HPC cluster at the University of Oslo. Socker is proven to be secure and simple to use, while introducing no additional overhead.

Keywords

SLURM; Docker; Linux containers; HPC

1. INTRODUCTION

Sharing of software tools is an essential demand among scientists and researchers in order to reproduce results. Virtual machines (VMs) are widely adopted as a software packaging method for sharing collections of tools, e.g. BioLinux [1]. Each VM contains its own operating system, known as the guest OS, on the top of which software packages are installed. A VM monitor, also known as hypervisor, is the platform for managing and monitoring VMs. VM technology is suitable for packaging collections of tools that run independently or dependently on the top of a specific OS platform, e.g. a GUI that runs python and R tools on the top of Ubuntu Linux. In many cases, the need is to only have a single software tool installed, rather than a collection of tools. In this case, installing a whole VM with a guest OS to run just one tool is resource consuming in terms of CPU, memory, and disk space. Linux containerization is an operating system level virtualization technology that offers lightweight virtualization. An application that runs as a container has its own root file-system, but shares kernel with the host operating system. This has many advantages over virtual machines. First, containers are much less resource consuming since there no guest OS. Second, a container process is visible on the host operating system, which gives the opportunity to system administrators for monitoring and controlling the behavior of container processes. Linux containers are monitored and managed by a container engine which is responsible for initiating, managing, and allocating containers. Figure 1 presents structural comparison between VMs and Linux containers.

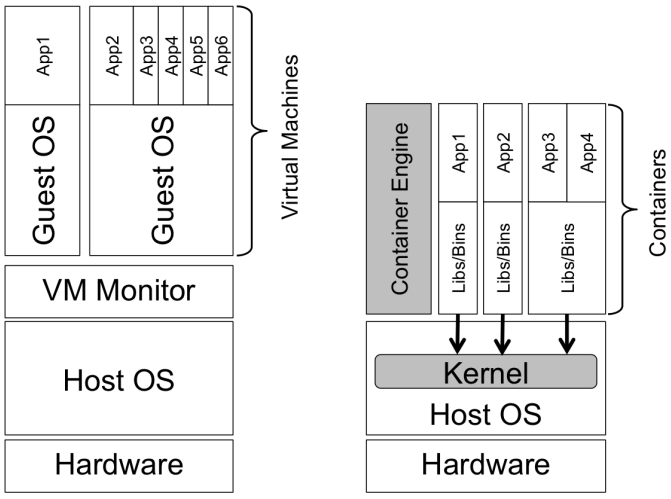

Figure 1: Virtual Machines vs Container

Docker [2] is the most popular platform among users and IT centers for Linux containerization. A software tool can be packaged as a Docker image and pushed to the Docker public repository, Docker hub, for sharing. A Docker image can

*SLURM: Simple Linux Utility for Resource Management.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

run as a container on any system that has a valid Linux kernel. There are other options in the field, e.g. Rocket [3] and LXD [4] for Linux, in addition to Drawbridge [5] for Windows. Despite the existence of different alternatives, Docker is still the most robust, user friendly, and well supported platform. To make the best use of containers, it has become demanding to enable containers on HPC platforms. HPC is widely used by researchers in different fields, e.g. physics and bioinformatics, to speedup applications that may take months or even years on one PC. Enabling containers on HPC platforms is beneficial for system administrators, removing the burden of software installation and maintenance hell, since a container that runs on a Linux system should run on any other Linux system. This is also beneficial for users, speeding up the process of software deployment on the HPC system since system administrators would need to exert minimal effort to deploy a containerized software. Our main HPC cluster at the University of Oslo, Abel [6], is based on SLURM [7]. Before deploying Docker containers on Abel in production, there are issues that need to be resolved. First, a Docker container process must run as the submitting user to be bounded with the user privileges and to know who is running what, while the default configuration of Docker is to run containers as root. Second, the resource consumption (CPU and memory) of a Docker container running in a SLURM job must be bounded with the limitations set by SLURM to this particular job. This position paper introduces *socker*, a wrapper for running Docker containers on SLURM. Socker is developed by the author and is under testing by the research computing group at the University Center for Information Technology, USIT. Socker is not replacing the Docker engine for running and managing containers. It enforces running Docker containers as the submitting user in addition to enforcing the membership of a running container inside a SLURM job in the cgroups assigned by SLURM to the job. Primary testing has been performed as a proof of concept, and socker seem to introduce almost no additional overhead. Further developments of Socker are ongoing.

2. RELATED WORK

There are numerous efforts in the direction of enabling Docker containers on HPC platforms. Docker Swarm [8] is the HPC platform offered by Docker. Swarm offers great scalability, but it is very poor in terms of security. In addition, the scheduling and resource management is too basic which makes it insufficient for large HPC systems in production. Google's Kubernetes [9] is a competitor to Docker Swarm. It offers more control over running containers, in addition to self-healing, but it is still poor in terms of scheduling and resource management. In addition, it is very complex to install and maintain. A common issue for both Docker Swarm and Kubernetes is that they are both designed to be standalone queuing systems, not runners or wrappers. So, one cannot have Swarm or Kubernetes plugged into already installed queuing system, e.g. TORQUE or SLURM, but has to replace the entire queuing system. HTCondor [10] has recently provided support for Docker through the *docker universe* [11]. HTCondor is more trustworthy for HPC systems since it is a well known and well tested platform for both HPC and HTC [1]. We have deployed Docker with HTCondor on a test environment, and the implementation was proven to be very user friendly. But it lacks flexibility, e.g. in terms of mounting volumes. In addition, it does not have any control over resource usage by the running containers. The most related effort is shifter [12] which is developed by the National Energy Research Scientific Computing Center (NERSC) and is deployed in production on a SLURM cluster of Cray supercomputers. Shifter is however not following the Docker standards and is not using the Docker engine for running and managing containers. It has its own image format to which both Docker images and VMs are converted. The Docker engine is replaced by *image manager* for managing the new formated images. Previously NERSC introduced MyDock [12] which is a wrapper for Docker that enforces accessing containers as the user. MyDock however did not provide a solution for enforcing the inclusion of a running container in the cgroups associated with the SLURM job. In addition, both shifter and myDock enforces accessing as the user by first running as root and mounting `/etc/passwd` and `/etc/group` in each single container, then lets the user access the container as him/herself. Socker adopts a more secure and less complex strategy by running a container as the user from the very beginning, avoiding any threats that may result in running as root. Shifter being immature, and relying on its own image format and own engine, is not attractive for us. It places the whole development and maintenance responsibility on the Shifter development team. Developing a runner that uses the Docker engine is more realistic since Docker is a well known and maintained platform in addition to being used by millions of users and a number of IT research support centers worldwide.

[1]High Throughput Computing

3. DOCKER

Docker is a new tool that automates the deployment of applications inside Linux containers. It provides a layer of abstraction and automation of operating-system level virtualization. Docker is not itself a novel technology, but is a high-level tool which is built on the top of LXC [13] Linux containers API, and provides additional functionality. Docker containers are executed and controlled by the Docker engine, which is different from the hypervisor for VMs. Since it does not include a full guest OS, a Docker container is smaller and faster to start up than a VM. A Docker container instead mounts a separate root filesystem, which contains the directory structure of the Unix-like OS plus all the configuration files, binaries and libraries required to run user applications. The boot file-system which contains the bootloader and kernel is not included in the container, but a container uses the host boot file-system [14]. When Docker mounts the container root file-system, it starts read-only, as in a traditional Linux boot, but then, instead of changing the file system to read-write mode, it uses union mount [15] to add a read-write file system over the read-only file system. It is also possible to have multiple read-only file systems stacked on top of each other. Each of those file-systems can be considered as a layer [16]. This structure is known as layered file-system, see Figure 2.

Since not all file-systems are layered, e.g. Ext, Docker builds its layered file-system on the top of the host file-system (known as *backing* filesystem in the Docker terminology). Docker supports different layered filesystem drivers, e.g. Overlay FS, AUFS, and BTRFS. A typical setup is

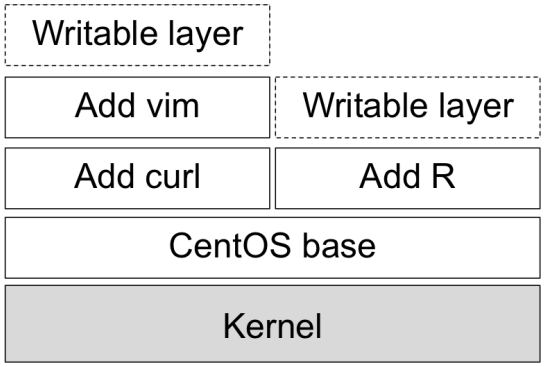

Figure 2: Docker Layered File-system

to have Docker with OverlayFS on the top of Ext4 backing filesystem. Based on the above description, one could start with e.g. a CentOS base image (which contains only the CentOS root file-system), and run it from the Docker engine on e.g. Ubuntu machine. Then install a piece of software, e.g. curl, on the top of this base image, and deploy this as a new image (containing both CentOS root file-system and curl installed). Using this new image as a base image, one could install another piece of software, e.g. vim, and make another image which has both curl and vim installed on CentOS, and so on.

Building own Docker image is performed in one of two scenarios, presented in Figure 3:

1. Interactive building which is carried out by starting a base image as a container (via `docker run`), running the commands to install the desired software on the running container, then committing the changes creating a new image on the local Docker repository.

2. Building from a Docker file which is carried out by writing the installation commands into a *Dockerfile*. Dockerfile is a script file which contains all the commands one would normally execute manually in order to build a Docker image, and is written in a Docker specific format. The Dockerfile starts by loading a base image, followed by the list of Docker formatted commands to install the desired software. The image is built by calling the docker build command having the Dockerfile as an argument. The building process will go step by step, executing the commands successively.

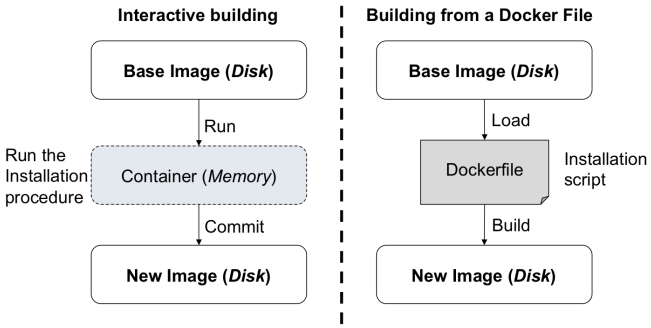

Figure 3: Building a Docker image

4. THE HPC INFRASTRUCTURE

4.1 Abel

Abel [6] is the high performance computing facility at the University of Oslo hosted by the University Center for IT and maintained by the Research Infrastructure Services group. Named after the famous Norwegian mathematician Niels Henrik Abel (1802 — 1829), the Linux cluster is a shared resource for research computing capable of 258 TFLOP/s theoretical peak performance. At the time of installation Abel reached position 96 on the Top500 list.

4.1.1 Hardware

Today, Abel's compute capacity is 702 nodes, 11392 cores, total memory of 40 TiB, and total local storage of 400 TiB using BeeGFS. Compute nodes are all dual Intel E5-2670 (Sandy Bridge) based running at 2.6 GHz, yielding 16 physical compute cores. Each node have 64 GiBytes of Samsung DDR3 memory operating at 1600 MHz, giving 4 GiB memory per physical core at about 58 GiB/s aggregated bandwidth using all physical cores. Compute nodes are connected to a large common scratch disk space. All nodes have FDR InfiniBand running x86_64 CentOS 6.7. The storage is provied as two equal size partitions */cluster* and */work* each capable of a performance of about 6-8 GiB/s when doing sequential IO. The file system is Fraunhofer Global Parallel File System (FhGFS)[2].

4.1.2 Software

Software provisioning on Abel is implemented using Environment Modules. There are currently 650+ installed modules on Abel, including: General software, Programming languages, Compilers, Debuggers, Libraries, and Others.

4.1.3 Queuing System

Abel uses the SLURM (Simple Linux Utility for Resource Management) workload manager [7] as the queuing system. SLURM has three main functions. First, allocating access to compute nodes users for a predefined time duration. Second, it provides a framework for starting, executing, and monitoring submitted jobs. Finally, it manages a queue of pending jobs. To ensure that running jobs won't consume more resources on the compute node(s) than what is assigned by the system, SLURM creates three main cgroups [17] for each running job (cpuset, memory, and freezer) then enforces inclusion of each process inside a running job in the three cgroups. If a process X in a running job consumes e.g. more memory than the value stored in the job's memory cgroup, it gets killed by SLURM.

4.2 Colossus

Colossus is compute cluster designed to be as similar to Abel. Colossus is the research computing within the Services for Sensitive Data (TSD) [18] which is a secure e-infrastructure for storing and processing sensitive data. Abel and Colossus have almost the same installed software modules.

5. MOTIVATION

[2]The hardware information is collected on Fri May 20 15:33:51 CEST 2016

Researchers from different fields of science develop software tools to perform various analysis. In many cases, a researcher doesn't have the proper experience and competence to develop a portable piece of software. This usually results in installation and maintenance problem when other researchers, or even system administrators, try to deploy this software on a different platform from the author's. In many cases, when a system administrator starts the process of deploying a software tool on a different platform, compilation and configuration problems pop up. Then the system administrator may spend many working hours trying to resolve the problems. In some cases, contacting the tool's author for inquiries is necessary. And some times, no response is received from the author because s/he was e.g. a PhD student and now already done with the PhD and is no longer maintaining the tool. This problem, even with this specific example, is well known in the field of reproducible science.

With the use of Linux containers and the very user friendly Docker engine, researchers can now install tools on their platform of choice, e.g. Ubuntu for bioinformaticians and CentOS for physicists, as a Docker image and publish it on the Docker hub or just share the Dockerfile with collaborators. Then anyone who has a Docker engine and a proper Linux kernel may run the image and get the same functionality. This makes life easier for software developers in that they no longer need to write multiple installation guides and test on different Linux distributions. It also makes life easier for system administrators, as instead of receiving software requests of type: "I need software X, and here is the installation guide, please install it!", requests will be of type: "I need software X, here is the name of its Docker image, please pull it". In addition, no software maintenance will be needed and no dependency conflicts will arise when installing new software.

On our Abel and Colossus clusters, we receive large number of software installation requests due to which we created a software request queue. In addition, we have a lot of already installed modules to maintain, see Section 4.1.2. Enabling Docker containers on Abel and Colossus would be very beneficial for both our users and system administrators.

6. SOCKER

Socker is a wrapper for running Docker containers inside SLURM jobs. It is mainly designed to enable the users of our HPC clusters (Abel and Colossus) to run Docker enabled jobs. Since our compute nodes are running CentOS 6.7, which standard kernel is 2.6, Socker is currently designed to work with Docker 1.7 (since support for CentOS 6 is dropped for newer Docker releases). Socker mainly provides two functions. First, run each container process as the user who submitted the job in order to make the container bounded by the user's capabilities. Second, bound the resource usage of any container, called inside a SLURM job, by the limits set by SLURM to the job. Socker is also designed to be simple and portable. Figure 4 shows the structure of Socker. Users are categorized into system administrators and regular users. System administrators are given privileges to run the *docker* command, i.e. members of the `docker` group. Regular users can run Docker only through Socker.

Socker is composed of two executables: *socker* binary, and

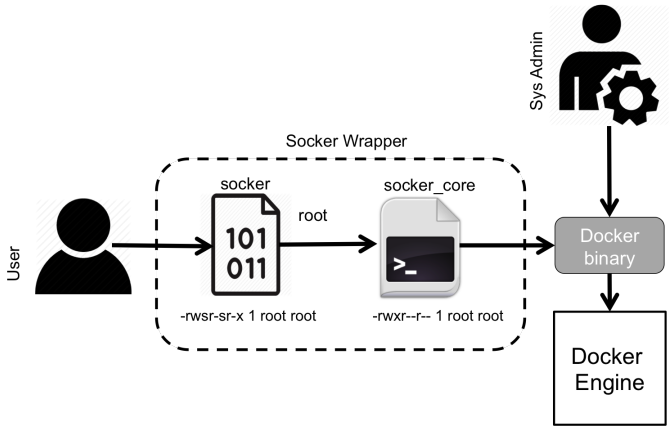

Figure 4: Socker Wrapper

socker_core script (currently bash). socker binary is the entry point for users to run inside SLURM job scripts. An example:

```
socker centos sleep 100
```

Where `centos` is the image name, and `sleep 100` is the command to run. socker binary performs the following:

1. Collects the image name and the command to be executed. These are provided by the user as command-line arguments.
2. Collects the user information, i.e. UID and GID.
3. Change the UID to 0, i.e. run the following command as root.
4. Calls socker_core passing the previously collected values as arguments.

To make socker binary executable by regular users and run commands as root at the same time, it needs to be configured as follows:

- Enables `setuid` for all: `chmod +s socker`. This will make it executable by everybody and give it permission to set the user ID, e.g. to root.
- socker binary must be a 'binary', not a script (that starts with #!), since most Linux systems for security reasons ignore the above configuration for scripts.

socker binary internally runs socker_core, which is executable only by root. socker_core performs the main Socker functionality:

1. Runs the Docker image as the user, using `docker run -u UID:GID`, and detach from the container.
2. Enforces membership of the container process in the cgroups created by SLURM for the job.

Algorithm 1 describes the overall Socker functionality.

Functionality is divided between a binary and a bash script instead of just one binary because Socker is designed to be simple and portable. Maybe for another SLURM cluster, system administrators would prefer to use their own, e.g. Python script, as socker_core to support different settings (e.g. mount specific volumes inside containers, support only a specific list of images, etc.)

Algorithm 1 Container launching procedure

Initialization:
IMG {Docker image to run}
CMD {Command to execute on the container}

Start:
uid ← getuid() {Store UID of the submitting user}
gid ← getgid() {Store GID of the user's default group}
jobid ← $SLURM_JOB_ID {Store the ID of the SLURM job}
Set UID ← 0 {Change the user to root}
call SOCKER_CORE(uid,gid,jobid,IMG,CMD) {Run socker_core as root}

Procedure SOCKER_CORE(UID,GID,JOBID,IMG,CMD): {*socker_call* script procedure}

Start:
cid← **call docker run**(detached=*true*,user=UID:GID, image=IMG, command=CMD) {Start container as detached and store it's ID}
pid←getPID(cid) {Store container's process ID}
if ∄ **process**(pid) **then** {Failed to start the container}
  Report Error
  Exit
for all cg ∈ **CGroup**(*slurm/UID/JOBID*) {for each cgroup created by SLURM for job: JOBID}
  classify process(pid) ∈ gc {Set container's process as a member in gc}
call docker wait(cid) {Wait for container cid to run the command and exit}
call docker remove(cid) {Clean up after the container exits}
End

7. SYSTEM CONFIGURATION

To enable Socker on Abel, Docker 1.7 is installed on the test compute nodes. Currently Docker doesn't support shared image repositories among hosts, so each compute node has it's local image repository (layered filesystem). To avoid storing images on the local disks of compute nodes, we are currently testing the allocation of all image repositories on the shared FhGFS. To do so, the storage has to be in the form of Ext4 images since Docker doesn't yet support FhGFS as a backing filesystem. Another issue is that we cannot rely on the compute nodes pulling images from the Docker hub, as this might be very slow sometimes in addition to requiring all compute nodes to have Internet access (which is not possible for Colossus). To resolve the issue, we installed a local Docker registry. Upon user requests, images are pulled from the Docker hub and pushed to the local registry. System configuration is depicted in Figure 5.

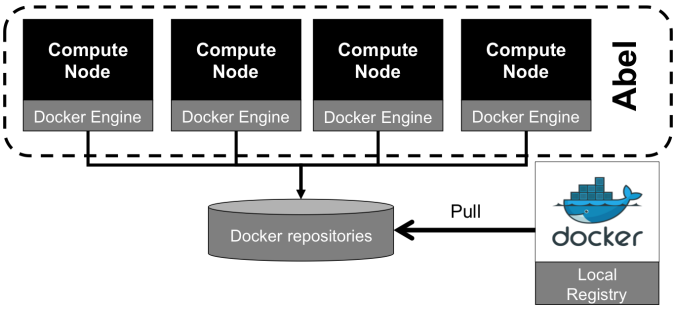

Figure 5: System Configuration

8. ELEMENTARY PERFORMANCE EVALUATION

Elementary benchmarking has been performed to indicate whether socker would introduce any computational overhead. We used a Docker image for Bowtie [19] to run sequence alignment (`genomicpariscentre/bowtie1` from Docker hub) . ≃5 GiB *Fastq* input file was used to be aligned against human genome *hg19*. We used the pMap package [20] to run the alignment in parallel using MPI. Three compute nodes on Abel (2*16 + 20 cores) were reserved for the testing. Testing included the following configurations:

1. pMap calling Native Bowtie
2. pMap running the Bowtie image using `docker run` and system administrator account.
3. pMap running the Bowtie image using Socker and regular user account.

For Docker and Socker configuration, the directory containing the reference genome index files together with the user's home directory were mounted inside the container as a volumes. Figure 6 presents the total run time against the number of parallel processes. 1, 8, 16, and 32 processes were used. Socker, as expected introduces almost no additional overhead which is obvious since it is just a wrapper for `docker run`.

9. KNOWN ISSUES

There are known Issues and limitations for running Docker on HPC in general and for Socker in particular:

- Containers won't solve the Hardware architecture incompatibility. For instance if there is image X with application Y that is compiled for a specific CPU architecture, Docker portability won't be useful here. Docker works above the kernel not below.

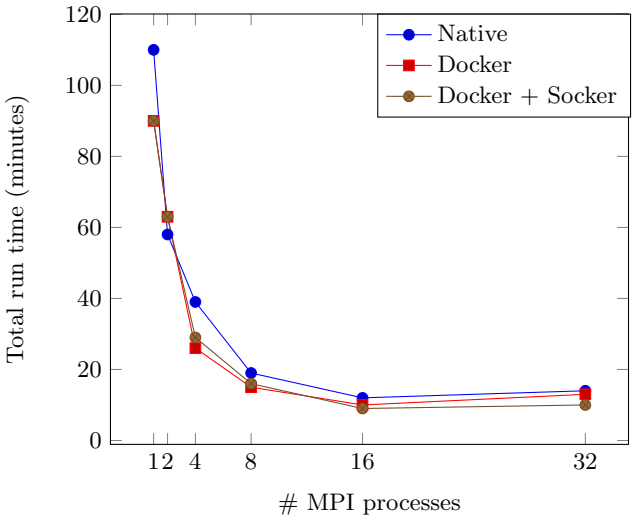

Figure 6: Run time of parallel pMap MPI processes using Bowtie aligner as: Bowtie binary, Bowtie Docker image using `docker run`, and Bowtie Docker image using Socker

- Containers won't clear operational maintenance mess. For instance if application A is compiled using a specific version of MPI, and application B is compiled using a different MPI version, then application C has both A and B as dependencies: C won't work even if A and B are Docker images. This is a reason multiple base images is not recommended by Docker despite the fact that it is technically supported.

- Containers are mainly designed for one application per container, while VMs can have a large collection of applications. Not all users publishing images on Docker hub understand this. Our system administrators may receive a request to pull image X which contains application Y, and image X is huge containing a lot of other stuff and would cause considerable memory overhead when loaded. In this case we would either find another lightweight image containing Y or modify the Dockerfile ourselves to include only what is necessary, which is a cost.

- Socker enforces inclusion of a running container in the SLURM cgroups by assigning the container's process as a member of these cgroups. This is based on the assumption that a container runs only one process. Some images are configured to start multiple processes, e.g. database servers. However, such images are not interesting for us in HPC. But again, we need to double check the Dockerfile to ensure that only one process is initially started.

- Socker is designed for Docker 1.7, in which running container as user X can assign X to only one user group. In Docker 1.8+, the `-group-add` option enables assigning additional groups. This is important in some cases. For instance on Abel, we have a shared area `/projects` where each research project has a directory accessible only by this project's members. This is done by creating a file group X for each project. A SLURM user who is a member of project X may want to mount `/projects/X` in his/her containers. To do so, the user needs to be configured (in the container) as a member of his/her default group (to access his/her home directory) in addition to the file group X (to access the project area). This is currently not supported due to the described limitation.

10. CONCLUSIONS

Linux containers is a useful technology for both users and system administrators supporting portability and less overhead than traditional VMs. Docker is a user friendly platform for creating and managing containers and is used by millions of users worldwide [3]. Enabling Docker containers on HPC clusters for scientific computing is demanded by a lot of researchers. Socker is a new wrapper for running Docker containers on SLURM, mainly targeting Abel HPC cluster at the University of Oslo. Socker is designed to be simple and portable so that it can be used on any SLURM cluster. Socker runs containers securely as the user, and enforces bounding the resource usage of running containers by the amount of resources assigned by SLURM. Socker is still in the testing phase, and further development is ongoing to resolve operational issues.

11. AKNOWLEDGEMENTS

This work is funded by the European ELIXIR [21] project, the Norwegian node, and the nordic Tryggve [22] project which is part of NeIC (Nordic e-Infrastructure Collaboration) [23], in addition to USIT (The university Center for Information Technology) at the University of Oslo. Testing is performed on Abel HPC cluster.

12. REFERENCES

[1] ""bio-linux overview". *http://environmentalomics.org/bio-linux/* accessed 2016-05-21."

[2] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[3] "rkt - app container runtime," https://github.com/coreos/rkt, accessed: 2016-05-21.

[4] "Lxd - linux containers," linuxcontainers.org/lxd, accessed: 2016-05-21.

[5] "Microsoft research - drawbridge," http://research.microsoft.com/en-us/projects/drawbridge/, accessed: 2016-05-21.

[6] "The abel computer cluster," http://www.uio.no/english/services/it/research/hpc/abel/, accessed: 2016-05-21.

[7] M. A. Jette, A. B. Yoo, and M. Grondona, "Slurm: Simple linux utility for resource management," in *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*. Springer-Verlag, 2002, pp. 44–60.

[8] "Docker swarm," https://docs.docker.com/swarm/, accessed: 2016-05-21.

[9] D. K. Rensin, *Kubernetes - Scheduling the Future at Cloud Scale*, 1005 Gravenstein Highway North

[3] 16 Docker repositories already exceeded 10M pulls each

Sebastopol, CA 95472, 2015. [Online]. Available: http://www.oreilly.com/webops-perf/free/kubernetes.csp

[10] M. Litzkow, M. Livny, and M. Mutka, "Condor - a hunter of idle workstations," in *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[11] "Docker and htcondor," https://research.cs.wisc.edu/htcondor/HTCondorWeek2015/presentations/ThainG_Docker.pdf, accessed: 2016-05-21.

[12] D. Jacobsen and S. Canon, "Contain this, unleashing docker for hpc," in *Cray User Group 2015*, April 23, 2015.

[13] "Lxc - linux containers," linuxcontainers.org/lxc, accessed: 2016-05-21.

[14] ""file system". http://docs.docker.com/terms/filesystem/. retrieved 2015-04-22."

[15] J.-S. Pendry and M. K. McKusick, "Union mounts in 4.4bsd-lite," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, ser. TCON'95. Berkeley, CA, USA: USENIX Association, 1995, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267411.1267414

[16] "Layers," https://docs.docker.com/terms/layer/, accessed: 2015-04-22.

[17] J. Corbet, "Notes from a container," https://lwn.net/Articles/256389/, October 29, 2007, accessed: 2016-05-21.

[18] "Tjenester for sensitive data (tsd)," http://www.uio.no/tjenester/it/forskning/sensitiv/, accessed: 2016-05-21.

[19] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome Biology*, vol. 10, no. 3, p. R25, 2009.

[20] "pmap: Parallel sequence mapping tool," http://bmi.osu.edu/hpc/software/pmap/pmap.html, accessed: 2016-05-21.

[21] "Elixir: A distributed infrastructure for life-science information," https://www.elixir-europe.org/, accessed: 2016-05-21.

[22] "Neic tryggve: Collaboration on sensitive data," https://wiki.neic.no/wiki/Tryggve, accessed: 2016-05-21.

[23] "Neic: Nordic e-infrastructure collaboration," https://neic.nordforsk.org/, accessed: 2016-05-21.