Docker containers on Maxwell

Overview

Docker is an open-source project that automates the deployment of applications inside software containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By doing so, thanks to the container, the developer can rest assured that the application will run on any other Linux machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code.

Docker is kind of like a virtual machine, but instead of creating a whole virtual operating system, it lets applications take advantage of the same Linux kernel as the system they're running on. That way, the application only has to be shipped with things that aren't already on the host computer instead of a whole new OS. This gives a significant performance boost and reduces the size of the application.

See https://www.docker.com/ for more info.

Docker on Maxwell

Docker has been deployed on Maxwell cluster. One can use it directly on workgroup servers or submit interactive or batch job via SLURM. Several scripts are available to facilitate the Docker usage.

The following sections will describe the steps necessary to run Docker on Maxwell cluster.

Please note that we are in a beta-testing mode yet, so things can go wrong anytime. Contact maxwell.service@desy.de for help/sharing your experience.

Installing Docker locally

Although not necessary, you may to install Docker on your local machine, so that you prepare and test a Docker image for runs on Maxwell. You can get installation instruction for your operating system on https://docs.docker.com/engine/installation/. Note that root privileges may be needed to install and run Docker commands. Note also that at the moment using Docker on "DESY Green Desktop" is not possible. Alternatively, one can use max-wgs server to prepare images.

Working with images

To start a Docker container you need to provide an image for the container. The images can be downloaded from one of the Docker Hub repositories (private or official) or downloaded from a private registry.

Typically one builds his own image on the base of one of many pre-existing images which can be downloaded from Docker Hub (https://hub.docker.com/). You then need to create a Dockefile which contains instructions to build an image (https://docs.docker.com/engine/reference/builder/).

For a single-node jobs any appropriate image can be used. For HPC jobs the image should have at least an MPI library (we recommend OpenMPI), Infiniband drivers and ssh service to be started. The easiest way is to start with the centos_mpi image (see example).

The images in the Docker cache of a local node are shared between all users. This means everyone can delete or replace, everyone's image in the period when a worker node is not allocated exclusively for your job or in anytime on other nodes. You should not expect that the image you've created some time ago is still there and you may want to verify that the image you create container from is the correct one by e.g.

1) always pull an image from a repository/load it from a tgz file
2) check the image digest and compare with the right one (save it somewhere before that): `docker images --digests --format '{{.Digest}}' <image name>`

Docker Hub

One can create an account on Docker Hub (https://hub.docker.com/) and keep his images there. A commercial version allows to have private registries as well.

User namespaces

By default Docker containers are isolated via user namespaces, so that process's user and group IDs inside a container are different than on the host system. This allows for the root user in a container to be mapped to a non uid-0 user outside the container, which mitigate the risks of container breakout. The drawback is that user inside the container may not have (or have limited) access to the data on the host system. Therefore under certain, below described conditions user namespaces can be disabled.

Running containers

With user namespace active (default)

You can use all spectrum of Docker commands (https://docs.docker.com/engine/reference/commandline/).

Without user namespace active

You can disable user namespaces via *--userns=host* option. In this case you should start a container as a user with your real id and group id and also activate *--security-opt no-new-privileges* parameter, for example:

```
$ docker run -u `id -u`:`id -g` --userns=host --security-opt no-new-privileges ...
```

In this case you will Docker will start a process as a user with your credentials. You can have full access to mounted files from host. Other useful parameters are *--net=host* to have network access, *--group-add* to add secondary group, *--privileged* to have access to host devices, etc. Note that there will be no real user on the container in this case (no $HOME, etc) so some applications may fail to run. To overcome this problem one can mount appropriate /passwd and /groups files into the container and create $HOME directory. You can also have a look at the *dockerrun* script for more options (*cat `which dockerrun`* from a Maxwell node) or even better use this script directly (see below).

Using the dockerrun script

To ease the usage of docker with host namespace we provide the script *dockerrun.* It is installed on Maxwell.

```
$ dockerrun <other Docker parameters>
$ dockerrun -it ubuntu bash # again start bash in ubuntu with host namespace
```

Note that *dockerrun* will replace original entrypoint in Docker image in order to be able to add user and mount the current folder as /docker_pwd and set it as working directory. In some cases it will not work as expected. In this case you can use *-no-add-user* and/or -no-add-wd option for *dockerrun*. No user will be created and/or current folder will not be mounted, but your id and group ids will be passed to the container.

Running parallel jobs with Docker

As usual, you allocate resources with SLURM.

Then you create a virtual cluster (start a Docker container with sshd daemon running on each of the allocated nodes using *dockercluster* script):

```
$ dockercluster [arguments] <Docker image name> <extra Docker arguments>
```

dockercluster can accept optional arguments:

-u update Docker image before start (calls docker pull <Docker image name> on every node)

-n <name> specify cluster name (default is docker_$SLURM_JOB_ID)

-p <port> specify port for sshd (default is 2022)

-s stop and remove cluster (calls docker rm -f <name> on every node). You do not have to call this command as containers will be removed during SLURM job clean-up

Usually you will only use -u to update an image, but you will need other parameters if you want to start more than one cluster within one SLURM job.

Note that in *dockercluster* script Docker arguments should be added after the image name, which is in contradiction to standard Docker way but is necessary to separate script own arguments from Docker ones.

After your virtual cluster is ready, you can call *docker exec* commands in it.

You again need to provide you id and gid:

```
$ docker exec -u `id -u`:`id -g` <image name> <command>
```

Alternatively you can use *dockerexec* script (you can set DOCKER_CONT_NAME environment variable to replace default cluster name):

```
$ dockerexec <command>
```

Mounting host data

You can use -v option when starting Docker container to mount host folder (or single file). Full path should be used:

```
$ docker run -v /data/gpfs/user:/data -v /beegfs/desy/user/$USER:/anotherdata ...
$ dockerrun -v /data/gpfs/user:/data -v /beegfs/desy/user/$USER:/anotherdata ...
$ dockercluster <imagename> -v /data/gpfs/user:/data -v /beegfs/desy/user/$USER:/anotherdata ...
```

You can use -w option to assign working directory (subsequent *docker exec* calls will use it as staring point as well)

If you are using *dockerrun* script (or *dockercluster*) your current folder (`pwd`) will be automatically mounted as /docker_pwd and set as working directory with *-w* option. You can use -no-add-wd option for *dockerrun to skip this*

To mount a host folder Docker need to have access to the folder (as root for bind mount). On some filesystems this is not possible due to root squash. E.g. -v /asap3/petra3/gpfs/pxx/2019/data/xxx:/beamtime_data will fail. A workaround is to mount the upper level folder /asap3/petra3/gpfs/pxx/2019/data:/data and access the subfolder /data/xxx from within the container.

Examples

Creating a Docker image

Running a single job with Docker

Running an MPI job with Docker