

# Building Cloud IaaS infrastructures

Rosaria Tornisiello

February 20, 2021

## 1 Generation of an HTCondor cluster and running of a test job

HTCondor is a specialized workload management system for compute-intensive jobs. It is a batch systems that provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. This section describes the process of generation of an HTCondor cluster composed by three Virtual Machines (VMs): a Master node and two Worker nodes. This small computing infrastructure is obtained on the Amazon Web Services (AWS) Cloud and is based on Red Hat Enterprise Linux (RHEL). The installed batch system is tested running a BWA (Burrows-Wheeler Aligner) test job.

### 1.1 Instantiation of the three VMs

The first step is the initialization of three instances on the Amazon Elastic Compute Cloud (Amazon EC2). For the worker nodes a **t2.large** instance type is chosen, with a 30 GiB SSD of storage, while for the master node a **t2.large** instance type is chosen, but with a 10 GiB of storage. The Amazon Machine Image (AMI) selected for the nodes is RHEL, version 7.6. Moreover, the same availability zone is chosen for all the nodes (us-east-1b). In order to control the inbound and outbound traffic, a unique security group is assigned to the VMs and the inbound rules are set as shown in figure 1.

Inbound rules	Outbound rules	Tags	
Inbound rules			
Type	Protocol	Port range	Source
All TCP	TCP	0 - 65535	sg-0aa52687d126e2a1c (launch-wizard-1)
SSH	TCP	22	0.0.0.0/0
All ICMP - IPv4	ICMP	All	sg-0aa52687d126e2a1c (launch-wizard-1)

Figure 1: Security group inbound rules

For simplicity, the security group is set to allow Transmission Control Protocol (TCP) for ports 0 - 65535 from the same security group in such a way that all the nodes are able to communicate between one another through HTCondor and share a volume by means of the Network File System server (NFS). Internet Control Message Protocol (ICMP) ports where opened to allow ping requests from the same security group. Finally, the security group opens TCP port 22 to allow Secure Shell (SSH) from everywhere.

### 1.2 Setting up the master node

As first step the HTCondor dependencies and HTCondor itself are installed running the following commands:

```
yum install wget
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
yum localinstall epel-release-latest-7.noarch.rpm
yum clean all

wget http://research.cs.wisc.edu/htcondor/yum/repo.d/htcondor-stable-rhel7.repo
cp htcondor-stable-rhel7.repo /etc/yum/repos.d/
wget http://htcondor.org/yum/RPM-GPG-KEY-HTCondor
rpm --import RPM-GPG-KEY-HTCondor
yum install condor-all
```

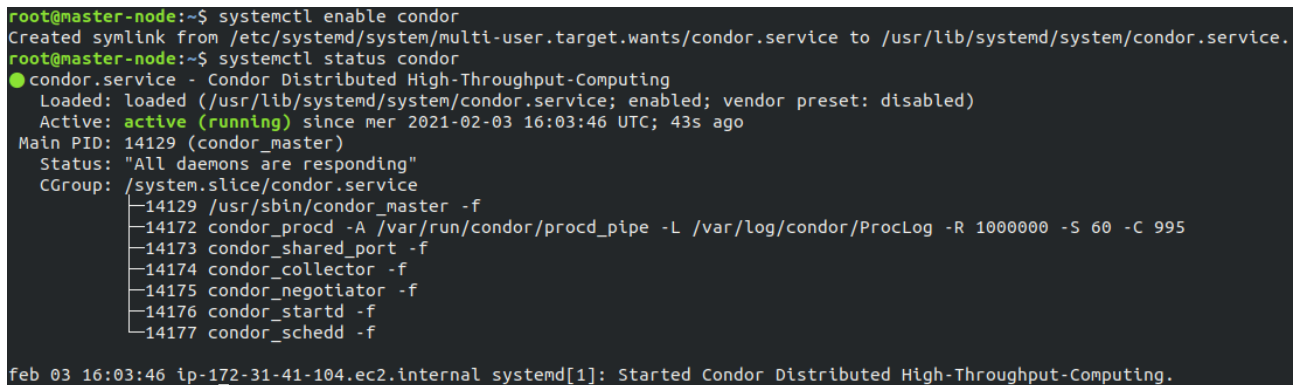
The next step is the modification of the HTCondor configuration file located at `/etc/condor/` adding the following lines:

```
CONDOR_HOST = Master node Private IP address
DAEMON_LIST = COLLECTOR, MASTER, NEGOTIATOR, STARTD, SCHEDD
HOSTALLOW_READ = *
HOSTALLOW_WRITE = *
HOSTALLOW_ADMINISTRATOR = *
```

Then HTCondor is started and enabled:

```
systemctl restart condor
systemctl enable condor
systemctl status condor
```

The output is shown in figure 2.



```
root@master-node:~$ systemctl enable condor
Created symlink from /etc/systemd/system/multi-user.target.wants/condor.service to /usr/lib/systemd/system/condor.service.
root@master-node:~$ systemctl status condor
● condor.service - Condor Distributed High-Throughput-Computing
   Loaded: loaded (/usr/lib/systemd/system/condor.service; enabled; vendor preset: disabled)
   Active: active (running) since mer 2021-02-03 16:03:46 UTC; 43s ago
 Main PID: 14129 (condor_master)
   Status: "All daemons are responding"
    CGroup: /system.slice/condor.service
           └─14129 /usr/sbin/condor_master -f
             └─14172 condor_procd -A /var/run/condor/procd_pipe -L /var/log/condor/ProcLog -R 1000000 -S 60 -C 995
               └─14173 condor_shared_port -f
                 └─14174 condor_collector -f
                   └─14175 condor_negotiator -f
                     └─14176 condor_startd -f
                       └─14177 condor_schedd -f

feb 03 16:03:46 ip-172-31-41-104.ec2.internal systemd[1]: Started Condor Distributed High-Throughput-Computing.
```

Figure 2: Condor status

The NFS server is then installed using the following commands:

```
yum install nfs-utils rpcbind
systemctl enable nfs-server
systemctl enable rpcbind
systemctl enable nfs-lock
systemctl enable nfs-idmap
systemctl start rpcbind
systemctl start nfs-server
systemctl start nfs-lock
systemctl start nfs-idmap
```

Then the file `/etc/exports` which controls which directories are exported to remote hosts and specifies options is modified: the following two lines are added to specify the private IPs of the worker nodes with which the master node should share files:

```
/data2 WorkerNode1-privateIP(rw, sync, no_wdelay)
/data2 WorkerNode2-privateIP(rw, sync, no_wdelay)
```

Lastly, an empty file called "test" is created and located at `/data2` in order to check from both the Worker Nodes if it is accessible.

### 1.3 Setting up the worker nodes and mounting a volume

Firstly, a new `t2.large` VM is instantiated and it is set as a worker node. The first phase is the installation on HTCondor using the same procedure adopted for the Master Node. In this case the lines added to the HTCondor configuration file are:

```
CONDOR_HOST = Master Node Private IP address
DAEMON_LIST = MASTER, STARTD
HOSTALLOW_READ = *
HOSTALLOW_WRITE = *
HOSTALLOW_ADMINISTRATOR = *
```

The NFS server is also installed on the worker node in order to set up the access to the shared volume, issuing the following command:

```
yum install nfs-utils
```

The next step is the creation of a volume from the snapshot BDP1.2020, used during the BDP course, on the AWS interface. This volume contains all the necessary files to run the scientific test job described in the following paragraph. It is attached to the master node through the AWS interface and mounted as `/data2`. Moreover the following line is added to the file `/etc/fstab` on the master node in such a way that the mounted volume persists across reboots:

```
/dev/xvdg1      /data2  ext4 defaults 0 0
```

Then, a new directory called `data2`, is created using the `mkdir` command on the master node and the volume is mounted using the `mount -a` command.

In order to check the correctness of the process, the command `df -h` is issued and the `/data2` directory is inspected. The output is shown in figure 3.

```
root@master-node:/data2$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/xvda2      10G   2,1G  8,0G  21% /
devtmpfs        1,9G   0    1,9G   0% /dev
tmpfs           1,9G   0    1,9G   0% /dev/shm
tmpfs           1,9G  17M   1,9G   1% /run
tmpfs           1,9G   0    1,9G   0% /sys/fs/cgroup
tmpfs           379M   0    379M   0% /run/user/1000
/dev/xvdf1       99G   16G   79G   17% /data2
root@master-node:/data2$ ll
total 20
drwxr-xr-x. 5 root root 4096 19 apr 2020 BDP1_2020
drwx-----. 2 root root 16384 26 apr 2020 lost+found
-rw-r--r--. 1 root root    0  3 feb 17.45 test
```

Figure 3: Mounted volume on master node.

Also on the worker node the volume is mounted and the file `/etc/fstab` is modified adding the following line:

```
Master Node private IP: /data2 /data2 nfs defaults 00
```

Then the volume is inspected to check if it accessible from the worker node (figure 4).

```
root@worker-node1:/home/ec2-user$ ll /data2
total 20
drwxr-xr-x. 5 root root 4096 19 apr 2020 BDP1_2020
drwx-----. 2 root root 16384 26 apr 2020 lost+found
-rw-r--r--. 1 root root    0  3 feb 17.45 test
```

Figure 4: Access volume from worker node.

As shown in the previous image, the file "test" is visible from the worker node, so the volume is correctly shared between the nodes.

The second worker node is built from the first worker node AMI through the AWS interface. This procedure shows how it is easy and scalable to replicate machines without the need of manually configure them.

## 2 Scientific application: a BWA test job

This section presents a simple example of a scientific application i.e. a BWA test job. BWA is a program for aligning sequencing reads against a large reference genome (e.g. human genome). In this case 5 reads of a patient are aligned against the entire human genome.

The first step is the installation of the BWA tool both on the master node and the worker nodes, using the following command:

```
yum install -y bwa
```

Successively, the `bwa_batch.job` file is created and written as follows:

```

##### The program that will be executed #####

Executable = align.py
number = $(Process)+1

##### Input Sandbox #####

Input      = patient2/read_${INT(number)}.fa
#Can contain standard input

transfer_input_files = patient2/read_${INT(number)}.fa

## Arguments that will be passed to the executable ##

Arguments  = read_${INT(number)}.fa

##### Output Sandbox #####

Log        = read_${INT(number)}.log
# will contain condor log

Output     = read_${INT(number)}.out
# will contain the standard output

Error      = read_${INT(number)}.error
# will contain the standard error

transfer_output_files = read_${INT(number)}.sam.gz, read_${INT(number)}.sai, md5.txt

##### condor control variables #####

should_transfer_files = YES
when_to_transfer_output = ON_EXIT

Universe    = vanilla

#####

Queue 5

```

The executable file align.py is the following python script:

```

#!/usr/bin/python
import sys,os
from timeit import default_timer as timer

# Control variables
#####

start = timer()

dbpath = "/data2/BDP1_2020/hg19/"
dbname = "hg19bwaidx"

queryname = sys.argv[1]

out_name = queryname[: -3]

md5file = "md5.txt"

command = "/data2/BDP1_2020/condor/hg/bwa aln -t 1 " + dbpath + dbname + " "
        + queryname + " > " + out_name + ".sai"
print "launching command: " , command
os.system(command)

command = "/data2/BDP1_2020/condor/hg/bwa samse -n 10 " + dbpath + dbname + " "
        + out_name + ".sai " + queryname + " > " + out_name + ".sam"
print "launching command: " , command
os.system(command)

print "Creating md5sums"
os.system("md5sum " + out_name + ".sai " + " > " + md5file)
os.system("md5sum " + out_name + ".sam " + " >> " + md5file)

print "gzipping out text file"

```

```

command = "gzip " + out_name + ".sam"
print "launching command: " , command
os.system(command)

execution_time = timer() - start
print "Total execution time: " + str(execution_time)

print "exiting"

exit(0)

```

Then, the `condor_status` is checked to ensure that all the three nodes are included in the cluster (figure 5).

```

ec2-user@master-node:/data2/BDP1_2020/hg19/bwa_rosi$ condor_status
Name                                         OpSys      Arch      State      Activity  LoadAv  Mem    ActvtyTime
slot1@ip-172-31-32-244.ec2.internal        LINUX      X86_64    Claimed    Busy      0.000   3909   0+00:00:03
slot2@ip-172-31-32-244.ec2.internal        LINUX      X86_64    Claimed    Busy      0.000   3909   0+00:00:03
slot1@ip-172-31-41-104.ec2.internal        LINUX      X86_64    Claimed    Busy      0.000   3909   0+00:00:11
slot2@ip-172-31-41-104.ec2.internal        LINUX      X86_64    Claimed    Busy      0.000   3909   0+00:00:11
slot1@ip-172-31-42-37.ec2.internal         LINUX      X86_64    Claimed    Busy      0.000   3909   0+00:00:03
slot2@ip-172-31-42-37.ec2.internal         LINUX      X86_64    Unclaimed  Idle      0.000   3909   0+00:05:03

Machines  Owner  Claimed  Unclaimed  Matched  Preempting  Drain
X86_64/LINUX      6      0        5          1          0          0      0
Total            6      0        5          1          0          0      0

```

Figure 5: Condor cluster status.

As it is evident from this output, all the nodes are running the job. Next, the submission of the job on the cluster is performed using the following command:

```
condor_submit bwa_batch.job
```

In order to check if the job has been submitted and to retrieve information about jobs in queue it is possible to use the command `condor_q`, which the output is shown in figure 6.

```

ec2-user@master-node:/data2/BDP1_2020/hg19/bwa_rosi$ condor_q

-- Schedd: ip-172-31-41-104.ec2.internal : <172.31.41.104:9618?... @ 02/07/21 09:21:39
OWNER   BATCH_NAME   SUBMITTED   DONE    RUN    IDLE   TOTAL  JOB_IDS
ec2-user ID: 52      2/7 09:21    _       4       1       5 52.0-4

Total for query: 5 jobs; 0 completed, 0 removed, 1 idle, 4 running, 0 held, 0 suspended
Total for ec2-user: 5 jobs; 0 completed, 0 removed, 1 idle, 4 running, 0 held, 0 suspended
Total for all users: 5 jobs; 0 completed, 0 removed, 1 idle, 4 running, 0 held, 0 suspended

```

Figure 6: Condor\_q output.

At this point BWA does not complete the alignment because of memory errors. In order to try to solve this issue the master node is temporarily switched to `t2.xlarge`. This solution works, so the cluster is able to conclude the job and to output the files correctly. Figure 7 shows the error file without errors of one of the aligned reads.

In order to compute the average time to carry out a single task, the following command is issued:

```
cat read_*.out | grep "Total execution time:"
```

Figure 8 shows the output.

The average execution time is equal to about 82.46 seconds.

## 2.1 Data Management model

In the described pipeline, the large input file that is the human genome index has been made available to the worker nodes by means of the NFS server. Regarding the small input files, they have been made available through

```

ec2-user@master-node:/data2/BDP1_2020/hg19/bwa_rosi$ cat read_1.error
[bwa_aln] 17bp reads: max_diff = 2
[bwa_aln] 38bp reads: max_diff = 3
[bwa_aln] 64bp reads: max_diff = 4
[bwa_aln] 93bp reads: max_diff = 5
[bwa_aln] 124bp reads: max_diff = 6
[bwa_aln] 157bp reads: max_diff = 7
[bwa_aln] 190bp reads: max_diff = 8
[bwa_aln] 225bp reads: max_diff = 9
[bwa_aln_core] calculate SA coordinate... 0.12 sec
[bwa_aln_core] write to the disk... 0.00 sec
[bwa_aln_core] 1000 sequences have been processed.
[main] Version: 0.7.15-r1140
[main] CMD: /data2/BDP1_2020/condor/hg/bwa aln -t 1 /data2/BDP1_2020/hg19/hg19bwaidx read_1.fa
[main] Real time: 36.052 sec; CPU: 1.596 sec
[bwa_aln_core] convert to sequence coordinate... 2.43 sec
[bwa_aln_core] refine gapped alignments... 0.51 sec
[bwa_aln_core] print alignments... 0.02 sec
[bwa_aln_core] 1000 sequences have been processed.
[main] Version: 0.7.15-r1140
[main] CMD: /data2/BDP1_2020/condor/hg/bwa samse -n 10 /data2/BDP1_2020/hg19/hg19bwaidx read_1.sai read_1.fa
[main] Real time: 55.746 sec; CPU: 2.962 sec

```

Figure 7: Error file.

```

ec2-user@master-node:/data2/BDP1_2020/hg19/bwa_rosi$ cat read_*.out | grep "Total execution time:"
Total execution time: 91.8468370438
Total execution time: 91.8276991844
Total execution time: 91.4526309967
Total execution time: 55.7811951637
Total execution time: 81.3991978168

```

Figure 8: Execution time of each read alignment.

the HTCondor input Sandbox. HTCondor works well without a shared file system between the submit machines and the worker nodes. The HTCondor file transfer mechanism allows the user to explicitly select which input files are transferred to the worker node before the job starts. HTCondor will transfer these files, potentially delaying this transfer request, if starting the transfer right away would overload the submit machine. To enable the file transfer mechanism, two commands are placed in the job's submit description file: `should_transfer_files` and `when_to_transfer_output`. Queuing requests like this prevents the crashes so common with too-busy shared file servers. These input files are placed into a scratch directory on the worker node, which is the starting current directory of the job. When the job completes, by default, HTCondor detects any newly-created files at the top level of this sandbox directory, and transfers them back to the submitting machine. The input sandbox is what we call the executable and all the declared input files of the job. Files in the `transfer_input_files` command are specified as they are accessed on the submit machine. The job, as it executes, accesses files as they are found on the execute machine. The `transfer_output_files` are also specified. The set of all files created by the job is the output sandbox. The output files of HTCondor are:

- the `log` file which is a listing of events in chronological order that occurred during the life of one or more jobs;
- the `out` file which captures any information the program would normally write to the screen (the standard output of the program);
- the `error` file which capture any error messages the program would normally write to the screen (the standard error).

### 3 Containerized version of the bwa application using Docker

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application. Container images become containers at runtime and in the case of Docker containers - images become containers when they run on Docker Engine. Containers are an abstraction at the app layer that packages code and dependencies together. This section describes the creation of a container image using Docker and its usage to run the very same scientific application described in the previous paragraph.

The first step is the installation of Docker on the three nodes. Since the installation from repository gives several

errors, the manual installation from the rpm files is implemented. The command `docker --version` shown in figure 9 confirms the correct installation.

```
ec2-user@master-node:~$ docker --version
Docker version 20.10.3, build 48d30b5
```

Figure 9: Docker version

The program is started and enabled through the following commands:

```
systemctl start docker
systemctl status docker
systemctl enable docker
```

The output is shown in figures 10 and 11.

```
ec2-user@master-node:~$ sudo systemctl start docker
ec2-user@master-node:~$ systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; disabled; vendor preset: disabled)
   Active: active (running) since lun 2021-02-08 15:33:52 UTC; 6s ago
     Docs: https://docs.docker.com
    Main PID: 4989 (dockerd)
      Tasks: 10
     Memory: 157.7M
    CGroup: /system.slice/docker.service
            └─4989 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
```

Figure 10: Docker status command output.

```
ec2-user@master-node:~$ sudo systemctl enable docker
Created symlink from /etc/systemd/system/multi-user.target.wants/docker.service to /usr/lib/systemd/system/docker.service.
```

Figure 11: Docker enable command output.

Finally, the following command is executed to add the user to the Docker group:

```
usermod -g docker <username>
```

After finishing the installation and configuration of Docker, the following lines are added to the `condor_config` file:

```
DOCKER_VOLUMES = BDP1_2020
DOCKER_VOLUME_DIR_BDP1_2020 = /data2
DOCKER_MOUNT_VOLUMES = BDP1_2020
```

In general, an administrator of a machine can optionally make additional directories on the host machine readable and writable by a running container. To do this, the admin must first give an HTCondor name to each directory with the `DOCKER_VOLUMES` parameter. Then, each volume must be configured with the path on the host OS with the `DOCKER_VOLUME_DIR_XXX` parameter. Finally, the parameter `DOCKER_MOUNT_VOLUMES` tells HTCondor which of these directories to always mount onto containers running on this machine. In this case, since the container will need the same input files used for the previous test job which are located in the `/data2/BDP1_2020` directory, the latter is set up as volume accessible by Docker.

For the purpose of this project, either a ready-made bwa container image could be pulled from Docker Hub or it could be built from a dockerfile written by the user. In this case the following dockerfile is created:

```
FROM ubuntu
COPY align.py align.py
RUN chmod +x align.py
RUN apt update
RUN apt install -y bwa
RUN apt install -y python
```

Afterwards, it is used to build the docker image executing the following command:

```
docker build -t <image name>
```



```
ec2-user@master-node: /data2/BDP1_2020/hg19/bwa_rosi$ sudo docker images
REPOSITORY          TAG             IMAGE ID         CREATED          SIZE
bwarosaria          latest          9f61049d0cc5    25 seconds ago  182MB
ubuntu              latest          f63181f19b2f    2 weeks ago     72.9MB
hello-world         latest          bf756fb1ae65    13 months ago   13.3kB
```

Figure 12: Docker images.

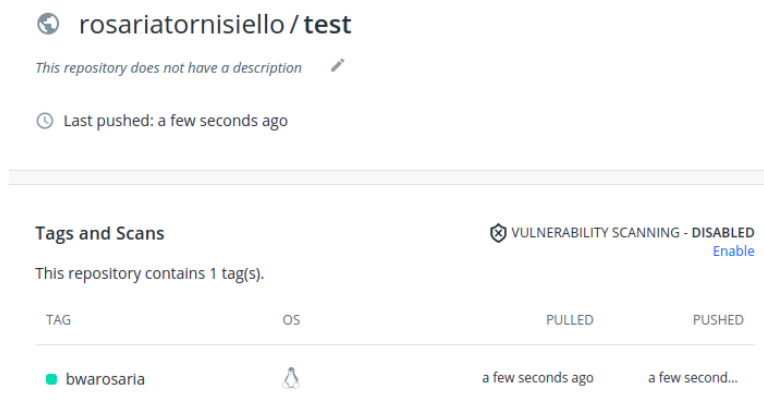


Figure 13: Docker hub repository.

Figure 12 shows the images created.

The docker image is then pushed in Docker hub as shown in figure 13.

In order to run the container just created inside the condor batch system, condor is added to the docker group:

```
sudo usermod -aG docker condor
```

Condor is reconfigured in all the machines executing the command `condor_reconfig` and in order to check if all the condor nodes have docker the following command is run:

```
condor_status -l | grep -i docker
```

As shown in figure 14, condor correctly detects docker as well as the mounted volume.

```
ec2-user@worker-node1:~$ condor_status -l | grep -i docker
DockerVersion = "Docker version 20.10.3, build 48d30b5"
HasDocker = true
HasDockerVolumeBDP1_2020 = true
StarterAbilityList = "HasJICLocalStdin,HasJICLocalConfig,HasTDP,HasPerFileEncryption,HasDocker,HasFileTransfer,HasTransferInputRemaps,HasVM,HasReconnect,HasMPI,HasFileTransferPluginMethods,HasJobDeferral,HasSelfCheckpointTransfer,HasRemoteSyscalls,HasCheckpointing"
DockerVersion = "Docker version 20.10.3, build 48d30b5"
```

Figure 14: Verify that condor detects that docker is installed.

The following step is the submission of the following test job file:

```
#####The program that will be executed #####

docker_image = rosariatornisiello/test:bwa_rosaria
Executable = align.py
number = $(Process)+1

##### Input Sandbox #####

Input      = patient2/read_${INT(number)}.fa
#Can contain standard input

transfer_input_files = patient2/read_${INT(number)}.fa

## Arguments that will be passed to the executable ##
```



```

Arguments = read_INT(number).fa

##### Output Sandbox #####

Log = read_INT(number).log
# will contain condor log

Output = read_INT(number).out
# will contain the standard output

Error = read_INT(number).error
# will contain the standard error

transfer_output_files = read_INT(number).sam.gz, read_INT(number).sai, md5.txt

##### condor control variables #####

should_transfer_files = YES
when_to_transfer_output = ON_EXIT

Universe = docker

#####

Queue 5

```

The job file is almost identical to the previous one but in this case the docker image is specified and the universe is switched to **docker**. A docker universe job instantiates a Docker container from a Docker image, and HTCondor manages the running of that container as an HTCondor job, on an execute machine. This running container can then be managed as any HTCondor job. The executable file is the same as the one used in the previous section.

After submitting the job, they are held because Docker has gone over memory limit (as shown in figure 15).

```

060.003: Job is held.

Hold reason: Error from slot2@ip-172-31-32-244.ec2.internal: Docker job has gone over memory limit of 1893 Mb

Last successful match: Mon Feb 8 18:23:21 2021

```

Figure 15: Docker hold reason.

In order to solve this issue, the following lines are appended to the `condor_config` file:

```

SLOT_TYPE_1 = cpus=100%, ram=100%, disk=100%, swap=100%
NUM_SLOT_TYPE_1 = 1

```

This serves the purpose of defining the slot types, in particular these parameters list how much of each system resource the user wants in the given slot type. This is done by defining configuration variables in the form `SLOT_TYPE_N` where the `N` represents an integer, which specifies the slot type defined. The number of slots of each type created is configured with `NUM_SLOTS_TYPE_N`.

After this reconfiguration of condor, the job are executed but the alignment of each read does not reach the completion due to the same error obtained in the previous section, i.e. BWA fails to allocate the memory. To solve this last problem, again the master node is converted to a t2.xlarge machine in order to have more RAM available.

The total execution time of each job is shown in figure 16.

```

ec2-user@master-node:/data2/BDP1_2020/hg19/bwa_rosi$ cat read_*.out | grep "Total execution time:"
Total execution time: 143.792137146
Total execution time: 143.811068058
Total execution time: 143.534133911
Total execution time: 4.3074259758
Total execution time: 33.0731101036

```

Figure 16: Docker total execution time.

In this case the average time needed to align a fasta is 93.70 seconds.

## 4 Non-containerized vs containerized application

Docker containers and virtual machines are both ways of deploying applications inside environments that are isolated from the underlying hardware. What's fundamentally different between containers and VMs is that at start time, a VM boots a new, dedicated kernel for its environment, and starts a often rather large set of operating system processes. This makes the size of the VM much larger than a typical container that only contains the application. Indeed, multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less resources than VMs, which is a critical feature for computational efficiency. Likewise, due to their lower levels of resource consumption, containers may help to save money on cloud computing costs, thus reducing the required overhead.

Moreover in cases of repetitive jobs like a BWA alignments, containers are a convenient choice because each running instance, based on a template image containing the binaries and libraries required can be shared and reused using container registries, which are basically repositories that host container images.

The cons of containers are mainly bounded to security: virtual machines are indeed more isolated from each other and from the host system than Docker containers. That is because virtual machines don't directly share any kernel or other resources with the host system.

In this project, non-containerized and containerized versions of the test job performed very similarly with only 10 seconds of difference in the mean execution time. Surprisingly, the containerized application performed slightly worse than the native one. The reason for this outcome may be bound to the low number of aligned reads that gives a biased average. Moreover, from other experiments found in literature (e.g. <https://arxiv.org/pdf/1807.01842.pdf>), it is possible to observe that the advantages of using containers become noticeable when the number of VMs and containers compared increases.

## 5 Real use-case

An example of real use-case can be a Whole Genome Sequencing (WGS) which in the last decades has emerged as a compelling paradigm for routine clinical diagnosis, genetic risk prediction, and rare diseases. Human DNA is comprised of approximately 3 billion base pairs. 30x coverage sequencing of a single genome will produce approximately 100 gigabytes (GB) of nucleotide bases, and its corresponding FASTQ file will be about 250 GB. In particular, considering a single patient and a read length of 150 bp, the total amount of reads is equal to about 600 million. Based on the simple infrastructure built for this project and considering the containerized application, it took 93.70 seconds on average to align a single fasta file that contains 1000 reads, so aligning 600 million reads would take about 21 months. Obviously this would be unfeasible and extremely expensive, given also that a WGS project usually comprises hundreds of patients. The overall infrastructure should be built according to the workload, increasing the number of worker nodes, the RAM, the CPUs and obviously the storage. Moreover in this project only one core is used for each job but the bwa alignment could be parallelized. For instance, the entire infrastructure could be composed by several master nodes and 100 worker nodes. Increasing the number of master nodes can add resiliency to the cluster in case of single node failure. If the computational time is approximately considered halved, doubling the number of cores, considering 100 `t2.2xlarge` machines which are characterized by 8 cores, the estimated time to align 600 million reads is approximately equals to 2 days. Given that a `t2.2xlarge` instance costs \$ 0.3712 per hour, the total amount for analyzing a single subject NGS data output would be of about \$ 1800.

Searching in the literature, it is possible to learn that many cloud-based services and bioinformatics platforms, applications, and resources have been developed to address the specific challenges of working with the large volumes of data generated by NGS technology. Cloud computing has created new possibilities to analyze NGS data at reasonable costs, especially for laboratories lacking a dedicated bioinformatics infrastructure. From the perspective of end users, there are three options to analyze NGS data on cloud computing:

- Commercial services: provide the users with well-established pipelines, user interfaces, and even application programming interfaces (APIs). This approach can reduce the time and effort required for setting up pipelines for NGS data analysis. For instance, DNAnexus and Seven Bridges offer various customizable NGS data analysis pipelines.
- Commercial or open bioinformatics platforms that are further customized to meet users' computational needs. For example CloudBioLinux which is a publicly accessible virtual machine (VM) based on an Ubuntu Linux distribution and is available to all Amazon EC2 users for free.
- Open-source tools that can be deployed into the cloud for any customized data analysis.

Moreover, AWS offers the possibility to use the Amazon EC2 spot instances which let the user take advantage

of unused EC2 capacity in the AWS cloud. Spot Instances are available at up to a 90% discount compared to On-Demand prices. Spot Instances can be used for various stateless, fault-tolerant, or flexible applications such as big data, containerized workloads, web servers, high-performance computing (HPC), and test development workloads.