European Commission

Joint Research Centre

Dynamic Batch service with HTCondor and Kubernetes

Presenter: Luca Marletta
Authors: Dario Rodriguez,
Luca Marletta,
Pierre Soille

European Commission
Joint Research Centre

JEODPP
The JRC Big Data Platform

September 25, 2019

Content

Introduction to Kubernetes (K8S)

Kubernetes can:

- orchestrate Docker containers across multiple hosts;
- make better use of hardware to maximize resources needed to run your apps;
- control and automate application deployments and updates;
- scale containerized applications and their resources on the fly;
- declaratively manage services;
- health-check and self-heal your apps with autoplacement, autorestart, autoreplication, and autoscaling

Operational schema of JRC Big Data Platform (JEODPP)

JEODPP provides ancillary services for analysing satellite images and big data for JRC.

JEODPP cluster LAN is not reachable from other JRC LANs and is accessible just over a web page as Jupyter Notebook or Guacamole (remote desktop).

JEODPP supports internal projects (Use Cases) providing interactive and batch processing (among other scenario).

A typical case is for user to submit through batch processing mode several workflows based on image analysis algorithms, to harvest subsequently statistics and indicators and then, to visualize the results in the interactive mode

JEODPP previous HTCondor working environment

Before introducing Kubernetes, the JEODPP cluster of nodes (about 130 servers for a processing power of about 3,000 cores) was partitioned according to the following services:

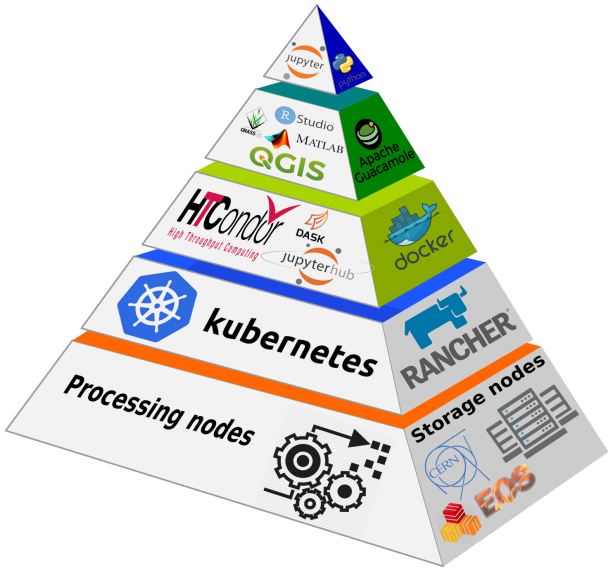
- interactive visualization;
- HTCondor batch processing (through Docker Universe);
- DASK under Kubernetes.

This led to an underuse of resources and several dificulties for managing diversity.

HTCondor pool was set to operate just in Docker universe, every job is executed within a container on all servers devoted to this usage.

SSH connections to the sched-machine is not allowed for standard users for security reason. Submission is realized via a web interface (IPython or shell in Guacamole)

Why migrating HTCondor cluster into K8S?



To improve:

- easy scalability of services;
- easy services and clusters management;
- optimization of resources;
- limitations for peaks in interactive visualization;
- idle cpu time when some some queues in HTCondor are completed.

Working schema in K8S: Full deployment

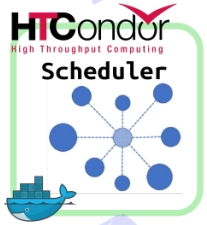
The deployment schema

Adminstration and deployment

kubectl create -f file.yaml

deployment

monitor

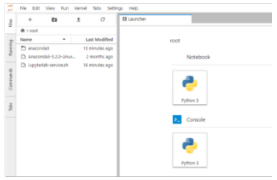


force workers to be
on different servers
with ANTIAFFINITY API call

Node: 1 IP: 139.191.240.100
cpu: 40 ram: 1.5 Tb

Node: 2 IP: 139.191.240.102
cpu: 40 ram: 1.5 Tb

Use case
Notebook



Submit and monitor
by final user

THE RELEVANT PART OF THE YAML FILE AND THE KEY DEVELOPMENT CONCEPTS

The deployment creates:
- 1 Jupyter Pod
- 1 Scheduler Pod
- 2 Workers pod (forced to be in different servers)

In our deployment strategy, the number of workers must be <= the number of servers dedicated to HTCondor processing.

WORKER SECTION

```yaml
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
            - key: htcondor
              operator: In
              values:
                - "true"
```

The servers dedicated to the htcondor cluster were labeled with "*htcondor*". And here the deployment is tageted to the servers with such label.

```yaml
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
            - key: htcondor
              operator: In
              values:
                - "true"
        topologyKey: kubernetes.io/hostname
```

Here the workers were forced to be deployed on different servers to take advantage of all CPU computational power present.

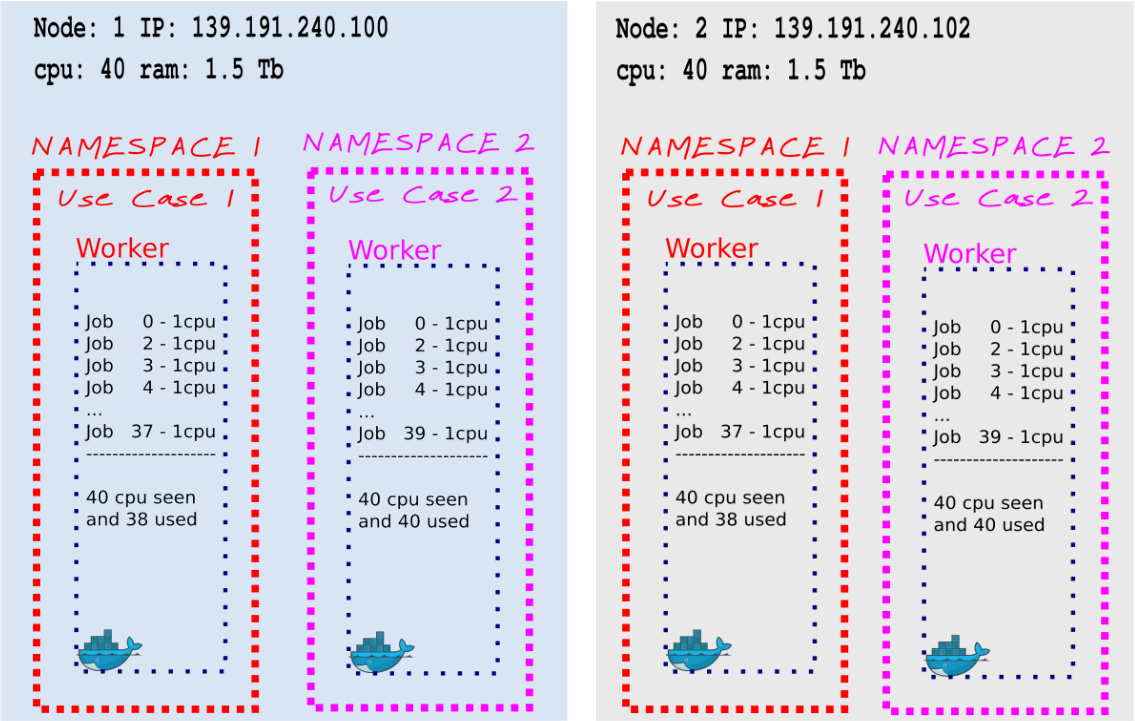
HTCondor submit file

the two namespaces have different input list
condor1.lst for the first namespace
condor2.lst for the second namespace

```
universe                = vanilla
executable              = ./start_process_sen2corfullcheck-bda.sh
transfer_input_files    = /<persistent storage>/lst/condor.lst, /<persistent storage>/aux_files/SYMB.sh
notification            = Error
arguments               = $(Process) $(Cluster) /<persistent storage>/lst/condor3.lst 2 /<persistent storage>/out
max_retries             = 3
Periodic_hold           = (time() - JobStartDate) > 10800
periodic_hold_reason    = "ERROR: long time in the queue"
should_transfer_files   = YES
when_to_transfer_output = ON_EXIT
output                  = /<persistent storage>/logs/job$(Process)_$(Cluster).out
error                   = /<persistent storage>/logs/job$(Process)_$(Cluster).err
log                     = /<persistent storage>/logs/job$(Process)_$(Cluster).log
request_cpus            = 1
JobBatchName            = Sen2cor_kubernetes-Htcondor_test
queue                   40
```

Working schema in K8S: Workers

2 NAMESPACES with 2 WORKERS each deployed on 2 KUBERNETES NODES



All namespace have workers on every node where HTCondor is allowed access and they share concurrently the resources of the same node.

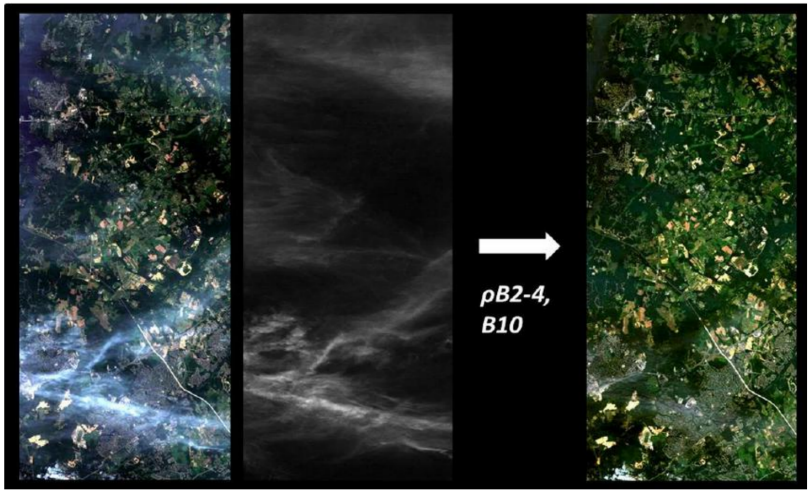
Particularly **they share CPU time** and **take advantage of some idle time** in the script running.

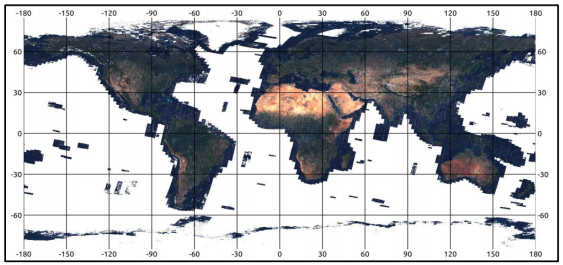
This way **the total performance of all submitting increase** of some percent point and more important there was any detrimental interaction between use cases/namespaces.

In this test, we have not used any priority or limits on resources and in case needed they could easly be applied at the Namespace level.

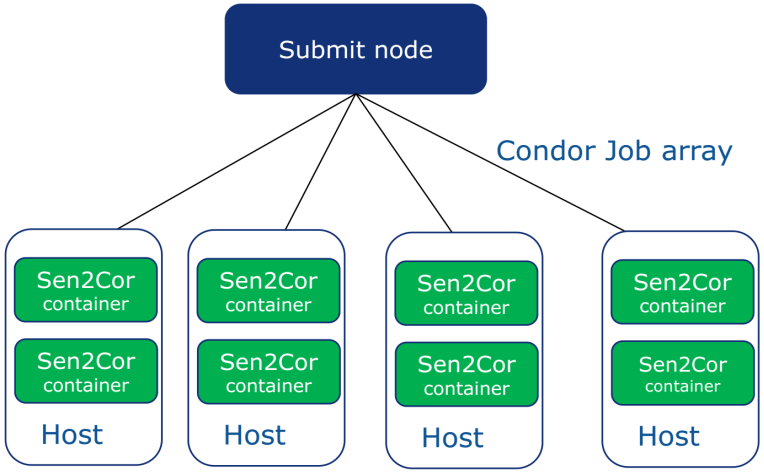
JEODPP USE CASE: Atmospheric correction of Sentinel-2 L1C Data at Global level

Sen2Cor is a processor for Sentinel-2 Level 2A product generation and formatting; it performs the atmospheric-, terrain and cirrus correction of Top-Of- Atmosphere Level 1C input data





- Pre-processing of the 0.5 millions products resulting in a volume of ~ 1 Petabyte;
- with ~940 concurrent jobs and 0.6 millions jobs in queue.
- One JOB takes in average 30min to complete
- More than one container by host. Dynamic slots



Working schema in K8S: Benchmark on performance

In order to show that K8S is capable of running several instances of HTCondor sharing the same servers and resources, two scenarios were created:

1 running a single Condor instance into a namespace in K8S;

2 running two Condor instances into two namespaces sharing the same resources used for the first scenario.

The purposes of this test is:

- to prove that in the second scenario the two instances do not interfere with each other;
- to compare performances between the namespace 1 in the two scenarios. Between one working alone Vs one working sharing resources.

Working schema in K8S: Benchmark on performance

One namespace running Vs. **Two namespaces sharing the same resources**

First scenario

One namespace running into 2 workers

Namespace1
time to complete 38 jobs
total: **32 hours** 00 minutes
average: 50 min 31 sec

Second scenario

Two namespaces running into 2 workers each

Namespace1
time to complete 38 jobs
total: **57 hours** 45 minutes
average: 1 hour 31 min

Namespace2
time to complete 40 jobs
total: 61 hours 15 minutes
average: 1 hour 31 min

Comparing the sum of time for jobs in the namespace1 in the two scenarios we have: **32 hours compared to 58 hours**

In the second scenario, namespace1 takes 180.50% over the time used in the first scenario where the resources were not shared, that is LESS than the expected.

The gain on performance highly depends on the particularities of the code running in the submit.

Generally, the CPU time is the bottleneck and so more the running script has idle time in CPU usage more it could be the gain in performance.

The sen2cor is a java script with high cpu usage but also some IO time and it process 1 satellite image with 1 cpu.