Institut de Recherche
en Informatique de Toulouse

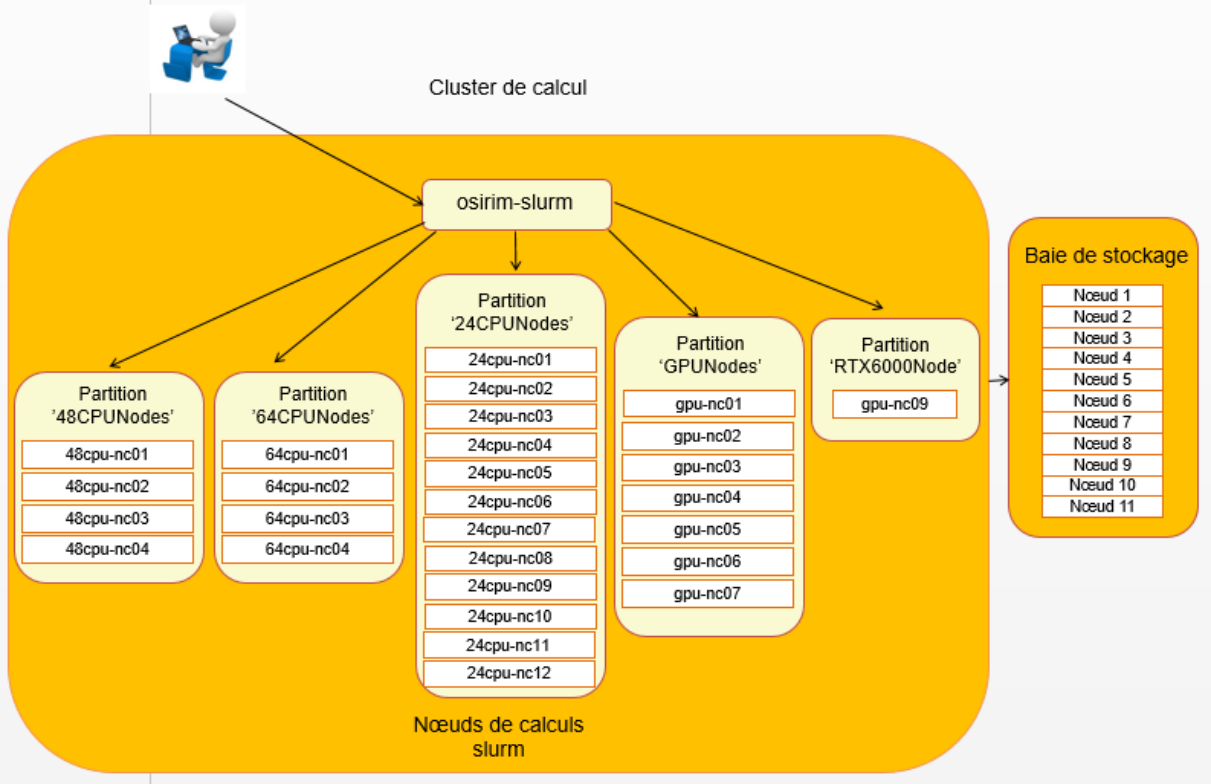
Using Slurm on OSIRIM

2020/07/16
V7.1

Table of contents

Introduction

The following figure shows the architecture of the Simple Linux Utility for Resource Management (SLURM) cluster on the OSIRIM platform



The cluster consists of :

An interactive node (osirim-slurm.irit.fr)

This is the node on which you must connect to access the compute cluster. This node (on Linux Centos7) can be used to validate programs before running them on the compute cluster. Since this node is shared among all users, it must not be used for running long jobs.

Compute nodes

These nodes (on Linux Centos7) are servers dedicated to calculations. The SLURM Job Manager manages on the compute nodes the distribution and execution of the processes that you launch from the interactive node. A process running on a compute node accesses data hosted on the storage array, performs a process and records the result on the array.

The calculation nodes are divided into 4 categories:

- 12 compute nodes bi processors Intel Xeon Gold 6136 3 Ghz, of 24 processors and 192 GB of RAM each. These nodes are grouped in a Slurm partition which we named "24CPUNodes". This partition will be adapted in most cases of use. Each process will however be limited to 24 threads and / or 180 GB of RAM. On the other hand, the number of processes created by a Job, a Job Step or a Task is limited only by the total size of the partition (and the availability of resources) : for example, a single Job will be able to execute, in parallel, 3 Steps of 2 Tasks each, each Task creating 24 threads. This Job will use 144 CPUs and will be distributed over 6 nodes. "24CPUNodes" is the default partition.

- 4 compute nodes bi processors AMD EPYC 7402 2,8 Ghz, of 48 processors and 512 GB of RAM each. These nodes are grouped in a Slurm partition that we named "48CPUNodes". This partition will be suitable for Jobs requiring more than 24 threads and / or 192 GB of RAM for the same process.

- 4 compute nodes of 64 processors and 512 GB of RAM each (Quadri Processors AMD Opteron 6262HE, 16 cores x 1,6Ghz). These nodes are grouped in a Slurm partition named "64CPUNodes". Although equipped with old generation processors, this partition will be suitable for jobs taking advantage of a parallelization on 64 cores.

- 7 compute nodes each with 4 Nvidia Geforce GTX 1080TI graphics cards. These nodes are grouped in a Slurm partition that we named "GPUNodes". This partition is intended for processing taking advantage of the computing power provided by GPU processors. The Deep Learning frameworks (TensorFlow, Theano, Pytorch, ...) are the perfect example.

- 1 compute node with 3 Nvidia Quadro RTX6000 graphics cards (24 Go RAM). This node is in a Slurm partition that we named "RTX6000Node". This partition is intended for processing taking advantage of the computing power provided by GPU processors and large memory size.

A storage array

With a capacity of approximately 1 Po, this storage is provided by an Isilon bay composed of 11 nodes. The data is accessible from the interactive node and the compute nodes via the NFS protocol.

Login to OSIRIM

As a user of a project hosted on the OSIRIM platform, you have an account on this platform.

The connection to this platform is done using the SSH protocol (Secure Shell) from your workstation under Linux, Windows or Mac environment.

The connection is made on the osirim-slurm.irit.fr server via SSH on port 22.

On /Linux :

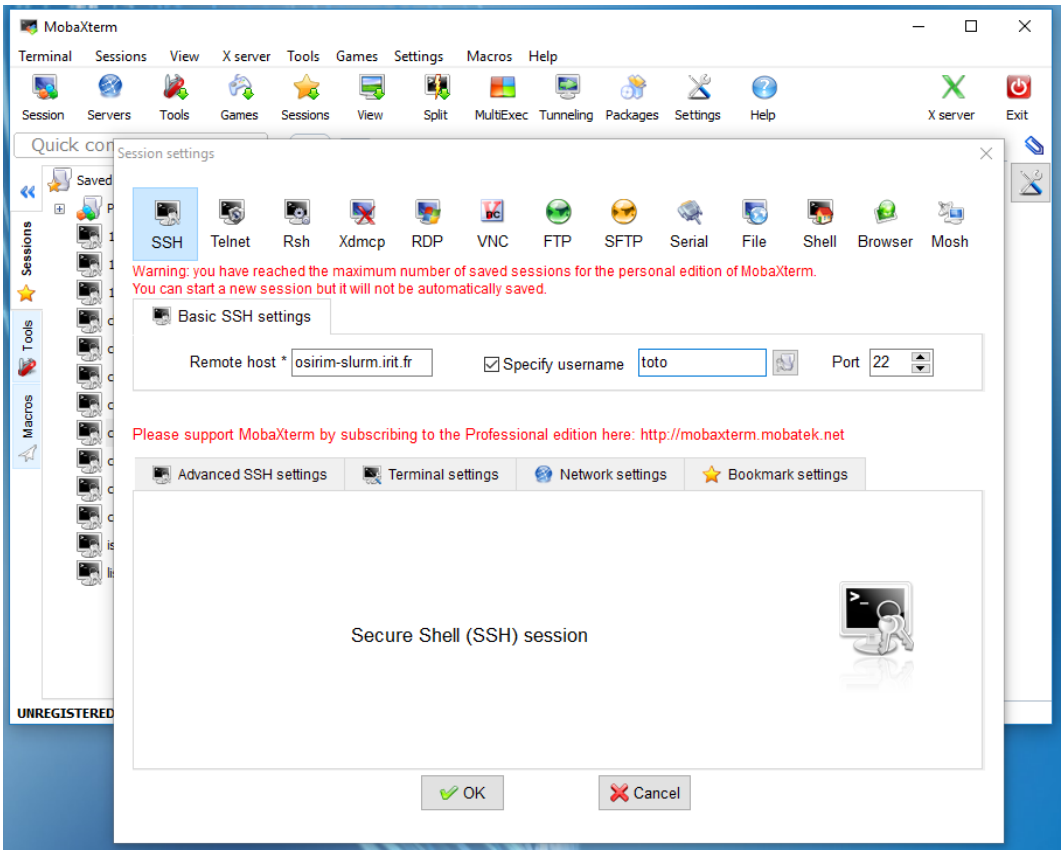
Open a console, and connect to the osirim-slurm.irit.fr server using the ssh command and specifying port 22 (default) :

ssh user@osirim-slurm.irit.fr

Then enter your password :

On Windows :

Use the MobaXterm client (http://mobaxterm.mobatek.net/) specifying the server address (osirim-slurm.irit.fr) and port 22 (default).



Once the connection is established on osirim-slurm.irit.fr, you can start using the platform (under Linux CentOS7). You can save your data in the storage area dedicated to your project, create a script, launch a job on the computing cluster, ...

Each user of the platform has a home user located under /users/ ... /

This user directory is not made to store binary programs, corpora and results ... you just store your program (c, python, java ..), your documentation ... if you wish.

The data is organized by "projects" under the directory /projects/ ...

You can save your data in the storage area dedicated to your project (example: /projects/melodi/ ...), create a script, launch a job on the computing cluster, ...

It is in this directory that you have to manage your data, corpus and the results of your treatments.

Software currently available on the platform is either installed at the system level or made available in /logiciels/ .

This list evolves throughout Osirim's life. You can use the command "module av" to consult the list of applications available via module.

Notes / System reminders

Process, CPUs and parallelization :

A program or process running on a machine will not automatically use all available cores / CPUs. **For a program to exploit multiple cores / CPUs, it must**:

- have been designed to be able to perform different tasks in parallel
- explicitly executes these tasks on different threads or subprocesses in order to run on multiple cores / CPUs.

NB : Many specific frameworks (written in Java, Lua, R ...) support this parallelization (but not always) and make it transparent for the user. However, be careful to configure (if possible) the number of threads or workers generated by the framework and to match the number of CPUs you reserve to optimize the total execution time.

Notes on running a program on a calculation server :

Running a program on a compute server is non-interactive :

- No entry: the user does not have the possibility to enter anything or answer a request of the program (no input, standard input deactivated). It must run completely autonomously..
- Screen output: the display returned by the program (output, standard output). is accumulated in a text file (whose name and location can be set in the batch) and will be accessible once the job has been completed.

Environment variables :

Retrieving the value of a "VAR" environment variable in ...

Shell/Bash/[T]CSH:	$VAR
Python:	os.environ.get('VAR') [OR] os.getenv('VAR')
Java:	System.getenv("VAR")
Lua	os.getenv(VAR)

Slurm Batch :

A Slurm Batch is a file that describes a request for resource allocation for the execution of a job. It consists of two parts :

- Job parameters via SBATCH options written in the form of Shell comments (Bash, TCSH ...). These options make it possible to specify the requested resources (CPUs, RAM, time ...), the name of the Job, the location of the output file, the email address for the notifications ...
- The treatment (shell script).

A Slurm Job can be structured to be instantiated in large numbers (Arrays), be split into Steps and can execute one or more Task (s). These notions are discussed later.

Batch example: Simple job

This first example shows the execution of a simple process (a single Step, implicit, of a single Task, implicit).

Job Description :

Encoding a video file. The encoding will be "multi-threaded" using "ffmpeg" and the "-threads" option.

Batch content (job.sh):

```
# SBATCH options :

#SBATCH --job-name=Encode-Simple    # Job name
#SBATCH --cpus-per-task=4           # 4 CPUs par Task allocation

#SBATCH --mail-type=END             # email notification at the
#SBATCH --mail-user=bob@irit.fr     # end of the job execution.
#SBATCH --partition=24CPUNodes

# Treatment
module purge                    # delete all loaded module environments
module load ffmpeg/0.6.5        # load ffmpeg module version 0.6.5

ffmpeg -i video.mp4 -threads $SLURM_CPUS_PER_TASK [...] video.mkv
```

Comments :

- It is not mandatory to specify the memory required per CPU in the batch. By default, each Job automatically has a RAM allocation depending of the partition
  - 48CPUNodes partition : 8000 Mo RAM allocation per CPU (4 CPUs will get 8000*4 = 32000 Mo ≈ 32 Go)
  - 24CPUNodes partition : 7500 Mo RAM allocation per CPU (4 CPUs will get 7500*4 = 30000 Mo ≈ 30 Go)
  - 64CPUNodes partition : 4096 Mo RAM allocation per CPU (4 CPUs will get 4096*4 = 16384 Mo ≈ 16 Go)
  - GPUNodes partition : 9000 Mo RAM allocation per CPU (4 CPUs will get 9000*4 = 36000 Mo ≈ 36 Go)
  - RTX6000Node partition : 4500 Mo RAM allocation per CPU (4 CPUs will get 4500*4 = 18000 Mo ≈ 18 Go)

- The SBATCH option "ntasks" is not necessary here because 1 is the default value.
- The selection of the partition to be used is done with the option SBATCH "partition":

```
#SBATCH --partition=24CPUNodes
```

- The environment variable "SLURM_CPUS_PER_TASK" contains the value of the SBATCH option "cpus-per-task" and is passed to ffmpeg to always use as many threads for encoding as CPUs available for the Job. Other Slurm environment variables are available ! (complete list)

Environment variable	Value (Corresponding #SBATCH option)
SLURM_JOB_PARTITION	Used partition (--partition)
SLURM_JOB_NAME	Job name (Warning, unlike the output / error options, the variables% j /% N ... are not replaced !)
SLURM_NTASKS	Number of Tasks (--ntasks)
SLURM_CPUS_PER_TASK	CPUs per Task (--cpus-per-task)
SLURM_JOB_NUM_NODES	Number of nodes requested / deducted (--nodes)
SLURM_JOB_NODELIST	List of nodes used (--nodelist)
…	

Execution of the batch :

The Batch is transmitted to Slurm via the "sbatch" command which, unless error or refusal, creates a Job and places it in the queue.

```
[bob@co2-slurm-client ~]$ sbatch job.sh
```

Batch Example : Job Arrays

To process a large number of files, it is not necessary to generate a batch per file (or group of files) to be processed; Slurm manages it via **Job Arrays**. They allow, in a single Batch file, to generate a large number of similar jobs and to set the number of jobs executed in parallel (for example, processing 10,000 files per batch of 50 maximum simultaneously).

A Job Array is created by simply adding the SBATCH option "--array" (or "-a"). The option accepts a list of indices (or an interval with, optionally, an increment), **and the maximum number of Jobs to run in parallel**. **Without this maximum, Slurm will run, by default, as many jobs as possible (depending on available resources). To limit the number of jobs in parallel, do not forget to specify a maximum !**

The choice of indices used in the array is free (arbitrary and values not necessarily continuous). The important thing is to **choose indices that will identify the Job** and thus select the resource (s) to use (file, database ID, ...).

Slurm makes this index accessible in the Batch by the environment variable "SLURM_ARRAY_TASK_ID".

Usage : `#SBATCH --array=<start>-<end>[:<step>][%<maxParallel>]`

Or : `#SBATCH --array=<list>[%<maxParallel>]`

Examples of use :

- `#SBATCH --array=0-15` `# 15 jobs (indices fom 0 to 15 included).`
- `#SBATCH --array=10-16:2` `# 4 jobs (indices : 10,12,14,16).`
- `#SBATCH --array=2,3,5,7,11,13` `# 6 jobs.`
- `#SBATCH --array=1-10000%32` `# 10 000 jobs, 32 jobs max in //`

Slurm Documentation – Job Arrays

Job Description :

The example below uses a Job Array to encode 5,000 videos in a batch of up to 8 in parallel. The indices chosen for the Array reflect the naming of the files to be processed (video- <index> .mp4).

Batch content (job.sh):

```
# SBATCH options :

#SBATCH --job-name=Encode-Batch      # Job name
#SBATCH --cpus-per-task=4            # Allocation of 4 CPU per Task

#SBATCH --mail-type=END              # email notification at the end
#SBATCH --mail-user=bob@irit.fr      # of the job execution.
#SBATCH --array=1-5000%8             # 5000 Jobs, 8 max in parallel

# Treatment
module purge                    # delete all loaded module environments
module load ffmpeg/0.6.5        # load ffmpeg module version 0.6.5

ffmpeg -i video-$SLURM_ARRAY_TASK_ID.mp4 -threads
$SLURM_CPUS_PER_TASK [...] video-$SLURM_ARRAY_TASK_ID.mkv
```

Comments :

- Each encoding (Task) using 4 CPUs, the Job Array will monopolize up to 8x4 or 32 CPUs simultaneously.
- The file to be encoded is determined according to the index of the Job Array.

Execution of the batch :

The Batch is transmitted to Slurm via the "sbatch" command which, unless error or refusal, creates a Job and places it in the queue.

```
[bob@co2-slurm-client ~]$  sbatch job.sh
```

Batch example: Job Steps and Tasks

Jobs Steps allow you to split a job into several logical sections. They are created by prefixing a command (script / program) with the command Slurm "srun" and can run sequentially and / or in parallel. For example, a batch may consist of 3 successive steps, each divided into 2 parts executed in parallel.

For this, Slurm uses an "allocation unit": the Task. A Task is a process with "cpus-per-task" CPUs.

A Step ("srun") uses one or more Task (s) (option "-n"), executed on one or more Node (s) (option "-N"). If omitted these options use, by default, the entire allocation of the Job.

The resources of a Job are then expressed in "cpus-per-task" and "ntasks" (number of Tasks) for a total CPU allocation of the Job of : cpus-per-task * ntasks.

Job description:

In this example, the Job encodes a video in two successive steps: a first stage of preparation (for example, copying of files, cutting of the video, 1st encoding pass ...), and a second stage which carries out two encodings (in H264 and VP9) executed in parallel. The resources to reserve for this Job are 2 Tasks of 4 CPUs each.

Batch content :

```
# SBATCH options:

#SBATCH --job-name=Encode-Steps     # Job Name
#SBATCH --cpus-per-task=4           # Allocation of 4 CPUs per Task
#SBATCH --ntasks=2                  # Number of Tasks : 2

#SBATCH --mail-type=END             # email notification at the
#SBATCH --mail-user=bob@irit.fr     # end of the job execution.

# Treatment
module purge                    # delete all loaded module environments
module load ffmpeg/0.6.5        # load ffmpeg module version 0.6.5

# first stage : Step of 2 Tasks (global ressources of the Job)
srun prep.sh

# second stage : 2 Steps in parallel (one Task per Step)
srun -n1 -N1 ffmpeg -i video.avi -threads $SLURM_CPUS_PER_TASK -c:v
libx264 […] -f mp4 video-h264.mp4 &

srun -n1 -N1 ffmpeg -i video.mp4 -threads $SLURM_CPUS_PER_TASK -c:v
libvpx-vp9 […] -f webm video-vp9.webm &


# wait the end of « childs » Steps (running in background)
wait

# third stage : finalisation (2 Tasks)
srun finish.sh
```

Comments :

- CPUs not used by a Task will be "lost", not usable by any other Task or Step. If the Task creates more processes / threads than allocated CPUs, these threads will share the CPUs. (see Process ...)
- One of the advantages of using Steps for iterative tasks (not executed in parallel) is their support in the jobs management functions (sstat, sacct) allowing both a step-by-step progress tracking. Job step during execution (Steps completed, in progress, their duration ...), and detailed statistics of resource use (CPU, RAM, disk, network ...) for each Step (after execution).
- When Steps are executed in parallel, it is imperative in the parent script (Job) to wait until the execution of the child processes with a "wait", otherwise they will be automatically interrupted (killed). once the end of the Batch is reached..
- The parallelization of the Steps is done by the SHELL ('&' at the end of the line), which executes the command "srun" in a sub-process (sub-shell) of the Job.
- A Task can not be run / distributed on multiple nodes; the number of Tasks must always be greater than or equal to the number of nodes (in the Batch as in a Step).

Steps Create Bash Structures (Bash) based on data source :

```
# Loop on the elements of an array (here files):
files=('file1' 'file2' 'file3' ...)
for f in "${files[@]}"; do
      # Adapt "-n1" and "-N1" according to your needs
      srun -n1 -N1 <command> [...] "$f" &
done


# Loop on files in a directory:
while read f; do
      # Adapt "-n1" and "-N1" according to your needs
      srun -n1 -N1 <command> [...] "$f" &
done < <(ls "/path/to/files/")
# Use "ls -R" or "find" for a recursive file path


# Reading a line by line of a file:
while read line; do
      # Adapt "-n1" and "-N1" according to your needs
      srun -n1 -N1 <command> [...] "$line" &
done <"/path/to/file"
```

Processing large volume of files

Although Job Steps and Job Arrays both allow for mass and parallel processing, their implementation and operation are very different :

- Job Arrays are very simple to set up (only one SBATCH "array" option to be added) and manage the quantity of Jobs to run simultaneously while Job Steps need to do the job manually (iteration on sources, creation steps in the background ...).
- A Job Array is a collection of Jobs. As a result, Jobs are executed individually based on available resources; As soon as one of the Jobs in the Array ends, Slurm immediately executes the next one, which reduces the total execution time and optimizes the use of resources. Job Steps, on the other hand, require that all requested resources are all available to run the Job.

To perform similar processing on a large number of sources, use Job Arrays.

If, on the other hand, the processing to be performed requires running very different processes in parallel (not just a source / file issue) and / or it is not possible to select a source from an index, use Jobs Steps.

It is also quite possible to use both at the same time!

Execution of the "job.sh" Batch :

The Batch is transmitted to Slurm via the "sbatch" command which, unless error or refusal, creates a Job and places it in the queue.

```
[bob@co2-slurm-client ~]$ sbatch job.sh
```

Using Machine Learning and Deep Learning frameworks :

Singularity containers :

Machine learning and deep learning frameworks are available on Osirim as containers.
To allow the use of multiple versions of Cuda / CuDNN / Python and to avoid dependency issues and conflicts between machine learning libraries, each framework runs in a dedicated container. Although Docker is the most widely used containerization solution, we use 'Singularity' on Osirim, a containerization solution for compute clusters.
https://www.sylabs.io/docs/

Singularity (.SIF) images of frameworks are built on Ubuntu 16.04 and CUDA 9, Ubuntu 18.04 and CUDA 10, CUDA11, available under /logiciels/containerCollections/
The directories 'Recipe Files' contain the files which made it possible to build the images (the equivalent of docker files)

```
/logiciels/containerCollections/
├── CUDA11
│   ├── tf2-NGC-20-06-py3.sif (tensorflow2, june 2020)
│   ├── tf1-NGC-20-06-py3.sif (tensorflow1, june 2020)
│   ├── pytorch-NGC-20-06-py3.sif (june 2020)
├── CUDA10
│   ├── keras-tf.sif
│   ├── keras-tf-NGC-19-07.sif (tensorflow1, july 2019)
│   ├── tf2-NGC-19-11-py3.sif (tensorflow2, november 2019)
│   ├── tf2-NGC-20-03-py3.sif (tensorflow2, march 2020)
│   ├── tf1-NGC-20-03-py3.sif (tensorflow1, march 2020)
│   ├── pytorch.sif
│   ├── pytorch-NGC-19-07.sif (july 2019)
│   ├── pytorch-NGC-19-11-py3.sif (november 2019)
│   ├── pytorch-NGC-20-03-py3.sif (march 2020)
│   ├── PGI-NGC-19-04.sif (april 2019)
│   ├── Kaldi-19-03-V7.sif (march 2019)
│   ├── README
│   ├── Recipe Files
│   │   ├── keras_tf_RecipeFile.def
│   │   ├── pytorch_RecipeFile.def
│   │   ├── tf_RecipeFile.def
│   │   └── vanilla_10.0_RecipeFile.def
│   ├── tf.sif
│   └── vanilla_10.0.sif
└── CUDA9
    ├── keras-tf.sif
    ├── keras-th.sif
    ├── pytorch.sif
    ├── pytorch_1.0.1.sif
    ├── README
    ├── Recipe Files
    │   ├── keras_tf_RecipeFile.def
    │   ├── keras_th_RecipeFile.def
    │   ├── pytorch_RecipeFile.def
    │   ├── pytorch_1.0.1_RecipeFile.def
    │   ├── tf_RecipeFile.def
    │   ├── th_RecipeFile.def
    │   ├── vanilla_9.0_RecipeFiles.def
    │   └── vanilla_9.2_RecipeFiles.def
    ├── tf.sif
    ├── th.sif
```

```
├── vanilla_9.0.sif
└── vanilla_9.2.sif
```

NGC images are images built from Nvidia docker containers available at https://ngc.nvidia.com

A more precise description of the content of each image can be found on the page https://osirim.irit.fr/site/en/articles/utilisation_gpu

To learn more about the software installed and available in each container, we invite you to consult the README file in the CUDA9 and CUDA10 directories.

Execution of a singularity image:

The execution of a singularity container is done via the command 'singularity exec' followed by the image of the container and the processing to be executed in the container.

Example :

```
module load singularity/3.0.3
cd /logiciels/containerCollections/CUDA10/
singularity exec ./tf2-NGC-20-03-py3.sif $HOME/moncode.sh
```

Note that the user environment variables are available in the containers as well as the /users /projects and /logiciels directories

Using Frameworks in a Slurm batch:

The available frameworks have the ability to run on CPUs or GPUs. So you can run the containers with Slurm on any partition of OSIRIM: '24CPUNodes', '48CPUNodes', '64CPUNodes', 'GPUNodes' or 'RTX6000Node'.

In the examples below, we want to take advantage of the GPUs, and execute the processing on the 'GPUNodes' partition. To tell Slurm that we want to use GPUs, 2 parameters must be mentioned:

```
#SBATCH --gres=gpu:1   (the number of cards you want to use, 4 max per server)
#SBATCH --gres-flags=enforce-binding
```

Examples of framework usage:

Content of slurm_job_tf.sh for running Tensorflow:

```
#!/bin/sh

#SBATCH --job-name=GPU-Tensorflow-Singularity-Test
#SBATCH --output=ML-%j-Tensorflow.out
#SBATCH --error=ML-%j-Tensorflow.err
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --partition=GPUNodes
#SBATCH --gres=gpu:1
#SBATCH --gres-flags=enforce-binding

module purge
module load singularity/3.0.3
```

```
srun singularity exec /logiciels/containerCollections/CUDA10/tf2-NGC-
20-03-py3.sif python "$HOME/tf-script.py"
```

Execution : [bob@co2-slurm-client ~]$ sbatch slurm_job_tf.sh

Content of slurm_job_ke.sh for running Keras on theano

```
#!/bin/sh

#SBATCH --job-name=GPU-keras-Singularity-Test
#SBATCH --output=ML-%j-keras.out
#SBATCH --error=ML-%j-keras.err
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --partition=GPUNodes
#SBATCH --gres=gpu:1
#SBATCH --gres-flags=enforce-binding

module purge
module load singularity/3.0.3
srun singularity exec /logiciels/containerCollections/CUDA9/keras-
th.sif python3 "$HOME/tf-script.py"
```

Execution : [bob@co2-slurm-client ~]$ sbatch slurm_job_ke.sh

Installing additional packages:

You may want to use non-default libraries in the containers available to you.

To install additional packages, you can use virtualenv, pip or conda to do this.

- For Python 2 use the following software: conda2, pip2 and virtualenv2
- For Python 3 use the following software: conda3, pip3 and virtualenv3

Here is the procedure below:

First, you need to create a virtual environment from your $HOME directory, by opening a shell in the container:

```
$ singularity shell /logiciels/containerCollections/CUDA9/tf.sif
```

To create a virtual environment, you must specify a path. For example to create one in the local directory called '`ENVNAME`', type the following:

```
(tf.sif) → $ mkdir $HOME/ENVNAME
```

Then,

```
(tf.sif) → $ virtualenv2 --system-site-packages $HOME/ENVNAME
## or by virtualenv for python 3,
(tf.sif) → $ virtualenv3 --system-site-packages $HOME/ENVNAME
## or by conda for python 2,
(tf.sif) → $ conda2 create -n ENVNAME
## or by conda for python 3,
(tf.sif) → $ conda3 create -n ENVNAME
```

NOTE: For conda (2 or 3) you don't need to create a local directory, all virtual environments will be automatically placed at $HOME/.conda/ .

The **--system-site-packages** flag will allow you to use all packages already installed with the python inside the container (e.g. tensorflow, keras, etc.) in the created virtual environment with the python used by ENVNAME.

Then, once the virtual environment is created, you can install the desired packages locally (example with the Shogun package of Reinforcement Learning Package), using pip or conda canal :

To avoid **"EnvironmentError: [Errno 28] No space left on device"** error , you should set the TMPDIR unix environment variable to be in your $HOME directory instead of the default one, which is the container TMPDIR (tiny space) and take into account that all containers are Read-Only images.

```
(tf.sif) → $ mkdir localTMP
(tf.sif) → $ TMPDIR=$HOME/localTMP
(tf.sif) → $ TMP=$TMPDIR
(tf.sif) → $ TEMP=$TMPDIR
(tf.sif) → $ export TMPDIR TMP TEMP
```

After exporting the TMPDIR, TMP and TEMP, your container environment is ready to start installing new packages using conda or pip.

Using Conda :

```
$ singularity shell /logiciels/containerCollections/CUDA9/tf.sif
#Before any activation you should source conda script
# For conda2
(tf.sif) → $ source /usr/local/miniconda2/etc/profile.d/conda.sh
(tf.sif) → $ conda2 activate  ENVNAME
(tf.sif) → $ conda2 install -c conda-forge shogun

Or

# For conda3
(tf.sif) → $ source /usr/local/miniconda3/etc/profile.d/conda.sh
(tf.sif) → $ source activate  ENVNAME
(tf.sif) → $ conda3 install -c conda-forge shogun
```

Or using pip:

```
$ singularity shell /logiciels/containerCollections/CUDA9/tf.sif
# For pip2
(tf.sif) → $ source ENVNAME/bin/activate
(tf.sif) → $ pip2 install shogun-ml --user

Or

# For pip3
(tf.sif) → $ source ENVNAME/bin/activate
(tf.sif) → $ pip3 install shogun-ml --user
```

NOTE : The **--user** flag will allow you to install all packages in your local activated python environment ENVNAME not inside the container .

Finally, you can use the new packages in your processes from an SLURM job:

Execution : [bob@co2-slurm-client ~]$ sbatch slurm_job_shogun.sh

Content of slurm_job_shogun.sh:

```
#!/bin/sh

#SBATCH --job-name=GPU-shogun-Singularity-Test
#SBATCH --output=ML-%j-shogun.out
#SBATCH --error=ML-%j-shogun.err

#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --partition=GPUNodes
#SBATCH --gres=gpu:4
#SBATCH --gres-flags=enforce-binding

module purge
module load singularity/3.0.3
srun singularity exec /logiciels/containerCollections/CUDA9/tf.sif
$HOME/ENVNAME/bin/python "$HOME/shogun-script.py"

#OU, sous conda2

srun singularity exec /logiciels/containerCollections/CUDA9/tf.sif
$HOME/conda/envs/ENVNAME/bin/python "$HOME/shogun-script.py"
```

IMPORTANT :
If you create a virtual python environment, with a specific python version using conda (e.g. conda2 create -n ENVNAME **python=2.7**), then you need to set all needed packages from scratch, like **Tensorflow** even if you are using **Tensorflow** singularity image (e.g. /logiciels/containerCollections/CUDA9/tf.sif).

Special Case : Installing Python Packages from GitHub

After activating your virtual python environment using conda or virtualenv, then you must set the path to the non-installed packages by the channel "pip" or "conda" using the Unix environment variable **PYTHONPATH** inside the container. To do this type the following:

```
(tf.sif) → $ source activate ENVNAME
(tf.sif) → $ mkdir mesPackagesGithub
(tf.sif) → $ cd mesPackagesGithub
(tf.sif) → $ wget https://github.com/tensorflow/compression/archive/v1.1.zip
(tf.sif ~/mesPackagesGithub) → $ unzip v1.1.zip
(tf.sif ~/mesPackagesGithub) → $ ls
        drwxr-xr-x  2 fdamoun celdev   820 compression-1.1
```

After finishing the manual "installation" of the package: tensorflow-compression, on ~/mesPackagesGithub. then we can export **PYTHONPATH** to import the new github packge, using two methods:

- By an export inside the Slurm script :

```
#!/bin/sh

#SBATCH --job-name=Multi-CPU-Test
#SBATCH --output=ML-%j-Tensorflow.out
#SBATCH --error=ML-%j-Tensorflow.err
```

```
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --partition=GPUNodes
#SBATCH --gres=gpu:1
#SBATCH --nodelist=gpu-nc04
#SBATCH --gres-flags=enforce-binding

export PYTHONPATH="~/mesPackagesGithub:$PYTHONPATH"

srun [...]
```

- Or in the header of your Python script to be executed

```
import sys
import os
sys.path.append('/users/celdev/fdamoun/mesPackagesGithub/compression-1.1')
import tensorflow_compression as tfc
[...]
```

Using JupyterLab

JupyterLab is offered on Osirim in the singularity images available to you.

To access it, open a browser at https://jupyter-slurm.irit.fr and enter your Osirim login and password.

You will then get a form in which you can choose :
- the singularity image you want to use (pytorch, keras, tensorflow, ...)
- the maximum duration of use of JupyterLab (19 h max)
- the 'GPUNodes' or ‘RTX6000Node’ partition (if you want to use GPU cards) or a CPU partition
- the number of GPU cards to reserve (1 by default), if you use the 'GPUNodes' partition..

The form triggers the launch of a Slurm Job and the execution of JupyterLab in the Singularity image you selected, on one of the compute servers of the chosen partition.

JupyterLab opens in your browser. Since JupyterLab, you have access to your home directory (in the banner on the left) as well as directories /projets/...

The Slurm job will be deleted automatically at the end of the run time you select in the form. You can also stop processing yourself by selecting 'Hub Control Panel' from the 'File' menu, then 'Stop my server'.

Viewing queued jobs

The command **"squeue"** allows to display Jobs in queue.

Official documentation: https://slurm.schedmd.com/squeue.html

Viewing all jobs in a waiting list :

```
$ squeue
```

"squeue" displays the pending jobs of all default users.

Showing your own Jobs only:

```
$ squeue -u <user>
      or (general)
$ squeue -u $(whoami)
```

The $ (whoami) command will return your "username" and serve as a filter for the squeue command.

Displaying information of a specific job in the queue:

```
$ squeue -j1234
```

Customizing the displayed fields:

```
$ squeue -o "%A %j %a %P %C %D %n %R %V"
```

The option "--format" ("-o" in short version) allows to select the fields to display. Refer to the order documentation for a complete list of available fields. This example will display the following fields:

```
JOBID NAME ACCOUNT PARTITION CPUS NODES REQ_NODES NODELIST(REASON)SUBMIT_TIME
```

Viewing the queue of a specific partition:

```
$ squeue -p 24CPUNodes
```

Track Job Progress, Job Steps and Resource Utilization

The **"sacct"** command allows you to obtain a lot of information on Jobs and their Steps.

Official Documentation : https://slurm.schedmd.com/sacct.html

The following command displays the default information for Job # 1234:

```
$ sacct -j1234
```

```
       JobID    JobName  Partition    Account  AllocCPUS      State ExitCode
------------ ---------- ---------- ---------- ---------- ---------- --------
1234         slurm-job+ 24CPUNodes       test          6    RUNNING      0:0
1234.0       slurm-tas+                  test          6  COMPLETED      0:0
1234.1       slurm-tas+                  test          6  COMPLETED      0:0
1234.2       slurm-tas+                  test          6    RUNNING      0:0
```

The first line corresponds to the entire Job and the following lines (JobID followed by a dot '.') Indicate the different Steps of the Job. Step "763.2" (3rd Step of Job 763) is here running (State: RUNNING).

The option "--format" (or "-o", short version) allows to choose the fields to display and their size on the screen (using '%'). Many attributes are available; consult the documentation for the complete list and their meaning.

```
$ sacct -j1234 -o JobID,JobName%-20,State,ReqCPUS,Elapsed,Start,ExitCode
```

```
       JobID              JobName      State  ReqCPUS    Elapsed               Start ExitCode
------------ -------------------- ---------- -------- ---------- ------------------- --------
1234         slurm-job-test-%j       RUNNING        6   00:05:49 2017-02-15T14:55:43      0:0
1234.0       slurm-task.sh         COMPLETED        6   00:01:34 2017-02-15T14:55:43      0:0
1234.1       slurm-task.sh         COMPLETED        6   00:01:31 2017-02-15T14:57:17      0:0
1234.2       slurm-task.sh           RUNNING        6   00:01:02 2017-02-15T14:58:48      0:0
```

The information displayed here is chosen and the JobName field now displays the full name of the Job and Steps.

Display of resource usage statistics (CPU / RAM / Disk ...):

```
$ sacct -j1234 -o JobID,JobName%-20,State,ReqCPUS,Elapsed,UserCPU,CPUTime,MaxRSS,Start

       JobID              JobName      State  ReqCPUS    Elapsed    UserCPU    CPUTime     MaxRSS               Start
------------ -------------------- ---------- -------- ---------- ---------- ---------- ---------- -------------------
1234         slurm-job-test-%j       RUNNING        6   00:06:16  35:55.461   00:37:36            15/02/2017 14:55:43
1234.0       slurm-task.sh         COMPLETED        6   00:01:34  09:02.638   00:09:24     36304K 15/02/2017 14:55:43
1234.1       slurm-task.sh         COMPLETED        6   00:01:31  08:55.011   00:09:06     33128K 15/02/2017 14:57:17
1234.2       slurm-task.sh           RUNNING        6   00:01:02  06:02.144   00:06:18     35128K 15/02/2017 14:58:48
```

Note : Some attributes are only available once Step is complete.

Note 2 : It is possible to modify the date display format by modifying the environment variable SLURM_TIME_FORMAT. The date format used by Slurm is that of the function C "strftime" (http://man7.org/linux/man-pages/man3/strftime.3.html).

The above example uses the French date format (JJ/MM/AAAA hh:mm:ss). To set the date format, the easiest way is to add the following line to the ".bashrc" file of your HOME user:

```
export SLURM_TIME_FORMAT='%d/%m/%Y %T'   # Sets the date / time display format for SLURM commands :JJ/MM/AAAA hh:mm:ss
```

Display for a Job Running Steps in Parallel :

```
$ sacct -j1234 -o JobID,JobName%-20,State,ReqCPUS,Elapsed,Start,ExitCode

       JobID              JobName      State  ReqCPUS    Elapsed               Start ExitCode
------------ -------------------- ---------- -------- ---------- ------------------- --------
1234         slurm-job-test-%j       RUNNING       18   00:00:43 15/02/2017 15:03:38      0:0
1234.0       slurm-task.sh           RUNNING        6   00:00:43 15/02/2017 15:03:38      0:0
1234.1       slurm-task.sh           RUNNING        6   00:00:43 15/02/2017 15:03:38      0:0
1234.2       slurm-task.sh           RUNNING        6   00:00:43 15/02/2017 15:03:38      0:0
```

Note : The allocation is here 18 CPUs instead of the previous 6 (3 Tasks of 6 CPUs are run in parallel).

For those who would like to retrieve this information to process it in a script (and / or format it in another language), sacct has the options "--parsable" and "--parsable2" which return the same information but whose fields are separated by a "pipe" ("|"). In "parsable" mode, field sizes ("% ..") are useless, uncut values are always returned. The difference between these two options is that "--parsable" adds a "|" at the end of the line whereas "--parsable2", no.

```
$ sacct -j1234 -o JobID,JobName,State,ReqCPUS,Elapsed,Start,ExitCode --parsable2

JobID|JobName|State|ReqCPUS|Elapsed|Start|ExitCode
1234|slurm-job-test-%j|RUNNING|6|00:05:49|15/02/2017 14:55:43|0:0
1234.0|slurm-task.sh|COMPLETED|6|00:01:34|15/02/2017 14:55:43|0:0
1234.1|slurm-task.sh|COMPLETED|6|00:01:31|15/02/2017 14:57:17|0:0
1234.2|slurm-task.sh|RUNNING|6|00:01:02|15/02/2017 14:58:48|0:0
```

Note : The "--noheader" option also does not display headers in the result.

FAQ

Q : **How to perform a treatment on the OSIRIM Cluster?**

R : You must log in SSH on the machine "osirim-slurm.irit.fr" (port 22) and send a batch file with the command "sbatch". See the SLURM [documentation] for detailed information.

Q : **My job runs smoothly and my data is processed well. Why does it appear as "Failed" in the end email and the command sacct?**

R : One of the possible reasons is:
The success or failure of any shell command (standard command, script, function, program ...) is determined by its exit code; zero (0) indicates success and "> 0" indicates a failure. In a shell script (such as the Slurm Batch), the exit-code of the script is determined by the exit-code of its last statement or command executed.
For example, if the end of a script is the following:

```
# Tests whether /some/file exists or not
if [[ -f '/some/file' ]]; then
      echo "=> Do something"
fi
```

- If the file exists, the last command is "echo", which will return 0 / Success.
- If it does not exist, the last command executed is the "-f" test of the file that returns 1 / Failed. No other command being evaluated, the entire script will return 1 / Failed. Solution: Add "exit 0" explicitly in cases where the script "succeeds" even if the previous test or command fails.

Q : **I need to launch a lot of similar jobs, how to do it?**
R : The easiest way is to create a Job Array.

Q : **I have requested <N> Tasks from <C> CPUs in my batch but my Job never uses more than <C> CPUs at the same time!**
R : Have you created Job Steps? If more than one Task is requested but no Step is explicitly declared, the total allocation will NOT EXCEED 1 Task (<cpus-per-task> CPUs)!

Q : **I'm trying to run Job Steps in parallel but they run one after the other!**
R : To run Job Steps in parallel, execute the srun command in "background" by adding '&' at the end of the line.

Q : **I'm trying to run Job Steps in parallel but they are not running, the Batch stops immediately!**
R : Once the Job Steps are declared, it is imperative to use the "wait" command so that the parent process (the Job) waits for the completion of the child processes (Steps).

Q : **In my "output" file, Slurm shows me the Warning: "srun: Warning: can't run 2 processes on 4 nodes, setting nnodes to 2."**
R : When running a Job Step with "srun", the "-n" (ntasks) parameter must be greater than or equal to "-N" (nodes), otherwise Slurm displays this warning and reduces -N to -n

Appendix 1: Common SBATCH Options

The "sbatch" command receives its parameters from the command line but also allows their definition via SBATCH "directives" in the form of a comment in the header of the file. Both methods produce the same result but those declared on the command line will have priority in case of conflict. In both cases, these options exist (for the most part) in short and long versions (example: -n or --ntasks).

For more information on "sbatch", see the official documentation at:

https://slurm.schedmd.com/sbatch.html

Extract from the most common SBATCH options (long versions only):

#SBATCH --partition=<part>

Choice of the SLURM partition to use for the job. See section: Partitions.

#SBATCH --job-name=<name>

Defines the name of the job as it will be displayed in the different Slurm commands (squeue, sstat, sacct)

#SBATCH --output=<stdOutFile>

#SBATCH --error=<stdErrFile>

#SBATCH --input=<stdInFile>

#SBATCH --open-mode=<append|truncate>

These options define the input / output redirections of the job (standard input / output / error).

The standard output (stdOut) will be redirected to the file defined by "--output" or, if not defined, a default file "slurm-% j.out" (Slurm will replace "% j" with the JobID).

The error output (stdErr) will be redirected to the file defined by "--error" or, if not defined, to the standard output.

Standard input can also be redirected with "--input". By default, "/dev/null" is used (none / empty).

The "--open-mode" option defines how to open (write) files and behaves like an open / fopen of most programming languages (2 possibilities: "append" to write after the file (if it exists) and "truncate" to overwrite the file each time the batch is run (default value)).

#SBATCH --mail-user=<e-mail>

#SBATCH --mail-type=<BEGIN|END|FAIL|TIME_LIMIT|TIME_LIMIT_50|...>

Allows to be notified by e-mail of a particular event in the life of the job: start of the execution (BEGIN), end of execution (END, FAIL and TIME_LIMIT) ... Consult the Slurm documentation for the complete list of supported events.

#SBATCH --cpus-per-task=<n>

Sets the number of CPUs to allocate by Task. The actual use of these CPUs is the responsibility of each Task (creation of processes and / or threads).

#SBATCH --ntasks=<n>

Defines the maximum number of Tasks executed in parallel.

#SBATCH --mem-per-cpu=<n>

Sets the RAM in MB allocated to each CPU. Default and max values depend on partition :

24CPUNodes : 7500 Mo default, 7500 Mo max

48CPUNodes : 8000 Mo default, 10000 Mo max

GPUNodes : 9000 Mo default, 9000 Mo max

#SBATCH --nodes=<minnodes[-maxnodes]>

Minimum number [-max] of nodes on which to distribute the Tasks.

#SBATCH --ntasks-per-node=<n>

Used in conjunction with --nodes, this option is an alternative to --ntasks that allows you to control the distribution of Tasks on different nodes.

Default values and / or deducted by Slurm:

Without explicit declaration of Job Step (s), only one Task will be created and the parameters "--ntasks", "--nodes", "--nodelist" ... will be ignored.

In general, when "--nodes" is not defined, Slurm automatically determines the number of nodes needed (depending on the use of the nodes, the number of CPUs-by-Nodes / Tasks-per-Node / CPUs -by-Task / Tasks, etc.).

If "--ntasks" is not defined, one Task per node will be allocated.

Note that the number of Tasks of a Job can be defined either explicitly with "--ntasks" or implicitly by defining "--nodes" and "--ntasks-per-node".

If the "--ntasks", "--nodes" and "--ntasks-per-node" options are all set, "--ntasks-per-node" will then indicate the maximum number of Tasks per node.