Institut de Recherche
en Informatique de Toulouse

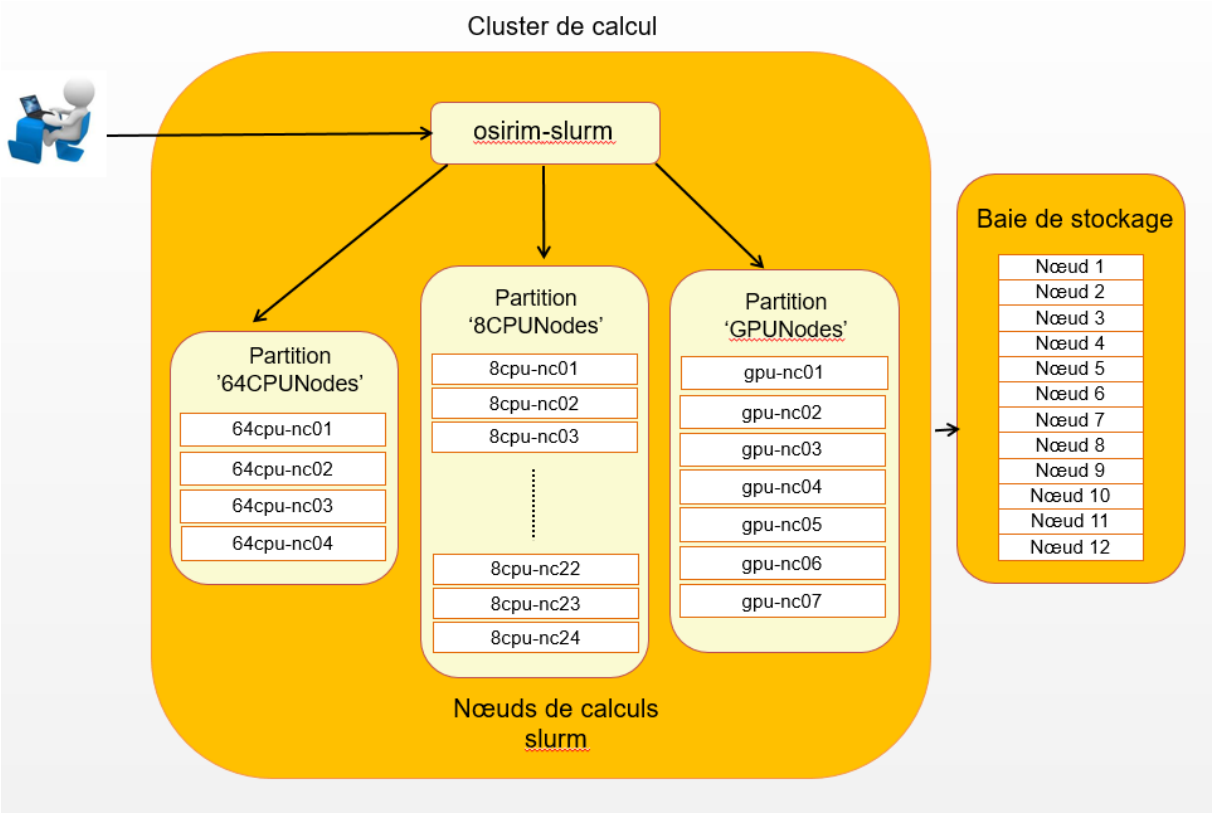
Utiliser Slurm sur OSIRIM

09/04/2018
V4.2

Table des matières

Introduction

La figure suivante présente l'architecture du Cluster SLURM (Simple Linux Utility for Resource Management) sur la plateforme OSIRIM :



Le cluster est constitué :

D'un nœud interactif (osirim-slurm.irit.fr)

C'est le nœud sur lequel vous devez vous connecter pour accéder au cluster de calcul. Ce nœud (sous système Linux Centos7) peut être utilisé pour valider les programmes avant de les lancer sur le cluster de calcul. Ce nœud étant partagé entre tous les utilisateurs, il ne doit pas être utilisé pour l'exécution de jobs longs.

De nœuds de calcul

Ces nœuds (sous système Linux Centos7) sont des serveurs dédiés aux calculs. Le gestionnaire de jobs SLURM gère sur les nœuds de calcul la répartition et l'exécution des traitements que vous lancez à partir du nœud interactif. Un processus qui tourne sur un nœud de calcul accède à des données hébergées sur la baie de stockage, effectue un traitement et enregistre le résultat sur la baie.

Les nœuds de calcul sont répartis en 3 catégories :

- des nœuds de calcul de 8 processeurs et 64 Go de RAM chacun. Au nombre de 24, ces nœuds sont regroupés dans une partition Slurm que nous avons nommée

« 8CPUNodes ». Cette partition sera adaptée dans la majorité des cas d'utilisation. Chaque processus sera cependant limité à 8 threads et/ou 64 Go de RAM. En revanche, le nombre de processus créés par un Job, un Job Step ou une Task n'est limité que par la taille totale de la partition (et la disponibilité des ressources) : un unique Job pourra donc exécuter, en parallèle, 3 Steps de 2 Tasks chacun, chaque Task créant 8 threads. Ce Job utilisera 48 CPUs et sera distribué sur 6 nœuds.
Il s'agit de la partition par défaut

- 4 nœuds de calcul de 64 processeurs et 512 Go de RAM chacun. Ces nœuds sont regroupés dans une partition Slurm que nous avons nommée « 64CPUNodes ». Cette partition sera adaptée aux Jobs nécessitant plus de 8 threads et/ou 64 Go de RAM pour un même processus. Les Jobs qui requièrent beaucoup de RAM et dont les performances d'allocation sont un critère important devraient, selon les cas, bénéficier du passage sur cette partition. La différence ne sera cependant appréciable que pour des allocations de RAM > 32 Go minimum.

- 7 nœuds de calcul dotés chacun de 4 de cartes graphiques Nvidia Geforce GTX 1080TI. Ces nœuds sont regroupés dans une partition Slurm que nous avons nommée « GPUNodes ». Cette partition est destinée aux traitements tirant partie de la puissance de calcul fournie par les processeurs des GPUs. Les frameworks de Deep Learning (TensorFlow, Theano, Pytorch, …) en sont le parfait exemple.

D'une baie de stockage

D'une capacité d'environ 1 Po, ce stockage est assuré par une baie Isilon composée de 12 nœuds. Les données sont accessibles depuis le nœud interactif et les nœuds de calcul via le protocole NFS.

Se connecter sur OSIRIM

En tant qu'utilisateur d'un projet hébergé sur la plateforme OSIRIM, vous disposez d'un compte sur cette plateforme.

La connexion à cette plateforme se fait en utilisant le protocole SSH (Secure Shell) depuis votre poste de travail sous environnement Linux, Windows ou Mac.

La connexion s'effectue sur le serveur osirim-slurm.irit.fr via SSH sur le port 22.

Sous Unix/Linux :

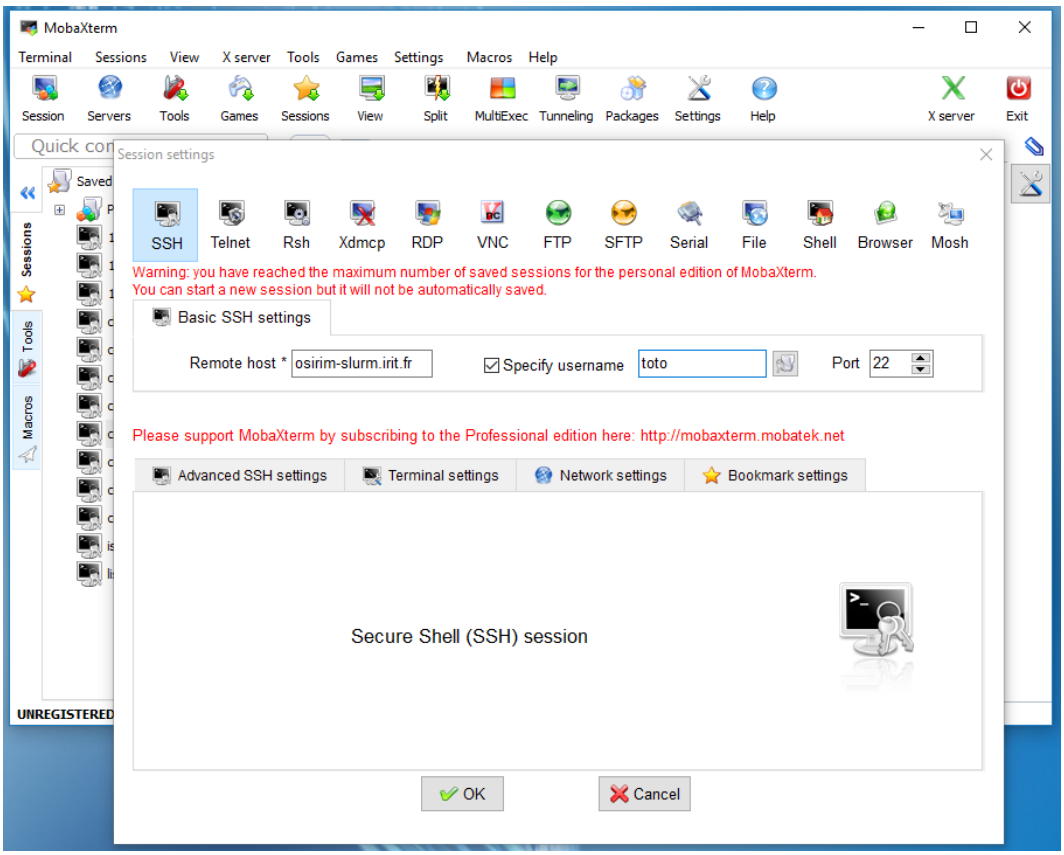
Ouvrez une console, et connectez-vous sur le serveur osirim-slurm.irit.fr en utilisant la commande ssh et en précisant le port 22 (par défaut) :

- **ssh osirim-slurm.irit.fr**

Entrez ensuite votre user / mot de passe

Sous Windows :

Utilisez le client MobaXterm (http://mobaxterm.mobatek.net/) en précisant l'adresse du serveur (osirim-slurm.irit.fr) et le port 22 (par défaut).



Une fois la connexion établie sur osirim-slurm.irit.fr, vous pouvez commencer à utiliser la plateforme (sous Linux CentOS7). Vous pouvez enregistrer vos données dans la zone de stockage dédiée à votre projet, créer un script, lancer un job sur le cluster de calcul, …

Chaque utilisateur de la plateforme possède un home utilisateur situé sous /users/*…/*

Ce répertoire utilisateur n'est pas fait pour stocker les programmes binaires, les corpus et les résultats... vous y stockez juste votre programme (c, python, java..), votre documentation... si vous le souhaitez.

Les données sont organisées par "projets" sous le répertoire /projets/…

Vous pouvez enregistrer vos données dans la zone de stockage dédiée à votre projet (exemple : /projets/melodi/…), créer un script, lancer un job sur le cluster de calcul, …

C'est dans ce répertoire que vous devez gérer vos données, corpus et les résultats de vos traitements.

Les logiciels actuellement disponibles sur la plateforme sont soit installés au niveau système, soit mis à disposition dans /logiciels/ . Vous trouverez la liste des logiciels disponibles sur le serveur Web d'Osirim : http://osirim.irit.fr/site/fr/articles/logiciels-disponibles

Notes / Rappels Système

Processus, CPUs et parallélisation :

Un programme ou processus exécuté sur une machine n'utilisera pas automatiquement tous les coeurs/CPUs disponibles. **Pour qu'un programme puisse exploiter plusieurs coeurs/CPUs, celui-ci doit :**

- avoir été conçu pour pouvoir exécuter différentes tâches en parallèle,
- explicitement exécuter ces tâches sur différents threads ou sous-processus afin d'être exécutés sur plusieurs coeurs/CPUs.

NB : Beaucoup de frameworks spécifiques (écrits en Java, Lua, R...) prennent en charge cette parallélisation (mais pas toujours) et la rendent transparente pour l'utilisateur. Attention cependant à bien configurer (si possible) le nombre de threads ou workers générés par le framework et faire correspondre le nombre de CPUs que vous réservez afin d'optimiser le temps total d'exécution.

Notes sur l'exécution d'un programme sur un serveur de calcul :

L'exécution d'un programme sur un serveur de calcul se fait de manière non-interactive :

- Pas de saisie : l'utilisateur n'a pas la possibilité de saisir quoi que ce soit ou de répondre à une requête du programme (pas d'input, entrée standard désactivée). Celui-ci doit s'exécuter de manière totalement autonome.
- Sortie écran : l'affichage renvoyé par le programme (output, sortie standard). est accumulée dans un fichier texte (dont le nom et l'emplacement sont paramétrables dans le batch) et sera accessible qu'une fois l'exécution du job terminée.

Variables d'Environnement

Récupération de la valeur d'une variable d'environnemnt "**VAR"** en…

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

Batch Slurm :

Un Batch Slurm est un fichier qui décrit une demande d'allocation de ressources pour l'exécution d'un traitement (Job). Il comprend deux parties :

- les paramètres du Job via des options SBATCH écrites sous forme de commentaires Shell (Bash, TCSH...). Ces options permettent de spécifier les ressources demandées (CPUs, RAM, temps...), le nom du Job, l'emplacement du ficher output, l'adresse email pour les notifications...
- le traitement (script shell).

Un Job Slurm peut être structuré pour être instancié en grand nombre (Arrays), être découpé en étapes (Steps) et peut exécuter une ou plusieurs Task(s). Ces notions sont abordées par la suite.

Exemple de Batch : Job simple

Ce premier exemple montre l'exécution d'un traitement simple (un seul Step, implicite, d'une seule Task, implicite).

Description du Job :

Encodage d'un fichier vidéo. L'encodage sera "multi-threadé" en utilisant "ffmpeg" et l'option "-threads".

Contenu du Batch :

```
# Options SBATCH :

#SBATCH --job-name=Encode-Simple    # Nom du Job
#SBATCH --cpus-per-task=4           # Allocation de 4 CPUs par Task

#SBATCH --mail-type=END             # Notification par email de la
#SBATCH --mail-user=bob@irit.fr     # fin de l'exécution du job.


# Traitement

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

Remarques :

- Il n'est pas obligatoire de spécifier la mémoire nécessaire par CPU dans le batch. Par défaut, chaque Job dispose automatiquement d'une allocation RAM de 4096 Mo par CPU demandé (soit 4 Go). 4 CPUs donneront droit à 4096*4 = 16384 Mo ≈ 16 Go

  L'allocation maximum de mémoire par CPU est de 7800 Mo. Si vous avez besoin d'une allocation RAM par CPU supérieure à 4 Go, vous pouvez utiliser dans votre batch l'option SBATCH "`mem-per-cpu`" avec la valeur souhaitée <= à 7800 Mo.

  ```
  #SBATCH --mem-per-cpu=7800
  ```

- L'option SBATCH "`ntasks`" n'est ici pas nécessaire car 1 est la valeur par défaut.
- La sélection de la partition à utiliser se fait avec l'option SBATCH "`partition`" :

  ```
  #SBATCH --partition=64CPUNodes
  ```

- La variable d'environnement "`SLURM_CPUS_PER_TASK`" contient la valeur de l'option SBATCH "`cpus-per-task`" et est transmise à ffmpeg de manière à toujours utiliser autant de threads pour l'encodage que de CPUs disponibles pour le Job. D'autres variables d'environnement Slurm sont disponibles ! (liste complète)

Variable d'environnement	Valeur (Option #SBATCH correspondante)
SLURM_JOB_PARTITION	Partition utilisée (--partition)
SLURM_JOB_NAME	Nom du Job (Attention, contrairement aux options output/error, les variables %j/%N... ne sont pas remplacées !)
SLURM_NTASKS	Nombre de Tasks (--ntasks)
SLURM_CPUS_PER_TASK	Nombre de CPUs par Task (--cpus-per-task)
SLURM_JOB_NUM_NODES	Nombre de nœuds demandés/déduis (--nodes)
SLURM_JOB_NODELIST	Liste des nœuds utilisés (--nodelist)
…	

Exemple de Batch : Job Arrays

Pour traiter un grand nombre de fichiers, il n'est pas nécessaire de générer un batch par fichier (ou groupe de fichiers) à traiter; Slurm le gère via les **Job Arrays**. Ils permettent, dans un seul fichier Batch, de générer un grand nombre de jobs similaires et de paramétrer le nombre de jobs exécutés en parallèle (par exemple, traiter 10 000 fichiers par lot de 50 maximum simultanément).

Un Job Array est créé en ajoutant simplement l'option SBATCH "`--array`" (ou "`-a`"). L'option accepte une liste d'indices (ou un intervalle avec, optionnellement, un incrément), et le **nombre maximum de Jobs à exécuter en parallèle**. **Sans ce maximum, Slurm exécutera, par défaut, autant de jobs que possible (selon les ressources disponibles). Pour limiter le nombre de jobs en parallèle, n'oubliez pas de spécifiez un maximum !**

Le choix des indices utilisés dans l'array est libre (arbitraire et valeurs non nécessairement continues). L'important est de **choisir des indices qui permettront d'identifier le Job** et donc de sélectionner la/les ressource(s) à utiliser (fichier, ID de base de données, …).

> Slurm rend cet indice accessible dans le Batch par la variable d'environnement "`SLURM_ARRAY_TASK_ID`".

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

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

Exemples d'utilisation :

- `#SBATCH --array=0-15` `# 15 jobs (indices de 0 à 15 inclus).`
- `#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 en //`

Documentation Slurm – Job Arrays

Description du Job :

> L'exemple ci-dessous utilise un Job Array pour encoder 5 000 vidéos par lot de 8 maximum en parallèle. Les indices choisis pour l'Array reflètent le nommage des fichiers à traiter (`video-<index>.mp4`).

Contenu du Batch :

```
# Options SBATCH :

#SBATCH --job-name=Encode-Batch      # Nom du Job
#SBATCH --cpus-per-task=4            # Allocation de 4 CPU par Task

#SBATCH --mail-type=END              # Notification par email de la
#SBATCH --mail-user=bob@irit.fr      # fin de l'exécution du job.
#SBATCH --array=1-5000%8             # 5000 Jobs, 8 max en parallèle

# Traitement

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

Remarques :

- Chaque encodage (Task) utilisant 4 CPUs, le Job Array monopolisera au maximum 8x4 soit 32 CPUs simultanément.
- Le fichier à encoder est déterminé en fonction de l'indice du Job Array.

Exemple de Batch : Job Steps et Tasks

Les Jobs Steps permettent de découper un Job en plusieurs sections logiques. Ils sont créés en préfixant une commande (script/programme) avec la commande Slurm "srun" et peuvent s'exécuter séquentiellement et/ou parallèlement. Par exemple, un batch peut être constitué de 3 étapes successives, chacune étant divisée en 2 parties exécutées en parallèle.

Pour cela, Slurm utilise une "unité d'allocation" : la Task. Une Task est un processus disposant de "`cpus-per-task`" CPUs.

Un Step ("`srun`") utilise une ou plusieurs Task(s) (option "-n"), exécutées sur un ou plusieurs Node(s) (option "-N"). Si omises ces options utilisent, par défaut, la totalité de l'allocation du Job.

Les ressources d'un Job sont alors exprimées en "`cpus-per-task`" et "`ntasks`" (nombre de Tasks) pour une allocation CPU totale du Job de : `cpus-per-task` * `ntasks`.

Description du Job :

Dans cet exemple, le Job encode une vidéo en deux étapes successives : une première étape de préparation (par exemple, copie de fichiers, découpage de la vidéo, 1ère passe d'encodage…), et une seconde étape qui effectue deux encodages (en H264 et VP9) exécutées en parallèles. Les ressources à réserver pour ce Job sont donc 2 Tasks de 4 CPUs chacune.

Contenu du Batch :

```
# Options SBATCH :

#SBATCH --job-name=Encode-Steps     # Nom du Job
#SBATCH --cpus-per-task=4           # Allocation de 4 CPUs par Task
#SBATCH --ntasks=2                  # Nombre de Tasks : 2

#SBATCH --mail-type=END             # Notification par email de la
#SBATCH --mail-user=bob@irit.fr     # fin de l'exécution du job.

# Traitement

# 1ère étape : Step de 2 Tasks (ressources globales du Job)
srun prep.sh

# 2e étape : 2 Steps en parallèle (une Task par Step)
srun -n1 -N1 /logiciels/ffmpeg/ffmpeg -i video.avi -threads
$SLURM_CPUS_PER_TASK -c:v libx264 […] -f mp4 video-h264.mp4 &

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


# Attendre la fin des Steps "enfants" (exécutés en arrière plan)
wait

# 3e étape : finalisation (2 Tasks)
srun finish.sh
```

Remarques :

- Les CPUs non utilisés par une Task seront "perdus", non-utilisables par aucune autre Task ou Step. Si la Task crée plus de processus/threads que de CPUs alloués, ces threads se partageront les CPUs. (cf. Processus…)

  Un des intérêts d'utiliser des Steps pour des tâches itératives (non éxécutées en parallèle) est leur prise en charge dans les fonctions de gestion de Jobs (sstat, sacct) permettant à la fois un suivi de l'avancement Step-by-Step du Job pendant l'exécution (Steps terminés, en cours, leur durée…), et des statistiques détaillées d'utilisation des ressources (CPU, RAM, disque, réseau...) pour chaque Step (après exécution).

- Lorsque des Steps sont exécutés en parallèle, il est impératif dans le script parent (Job), d'attendre la fin de l'exécution des processus enfants avec un "wait", faute de quoi ces derniers seront automatiquement interrompus (killed) une fois la fin du Batch atteinte.

- La parallélisation des Steps est réalisée par le SHELL ('**&**' en fin de ligne), qui exécute la commande "srun" dans un sous-processus (sub-shell) du Job.

- Une Task ne peut être exécutée/distribuée sur plusieurs nœuds ; le nombre de Tasks doit donc toujours être supérieur ou égal au nombre de nœuds (dans le Batch comme dans un Step).

Structures Shell (Bash) de création de Steps en fonction de la source des données :

```
# Boucle sur les éléments d'un tableau (ici des fichiers) :
files=('file1' 'file2' 'file3' ...)
for f in "${files[@]}"; do
      # Adaptez "-n1" et "-N1" en fonction de vos besoins
      srun -n1 -N1 <command> [...] "$f" &
done


# Boucle sur les fichiers d'un répertoire :
while read f; do
      # Adaptez "-n1" et "-N1" en fonction de vos besoins
      srun -n1 -N1 <command> [...] "$f" &
done < <(ls "/path/to/files/")
# Utilisez "ls -R" ou "find" pour un parcours récursif des dossiers


# Lecture ligne par ligne d'un fichier :
while read line; do
      # Adaptez "-n1" et "-N1" en fonction de vos besoins
      srun -n1 -N1 <command> [...] "$line" &
done <"/path/to/file"
```

Traitement de gros volume de fichiers

Bien que les Job Steps et les Job Arrays permettent tous les deux de lancer des traitements en nombre et en parallèle, leur mise en place et leur fonctionnement sont très différents :

- Les Job Arrays sont très simples à mettre en place (une seule option SBATCH "array" à ajouter) et gèrent la quantité de Jobs à exécuter simultanément alors que des Job Steps nécessitent d'effectuer le travail manuellement (itération sur les sources, création des steps en arrière plan…).
- Un Job Array est une collection de Jobs. Par conséquent, les Jobs sont exécutés individuellement en fonction des ressources disponibles ; dès qu'un des Jobs de l'Array se termine, Slurm exécute immédiatement le suivant, ce qui réduit le temps total d'exécution et optimise l'utilisation des ressources. Les Job Steps, en revanche, nécessitent que la totalité des ressources demandées soient toutes disponibles pour exécuter le Job.

Pour exécuter un traitement similaire sur un grand nombre de sources, utilisez des Job Arrays.

Si, par contre, le traitement à effectuer nécessite d'exécuter en parallèle des traitements très différents (pas seulement une question de source/fichier) et/ou il n'est pas possible de sélectionner une source à partir d'un index, utilisez des Jobs Steps.

Il est par ailleurs tout à fait possible d'utiliser les deux en même temps !

Exécution du Batch "job.sh" :

Le Batch est transmis à Slurm via la commande "sbatch" qui, sauf erreur ou refus, crée un Job et le place dans la file d'attente.

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

Utilisation du cluster GPU :

Des frameworks de « machine learning » sont disponibles sur 7 serveurs équipés chacun de cartes graphiques Nvidia Geforce GTX 1080TI. Ces serveurs sont accessible dans la partition « GPUNodes ».

Afin de permettre l'utilisation de plusieurs versions de Cuda/CuDNN/Python et éviter les problèmes de dépendances et les conflits entre librairies de machine learning, chaque framework s'exécute dans un conteneur Docker dédié.

Chaque image docker de framework est bâtie sur une image Ubuntu 16.04 dans laquelle les logiciels suivants sont installés :

- CUDA 9
- cuDNN 7
- Miniconda avec numpy, scipy, scikit-cuda, mkl, mkl-service nccl (avec leurs dépendances)
- Julia

Utiliser les frameworks de machine learning dans un batch Slurm :

Comme la commande shell pour exécuter un script d'exécution de framework dans une instance Docker est volumineuse, des commandes shell par framework ont été créées pour faciliter le processus.

Ces commandes sont des liens symboliques vers un script qui réalise les taches suivantes :

- Créée l'instance Docker,
- Positionne le user/group qui exécutera le job,
- Passe les répertoires /users et /projets à l'instance de façon à ce que ces répertoires soient accessibles pour les traitements lancés dans l'instance Docker
- Positionne le répertoire de travail Slurm comme répertoire de travail,
- Positionne les variables d'environnement :
  - HOME — Home directory de l'utilisateur
  - SLURM_JOB_ID — Slurm Job Id
  - SLURM_ARRAY_TASK_ID — TaskID du Slurm Job Array courant
  - SLURM_SUBMIT_DIR — Répertoire de travail Slurm (où la commande sbatch est exécutée)

Frameworks disponibles :

Framework	OS	CUDA	cuDNN	Python	Commande
Tensorflow	Ubuntu 16.04	9.0	7	3	tf-py3
Keras (TF)	Ubuntu 16.04	9.0	7	3	keras-py3-tf
pytorch	Ubuntu 16.04	9.0	7	3	pytorch-py3
theano	Ubuntu 16.04	9.0	7	3	theano-py3
Keras (theano)	Ubuntu 16.04	9.0	7	3	keras-py3-th

Problèmes connus :

Comme le script est exécuté dans un conteneur, arrêter le job avec la commande « scancel » termine le job Slurm **MAIS N'ARRETE PAS** le process exécuté sur le GPU. Slurm considèrera que le GPU est disponible pour un nouveau Job, et lancera un nouveau traitement sur ce GPU, alors que l'ancien traitement « zombie » s'exécute toujours.
Pour contourner ce problème, nous avons mis en place une fonction spécifique : si vous avez besoin d'arrêter un job en cours, connectez-vous sur le site web d'Osirim, authentifiez-vous dans la partie intranet, allez sur la page « Etat du cluster Slurm », sélectionnez la partition « GPUNodes », et dans la colonne « actions » du tableau des jobs en cours, actionnez la croix rouge correspondant au job que vous voulez arrêter. Cette action arrêtera proprement le job Slurm ainsi que le process exécuté sur le GPU.

Exemples d'utilisation des framewoks :

TensorFlow

```
#SBATCH --job-name=GPU-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

srun tf-py3 python "$HOME/tf-script.py"
```

Pytorch

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

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

srun pytorch-py3 python "$HOME/pytorch-script.py"
```

Installation de packages supplémentaires :

Les conteneurs dockers qui hébergent les frameworks de Machine Learning contiennent la version 3.6 de Python. Certains packages Python complémentaires ont été installés (voir la liste des packages disponibles sur le serveur Web d'Osirim : http://osirim.irit.fr/site/fr/articles/logiciels-disponibles).

Si vous souhaitez utiliser, à partir des conteneurs dockers, des packages supplémentaires non installés dans le docker, vous ne pourrez pas les installer directement dans le docker, car vous n'avez pas les droits « root ».

Pour installer des packages complémentaires, vous pouvez pour cela utiliser un « virtual env » python, que vous installerez depuis le docker. Voici ci-dessous la procédure à suivre :

Tout d'abord, vous devez créer un virtual env à partir d'un job, depuis le conteneur docker que vous souhaitez utiliser (exemple avec un docker Tensorflow) :

```
#!/bin/sh

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

srun tf-py3 virtualenv --system-site-packages /users/sig/toto/venv
```

Le paramètre --system-site-packages vous permettra d'utiliser dans le virtual env tous les packages déjà installés avec le python utilisé pour installer le vrtual env (Python 3.6 à l'intérieur du docker Tensorflow dans l'exemple).

Ensuite, une fois le virtual env créé, vous pouvez installer les packages voulus :

```
#!/bin/sh

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

srun  tf-py3 /users/sig/toto/venv/bin/pip3 install ...
```

Enfin, vous pouvez utiliser les nouveaux packages dans vos traitements, en exécutant le python installé dans votre virtual env :

```
#!/bin/sh

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

srun tf-py3 /users/sig/toto/venv/bin/python "$HOME/your-script.py"
```

Visualisation des jobs en file d'attente

La commande **"squeue"** permet d'afficher les Jobs en file d'attente.

Documentation officielle : https://slurm.schedmd.com/squeue.html

Afficher l'ensemble des jobs en liste d'attente :

```
$ squeue
```

"squeue" affiche les Jobs en attente de tous les utilisateurs par défaut.

Afficher ses propres Jobs uniquement :

```
$ squeue -u <user>
```

ou (générique)

```
$ squeue -u $(whoami)
```

La commande $(whoami) retournera votre "*username*" et servira de filtre à la commande squeue.

Affichage des informations d'un Job spécifique en file d'attente :

```
$ squeue -j1234
```

Personnalisation des champs affichés :

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

L'option "--format" ("-o" en version courte) permet de sélectionner les champs à afficher. Se référer à la documentation de la commande pour une liste complète des champs disponibles. Cet exemple affichera les champs suivants :

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

Afficher la file d'attente d'une partition spécifique :

```
$ squeue -p 8CPUNodes
```

Suivi de l'avancement des Jobs, Job Steps et utilisation des ressources

La commande **"*sacct*"** permet d'obtenir un grand nombre d'information sur les Jobs et leurs Steps.

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

La commande suivante affiche les informations par défaut du Job n°1234 :

```
$ sacct -j1234
```

```
       JobID    JobName  Partition    Account  AllocCPUS      State ExitCode
------------ ---------- ---------- ---------- ---------- ---------- --------
1234         slurm-job+ 8CPUNodes        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
```

La première ligne correspond au Job entier et les lignes suivantes (JobID suivi d'un point '.') indiquent les différents Steps du Job. Le Step "763.2" (3ème Step du Job 763) est ici en cours d'exécution (State : RUNNING).

L'option "--format" (ou "-o", version courte) permet de choisir les champs à afficher et leur taille à l'écran (en utilisant '%'). De nombreux attributs sont disponibles ; consultez la documentation pour la liste complète et leur signification.

```
$ 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
```

Les informations affichées sont ici choisies et le champ JobName affiche désormais le nom complet du Job et des Steps.

Affichage des statistiques d'utilisation des ressources (CPU/RAM/Disque…) :

```
$ 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 : Certains attributs ne sont disponibles qu'une fois le Step terminé.

Note 2 : Il est possible de modifier le format d'affichage des dates en modifiant la variable d'environnement SLURM_TIME_FORMAT. Le format de date utilisé par Slurm est celui de la fonction C "strftime" (http://man7.org/linux/man-pages/man3/strftime.3.html).

L'exemple ci-dessus utilise le format de date français (JJ/MM/AAAA hh:mm:ss). Pour définir le format des dates, le plus simple est d'ajouter la ligne suivante au fichier ".bashrc" de votre HOME utilisateur :

```
export SLURM_TIME_FORMAT='%d/%m/%Y %T'   # Définit le format d'affichage date/heure des commandes SLURM:JJ/MM/AAAA hh:mm:ss
```

Affichage pour un Job exécutant des Steps en parallèle :

```
$ 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 : L'allocation est ici de 18 CPUs au lieu des 6 précédents (3 Tasks de 6 CPUs sont exécutées en parallèle).

Pour ceux qui souhaiteraient récupérer ces informations pour les traiter dans un script (et/ou les formater dans un autre langage), sacct dispose des options "--parsable" et "--parsable2" qui retournent les même informations mais dont les champs sont séparées par un "pipe" ("|"). En mode "parsable", les tailles de champs ("%..") sont inutiles, les valeurs non tronquées étant toujours retournées. La différence entre ces deux options est que "--parsable" ajoute un "|" en fin de ligne alors que "--parsable2", non.

```
$ 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 : L'option "--noheader" permet par ailleurs de ne pas afficher les en-têtes dans le résultat.

FAQ

Q : **Comment exécuter un traitement sur le Cluster OSIRIM ?**

R : Vous devez vous connecter en SSH sur la machine "osirim-slurm.irit.fr" (port 22) et transmettre un fichier batch avec la commande "sbatch". Consultez la [documentation] SLURM pour des informations détaillées.

Q : **Mon Job se déroule correctement et mes données sont bien traitées. Pourquoi apparaît-il comme "Failed" dans l'email de fin et la commande sacct ?**

R : Une des raisons possibles est la suivante :
Le succès ou l'échec de toute commande shell (commande standard, script, fonction, programme…) est déterminé par son code de retour (exit code) ; zero (0) indique succès et ">0" indique un échec. Dans un script shell (comme le Batch Slurm), l'exit-code du script est déterminé par l'exit-code de sa dernière instruction ou commande exécutée.
Par exemple, si la fin d'un script est la suivante :

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

- Si le fichier existe, la dernière commande est "`echo`", qui retournera 0/Succès.
- S'il n'existe pas, la dernière commande exécutée est le test "-f" du fichier qui retourne 1/Echec. Aucune autre commande n'étant évaluée, le script entier retournera 1/Echec.

Solution : Ajouter "exit 0" explicitement dans les cas ou le script "réussit" même si le test ou la commande précédent(e) échoue.

Q : **J'ai besoin de lancer un grand nombre de jobs similaires, comment faire ?**
R : Le plus simple est de créer un Job Array.

Q : **J'ai demandé <N> Tasks de <C> CPUs dans mon batch mais mon Job n'utilise jamais plus de <C> CPUs en même temps !**
R : Avez-vous créé des Job Steps ? Si plus d'une Task est demandée mais qu'aucun Step n'est explicitement déclaré, l'allocation totale NE DÉPASSERA PAS 1 Task (<cpus-per-task> CPUs) !

Q : **J'essaie de lancer des Job Steps en parallèle mais ils s'exécutent l'un après l'autre !**
R : Pour exécuter des Job Steps en parallèle, il faut exécuter la commande srun en "arrière plan" en ajoutant '&' en fin de ligne.

Q : **J'essaie de lancer des Job Steps en parallèle mais ils ne sont pas exécutés, le Batch s'arrête immédiatement !**
R : Une fois les Job Steps déclarés, il faut impérativement utiliser la commande "wait" pour que le processus parent (le Job) attende la fin de l'exécution des processus enfant (Steps).

Q : **Dans mon fichier "output", Slurm m'affiche le Warning: "srun: Warning: can't run 2 processes on 4 nodes, setting nnodes to 2."**
R : Lors de l'exécution d'un Job Step avec "srun", le paramètre "-n" (ntasks) doit être supérieur ou égal à "-N" (nodes), sinon Slurm affiche ce warning et réduit -N à -n.

Annexe 1 : Options SBATCH courantes

La commande "sbatch" reçoit ses paramètres en ligne de commande mais autorise également leur définition via des "directives" SBATCH sous forme de commentaire en en-tête du fichier. Les deux méthodes produisent le même résultat mais ceux déclarés en ligne de commande auront la priorité en cas de conflit. Dans les deux cas, ces options existent (pour la plupart) en version courte et longue (exemple : -n ou --ntasks).

Pour plus d'information sur "sbatch", voir la documentation officielle à l'adresse :

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

Extrait des options SBATCH les plus courantes (versions longues uniquement) :

#SBATCH --partition=<part>

Choix de la partition SLURM à utiliser pour le job. Voir la section : Partitions.

#SBATCH --job-name=<name>

Définit le nom du job tel qu'il sera affiché dans les différentes commandes Slurm (squeue, sstat, sacct)

#SBATCH --output=<stdOutFile>

#SBATCH --error=<stdErrFile>

#SBATCH --input=<stdInFile>

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

Ces options définissent les redirections d'entrée/sorties du job (entrée/sortie/erreur standards).

La sortie standard (stdOut) sera redirigée vers le fichier défini par "--output" ou, si non définie, un fichier par défaut "slurm-%j.out" (Slurm remplacera "%j" par le JobID).

La sortie d'erreur (stdErr) sera redirigée vers le fichier défini par "--error" ou, si non définie, vers la sortie standard.

L'entrée standard peut aussi être redirigée avec "--input". Par défaut, "/dev/null" est utilisé (aucune/vide).

L'option "--open-mode" définit le mode d'ouverture (écriture) des fichiers et se comporte comme un open/fopen de la plupart des langages de programmation (2 possibilités : "append" pour écrire à la suite du fichier (s'il existe) et "truncate" pour écraser le fichier à chaque exécution du batch (valeur par défaut)).

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

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

Permet d'être notifié par e-mail d'un évènement particulier dans la vie du job : début de l'exécution (BEGIN), fin d'exécution (END, FAIL et TIME_LIMIT)... Consulter la documentation Slurm pour la liste complète des évènements pris en charge.

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

Définit le nombre de CPUs à allouer par Task. L'utilisation effective de ces CPUs est à la charge de chaque Task (création de processus et/ou threads).

#SBATCH --ntasks=<n>

Definit le nombre maximum de Tasks exécutées en parallèle.

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

Définit la RAM en Mo allouée à chaque CPU. Par défaut, 4096 Mo sont alloués à chaque CPU, l'utilisation de cette variable permet de spécifier une taille de RAM spécifique, inférieure ou égale à 7800 Mo (maximum allouable par CPU)

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

Nombre minimum[-maximum] de nœuds sur lesquels distribuer les Tasks.

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

Utilisée conjointement avec --nodes, cette option est une alternative à --ntasks qui permet de contrôler la distribution des Tasks sur les différents noeuds.

Valeurs par défaut et/ou déduites par Slurm :

Sans déclaration explicite de Job Step(s), une seule Task sera créée et les paramètres "--ntasks", "--nodes", "--nodelist"... seront ignorés.

De manière générale, lorsque "--nodes" n'est pas définie, Slurm détermine automatiquement le nombre de nœuds nécessaires (en fonction de l'utilisation des nœuds, du nombre de CPUs-par-Nœuds/Tasks-par-Nœud/CPUs-par-Task/Tasks, etc.).

Si "--ntasks" n'est pas définie, une Task par nœud sera allouée.

A noter que le nombre de Tasks d'un Job peut être défini soit explicitement avec "--ntasks" ou implicitement en définissant "--nodes" et "--ntasks-per-node".

Si les options "--ntasks", "--nodes" et "--ntasks-per-node" sont toutes définies, "--ntasks-per-node" indiquera alors le nombre maximum de Tasks par noeud.