

# Introduction au Calcul à Haute Performance

## Utilisation d'un cluster avec SLURM

Module de l'Ecole Doctorale SPI UCA  
H. Toussaint

23 mars 2021



# Présentation générale

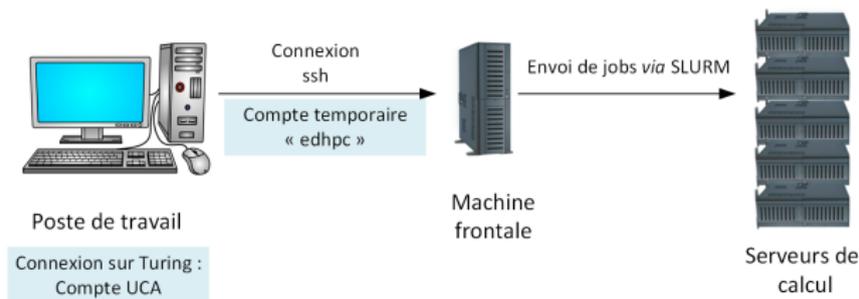
- Tous les TP du module ont lieu sur le cluster du LIMOS (comptes temporaires => pensez à récupérer votre travail)
- Un mésocentre accessible à tous les personnels UCA (compte créé sur demande) <https://mesocentre.uca.fr/ressources/>
- Documentation LIMOS + exemples : <https://hpc.isima.fr/>



# Cluster SLURM

## Accès au cluster :

- Un gestionnaire de ressources : SLURM (Simple Linux Utility for Resource Management)
- Connexion en ssh sur la machine frontale uniquement
- Accès aux noeuds de calcul via SLURM



# Cluster SLURM

## Intérêt d'un gestionnaire de ressources :

- Possibilité de réserver des ressources
- Mise en file d'attente si pas de ressource disponible
- Reproductibilité des temps de calcul (attention à l'hyperthreading, et turbo boost : adaptation dynamique à la charge de travail)



# Présentation de SLURM

## Qu'est-ce que SLURM ?

- Gestionnaire de ressources et ordonnanceur de tâches
- Répartit au mieux les ressources de calcul (CPU, GPU, RAM) entre utilisateurs en gérant des files d'attente avec priorité
- Confinement des ressources (CPU notamment) : un utilisateur ne peut pas accéder aux ressources réservées par d'autres
- Documentation complète : <https://slurm.schedmd.com/>



## Vocabulaire

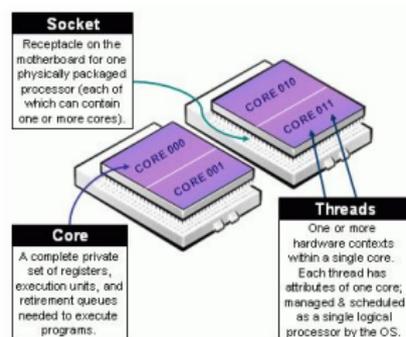


Figure – [https://slurm.schedmd.com/mc\\_support.html](https://slurm.schedmd.com/mc_support.html)

- Nœud de calcul (**node**) = une machine physique
- Un noeud contient des processeurs physiques (**sockets**) contenant eux-mêmes des cœurs physiques (**cores**)
- Si hyperthreading activé, chaque cœur physique donne 2 cœurs logiques (**threads**)
- Un **CPU** au sens SLURM = un cœur logique

# Vocabulaire

## Jobs et Tasks :

- Un **job** correspond à une requête d'allocation de ressources (CPU, RAM, temps de calcul)
- Un job est composé d'une ou plusieurs étapes (**steps**) et chaque étape effectue une tâche (**task**)
- Une task peut utiliser un ou plusieurs CPU.



# Vocabulaire

## Partition :

- Une **partition** dans SLURM correspond à un groupement logique de nœuds de calcul
- Chaque partition est associée à des contraintes en termes de ressources (en particulier le temps de calcul maximum d'un job)
- Une partition peut être vue comme une file d'attente

## Features :

- L'administrateur peut affecter des **features** (tags) aux nœuds en fonction de leurs caractéristiques (RAM / GPU / type de processeur ...).
- L'utilisateur peut préciser les features nécessaires pour ses jobs afin que SLURM affecte les jobs uniquement aux nœuds possédant le / les features demandés.



# Obtenir des informations sur le cluster

Informations sur les nœuds et les partitions :

- La commande `sinfo` affiche la liste des partitions disponibles
- La commande `sinfo -N` donne l'état des nœuds :
  - **alloc** : le nœud est entièrement utilisé
  - **mix** : le nœud est en partie utilisé
  - **idle** : aucun job ne tourne sur le nœud
  - **drain** : le nœud termine les jobs qui lui ont été soumis mais n'en accepte plus d'autres (typiquement le nœud est sur le point d'être arrêté pour une opération de maintenance)
- La commande `sinfo -Nl` donne l'état des nœuds avec plus de détails



# Soumission d'un job

2 solutions :

- Définir le job dans un script et le lancer à l'aide de la commande `sbatch` (**solution conseillée**)
- Lancer directement le job en ligne de commande à l'aide de la commande `srun`

**Attention** : les jobs ne doivent en aucun cas s'exécuter sur le frontal (sous peine d'être tués par l'administrateur)



# Soumission d'un job avec un script

- Ecriture d'un script bash (.sh)
- Possibilité de mettre des options (toutes les options sont facultatives)
- Soumission du script avec la commande `sbatch`
- SLURM attribue un numéro au job
- Quand le job est terminé un fichier `slurm-<numJob>.out` est créé par SLURM, il contient la sortie du programme

Voir `exemple1.sh` et `exempleC`

```
exemple1.sh
1  #!/bin/bash
2
3  # -----
4  # programme (ligne de commande à exécuter)
5
6  hostname
```

```
[toussain@frontalhpc formationSLURMDec2017]$ sbatch exemple1.sh
Submitted batch job 19968
[toussain@frontalhpc formationSLURMDec2017]$ cat slurm-19968.out
iroise.rcisima.isima.fr
```



## Ajouter des options

- Par défaut (sur le cluster du LIMOS) SLURM attribue à un job :
  - 1 CPU
  - 2Go de RAM
  - 7 jours max de temps de calcul (partition par défaut)
- On peut modifier ce comportement par défaut à l'aide d'options
- Les options sont précisées dans le script à l'aide du mot clé **#SBATCH**

```
example1.sh x3  example2.sh x3
1  #!/bin/bash
2
3  # =====
4  #  exemples d'options
5
6  #SBATCH --partition=normal    # choix de la partition où soumettre le job
7  #SBATCH --time=10:0          # temps max alloué au job (format = m:s ou h:m:s ou j-h:m:s)
8  #SBATCH --ntasks=1          # nb de tasks total pour le job
9  #SBATCH --cpus-per-task=1    # 1 seul CPU pour une task
10 #SBATCH --mem=1000           # mémoire nécessaire (par noeud) en Mo
11
12 # =====
13 #exécution du programme
14 hostname
```



## Quelques options

La commande `sbatch --help` donne une synthèse des options existantes.

### Options de gestion du job

option	utilisation
-a, --array=<indexes>	valeurs des indices pour un tableau de jobs
-e, --error=<fic>	nom du fichier de sortie pour les erreurs
-l, --immediate	le job est tué si les ressources ne sont pas immédiatement disponibles
-J, --job-name=<jobname>	donne un nom au job
	--mail-type=<type> notification par mail, type = BEGIN, END, FAIL ou ALL
	--mail-user=<user> adresse mail à qui envoyer les notifications
-o, --output=<out>	fichier de sortie
-t, --time=<time>	temps max alloué au job (format = m:s ou h:m:s ou j-h:m:s)

## Quelques options

option	utilisation
-c, --cpus-per-task=<ncpus>	nombre de CPU pour une tâche
-n, --ntasks=<ntasks>	nombre de tâches du job
--ntasks-per-node=<n>	nombre de tâches désirées sur chaque noeud
-N, --nodes=<N>	nombre de nœuds désirés (N = min[-max]), N = 1 par défaut
-C, --constraint=<features>	<i>features</i> désirées (plusieurs features peuvent être spécifiées, dans ce cas les séparer par &)
--mem=<MB>	quantité minimale de mémoire pour le job en Mo
--mincpus=<n>	nombre minimal de CPU (coeur logique) par noeuds = nb de threads
-w, --nodelist=<hosts>	précise les nœuds désirés (séparés par des virgules), cette liste doit contenir autant que noeuds que le nombre de noeuds réservés par l'option --nodes
-x, --exclude=<hosts>	exclut certains nœuds de la partition (séparés par des virgules)

Voir exemple2.sh



# Suivi des jobs

File d'attente :

- `squeue` affiche la liste des jobs en cours et en attente
  - `squeue -u <user>` affiche les jobs en cours et en attente pour l'utilisateur `user`
  - `squeue -p <nomPart>` affiche les jobs en cours et en attente pour la partition demandée
  - `squeue -i <sec>` actualise la liste des jobs en cours toutes les `sec` secondes

Suppression d'un job :

- `scancel <jobID>` supprime le job `<jobID>` (en cours ou en attente)
- `scancel -u <user>` supprime les jobs de l'utilisateur `<user>` (en cours ou en attente)



# Suivi des jobs

Informations sur l'état d'un job :

- **sacct** affiche l'état des jobs de l'utilisateur qu'ils soient en cours ou déjà terminés :
  - **CA, cancelled** : le job a été annulé par l'utilisateur ou l'administrateur
  - **CD, completed** : le job s'est terminé avec succès
  - **CG, completing** : job en cours
  - **F, failed** : le job s'est terminé avec un échec
  - **PD, pending** : le job attend des ressources
  - **R, running** : le job est en cours d'exécution
  - **TO, timeout** : le job s'est terminé car il a atteint son temps d'exécution limite
- **scontrol show job** <jobID> donne des informations détaillées sur le job <jobID> (en cours)



# Tableaux de jobs

A quoi sert un tableau de jobs ?

- Soumettre à l'aide d'un seul script plusieurs jobs indépendants.
- Utile par exemple pour appliquer le même programme à différentes données d'entrée

Comment le définir ?

- A l'aide de l'option `--array=indiceMin-indiceMax`

**Remarque** : soyez parcimonieux dans vos demandes d'allocation.



## Tableaux de jobs : exemple

Contexte :

- On dispose de 10 instances : insA.txt à insJ.txt
- On souhaite exécuter le programme exe sur chacune de ces instances

Voir le dossier `exempleTableau`

```
exemple3.sh x test.c x submit.sh x
1  #!/bin/bash
2
3  #===== OPTIONS (s'applique à chaque job du tableau) =====
4  #SBATCH --array=0-9          # création d'un tableau de 10 jobs indicés de 0 à 9
5  #SBATCH --ntasks=1         # chaque job possède une seule task
6
7  #===== execution =====
8  tab=(A B C D E F G H I J)
9  ./exe ins${tab[${SLURM_ARRAY_TASK_ID}]} .txt
```

On lance le script de la manière habituelle avec `sbatch`.

SLURM attribue à chaque job du tableau à un identifiant propre constitué du numéro du job principal et de son indice dans le tableau

# Variables d'environnement

Les variables d'environnement peuvent être utilisées dans les scripts pour afficher des informations sur le job :

- `SLURM_JOB_ID` : numéro du job
- `SLURM_JOB_NAME` : nom du job
- `SLURM_JOB_NUM_NODES` : nombre total de nœuds alloués pour le job
- `SLURM_SUBMIT_DIR` : répertoire à partir duquel le job est soumis
- `SLURMD_NODENAME` : nom du nœuds sur lequel le job s'exécute
- `SLURM_JOB_PARTITION` : nom de la partition sur laquelle le job s'exécute



# Variables d'environnement

Exemple d'utilisation :

```
exemple3.sh x
1  #!/bin/bash
2
3  # =====
4  # OPTIONS
5  #SBATCH --job-name=testVarEnv
6
7  # =====
8  # programme (ligne de commande à exécuter)
9  echo mon job $SLURM_JOB_NAME, num $SLURM_JOB_ID,
10 echo "s'exécute" dans la partition $SLURM_JOB_PARTITION sur le noeud $SLURMD_NODENAME
11 hostname
12 sleep 60
```

# Variables d'environnement

D'autres variables sont liées aux tableaux de jobs :

- `SLURM_ARRAY_JOB_ID` : numéro du job principal
- `SLURM_ARRAY_TASK_ID` : indice de la tâche dans le tableau
- `SLURM_ARRAY_TASK_COUNT` : nombre total de tâches dans un tableau de jobs
- `SLURM_ARRAY_TASK_MAX` : indice max du tableau de jobs
- `SLURM_ARRAY_TASK_MIN` : indice min du tableau de jobs



# Scripts pour programme multithread

## Mémoire partagée (type openMP)

- une task ...
- ... qui utilise plusieurs CPU

Voir l'exemple dans le dossier `exempleOpenMP`

```
exemple3.sh [x] test.c [x] submit.sh [x] submit.sh [x] prodMatrice_parallel.c [x]
1  #!/bin/bash
2
3  #===== OPTIONS de sbatch =====
4  #SBATCH --job-name=openMP
5  #SBATCH --ntasks=1           # on a une seule tâche, elle utilise 8 cœurs (mémoire partagée)
6  #SBATCH --cpus-per-task=8    # Allocation de 8 CPU
7  #SBATCH --mem-per-cpu=10240  # RAM par CPU (en Mo)
8
9  #SBATCH --partition=court
10 #SBATCH --nodelist=dellware
11
12 #SBATCH --time=2:00
13
14 #===== programme a executer=====
15 ./exe
```



# Scripts pour programme multithread

Mémoire partagée (type openMP)

Pour avoir de bonnes performances :

- Le nombre de threads utilisés par openMP ne doit pas excéder le nombre de CPU réservé par SLURM

## **Attention aux librairies "multithreadées" :**

- Certaines utilisent par défaut autant de threads que de cœurs disponibles (Cplex par exemple)
- Il faut toujours leur imposer un nombre de threads max  $\leq$  au nombre de CPU réservé par SLURM (sinon perte de performances)



# Scripts pour programme multiprocessus

## Mémoire distribuée (type MPI)

Pour un programme à mémoire distribuée on a :

- $n$  tasks ( $\approx n$  processus)
- Possibilité de distribuer les tasks sur plusieurs nœuds d'une même partition
- Bonne pratique : attribuer 1 task par CPU

Voir le dossier exempleMPI (**attention modif à faire dans le `.bashrc`**).

```
1 #!/bin/bash
2
3 # REMARQUE : SLURM distribue automatiquement les processus de MPI
4 # sur les noeuds -> inutile de préciser l'option -np
5
6 #SBATCH --ntasks=20
7 #SBATCH --ntasks-per-core=1
8 #SBATCH --job-name=testMPI
9
10 mpirun ./hello
```



# Les shells interactifs

On peut utiliser un job SLURM pour ouvrir un shell interactif (i.e. une session bash) sur un noeud.

Utilité :

- Débogage
- Compilations longues
- Programmes nécessitant une interaction avec l'utilisateur
- Suivre la consommation CPU / RAM en temps réel (commande `top` ou `htop`)

**Important :** Pensez toujours à bien quitter la session interactive à l'aide de la commande `exit` pour éviter de laisser des ressources oisives et indisponibles pour les autres utilisateurs.



# Les shells interactifs

Demander un shell interactif :

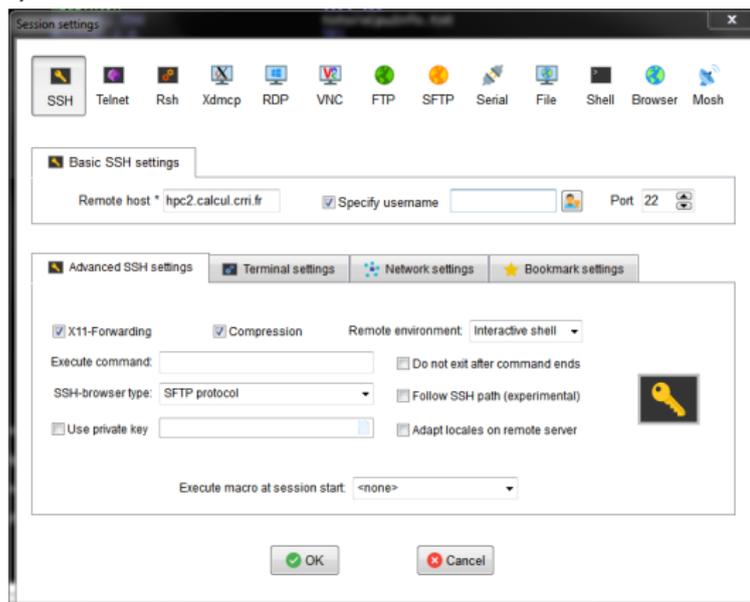
- En ligne de commande
- Commande basique : `srun --pty bash`
- Possibilité d'ajouter des options (les mêmes que dans les scripts)
- Exemple : `srun --ntasks=1 --cpus-per-task=2  
--partition=court --pty bash`

Important : Pensez toujours à bien quitter la session interactive à l'aide de la commande `exit` pour éviter de laisser des ressources oisives et indisponibles pour les autres utilisateurs.



# Accès au cluster du mésocentre

- Accès au cluster du mésocentre (après création du compte!) :
  - connexion en ssh depuis linux
  - utilisation d'un client ssh depuis windows (putty, mobaXterm, Secure Shell, ...)



## Modules (disponibles uniquement au mésocentre)

- Permet de charger un environnement de travail avec des librairies spécifiques (boost, MPI, ...) : modifie les variables d'environnement (PATH, ...)

Quelques commandes :

- `module list` : liste les modules actifs
- `module avail` : liste les modules disponibles
- `module spider` : liste les modules disponibles (+ détaillé)
- `module load <nomDuModule>` : charge le module



# Exercices

- 1 Créer un script qui permet de lancer la commande `hostname` sur un des nœuds de la partition `normal` à l'exception de `django`
- 2 Créer un script qui lance la commande `hostname` sur `kephren`
- 3 Ouvrir un shell interactif sur `kephren` permettant d'exécuter un programme nécessitant 10 Go de RAM et 4 CPU
- 4 Créer un script qui permet de lancer le programme MPI sur l'ensemble des CPU disponibles dans la partition `debug`

