Utilisation du gestionnaire de ressources SLURM

(Basée sur les documents d'Hélène Toussaint : https://hpc.isima.fr)

0. Préambule

Les TP du « Calcul parallèle » seront effectués sur le cluster du LIMOS (Laboratoire de recherche en Informatique qui se trouve dans les bâtiments de l'Institut Informatique).
Un cluster est un regroupement des machines dédiées au calcul. Il est composé d'une machine frontale **frontalhpc** ("porte d'entrée" du cluster) et de plusieurs machines de calcul, appelées les nœuds. Les nœuds sont répartis dans des partitions ayant des caractéristiques ou d'utilité différente. Les TP du « Calcul parallèle » s'effectueront sur les nœuds réservés aux TP, c'est-à-dire le **node27** et le **node29** qui forme la partition **part-etud**.

1. Accès aux nœuds

On accède au cluster en se connectant (en **ssh**) sur le serveur **frontalhpc** (frontalhpc.rcisima.isima.fr), en utilisant le login / mot de passe de son compte habituel. Pour exécuter un programme sur un nœud de calcul, l'utilisateur doit obligatoirement passer par l'ordonnanceur de tâches SLURM.

2. Commandes du SLURM

SLURM (Simple Linux Utility for Resource Management) est un gestionnaire de ressources et ordonnanceur de tâches pour des clusters LINUX. Il permet de répartir au mieux les ressources de calcul (CPU, GPU, RAM) entre utilisateurs.
Une documentation se trouve sur le site HPC du LIMOS : https://hpc.isima.fr

Les commandes et options en gras sont les minimums que vous devez maitriser pour les TP.

Commandes du SLURM :

- **sinfo** : affiche la liste des partitions disponibles
- sinfo –N : donne l'état des nœuds
    - *alloc* : le nœud est entièrement alloué
    - *idle* : aucun job ne tourne sur le nœud
    - *mix* : le nœud est en partie utilisé
    - *drain* (*drng*) : le nœud termine les jobs qui lui ont été soumis mais n'en accepte plus d'autres
- srun : exécution directe du job en ligne de commande. Les options sont similaires à celles de sbatch. Synopsis : srun [options] executable [args(0)…]
- **sbatch** : exécution en batch des jobs via script (**solution que nous utilisons**).
  Synopsis : **sbatch [options] script [args(0)…]**
- sbatch --help : donne la liste des options de sbatch comme exemples :
    - –c (**--cpus-per-task=<ncpus>**) : nombre de CPU pour une tâche (effectuée par un processus), correspond au nombre de threads par processus
    - –n (**--ntasks=<ntasks>**) : nombre de tâches du job, correspond au nombre de processus
    - (**--ntasks-per-core=<n>**) : nombre de tâches désirées sur chaque core
    - (--ntasks-per-node=<n>) : nombre de tâches désirées sur chaque nœud, utilisé comme le nombre maximum de tâches par nœud s'il est utilisé en même temps que l'option –ntasks (–ntasks prédomine --ntasks-per-node)
    - –N (--nodes=<N>) : nombre de nœuds désirés (N=min[-max]), N=1 par défaut
    - –w (**--nodelist=<hosts>**) : précise les nœuds désirés (séparés par des virgules)
    - (--mem=<MB>) : quantité minimale de mémoire pour le job, en Mo

- (--mincpus=<n>) : nombre minimal de cœurs logiques par nœud = nombre de threads
- -o (--output=<out>) : fichier de sortie
- -e (--error=<errorout>) : fichier de sortie des erreurs
- -J (--job-name=<jobname>) : donne un nom au job

- **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
  - Job State Code :
    - *CA* : CAncelled
    - *CD* : CompleteD; All processes on all nodes with an exit code of 0
    - *CG* : Completing; Job is in process of completing. Some processes on some nodes may still be active.
    - *F* : Failled
    - *PD* : PenDing; Job is waiting resource allocation
    - *R* : Running
    - *ST* : STopped; Job has an allocation, but execution has been stopped by SIGSTOP. CPUS have been retained by this job.
    - *S* : Suspended; Job has an allocation, but execution has been suspended. CPUS have been retained by this job.
    - *TO* : TimeOut

- **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)

- **sacct** : affiche l'état des jobs de l'utilisateur qu'ils soient en cours ou terminés :
  - *CA* : CAncelled
  - *CD* : CompleteD
  - *CG* : Completing
  - *F* : Failed
  - *PD* : PenDing
  - *R* : Running
  - *TO* : TimeOut
- scontrol show job <jobID> : donne des informations détaillées sur le job <jobID> en cours

Attention : les jobs ne doivent en aucun cas s'exécuter sur le frontal du cluster.

3. Shell interactif

Pour avoir un shell interactif sur un nœud particulier, par exemple sur **node29**, vous pouvez utiliser la commande : **srun -A tp-etud -p part-etud -w node29 --pty bash**
A partir de là, on est sur le nœud **node29**, on peut exécuter les commandes usuelles sur le nœud.

4. Exécution d'un programme multi-threads (OpenMP)

Pour une exécution en utilisant SLURM, la réservation des ressources étant effectuée par SLURM, il n'y a pas à indiquer le nombre de threads dans le programme.

```
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"

int main()
{
   printf("Before PARALLEL REGION : There are %d threads\n\n" ,
          omp_get_num_threads()) ;

/* #pragma omp parallel num_threads(8)*/ /* pour execution directe */
   #pragma omp parallel                  /* pour execution avec SLURM */
   {
       printf("In the PARALLEL REGION : There are %d threads\n\n" ,
              omp_get_num_threads()) ;
       printf("Hello World from TID %d!\n", omp_get_thread_num());
    }

   printf("After PARALLEL REGION : There are %d threads\n\n" ,
          omp_get_num_threads()) ;

   return EXIT_SUCCESS;
}
```

- Compiler le programme avec la commande : **gcc -fopenmp hello_omp.c -o hello_omp** ; Vous pouvez ajouter les options habituelles de compilation ; La compilation peut être effectuée sur le nœud frontal ou un nœud particulier.
- Exécution du programme avec SLURM en utilisant sbatch :
  $ sbatch submit_hello_omp.sh

```
#!/bin/bash

# submit_hello_omp.sh fichier pour l'execution de hello avec sbatch

# Options de sbatch
#SBATCH --partition=part-etud  # partition pour tps etudiants
#SBATCH --ntasks=1             # 1 task / processus
#SBATCH --cpus-per-task=8      # de 8 threads
#SBATCH job-name=testOpenMP

# Execution du programme
./hello_omp
```

- Vous pouvez aussi exécuter le programme en ligne de commande avec SLURM :
  srun --partition=part-etud --ntasks=1 --cpus-per-task=8 ./hello_omp

- Ou faire une exécution directe sur le serveur local **: /* NE PAS FAIRE CELA SUR LE CLUSTER.** Vous pouvez tester cela sur **etud** ou **gromit**. Voir les PPT du cours pour la définition du nombre de threads. ***/**
  $ ./hello_omp

5. Exécution d'un programme multi-processus (MPI)

Pour une exécution avec SLURM, le nombre de processus peut être précisé via les options de commandes SLURM.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include <sched.h>

int main(int argc, char **argv)
{
      int  rang=-1, nbProcs=0, nameLength=0;
      int  cpuId=-1; /* Numero du core du nœud utilise */
      char proc_name[MPI_MAX_PROCESSOR_NAME]; /* Nom du nœud utilise */

      MPI_Init( &argc, &argv );
      MPI_Comm_rank( MPI_COMM_WORLD, &rang );
      MPI_Comm_size( MPI_COMM_WORLD, &nbProcs );

      MPI_Get_processor_name( proc_name, &nameLength) ;
      printf( " Hello from proc. %d of %d on %s\n ",
            rang, nbProcs, proc_name);

      MPI_Finalize();
      return EXIT_SUCCESS;
}
```

- Compiler le programme avec : **mpicc hello_mpi.c -o hello_mpi** ; Vous pouvez ajouter les options habituelles de compilation ;

- Exécution avec SLURM en utilisant sbatch :
  $ sbatch submit_hello_mpi.sh

```
#!/bin/bash

# submit_hello_mpi.sh script d'execution de hello avec sbatch

# Options de sbatch
#SBATCH --partition=part-etud # partition pour execution courte
#SBATCH --ntasks=8            # 8 tasks
#SBATCH --ntasks-per-core=1   # 1 task par core
#SBATCH --job-name=testMPI

#execution
mpiexec ./hello_mpi
```

- Utiliser `--ntasks-per-node=1` à la place de `--ntasks-per-core=1` permet d'exécuter 1 task par nœud.