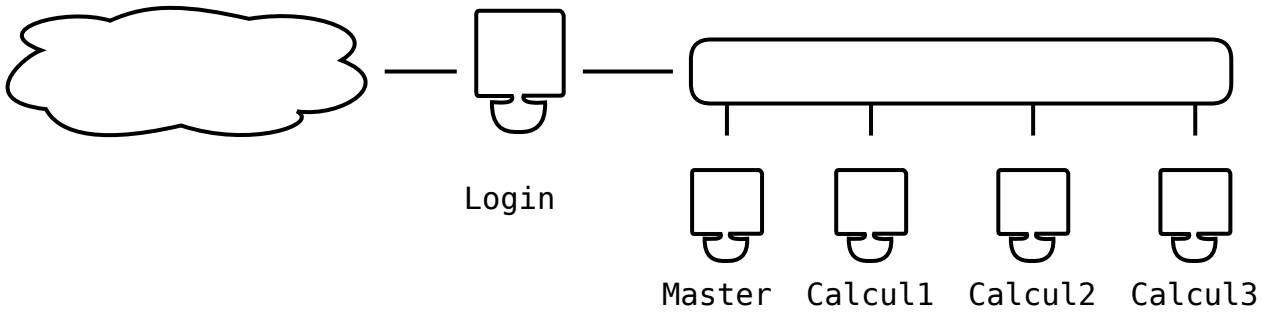
Création d'un cluster HPC sur Raspberry

Table of Contents

Un cluster **HPC** (**H**aute **P**erformance de **C**alcul) peut être vu comme un ensemble de machines coopérants pour résoudre un problème non soluble dans un temps raisonnable par une machine seule. Nous allons voir comment créer un cluster sous Linux en utilisant des logiciels libres.

L'objectif est de construire un cluster **HPC** basé sur des logiciels libres. Pour cela on considèrera que les services usuels sont installés et opérationnels (notamment DNS, répertoires utilisateurs distribués et service d'annuaires pour les comptes systèmes).

Présentation



L'utilisateur se connecte à un nœud frontal (login) où il trouve son environnement ainsi que sa home montée via un système de fichiers distribué sur l'ensemble du cluster. Tous les programmes sont compilés sur le frontal. L'utilisateur interagit ensuite avec un job scheduler localisé sur le nœud d'administration (admin) via un jeu de commandes installé sur le nœud frontal pour demander la réalisation de ses calculs. Le job scheduler distribue ensuite le travail sur les nœuds de calcul (compute1 et compute2). L'utilisateur ne peut donc pas se connecter en direct sur les nœuds de calcul. Il doit décrire les ressources qu'il demande (nombre de CPU, de nœuds, quantité de mémoire etc.) ainsi que la façon de lancer ses calculs (exécutable, script, variables d'environnements etc.). C'est ce qu'on appelle un « job ».

L'installation du cluster va se faire au-dessus d'un Debian dans lequel on va déployer quatre composants :

- **Kanif**: propagation de commandes sur la machine;

- **SLURM**: job scheduler libre;
- **OpenMP** et **OpenMPI** : librairies incontournables dans le domaine du calcul.

Phase 1 : Kanif

Kanif propose deux commandes très utiles pour administrer un cluster HPC :

1. **kash** qui exécute une commande sur un ensemble de machine ;
2. **kaput** qui copie des fichiers / répertoires depuis une source sur un ensemble de machines.

L'installation se fait très simplement via le gestionnaire de paquet Debian :

```
apt-get install kanif
```

Il faut ensuite avoir une authentification par clé SSH fonctionnelle. Cette authentification se fait depuis le compte root du nœud frontal vers les comptes root des nœuds de calcul. Côté nœuds de calcul, il faut juste installer **TakTuk** :

```
apt-get install taktuk
```

Tester :

```
root@login:~# kash -n compute[1-2] hostname

-------------------------------------------------------------------------------

STDOUT

-------------------------------------------------------------------------------

-------------------------------------------------------------------------------

compute[1-2] (2 HOST)

-------------------------------------------------------------------------------

compute1

compute2
```

La commande hostname s'est bien exécutée sur compute1 et compute2 depuis le nœud de login.

Phase 2 : SLURM

SLURM est un job scheduler. Son rôle est de maintenir une image aussi précise que possible de l'état d'utilisation des ressources sur le cluster. Il se base sur cette image pour placer les jobs des utilisateurs en fonction des ressources libres.

L'utilisateur se connecte au nœud frontal et utilise la commande **srun** ou **sbatch** pour envoyer un job sur le cluster. Ces deux commandes sont respectivement utilisées pour le mode interactif et batch. Une connexion à un service **slurmctld** en exécution sur le nœud d'administration est réalisée. Ce service connaît la topologie du cluster ainsi que l'état d'utilisation des ressources. Si les ressources demandées par l'utilisateur sont libres, **slurmctld** demande au(x) service(s) **slurmd** de(s) noeud(s) de calcul(s) réservé(s) d'exécuter le job.

Le service **slurmd** accepte la requête du service **slurmctld** et fork immédiatement un processus **slurmstepd** qui va manager la consommation de ressources du job ainsi que ses entrées / sorties. Le job est un processus fils de **slurmstepd** exécuté sous l'identité de l'utilisateur (setuid). Pour installer SLURM, la première chose à faire est de créer un utilisateur système dédié consistant sur le réseau (c'est à dire avec le même UID partout).

```
useradd -d /var/slurm -r -u 600 -U slurm
```

Il faut ensuite créer tous les répertoires nécessaires à SLURM sur tous les nœuds du cluster :

```
mkdir -p /var/slurm/slurmd && mkdir /var/slurm/slurmctld && chown -R
slurm:slurm /var/slurm
mkdir /var/log/slurm && chown -R slurm:slurm /var/log/slurm
```

Ces deux opérations doivent être répétées pour le nœud d'administration et les nœuds de calculs. Les communications entre **slurmctld** et **slurmd** sont authentifiées avec **Munge** (clé privée partagée par les nœuds). Il faut donc installer **Munge** sur tous les nœuds du cluster :

```
apt-get install munge
```

Et distribuer la clé privée :

```
kaput -n compute[1-2] /etc/munge/munge.key /etc/munge
scp /etc/munge/munge.key root@admin:/etc/munge
```

Le système est prêt à accueillir SLURM. Il reste juste à installer les librairies de développement de Munge sur le nœud frontal pour pouvoir compiler le support de Munge dans SLURM :

```
apt-get install libmunge-dev
```

Passons à la compilation de SLURM :

```
bunzip2 slurm-14.11.7.tar.bz2 && tar -xvf slurm-14.11.7.tar && cd
slurm-14.11.7
./configure –prefix=/usr/local/compiled/slurm-14.11.7
ln -s /usr/local/compiled/slurm-14.11.7 /usr/local/compiled/slurm
make && make install
```

SLURM sera donc installé dans `/usr/local/compiled/slurm` (on peut choisir un autre préfixe). On réplique l'installation sur les nœuds de calcul et le nœud d'admin en y copiant le répertoire `/usr/local/compiled/slurm-14.11.7` et en réalisant le lien.

On rédige ensuite un fichier minimal de configuration qui est partagé à la fois par les outils clients **srun** et **sbatch**, **slurmctld** et **slurmd**. Ce fichier doit être identique sur tous les nœuds du cluster :

```
ClusterName=magi
ControlMachine=admin
SlurmUser=slurm
AuthType=auth/munge
MailProg=/usr/bin/mail
MpiParams=ports=12000-12999
SlurmdDebug=debug
StateSaveLocation=/var/slurm/slurmctld
SlurmdSpoolDir=/var/slurm/slurmd
SlurmctldPidFile=/var/run/slurmctld.pid
SlurmdPidFile=/var/run/slurmd.pid
SlurmdLogFile=/var/log/slurm/slurmd.log
SlurmctldLogFile=/var/log/slurm/slurmctld.log
SlurmdTimeout=600
TmpFS=/scratch
NodeName=compute[1-2] Procs=8 State=UNKNOWN
PartitionName=COMPUTE Nodes=magi6 Default=YES MaxTime=INFINITE State=UP
Shared=YES
```

Ce fichier définit le nom du cluster (ClusterName), la machine où tourne **slurmctld** (ControlMachine), l'utilisateur système associé à **slurmctld** (SlurmchrUser) et le type d'authentification (AuthType). Ensuite viennent les paramètres propres à chaque service. Pour slurmd : MpiParams fixe les ports réseau utilisés pour les communications MPI (nous y reviendrons), SlurmdSpoolDir donne l'emplacement des fichiers d'état des jobs sur la machine, Slurm[Pid|Log]File sont équivoques et SlurmdTimeout fixe le temps de non-réponse au-delà duquel un nœud de calcul est jugé down par **slurmctld**. Pour **slurmctld** : StateSaveLocation fixe l'emplacement des fichiers contenant l'affectation des ressources sur le cluster. Slurmctld[Pid|Log]File sont également équivoques. Le mode debug est activé pour voir si l'exécution des services **slurmctld** et **slurmd** est exempte de toute erreur.

Les deux dernières lignes renseignent la topologie du cluster. La première décrit les nœuds de calcul au niveau physique (CPU, mémoire, technologie d'interconnexion etc.). Ici il s'agit de deux machines compute1 et compute2 équipées chacune de 8 cœurs. La seconde ligne instancie ce que l'on appelle une partition. Une partition peut être vue comme un groupe de machine partageant des caractéristiques communes. Nous pourrions avoir une partition pour les machines sur base Intel, une autre pour les IBM Blue Gene, une autre encore pour les GPU etc. L'utilisateur choisit alors de lancer ses jobs sur l'une ou l'autre des partitions. Ici nous avons une partition COMPUTE contenant les

machines compute1 et compute2. Les machines doivent obligatoirement être décrite via la directive NodeName avant d'intégrer une partition. Une machine ne peut pas être dans plusieurs partitions.

Ce fichier de configuration est installé dans `/usr/local/compiled/slurm/etc/slurm.conf` sur tous les nœuds. Il nous faut maintenant rédiger les scripts de démarrage pour le très controversé systemd. Il va falloir en faire un pour **slurmctld** sur le nœud d'administration et un autre pour **slurmd** sur les nœuds de calcul. Commençons par **slurmctld** en créant `/lib/systemd/system/slurmctld.service` :

```
[Unit]
Description=Slurmctld
After=syslog.target
[Service]
Type=forking
ExecStart=/usr/local/compiled/slurm/sbin/slurmctld -f
/usr/local/compiled/slurm/etc/slurm.conf
[Install]
WantedBy=multi-user.target
```

On notera qu'avec l'ajout du paramètre -D la commande affectée à ExecStart peut être utilisée sur le terminal pour débugger le lancement de slurmctld. Activons et démarrons le service :

```
systemctl enable slurmctld.service
service slurmctld start
```

Testons qu'il soit en exécution :

```
netstat -laputn -A inet | grep slurmctld

tcp        0      0 0.0.0.0:6817            0.0.0.0:*               LISTEN
1421/slurmctld
```

On trouve bien un processus **slurmctld** en écoute sur le port TCP 6817. En complément on peut aller voir dans les logs `/var/log/slurm/slurmctld.log` pour traquer d'éventuels warnings.

La même manipulation doit être réalisée sur les nœuds de calcul pour **slurmd**. On peut vérifier la disponibilité réseau du service **slurmd** en combinant **kash** et **netstat**. À ce stade, SLURM est opérationnel et prêt à recevoir du travail. Se reconnecter en simple utilisateur et fixer quelques variables d'environnements pour localiser le fichier de configuration et les exécutables de SLURM :

```
export SLURM_CONF=/usr/local/compiled/slurm/etc/slurm.conf
export PATH=/usr/local/compiled/slurm/bin:$PATH
```

Commençons par vérifier l'état du cluster avec un sinfo :

```
sinfo
```

```
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
COMPUTE*     up   infinite      2   down compute[1-2]
```

On voit que les deux machines de la partition COMPUTE sont down. C'est normal car elles viennent d'être ajoutées dans le cluster. On va les passer en idle (qui est l'état dans lequel les nœuds attendent du travail) avec la commande **scontrol**:

```
sudo scontrol update nodename=compute[1-2] state=idle
```

Vérifier l'état de la partition :

```
sinfo

PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
COMPUTE*     up   infinite      2   idle compute[1-2]
```

Les deux nœuds sont en idle on peut lancer un premier job. On va utiliser la commande **srun** qui soumet un job à SLURM et en affiche les sorties sur la sortie standard du terminal :

```
srun -N 2 hostname

compute1
compute2
```

Dans cet exemple, l'utilisateur demande de lancer la commande hostname sur deux nœuds la partition par défaut (COMPUTE). Le service **slurmctld** demande donc aux services **slurmd** de compute1 et compute2 de s'exécuter et les résultats sont envoyés sur la sortie standard de l'utilisateur. Le terminal de l'utilisateur est donc bloqué le temps de l'exécution du job. C'est ce qu'on appelle le mode interactif. On verra un peu plus loin comment utiliser le mode batch pour soumettre un job et relâcher la session de l'utilisateur en redirigeant les sorties.

Phase 3 : OpenMP

On a maintenant un cluster opérationnel. On va voir les deux librairies de parallélisation stars du domaine du HPC : **OpenMP** et **OpenMPI**. **OpenMP** est une surcouche du multithreading dans le sens où elle permet de paralléliser des boucles très simplement via des pragmas ajoutés au code source. **OpenMP** est clairement orienté programmation à mémoire partagée (c'est à dire que la mémoire doit être accessible aux différents fils d'exécution du programme compilé avec **OpenMP**), ce qui cantonne le programme à une machine isolée (même s'il existe des systèmes à image unique, mais c'est une autre histoire !). Voici un code **OpenMP** très simple :

```
#include <utmpx.h>
int main (int argc, char *argv[]) {
   int nthreads, thread_id;
   #pragma omp parallel private(nthreads, thread_id) {
```

```
      thread_id = omp_get_thread_num();
      printf("Thread %d says: Hello World on core %i\n",
thread_id,sched_getcpu());
      if (thread_id == 0) {
         nthreads = omp_get_num_threads();
         printf("Thread %d reports: the number of threads are %d\n",
                        thread_id, nthreads); } }
  return 0; }
```

Ce code crée N fils d'exécution au niveau du pragma omp parallel et affiche le numéro du processeur sur lequel le code du fil s'est exécuté. Commencer par le compiler en ajoutant la directive `-fopenmp` à une compilation classique :

```
gcc -fopenmp testomp.c -o testomp
```

Pour exécuter ce code, il est nécessaire de spécifier combien de fils d'exécution sont requis. Cela se fait au moyen de la variable d'environnement **OMP_NUM_THREADS**. On le fixe et exécute le programme dans la foulée :

```
export OMP_NUM_THREADS=4
./testomp

Thread 1 says: Hello World on core 2
Thread 0 says: Hello World on core 3
Thread 0 reports: the number of threads are 4
Thread 2 says: Hello World on core 5
Thread 3 says: Hello World on core 4
```

On voit bien les quatre fils d'exécution et que chaque fil est localisé sur un cœur différent. L'affectation des fils sur les cœurs est laissée à la discrétion de l'ordonnanceur du noyau Linux. On va lancer maintenant ce programme sur le cluster :

```
srun -N 1 ./testomp

Thread 0 says: Hello World on core 2
Thread 0 reports: the number of threads are 4
Thread 3 says: Hello World on core 1
Thread 2 says: Hello World on core 0
Thread 1 says: Hello World on core 3
```

Sans surprise on retrouve le même genre de sortie. On notera que la variable d'environnement **OMP_NUM_THREADS** a été propagée sur le nœud de calcul choisi à l'exécution.

Phase 4 : OpenMPI

Par opposition à **OpenMP**, **OpenMPI** est utilisé dans les infrastructures à mémoire distribuée.

OpenMPI crée une infrastructure de communication au-dessus du réseau physique (Ethernet, Infiniband etc.) qui abstrait le réseau au niveau du programmeur. Celui-ci peut alors déployer des processus MPI un peu partout sur l'infrastructure sans se soucier du placement ou de la gestion des connexions entre les différentes machines de son infrastructure. On va récupérer les sources du dernier **OpenMPI** stable, les compiler et les installer dans `/usr/local/compiled/openmpi`. On notera qu'il existe une option de compilation `--with-slurm` qui optimise les performances de **OpenMPI** lorsque celui-ci est utilisé avec SLURM.

L'Arch User Repository (AUR) propose un package openmpi-slurm. Pour le construire cloner le répertoire git puis exécuter `makepkg -sri` dans le répertoire.

```
wget
http://www.open-mpi.org/software/ompi/v1.8/downloads/openmpi-1.8.5.tar.gz
tar -xvzf openmpi-1.8.5.tar.gz && cd  openmpi-1.8.5
./configure --prefix=/usr/local/compiled/openmpi-1.8.5 --with-slurm
make && make install
ln -s /usr/local/compiled/openmpi-1.8.5 /usr/local/compiled/openmpi
```

Il faut également ajouter les librairies **OpenMPI** dans le cache des librairies sur le nœud frontal :

```
echo '/usr/local/compiled/openmpi' > /etc/ld.co.conf.d/hpc.conf
ldconfig
ldconfig -p | grep -i mpi
```

Il faut répéter cette manipulation sur les nœuds de calcul. C'est bien utile kash non ?

Voici un petit programme qui crée simplement des processus MPI qui affichent un « Hello world » et la machine depuis laquelle ils nous font leur petit coucou :

```
#include <stdio.h>
#include <mpi.h>
int main (argc, argv) {
  int rank, size;
  char hostname[40];
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  gethostname(hostname,40);
  printf( "Hello world from process %d of %d at %s\n", rank, size, hostname
);
  MPI_Finalize();
  return 0 ; }
```

Repasser en simple utilisateur pour tester ce programme. Commencer par fixer le PATH afin d'avoir les exécutables **OpenMPI**. La compilation des programmes **OpenMPI** se fait avec **mpicc** qui est un

wrapper autour de gcc avec les bonnes librairies et les includes positionnés.

```
export PATH=/usr/local/compiled/openmpi/bin:$PATH
mpicc testmpi.c -o testmpi
```

Pour exécuter un programme **OpenMPI**, il faut s'appuyer sur le lanceur **mpiexec**:

```
mpiexec -n 4 ./test mpi

Hello world from process 0 of 4 at login
Hello world from process 1 of 4 at login
Hello world from process 2 of 4 at login
Hello world from process 3 of 4 at login
```

On veut créer quatre processus MPI (-n 4) qui exécutent le programme. Ce programme affiche le rang (identifiant) du processus et sort. C'est par le rang que les processus communiquent entre eux. Par exemple on pourrait avoir un programme qui est organisé de la manière suivante :

- si je suis le processus de rang , je découpe mon jeu de données en N-1 parties et j'envoie la partie 1 au processus de rang 1, la partie 2 au processus de rang 2 etc.
- si je suis le processus de rang N j'attends mon jeu de données provenant du processus de rang , je le traite et le renvoie au processus de rang .

Nous voyons que ce type de programmation ne fait pas apparaître de notion d'IP, port, socket etc. La couche réseau est abstraite. Lançons le même programme par le job scheduler. Cette fois-ci le programme OpenMPI va s'exécuter sur les nœuds de calcul :

```
srun --resv-ports --mpi=pmi2 -n 4 ./testmpi

Hello world from process 1 of 4 at compute1
Hello world from process 2 of 4 at compute1
Hello world from process 0 of 4 at compute1
Hello world from process 3 of 4 at compute1
```

Tous est exécuté sur le nœud compute1 car celui-ci a suffisamment de cœurs libres pour encaisser les quatre processus MPI. Cependant en jouant avec les paramètres de réservation de ressource de SLURM il est possible de changer ce comportement :

```
srun --resv-ports --mpi=pmi2 -N 2 --ntasks-per-node=2 ./testmpi

Hello world from process 1 of 4 at compute1
Hello world from process 0 of 4 at compute2
Hello world from process 2 of 4 at compute2
Hello world from process 3 of 4 at compute1
```

Cette fois ci, l'exécution du programme est distribuée sur plusieurs nœuds de calcul car on lui demande deux nœuds (-N 2) et surtout deux tâches par nœud (–ntasks-per-node=2). Si on n'avait pas précisé le nombre de tâches par nœud et juste réservé deux nœuds, SLURM aurait tout mis sur le

même car c'est plus intéressant de tout avoir sur la même machine s'il y a suffisamment de cœurs libres.

Résultat : Job en mode batch

Le fonctionnement en mode batch demande à l'utilisateur de rédiger un script de soumission. C'est de cette manière qu'il faut utiliser un cluster HPC. Ce script contient deux sections : les ressources demandées et la façon de lancer le programme. Les variables destinées à SLURM sont précédées d'un #SBATCH.

```
#!/bin/bash
#SBATCH --job-name=hello
#SBATCH --output=slurm.out
#SBATCH --error=slurm.err
#SBATCH --mail-type=end
#SBATCH --mail-user=nicolas.greneche@univ-paris13.fr
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=2

srun --resv-ports --mpi=pmi2 ./testmpi
```

Dans cet exemple, un nom est donné au job (–job-name) un fichier de sortie standard et erreur est spécifié (–output et –error), une notification mail de fin de job est placée (–mail-type et –mail-user) et enfin les ressources demandées sont décrites (–nodes et –ntasks-per-node). Il ne reste plus qu'à soumettre ce job via la commande **sbatch** :

```
sbatch run.slurm

Submitted batch job 164515
```

Le job scheduler renvoi un JOBID. La commande squeue permet de vérifier l'état de la file d'attente des jobs :

```
squeue

  JOBID PARTITION     NAME     USER  ST       TIME  NODES NODELIST(REASON)
 164515   COMPUTE    hello     nico   R 0-00:01:25      2 compute[1-2]
```

On voit le job en exécution (ST=R) sur la partition COMPUTE. On peut également voir sur la sortie du fichier slurm.out :

```
login:~$ cat slurm.out

Hello world from process 0 of 4 at compute1
Hello world from process 2 of 4 at compute2
```

```
Hello world from process 1 of 4 at compute1
Hello world from process 3 of 4 at compute2
```

Sans surprise on retrouve le même genre de sortie qu'avec **srun**.