Universidad de Alcalá

DEPARTAMENTO DE AUTOMÁTICA
ARQUITECTURA Y TECNOLOGÍA DE COMPUTADORES

Grado en Ingeniería de Computadores
COMPUTACIÓN DE ALTAS PRESTACIONES

SLURM: gestor de carga computacional para el cluster

Introducción al uso de SLURM

Resumen

SLURM es un gestor de clusters y, a la vez, un sistema de distribución de trabajos para clusters Linux de pequeño y gran tamaño, con capacidad de tolerancia a fallos y alta escalabilidad. SLURM no exige cambios en el kernel y es, más o menos, auto-contenido. Como sistema de distribución de trabajos, SLURM realiza tres funciones básicas:

- reserva a un usuario el acceso exclusivo (o no exclusivo) a los recursos (típicamente, los nodos de cómputo) por un periodo de tiempo, para que pueda llevar a cabo un trabajo (*job*);
- proporciona un entorno para manejar ese trabajo: arrancarlo, pararlo, monitorizarlo, etc;
- arbitra el uso de los recursos entre todos los usuarios, típicamente manejando una cola de trabajos pendientes, de entre los que va seleccionando los que se convierten en candidatos a ejecutar, al tiempo que maximiza el uso de los recursos (los nodos de cómputo).

1. Arquitectura

SLURM ejecuta un *daemon*, llamado `slurmd`, en cada nodo de cómputo, y un *daemon* central, llamado `slurmctld` en un nodo de gestión. Los procesos `slurmd`s proporcionan toda la comunicación con el nodo de gestión y realizan las tareas de arranque y monitorización de los trabajos de los usuarios en cada nodo gestionado.

Las entidades que manejan los *daemons* de SLURM incluyen

- nodos, que son el recurso básico de computación;
- particiones, que agrupan los nodos en ciertos conjuntos (no necesariamente disjuntos);
- trabajos o *jobs*, que constituyen reservas de recursos asignados a un usuario para realizar ciertas tareas;
- *job steps*, que son cada una de las tareas singulares (potencialmente ejecutables en paralelo) dentro de un trabajo o *job*.

Las particiones pueden verse también como *colas* a las que los usuarios envían sus trabajos. Cada una está definida mediante un conjunto de restricciones: usuarios a los que se les permite

usarla, tiempo máximo que un usuario puede reservar para ejecutar su trabajo, tamaño máximo de un trabajo (es decir, cantidad máxima de memoria que puede usar), etc.

Entendemos un *trabajo* como una lista de una o más órdenes atómicas (que pueden ser un comando del sistema operativo, una aplicación de usuario, etc) que se deben ejecutar sucesivamente de principio a fin. Frecuentemente, si el trabajo es largo, esas órdenes se agrupan en un fichero, que llamamos guión o *script*. En la terminología de SLURM cada orden atómica recibe el nombre de *job step*.

Los trabajos se encolan en principio por orden de llegada, pero pueden ser priorizados (si al usuario le está permitido). El gestor analiza sucesivamente los recursos necesarios para cada trabajo y, si están disponibles, genera una reserva de ellos y permite el comienzo de la ejecución. Este paso se repite hasta que se acaban o bien los recursos o bien los trabajos encolados.

Cuando un trabajo termina, libera sus recursos. Si aún quedan trabajos pendientes, el gestor analiza ordenadamente si algún trabajo puede arrancar usando los recursos ahora disponibles, repitiendo básicamente los mismos pasos que antes.

2. Comandos

Se pueden consultar las páginas del manual `man` para ver con detalle cómo funcionan los comandos. Ahora daremos una pequeña lista de los más imprescindibles, explicando muy resumidamente su función.

- `sinfo`: informa acerca del estado de las particiones y nodos bajo control de SLURM.
- `srun`: este comando lo usamos típicamente para ejecutar un *job step* en paralelo sobre varias máquinas de una partición. El comando admite una serie de argumentos que permiten especificar cuántos procesos paralelos se requieren, qué recursos son necesarios, etc.
- `sbatch`: es el comando más interesante, que se usa para enviar a la cola un trabajo (*job*) para posterior ejecución. El trabajo se expresa por medio de un fichero que contiene líneas con detalles para la ejecución del `sbatch`; y con los comandos en los que consiste el trabajo, a razón de una línea por comando.

  Es frecuente que alguno de los *job steps* se lance mediante `srun`, con lo que los detalles de la ejecución se aplicarán a ese comando también: por ejemplo, cuántos procesos paralelos necesitamos, o cuántos nodos.
- `squeue`: informa sobre el estado de los trabajos que están encolados en un momento dado. La salida se puede filtrar según interese: por usuario, por cola, etc. Si no se dice nada especial, informa por orden de prioridad acerca de los trabajos que se estén ejecutando, y de los que estén en espera.
- `scancel`: es un comando que se usa para cancelar un trabajo pendiente o uno que ya se esté ejecutando. También se puede enviar una señal del sistema operativo a los procesos asociados con un *job* o *job step*.
- `scontrol`: es un comando de gestión que permite ver o modificar (si se tienen permisos administrativos) la configuración de SLURM.

3. Uso de los comandos

En esta sección usaremos los comandos más típicos en un escenario simplificado. No hay que olvidar que cada comando tiene literalmente docenas de posibilidades de uso, a través de los pa-

rámetros de invocación, que deben consultarse en el man. Aquí solo daremos unas pinceladas para introducirnos suavemente.

Los ejemplos suponen un usuario (rduran) conectado al nodo *front-end* sesamo, que teclea un comando y obtiene la respuesta. Téngase en cuenta que la mayoría de los comandos pueden abreviarse.

3.1. *Cómo ver información sobre el cluster*

Antes que nada, debemos averiguar qué particiones o colas existen en nuestro sistema, de qué nodos constan y en qué estado se encuentran. Si usamos el comando sinfo, obtenemos:

```
rduran@sesamo:~$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
Sesamo*      up    1:00:00      2   idle Blas,Epi
Fujis        up    1:00:00      4   idle fuji-s[0-3]
Lupes        up    1:00:00      4   idle Lupe-s[0-3]
HPC          up    1:00:00     10   idle Blas,Epi,Lupe-s[0-3],fuji-s[0-3]
```

Tenemos cuatro particiones, que corresponden en realidad a tres grupos de máquinas (las que integran Sesamo, Fujis, y Lupes), y un cuarto grupo que engloba a todas. El '*' que sigue a Sesamo indica que esa es la partición por defecto, la que recibirá cualquier trabajo que se envíe, salvo que se indique explícitamente otra partición. Vemos también que están en estado up y el número de nodos que integra cada una. Los nodos están idle, es decir, no trabajando. Todas las particiones tienen un tiempo máximo de CPU permitido de 1 hora. La sintaxis fuji-s[0-3] indica que esa partición consta de los nodos fuji-s0, fuji-s1, fuji-s2, y fuji-s3. El comando sinfo tiene muchas opciones que permiten ver más o menos información.

Podemos ver una información más detallada de todo el sistema con el comando

```
scontrol show partitions
```

que, en nuestro caso, nos dice lo siguiente:

```
rduran@sesamo:~$ scontrol show partitions
PartitionName=Sesamo
   AllowGroups=ALL AllowAccounts=ALL AllowQos=ALL
   AllocNodes=ALL Default=YES
   DefaultTime=NONE DisableRootJobs=NO GraceTime=0 Hidden=NO
   MaxNodes=UNLIMITED MaxTime=01:00:00 MinNodes=1 LLN=NO MaxCPUsPerNode=UNLIMITED
   Nodes=Epi,Blas
   Priority=1 RootOnly=NO ReqResv=NO Shared=NO PreemptMode=OFF
   State=UP TotalCPUs=16 TotalNodes=2 SelectTypeParameters=N/A
   DefMemPerNode=UNLIMITED MaxMemPerNode=UNLIMITED

PartitionName=Fujis
   AllowGroups=ALL AllowAccounts=ALL AllowQos=ALL
   AllocNodes=ALL Default=NO
   DefaultTime=NONE DisableRootJobs=NO GraceTime=0 Hidden=NO
   MaxNodes=UNLIMITED MaxTime=01:00:00 MinNodes=1 LLN=NO MaxCPUsPerNode=UNLIMITED
   Nodes=fuji-s[0-3]
   Priority=1 RootOnly=NO ReqResv=NO Shared=NO PreemptMode=OFF
   State=UP TotalCPUs=16 TotalNodes=4 SelectTypeParameters=N/A
   DefMemPerNode=UNLIMITED MaxMemPerNode=UNLIMITED

PartitionName=Lupes
```

```
   AllowGroups=ALL AllowAccounts=ALL AllowQos=ALL
   AllocNodes=ALL Default=NO
   DefaultTime=NONE DisableRootJobs=NO GraceTime=0 Hidden=NO
   MaxNodes=UNLIMITED MaxTime=01:00:00 MinNodes=1 LLN=NO MaxCPUsPerNode=UNLIMITED
   Nodes=Lupe-s[0-3]
   Priority=1 RootOnly=NO ReqResv=NO Shared=NO PreemptMode=OFF
   State=UP TotalCPUs=32 TotalNodes=4 SelectTypeParameters=N/A
   DefMemPerNode=UNLIMITED MaxMemPerNode=UNLIMITED

PartitionName=HPC
   AllowGroups=ALL AllowAccounts=ALL AllowQos=ALL
   AllocNodes=ALL Default=NO
   DefaultTime=NONE DisableRootJobs=NO GraceTime=0 Hidden=NO
   MaxNodes=UNLIMITED MaxTime=01:00:00 MinNodes=1 LLN=NO MaxCPUsPerNode=UNLIMITED
   Nodes=Epi,Blas,fuji-s[0-3],Lupe-s[0-3]
   Priority=1 RootOnly=NO ReqResv=NO Shared=NO PreemptMode=OFF
   State=UP TotalCPUs=64 TotalNodes=10 SelectTypeParameters=N/A
   DefMemPerNode=UNLIMITED MaxMemPerNode=UNLIMITED
```

Información detallada de un nodo en particular la obtenemos con

```
rduran@sesamo:~$ scontrol show node Lupe-s0
NodeName=Lupe-s0 Arch=x86_64 CoresPerSocket=4
   CPUAlloc=0 CPUErr=0 CPUTot=8 CPULoad=0.02 Features=(null)
   Gres=(null)
   NodeAddr=Lupe-s0 NodeHostName=Lupe-s0 Version=14.03
   OS=Linux RealMemory=12038 AllocMem=0 Sockets=2 Boards=1
   State=IDLE ThreadsPerCore=1 TmpDisk=107035 Weight=1
   BootTime=2017-03-13T19:15:17 SlurmdStartTime=2017-03-17T20:16:20
   CurrentWatts=0 LowestJoules=0 ConsumedJoules=0
   ExtSensorsJoules=n/s ExtSensorsWatts=0 ExtSensorsTemp=n/s
```

Este comando nos muestra, para el nodo `Lupe-s0`, el número de CPUs totales, el número de zócalos (e, indirectamente, el número de CPUs por zócalo), la memoria (en MB), hora de arranque de la máquina, hora de arranque del *daemon* `slurmd`, etc.

3.2. *Cómo ver qué trabajos están en ejecución*

Usamos el comando `squeue -l` para ver qué trabajos se están ejecutando en este momento:

```
rduran@sesamo:~$ squeue -l
Fri Mar 17 18:43:05 2017
JOBID PARTITION NAME     USER   STATE   TIME TIMELIMIT NODES NODELIST(REASON)
349   Lupes     cnbmmm.s rduran PENDING 0:00 1:00:00   4     (Resources)
348   Lupes     cnbmmm.s rduran RUNNING 0:02 1:00:00   4     Lupe-s[0-3]
```

Vemos que hay dos trabajos en la partición `Lupes`, uno en ejecución y otro en espera. El campo de la derecha nos indica la razón para la espera: que no hay recursos disponibles. En efecto, el trabajo `349` requiere 4 nodos de la partición pero los 4 nodos están siendo utilizados por el trabajo `348`. Mientras este no finalice, aquel seguirá en espera. También este comando tiene multitud de opciones: en el ejemplo hemos usado la opción '`-l`' que nos proporciona una información más ampliada.

Veamos otro ejemplo de estado de particiones:

```
rduran@sesamo:~$ squeue -l
Fri Mar 17 19:03:15 2017
```

```
JOBID       PARTITION NAME     USER   STATE   TIME TIMELIMIT NODES NODELIST(REASON)
353_[2,4,8] Fujis     computar rduran PENDING 0:00 1:00:00   4     (Resources)
353_1       Fujis     computar rduran RUNNING 0:02 1:00:00   4     fuji-s[0-3]
352         Lupes     cnbmmm.s rduran RUNNING 0:02 1:00:00   4     Lupe-s[0-3]
```

En este caso, la partición `Lupes` está ejecutando un trabajo y, simultáneamente, también lo está haciendo la partición `Fujis`. Pero, además, esta última tiene en espera una lista de trabajos identificados como `353_[2,4,8]`, es decir, son en realidad tres trabajos en espera: `353_2`, `353_4` y `353_8`, cada uno de los cuales requiere un total de 4 nodos y, por tanto, se ejecutarán sucesivamente.

Si en ese mismo momento usamos el comando `sinfo`, obtenemos:

```
rduran@sesamo:~$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
Sesamo*      up    1:00:00      2   idle Blas,Epi
Fujis        up    1:00:00      4  alloc fuji-s[0-3]
Lupes        up    1:00:00      4  alloc Lupe-s[0-3]
HPC          up    1:00:00      8  alloc Lupe-s[0-3],fuji-s[0-3]
HPC          up    1:00:00      2   idle Blas,Epi
```

Vemos un resumen de la situación del cluster, diferenciado por particiones. La partición `HPC` está en parte activa y en parte en reposo.

3.3. *Cómo enviar trabajos a ejecutar*

El comando más interesante es el que usamos para remitir trabajos a ejecutar: se trata de `sbatch`. Este comando precisa de un fichero de comandos (un guión, o *script*) en donde debemos establecer

- los recursos que es necesario reservar para ejecutar el trabajo;
- los pasos de computación que componen el trabajo, a razón de un paso por línea.

Es importante comprender cómo se reservan los recursos ya que esto es lo que va a permitir que, cuando nuestro trabajo comience a ejecutarse, se le garantice el uso exclusivo de esos recursos. Como ejemplo sencillo, consideremos el siguiente fichero guión, al que damos el nombre de '`host.script`':

```
#!/bin/bash
#SBATCH -p Lupes # partición
#SBATCH --ntasks 16
#SBATCH --ntasks-per-node 4
#SBATCH -o slurm.%N.%j.out # salida estándar
#SBATCH -e slurm.%N.%j.err # salida de error
srun -l hostname
exit 0
```

El guión comienza con la invocación del `bash` bajo el cual se ejecutan el resto de los comandos. Inmediatamente (esto es importante) siguen las directivas para la reserva de recursos que van dirigidas al sistema SLURM. En este caso establecemos que queremos utilizar los nodos de la partición `Lupes`. Para que el sistema realice bien la reserva, le informamos de que uno o más pasos de ejecución lanzarán 16 procesos y que queremos que no haya más que, como máximo, 4 procesos por nodo.

Observemos que, en este caso, el paso que va a lanzar los 16 procesos es precisamente el `srun`, pues tiene la capacidad de lanzar procesos en paralelo (tantos como se le hayan indicado). Como hemos solicitado que no haya más de 4 procesos por nodo, el trabajo solo comenzará cuando sea posible reservar 4 nodos (en este caso, son todos los que tiene la partición). Si, por el contrario, no hubiéramos solicitado ese máximo (si eliminamos la línea `#SBATCH --ntasks-per-node 4`), el

trabajo podría comenzar cuando simplemente haya dos nodos desocupados en esa partición, pues cada nodo goza de 8 CPUs. Tengamos en cuenta que, salvo que se diga lo contrario explícitamente, el sistema nunca arranca en un nodo más tareas que el número de CPUs que tenga ese nodo.

Es importante comprender que la reserva de recursos es independiente del uso que luego hagan de ellos los pasos de ejecución. El usuario establece el mínimo de recursos lo que provocará la retención del trabajo mientras ese conjunto de recursos no esté disponible: pero eso no quiere decir que, de hecho, la ejecución haga uso de todos. Es responsabilidad del usuario reservar los recursos juiciosamente, en beneficio propio y ajeno.

Las directivas

```
#SBATCH -o slurm.%N.%j.out # salida estándar
#SBATCH -e slurm.%N.%j.err # salida de error
```

son interesantes pues nos muestran cómo construir nombres para los ficheros en que guardarán la salida estándar y la salida de error a partir de dos parámetros que reciben su valor en tiempo de ejecución: el parámetro `%N` representa el nombre del nodo ejecutor del guión, y el parámetro `%j` representa el `JOBID` que el sistema asigne al trabajo.

Remitimos el guión para ejecución tecleando

```
rduran@sesamo:~$ sbatch host.script
Submitted batch job 409
```

El sistema ha contestado informando de que ha sido encolado con la identificación 409. Podemos ver el estado de ejecución tecleando

```
rduran@sesamo:~$ squeue -l
Tue Mar 28 13:37:27 2017
JOBID PARTITION NAME     USER   STATE   TIME TIMELIMIT NODES NODELIST(REASON)
409   Lupes     host.scr rduran RUNNING 0:00 1:00:00   4     Lupe-s[0-3]
```

Al cabo de un pequeño espacio de tiempo, el trabajo termina y observamos cómo en el directorio actual aparecen dos archivos, '`slurm.Lupe-s0.409.out`' y '`slurm.Lupe-s0.409.err`', este último vacío, pues no ha habido errores. El contenido de la salida es

```
rduran@sesamo:~$ cat slurm.Lupe-s0.409.out
00: Lupe-s0
01: Lupe-s0
02: Lupe-s0
03: Lupe-s0
04: Lupe-s1
05: Lupe-s1
06: Lupe-s1
07: Lupe-s1
08: Lupe-s2
09: Lupe-s2
10: Lupe-s2
11: Lupe-s2
12: Lupe-s3
13: Lupe-s3
14: Lupe-s3
15: Lupe-s3
```

La salida responde a lo que hemos solicitado: ejecutar un total de 16 procesos, a razón de 4 procesos por nodo: y así vemos cómo los procesos que el `srun` nos numera (nos los numera porque así lo hemos solicitado mediante el parámetro '`-l`') como `00-03` van a ejecutarse a `Lupe-s0`, los siguientes cuatro a `Lupe-s1`, etc.

Observemos también que el fichero guión, como tal, que ejecuta una instancia del comando `bash` se ejecuta en el primer nodo de la partición `Lupes`, razón por la cual el fichero de salida lleva como parte del nombre dicho nodo, a saber, `Lupe-s0`.

3.4. *Cómo ejecutar comandos interactivos*

Se trata de poder lanzar un comando (típicamente, un programa) sobre una partición de manera que, si existen recursos suficientes, se ejecuta en paralelo el número de tareas que se le especifique y en la partición que se le especifique a través de los parámetros.

El uso que del comando `srun` se ha hecho más arriba, en el fichero guión, puede hacerse de modo directo, pasando las directivas directamente como argumentos, de la siguiente manera

```
rduran@sesamo:~$ srun -p Lupes --ntasks 16 --tasks-per-node 4 -l hostname
00: Lupe-s0
01: Lupe-s0
02: Lupe-s0
03: Lupe-s0
04: Lupe-s1
05: Lupe-s1
06: Lupe-s1
07: Lupe-s1
08: Lupe-s2
09: Lupe-s2
10: Lupe-s2
11: Lupe-s2
12: Lupe-s3
13: Lupe-s3
14: Lupe-s3
15: Lupe-s3
```

Vemos cómo la salida producida es la misma que la que encontramos en el fichero de salida del `sbatch`.

Naturalmente, la interactividad solo es posible si los recursos están disponibles. En caso contrario, el comando nos informa de que estamos a la espera de la liberación de recursos y la ejecución queda bloqueada. Por ejemplo, podríamos tener este resultado

```
rduran@sesamo:~$ srun -p Lupes --ntasks 16 --tasks-per-node 4 -l hostname
srun: job 412 queued and waiting for resources
```

(aquí transcurre cierto tiempo de espera...)

```
srun: job 412 has been allocated resources
00: Lupe-s0
01: Lupe-s0
02: Lupe-s0
03: Lupe-s0
04: Lupe-s1
05: Lupe-s1
06: Lupe-s1
07: Lupe-s1
08: Lupe-s2
09: Lupe-s2
10: Lupe-s2
11: Lupe-s2
12: Lupe-s3
13: Lupe-s3
```

```
14: Lupe-s3
15: Lupe-s3
```

En el ejemplo anterior, el trabajo identificado con el número `JOBID 412` sufre una espera bloqueante, pero en cuanto los recursos están disponibles, pasa a ejecución. El comando `srun` resulta interesante para probar y depurar nuestros programas sin necesidad de remitirlos mediante un fichero guión. Pero no olvidemos que este comando es bloqueante y la espera puede tener una duración indefinida.

3.5. *Cómo ejecutar programas desarrollados para* MPI

El sistema SLURM dispone de una comunicación especial con la librería MPI que es específica para desarrollar programas mediante el paradigma de paso de mensajes. Esto se manifiesta en el significado particular de algunas de las directivas que hemos visto antes cuando se trata de ejecutar programas desarrollados con esta librería.

Supongamos que hemos desarrollado un ejecutable con el nombre `cnbmmm` utilizando el paradigma de paso de mensajes con la librería MPI. Hacemos un guión '`cnbmmm.script`' con los siguientes comandos y directivas

```
#!/bin/bash
#SBATCH -p Lupes
#SBATCH -o cnbmmm.%N.%j.out
#SBATCH -e cnbmmm.%N.%j.err
#SBATCH --ntasks 16
#SBATCH --ntasks-per-node 4
module load mpe2
mpiexec ./cnbmmm
exit 0
```

El sistema SLURM se comunica con el comando `mpiexec` de la librería MPI, de modo que el número de procesos que se arrancarán serán justo los indicados mediante la directiva `--ntasks`.

Como cada nodo de la partición `Lupes` tiene 8 CPUs, si pretendemos que el trabajo use los cuatro nodos de la partición, a razón de 4 procesos por nodo, es necesaria la directiva `--ntasks-per-node`. Sin esa directiva, el sistema ejecutaría 8 procesos por nodo, pues de ese modo le bastaría reservar dos nodos para nuestro trabajo, en vez de cuatro. En cambio, con la directiva activada, le obligamos a utilizar cuatro nodos, como queríamos.

Observemos que en nuestro cluster es necesaria la orden

```
module load mpe2
```

para que se carguen las variables de entorno necesarias en aplicaciones que usan la librería MPI.

3.6. *Cómo cancelar trabajos en cola o en ejecución*

El comando `scancel` se invoca pasándole como argumento simplemente el parámetro `JOBID` que identifica el trabajo o paso que queremos cancelar. Esta operación se puede hacer tanto si está aún en espera (con lo que nunca pasaría a ejecutarse) como si ya está en ejecución, en el cual caso el gestor interrumpe el proceso en el punto en que se encuentre.