Using Slurm

Overview

In Guide 7.2 we worked through installing and testing the Slurm workload manager. In this guide we'll get a brief introduction to how to use Slurm from the user perspective.

Anatomy of a Slurm job

A Slurm job consists of an allocation and a number of steps that occur within the allocation. An allocation can be gained directly by use of the `salloc` command. It can also be obtained through use of a batch script and the `sbatch` command.

Steps within a job are allocated through the `srun` command. One allocation may consists of many steps.

An allocation (`salloc`) is granted a specified amount of resources. For our purposes, we'll stick to three resources: nodes, CPUs (cores), and memory.

A step (`srun`) is granted a certain number of resources within the allocation. By default, it will be granted all of the resources of the allocation, but it doesn't have to be.

An `salloc` command has the structure:

```
salloc [options] <command>
```

Some basic options are:

- `-N` # - number of nodes
- `-n` # - number of tasks (parallel)
- `-c` # - number of CPUs per task
- `--mem=#` - memory per node
- `--mem-per-cpu=#` - memory per CPU
- `--time` or `-t` - limit on runtime

The `<command>` gets executed locally on the host where the allocation is requested. In this case, its job is primarily to coordinate steps `sruns` to run on the computing resources.

The allocation is bound to the lifetime of the `<command>`. When the `<command>` exits, the allocated resources are returned to the free resource pool.

The format of `srun` is similar to `salloc`:

```
srun [options] <command>
```

Many of the same resource selections apply as `salloc`, but the resources are taken from within the existing allocation.

- `-N` # - number of nodes
- `-n` # - number of tasks (parallel)
- `-c` # - number of CPUs per task
- `--mem=#` - memory per node
- `--mem-per-cpu=#` - memory per CPU
- `--time` or `-t` - limit on runtime
- `--pty` - allocate an interactive session inside the allocation

We can call `srun` multiple times within one allocation. If our `srun` does not use all of the resources of our allocation, we could have multiple `srun`'s running at the same time.

If the `--pty` otpion is given, the `<command>` is allocated one of the allocated hosts. In this case, it gives us interactive access to a compute host.

When we run an `srun` with `-N` or `-n > 1`, the process is launched that many times.

Step 1: Simple interactive session

Suppose we want to just poke around on a compute node a bit. We don't need significant resources, and we only need one CPU. This will need to be an interactive session then. We can do this with:

```
[lowell@te-master ~]$ salloc -n1 /bin/bash
salloc: Granted job allocation 9
[lowell@te-master ~]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES
NODELIST(REASON)
                 9   cluster     bash   lowell  R       0:01      1 te01
[lowell@te-master ~]$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
cluster*     up   infinite      1    mix te01
cluster*     up   infinite      8   idle te[02-10]
[lowell@te-master ~]$ srun --pty /bin/bash
[lowell@te01 ~]$ uptime
 20:07:36 up 11:05,  0 users,  load average: 0.00, 0.01, 0.05
[lowell@te01 ~]$ exit
exit
[lowell@te-master ~]$ exit
exit
salloc: Relinquishing job allocation 9
```

In this command we allocated one CPU and our allocation process was `bash`. We see our running allocation with the `squeue` command. We see in `sinfo` command we see that `te01` is in the state `mix`. This means that the node has an allocation running on it, but parts of the node are idle. That happened because we only requested one CPU, so all of the rest are free.

We then ran the `srun` command with the `--pty` option, again running `/bin/bash`. This gave us an interactive step on the actual node.

Finally, we exited both shells and were told that the allocation was relinquished.

We could have simplified this by making `srun` the command we run with `salloc`. In fact this is commonly done:

```
[lowell@te-master ~]$ salloc -n1 srun --pty /bin/bash
salloc: Granted job allocation 19
[lowell@te01 ~]$ exit
exit
salloc: Relinquishing job allocation 19
```

In fact, if we look in our `slurm.conf`, this is set to be the default action if we give an `salloc` without a command:

```
[lowell@te-master ~]$ grep SallocDefaultCommand /etc/slurm/slurm.conf
SallocDefaultCommand="/usr/bin/srun -n1 -N1 --mem-per-cpu=0 --pty --
preserve-env --mpi=none $SHELL"
```

So, really, we can just do:

```
[lowell@te-master ~]$ salloc
salloc: Granted job allocation 20
[lowell@te01 ~]$ exit
exit
salloc: Relinquishing job allocation 20
```

Note that this doesn't have to be the default command; we set it up this way.

Step 2: Running a multi-processor application

Suppose we want our application to run on 8 processors, with 1 GB memory per processor. We can make an allocation like this with:

```
[root@te-master ~]# salloc -n8 --mem-per-cpu=1 -N1 /bin/bash
salloc: Granted job allocation 59
[root@te-master ~]# srun hostname
te01
```

```
te01
te01
te01
te01
te01
te01
```

We specified both `-n8` and `-N1` to make sure that we get all 8 tasks on one node.

Notice that `srun` will, by default, launch one tasks for each allocated task. We can control this with `--cpus-per-task`, or `-c`.

```
[root@te-master ~]# salloc -n1 -N1 -c8 /bin/bash
salloc: Granted job allocation 62
[root@te-master ~]# srun hostname
te01
```

In this example, we got 1 task, but 8 CPUs in the task. This just means that Slurm found a node with 8 free CPUs for our single task.

We could also get 8 tasks on 4 nodes:

```
[root@te-master ~]# srun hostname
te04
te01
te03
te02
te01
te01
te01
te01
```

Notice that this did not evenly distribute our tasks across our nodes. We need to specifically request that if it is what we want.

```
[root@te-master ~]# salloc -n8 -N4 --tasks-per-node=2 /bin/bash
salloc: Granted job allocation 64
[root@te-master ~]# srun hostname
te01
te01
te04
te03
te02
te04
te02
te03
[root@te-master ~]# exit
```

```
exit
salloc: Relinquishing job allocation 64
```

Now there are two tasks on every node.

Using `-N`, `-n`, and `-c` allong with various `--<x>-per-<y>` style parameters we can control how our job gets allocated in quite a bit of detail.

Step 3: Batch scripts

Everything we've done so far requires more interaction than we would typically want to do. In reality, we usually want to bundle our jobs up in scripts that can run whenever the resources are available. These are "sbatch" scripts, and can be submitted with `sbatch`.

An sbatch script is an ordinary shell script, but it has the option of containing lines starting with `#SBATCH`. These lines can be followed with arbitrary `srun`/`salloc` command arguments. Usually, an sbatch script does all of its work in `srun` steps just like we have been doing.

Here is an example of a simple sbatch script:

```
#!/bin/bash

#SBATCH --job-name=Test
#SBATCH --ntasks=2
#SBATCH --nodes=1
#SBATCH --cpus-per-task=2
#SBATCH --mem-per-cpu=1
#SBATCH --output=test_%j.log
pwd; date

srun hostname
srun sleep 60

date
```

This sbatch script asks for a job with 2 tasks, 1 node, 2 cpus per task (a total of 4 CPUs), and 1 GB of memory per task. It is assigned the name "Test". A job name can help us identify jobs in the queue. We have also requested that output be sent to `test_<job_id>.log`. There are a number of macros like `%j` for job_id that can be used in these scripts.

While it's not necessary, it's common practice in sbatch scripts to:

- Have a single parameter per `#SBATCH` line. This makes the file easy to read and easy to edit.
- Use full-length arguments instead of short arguments (e.g. `--ntasks`, not `-n`). This improves readability.

We can submit this sbatch script (assuming it's called `test.sbatch`):

```
[lowell@te-master ~]$ sbatch test.sbatch
Submitted batch job 68
[lowell@te-master ~]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES
NODELIST(REASON)
                68   cluster     Test   lowell  R       0:02      1 te01
[lowell@te-master ~]$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
cluster*     up   infinite      1    mix te01
cluster*     up   infinite      9   idle te[02-10]
[lowell@te-master ~]$ scontrol show job 68
JobId=68 JobName=Test
   UserId=lowell(1000) GroupId=lowell(1000) MCS_label=N/A
   Priority=4294901693 Nice=0 Account=(null) QOS=(null)
   JobState=RUNNING Reason=None Dependency=(null)
   Requeue=1 Restarts=0 BatchFlag=1 Reboot=0 ExitCode=0:0
   RunTime=00:00:09 TimeLimit=UNLIMITED TimeMin=N/A
   SubmitTime=2019-06-14T04:26:35 EligibleTime=2019-06-14T04:26:35
   AccrueTime=Unknown
   StartTime=2019-06-14T04:26:35 EndTime=Unknown Deadline=N/A
   PreemptTime=None SuspendTime=None SecsPreSuspend=0
   LastSchedEval=2019-06-14T04:26:35
   Partition=cluster AllocNode:Sid=te-master:31714
   ReqNodeList=(null) ExcNodeList=(null)
   NodeList=te01
   BatchHost=te01
   NumNodes=1 NumCPUs=4 NumTasks=2 CPUs/Task=2 ReqB:S:C:T=0:0:*:*
   TRES=cpu=4,mem=4M,node=1,billing=4
   Socks/Node=* NtasksPerN:B:S:C=0:0:*:* CoreSpec=*
   MinCPUsNode=2 MinMemoryCPU=1M MinTmpDiskNode=0
   Features=(null) DelayBoot=00:00:00
   OverSubscribe=OK Contiguous=0 Licenses=(null) Network=(null)
   Command=/home/lowell/test.sbatch
   WorkDir=/home/lowell
   StdErr=/home/lowell/test_68.log
   StdIn=/dev/null
   StdOut=/home/lowell/test_68.log
   Power=
```

As you can see, `scontrol show job <job-id>` will tell you information about a specific job, including its resource limits.

After a minute, the job will complete.

```
[lowell@te-master ~]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES
NODELIST(REASON)
[lowell@te-master ~]$ cat test_68.log
/home/lowell
Fri Jun 14 04:26:36 MDT 2019
te01
```

```
te01
Fri Jun 14 04:27:36 MDT 2019
```

This shows us the output from our job.

We can potentially have a lot of jobs submitted at the same time. The scheduler will launch them as resources become available.

Suppose we run it again, but decide we don't want it anymore. We can `scancel` the job:

```
[lowell@te-master ~]$ sbatch test.sbatch
Submitted batch job 69
[lowell@te-master ~]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES
NODELIST(REASON)
                69   cluster     Test   lowell  R       0:02      1 te01
[lowell@te-master ~]$ scancel 69
[lowell@te-master ~]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES
NODELIST(REASON)
```

The output file for jobs is updated live. We can, for instance, `tail -f` the output of a running job. We can see this by noting that our canceled job still has an output file, but it is partial:

```
[lowell@te-master ~]$ cat test_69.log
/home/lowell
Fri Jun 14 04:29:07 MDT 2019
te01
te01
srun: Job step aborted: Waiting up to 32 seconds for job step to finish.
slurmstepd: error: *** STEP 69.1 ON te01 CANCELLED AT 2019-06-14T04:29:13
***
slurmstepd: error: *** JOB 69 ON te01 CANCELLED AT 2019-06-14T04:29:13 ***
```

Slurm also tells us that the job was canceled.

Let's modify our job a little to make it have more regular output. Let's call this one `test2.sbatch`:

```
#!/bin/bash

#SBATCH --job-name=Test2
#SBATCH --ntasks=1
#SBATCH --nodes=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=1
#SBATCH --output=test2_%j.log
pwd; date
```

```
srun hostname
srun /bin/bash -c 'for i in {1..60}; do echo $i; sleep 1; done'

date
```

Note that we adjusted our tasks/CPUs to just be one.

We can run this job, and while it's running attach to its output to see what's going on:

```
[lowell@te-master ~]$ sbatch test2.sbatch
Submitted batch job 71
[lowell@te-master ~]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES
NODELIST(REASON)
                71   cluster    Test2   lowell  R       0:02      1 te01
[lowell@te-master ~]$ sattach 71.1
1
2
3
4
5
6
7
8
9
10
...
```

Note that we must specify `<job_id>.<step_id>` for `sattach`, not just the `<job_id>`.

Conclusion

There are many features for submitting jobs not covered here, but this guide should provide a basic overview of how jobs are submitted in a Slurm cluster. Specifically, we haven't covered how to submit and use MPI jobs, but that will have to be covered later when we actually have an MPI application to run.

Meanwhile, your best resource for submitting slurm jobs are the `salloc`, `srun`, and `sbatch` man pages.