Using the Cluster – Advanced Usage

Now that you have mastered Slurm Basic Commands and ran a few jobs, you are ready for ***advanced usage***.

This document will demonstrate some more advanced usage such as how to write a “parameter-sweep” bash script that submits many jobs, how to submit your jobs at various Qualities-of-Service, and how to construct series of jobs that depends on each others completion to progress.

Looping

Researchers typically do not come to Research Computing to submit a single job, wait for it to complete, look at the results, and call it a day. More typically, new users come because they’ve reached a point where they have developed a model they think is feasible and they want to simulate it under different conditions (often under many different conditions!).

For instance, say that you had a model that (you think) predicts the weather and you have weather data over a 10 week period. You can write a script that submits 10 cluster jobs, each of which reads in the data from the previous week and predicts the weather for the next.

You may instead have a model that simulates a black hole. You could write a script that submits 1000 cluster jobs, each of which performs the simulation with a different quantity of initial mass for the simulation.

We’ll proceed here with a more abstract example that loops over *two* parameters, alpha and beta.

Change into the directory of example 2 from the example provided by running `grab-examples`:

```
[abc1234@sporcsumbit example-2-basic-looping []]$ cd ~/slurm-examples-2011-12-09/example-2-basic-looping/
[abc1234@sporcsubmit example-2-basic-looping []]$ ls -alh
total 128K
drwxr-x--- 2 abc1234 abc1234 2.0K Dec  9 16:36 .
drwxrwx--- 5 abc1234 abc1234 2.0K Dec  9 16:36 ..
-rwxr-x--- 1 abc1234 abc1234 1.1K Dec  9 16:36 slurm-payload.sh
-rwxr-x--- 1 abc1234 abc1234 1.3K Dec  9 16:36 submit-many-jobs.sh
```

Take a look first at slurm-payload.sh. It’s a Slurm submission file that contains metadata about a job, just like in example 1:

```
[abc1234@sporcsubmit example-2-basic-looping []]$ cat slurm-payload.sh
#!/bin/bash -l
# NOTE the -l flag!
#

# Where to send mail...
#SBATCH --mail-user abc1234@rit.edu

# notify on state change: BEGIN, END, FAIL or ALL
#SBATCH --mail-type=ALL

# Request 5 minutes run time MAX, anything over will be KILLED
#SBATCH -t 0:5:0

# Put the job in the "debug" partition and request one core
# You probably want to change "debug" to "tier1/tier2/tier3" once you have a sense of how
# this is working.
#SBATCH -p debug -n 1

# Job memory requirements in MB
#SBATCH --mem=300

echo "I am a job..."
echo "My value of alpha is $alpha"
echo "My value of beta is $beta"
echo "And now I'm going to simulate doing work based on those parameters..."

sleep 20

echo "All done with my work.  Exiting."
```

Nothing much new here, except for the appearance of $alpha and $beta. Those are bash environment variables that represent... something. It suffices to say that they can either have values and that they can be used to run your program and workload with particular parameters instead of the benign sleep 20 statement. That part is up to you!

Let’s take a look at the other (more interesting) file:

```
[abc1234@sporcsubmit example-2-basic-looping []]$ cat submit-many-jobs.sh
#!/bin/bash

# Just a constant variable used throughout the script to name our jobs
#   in a meaningful way.
basejobname="test"

# Another constant variable used to name the slurm submission file that
#   this script is going to submit to slurm.
jobfile="slurm-payload.sh"

param_limit_alpha=5
param_limit_beta=5

# Make an output directory if it doesn't already exist.
mkdir -p output

# Loop and submit all the jobs
echo
echo " * Getting ready to submit a number of jobs:"
echo
for alpha in $(seq 1 $param_limit_alpha); do
    for beta in $(seq 1 $param_limit_beta); do
        # Give our job a meaningful name
        jobname=$basejobname-$alpha-$beta
        echo "Submitting job $jobname"

        # Setup where we want the output from each job to go
        outfile=output/output-alpha.$alpha-beta.$beta.txt

        # "exporting" variables in bash make them available to your slurm
        # workload.
        export alpha;
        export beta;

        # Actually submit the job.
        sbatch --partition=<tier1/tier2/tier3> -J $jobname -o $outfile $jobfile
    done;
done

echo
echo " * Done submitting all those jobs (whew!)"
```

This file is a *standard* basic script (not a Slurm submission file itself). If you were to run it, it would set up some variables, enter a nested loop over two variables (our $alpha and $beta) and submit a job for each combination.

The -J some_name and -o some_file options give your job a name and an output file location, respectively. You've seen them before, but in a different context.

In example 1, these options were found inside the Slurm submission/payload script itself as #SBATCH meta commands. Now they appear as command-line options for the sbatch command itself. Every metacommand can be used interchangably as a command line option to sbatch and there are *a lot* of them. For the purposes of these tutorials, we’ve tried to prune down and show you only what’s necessary to get off the ground, but to see the whole list of options/metacommands, issue the command $ man sbatch to get the user-manual for Slurm submission.

Let’s see what this submit-many-jobs.sh script does. Give it a run:

```
[abc1234@sporcsubmit example-2-basic-looping []]$ ./submit-many-jobs.sh

 * Getting ready to submit a number of jobs:

Submitting job test-1-1
Submitted batch job 732
Submitting job test-1-2
Submitted batch job 733
... (snip) ...
Submitting job test-5-4
Submitted batch job 755
Submitting job test-5-5
Submitted batch job 756

 * Done submitting all those jobs (whew!)
```

Now check the slurm queue to see what kind of effect this had, if any. Run:

```
[abc1234@sprocsubmit example-2-basic-looping []]$ squeue
  JOBID PARTITION     NAME     USER  ST       TIME  NODES NODELIST(REASON)
    736     debug test-1-5   abc1234  PD       0:00      1 (Resources)
    737     debug test-2-1   abc1234  PD       0:00      1 (Priority)
    738     debug test-2-2   abc1234  PD       0:00      1 (Priority)
    739     debug test-2-3   abc1234  PD       0:00      1 (Priority)
    740     debug test-2-4   abc1234  PD       0:00      1 (Priority)
    741     debug test-2-5   abc1234  PD       0:00      1 (Priority)
    742     debug test-3-1   abc1234  PD       0:00      1 (Priority)
    743     debug test-3-2   abc1234  PD       0:00      1 (Priority)
    744     debug test-3-3   abc1234  PD       0:00      1 (Priority)
    745     debug test-3-4   abc1234  PD       0:00      1 (Priority)
    746     debug test-3-5   abc1234  PD       0:00      1 (Priority)
    747     debug test-4-1   abc1234  PD       0:00      1 (Priority)
    748     debug test-4-2   abc1234  PD       0:00      1 (Priority)
    749     debug test-4-3   abc1234  PD       0:00      1 (Priority)
    750     debug test-4-4   abc1234  PD       0:00      1 (Priority)
    751     debug test-4-5   abc1234  PD       0:00      1 (Priority)
    752     debug test-5-1   abc1234  PD       0:00      1 (Priority)
    753     debug test-5-2   abc1234  PD       0:00      1 (Priority)
    754     debug test-5-3   abc1234  PD       0:00      1 (Priority)
    755     debug test-5-4   abc1234  PD       0:00      1 (Priority)
    756     debug test-5-5   abc1234  PD       0:00      1 (Priority)
    734     debug test-1-3   abc1234   R       0:01      1 escher
    735     debug test-1-4   abc1234   R       0:01      1 escher
    732     debug test-1-1   abc1234   R       0:05      1 bach
    733     debug test-1-2   abc1234   R       0:05      1 bach
    731      work ONPNTN-I    luvmet   R   21:04:16      1 einstein
    730      work ONPNTN-I    luvmet   R   21:04:35      1 einstein
```

Great. We can see that 25 jobs were submitted, all to the debug partition. Four of them are running right now and the other 21 are in the *pending* state.

We set the jobs to each direct their own output to their own file in an output/ directory. Let's take a look:

```
[abc1234@tropos example-2-basic-looping []]$ ls -alh output/
total 192K
drwxrwx--- 2 abc1234 abc1234 2.0K Dec 14 10:39 .
drwxr-x--- 3 abc1234 abc1234 2.0K Dec 14 10:38 ..
-rw-rw---- 1 abc1234 abc1234  349 Dec 14 10:39 output-alpha.1-beta.1.txt
-rw-rw---- 1 abc1234 abc1234  219 Dec 14 10:39 output-alpha.1-beta.2.txt
-rw-rw---- 1 abc1234 abc1234  221 Dec 14 10:39 output-alpha.1-beta.3.txt
-rw-rw---- 1 abc1234 abc1234  221 Dec 14 10:39 output-alpha.1-beta.4.txt
```

And there are the first four output files, slowly accumulating output from their execution on the remote nodes bach and escher.

In conclusion, this template includes:

> 1. *A slurm payload script that describes constant metadata about each job and how to run it.*
> 2. *An outer-loop script that actually submits each parameterized job*

You can use the template and modify it to do the computations you're really setting out to do (instead of just printing out the parameters and sleeping for 20 seconds).

Dependent Jobs

Sometimes you are running a large number of jobs, some of which need to begin and complete in a specific order. The problem here is that one job might get stalled out on a slow operation (like I/O) and not complete before a subsequent and dependent job is scheduled to start.

To solve this, Slurm provides a --dependency flag which argues that a particular job should not start until another specified job has completed. Here is an example of the modified submit-many-jobs.sh script from example 1 that demonstrates its usage

```bash
#!/bin/bash
#
# DESCRIPTION:
#   Submit 15 jobs, each of which will depend on its predecessor.
#
# Authors:  Justin Talbot (jdt0127)
#           Ralph Bean (rjbpop)
#

# Just a constant variable used throughout the script to name our jobs
#   in a meaningful way.
jobname="test"

# Another constant variable used to name the slurm submission file that
#   this script is going to submit to slurm.
jobfile="slurm-job-file.sh"

# Stores the command to execute in the variable $command for easy reading
#   More information on this below inside the 'for' loop
outfile=output-iteration-0.out
command="sbatch --partition =<tier1/tier2/teir3> -o $outfile -e $outfile -J $jobname-0 $jobfile"

# Use the command above to submit our first job.  It doesn't depend on any
#   other jobs.  Its output to the terminal will be something like:
#                 Successfully submitted job 4023
#   We are going to use that output and grab the number 4023 (the fourth word
#   in the output, hence the $4) and stuff that number in the variable latest_id
latest_id=$($command | awk ' { print $4 }')

# Now submit the remaining jobs so that they depend on their predecessors.
# Loop from 1 to 14 putting the value in the variable $i
for i in `seq 1 14` ; do
    # Do a little output to see what's up as this executes.
    echo "Submitting my ${i}th job that depends on SLURM job $latest_id"

    # Name a file dynamically where we want all of our 'messages' to go.
    outfile=output-iteration-$i.out

    # Submit the next job and..
    #   1) Do not start the job until the last was finished SUCCESSFULLY
    #   2) Name it based on the sequence of jobs
    #   3) Use $outfile for standard output and error
    #   4) Use the job described in filename $jobfile
    # The backslashes '\' here only allow us to break the command into
    #   multiple lines to make it easier to read and understand as humans.
    sbatch \
            --partition=<tier1/tier2/tier3> \
            --dependency=afterok:$latest_id \
            -J $jobname-$i \
            --output=$outfile \
            --error=$outfile \
            $jobfile

    # Increment the job on which the next should depend by one
    latest_id=`echo $latest_id + 1 | bc`
done
```