


Princeton Research Computing

 researchcomputing.princeton.edu/support/knowledge-base/slurm

Introducing Slurm

OUTLINE

Introduction

On all of the cluster systems (except Nobel and Tigressdata), users run programs by submitting scripts to the Slurm  job scheduler. A Slurm script must do three things:

1. prescribe the resource requirements for the job
2. set the environment
3. specify the work to be carried out in the form of shell commands

Below is a sample Slurm script for running a Python code using a Conda environment:

```
#!/bin/bash
#SBATCH --job-name=myjob           # create a short name for your job
#SBATCH --nodes=1                 # node count
#SBATCH --ntasks=1                # total number of tasks across all nodes
#SBATCH --cpus-per-task=1         # cpu-cores per task (>1 if multi-threaded tasks)
#SBATCH --mem-per-cpu=2G          # memory per cpu-core (4G is default)
#SBATCH --time=00:01:00           # total run time limit (HH:MM:SS)
#SBATCH --mail-type=begin         # send email when job begins
#SBATCH --mail-type=end           # send email when job ends
#SBATCH --mail-user=<YourNetID>@princeton.edu

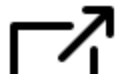
module purge
module load anaconda3/2020.11
conda activate pytools-env

python myscript.py
```

The first line of a Slurm script specifies the Unix shell to be used. This is followed by a series of #SBATCH directives which set the resource requirements and other parameters of the job. The script above requests 1 CPU-core and 4 GB of memory for 1 minute of run time. The necessary changes to the environment are made by loading the anaconda3/<version> environment module and activating a particular Conda environment. Lastly, the work to be done, which is the execution of a Python script, is specified in the final line.

See below for information about the correspondence between tasks and CPU-cores. If your job fails to finish before the specified time limit then it will be killed. You should use an accurate value for the time limit but include an extra 20% for safety.

A job script named `job.slurm` is submitted to the Slurm scheduler with the [sbatch](#)



command:

```
$ sbatch job.slurm
```

The job should be submitted to the scheduler from the login node of a cluster. The scheduler will queue the job where it will remain until it has sufficient [priority](#) to run on a compute node. Depending on the nature of the job and available resources, the queue time will vary between seconds to many days. When the job finishes, the user will receive an email. To check the status of queued and running jobs, use the following command:

```
$ squeue -u <YourNetID>
```

To see the expected start times of your queued jobs:

```
$ squeue -u <YourNetID> --start
```

See Slurm scripts for [Python](#), [R](#), [MATLAB](#), [Julia](#) and [Stata](#).

Useful Slurm Commands

Before Submitting Your Job (i.e. Learning About the System)

Command	Description
checkquota	see your usage of disk space, check that you have enough space in your folders, link at the bottom to ask for more space
cat /etc/os-release	information about operation system
lscpu	information about the CPUs on that particular node
snodes	information about compute nodes
shownodes	information about compute nodes (easier to read)
sinfo	see how nodes are being used
sinfo -p gpu	see how GPUs are being used

Command	Description
qos	"quality of service" - see how jobs are partitioned, and the limits on each partition
top	displays processor activity, commands being run (press 'q' to quit)
htop	displays processor activity, commands being run, with color (press 'q' to quit)
sshare	shows cluster shares assigned by group
sprio -w	see how job priority is designated (shows weights given to each factor)

To Submit a Job

Command	Description
sbatch <name-of-slurm-script>	submits your job to the scheduler
salloc	requests an interactive job on compute node(s) (see below)

After Job is Submitted

Command	Description
squeue	displays all jobs that are running or waiting to run
squeue -u <your-netid>	see your jobs that are running or waiting to run
squeue --start	report the expected start time for pending jobs
squeue -j <jobid>	show the nodes being used for your running job
scontrol show jobid <jobid>	displays detailed info about a job
sprio	show priorities assigned to pending jobs
slurmtop	displays current jobs
scancel	cancel a job (e.g. scancel 2534640)

After Job is Completed

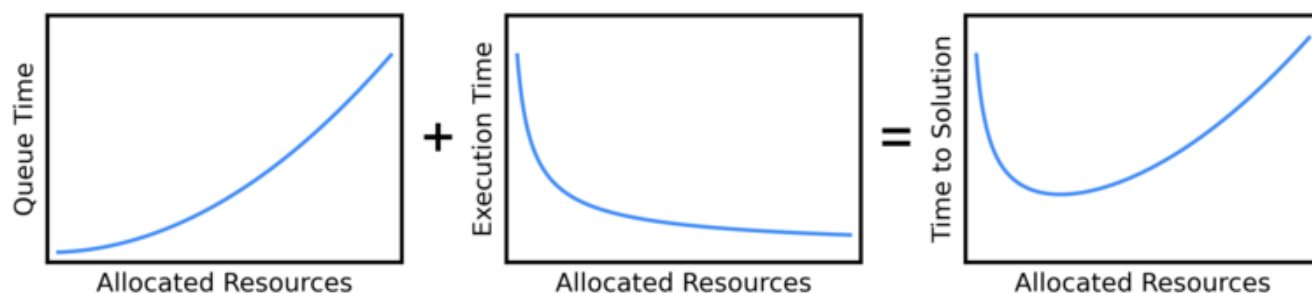
Command	Description
---------	-------------

Command	Description
seff <jobid>	see results of completed job
shistory	displays your job history
jobstats	see accurate metrics of memory, CPU, and GPU from jobs. See more about job stats .

Time to Solution

When a job is submitted to the Slurm scheduler, the job first waits in the queue before being executed on the compute nodes. The amount of time spent in the queue is called the queue time. The amount of time it takes for the job to run on the compute nodes is called the execution time.

The figure below shows that for a parallel code the queue time increases with increasing resources (e.g., CPU-cores) while the execution time decreases with increasing resources. One should try to find the optimal set of resources that minimizes the "time to solution" which is the sum of the queue time and the execution time. The procedure for doing this is called a scaling analysis and it is described in [Choosing the Number of Nodes, CPU-cores and GPUs](#).



Note that the above does not apply to serial codes which can only use a single CPU-core.

Considerations

Some things to think about:

- Make sure your Slurm script loads any dependencies or path changes (i.e., you need python3, so **module load anaconda3/2020.11**). See [Environment Modules](#) for more.
- Make sure you call your executable with its full path or **cd** to the appropriate directory.

- If you call the executable directly and not via an interpreter like `sr`un or `python` or `Rscript`, etc., make sure you have `+x` permissions on it.
- Think about the filesystems. Different ones are useful for different things, have different sizes, and they don't all talk to each other (i.e., `/tmp` is local to a specific node while `/scratch/gpfs/<YourNetID>` is local to each large cluster). See [Data Storage](#) for more.
- `/home` has a default 10 GB quota on most clusters and should be used mostly for code and packages needed to run tasks. You can request an increase up to 10 GB for your `/home` directory if necessary for larger packages (see [checkquota](#)).

Your First Slurm Job

If you are new to the HPC clusters or Slurm then see this [tutorial](#) for running your first job. Read the [Storage page](#) to know where to write the output files of your jobs.

Terminology

When you SSH to a cluster you are connecting to the login node, which is shared by all users. Running jobs on the login node is prohibited. Batch and interactive jobs must be submitted from the login node to the Slurm job scheduler using the `"sbatch"` and `"salloc"` commands. After waiting in the queue, jobs are sent to the compute nodes where the actual computational work is carried out. Each compute node at Princeton has two or more CPUs where each CPU has many CPU-cores. Run the `"snodes"` command and look at the `"CPUS"` column in the output to see the number of CPU-cores per node for a given cluster. You will see values such as 28, 32, 40, 96 and 128. If your job requires the number of CPU-cores per node or less then almost always you should use `--nodes=1` in your Slurm script. The values for `--ntasks` and `--cpus-per-task` are best explained by the examples below. The mathematical product of nodes, ntasks and cpus-per-task is equal to the total number of CPU-cores that you are requesting in your Slurm script. For a serial job these three quantities should be 1. For parallel jobs the product of these quantities will be greater than 1. Note that with the `"CPUS"` column from `"snodes"` and the `"cpus-per-task"` directive, Slurm is using `"CPUS"` or `"cpus"` to mean CPU-cores. The `--time` directive sets the maximum runtime needed for your job. You should set this accurately but include an extra 20% since the job will be killed if it does not finish before the limit is reached. For setting the memory requirements of the job using `--mem-per-cpu` or `--mem` see our [memory page](#). For more definitions of HPC terms see the [glossary](#).

Serial Jobs

Serial jobs use only a single CPU-core. This is in contrast to parallel jobs which use multiple CPU-cores simultaneously. Below is a sample Slurm script for a serial R job:

```
#!/bin/bash
#SBATCH --job-name=slurm-test    # create a short name for your job
#SBATCH --nodes=1               # node count
#SBATCH --ntasks=1              # total number of tasks across all nodes
#SBATCH --cpus-per-task=1        # cpu-cores per task (>1 if multi-threaded tasks)
#SBATCH --mem-per-cpu=4G         # memory per cpu-core (4G is default)
#SBATCH --time=00:01:00         # total run time limit (HH:MM:SS)
#SBATCH --mail-type=begin        # send email when job begins
#SBATCH --mail-type=end          # send email when job ends
#SBATCH --mail-user=<YourNetID>@princeton.edu

module purge
Rscript myscript.R
```

Slurm scripts are more or less shell scripts with some extra parameters to set the resource requirements:

- `--nodes=1` - specify one node
- `--ntasks=1` - claim one task (by default 1 per CPU-core)
- `--time` - claim a time allocation, here 1 minute. Format is DAYS-HOURS:MINUTES:SECONDS

The other settings configure automated emails. You can delete these lines if you prefer not to receive emails.

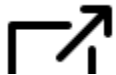
Submit the job to the Slurm job scheduler using the following command:

```
$ sbatch job.slurm
```

In the command above, `job.slurm` is the filename of your Slurm script. Feel free to use a different name such as `submit.sh`.

As a Slurm job runs, unless you redirect output, a file named `slurm-#####.out` will be produced in the directory where the `sbatch` command was ran. You can use `cat`, `less` or any text editor to view it. The file contains the output your program would have written to a terminal if run interactively.

Interactive Allocations with `salloc`

The login node of a cluster can only be used for very light interactive work using up to 10% of the machine (CPU-cores and memory) for up to 10 minutes. This is strictly enforced because violations of this rule can often adversely affect the work of other users. Intensive interactive work must be carried out on the compute nodes using the `salloc`  command. To work interactively on a compute node with 1 CPU-core and 4 GB of memory for 20 minutes, use the following command:

```
$ salloc --nodes=1 --ntasks=1 --mem=4G --time=00:20:00
```

As with batch jobs, interactive allocations go through the queuing system. This means that when the cluster is busy you will have to wait before the allocation is granted. You will see output like the following:

```
salloc: Pending job allocation 32280311
salloc: job 32280311 queued and waiting for resources
salloc: job 32280311 has been allocated resources
salloc: Granted job allocation 32280311
salloc: Waiting for resource configuration
salloc: Nodes della-r4c1n13 are ready for job
[aturing@della-r4c1n13 ~]$
```

After the wait time, you will be placed in a shell on a compute node where you can begin working interactively. Note that your allocation will be terminated when the time limit is reached. You can use the **exit** command to end the session and return to the login node at anytime.

GPUs

To request a node with a GPU:

```
$ salloc --nodes=1 --ntasks=1 --mem=4G --time=00:20:00 --gres=gpu:1
```

Graphics

If you are working with graphics then to enable X11 forwarding (and see [additional requirements](#)):

```
$ salloc --nodes=1 --ntasks=1 --mem=4G --time=00:20:00 --x11
```

Multithreaded Jobs

Some software like the linear algebra routines in NumPy and MATLAB are able to use multiple CPU-cores via libraries that have been written using shared-memory parallel programming models like OpenMP, pthreads or Intel Threading Building Blocks (TBB). OpenMP programs, for instance, run as multiple "threads" on a single node with each thread using one CPU-core.

Below is an appropriate Slurm script for a multithreaded job:


```
#!/bin/bash
#SBATCH --job-name=multithread # create a short name for your job
#SBATCH --nodes=1              # node count
#SBATCH --ntasks=1            # total number of tasks across all nodes
#SBATCH --cpus-per-task=4      # cpu-cores per task (>1 if multi-threaded tasks)
#SBATCH --mem-per-cpu=4G       # memory per cpu-core (4G is default)
#SBATCH --time=00:15:00        # maximum time needed (HH:MM:SS)
#SBATCH --mail-type=begin      # send email when job begins
#SBATCH --mail-type=end        # send email when job ends
#SBATCH --mail-user=<YourNetID>@princeton.edu
```



```
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
```

```
module purge
```

```
module load matlab/R2019a
```

```
matlab -nodisplay -nosplash -r for_loop
```

In the script above, the **cpus-per-task** parameter is used to tell Slurm to run the multithreaded task using four CPU-cores. In general, as cpus-per-task increases, the execution time of the job decreases while the queue time increases and the parallel efficiency decreases. The optimal value of cpus-per-task must be determined empirically by conducting a [scaling analysis](#). Here are some examples of multithreaded codes: [C++](#)  , [Python](#)

 and [MATLAB](#)  . A list of external resources for learning more about OpenMP is [here](#).

For a multithreaded, single-node job make sure that the product of ntasks and cpus-per-task is equal to or less than the number of CPU-cores on a node. Use the "snodes" command and look at the "CPUS" column to see the CPU-cores per node information.

IMPORTANT: Only codes that have been explicitly written to use multiple threads will be able to take advantage of multiple CPU-cores. Using a value of cpus-per-task greater than 1 for a code that has not been parallelized will not improve its performance. Instead, doing so will waste resources and cause your next job submission to have a lower priority.

Multinode or Parallel MPI Jobs

Many scientific codes use of a form of distributed-memory parallelism based on MPI (Message Passing Interface). These codes are able to use multiple CPU-cores on multiple nodes simultaneously. For example, the script below uses 32 CPU-cores on each of 2 nodes:


```
#!/bin/bash
#SBATCH --job-name=multinode      # create a short name for your job
#SBATCH --nodes=2                 # node count
#SBATCH --ntasks-per-node=32     # number of tasks per node
#SBATCH --cpus-per-task=1        # cpu-cores per task (>1 if multi-threaded tasks)
#SBATCH --mem-per-cpu=4G         # memory per cpu-core (4G is default)
#SBATCH --time=00:05:00         # total run time limit (HH:MM:SS)
#SBATCH --mail-type=begin        # send email when job begins
#SBATCH --mail-type=end          # send email when job ends
#SBATCH --mail-user=<YourNetID>@princeton.edu
```

```
module purge
module load intel/19.0/64/19.0.5.281 intel-mpi/intel/2019.5/64
```

```
srun /home/aturing/.local/bin/myprog <args>
```

IMPORTANT: Only codes that have been explicitly written to run in parallel can take advantage of multiple cores on multiple nodes. Using a value of `--ntasks` greater than 1 for a code that has not been parallelized will not improve its performance. Instead you will waste resources and have a lower priority for your next job submission.

IMPORTANT: The optimal value of `--nodes` and `--ntasks` for a parallel code must be determined empirically by conducting a [scaling analysis](#). As these quantities increase, the parallel efficiency tends to decrease and queue times increase. The parallel efficiency is the serial execution time divided by the product of the parallel execution time and the number of tasks. If multiple nodes are used then in most cases one should try to use all of the CPU-cores on each node.

You should try to use all the cores on one node before requesting additional nodes. That is, it is better to use one node and 32 cores than two nodes and 16 cores per node.

MPI jobs are composed of multiple processes running across one or more nodes. The processes coordinate through point-to-point and collective operations. More information about running MPI jobs is in [Compiling and Running MPI Jobs](#).

The Princeton HPC clusters are configured to use **srun** which is the Slurm analog of **mpirun** or **mpiexec**. Do not use **mpirun** or **mpiexec**.

Multinode, Multithreaded Jobs

Many codes combine multithreading with multinode parallelism using a hybrid OpenMP/MPI approach. Below is a Slurm script appropriate for such a code:

```
#!/bin/bash
#SBATCH --job-name=hybrid      # create a short name for your job
#SBATCH --nodes=2             # node count
#SBATCH --ntasks-per-node=8   # total number of tasks per node
#SBATCH --cpus-per-task=4     # cpu-cores per task (>1 if multi-threaded tasks)
#SBATCH --mem-per-cpu=4G      # memory per cpu-core (4G is default)
#SBATCH --time=00:01:00       # total run time limit (HH:MM:SS)
#SBATCH --mail-type=begin     # send email when job begins
#SBATCH --mail-type=end       # send email when job ends
#SBATCH --mail-user=<YourNetID>@princeton.edu
```

```
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
```

```
module purge
module load intel/19.0/64/19.0.5.281 intel-mpi/intel/2019.5/64
```

```
srun /home/aturing/.local/bin/myprog <args>
```

The script above allocates 2 nodes with 32 CPU-cores per node. For OpenMP/MPI codes, the script above would produce 8 MPI processes per node. When an OpenMP parallel directive is encountered, each process would execute the work using 4 CPU-cores. For a simple C++

example of this see [this page](#)  . For more details about the SBATCH options see [this page](#).

As discussed above, the optimal values of nodes, ntasks-per-node and cpus-per-task must be determined empirically by conducting a [scaling analysis](#). Many codes that use the hybrid OpenMP/MPI model will run sufficiently fast on a single node.

Make sure that the product of ntasks and cpus-per-task is equal to or less than the number of CPU-cores on a node. Use the "snodes" command and look at the "CPUS" column to see the CPU-cores per node information.

The Princeton HPC clusters are configured to use **srun** which is the Slurm analog of **mpirun** or **mpiexec**. Do not use mpirun or mpiexec.

GPU Jobs

GPUs are available on Tiger, Adroit and Traverse. On Tiger and Traverse there are four GPUs on each GPU-enabled compute node. To use GPUs in a job add an SBATCH statement with the **--gres** option:

```
#!/bin/bash
#SBATCH --job-name=mnist          # create a short name for your job
#SBATCH --nodes=1                 # node count
#SBATCH --ntasks=1                # total number of tasks across all nodes
#SBATCH --cpus-per-task=1         # cpu-cores per task (>1 if multi-threaded tasks)
#SBATCH --mem-per-cpu=4G          # memory per cpu-core (4G is default)
#SBATCH --gres=gpu:1              # number of gpus per node
#SBATCH --time=00:01:00           # total run time limit (HH:MM:SS)
#SBATCH --mail-type=begin         # send email when job begins
#SBATCH --mail-type=end           # send email when job ends
#SBATCH --mail-user=<YourNetID>@princeton.edu

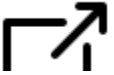
module purge
module load anaconda3/2020.11
conda activate tf2-gpu

python myscript.py
```

To use, for instance, four GPUs per node the appropriate line would be:

```
#SBATCH --gres=gpu:4
```

IMPORTANT: Only codes that have been explicitly written to run on GPUs can take advantage of GPUs. Adding the **--gres** option to a Slurm script for a CPU-only code will not speed-up the execution time but it will waste resources, increase your queue time and lower the priority of your next job submission. Furthermore, some codes are only written to use a single GPU so avoid requesting multiple GPUs unless your code can use them. If the code can use multiple GPUs then you should conduct a [scaling analysis](#) to find the optimal number of GPUs to use.

The [TigerGPU Utilization](#) page gives additional tips on effectively using GPUs. See the [Intro to GPU Programming](#)  workshop for more information about GPUs at Princeton.

Note that several codes require that multiple CPU-cores be used in addition to the GPU for optimal performance.

Job Arrays

Job arrays are used for running the same job a large number of times with only slight differences between the jobs. For instance, let's say that you need to run 100 jobs, each with a different seed value for the random number generator. Or maybe you want to run the same analysis script on data for each of the 50 states in the USA. Job arrays are the best choice for such cases.

Below is an example Slurm script for running Python where there are 5 jobs in the array:

```
#!/bin/bash
#SBATCH --job-name=array-job      # create a short name for your job
#SBATCH --output=slurm-%A.%a.out # stdout file
#SBATCH --error=slurm-%A.%a.err  # stderr file
#SBATCH --nodes=1                # node count
#SBATCH --ntasks=1               # total number of tasks across all nodes
#SBATCH --cpus-per-task=1        # cpu-cores per task (>1 if multi-threaded tasks)
#SBATCH --mem-per-cpu=4G         # memory per cpu-core (4G is default)
#SBATCH --time=00:01:00          # total run time limit (HH:MM:SS)
#SBATCH --array=0-4              # job array with index values 0, 1, 2, 3, 4
#SBATCH --mail-type=all          # send email on job start, end and fault
#SBATCH --mail-user=<YourNetID>@princeton.edu

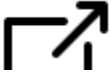
echo "My SLURM_ARRAY_JOB_ID is $SLURM_ARRAY_JOB_ID."
echo "My SLURM_ARRAY_TASK_ID is $SLURM_ARRAY_TASK_ID"
echo "Executing on the machine:" $(hostname)

module purge
module load anaconda3/2020.11
conda activate myenv

python myscript.py
```

The key line in the Slurm script above is:

```
#SBATCH --array=0-4
```

In this example, the Slurm script will run five jobs. Each job will have a different value of `SLURM_ARRAY_TASK_ID` (i.e., 0, 1, 2, 3, 4). The value of `SLURM_ARRAY_TASK_ID` can be used to differentiate the jobs within the array. See a [full example](#)  for Python.

One can either pass `SLURM_ARRAY_TASK_ID` to the executable as a command-line parameter or reference it as an environment variable. Using the latter approach, the first few lines of a Python script (called `myscript.py` above) might look like this:

```
import os
idx = int(os.environ["SLURM_ARRAY_TASK_ID"])
parameters = [2.5, 5.0, 7.5, 10.0, 12.5]
myparam = parameters[idx]
# execute the rest of the script using myparam
```

For an R script you can use:

```
idx <- as.numeric(Sys.getenv("SLURM_ARRAY_TASK_ID"))
parameters <- c(2.5, 5.0, 7.5, 10.0, 12.5)
myparam <- parameters[idx + 1]
# execute the rest of the script using myparam
```

For MATLAB:

```
idx = uint16(str2num(getenv("SLURM_ARRAY_TASK_ID")))
parameters = [2.5, 5.0, 7.5, 10.0, 12.5]
myparam = parameters[idx + 1]
# execute the rest of the script using myparam
```

For Julia:

```
idx = parse{Int64, ENV["SLURM_ARRAY_TASK_ID"]}
parameters = (2.5, 5.0, 7.5, 10.0, 12.5)
myparam = parameters[idx + 1]
# execute the rest of the script using myparam
```

Be sure to use the value of SLURM_ARRAY_TASK_ID to assign unique names to the output files for each job in the array. Failure to do this will result in all jobs using the same filenames.

You can set the array numbers to any arbitrary set of numbers and ranges, for example:

```
#SBATCH --array=0,100,200,300,400,500
#SBATCH --array=0-24,42,56-99
#SBATCH --array=1-1000
```

Note that it is normal to see (QOSMaxJobsPerUserLimit) listed in the NODELIST(REASON) column of **squeue** output for job arrays. It indicates that you can only have a certain number of jobs actively queued. Just wait and all the jobs of the array will run. Use the **qos** command to see the limits. A maximum number of simultaneously running tasks from the job array may be specified using a "%" separator. For example "--array=0-15%4" will limit the number of simultaneously running tasks from this job array to 4.

To see the limit on the number of jobs in an array:

```
# ssh della
$ scontrol show config | grep Array
MaxArraySize      = 2501
```

If you encounter a QOSMaxSubmitJobPerUserLimit error with sbatch when the number of jobs in the array is less than MaxArraySize then it probably means that your run time limit is causing you to land in the test QOS which has a limit of only a few hundred jobs (run the "qos" command and see the MaxSubmit column for the exact value). The solution is to increase your run time limit to 62 minutes which will cause you to land in a production QOS. The value of MaxSubmit may also be smaller than MaxArraySize for other QOS partitions. Ultimately, the number of allowable jobs within an array is set by the minimum of MaxArraySize and MaxSubmit for a given QOS. See the "Job Scheduling" section of a cluster webpage (e.g., [Della](#)) for the relationship between QOS and run time limits.

Each job in the array will have the same values for **nodes**, **ntasks**, **cpus-per-task**, **time** and so on. This means that job arrays can be used to handle everything from serial jobs to large multi-node cases.

See the [Slurm documentation](#)  for more on job arrays.

Running Multiple Jobs in Parallel as a Single Job

In general one should use job arrays for this task but in some cases different executables need to run simultaneously. In the example below all the executables are the same but this is not required. If we have, for example, three jobs and we want to run them in parallel as a single Slurm job, we can use the following script:


```
#!/bin/bash
#SBATCH --job-name=myjob           # create a short name for your job
#SBATCH --nodes=1                  # node count
#SBATCH --ntasks=3                 # total number of tasks across all nodes
#SBATCH --cpus-per-task=1          # cpu-cores per task (>1 if multi-threaded tasks)
#SBATCH --mem-per-cpu=4G           # memory per cpu-core (4G is default)
#SBATCH --time=00:01:00            # total run time limit (HH:MM:SS)
#SBATCH --mail-type=begin          # send email when job begins
#SBATCH --mail-type=end            # send email when job ends
#SBATCH --mail-user=<YourNetID>@princeton.edu

module purge
module load anaconda3/2020.11

srun -N 1 -n 1 --exclusive python demo.py 0 &
srun -N 1 -n 1 --exclusive python demo.py 1 &
srun -N 1 -n 1 --exclusive python demo.py 2 &
wait
```

Since we want to run the jobs in parallel, we place the & character at the end of each srun command so that each job runs in the background. The wait command serves as a barrier keeping the overall job running until all jobs are complete.

Do not use this method if the tasks running within the overall job are expected to have significantly different execution times. Doing so would result in idle processors until the longest task finished. Job arrays should always be used in place of this method when possible.

If you want to run a large number of short jobs while reusing the same Slurm allocation then see [this example](#)  .

For GPU jobs, make sure your script uses these elements:

```
#SBATCH --nodes=1
#SBATCH --ntasks=2
#SBATCH --gres=gpu:2
...
srun -N 1 -n 1 --gres=gpu:1 --exclusive python demo.py 0 &
srun -N 1 -n 1 --gres=gpu:1 --exclusive python demo.py 1 &
wait
```

For more see "MULTIPLE PROGRAM CONFIGURATION" on [this page](#)



Large Memory Jobs

One advantage of using the HPC clusters over your laptop or workstation is the large amount of RAM available per node. On some clusters you can run a job with 100's of GB of memory, for example. This can be very useful for working with a large data set. To find out how much memory each node has, run the **snodes** command and look at the MEMORY column which is in megabytes.

```
#!/bin/bash
#SBATCH --job-name=slurm-test      # create a short name for your job
#SBATCH --nodes=1                  # node count
#SBATCH --ntasks=1                 # total number of tasks across all nodes
#SBATCH --cpus-per-task=1          # cpu-cores per task (>1 if multi-threaded tasks)
#SBATCH --mem=100G                 # memory per node (4G per cpu-core is default)
#SBATCH --time=00:01:00           # total run time limit (HH:MM:SS)
#SBATCH --mail-type=begin          # send email when job begins
#SBATCH --mail-type=end            # send email when job ends
#SBATCH --mail-user=<YourNetID>@princeton.edu

module purge
module load anaconda3/2020.11
conda activate micro-env

python myscript.py
```

The example above runs a Python script using 1 CPU-core and 100 GB of memory. In all Slurm scripts you should use an accurate value for the required memory but include an extra 20% for safety. For more see [Allocating Memory](#).

Running Jobs in a Sequence (Job Dependencies)

Here we explain how to automatically run a sequence of jobs using Slurm job dependencies. This is useful for jobs that need to run for much longer than the time limit. Or when a second job can only be started after the first job is finished because of a dependency between the two.

To use Slurm job dependencies for running a long job in steps your application must have a way of writing a checkpoint file and it must be able to figure out which checkpoint file to read at the start of each job step. If your application does not provide these two requirements then one can typically write scripts to deal with it.

The "singleton" job dependency works by queueing subsequent jobs with the same job-name until the previous job with that name finishes. Consider the Slurm script (job.slurm) below and note the --job-name and --dependency directives:


```
#!/bin/bash
#SBATCH --job-name=LongJob      # create a short name for your job
#SBATCH --nodes=1               # node count
#SBATCH --ntasks=1              # total number of tasks across all nodes
#SBATCH --cpus-per-task=1       # cpu-cores per task (>1 if multi-threaded tasks)
#SBATCH --mem=4G                # memory per node (4G per cpu-core is default)
#SBATCH --time=00:01:00         # total run time limit (HH:MM:SS)
#SBATCH --dependency=singleton  # job dependency
#SBATCH --mail-type=begin       # send email when job begins
#SBATCH --mail-type=end         # send email when job ends
#SBATCH --mail-user=<YourNetID>@princeton.edu
```

```
module purge
module load anaconda3/2020.11
conda activate galaxy-env
```

```
python myscript.py
```

To run the code in a sequence of five successive steps:

```
$ sbatch job.slurm  # step 1
$ sbatch job.slurm  # step 2
$ sbatch job.slurm  # step 3
$ sbatch job.slurm  # step 4
$ sbatch job.slurm  # step 5
```

The first job step can run immediately. However, step 2 cannot start until step 1 has finished and so on. That is, each step after the first will wait for the one before it since each is waiting for "LongJob" to finish. Read more about job dependencies on the [Slurm](#)  website. Note that job dependencies are discouraged on Stellar.

Efficiency Reports

If you include the appropriate SBATCH mail directives in your Slurm script then you will receive an email after each job finishes. Below is a sample report:

Job ID: 670018
Cluster: adroit
User/Group: aturing/math
State: COMPLETED (exit code 0)
Cores: 1
CPU Utilized: 05:17:21
CPU Efficiency: 92.73% of 05:42:14 core-walltime
Job Wall-clock time: 05:42:14
Memory Utilized: 2.50 GB
Memory Efficiency: 62.5% of 4.00 GB

The report provides information about runtime, CPU usage, memory usage and so on. You should inspect these values to determine if you are using the resources properly. Your queue time is in part determined by the amount of resources your are requesting. Your fairshare value, which in part determines the priority of your next job, is decreased in proportion to the amount of resources you have used over the previous 30 days.

Why Won't My Job Run?

The start time of a job is determined by job priority.

More Slurm Resources

See a comprehensive list of [external resources for learning Slurm](#).