OpenMind Tutorial (III): Slurm Job Scheduler and Best Practices

Shaohao Chen
Brian and Cognitive Sciences (BCS)
Massachusetts Institute of Technology (MIT)
Outline

- **Using Slurm:**
  - Recap: interactive session, batch job, check job info.
  - Resources and quota: partition, QOS and priority.
  - Using CPU and memory: multiple cores, hyperthreading.
  - Using GPU
  - Using Multiple nodes
  - Running many programs in parallel
  - More on job and cluster information.
SLURM Job Scheduler

Why using job scheduler?

• There is only one head node (openmind7) for all users. **It is not allowed to run time-consuming programs on the head node!**
• There are around 90 compute nodes. **All time-consuming programs should run on compute nodes!**
• To run programs on compute nodes, submit batch jobs using the Slurm job scheduler.
• **Job scheduler:** submit jobs in a batch mode, assign appropriate compute resources to jobs, queue jobs in priority order, monitor job status, and save useful job information.

SLURM (Simple Linux Utility for Resource Management)

• An open source, fault-tolerant, and highly scalable cluster management and job scheduling system for Linux clusters.
• Widely used in many computer clusters in the world.
Recap: Interactive session

- **Interactive session**: work interactively with requested resources (e.g. CPU, GPU, memory, walltime).
- **Purpose**: Run your program using the requested resources and get the standard output right away.
- **Typical cases**: Run testing programs; Debug codes; Use GUI; Visualizing data.

- **Examples**:

  ```bash
  srun -n 1 -t 02:00:00 --pty bash  # Request 1 CPU core for 2 hours (default 10 minutes)
  srun -n 1 -t 03:00:00 --x11 --pty bash  # Support GUI (login using ssh -Y)
  srun -n 1 -t 01:30:00 --mem=20G --pty bash  # Request 1 CPU core and 20 GB RAM (default 2GB/core)
  srun -n 1 --gres=gpu:1 --constraint=any-gpu --pty bash  # Request 1 CPU core and 1 GPU
  
  hostname  # Check the host name of the node
  ```
Recap: Submit a batch job

- **Submit jobs to background**: continue running no matter your terminal is connected to the cluster or not.
- An example script to run Python or MATLAB programs (e.g. named main.py or main.m):

```bash
#!/bin/bash
#SBATCH -t 01:00:00  # walltime = 1 hour
#SBATCH -N 1        # one node
#SBATCH -n 1        # one CPU core
#SBATCH --mem=5G    # 5 GB memory
module load openmind/anaconda/3-2019.10  # Load an Anaconda module
python main.py     # Run a Python program
module load mit/matlab/2020a    # Load a MATLAB module
matlab -nodisplay -r "maxNumCompThreads(1), main, exit"  # Run a MATLAB program with 1 thread
```

- **Submit the job script**:

```
sbatch example.sh
```
Recap: Check and cancel jobs

• Check job status for a user:
  ```bash
  squeue -u $USER
  ```

• Check information of a job:
  ```bash
  sacct -j <JobID>
  ```

• Cancel a job:
  ```bash
  scancel <JobID>
  ```

• Output files:
By default, both standard output and standard error are directed to the same file. The default file name is "slurm-%j.out", where the "%j" is replaced by the job ID. Use -o option to specify an output file name. Use -e option to direct the standard error to a specified file.
Resources and Quota

• **Public resources:** purchased by the department.
  Nodes 031, 033-036, 041, 043, 053, 083-084, 089-090, 100, 103, 107, and 109-115

• **Buy-in resources:** “semi-private”. Everyone can use. The owner lab has the highest priority.
  Nodes 55-116 except the public nodes

• **Lab-priority quota:**
  **Basic quota:** 80 virtual CPU cores, 4 GPUs, and 600GB RAM.
  Multiple of compute-only quota.
  Lab buy-in resources.
Partition

- **Partitions** group nodes into logical sets. Possibly overlapping.

- Check partition information:
  
  ```bash
  sinfo           # Check state, node list, etc.
  sinfo -N -p normal    # List each node in one line.
  ```

- **State:** idle, alloc, mix, drain.

- **Partitions on the cluster:**
  
  1. **use-everything:** All nodes.
  2. **normal:** All nodes.
  3. **labname:** shared nodes + lab buy-in nodes (if applied)
Quality of Service (QOS)

- QOS affects the following: job scheduling priority, job preemption, and job limits.
- Partition QOS: A QOS is attached to a partition.
- QOS on the cluster:
  1. *use-everything*: Lowest priority. No limit on total resources.
  2. *normal*: Medium priority. Total resources per user is limited to around 10% of all resources on the cluster.
  3. *labname*: Highest priority. Total resources is limited to the lab-priority quota.

- Check QOS info:
  
  ```
  sacctmgr show qos --parsable
  sacctmgr show qos format=name,priority,GraceTime,Preempt,PreemptMode,UsageFactor,GrpTRES
  ```
SLURM partitions

Resources = (CPUs, RAM, GPUs)
Public resources: green box
Lab purchased resources: yellow box

Partition = resources available for a type of jobs
Lab-name: blue-line box, lab subscribed + lab purchased
Normal: red-line box, per-user limit = 10% of all resources
Use-everything: pink-line box, per-user limit = all resources

Priority order: lab-name > normal > use-everything
(Thicker border means higher priority in the figure.)
## Job Limits

<table>
<thead>
<tr>
<th>Partition QOS Name</th>
<th>Max submitting jobs per user</th>
<th>Max running jobs per user</th>
<th>Max resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>labname</td>
<td>1000</td>
<td>up to using up max resources</td>
<td>all of the lab's priority resources</td>
</tr>
<tr>
<td>normal</td>
<td>2000</td>
<td>700 or up to using up max resources</td>
<td>700 virtual CPU cores, 30 GPUs, and 5 TB memory per user [3]</td>
</tr>
<tr>
<td>use-everything</td>
<td>7000</td>
<td>up to using up max resources</td>
<td>all resources on the cluster</td>
</tr>
</tbody>
</table>

- **Time limits**: 7 days for all partitions.
Submit jobs with Partition QOS

• The default partition QOS (without -p option) is *normal*. Use the default when it is unnecessary to start your job immediately or unnecessary to use your lab priority resources.

```bash
srun -n 1 --pty bash
```

• When necessary, request your lab-priority resources:

```bash
srun -p <lab> -n 1 --pty bash  # Provide your lab name.
srun --partition=<lab> -n 1 --pty bash  # The same as above.
```

• To submit many low-priority jobs, use *use-everything*

```bash
sbatch -p use-everything --array=1-1000 ./job.sh
```
Preemption

*Preempt:* terminate a running job, free-up the running job’s resources and allocate them to a queued job.

*Labname* jobs preempt *normal* and *use-everything* jobs.

*Normal* jobs preempt *use-everything* jobs.

*An example:* If a user submits job A using `-p labname` and the job is scheduled on the resources where job B with `-p normal` or `-p use-everything` is running, job B will be terminated and requeued, and job A will start to run immediately.

*Preemption will not occur unless the job can’t be scheduled elsewhere.*

*Usually, preemption will occur when the cluster is busy or the requested resources (e.g. a certain type of GPUs) are used up.*

*Checkpoint:* When possible, set checkpoints to save data and resume the computation from where it stopped. The checkpoint feature is available in many applications, otherwise use DMTCP.

*Preempted jobs will be requeued.*

*Preemption will occur immediately once the conditions are satisfied.*
Job Priority

• A job with a higher priority will start to run sooner than other jobs in the queue (except backfill cases).
• Multifactor priority:

\[
\text{Job\_priority} = (\text{PriorityWeightAge}) \times (\text{age\_factor}) + \\
(\text{PriorityWeightFairshare}) \times (\text{fair\_share\_factor}) + \\
(\text{PriorityWeightTRES}) \times (\text{tres\_factor}) + \\
(\text{PriorityWeightPartition}) \times (\text{partition\_factor}) + \\
(\text{PriorityWeightQOS}) \times (\text{QOS\_factor}) + \ldots
\]

• Weights are customized as following:
  - **Weight QOS** (The most dominant. To guarantee lab priority on a certain amount of resources.)
  - **Weight TRES** (Secondly dominant. Favor GPU jobs.)
  - **Weight Fair Share** (Thirdly dominant. Prevent a few users from using most of the cluster.)
  - **Weight Age** (Fourthly dominant. Favor older jobs.)
Priority Factors

- The factors are renormalized to the range $0 - 1$.
- QOS factor: $labname > normal > use-everything$
- TRES factor: GPU jobs > CPU-only jobs
- Age factor: The longer the job waits in the queue, the greater the factor is.
- Fare-share factor: The more resources (e.g. CPU hours) a user has consumed, the less this factor is.
  - Usage Reset Period = MONTHLY. Jobs within the past month affect this factor.
  - Decay Half Life = 7 days. Jobs in the past 7 days affect the most.

- Check job priority:
  ```bash
  squeue -a --start --sort=-p  # List all pending jobs in descending-priority order
  squeue -a --sort=-p -o %A,%u,%P,%Q,%p   # Print job priority for all current jobs
  ```
SLURM partitions

Partition = resources available for a type of jobs
Lab-name: lab subscribed + lab purchased
Normal: red-line boxes, per-user limit = 10% of all resources
Use-everything: per-user limit = all resources

Preemption: kick out running jobs
Priority order: lab-name > normal > use-everything
Preemption happens when jobs are assigned to the same resources at the same time (i.e. overlaps of boxes).

Resources = (CPUs, RAM, GPUs)
Public: the big green box
Lab subscribed (= part of public): blue-line boxes.
Lab purchased: yellow boxes.
More on Partition QOS

• For any user: Jobs with -p normal or -p use-everything can run on any resources. When possible, set checkpoints for normal or use-everything jobs to protect the lost caused by preemptions.

• For non-buy-in labs: A job with -p labname can run on any public resources.

• For buy-in labs: A job with -p labname can run on any public resources or the lab’s buy-in resources. A job with -p labname --gres=gpu:type:x will likely run on the lab’s buy-in resources.

• Request a specific node if necessary:

  srun -p labname -w nodeXXX -n 1 --pty bash  # Provide the lab name and host name (nodeXXX).

• If lab-priority quota is used up and users in this lab continue to submit jobs using -p labname, the jobs will be pending. Suggest to use -p normal instead.

• An example: If the lab-priority GPUs are not used up but the lab-priority CPUs or memory are used up, the new submitting job will be pending too.

• The use-everything partition QOS is for the purpose of utilizing all resources on the cluster. It is useful when the cluster is not busy.
Using CPU and memory (1)

CPU concerns:

- **Number of requested CPUs** (with `-n` or `-c` options) vs. **number of used CPUs** (implicitly or explicitly specified in user’s program).

- If your program tries to use **more CPUs than requested**, it will only run on the number of requested CPUs, because it is limited by the *cgroups* (a Linux kernel feature) attached to Slurm.

- If your program uses **less CPUs than requested**, the resources are wasted and your fair-share factor is decreased unnecessarily.

Memory concerns:

- If your program costs **more memory than requested**, your job will be automatically killed by the job scheduler.

- If your program costs **too less memory than requested**, the resource is wasted.

Best practice: run your program using the requested number of CPUs and request a memory size that is reasonably larger than the actual-used.
Using CPU and memory (2)

• Check requested CPUs and memory of a running/pending job:
  
  ```
  squeue -j JobID -o "%A %u %N %D %C %m"
  ```

• Check CPU and memory usage of a running job:
  
  ```
  sstat -j JobID -o ntasks,avecpu,maxrss,averss
  ```

• Check CPU and memory usage of a complete job:
  
  ```
  sacct -j JobID -o alloc_cpus,elapsed,cputime,totalcpu,systemcpu,usercpu,reqmem,maxrss,averss
  ```

• Check CPU and memory efficiency of a running or complete job:
  
  ```
  seff JobID
  ```
Using CPU and memory (3)

• How to set the number of CPUs in my application?
  In many applications, users can explicitly specify the number. In some cases, it is implicit or uncontrolled by users.

• How to control the memory usage in my application?
  Modify the size of the problem. Optimize source codes especially for big arrays.

• Submit a batch job using multiple CPU cores and appropriate memory (a MATLAB example):

```bash
#!/bin/bash
# Bash shell
#SBATCH --N 1  # one node
#SBATCH --c 8 # 8 virtual CPU cores. -n is equivalent to --cpus-per-task. Saved in SLURM_CPUS_PER_TASK
#SBATCH --mem=6G # 6 GB memory. Should be reasonably larger than the maximum used memory.
module load mit/matlab/2019a
# Load MATLAB module
# Run the program using requested number of CPU cores (=8 here).
matlab -nodisplay -r "maxNumCompThreads(8), my_program, exit"
```
Using CPU and memory (4)

• Submit a batch job using multiple CPU cores and appropriate memory (a Python example):

```bash
#!/bin/bash  # Bash shell
#SBATCH -N 1  # one node
#SBATCH -c 8  # 8 virtual CPU cores. -n is equivalent to --cpus-per-task. Saved in SLURM_CPUS_PER_TASK
#SBATCH --mem=5G  # 5 GB memory. Should be reasonably larger than the maximum used memory.
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK  # Set number of threads for the application (=8 here).
python my_multithread_program.py  # Run a multithread Python program.
```
Hyperthreading

- On OpenMind, there are two hyperthreaded (virtual) cores per physical CPU core, sharing the same level of cache.
- By default, if using `-n x` or `-c x` without specifying the other, the number of assigned virtual cores will be x when x is even, or x+1 when x is odd. This ensures that whole physical cores are assigned.
- **Two different jobs will not share a physical core.**
- If using `-n x` and `-c y` together, x*y or x*y+1 hyperthreaded cores will be assigned.
- There are 40, 48 or 80 virtual cores on one node on Openmind.
  
  **An example:** Use `-N 1 -c 80` to request a whole node with 80 virtual cores.
- Hyperthreading can be beneficial for many programs, but it can also degrade performance in some cases.
- Run test cases to determine whether hyperthreading is worthy for your program or not.
- **To use hyperthread:** set the number of virtual cores using `-c`, then the hyperthreads are automatically used by the applications in many cases, or set an equal number of threads in the application.
- **Not to use hyperthread:** set `--ntasks-per-core=1` to turn off hyperthreading, then set the number of physical cores using `-n`, and set an equal number of threads/tasks in the application.
Using GPU (1)

• Request for an interactive session with GPU:

```bash
srun -n 1 --gres=gpu:1 --constraint=any-gpu --pty bash  # one GPU
```

• A batch job script for using GPU:

```bash
#!/bin/bash # Bash shell
#SBATCH -n 1 # one physical CPU core (two virtual cores)
#SBATCH --gres=gpu:1 # Any GPU
python my_gpu_program.py  # Run a program using GPU (e.g. a Python program).
```

• Other useful options:

```bash
srun -n 1 --gres=gpu:A100:1 --pty bash  # one A100 GPU.
srun -n 1 --gres=gpu:1 --constraint=any-A100 --pty bash  # one A100 GPU.
srun -n 1 --gres=gpu:1 --constraint=high-capacity --pty bash  # Any high-capacity GPU
srun -n 1 --gres=gpu:1 --constraint=11GB --pty bash  # Any GPU with at least 11 GB memory.
srun -n 1 --gres=gpu:1 --constraint=vgl --pty bash  # GPU with VirtualGL support.
```
Using GPU (2)

• Show available GPU types, numbers and constraints:
  
  `sinfo -N -o "%f, %G"`

• More details on GPUs: [https://github.mit.edu/MGHPCC/OpenMind/wiki/How-to-submit-GPU-jobs%3F](https://github.mit.edu/MGHPCC/OpenMind/wiki/How-to-submit-GPU-jobs%3F)

• Using multiple (same-type) GPUs within one node.
  
  `srun -N 1 -c 4 --gres=gpu:A100:2 --pty bash`  # Four virtual cores and two GPUs.

• A batch job script for using multiple GPUs on one node:

  ```bash
  #!/bin/bash  # Bash shell
  #SBATCH --N 1  # one node
  #SBATCH -c 4  # 4 virtual CPU cores
  #SBATCH --gres=gpu:A100:2  # 2 A100 GPUs
  python my_gpu_program.py  # Run a program using multiple GPUs
  ```
Multiple Nodes (1)

• The performance of many parallel programs can be scaled up with multiple nodes.
• A batch job script for using multiple nodes:

```bash
#!/bin/bash  # Bash shell
#SBATCH    # Two nodes
#SBATCH --ntasks-per-core=1  # Turn off hyperthreads.
#SBATCH --ntasks-per-node=24  # 24 CPU cores per node. Request a whole node instead of part of a node.
srun hostname  # Print the host name on all requested cores
module load openmpi/gcc/64/1.8.1  # Load an MPI implementation.
mpirun -np $SLURM_NTASKS ./my_mpi_program.exe  # Run an MPI program using all requested cores.
# mpirun -np $SLURM_NTASKS python my_mpi_program.py  # Or, run a python program with MPI.

• Ex 1: MPI programs.
• Ex 2: Parallel I/O.
## Multiple Nodes (2)

- Using hyperthreading technique on each node of multiple nodes:

```bash
#!/bin/bash  # Bash shell
#SBATCH -N 2  # Two nodes
#SBATCH --ntasks-per-node=1  # One task per node
#SBATCH -c 48  # 48 virtual cores per task.
srun hostname  # Print the host name on all requested cores
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK  # Set number of threads per task (48 here).
module load openmpi/gcc/64/1.8.1  # Load an MPI implementation.
mpirun -npernode 1 ./my_hybrid_program.exe  # Run an hybrid MPI-OpenMP program: 1 task per node
```
Typical Cases of Requesting CPU and GPU

- 2 virtual CPU cores: -n 1, or -c 2
- 8 physical CPU cores on one node: -N 1 -n 8 --ntasks-per-core=1
- 16 virtual (hyperthreading) CPU cores on one node: -N 1 -c 16
- 1 24-physical-core node: -N 1 -n 24 --ntasks-per-core=1
- 1 48-virtual-core node: -N 1 -c 48
- 2 24-physical-core node: -N 2 --ntasks-per-node=24 --ntasks-per-core=1
- 2 48-virtual-core nodes: -N 2 -c 48
- 1 physical core and 1 default GPU: -n 1 --gres=gpu:1
- 1 physical core and 1 GPU of any type: -n 1 --gres=gpu:1 --constraint=any-gpu
- 1 physical core and 1 GPU with at least 11 GB memory: -n 1 --gres=gpu:1 --constraint=11GB
- 1 physical core and 1 specific type of GPU: -n 1 --gres=gpu:type_name:1
- 1 physical core and 1 GPU with VirtualGL support: -n 1 --gres=gpu:1 --constraint=vgl
- 2 physical cores and 2 GPUs on one node: -N 1 -c 4 --gres=gpu:type_name:2
Run Many Programs in Parallel

- It is often necessary to run the same program for many times with different input parameters.
- To submit batch jobs running many programs simultaneously:
  1. **Parallel submissions:** Execute the `sbatch` command in a loop to submit many individual batch jobs.
  2. **Job Array:** Execute the `sbatch` command only once to submit one job array with many jobs.
  3. **Parallel executions:** Execute the `sbatch` command only once, running the program many times in the job script.
- Take a python program (e.g. named `arange.py`) as a demonstrating example.

```python
import sys  # Import system package
import numpy as np  # Import numpy package
n = int(sys.argv[1])  # Use the first input argument from command line
a=np.arange(n)  # Create a numpy array sized n
print(a)  # Print the result
```
Parallel Submissions

• A batch job script (e.g. named `job.sh`) to run the program:

```bash
#!/bin/bash
# Bash shell
#SBATCH -n 1 # 1 physical CPU core (two virtual cores)
python arange.py $1 # Execute the Python program using the first input argument.
```

• A bash script (e.g. named `loop_jobs.sh`) to execute the the `sbatch` command in a loop.

```bash
#!/bin/bash
# Bash shell
for i in `seq 1 10` # Loop from 1 to 10
do
    sbatch job.sh $i # Submit the batch job using the loop index as input.
done

chmod +x loop_jobs.sh # Change the script to be executable
./loop_jobs.sh # Execute the bash script
```
Job Array

- A batch job script (e.g. named `array_job.sh`) to submit a job array:

```bash
#!/bin/bash
# Bash shell
#SBATCH -n 1 # 1 physical CPU core (2 virtual cores) per job
#SBATCH --array=1-10 # Array task ID from 1 to 10
python arange.py $SLURM_ARRAY_TASK_ID # Submit the batch job using the array task ID as input.
```

- Use `SLURM_ARRAY_TASK_ID` to control different input parameters/files for different jobs.
- Submit the job array:

```bash
sbatch array_job.sh # Submit the job array
```

- The default maximum number of jobs in one job array is 3000 (The maximum task ID is 3000).
- Submit multiple job arrays if needed (e.g. to submit more than 3000 jobs).
- Set a subset of jobs are running at the same time, for example, `--array=1-100:10`. 
Parallel Executions

• A batch job script to run the program many times.

```bash
#!/bin/bash # Bash shell
#SBATCH -N 1 # 1 node
#SBATCH --ntasks-per-core=1 # 1 task per CPU core: turn off hyperthreading.
#SBATCH -n 10 # 10 tasks (i.e. 10 physical CPU cores)
for i in `seq 1 $SLURM_NTASKS` # Loop from 1 to number of tasks (=10).
do
    python arange.py $i & # Execute the python program in background, using the loop index as input
done
wait # Wait for all programs to be completed, then exit the batch job.
```

• The `&` mark is to put the program to the background, then all of the 10 programs will run simultaneously.
• The `wait` command at the end is to ensure that the batch job will not be terminated until all background programs are completed.
Combine Job Array and Serial Executions

- Sometimes it is not worthy to put too many jobs in the queue, for example:
  - When the number of programs is large (e.g. >100) and each program runs for a short time (e.g. <5 mins), it is not worthy to run only one program in one batch job, because the scheduling and queuing time is probably longer than the run time.
  - When the cluster is too busy, it takes a long time for a queuing job to start.
- **A solution:** combine job array and serial executions (see an example below).

- Submit one job array with 10 batch jobs
  ```bash
  #!/bin/bash  # Bash shell
  #SBATCH -n 1  # Request 1 CPU core
  #SBATCH --array=0-9  # Job task ID from 0 to 9
  nmax=$SLURM_ARRAY_TASK_COUNT  # Num of tasks
  j=$SLURM_ARRAY_TASK_ID  # Task ID
  ./run_serial.sh $j $nmax  # Execute a bash script
  ```

- Run 10 programs in each batch job
  ```bash
  # This is the bash script named run_serial.sh
  for i in `seq 1 10`  # Loop from 1 to 10
  do
    index=$((1*$2+$i))  # Global index
    python arange.py $index  # Use global index as input
  done
  ```
Best practice for running many programs

- **Parallelly submitting** many jobs (i.e. executing the `sbatch` command many times in a loop) costs much scheduling time. It is still fine if the number of jobs is small (e.g. n<50), but it is not recommended if the number of jobs is large.

- Scheduling **job array** is very efficient in Slurm. It is recommended to submit a number (especially a large number) of jobs using job array when possible.

- **Parallel execution** is useful if all programs run on one node. It can reduce the number of jobs in the queue.

- **Combination of job array and serial executions** is useful to run many short-time programs.
Job Dependency

Job dependency clauses:
• The **after** clause is satisfied after all tasks in the job array start.
• The **afterany** clause is satisfied after all tasks in the job array complete.
• The **aftercorr** clause is satisfied after the corresponding task ID in the specified job has completed successfully (ran to completion with an exit code of zero).
• The **afterok** clause is satisfied after all tasks in the job array complete successfully.
• The **afternotok** clause is satisfied after all tasks in the job array complete with at least one tasks not completing successfully.

An example:
• Job 2 depends on the success of all jobs in job array 1, and job array 3 depends on the success of job 2.

```bash
jid1=`sbatch --array=1-10 job1.sh | awk '{ print $4 }'`  # Submit job 1 and get the job ID.
jid2=`sbatch --dependency=afterok:$jid1 job2.sh | awk '{ print $4 }'` # Job 2 will start after job array1 succeeds.
sbatch --dependency=aftercorr:$jid2 job3.sh  # Job 3 will start after job 2 succeeds.
```
# Quotes in Bash

- `bash-4.2$ cmd="cat /etc/hostname"`  
  # Double quotes: a string (a command here).

- `bash-4.2$ echo $cmd`  
  # Print the value of a variable  
  `cat /etc/hostname`

- `bash-4.2$ host=`cat /etc/hostname``  
  # Back quotes: the standard output of a command.

- `bash-4.2$ echo $host`  
  # Print the value of a variable  
  `openmind7`

- `bash-4.2$ echo "The hostname is $host."`  
  # Double quotes: the $host is treated as the value of a variable.  
  The hostname is `openmind7`.

- `bash-4.2$ echo 'The hostname is $host.'`  
  # Single quotes: the $host is treated as characters.  
  The hostname is `$host`.

- `bash-4.2$ echo "The hostname is '$host'."`  
  # Single quotes in double quotes.  
  The hostname is `openmind7`.

- `bash-4.2$ echo "The hostname is `$cmd`."`  
  # Back quotes in double quotes.  
  The hostname is `openmind7`.  

- `bash-4.2$ echo "The hostname is `cmd`."`  
  # Back quotes in double quotes.
Job and cluster Information (1)

• For running/pending jobs:

```bash
squeue -a  # Info for current jobs of all partitions
squeue -u $USER -o "%A %a %u %t %N %D %C %m %b"  # Current requested resources by a user
squeue -p normal -o "%A %a %u %t %N %D %C %m %b"  # Current requested resources by a QOS
/cm/shared/admin/bin/slurm-qos-usage <lab_name>  # A script to check current usage of lab-priority resources.
```

• For pending jobs:

```bash
squeue -u $USER --start  # Check reason for pending and expected starting time.
```

• For complete jobs:

```bash
sacct -j JobID --format ALL  # Check all information of a job.
sacct -S 2019-08-01 -E 2019-08-15 -u $USER  # Check all jobs a user submitted in a time period
```
Job and cluster Information (2)

• Node information:

  ```
  sinfo -a  # Information of all partitions
  scontrol show nodes  # Information of all nodes
  /cm/shared/admin/bin/free-nodes  # OM script: Current available CPUs, memory and GPUs
  ```

• Check usage by a lab or a user:

  ```
  sreport cluster AccountUtilizationByUser accounts=<labname> --tres="cpu,gres/gpu" Start=01/01/23 End=02/01/23 tree -t hourper  # CPU and GPU usage by users in a group in a time period
  ```

• Check job efficiency for users in a lab:

  ```
  /cm/shared/admin/bin/slurm-group-eff <labname> 2020-08-01 2020-09-01
  ```
Further Help

• OpenMind documentations on MIT Github:  https://github.mit.edu/MGHPCC/OpenMind/wiki
• Get started:  https://github.mit.edu/MGHPCC/OpenMind/wiki/Getting-started
• FAQ:  https://github.mit.edu/MGHPCC/OpenMind/wiki/FAQ-and-Best-Practices
• OpenMind website (including tutorial slides and recordings):  https://openmind.mit.edu/
• Submit an issue:  https://github.mit.edu/MGHPCC/openmind/issues
  “Not watching”: Be notified when participating or @mentioned.
  “Watching”: Be notified of all conversations. To avoid a deluge of emails, set up email filtering.
  If you know the answer or have seen the problem before, consider replying it. We need communication between users to keep it functioning optimally.
• Slack channel for OM users:  openmind-46.slack.com
• User meetings: Walk-in for quick questions. Make an appointment for discussions.
• Group-rep meetings: Monthly. Communicate with your group reps.
• Email:  openmind-help@mit.edu  .  Office:  46-4115D.