Current version of the intro to the DCC slides.

# Introduction to SLURM

Most DCC partitions are lab-owned machines. These can only be used by members of the group. Submitting to a group partition gives “high-priority”.

Submit to partitions with

```
sbatch –p (partition name) --account=(account name)
```

(in a job script) or

```
srun –p (partition name) --account=(account name) --pty bash –i
```

(interactively) In general, the partition name and account name will be the same for most lab-owned machines.

## Partitions

SLURM partitions are separate queues that divide up a cluster's nodes based on specific attributes. Each partition has its own constraints, which control which jobs can run in it. There are many DCC Partitions, if a partition is not specified, the default partition is the common partition.

## Running an interactive job

Reserve a compute node by typing `srun –pty bash -i`

```
tm103@dcc-login-02 ~ $ srun --pty bash -i
srun: job 186535 queued and waiting for resources
srun: job 186535 has been allocated resources
tm103@dcc-core-11 ~ $
tm103@dcc-core-11 ~ $ squeue -u tm103
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
186535 common bash tm103 R 0:14 1 dcc-core-11
```

I now have an interactive session in the common partition on node dcc-core-11

## SLURM commands

`sbatch` – Submit a batch job

`#SBATCH` – Specify job parameters in a job script

`squeue` – Show lists of jobs

`scancel` – Delete one or more batch jobs

`sinfo` – Show info about machines

`scontrol` – Show cluster configuration information

Use `sbatch` (all lower case) to submit text file job scripts, e.g. test.sh

```
sbatch test.sh
```

Use `#SBATCH` (upper case) in your scripts for scheduler directives, e.g.

```
#SBATCH --mem=1G
#SBATCH --output=matlab.out
```

All SLURM directives can be given on the command line instead of the script. slurm.schedmd.com

## Slurm memory directives

The default memory request (allocation) is 2 GB RAM. This is a hard limit, always request a little more. To request a total amount of memory for the job, use one of the following: * `--mem=<MB>` additional memory * `--mem=<Gigabyte>G` the amount of memory required per node, or * `--mem-per-cpu=<MB>` the amount of memory per CPU core, for multi-threaded jobs

**Note:** –mem and –mem-per-cpu are mutually exclusive

## Slurm parallel directives

All parallel directives have defaults of 1

`-N <number>` How many nodes (machines)

`-n <number>` or `--ntasks=<number>` How many parallel jobs (“tasks”)

`-c <number>` or `--cpus-per-task=<number>`

Use `-n` and `-N` for multi-node jobs (e.g. MPI)

Use `-c` (–cpus-per-task) for multi-threaded jobs

## Job script examples

```
#!/bin/bash
#SBATCH ---output=test.out
hostname # print hostname
```

This prints the name of the compute node in the file “test.out”

```
tm103@dcc-login-02 ~/slurm $ sbatch simple.sh
Submitted batch job 186554tm103@dcc-login-02 ~/slurm $ cat test.out
dcc-core-14
```

### Long-form commands example

```
#SBATCH --output=slurm.out
#SBATCH --error=slurm.err
#SBATCH --mem=100 # 100 MB RAM
#SBATCH --partition=scavenger#
hostname 1>&2  #prints hostname to the error file
```

This job will run in low priority on a lab node in the “scavenger” partition

## Short-form commands example.

SLURM short commands don’t use “=“ signs

```
#!/bin/bash
#SBATCH -o slurm.out
#SBATCH -e slurm.err
#SBATCH --mem=4G # 4 GBs RAM
#SBATCH –p scavenger
hostname 1>&2  #prints hostname to the error file
```

## R example script

```
#!/bin/bash
#SBATCH –e slurm.err
#SBATCH --mem=4G # 4 GB RAM
module load R/3.6.0
R CMD BATCH Rcode.R
```

This loads the environment module for R/3.6.0 and runs a single R script (“Rcode.R”)

The `#SBATCH –mem=4G` requests additional RAM

## Multi-threaded (multi-core) example

```
!/bin/bash
#SBATCH –J test
#SBATCH –o test.out
#SBATCH –c 4
#SBATCH –-mem-per-cpu=500 #(500 MB)
myApplication –n $SLURM_CPUS_PER_TASK
```

The value of `$SLURM_CPUS_PER_TASK` is the number after `-c`. This example starts a single, multi-threaded job that uses 4 CPU cores and 2 GB (4x500MB) of RAM

## OpenMP multicore example

```
!/bin/bash
#SBATCH –J openmp-test
#SBATCH –o slurm.out
#SBATCH –c 4
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
myOpenMPapp # will run on 4 CPU cores
```

This sets `$OMP_NUM_THREADS` to the value of `$SLURM_CPUS_PER_TASK`

# Slurm job arrays

Slurm job arrays are a mechanism for submitting and managing collections of similar jobs using one job script and one application program.

- Add `--array` or `-a` option to the job script
- Each job task will inherit a `SLURM_ARRAY_TASK_ID` environment variable with a different integer value

- Each job array can be up 100,000 job tasks on the DCC
- Job arrays are only supported for batch jobs
- Job array "tasks" must be independent: slurm.schedmd.com/job_array.html

For example, in a job script, add the line `#SBATCH --array=1-30` or, alternatively, `#SBATCH -a 1-30` to submit 30 job tasks. The job array indices can also be specified on the command line, e.g.

```
sbatch -a 1-30 myjob.sh
```

The index values can be continuous, e.g.

```
-a 0-31 (32 tasks, numbered from 0,1,2,…,31)
```

or discontinuous, e.g.

```
-a 3,5,7-9,12 (6 tasks, numbers 3,5,7,8,9,12)
```

It can also be a single job task, e.g.

```
-a 7
```

The discontinous notation is useful for resubmitting specific job tasks that had previously failed. Each job task is assigned the enviromental variable `$SLURM_ARRAY_TASK_ID` set to it's index value.

```
tm103@dcc-login-02  ~/misc/jobarrays $ cat array-test.sh
#!/bin/bash
echo $SLURM_ARRAY_TASK_ID
tm103@dcc-login-02  ~/misc/jobarrays $ sbatch -a 1-3 array-test.sh
Submitted batch job 24845830
tm103@dcc-login-02  ~/misc/jobarrays $ ls slurm-24845830*
slurm-24845830_1.out  slurm-24845830_2.out  slurm-24845830_3.out
tm103@dcc-login-02  ~/misc/jobarrays $ cat slurm-24845830*
1
2
3
tm103@dcc-login-02  ~/misc/jobarrays $
```

### Python job array example

```
#!/bin/bash
#SBATCH -e slurm_%A_%a.err
#SBATCH -o slurm_%A_%a.out
#SBATCH --array=1-5000
python myCode.py

$ cat test.py
import os
taskID=int(os.environ['SLURM_ARRAY_TASK_ID'])
...
```

Start 5000 Python jobs, each with a different "taskID", initialized from `$SLURM_ARRAY_TASK_ID`

## Importing Environmental Variables

### Python:

```
numCPUs=int(os.environ['SLURM_CPUS_PER_TASK'])
taskID=int(os.environ['SLURM_ARRAY_TASK_ID'])
```

R:

```
numCPUs <- as.integer(Sys.getenv(SLURM_CPUS_PER_TASK))
taskID <- as.integer(Sys.getenv(SLURM_ARRAY_TASK_ID))
```

MATLAB:

```
numCPUs = str2num(getenv('SLURM_CPUS_PER_TASK'))
taskID = str2num(getenv('SLURM_ARRAY_TASK_ID'))
```

## Processing separate input files

Process an existing file list, e.g. files.txt

```
#!/bin/bash
readarray -t FILES < files.txt
FILENAME=${FILES[(($SLURM_ARRAY_TASK_ID - 1))]}
myapp $FILENAME
```

Dynamically generate a file list from “ls”

```
#!/bin/bash
export FILES=($(ls -1 myfile*))
FILENAME=${FILES[(($SLURM_ARRAY_TASK_ID - 1))]}
myapp $FILENAME
```

Example: Using the taskID as part of the file name and output directory for the case with input file names of the form input1,input2,...,inputN for -a 1-N, e.g.

```
#!/bin/bash
#SBATCH -e slurm_%A_%a.err
#SBATCH -o slurm_%A_%a.out
mkdir out_${SLURM_ARRAY_TASK_ID}
cd out_${SLURM_ARRAY_TASK_ID}
myapp ../input_${SLURM_ARRAY_TASK_ID}.txt
```

where output directories out1, out2, ... are created for input files input1.txt, input2.txt,...

## “Unrolling” for loops example

Original “serial” code (Python)

```
fibonacci = [0,1,1,2,3,5,8,13,21]
for i in range(len(fibonacci)):
    print(i,fibonacci[i])
```

Job array version

```
import os
i=int(os.environ['SLURM_ARRAY_TASK_ID'])
fibonacci = [0,1,1,2,3,5,8,13,21]
```

```
#for i in range(len(fibonacci)):
    print(i,fibonacci[i])
```

where the for loop is commented-out and each job task is doing a single “iteration”

```
tm103@dcc-login-02  ~/misc/jobarrays $ cat fib-array.sh
#!/bin/bash
#SBATCH -e slurm.err
module load Python/2.7.11
python fibonacci.py
tm103@dcc-login-02  ~/misc/jobarrays $ sbatch –a 1-8 fib-array.sh
Submitted batch job 24856052
tm103@dcc-login-02  ~/misc/jobarrays $ ls slurm-24856052_*
slurm-24856052_1.out  slurm-24856052_3.out  slurm-24856052_5.out  slurm-24856052_7.out
slurm-24856052_2.out  slurm-24856052_4.out  slurm-24856052_6.out  slurm-24856052_8.out
tm103@dcc-login-02  ~/misc/jobarrays $ cat slurm-24856052*
(1, 1)
(2, 1)
(3, 2)
(4, 3)
(5, 5)
(6, 8)
(7, 13)
(8, 21)
tm103@dcc-login-02  ~/misc/jobarrays $
```

## Running MPI jobs

Supported MPI versions are Intel MPI and OpenMPI

Compiling with OpenMPI

```
tm103@dcc-login-02  ~ $ module load OpenMPI/4.0.5-rhel8
OpenMPI 4.0.5-rhel8
tm103@dcc-login-03  ~ $ mpicc -o openhello hello.c
tm103@dcc-login-02  ~ $ ls -l openhello
-rwxr-xr-x. 1 tm103 scsc 9184 Sep  1 16:08 openhello`
```

OpenMPI job script

```
#!/bin/bash
#SBATCH -o openhello.out
#SBATCH -e slurm.err
#SBATCH -n 20
module load OpenMPI/4.0.5-rhel8
mpirun -n $SLURM_NTASKS openhello
```

OpenMPI example output

```
tm103@dcc-login-02 ~/misc/slurm/openmpi $ cat openhello.out
dcc-core-01, rank 0 out of 20 processors
dcc-core-01, rank 1 out of 20 processors
dcc-core-01, rank 2 out of 20 processors
dcc-core-01, rank 3 out of 20 processors
dcc-core-01, rank 4 out of 20 processors
dcc-core-03, rank 13 out of 20 processors
dcc-core-03, rank 14 out of 20 processors
dcc-core-03, rank 10 out of 20 processors
dcc-core-03, rank 11 out of 20 processors
dcc-core-03, rank 12 out of 20 processors
dcc-core-02, rank 8 out of 20 processors
dcc-core-02, rank 9 out of 20 processors
```

```
dcc-core-02, rank 5 out of 20 processors
```

# GPU jobs

To run a GPU batch job, add the job script lines

```
#SBATCH -p gpu-common
#SBATCH --gres=gpu:1
#SBATCH --exclusive
```

To get an interactive GPU node session, type the command line `srun -p gpu-common --gres=gpu:1 --pty bash -i` tm103@dcc-clogin-02 ~ $ srun -p gpu-common --gres=gpu:1 --pty bash -i tm103@dcc-gpu-01 ~ $ /usr/local/cuda-7.5/samples/1_Utilities/deviceQuery/deviceQuery ... Detected 1 CUDA Capable device(s)

```
Device 0: "Tesla K80"
    CUDA Driver Version / Runtime Version          7.5 / 7.5
    CUDA Capability Major/Minor version number:    3.7
    Total amount of global memory:                 11520 MBytes (12079136768 bytes)
    (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
```

## Cluster GPU Types

Request a specific GPU type and GPU device number with

```
#SBATCH --gres=gpu:(GPU type):(GPU number)
```

where "GPU number" was a number from 1 to 4 and "GPU type" was from the list below.

| GPU type | GPU full name | VRAM (GB) | availability notes |
|---|---|---|---|
| 2080 | nvidia_geforce_rtx_2080_ti | 11 | gpu-common, researcher owned, and scavenger-gpu |
| 5000_ada | nvidia_rtx_5000_ada_generation | 32 | gpu-common, researcher owned, and scavenger-gpu |
| a6000 | nvidia_rtx_a6000 | 30 | researcher owned and scavenger-gpu |
| 6000_ada | nvidia_rtx_6000_ada_generation | 48 | researcher owned and scavenger-gpu |
| p100 | tesla_p100-pcie-16gb | 16 | courses, scavenger-gpu |
| a5000 | nvidia_rtx_a5000 | 24 | researcher owned and scavenger-gpu |
| a100 | nvidia_a100_80gb_pcie | 80 | researcher owned |
| 6000 | quadro_rtx_6000 | 24 | researcher owned |
| h100 | nvidia_h100_nvl | 80 | researcher owned |

To see all GPU types and count on the cluster, run `gpuavail` from a login node. This command will also provide a

realtime listing of GPUs available.

```
kk338@dcc-login-04  ~ $ gpuavail

     CONFIGURATION
  NODE                          NODE
  TYPE                         COUNT
  ---------------------
  gpu:a5000:1                   76
  gpu:2080:1                    74
  gpu:5000_ada:1                28
  gpu:6000_ada:1                24
  gpu:p100:2                    10
  gpu:a6000:1                    8
  gpu:a5000:4                    6
  gpu:2080:4                     4
  gpu:6000_ada:4                 1
  gpu:2080:3                     1
  gpu:5000_ada:4                 1


                AVAILABILITY
  NODE                          GPU       GPU     GPUs    CPUs    GB MEM
  NAME                          TYPE      COUNT   AVAIL   AVAIL   AVAIL
  ------------------------------------------------------------------
  dcc-allenlab-gpu-01           a6000     1       1       9       114
  dcc-allenlab-gpu-02           a6000     1       1       9       114
  dcc-allenlab-gpu-03           a6000     1       1       9       114
  dcc-allenlab-gpu-04           a6000     1       1       9       114
  dcc-allenlab-gpu-05           6000_ada  1       1       11      118
  dcc-allenlab-gpu-06           6000_ada  1       1       11      118
  dcc-allenlab-gpu-07           6000_ada  1       1       11      118
  dcc-allenlab-gpu-08           6000_ada  1       1       11      118
  dcc-allenlab-gpu-09           6000_ada  1       1       11      118
  dcc-allenlab-gpu-10           6000_ada  1       1       11      118
  dcc-allenlab-gpu-11           6000_ada  1       1       11      118
  dcc-brunellab-gpu-01          2080      1       1       13      51
  dcc-brunellab-gpu-02          2080      1       1       21      83
  dcc-brunellab-gpu-03          2080      1       1       21      83
  dcc-brunellab-gpu-04          2080      1       1       21      83
  dcc-carlsonlab-gpu-01         2080      1       1       21      83
```

# Job dependencies

Submit a job that waits for another job to finish.

```
$ sbatch dep1.q
Submitted batch job 666898
```

Make a note of the assigned job ID of dep1

```
$ sbatch --dependency=afterok:666898 dep2.q
Job dep2 will not start until dep1 finishes
```

## Job dependencies with arrays

Wait for specific job array elements

```
sbatch --depend=after:123_4 my.job sbatch --depend=afterok:123_4:123_8 my.job2
```

Wait for entire job array to complete

```
sbatch --depend=afterany:123 my.job
```

Wait for entire job array to complete successfully

```
sbatch --depend=afterok:123 my.job
```

Wait for entire job array to complete and at least one task fails

```
sbatch --depend=afternotok:123 my.job
```

More information: hcc-docs.unl.edu/display/HCCDOC/Job+Dependencies

## Additional Resources

http://schedmd.com