Notice

2017-08-31: Configuration change to allow allocation on CPUs and RAM. Please read the 'Default Quota' section under https://howto.cs.uchicago.edu/techstaff:slurm#usage

Peanut Job Submission Cluster

We are currently **alpha** testing and gauging user interest in a cluster of machines that allows for the submission of long running compute jobs. Think of these machines as a dumping ground for discrete computing tasks that might be rude or disruptive to execute on the main (shared) shell servers (i.e., linux1, linux2, linux3).

For job submission we will be using a piece of software called SLURM. Simply put, SLURM is a queue management system and stands for **S**imple **L**inux **U**tility for **R**esource **M**anagement; it was developed at the Lawrence Livermore National Lab. It currently supports some of the largest compute clusters in the world. The best description of SLURM can be found on its homepage:

"Slurm is an open-source workload manager designed for Linux clusters of all sizes. It provides three key functions. First it allocates exclusive and/or non-exclusive access to resources (computer nodes) to users for some duration of time so they can perform work. Second, it provides a framework for starting, executing, and monitoring work (typically a parallel job) on a set of allocated nodes. Finally, it arbitrates contention for resources by managing a queue of pending work."[1)]

SLURM is similar to most other queue systems in that you write a batch script, then submit it to the queue manager. The queue manager schedules your job to run on the queue (or partition in SLURM parlance) that you designate. Below is an outline of how to submit jobs to SLURM, how SLURM decides when to schedule your job, and how to monitor progress.

Where to begin

SLURM is a set of command line utilities that can be accessed via the command line from **most** any computer science system you can login to. Using our main shell servers (linux.cs.uchicago.edu) is expected to be our most common use case, so you should start there.

```
ssh user@linux.cs.uchicago.edu
```

Mailing List

If you are going to be a user of this cluster please sign up for the mailing list. Downtime and other relevant information will be announced here.

Mailing List

Documentation

The SLURM webiste should be your primary source for documentation. If you Google SLURM questions, you'll often see the outdated Lawrence Livermore pages.

A great way to get details on SLURM commands are the manuals that are already on the cluster. For example, if you type the following command:

```
man sbatch
```

you will get the manual page for the `sbatch` command.

Resources

- Common SLURM commands
- Official SLURM website
- Official SLURM documentation
- SLURM tutorial videos
- LLNL quick start user guide
- Yale's User Guide

Infrastructure

Hardware

Our cluster contains nodes with the following specs:

- 16 Cores (2x 8core 3.1GHz Processors), 16 threads
- 64gb RAM
- 2x 500GB SATA 7200RPM in RAID1

Storage

There is slow scratch space mounted to `/scratch`. It is a ZFS pool consisting of 10x 2TB 7200RPM SAS drives connected via a LSI 9211-8i and is made up of 5 mirrored VDEVs, which is similar to a RAID10. The servers uplink is 1G ethernet.

- Files older than 90 days will be deleted automatically.
- Scratch space is shared by all users.

Access

Scratch space is only mounted on nodes associated with the cluster. If you want to be able to transfer files to the scratch space you will want to run an interactive shell. Now you will be able to use

standard tools such as `scp` or `rsync` to transfer files.

1. You should only do a file transfer via the debug partition: `srun -p debug --pty --mem 500 /bin/bash`
2. Now you can create a directory of your own: `mkdir /scratch/$USER` You should store any files you create in this directory.

Example

Request interactive shell

```
user@csilcomputer:~$ srun --pty --mem 500 /bin/bash
```

Create a directory on the scratch partition if you don't already have one:

```
user@slurm1:~$ mkdir -p /scratch/$USER
```

Change into my scratch directory:

```
user@slurm1:~$ cd /scratch/$USER/
```

Get the files I need:

```
user@slurm1:/scratch/user$ scp user@csilcomputer:~/foo .
foo                           100%  103KB 102.7KB/s   00:00
```

Check that the file now exists:

```
user@slurm1:/scratch/user$ ls -l foo
-rw------- 1 user user 105121 Dec 29  2015 foo
```

I can now exit my interactive shell.

Performance is slow

This is expected. The maximum speed this server will ever be able to achieve is 1Gb/s because of its single 1G ethernet uplink. If this cluster gains in popularity we plan on upgrading the network and storage server.

Utilization Dashboard

Sometimes it is useful to see how much of the cluster is utilized. You can do that via the following URL: http://peanut.cs.uchicago.edu

Partitions / Queues

To find out what partitions we offer, checkout the sinfo command.

As of December, 2015 we have will have at least 2 partitions in our cluster; 'debug' and 'general'.

Partition Name	Description
debug	The partition your job will be submitted to if none is specified. The purpose of this partition is to make sure your code is running as it should before submitting a long running job to the general queue.
general	All jobs that have been thoroughly tested can be submitted here. This partition will have access to more nodes and will process most of the jobs. If you need to use the `--exclusive` flag it should be done here.
pascal	2018-05-04: 1x Nvidia GTX1080. You will be forced to use this server exclusively for now. Please keep your time in interactive mode to a minimum.
titan	2018-05-04: 4x Nvidia GTX1080Ti. This partition is shared and you MUST use the `--gres` to specify the resources you wish to use. It is also encouraged to specify cpu and memory.

Job Submission

Jobs submitted to the cluster are run from the command line. Almost anything that you can run via the command line on any of our machines in our labs can be run on our job submission server agents.

The job submission servers run Ubuntu 14.04 with the same software as you will find on our lab computers, but without the X environment.

You can submit jobs from the departmental computers that you have access to. You will not be able to access the job server agent directly.

Command Summary

Cheat Sheet

	SLURM	Example
Submit a batch serial job	sbatch	sbatch runscript.sh
Run a script interactively	srun	srun –pty -p interact -t 10 –mem 1000 /bin/bash /bin/hostname
Kill a job	scancel	scancel 4585
View status of queues	squeue	squeue -u cnetid
Check current job by id	sacct	sacct -j 999999

Usage

Below are some common examples. You should consult the documentation of SLURM if you need further assistance.

Default Quotas

By default we set a job to be run on one CPU and allocate 100MB of RAM. If you require more than that you should specify what you need. Using the following options will do: `--mem-per-cpu`, `--nodes`, `--ntasks`.

Exclusive access to a node

You will need to add the `--exclusive` options to your script or command line options. This option will ensure that when your job runs it is the only job running on that particular node.

sbatch

The sbatch command is used for submitting jobs to the cluster. sbatch accepts a number of options either from the command line, or (more typically) from a batch script. An example of a SLURM batch script is shown below:

Sample script

Make sure you create a directory in which to deposit the STDIN, STDOUT, STDERR files.

```
mkdir -p $HOME/slurm/out
```

```
#!/bin/bash
#
#SBATCH --mail-user=cnetid@cs.uchicago.edu
#SBATCH --mail-type=ALL
#SBATCH --output=/home/cnetid/slurm/out/%j.%N.stdout
#SBATCH --error=/home/cnetid/slurm/out/%j.%N.stderr
#SBATCH --workdir=/home/cnetid/slurm
#SBATCH --partition=debug
#SBATCH --job-name=check_hostname_of_node
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --mem-per-cpu=500
#SBATCH --time=15:00

hostname
```

If any of the above options are unclear as to what they do please check the man page for `sbatch`

```
man sbatch
```

Make sure to replace all instances of the word `cnetid` with your CNETID.

Submitting job script

Using the above example you will want to place your tested code into a file. 'hostname.job' is the file name in this example.

```
sbatch hostname.job
```

You can then check the status via squeue or see the output in the output directory '$HOME/slurm/slurm_out'.

srun

Used to submit a job to the cluster that doesn't necessarily need a script.

```
user@host:~$ srun -n2 hostname
research2
research2
```

`srun` will remain in the foreground until the job has finished.

```
user@host:~$ srun -n1 sleep 400
```

squeue

This command will show jobs in the queue.

```
user@host:~$ squeue
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
   29     debug    sleep     user  R       0:11      1 research2
```

scancel

Cancel one of your own jobs. Please read the `scancel` manual page (`man scancel`) as there are many ways of canceling your jobs if they are of any complexity.

```
scancel 29
```

sinfo

View information about SLURM nodes and partitions.

The following code block shows the what happens when you run the `sinfo` command. You get a list of 'partitions' on which you can run your code. Each partition is comprised of certain types of nodes. In the case below the default (denoted by a *) is 'debug'. The job time limit is short and is meant only to debug your code. The other partitions will usually have a particular purpose in mind. 'hardware', for example, is to be used if you require direct access to the hardware instead of the KVM layer between the hardware and the OS.

```
user@host:~$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
debug*       up       30:00      1   idle slurm1
general      up 14-00:00:00      6   idle slurm[2-6,8]
pascal       up  3-00:00:00      1   idle gpu2
tesla        up  3-00:00:00      1   idle gpu1
```

Monitoring Jobs

`squeue` and `sacct` are two different commands that allow you to monitor job activity in SLURM. `squeue` is the primary and most accurate monitoring tool since it queries the SLURM controller directly. `sacct` gives you similar information for running jobs, and can also report on previously finished jobs, but because it accesses the SLURM database, there are some circumstances when the information is not in sync with squeue.

Running `squeue` without arguments will list all currently running jobs. It is more common, though to list jobs for a particular user (like yourself) using the `-u` option…

```
squeue -u cnetid
```

or for a particular job id.

```
squeue -j 7894
```

Interactive Jobs

Though batch submission is the best way to take full advantage of the compute power in the job submission cluster, foreground, interactive jobs can also be run.

An interactive job differs from a batch job in two important aspects:

1. The partition to be used is the interact partition
2. Jobs should be initiated with the srun command instead of sbatch.

This command:

```
srun -p general --pty --cpus-per-task 1 --mem 500 -t 0-06:00 /bin/bash
```

will start a command line shell (`/bin/bash`) on the 'general' queue with 500 MB of RAM for 6 hours; 1 core on 1 node is assumed as these parameters (`-n 1 -N 1`) were left out. When the interactive session starts, you will notice that you are no longer on a login node, but rather one of the compute nodes dedicated to this queue. The `--pty` option allows the session to act like a standard terminal.

Job Scheduling

We use a multifactor method of job scheduling. Job priority is assigned by a combination of fair-share, partition priority, and length of time a job has been sitting in the queue. The priority of the queue is the highest factor in the job priority calculation. For certain queues this will cause jobs on lower priority queues which overlap with that queue to be requeued. The second most important factor is fair-share score. You can find a description of how SLURM calculates Fair-share here. The third most important is how long you have been sitting in the queue. The longer your job sits in the queue the higher its priority grows. If everyone's priority is equal then FIFO is the scheduling method. If you want to see what your current priority is just do `sprio -j JOBID` which will show you the calculation it does to figure out your job priority. If you do `sshare -u USERNAME` you can see your current fair-share and usage.[2)]

We also have backfill turned on. This allows for jobs which are smaller to sneak in while a larger higher priority job is waiting for nodes to free up. If your job can run in the amount of time it takes for the other job to get all the nodes it needs, SLURM will schedule you to run during that period. **This means knowing how long your code will run for is very important and must be declared if you wish to leverage this feature. Otherwise the scheduler will just assume you will use the maximum allowed time for the partition when you run.**[3)]

Common Issues

Error	**What does it mean?**
JOB <jobid> CANCELLED AT <time> DUE TO TIME LIMIT	You did not specify enough time for your job to run. The `-t` flag will allow you to set the time limit.
Job <jobid> exceeded <mem> memory limit, being killed	Your job is attempting to use more memory that you have requested for it. Either increase the amount of memory you have requested or reduce the amount of memory usage your application is trying to use.
JOB <jobid> CANCELLED AT <time> DUE TO NODE FAILURE	There can be many reasons for this message, but most often it means that the node your job was set to run on can no longer be contacted by the the SLURM controller.
error: Unable to allocate resources: More processors requested than permitted	It usually has **nothing** to do with priviledges you may or may not have. Rather, it usually means that you have allocated more processors than one compute node actually has.

Using the GPU

GRES Multiple GPU's on one system

GRES: Generic Resource. As of 2018-05-04 these only include GPU's.

Jobs will not be allocated any generic resources unless specifically requested at job submit time using the `--gres` option supported by the `salloc`, `sbatch` and `srun` commands. The option requires an argument specifying which generic resources are required and how many resources. The resource specification is of the form `name[:type:count]`. The name is the same name as specified by the GresTypes and Gres configuration parameters. type identifies a specific type of that generic resource (e.g. a specific model of GPU). count specifies how many resources are required and has a default value of 1. For example:

```
sbatch --gres=gpu:titan:2 ....
```

Jobs will be allocated specific generic resources as needed to satisfy the request. If the job is suspended, those resources do not become available for use by other jobs.

Job steps can be allocated generic resources from those allocated to the job using the `--gres` option with the `srun` command as described above. By default, a job step will be allocated all of the generic resources allocated to the job. If desired, the job step may explicitly specify a different generic resource count than the job. This design choice was based upon a scenario where each job executes many job steps. If job steps were granted access to all generic resources by default, some job steps would need to explicitly specify zero generic resource counts, which we considered more confusing. The job step can be allocated specific generic resources and those resources will not be available to other job steps. A simple example is shown below.

Ok, but I don't want to read the wall of text above

Fine.

The `--gres` (man srun) is required if you want to make use of a gpu.

```
  --gpu=gpu:N    # where 'N' is the number of GPUs requested.
                 # Please try to limit yourself to one GPU per person.
```

Example when using tensorflow:

Given the file `f`:

```
#!/usr/bin/env python3
from tensorflow.python.client import device_lib
print(device_lib.list_local_devices())
```

Here we can see that no GPU was allocated to us because we did not specify the `--gres` option

```
user@bulldozer:~$ srun -p titan --pty /bin/bash
user@gpu3:~$ ./f 2>&1 | grep physical_device_desc
user@gpu3:~$
```

If we request only 1 GPU.

```
user@bulldozer:~$ srun -p titan --pty --gres=gpu:1 /bin/bash
user@gpu3:~$ ./f 2>&1 | grep physical_device_desc
physical_device_desc: "device: 0, name: GeForce GTX 1080 Ti, pci bus id:
0000:19:00.0, compute capability: 6.1"
```

If we request 2 GPUs.

```
user@bulldozer:~$ srun -p titan --pty --gres=gpu:2 /bin/bash
user@gpu3:~$ ./f 2>&1 | grep physical_device_desc
physical_device_desc: "device: 0, name: GeForce GTX 1080 Ti, pci bus id:
0000:19:00.0, compute capability: 6.1"
physical_device_desc: "device: 1, name: GeForce GTX 1080 Ti, pci bus id:
0000:1a:00.0, compute capability: 6.1"
```

If we request more GPUs then are available.

```
kauffman3@bulldozer:~$ srun -p titan --pty --gres=gpu:5 /bin/bash
srun: error: Unable to allocate resources: Requested node configuration is
not available
```

Cool, but how do I know where and what resources are available

Turns out the `sinfo` command is super useful.

```
$ sinfo -O partition,nodelist,gres,features,available
PARTITION           NODELIST            GRES                FEATURES
AVAIL
debug*              slurm1              (null)              (null)
up
general             slurm[2-6,8]        (null)              (null)
up
pascal              gpu2                gpu:gtx1080:1       'pascal,gtx1080'
up
titan               gpu3                gpu:gtx1080ti:4
'pascal,gtx1080ti'  up
```

FEATURES: Is actually just an arbitrary string in the configuration file that defines a node. However, techstaff hopes it actually provides some useful info.

GRES: Don't depend on this being accurate, however it will definitely give you a clue as to how many generic resources are in a partition.

Checking how many Generic RESources are being consumed

Simple use the `-O` option for `squeue` and you can see how many generic resources any particular job is consuming.

```
$ squeue -O username,nodelist,gres
USER                NODELIST            GRES
someusername        gpu3                gpu:1
otherusername       gpu3                gpu:3
...
```

Environment Variables

CUDA_HOME, LD_LIBRARY_PATH

Please make sure you specify $CUDA_HOME and if you want to take advantage of CUDNN libraries you will need to append /usr/local/cuda-x.x/lib64 to the $LD_LIBRARY_PATH environment variable.

```
cuda_version=9.2
export CUDA_HOME=/usr/local/cuda-${cuda_version}
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CUDA_HOME/lib64
```

Currently we support the same versions of CUDA that the latest version of CUDNN supports. This is not written in stone and we can accommodate most other versions if required; just let techstaff know what your needs are.

PATH

You may also need to add the following to your $PATH

```
export PATH=$PATH:/usr/local/cuda/bin
```

CUDA_VISIBLE_DEVICES

Do not set this variable. It will be set for you by SLURM.

The variable name is actually misleading; since it does NOT mean the amount of devices, but rather the physical device number assigned by the kernel (e.g. /dev/nvidia2).

For example: If you requested multiple gpu's from SLURM (–gres=gpu:2), the CUDA_VISIBLE_DEVICES variable should contain two numbers(0-3 in this case) separated by a comma (e.g. 1,3).

Example

This sbatch script will get device information from the installed Tesla gpu.

```
#!/bin/bash
#
#SBATCH --mail-user=cnetid@cs.uchicago.edu
#SBATCH --mail-type=ALL
#SBATCH --output=/home/cnetid/slurm/slurm_out/%j.%N.stdout
#SBATCH --error=/home/cnetid/slurm/slurm_out/%j.%N.stderr
#SBATCH --workdir=/home/cnetid/slurm
#SBATCH --partition=gpu
#SBATCH --job-name=get_tesla_info

export PATH=$PATH:/usr/local/cuda/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH=/usr/local/cuda/lib

cat << EOF > /tmp/getinfo.cu
#include <stdio.h>

int main() {
  int nDevices;

  cudaGetDeviceCount(&nDevices);
  for (int i = 0; i < nDevices; i++) {
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, i);
    printf("Device Number: %d\n", i);
    printf("  Device name: %s\n", prop.name);
    printf("  Memory Clock Rate (KHz): %d\n",
           prop.memoryClockRate);
    printf("  Memory Bus Width (bits): %d\n",
           prop.memoryBusWidth);
    printf("  Peak Memory Bandwidth (GB/s): %f\n\n",
           2.0*prop.memoryClockRate*(prop.memoryBusWidth/8)/1.0e6);
  }
}
EOF

/usr/local/cuda/bin/nvcc /tmp/getinfo.cu -o /tmp/a.out
/tmp/a.out
rm /tmp/a.out
rm /tmp/getinfo.cu
```

Output

STDOUT will look something like this:

```
cnetid@linux1:~$ cat $HOME/slurm/slurm_out/12567.gpu1.stdout
Device Number: 0
  Device name: Tesla M2090
  Memory Clock Rate (KHz): 1848000
  Memory Bus Width (bits): 384
  Peak Memory Bandwidth (GB/s): 177.408000
```

STDERR should be blank.

More

If you feel this documentation is lacking in some way please let techstaff know. Email techstaff@cs.uchicago.edu, call (773-702-1031), or stop by our office (Ryerson 154).

[1)]

http://slurm.schedmd.com/

[2)] , [3)]

https://rc.fas.harvard.edu/resources/running-jobs