Downstate Compute Cluster

What It Comprises

Sixteen 64-core nodes, each with 512 GB of RAM and 7 TB of local working disk (job tmp. space), one 32-core node with two nVIDIA A30 GPUs, 1024 GB of RAM, and 5 TB of local working disk, and, one 16-core "dev." node with 128 GB of RAM and 400 GB of local working disk. All nodes are served by 1.2 PB of high-performance storage and the entire framework is connected by 100-Gb Ethernet. The job scheduler is (modernized) Grid Engine, and the OS is Linux, so users are expected to have a basic understanding of Linux and its commonly used commands.

What It Is Ideal For

CPU- and GPU-based jobs that are too demanding of a desktop, but that are not worth the hassles associated with state and national clusters. The cluster is NOT certified for the storage of HIPAA-regulated data, so, absolutely NO HIPAA-regulated data are to be stored or processed on this cluster.

Getting Started

To gain access to the compute cluster, storage is rented in 1-TB quanta, at least 1 TB to start, and at a cost of $50/TB/year. Users do not have to have individual billings associated with themselves: as long as someone (e.g., a PI) is paying for their storage, they have access. Storage quotas are only limited by funds available, and, can be easily and quickly changed whenever necessary/requested.

Once the amount of storage has been settled and paid for, please send two e-mails to, scott.bunnell@downstate.edu: the first is a statement confirming that you understand that no HIPAA-regulated data are ever to be stored or processed on the cluster; the second provides a desired username, as well as a mobile number, so that I can text to you your initial password. Please note that if you decide to change your password, any new password *must be* 15+ characters in length.

The access point for the compute cluster is the gateway machine, lethe.downstate.edu

Only SSH-based connectivity is allowed. Windows users can connect via PuTTY (or MobaXterm). The gateway is, by default, open to IP addresses from Canada, the USA, and Europe, but, if you need a specific subnet allowance, please let me know. The SSH port has been changed to 1415, so, to effect a successful connection, you will have to add, '-p1415', to your SSH client statement, e.g.:

ssh -p1415 sbunnell@lethe.downstate.edu

Please (please!!!) be careful establishing connections: multiple unsuccessful connection attempts (>5) within a 24-hour period will trigger a permanent ban of the client IP address (if you find yourself banned, please e-mail me with the IP address in question).

Also please heed the warning presented once connected to lethe. It is VERY important that the gateway server NOT be used as a data-processing server. If you need clarification regarding this, please e-mail me.

Moving Data To/From Cluster Storage

It is recommended that data transfers be effected with 'rsync' (available by default for Mac/Linux, and, as part of MobaXterm for Windows users). Here is an example of moving the data folder, "testdata", from my computer to my cluster home:

rsync -av -e 'ssh -p1415' testdata sbunnell@lethe.downstate.edu:/ddn/sbunnell/

And an example moving data from the cluster to my computer (to my current directory: note the period at the end of the command):

rsync -av -e 'ssh -p1415' sbunnell@lethe.downstate.edu:/ddn/sbunnell/testdata .

Using The Cluster: Non-Interactive Jobs

The most common job submission is a non-interactive one. The provided template.sh script is a framework that most jobs will follow.

template.sh comprises three sections. The first section is the shell call:

#!/bin/bash

and that can be left alone, unless you want to use a different shell. The second section contains the options provided to the cluster scheduler:

#$ -N <jobname>
#$ -cwd
#$ -pe smp <corecount>
#$ -l h_vmem=<memoryvalue>G

Replace, <jobname>, with the name that you want your cluster job to have.

'-cwd', makes the current working directory (most likely the job-execution directory in your home on the shared storage) the "job working directory", and your standard output file will appear here.

Replace, <corecount>, with the degree of (multithreaded) parallelism (i.e., cores) you attach to your job (up to 64), and this line needs to be deleted or commented

out if not relevant (e.g., the job in question is a single-core job).

Replace, <memoryvalue>, with the value (in GB), that you want your job's memory space to be (up to 494), and all jobs MUST HAVE a memory-space assignment. If you are unsure about memory footprints, err on assigning a generous memory space to your job so that it does not get prematurely terminated (the value given to h_vmem is a hard memory ceiling). However, it is prudent to refine the value of h_vmem for future jobs that are of a similar nature (such that it more closely matches the actual job memory footprint), as jobs that request a low-memory footprint are more likely to load on a busy cluster. Completed jobs list the maximum value of memory reached, and this information can be gleaned with:

qacct -j <jobID> | grep maxvmem

The final section of the job script is what is expected to be executed. A common execution framework comprises a data-movement process from the shared cluster storage (e.g, your home directory) to the local disk of the compute node (represented by TMPDIR), the program call (the actual data analysis) that generates results on *the local disk*, and, finally, a data-movement line that copies the results from the local disk to the shared cluster storage. Why use TMPDIR (the local disk of the compute node) and not just effect job I/O directly on the shared cluster storage? Performance. The local disk of each compute node is a large RAID 0 of SSD drives, and, no parallel file system can handle massive chronic data I/O, so, file system slowdowns on the shared storage are kept to a minimum if jobs that expect to touch data-input and -output files more than once make use of TMPDIR (as the shared file system is only then accessed at the start and end of a job). As stated at the beginning of this document, most nodes have 7 TB of local working disk, which should easily accommodate all data-processing tasks. Note, though, and this is extremely important, that $TMPDIR *only exists for the duration of the job, so, it is very important to copy relevant results from it at job's end*.

Reviewing the final section of the template.sh script:

```
cd $TMPDIR
mkdir input
rsync -av $SGE_O_WORKDIR/references/ input/
mkdir results
PROGRAM --inFiles $TMPDIR/input --outFiles $TMPDIR/results --parallel $NSLOTS
rsync -av $TMPDIR/results/ $SGE_O_WORKDIR/processed/
```

The job is instructed to change to $TMPDIR, to make the new directory, "input", to copy data from, "$SGE_O_WORKDIR/references/", to "input/" ($SGE_O_WORKDIR represents the "job working directory": see '-cwd', above), to make the new directory, "results", to execute the hypothetical program, PROGRAM, (which takes input from, "$TMPDIR/input", and writes output to, "$TMPDIR/results"), and, finally, to copy the total end results to an existing directory, "processed", that is located in the current working directory (again, likely [somewhere in] your home on the shared storage). Note the presence in the program call of the variable, NSLOTS. Often programs able to process data in parallel require the user to declare the

desired degree of parallelism, and $NSLOTS resolves to the value provided to the scheduler option, '-pe smp'.

Jobs cannot be submitted from lethe. Please log in to grid ('ssh grid') for job submissions. Right now is also an excellent time to consider setting up SSH keys for password-less login between machines in the cluster framework: if you need help with that, please let me know.

Once that your job script is ready, you can submit it with:

qsub ./<myscript>.sh

You can check the status of your job(s) with:

qstat -f -u <username>

And to see all jobs:

qstat -f -u "*"

If you need to delete a job, the syntax is:

qdel <jobID>

Note that jobs that do not specify output direction will generate a standard output file, in the directory that the job was submitted from, in the name format of, "<jobname>.o<jobID>". Important information is often written to this file.

Using The Cluster: Interactive Jobs

Not all jobs are best served by scripting (e.g., jobs that require X11 interaction); for those, there is the interactive job. Interactive jobs are given a full node of resources (i.e., 64 cores and 494 GB of memory). An example of invoking, 'qlogin':

qlogin -N qlogin-test

Please do not forget to log off of the node in question once you have finished interacting (otherwise you will idle, but, idling still counts against your core limit and your usage history for future job scheduling).

Using The Cluster: GPU Jobs

Jobs are by default sent to the CPU queue. For jobs that take advantage of GPUs for data processing, there is a GPU queue that can be specified by adding, '-q gpu.q', to the, 'qsub' statement, e.g.,

qsub -q gpu.q ./<mygpuscript>.sh

The status of the GPU queue can be queried with:

qstat -f -u "*" -q gpu.q

And you can query, nvidia-smi, on the GPU node with:

ssh node01 nvidia-smi

Only two GPU-queue jobs are allowed at any given time, and, no user may have more than one running job at any given time (excess jobs will sit, waiting, in the queue). All GPU jobs are allotted one GPU, 16 CPU cores, and, 499 GB of memory. Please do not execute non-GPU-utilizing jobs in the GPU queue (i.e., be sure that your job utilizes GPU processing before submission). Because of the nature of interactive jobs, and, the limited GPU resources afforded, no interactive jobs are allowed on the GPU queue (they are automatically rejected).

Using The Cluster: Development Jobs

There is a special queue set aside for jobs that are in testing phase (for proper syntax, for example), which is especially handy when the main queue is completely full of long-run data-processing tasks. This development queue accepts jobs that are submitted with, '-q dev.q':

qsub -q dev.q ./<mydevscript>.sh

And querying the queue status is similar to that effected for the GPU queue:

qstat -f -u "*" -q dev.q

Four dev.-queue jobs are possible at any given time, with users limited to two running jobs (at any given time). All dev. jobs are allotted four CPU cores and 29 GB of memory. Runtime is capped at *five minutes*. The dev. node has very limited local working disk, so refrain from large data transfers (again, this is a queue simply to test job frameworks). Like for the GPU queue, no interactive jobs are allowed on the development queue.

Important Notes/Caveats

Jobs run in the main queue and the GPU queue have a maximum runtime of one week.

Core limit for users is capped at 512 cores.

The cluster is, "first come, first served", until it fills, and then job position in the queue is based upon past usage. The scheduler does attempt to fill gaps, though, so sometimes a single-core job of lower priority will schedule before a multi-core

job of higher priority, for example.

Jobs should NEVER be run outside of the province of the scheduler. Repeat offenders risk loss of compute privileges.

Files/scripts must have Linux line breaks in them (not Windows ones).

If you are going to move many many files between shared storage and local disk, tar everything up first.

Watch storing lots of little files (hundreds of thousands per TB) in your shared-storage home: Lustre has a finite number of inodes (file pointers) that can easily become exhausted if not careful (this is BAD), and, as such, your storage quota includes a file limit, too. It is prudent to keep collections of small files in a tar archive. Storage and file numbers can be checked with:

lfs quota -u <username> /ddn

And for group quotas, with:

lfs quota -g <groupname> /ddn

Software
The cluster does not provide users with a panoply of pre-installed software. For items that do *not* require root management, please install those in your home. Anything that does require root will need to be effected in a Singularity container.

Singularity
The ability to run a self-contained and immutable runtime framework within the confines of the existing cluster environment (that also retains the ability to interact with said cluster environment) is of massive benefit, primarily to those requiring root-level manipulations and/or a differing operating-system distributions (than what the compute nodes run). Containers built in Singularity offer such a runtime framework.

Singularity v3.11 is available on the cluster, and all users have had it added to their PATH, as well as shell-sourced autocompletion for the Singularity commands.

Containers are almost always built as root, which means that users should install Singularity on their personal machine(s) and build containers there *as root*. (In theory, users should be able to build non-root containers on the cluster, BUT, the Lustre file system restricts many common build options, such that non-root containers may not function properly, or, as planned.)

The software and installation and usage instructions are found here:

https://docs.sylabs.io/guides/latest/user-guide/index.html

What follows is a summary of the salient issues guiding the use of Singularity (see the aforementioned URL for detailed instructions and guidance), presenting a typical workflow, and finishing with an example of Singularity on the cluster.

A prudent technical aphorism is, “Build on my machine; run on the cluster.”.

Singularity containers are essentially either built from an existing container, or are built from scratch. Building from scratch requires a recipe file. Two types of Singularity containers can be built: sandbox; squashfs/SIF. Sandbox containers are often used as development containers, are essentially a directory in an existing read-write space, and, are, “persistent writable” (meaning that changes can persist after the Singularity session ends). Squashfs/SIF containers are a read-only compressed "file”. Sandbox containers are easily converted into squashfs/SIF ones.

A typical workflow, therefore, is thus:
1- as root on your workstation, build a squashfs/SIF container, optionally using sandbox mode to first test container viability/functionality.
2- copy to your cluster home this new (squashfs/SIF) container.
3- run the new container as part of a Grid Engine job.

Now, the example. Here, I use container-provided, diff, to compare the contents of /etc/redhat-release on the compute node (running Rocky Linux 8) to that of the container, which is a CentOS 7 build.

First, as root, I built a very basic CentOS container using this .txt recipe:
——
BootStrap: yum
MirrorURL: http://mirror.centos.org/centos-7/7/os/$basearch/
Include: diffutils
——

The first line indicates that it is to be a CentOS container; the second line provides the install location for the OS, and, finally, the third line presents the package that I want installed on top of the base OS.

The squashfs/SIF container was then created with:

singularity build Base7 baseCentOS.txt

I then changed the ownership of the container to sbunnell:sbunnell, and, copied it over to my cluster home.

After logging in to lethe, and, moving over to grid (recall that you cannot run cluster jobs from lethe), I wrote the simple job script, containerexample.sh:

——

```
#!/bin/bash

#$ -cwd
#$ -l h_vmem=100G

more /etc/redhat-release | singularity exec /ddn/sbunnell/containers/Base7 diff - /
etc/redhat-release > difference.txt
——
```

The execution line feeds the contents of /etc/redhat-release from the compute node into the container, where it performs the difference comparison between that and the contents of the container's /etc/redhat-release, writing the results to a text file. Note that I do not make use of TMPDIR here, as no aspect of the job is touched more than once.

The job script was submitted as so:

```
qsub ./containerexample.sh
```

And the results file, difference.txt, contains:

```
——
1,4c1
< ::::::::::::::
< /etc/redhat-release
< ::::::::::::::
< Rocky Linux release 8.7 (Green Obsidian)
---
> CentOS Linux release 7.9.2009 (Core)
——
```