Working on the openmind computing cluster

BCS computational tutorial series, January 12 2016
Objectives

1. Learn the basics of how to use Openmind, a computing cluster managed using SLURM (Simple Linux Utility for Resource Management).
   a. See some example workflows.
   b. Learn best practices for effective usage.
2. More generally, see some strategies for using remote computing resources

Along the way...

- Learn to use the terminal
- Introduction to shell scripting
- Learn some basics of Linux operating systems
What is Openmind?

- A set of managed computers (called nodes) accessible through a single portal (openmind.mit.edu), enabling users to perform computations which are infeasible or too time consuming for a personal workstation/laptop.
- Different types of nodes with different numbers of CPUs and GPUs:
  - ~20 cores (up to 24) or (40 - 48 hyperthreaded)
  - most have 256 GB RAM, some 512GB, and one with 1 TB
  - 0 or several (up to 4) GPUs
- Lots of disk space
  - 483 TB total, all directories available from any node through Lustre filesystem
  - 10 TB quota per group (/om), plus 5 GB per user (/home)
- 10Gb link to MIT
- Possibilities for expansion - subject to space constraints!
Openmind architecture

Login Node (openmind.mit.edu) - Interactive use, code development.

Compute Nodes – Run “Batch Jobs” on, both single and multi-processor.

Names like node001.mit.edu, node001.mit.edu ... node032.mit.edu ...
Getting an account

Submit a new issue to request an account:

Title: Account request

Message box: Attn @s-b: "mit_username beta_testers mit_email" PI: pi_name

For example, Attn @s-b: "ereming beta_testers ereming@mit.edu" PI: Mehrdad Jazayeri

As part of getting an account, you will also be subscribed to the openmind-announce mailing list at MIT. However, to get appropriate notifications from GitHub@MIT you will need to click on your settings and choose your primary email address.
How openmind works: Slurm

Simple Linux Utility for Resource Management. From slurm.schedmd.com:

Slurm is an open-source workload manager designed for Linux clusters of all sizes. It provides three key functions:

1. It allocates exclusive and/or non-exclusive access to resources (computer nodes) to users for some duration of time so they can perform work.
2. It provides a framework for starting, executing, and monitoring work (typically a parallel job) on a set of allocated nodes.
3. It arbitrates contention for resources by managing a queue of pending work.”
Connect to Openmind: necessary software

Terminal emulator:

- Windows - MobaXterm
  - http://mobaxterm.mobatek.net/
- Macintosh – Built-in Terminal application
- Linux – Terminal application

SSH Client – To log in to the head node at openmind.mit.edu

File Transfer – Transferring files between openmind and your local machine.

X Forwarding – Display graphics for those programs with a GUI interface (such as MATLAB) or that otherwise display images.
The Linux shell

(Terminal, OSX)

(MobaXterm, Windows)
Connect to Openmind

From your ssh/terminal application on your tutorial workstation or your laptop or on a machine at home:

```
$ ssh -Y <username>@openmind.mit.edu
```

- `Y` enables X11 forwarding.

You will be prompted for your MIT password.
Interactive Jobs

Interactive jobs are useful for debugging. To start an interactive job with a command line terminal:

```
$ srun -p om_interactive --pty bash
```

"om_interactive" is an argument to the "-p" (partition) option.

To use a GUI on openmind, use ssh X window tunneling.

```
$ module add openmind/srun.x11/v1
$ srun.x11 -p om_interactive
$ module add mit/matlab/2015a
$ matlab
```

"add" is a sub-command of "module." Try typing $ module --help.
Batch jobs

Batch jobs are jobs which do not require user intervention to complete.

Typically each job will be defined by two or more scripts:

- The submission (sbatch) script which asks for a resource allocation and launches the job.
- The job script which defines the actual task (i.e. Matlab, Python, Freesurfer, etc.)

$ sbatch my_job_script.sh

This command will return a string informing the user of the Job ID number.
Requesting resources

Resources are what SLURM manages. These are CPUs, Memory (RAM), and sometimes GPUs. Specify using options.

Form is ‘-<single letter flag><optional value>’ or ‘--<long form flag>=<optional value>’

Example:

```bash
$ sbatch -n1
```

or

```
$ sbatch --ntasks=1
```

“=” only used in conjunction with long-form flags. No spaces!

Some options only have a long form flag.
Requesting resources

- `-c, --cpus-per-task=<ncpus>`: advise the Slurm controller that ensuing job steps will require `ncpus` number of processors per task. Without this option, the controller will just try to allocate one processor per task.
- `--mem=<MB>`: specify the real memory required per node in MegaBytes. Use this or `--mem-per-cpu`.
- `--mem-per-cpu=<MB>`: minimum memory required per allocated CPU in MegaBytes. Use this or `--mem`.
- `-n, --ntasks=<number>`: request resources for a certain number of tasks
  - Note: the
- `-N, --nodes=<minnodes[-maxnodes]>`: request a minimum and maximum number of nodes to be allocated to the job.
- `-t, --time=<time>`: set a limit on the total run time of the job allocation. E.g. "hours:minutes:seconds" or "days-hours:minutes:seconds"

Quick example

Create script:

$ vim testScript.sh

vim text editor basics: [http://www.oregonwebradio.net/backup_fedora/tutorials/vim_li/quickstart.html](http://www.oregonwebradio.net/backup_fedora/tutorials/vim_li/quickstart.html)

Script:

```
#!/bin/sh
#
#SBATCH --ntasks=1
#SBATCH --time=10:00
#SBATCH --mem-per-cpu=100
hostname
hostname
```

requests one CPU and 100 MB of RAM, in the default queue for 10 minutes. When started, the job would run a first job step `hostname`, then when this is completed will run the second job step, which is also `hostname`.

Now submit:

$ sbatch testScript.sh

Load output file:

$ vim slurm-<jobid>.out

Output file

node002
srun command

“srun is used to submit a job for execution or initiate job steps in real time. srun has a wide variety of options to specify resource requirements, including: minimum and maximum node count, processor count, specific nodes to use or not use, and specific node characteristics (so much memory, disk space, certain required features, etc.). A job can contain multiple job steps executing sequentially or in parallel on independent or shared nodes within the job's node allocation.”

Using srun is not necessary for scripts which have only one task (i.e. run this Matlab script).

If srun is used to launch non-MPI tasks in a script where either -n or -N are larger than one, then the script will be launched -n or -N times. -n and -N values must be compatible; SLURM will adjust -N.
Parallel example

Create script:

$ vim testScript.sh

Script:

#!/bin/sh
#
#SBATCH --nodes=2
#SBATCH --ntasks=2
#SBATCH --time=10:00
#SBATCH --mem-per-cpu=100
srun hostname
srun hostname

requests one CPU and 100 MB of RAM for two steps, in the default queue for 10 minutes. When started, the job would run a first job step `hostname` on each of two nodes, then when this is completed will run the second job step again on the same two nodes. We could have set `--nodes=1` if we wanted both tasks on the same node, or not set it at all if we didn’t care.

Now submit:

$ sbatch testScript.sh

Load output file:

$ vim slurm-<jobid>.out

Output file:

node002
node003
node002
node003
Other useful sbatch options

- `-a, --array=<indexes>`: submit a job array, multiple jobs to be executed with identical parameters. An environment variable will be created called SLURM_ARRAY_TASK_ID. This can be passed to your function in the in the script call command. This will request the amount of resources you’ve asked for multiplied by the number of elements in the index.
- `-d, --dependency=<dependency_list>`: defer the start of this job until the specified dependencies have been satisfied completed.
  - Unfortunately, ‘dependency_list’ must be a jobid, which are only generated at time of job submission, making scripting trickier.
- `--mail-type=<type>`: Notify user by email when certain event types occur. Valid type values are NONE, BEGIN, END, FAIL, REQUEUE, ALL. Use with `--mail-user`
- `--mail-user=<user/email>`: User to receive email notification of state changes as defined by `--mail-type`.
- `-p, --partition=<partition_names>`: request a specific partition for the resource allocation.
- `--qos=<qos>`: request a quality of service for the job.
- `-w, --nodelist=<node name list>`: Request a specific list of nodes.
- `-x, --exclude=<node name list>`: explicitly exclude certain nodes.
Let’s do something useful: spike sorting!

Outline:

1. Create a Conda (Python) environment in which the software and dependencies can be installed
2. Install the software: phy.readthedocs.org/en/latest/
3. Write a script to batch spike sort multiple experiments simultaneously using the openmind cluster
4. View/manually edit the results

Why this example? Powerful open source data analysis tools have been created by and for scientists. Many of these are great use cases for a powerful computing cluster. However, these tools are not always straightforward to install and run on custom configured computing platforms.

Phy: Kenneth Harris et al.
Cconda and Python environments

An **environment** consists of the settings and software, including the particular version of Python, that exist around the programs that the user runs.

In some cases it is useful to have multiple environments when using multiple tools which are dependent on different versions of supporting packages or Python versions.

**Cconda** is a Python *package manager* application that installs, runs, and updates packages and their dependencies. It has several distributions.

**Miniconda** is a light distribution that includes only conda and conda-build, and installs Python. **Anaconda** also includes over 100 automatically installed scientific “packages” and their dependencies.
Install Phy

1. Add miniconda module:
   ```bash
   module add openmind/miniconda3/3.9.1
   ```
2. Create text file in your home directory called `.condarc` containing the text:
   ```yaml
   envs_dirs:
   - /<directory>/envs
   ```
3. Create environment with permissions for installations:
   ```bash
   conda create -n phy --clone=/cm/shared/openmind/miniconda3/3.9.1
   ```
4. Install dependencies:
   a. ```bash
      conda install -n phy pip numpy matplotlib scipy h5py pyqt ipython-notebook requests --yes
   ```
   b. ```bash
      source activate phy
   ```
   c. ```bash
      pip install cython
   ```
   d. ```bash
      pip install vispy
   ```
5. Install Phy
   a. ```bash
      pip install klustakwik2
   ```
   b. ```bash
      pip install phy
   ```
Install Phy

1. Add miniconda module:
   ```bash
   $ module add openmind/miniconda3/3.9.1
   ```

2. Create text file in your home directory called `condarc` containing the text:
   ```
   envs_dirs:
   - /<directory>
   ```

In Openmind, users do not have access to the “root” directory, which is where the default environment resides. It also defaults to creating new environments in the same directory. A good alternative is `/om/user/<user>/envs` (don’t put this on your home directory unless you have >1 GB of free space!)

The conda configuration file (`condarc`) is an optional runtime configuration file which allows advanced users to configure various aspects of conda, such as which channels it searches for packages, proxy settings, environment directories, etc.

*This isn’t the only situation where lack of root access can require alternate software configurations!*
3. Create environment with permissions for installations:

```
conda create -n phy --clone=/cm/shared/openmind/miniconda3/3.9.1
```

This creates a new environment which starts off as being identical to the default environment, with Python 3.4.
Install Phy

4. Install dependencies:
   a. `$ conda install -n phy pip numpy matplotlib scipy h5py pyqt ipython-notebook requests --yes`
   b. `$ source activate phy`
   c. `$ pip install cython`
   d. `$ pip install vispy`

   pip is another package manager and is used here where the packages are not supported by conda.
Install Phy

5. Install Phy
   a. $ pip install klustakwik2
   b. $ pip install phy
Download sample files into test directory

```
$ mkdir ~/PhyTutorial
$ cd ~/PhyTutorial
$ phy download hybrid_10sec.dat
$ phy download hybrid_10sec.prm
$ source deactivate
```

Alternately, the test directory could be placed into /om/user/<user>, and you would definitely want to do this will real data.
Copy scripts to Openmind

Using scp (secure copy):

```
$ scp <user-id>@<client-url>:<path-to-file>/Phy.sh <mit-id>@openmind.mit.edu:~/PhyTutorial
$ scp <user-id>@<client-url>:<path-to-file>/CopyFiles.sh <mit-id>@openmind.mit.edu:~/PhyTutorial
$ scp <user-id>@<client-url>:<path-to-file>/ProcessFiles.sh <mit-id>@openmind.mit.edu:~/PhyTutorial
```

MobaXterm users can drag and drop using the GUI.
Update .prm file

Replace:

```python
eperiment_name = 'hybrid_10sec'
```

with:

```python
import os
experiment_name = next(filename for filename in os.listdir('.') if filename.endswith('.dat'))
experiment_name = experiment_name[:-4]
```
Replicate sample data

Interpreter designation not necessary here if running script using `sh` command.

```bash
#!/bin/sh
ext=.dat
dir =~/PhyTutorial
fileName=$(basename $dir"/"*$ext)
baseFileName=${fileName%.*}

echo "Found file $baseFileName."
suffix=_phy # suffix for directories for individual datafiles

for i in {1..5}
do
  name=$baseName$i # file stripped of extension
  echo "copying $baseFileName$ext to $baseFileName$i$ext"
  cp $baseFileName$ext $baseFileName$i$ext
done

$ sh CopyFiles.sh
```

Pattern matching to find the .dat file we want to copy.
#!/bin/sh

ext=.dat
dir=~/PhyTutorial/
suffix=_phy # suffix for directories for individual datafiles
FILES=$dir*$ext # pattern matching to make list of datafiles in directory

count=0
maxCount=20

for i in $FILES
do
    name=${i%$ext} # file stripped of extension
    name=${name##*/} # file stripped of path
    echo "Processing $name ..."
    # Make folder for .dat file.
    folder=$dir$name$suffix
    mkdir $folder
    # Move .dat file into its own folder.
    mv $name$ext $folder"/$name$ext
    # Copy .prm file to each directory.
    echo "copying $prmfile ..."
    cp hybrid_10sec.prm $folder
    # Copy phy sbatch script to each directory.
    phyfile=$folder"/phy.sh"
    echo "copying $phyfile ..."
    cp hy.sh $folder
    # Run spike sorting
    scriptsDir=$(pwd)
    cd $folder
    sbatch phy.sh
    cd $scriptsDir
    echo "Finished processing $count files ..."
    ((count+=1))
    if [ "$count" -ge "$maxCount" ]; then
        echo "Job submission limit reached, exiting."
        break
    fi
done

$ sh ProcessFiles.sh
Submit!

```bash
sh ProcessFiles.sh
```

Check on submitted jobs with `squeue -u<user>` or `squeue -j<jobid>`.

To check on jobs which have already completed (or failed!) use `sacct -j<jobid>`.
Use X tunneling to do manual sorting

Start interactive job

```
$ module add openmind/srun.x11/v1
$ srun.x11 -p om_interactive
```

Run manual sorting GUI

```
$ source activate phy
$ cd hybrid_10sec_phy
$ phy cluster-manual hybrid_10sec.kwik
```

Alternately, on openmind7.mit.edu...

```
$ srun --x11 xterm # only on openmind7
```
Advanced topics:

Other parallel programming patterns:

All of the “parallelism” shown in this tutorial employed the “embarrassingly parallel” programming pattern.

For more complicated tasks which require communication between the individual tasks, there are other patterns, including those which don’t require lots of extra effort, such as parallel loops and GPU computing (e.g. Matlab). Others may require the use of special parallel programming interfaces, such as message passing interface (MPI) and openMP.

GPU computing can also be a much more efficient implementation for certain problems.

Virtualization and containers:

Provide ways to make software and hardware environments portable, allowing for repeatable computations across machines.
Intermission: best practices

Help keep openmind running smoothly!

Examples:

- Don’t run intensive computations on the head node!
- Don’t bypass the scheduler and ssh into a compute node!
- Install software into your user space
- Separate code, data, and temporary output
  - use version control (GitHub, GitHub@MIT)
  - separate interactive and non-interactive bits
- Understand your code
- Post issues when you run into something
  - search before you post
- Let your PIs know when things are working well or not
Satra’s tidbits

- Use **mosh** to connect
- Use **xpra** for X11 connections or other interactive software
- Use Python via **MiniConda** for most software pieces, modules otherwise
  - search for packages here: [https://anaconda.org](https://anaconda.org)
- Use **nipype** for dataflow computing
- Use **jupyter** for interactive computing and remote access
  - tunnel via SSH
- Separate environments, data, code, scratch, output
  - Different bash functions initiate different environments
    - project based separate shell environments and python environments
  - Git-based version control
  - Jupyter Notebooks
  - Dropbox API
- **Vagrant** for things that don’t work directly on openmind
- **StarCluster** for jobs on AWS
Interactive persistent connections

local$ mosh openmind.mit.edu

openmind$ xpra start :100 xterm

--- connect from remote client

openmind$ module add openmind/srun.x11/v1
openmind$ srun.X11
or
openmind$ srun.X11 -p gpu --gres=gpu:1

node0XX$ xterm
Interactive persistent connections - openmind7

local$ mosh openmind7.mit.edu

openmind7$ xpra start :100 xterm

--> connect from remote client

openmind$ DISPLAY=:100 srun --x11 --pty xterm
Remote Jupyter notebooks

$ mosh openmind.mit.edu

openmind$ srun -N1 -c2 --mem=10GB --pty bash

# https://github.mit.edu/MGHPC/OpenMind/issues/512#issuecomment-6931
	node0XX$ jupyter-notebook

# on local client terminal

local$ ssh satra@openmind.mit.edu -L1234:node0XX:8888

# open browser and point to http://localhost:1234
Using dataflow computing

An example: Nipype (nipy.org/nipype)

- Dataflow scripting platform
- Built in support for SLURM
- Allows embarrassing parallelism
- Written and scripted in Python
- Allows node level specification
  - gres
  - procs
- Allows running any command line tool (including MATLAB)
- Generates crashfiles that can be used to figure out failure modes
- Tracks provenance
Cluster upgrades

- Located in Holyoke, MA at the Massachusetts Green High Performance Computing Center (MGHPCC), a collaboration between 5 major universities and the Commonwealth of Massachusetts.
- Fair bit of room for expansion, but please ask early
- Contribute nodes or GPUs
  - provides additional priority for your group
  - maintenance done by openmind
  - contact before purchase to determine suitability