The Faculty of Arts and Sciences High Performance Computing Core

Advanced Computational Support for Scientific Research at Yale

FAS HPC Getting Started Workshop
Spring 2014

Andrew Sherman
Research Scientist & Lecturer in Computer Science
andrew.sherman@yale.edu

Agenda

- **What is HPC?**
  - **Background & History**
- HPC at Yale
  - On-Campus HPC & Data Storage Facilities
  - HPC Staff
  - Getting Assistance with HPC at Yale
  - National HPC Infrastructure
- Practicalities
  - Running Jobs on a Cluster (Interactive & Batch)
  - Requesting Resources
  - Module System
  - Screen
  - Pitfalls and Tips
  - Simple Queue

High Performance Computing

- What is it?
  - Use of most powerful/advanced computers to solve problems
  - Performance for scientific applications measured using “Linpack Benchmark”
- “Ancient” history: Very large monolithic computers, often “1-offs”

IBM 7094
(Similar to Yale computer in 1960-1970 timeframe)

CDC 7600 (c. 1970)
(36 MegaFlops Peak)

Cray 1 (c. 1976)
(250 MegaFlops Peak)

High Performance Computing

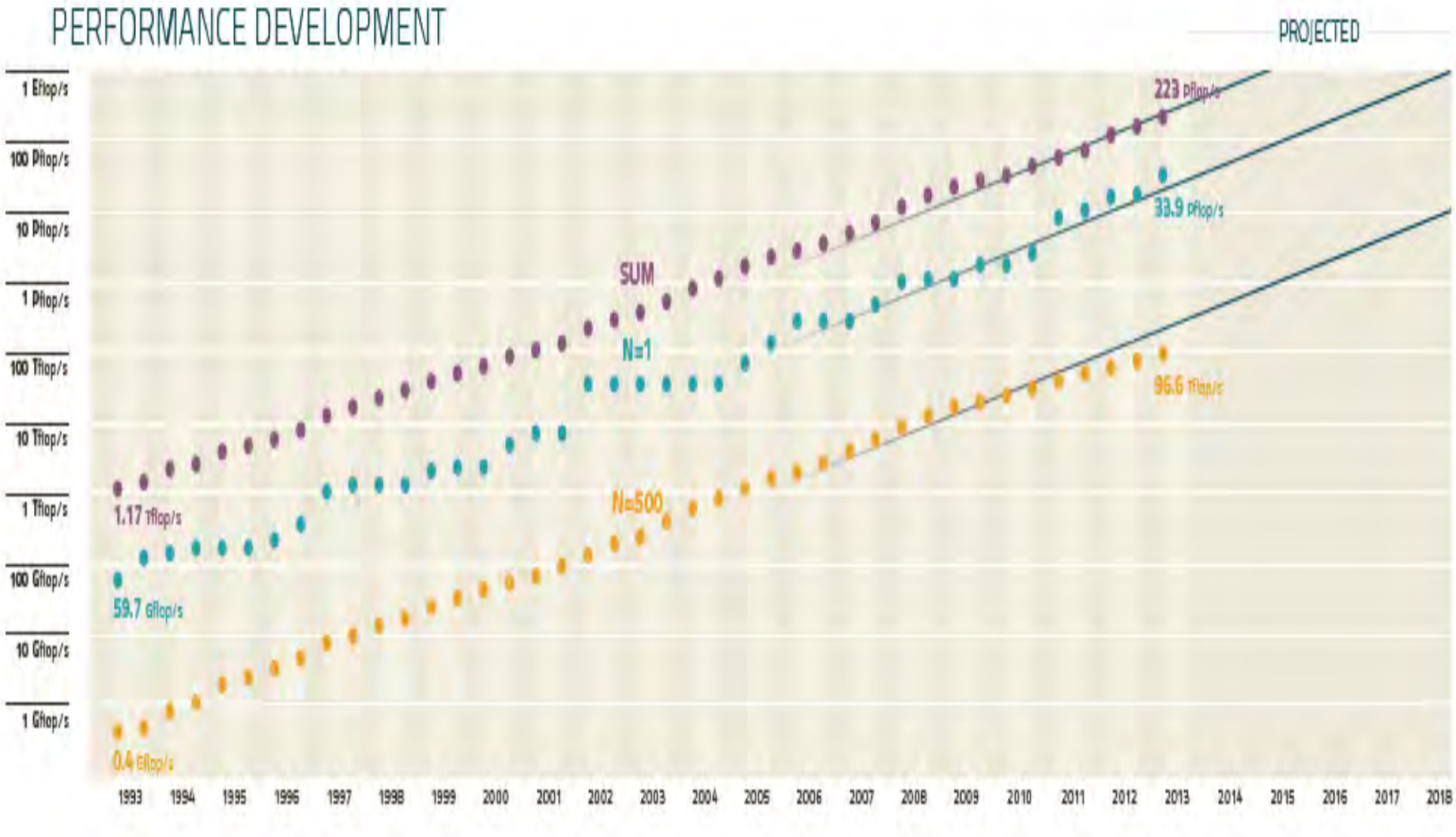
- What is it?
  - Use of most powerful/advanced computers to solve problems
  - Performance for scientific applications measured using “Linpack Benchmark”
- “Ancient” history: Very large monolithic computers, often “1-offs”
- Today: Almost always “parallel computers” (often networked “clusters” of many small commodity processors, but sometimes based on special-purpose computers such as the graphics processors on PC video cards)

Yale BulldogI Cluster
Similar to BulldogJ,
which has 1,024 cpus

Yale Omega Cluster
5632 cpus
57.8 Linpack TeraFlops

Cray XT5 (2011)
112,800 cpus
919 Linpack TeraFlops

Top 500 Historical Performance Development



Source: www.top500.org

High Performance Computing

- The Tianhe-2 Computer:
  - Fastest current supercomputer
    - Uses Ivy Bridge Intel Xeons & Intel Phi
    - 3,120,000 cpus
    - 1 Petabyte of memory
    - 33.9 Petaflops
    - 17.8 Megawatts of power
  - Located at National University of Defense Technology in China

- Why?
  - Research: Not just physical/life sciences and engineering any more! Increasingly data analysis/modeling ("Big Data"), especially in the social sciences.
  - Applications: Important "real-world" applications (and not just the obvious ones). Examples range from weather forecasting, to financial modeling, to Internet search and data analysis.

Agenda

- What is HPC?
  - Background & History
- **HPC at Yale**
  - **On-Campus HPC & Data Storage Facilities**
  - **HPC Staff**
  - **Getting Assistance with HPC at Yale**
  - **National HPC Infrastructure**
- Practicalities
  - Running Jobs on a Cluster (Interactive & Batch)
  - Requesting Resources
  - Module System
  - Screen
  - Pitfalls and Tips
  - Simple Queue

Yale HPC Facilities

- Two “Virtual” HPC Centers
  - Two physical locations: 300 George St. and West Campus Collections Building (aka A21)
  - **FAS HPC Core**: Serves main-campus science & engineering depts., including social sciences and non-NIH life sciences research
    - As of mid-2014 will run 2 clusters (Omega & Grace) at A21
    - Funded by a combination of university funds (mainly) & grant funds
    - General purpose facility for all FAS faculty
  - **Keck Biomedical HPC Center**: Serves NIH-funded biomedical researchers, both in Yale School of Medicine & FAS
    - Currently operates 2 primary clusters (BulldogN at A21; Louise at 300G);
    - BulldogN is dedicated to the Yale Center for Genome Analysis (YCGA)
    - Funded mainly by NIH grants and university funds
    - Close relationship with biomedical research centers on West Campus

FAS HPC Center Facilities (as of mid-2014)

Processing Power

Cluster	Date	Processors (Intel Xeon)	Nodes	Cores	Mem/Node	Users
Omega	2010	2.26 GHz E5520	256	2,048	48 GB	FAS
	2011	2.8 GHz X5560	704	5,632	36/48 GB	FAS & Condos
	2013	2.0 GHz E5-2620	64	768	64 GB	ESI
	2013	2x2.0 GHz E5-2650, 3 GPUs	4	64+GPUs	128 GB	FAS
Grace	2014	2.2 GHz E5-2660v2	72	1,440	128 GB	FAS & Condos

Network/Storage Space

Cluster	Primary Networks	Storage
Omega	QDR InfiniBand	1.4 PB Lustre (local) & 1 PB GPFS (shared)
Grace	FDR InfiniBand	1 PB GPFS (shared)

Keck Biomedical HPC Center Facilities

Processing Power

Cluster	Nodes (Intel Xeon)	Cores	Mem/Node	Users
BulldogN	13 (4x16x2.3GHz)	832	512 GB	YCGA Only (Sequencer Data)
	8 (4x12x2.3GHz)	384	128 GB	
	16 (2x8x2.60GHz)	256	128 GB	
	112 (2x4x2.27GHz)	896	48 GB	
Louise	16 (4x16x2.3GHz)	1024	512 GB	"Condo" and General Purpose for Life Sciences Research
	45 (2x8x2.60GHz)	720	128 GB	
	160 (2x4x2.33GHz)	1280	16-48 GB	
	110 (2x2x3.00GHz)	440	16 GB	

Network/Storage Space

Cluster	Primary Networks	Storage
BulldogN	1 or 10 Gigabit Ethernet	~3 PB (networked)
Louise	1 or 10 Gigabit Ethernet	

Yale Omega Cluster

Yale Omega Cluster (Custom Yale-Designed Pod)

HPC Staff

- System Administration Staff
  - 6 ITS staff based at West Campus who administer all HPC clusters
  - Hardware, OS-level software, and related user support
- User Support Staff (FAS HPC Center)
  - Application-level software and user support

Andy Sherman	**Steve Weston**
andrew.sherman@yale.edu	stephen.weston@yale.edu
Office: AKW 104 (51 Prospect St.)	Office: AKW 105 (51 Prospect St.)
Phone: 436-9171	Phone: 432-1236

- User Support Staff (Keck Biomedical HPC Center & YCGA)
  - **Nick Carriero**, **Rob Bjornson** (both based at 51 Prospect St.)
- ITS Research Technologies
  - New unit of ITS focused on support for research computing
  - Includes staff for HPC system admin, storage, and services
  - Part of Academic IT Services

Assistance with HPC at Yale

- HPC Info
  - Basic Info: http://its.yale.edu/services/research-technologies/high-performance-computing
  - Wiki: http://research.yale.edu/hpc/
- FAS Getting Started Page
  - http://hpc.research.yale.edu/wiki/index.php/Getting_started
- For account information, help using the clusters, or to provide feedback, contact us by email at hpc@yale.edu.
- Getting Started &Training Opportunities
  - Occasional “Getting Started” classes: general or on-demand
  - Parallel Programming: CPSC 424/524 (alternate years)
  - Other short courses will be coming

One Goal: Continuous HPC User Experience

National HPC Infrastructure

Yale HPC Centers

Desktops & Labs

National HPC Infrastructure (NSF)

- XSEDE (Xtreme Science & Engineering Discovery Environment)
  - Formerly TeraGrid - http://www.xsede.org
  - Large scale HPC resources for computing and visualization. E.g.:
    - NICS (ORNL-UTK): Kraken-XT5 (1.2 PF); Keeneland -GPU(615 TF)
    - TACC (UT Austin): Stampede (6 PF); Lonestar4 (302 TF)
    - SDSC (UCSD): Gordon (341 TF); Trestles (100 TF)
    - PSC: Blacklight (36 TF SMP)
  - Funded by NSF, but open to all
  - Allocations available for faculty-led research groups, but annual proposals are required.
    - Startup: Nearly automatic; for experimentation & development
    - Education: Academic or training classes (fixed dates)
    - Research: Competitive proposals, reviewed quarterly
  - Yale Campus Champions: Andy Sherman, Brian Dobbins
    - Advice about application process
    - Small, pre-startup testing allocations

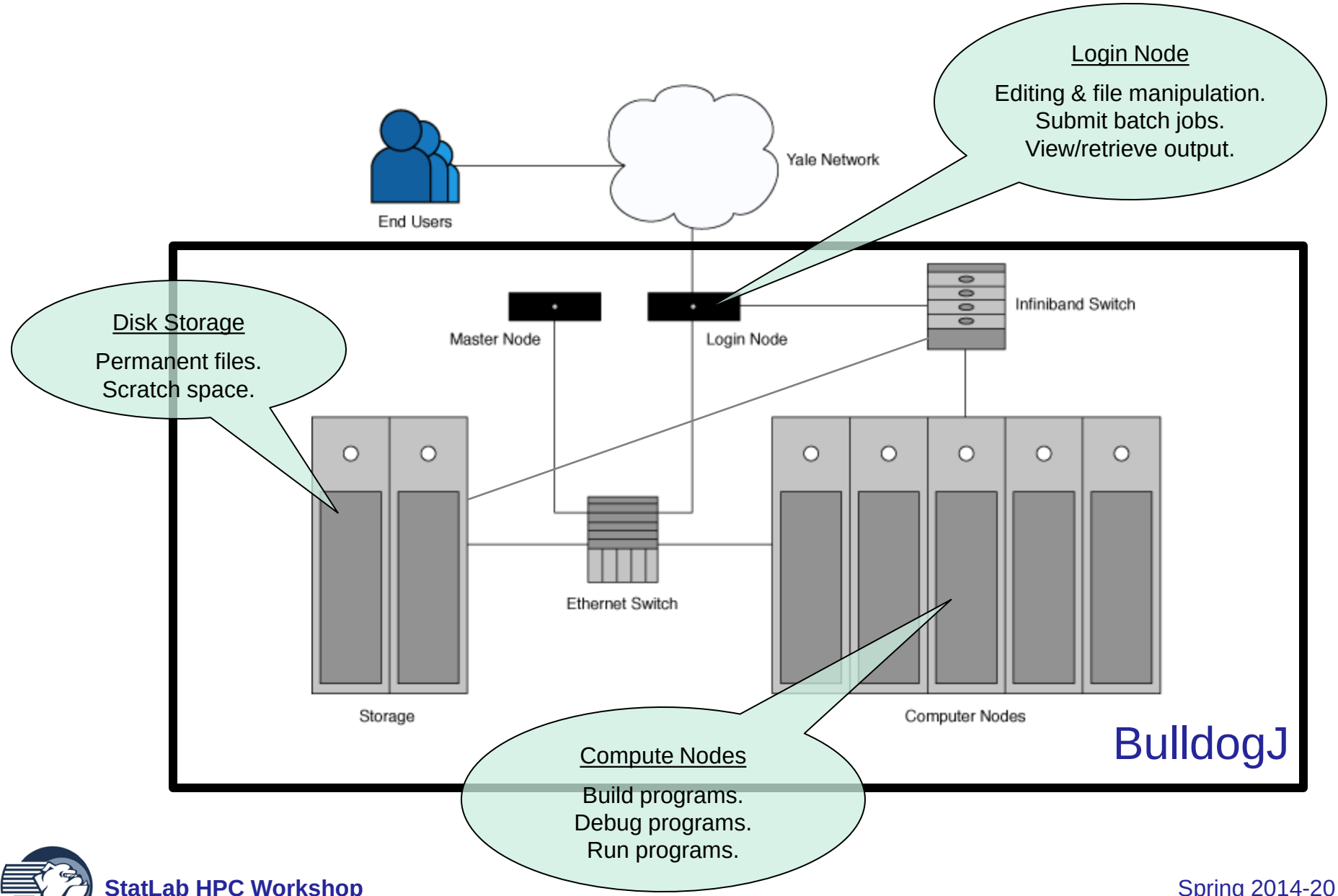
National HPC Infrastructure (DOE)

- DOE INCITE Facilities
  - 4.7 Billion of processor hours by competitive annual awards
  - Open to all researchers; only for large, highly-parallel computations
  - http://hpc.science.doe.gov/allocations/calls/incite2014
  - Oak Ridge Leadership Computing Facility (OLCF at ORNL)
    - http://www.nccs.gov
    - Titan (Cray XK7; 27 PF)
  - Argonne Leadership Computing Facility (ALCF at ANL)
    - http://www.alcf.anl.gov
    - Mira (IBM Blue Gene/Q; 10 PF); Intrepid (IBM Blue Gene/P; 557 TF)
- Other DOE Facilities
  - National Energy Research Supercomputer Center (NERSC at UCB)
    - http://www.nersc.gov
    - Edison (Cray XC30; 2.4 PF); Hopper (Cray XE6 cluster; 1.3 PF)

Agenda

- What is HPC?
  - Background & History
- HPC at Yale
  - On-Campus HPC & Data Storage Facilities
  - HPC Staff
  - Getting Assistance with HPC at Yale
  - National HPC Infrastructure
- **Practicalities**
  - **Running Jobs on a Cluster (Interactive & Batch)**
  - **Requesting Resources**
  - **Module System**
  - **Pitfalls and Tips**
  - **Simple Queue**

What's in a Cluster?



Running Jobs on a Cluster

There are two ways to run jobs: **interactively** or in **batch** mode:

Uses for Interactive Mode:

- Setting up data files
- Testing scripts on smaller problems
- Installing packages/modules
- Compiling/building/debugging programs

Uses for Batch Mode:

- Running your real work

Basic Steps for Running an Interactive Job

Run on your desktop

- Copy scripts and data to a cluster login node
  - scp or rsync are good choices from Linux or Mac machines

```
$ rsync -azv ~/workdir netid@omega.hpc.yale.edu:
```

  - WinSCP for Windows (available from Yale Software Page)
- ssh to a login node

```
$ ssh -Y -A netid@omega.hpc.yale.edu
```

Run on the login node

- Allocate a compute node

```
$ qsub -I -X -l nodes=1:ppn=1,mem=4gb,walltime=60:00 -q fas_devel
```

Run on the compute node

- Change to appropriate directory

```
$ cd ~/workdir
```

- Load module file(s)

```
$ module load Applications/R/2.15.3
```

- Run your script

```
$ R --slave -f compute.R
```

Copy scripts and data to the cluster

On Linux and Mac OS/X, use the standard **scp** and **rsync** commands to copy scripts and data files from your local machine to the cluster, or more specifically, to the login node. The **rsync** command is particularly useful if you have large files, some of which change occasionally. **rsync** tries to minimize the amount of data that is transferred, only copying files that have changed.

```
$ scp -r ~/workdir netid@omega.hpc.yale.edu:
$ rsync -azv ~/workdir netid@omega.hpc.yale.edu:
```

On Windows, the **WinSCP** application is available from the Yale Software Library:

http://software.yale.edu/Library/Windows/DT-WinSCP-4.2.9-W-D

Another option is "**Bitvise ssh**" which is free for individual use. Visit:

http://www.bitvise.com/ssh-client-download

Ssh to a login node

You must first login to a cluster login node using **ssh**. From a Mac or Linux machine, you simply use the **ssh** command:

```
$  ssh netid@omega.hpc.yale.edu
$  ssh -Y netid@omega.hpc.yale.edu
```

From a Windows machine, you can choose from programs such as **PuTTY** or **XWin32**, both of which are available from the Yale Software Library:

http://software.yale.edu/Library/Windows/DT-PuTTY-PuTTY-.6-W-P

http://software.yale.edu/Library/Windows/DT-Starnet-XWin2012-w

For more information on using PuTTY, search for ”create ssh key” at

https://hpc.research.yale.edu

XWin32 (Windows) and FastX or Xquartz (Macs) are “X Windows Servers” that can display graphics

(XWin32 and FastX are in the ITS software library; Xquartz is free at xquartz.macosforge.org.)

Allocating a Compute Node

- Allocate a single node using qsub:

```
$ qsub -I -l nodes=1:ppn=8,mem=34gb,walltime=60:00 -q fas_normal
```

- When requesting a compute node, it is very important to specify the computing resources that are needed via qsub's -l option.

- For a sequential (i.e., non-parallel) job, the most important resources are **walltime** and **mem**. It is best to always specify these resources, since they are also limits, and your job will be killed if it exceeds them.
  - **walltime**: Maximum amount of wall-clock time for job (dd:hh:mm:ss)
  - **mem**: Total amount of memory needed by job ("mb" or "gb")

Module Loads for Scripting Languages

Here are some “module load” commands for scripting languages on Omega:

```
$ module  load  Langs/Python/2.7.5
$ module  load  Langs/Perl/5.14.2
$ module  load  Apps/R/2.15.3
$ module  load  Apps/Matlab/R2012b
$ module  load  Apps/Mathematica/9.0.1
```

Run Your Script

When you're finally ready to run your script, you may have some trouble determining the correct command line, especially if you want to pass arguments to the script. Here are some examples:

```
$ python compute.py input.dat
$ R --slave -f compute.R --args input.dat
$ matlab -nodisplay -nosplash -nojvm < compute.m
$ math -script compute.m
$ MathematicaScript -script compute.m input.dat
```

You often can get help from the command itself using:

```
$ matlab -help
$ python -h
$ R --help
```

Notes on Interactive Mode

Enable X11 forwarding from ssh and qsub:

```
$  ssh  -Y  netid@omega.hpc.yale.edu
$  qsub  -X  -I  -q  fas_devel  -l  mem=34gb
```

Faster alternative:

```
$  ssh netid@omega.hpc.yale.edu
$  qsub -I -q fas_devel -l mem=34gb
```

Once job is running, execute from a different terminal window:

```
$  ssh -Y netid@omega.hpc.yale.edu
$  ssh -Y compute-XX-YY
```

where "**compute-XX-YY**" is the node allocated to your job.

Basic Steps for Running a Batch Job

Run on your desktop

- Copy scripts and data to a cluster login node
  - scp or rsync are good choices from Linux or Mac machines

```
$ rsync -azv ~/workdir netid@omega.hpc.yale.edu:
```

  - WinSCP for Windows (available from Yale Software Page)
- ssh to a login node

```
$ ssh -Y -A netid@omega.hpc.yale.edu
```

Run on the login node

- Submit a job script

```
$ qsub batch.sh
```

Components of a Torque/PBS Batch Script

- Heading

```
#!/bin/bash
```

- Instructions to the queuing system (Torque/Moab)

```
#PBS -l nodes=1:ppn=1,mem=4gb,walltime=1:00:00
#PBS -q fas_devel
```

- Change to appropriate directory

```
cd $PBS_O_WORKDIR
```

- Load module file(s)

```
module load Apps/R/2.15.3
```

- Run your script

```
R --slave -f compute.R
```

Batch Scripts

```
#!/bin/bash                          ..... Scripting language interpreter
#PBS -l nodes=4:ppn=8                ..... # nodes; # cpus per node
#PBS -l mem=60gb                     ..... Total memory (4x15gb)
#PBS -l walltime=24:00:00            ..... Wallclock time limit
#PBS -q fas_normal                   ..... Queue name
#PBS -j oe                           ..... Join stdout with stderr as stdout
#PBS -m abe -M your-email-here       .... Email notification
#PBS -N MyJob                        ..... Job name

cd $PBS_O_WORKDIR                    ..... cd to the submission directory

#Commands to run go here
pwd
module load Apps/Jaguar
time par_jaguar file.in
```

Standard Output files:	stdout:	**<job-name>.o<job-number>**
	stderr:	**<job-name>.e<job-number>**

These files are placed in the submission directory by default.

Checking on Batch Jobs

- qsub – Submit job for execution

```
$ qsub -q fas_devel –l nodes=1:ppn=8,mem=34gb,walltime=4:00:00 batch.sh
```

- qstat/showq – View status of jobs

```
$  qstat  -u  sw464
$  showq  -w  user=sw464
$  showq  -w  class=fas_normal
```

- showstart – Get estimate of when a job will start

```
$  showstart  1273896
```

- checkjob – Get information about a job

```
$  checkjob  -v  1273896
```

Requesting walltime with qsub

The setting of **walltime** is particularly important because if you ask for too little time, your job will be killed before it finishes, but if you ask for too much, it may not be scheduled to run for a long time.

- Specified as DD:HH:MM:SS (days, hours, minutes, seconds); Default is 1 hour
- Try to determine expected runtime during testing
- The following requests four hours of time:

```
$  qsub -I -q fas_devel -l nodes=1,mem=4gb,walltime=4:00:00
```

- If your job checkpoints itself periodically, then this decision is less critical, since you can always restart from a checkpoint

Requesting memory with qsub

mem is important because your job is likely to be killed if you exceed your request. But, don't ask for more than any node has, since your job won't be rejected, but will never run. (The scheduler is optimistic about future purchases. ☺) Omitting a **mem** request lets you use all of the memory on a node except for what the OS uses.

Rather than figuring out exactly how much you need, try the formula:

$$mem = numnodes * (mempernode - 1GB)$$

Key arguments here are:

- *numnodes* since **mem** is the **total memory** needed for the job
- *Mempernode* since the amount of memory per node varies

For example, if you want 10 36-gb nodes on Omega, you might request 350 GB of memory:

```
$  qsub –I –l nodes=10:ppn=8,mem=350gb -q fas_normal
```

Notes: Most queues use "whole node allocation", so you may as well request all of a node's resources. If you don't need all the resources on a node, then some queues like **fas_devel** will schedule multiple jobs per node if you don't request everything.

Specifying the Queue

You must always specify a queue via qsub's **-q** argument on the FAS clusters.

For most jobs, either desired priority or run time is the factor that discriminates among queues, though the walltime is also a factor, since you won't be able to submit jobs to a queue if you exceed the maximum allowable walltime.

fas_normal: normal priority; normal cost

fas_low: low priority; 10% of normal cost

fas_high: high priority; 10x normal cost

fas_bg: no priority (runs when nothing else is eligible); no cost

fas_devel: short jobs; limited number of cores; normal cost; dedicated nodes

fas_long: long jobs; normal priority; limited number of nodes available

fas_very_long: very long jobs; normal priority; limited number of nodes

Maximum Walltimes

Current Limits on FAS clusters:

fas_devel:	4 hours (4 nodes dedicated to queue)
fas_long:	3 days (no more than 32 nodes in total at once)
fas_very_long:	28 days (no more than 16 nodes in total at once; 8 per job)
All others:	24 hours (no more than 32 nodes per job)

Note that queue wait times tend to be rather long for **fas_long** and **fas_very_long**.

These are likely to change when Grace comes on line.

Software Environment

We manage the overall software environment system-wide so that:

(a) Software used by multiple users is installed in a central location.

(b) Software updates can be properly managed.

(c) Installed software can be optimized appropriately for the clusters.

Most non-system software is installed in “nonstandard” locations, mostly under the **/usr/local/cluster/hpc** directory. We use the **module system** to help users manage their environments for the software they use.

Module System - I

- Mechanism to manage environments for software packages
- Manual approach (complicated; error prone):
  - You need to know where each package is on the machine
  - You need to set all the environment variables for each package:
    - **`PATH`**: ordered search path of places to find executables
    - **`INCLUDE`**: one search path for include files
    - **`LD_LIBRARY_PATH`**: one search path for libraries
- Module allows you to set up your environment for the software you use. Well-written module files provide reasonable defaults and can be aware of other software that may loaded.
  - Examples
    - **`module load Compilers/Intel`** [loads a compiler]
    - **`module unload Compilers/Intel`** [unloads a compiler]
    - **`module list`** [lists loaded modules]
    - **`module display Compilers/Intel`** [shows module file content]

Module System - II

- Where module files live:
  - System module file directories (set up automatically during login)
  - **`~/privatemodules`** (for your own private modules)
  - Search path specified by MODULEPATH environment variable
- modulefind

  [**`/usr/local/cluster/hpc/Modules/modulefind`**]

  - Returns names of module files whose paths contain a given string (not case sensitive)
  - Useful w/o argument for finding what software is available on a cluster
  - Examples:
    - **`/usr/local/cluster/hpc/Modules/modulefind python`**
    - **`/usr/local/cluster/hpc/Modules/modulefind gcc`**
    - **`/usr/local/cluster/hpc/Modules/modulefind mpi`**

Compilers & MPI

- Compilers:
  - C/C++
    - "Standard System Compilers": Gnu v. 4.4.6
    - Better Gnu Compilers: Gnu v. 4.7.2 (modulefile: Langs/GCC)
    - Intel Compilers: Intel v.2013 and v.14 (modulefile: Langs/Intel)
    - Portland Group Compilers: v. 13.6 (modulefile: Langs/PGI)
  - Fortran (Fortran-77 and Fortran-9x)
    - "Standard System Compilers": **`f77`** (v. 3.4.6); gfortran (v. 4.4.6)
    - Better Gnu Fortran: **`gfortran`** (v. 4.7.2)
    - Intel Fortran: **`ifort`** (various v. 2013 and v.14 versions)
    - Portland Group Fortran: **`pgf77`**, **`pgf90 (v. 13.6)`**
  - Module files available to load full compiler suites (C/C++; Fortran)
    - Note: Intel & Gnu Fortrans are incompatible (naming; complex functions)
- MPI (Message Passing Interface; must select compilers first)
  - OpenMPI
  - Intel MPI

Building and Running Jobs

- Login to the cluster login node:
  - **`ssh netid@omega.hpc.yale.edu`**
  - Use **`qsub`** to start an interactive session; Build on the assigned node
- Set up your environment
  - Standard setup: in your **`.bashrc`** initialization file
  - Per-session setup: run module or use bash/c-shell commands
- Build your program
  - Use compilers directly (better: build a Makefile or use a software development environment)
  - **DO NOT BUILD ON LOGIN NODE** (software/library incompatibilities)
- Run your program
  - **NEVER on the login node!**
  - Use **`qsub`** to start an interactive session or submit a batch job

Managing Jobs in Torque/Moab Scheduler

- After submission, the Job Server assigns a number to your job:

```
[ahs3@login-0-1]$ qsub run_torque.sh
218439.omega-rocks.local
```

- For an estimate of when a job will run:
  - **`showstart <job-number>`**
- To kill a job
  - **`qdel <job-number>`**
- To see the status of your jobs or all jobs:
  - **`qstat [-f] <job-number>`** (-f gives details)
  - **`checkjob <job-number>`** (Alternative)
  - **`qstat –u <netid>`** (Jobs by user)
  - **`qstat –Q[f] [<queue-name>]`** (Jobs by queue)
  - **`qstat –a`** (All jobs)
  - **`qstat –a | grep " Q "`** (All Queued jobs)
  - **`qstat –a | grep " R "`** (All Running jobs)

Screen

- Creates a session on the cluster login node that persists
- Session can contain multiple shell instances (windows)
- You can disconnect and reconnect, even from different locations
- Very useful for interactive qsubs
- Also useful for saving context over time
- Uses ctrl-A as prefix to screen commands

Some screen commands

Key Seq	Command
^a c	New window
^a “	Show all windows
^a A	Name window
^a ^a	Go to last window
^a ?	Help
^a d	Detach
^a k	Kill current window
^a a	^a (e.g. for emacs)

Command line commands

Start new screen

$ screen

Attach to existing screen from new shell

$ screen -d -r

List screens

$ screen -list

Cluster Pitfalls: Resource Issues

- Underestimating Resources
  - Exceeding requested limits can get your job killed
  - Always specify mem and walltime
  - Determining memory requirements may be difficult
- Using too much memory
  - Best case: Your job is killed
  - Worst case: Your nodes crash, which may affect other jobs
- Avoiding long queue wait times
  - Asking for a lot of walltime or nodes can force your job to wait
  - Use **showstart** command to see how long you're likely to wait
  - Consider deleting (**qdel**) and resubmitting with smaller requests
  - Asking for less memory may also help in some circumstances

Cluster Pitfalls: Inefficiencies

- Submitting hundreds of small jobs
  - Puts a big load on the scheduler (queuing system), which slows the entire system down for everyone
  - Use **SimpleQueue** instead---that's what it's designed for
- Performing many small file operations
  - Puts a big load on file servers, which may slow everyone down
  - Use Unix pipes instead
    - Ex: "**`pgm | sort`**" instead of "**`pgm >file; sort < file`**"
  - Use /tmp when appropriate
    - 80 GB on most Omega nodes
    - Clean up when job is done!!!
  - Use /dev/shm when appropriate
    - In-memory file system that is much faster
    - Need to be careful about total memory usage

Programming/Scripting Tips for Clusters

- Develop and test on your desktop as much as possible
- Start testing on cluster with a smaller problem using fas_devel
- Use checkpointing if possible
- Use logging
- Try to reduce file I/O as much as possible
- Monitor the node during execution: **top**, **ps**, **pstree**
- If it's not as fast/efficient as you'd hoped, come see us

Automated Task Execution

- Appropriate for execution of a large number of independent tasks
  - Monte Carlo
  - Parameter sweeps
  - Independent data analyses
  - BLAST searches
- Approach 1: Automated job submissions
  - Creates large number of jobs, bogging down the system
  - May be inconvenient to keep track of so many jobs
  - For single-core tasks, may waste resources under Torque
- Approach 2: SimpleQueue
  - Submits 1 job that automatically schedules tasks on available resources
  - Keeps track of which tasks succeed, which fail, and which never run
  - Easy to restart to complete any leftover tasks
  - Designed for single-node tasks (1 core or multicore)
  - Module file: Tools/SimpleQueue/3.0

SimpleQueue Introduction

SimpleQueue is a tool that executes commands in parallel on a cluster. It can be an easy way to parallelize any kind of script without requiring you to learn about that scripting language's parallel programming tools.

- Simple parallel command execution
- Written at Yale (so we can support/enhance it!)
- Can be applied to any program that can be executed from the command line of your shell
- Similar to "GNU Parallel"

SimpleQueue Advantages

- Provides easy way to parallelize your script
- Makes checkpointing easy
- More flexible walltime requirements; avoids fas_very_long
- Easier to manage than submitting multiple batch jobs
- Integrated with Torque/Moab for ease of use
- Customized for FAS clusters (different from version on Louise/BDN)
- Much lower overhead per task compared to multiple qsubs
- Don’t need to know anything about MPI or parallel languages

SimpleQueue Notes

Steps to parallelize a script using SimpleQueue:

- If necessary, modify the script to compute a subset of job
- Create a task list containing commands to execute
- Create submit script using sqCreateScript command
- Submit the submit script

The lines in the task file often include commands that:

- Move to appropriate directory
- Load the module file(s)
- Run the script with appropriate arguments

Commands within a task should be separated by semi-colons. Tasks **must** be on a single line, although it can be a very long line.

SimpleQueue Example

Here's an example task file called **tasks.txt**:

```
cd ~/job; module load Apps/R; R --slave -f x.R --args i1.dat
cd ~/job; module load Apps/R; R --slave -f x.R --args i2.dat
cd ~/job; module load Langs/python; python x.py i3.dat > o3.dat
cd ~/job; module load Langs/python; python x.py i4.dat > o4.dat
[  rest  of  the  task  file  not  shown  ]
```

To run this in parallel on 4 nodes for 8 hours:

```
$ module load Tools/SimpleQueue/3.0
$ sqCreateScript –n 4 –w 8:00:00 tasks.txt > batch.sh
$ qsub batch.sh
```

Note: You may execute **`sqCreateScript`** on the login node.

Steps for Using SimpleQueue

- Create file of single-line tasks to run (e.g., **pp_sel_2_0_26.tasks**)

```
cd /home1/aeo2/jetHadron; ./run_dphi.sh ./Output/pp_ht_10_0.root 2 0 > ./log/pp_ht_sel_2_0.out
cd /home1/aeo2/jetHadron; ./run_dphi.sh ./Output/pp_ht_10_1.root 2 1 > ./log/pp_ht_sel_2_1.out
cd /home1/aeo2/jetHadron; ./run_dphi.sh ./Output/pp_ht_10_2.root 2 2 > ./log/pp_ht_sel_2_2.out
```

- Create a batch submission script

```
sqCreateScript -N au_sel_1_0_66 -n 2 -m 30 -p 7 -w 24:00:00 -q caines au_sel_1_0_66.tasks >job.sh
```

- Submit the script

```
qsub job.sh
```

- SimpleQueue task management output will appear in:

```
PBS_au_sel_1_0_66_out.txt     stdout
PBS_au_sel_1_0_66_err.txt     stderr
pp_sel_2_0_26.tasks.STATUS    data for each task run
pp_sel_2_0_26.tasks.REMAINING list of incomplete or failed tasks
pp_sel_2_0_26.tasks.ROGUES    list of unresponsive tasks
SQ_Files_<jobid>.master       additional detailed logging info
```

SimpleQueue Example: Original

```
#!/bin/csh
echo 'Do STAR Au+Au HT analysis'
echo ‘‘
set myShell =  '# ! /bin/csh' ;
set num=0;
set sel=$1;
set max=26;

while($num <= $max)
    echo "Analyze HT bin ${num}“
    set jobName = pp_sel_${sel}_${num}
    set outFile = ./log/pp_ht_sel_${sel}_${num}.out
    set fileName = ./Output/pp_ht_10_${num}.root
    echo "qsub -q caines -N ${jobName} -j oe -o ${outFile} submit_job.sh"

    echo  ${myShell} > ./submit_job.sh
    echo  cd /home1/aeo2/jetHadron >> ./submit_job.sh
    echo  ./run_dphi.sh ${fileName} ${sel} ${num} >> ./submit_job.sh

    qsub -q caines -l mem=2gb -l walltime=3:00:00 -N ${jobName} -j oe -o ${outFile} submit_job.sh
    rm ./submit_job.sh
    @ num++
    sleep 1
end
```

SimpleQueue Example: SimpleQueue Script - I

```
#!/bin/csh
echo 'Do STAR Au+Au HT analysis'
echo ''
set myShell =  '# ! /bin/csh' ;
set num=0;
set sel=$1;
set max=26;
set jobName = pp_sel_${sel}_${num}_${max}

# Create the task file

set taskFile = ${jobName}.tasks;
while($num <= $max)
    echo "Analyze HT bin ${num}"
    set outFile = ./log/pp_ht_sel_${sel}_${num}.out
    set fileName = ./Output/pp_ht_10_${num}.root
    echo "cd /home1/aeo2/jetHadron; ./run_dphi.sh ${fileName} ${sel} ${num} > ${outFile}"

    if ($num == 0) then
        echo  "cd /home1/aeo2/jetHadron; ./run_dphi.sh ${fileName} ${sel} ${num} > ${outFile}" > ${taskFile};
    else
        echo  "cd /home1/aeo2/jetHadron; ./run_dphi.sh ${fileName} ${sel} ${num} > ${outFile}" >> ${taskFile};
    endif
    @ num++
end
```

SimpleQueue Example: SimpleQueue Script - II

```
# Create the PBS Submission Script

set memnode = 15; # Use 47 for BulldogL, 35 for Omega
set memtask = 2; # Set to required memory per task

@ tpn = (${memnode} / ${memtask}); # Compute max number of tasks per node

# Case 1: Run all at once
@ nodes = ((${max} + ${tpn}) / ${tpn}); # Calculate number of nodes to run all tasks at once
@ mem = (${nodes} * ${memnode}); # Calculate aggregate memory for job
# Call sqCreateScript
sqCreateScript -N ${jobName} -n ${nodes} -m ${mem} -p ${tpn} -w 3:00:00 -q caines ${taskFile} > submit_job.sh

# Case 2: Run in 24 hours
#@ nodes=((((${max} + 9) / 8) + ${tpn} - 1) / ${tpn}); # Calculate number of nodes to run all tasks within 24 hours (3 hrs/task)
#@ mem=(${nodes} * ${memnode}); # Calculate aggregate memory for job
# Call sqCreateScript
#sqCreateScript -N ${jobName} -n ${nodes} -m ${mem} -p ${tpn} -w 3:00:00 -q caines ${taskFile} > submit_job.sh

qsub submit_job.sh
```