PracticalHPC

March 28, 2018

```
In [2]: %%html
        <style>
        table {float:left}
        </style>
```

```
<IPython.core.display.HTML object>
```

#
Practical HPC
##
Robert Bjornson
##
Yale Center for Research Computing
##
March 2018

0.1 Installing Anaconda (includes Jupyter notebook)

1. at www.continuum.io, download python 3.6 version of anaconda, and install

0.2 To get tutorial files

1. browse to https://github.com/ycrc/PracticalHPC
2. click "clone or download" then "download zip"
3. unzip file into desired location

0.3 To follow presentation in Jupyter notebook

1. in a terminal, cd to PracticalHPC
2. run this command: jupyter notebook PracticalHPC.ipynb

0.4 What is the Yale Center for Research Computing?

- Independent center under the Provost's office
- Created to support your research computing needs
- Focus is on high performance computing and storage
- ~15 staff, including applications specialists and system engineers

- Available to consult with and educate users
- Manage compute clusters and support users
- Located at 160 St. Ronan st, at the corner of Edwards and St. Ronan
- http://research.computing.yale.edu

0.5 Overview

- Understanding the HPC environment
- Understanding your program's behavior
- Improving performance
- Running parallel programs
- Creating parallel programs

1 Assumptions

- Have logged in to a cluster and run simple jobs
- understand basics of Slurm
- want to run more effectively and efficiently

2 The HPC Environment

2.1 What is HPC?

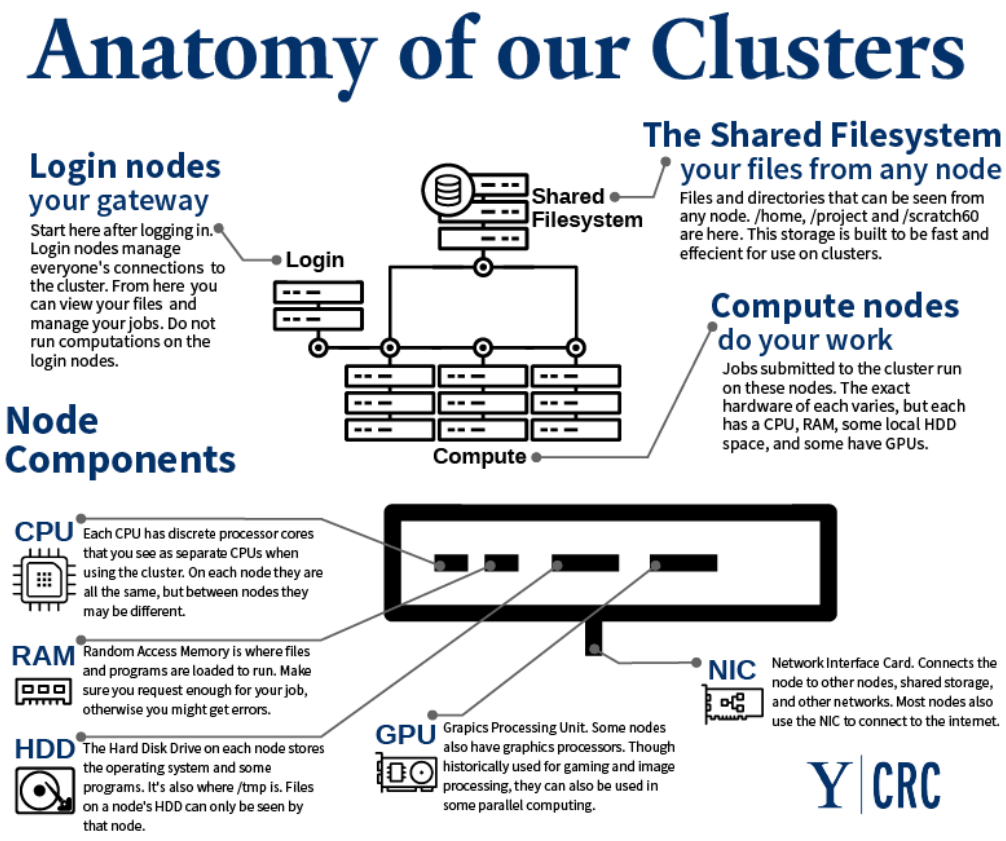
- running jobs remotely on a compute cluster
- linux-based
- using more compute resources (cpus/memory) than you have locally
- access to huge shared storage
- access to specialized hardware, e.g. GPUs
- access to specialized software
- running large job in parallel, or many sequential jobs

3 Typical Cluster

4 Typical compute node

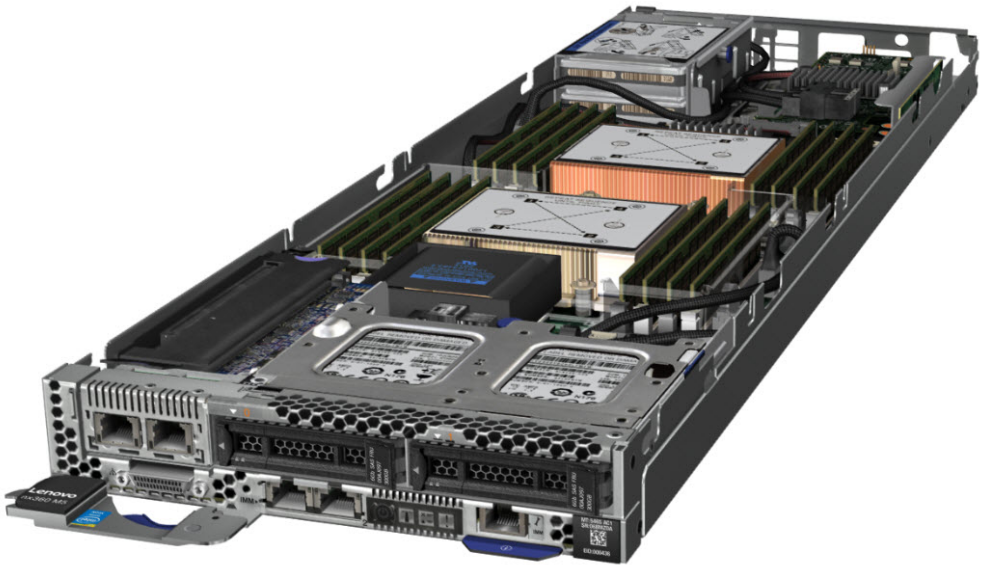
4.1 Typical Cluster (Yale, or elsewhere)

- Several hundred servers (nodes)
- Each node has 10s of cpus -> Thousands of total cpus
- Each node has 100s of GB RAM
- Multiple PB of shared storage
- Fast network connecting nodes and networks
- Shared by 100s of users, 1000s of jobs


Anatomy of our Clusters
Login nodes
your gateway
Start here after logging in. Login nodes manage everyone's connections to the cluster. From here you can view your files and manage your jobs. Do not run computations on the login nodes.
Login
Shared Filesystem
The Shared Filesystem
your files from any node
Files and directories that can be seen from any node. /home, /project and /scratch60 are here. This storage is built to be fast and effecient for use on clusters.
Compute nodes
do your work
Jobs submitted to the cluster run on these nodes. The exact hardware of each varies, but each has a CPU, RAM, some local HDD space, and some have GPUs.
Compute
Node Components
CPU
Each CPU has discrete processor cores that you see as separate CPUs when using the cluster. On each node they are all the same, but between nodes they may be different.
RAM
Random Access Memory is where files and programs are loaded to run. Make sure you request enough for your job, otherwise you might get errors.
HDD
The Hard Disk Drive on each node stores the operating system and some programs. It's also where /tmp is. Files on a node's HDD can only be seen by that node.
GPU
Grapics Processing Unit. Some nodes also have graphics processors. Though historically used for gaming and image processing, they can also be used in some parallel computing.
NIC
Network Interface Card. Connects the node to other nodes, shared storage, and other networks. Most nodes also use the NIC to connect to the internet.
Y | CRC

alt text

alt text



alt text

4.2 Cluster Resources

These are managed by slurm, cgroups, and storage quotas - Nodes and CPUs - Node Memory (RAM) - Job Runtime - Cluster Disk Storage - GPUs

4.2.1 Note

- jobs (unrelated) normally share nodes. Slurm+cgroups makes sure you have exclusive access to your resources

5 Quick Review of Slurm allocations

$ sbatch [options] batch.sh
$ srun –pty [options] bash

option	example	comment
-p *partition*	-p general	partitions(s) to run on
-c *cpus*	-c 20	cpus/task on single node
-n *tasks*	-n 4	mpi progs only
-N *nodes*	-N 10	forces layout, rarely useful
-t *time*	-t 7-, -t 3:00	job killed if exceeded
–mem=*mem*	–mem=16g	ditto
–mem-per-cpu=*mem*	–mem-per-cpu=8g	ditto
-J *name*	-J myjob	name job
–gres	–gres=gpu:p100:2	request gpus
–array *spec*	–array 1-10	array job

5.1 Fairshare Scheduling with backfill

- Slurm tracks cpu usage by user and group[5]
- Job priority is determined by amount of recent usage
- Users and Groups with heavy recent usage have lower priority
- Backfill can run short, lower priority jobs

5.3 Memory

- default of 5 GB per allocated cpu on our clusters
- strictly enforced; jobs exceeding limit are killed
- users have limits (e.g. 640 GB)
- you can request custom memory per node or core with sbatch or srun:

```
--mem=6g
--mem-per-cpu=6g
```

5.4 Finding compute resources

```
sinfo -p general
sinfo -p general -e -t IDLE -o "%P %.5a %c %m %.10l %.6D %.6t %N"
alias findidle='sinfo -e -t IDLE -o "%P %.5a %c %m %.10l %.6D %.6t %N"'
findidle -p general
```

5.5 Graphical Tools

http://overwatch.ycrc.yale.edu/farnam/orwell/
http://overwatch.ycrc.yale.edu/grace/orwell/

5.6 Scavenge Partition

```
$ sbatch -p scavenge
```

- Compute nodes in other partions are available via scavenge partition
- jobs subject to being killed at any time
- separate per user limits apply
- works best for short jobs, dSQ/array jobs, or jobs that checkpoint
- can include gpus

5.7 Special Nodes

5.8 Large Memory Nodes

- Compute nodes with 512GB-1.5TB of RAM
- Reserved for applications with large memory needs. Please be considerate.
- Separate slurm partition: bigmem

Typical allocation:

```
srun/sbatch -p bigmem --mem=1500g ...
```

5.9 GPU Nodes

- Some applications have been ported to GPUs with impressive performance improvement
- Gpu nodes have conventional cpus with multiple cores, and 1-4 GPUs.

- To use GPUs, you must:
- request node(s) with GPUs
- request the type and number of GPUs

Typical allocation:

```
srun --pty -p gpu -c 8 --gres=gpu:1080ti:4 ... bash
sbatch -p gpu -c 8 --gres=gpu:1080ti:4 ... batch.sh
```

5.10 GPU Nodes continued

You can also scavenge gpu nodes:

```
srun --pty -p scavenge -c 8 --gres=gpu:1080ti:4 ... bash
sbatch -p scavenge -c 8 --gres=gpu:1080ti:4 ... batch.sh
```

Note that partition names, types and number of GPUs vary by cluster.

Type	FP	Clusters
1080ti	single	farnam
K80	double	grace,farnam
P100	double	grace

6 Your Job's Behavior

6.1 What is "Running Efficiently"?

- All allocated cores are running near 100%
- GPUs (if allocated) are used
- memory requested is adequate and reasonable

6.2 Monitoring your job

```
squeue -u netid -l (your jobs)
squeue -j jobid -l (particular job)
scontrol show job jobid (during job)
sacct -j jobid -l (after job finishes)
```

Default output for squeue and sacct is not ideal. Put in .bashrc:

```
export SACCT_FORMAT="JobID%-20,JobName,User,Partition,NodeList,Elapsed,State,ExitCo
export SQUEUE_FORMAT="%.16i %.12P %.12j %.8u %.2t %.12M %.12l %24R %.4D %.4C %m %8
```

in PracticalHPC:
`source SLURM.env`

6.3 Checking cpu and memory utilization for running jobs

Using top/htop - use `squeue -u netid` to determine where your job is running - ssh to your nodes, especially 2nd and on, and run `top -u netid` (or `htop`) - look at %CPU and RES columns - should see ~%100 cpu for each allocated core

6.4 Postmortem checking

1. Run program under /usr/bin/time

```
$ /usr/bin/time -a myprog arg1 arg2 ...
```

2. After run completes, determine actual usage (See format comment):

```
$ sacct -j jobid
```

6.5 Example: Performance

```
cd Examples/Performance
sbatch bwa.sh # runs out of RAM (default 5GB/core)
sbatch --mem=10g bwa.sh # also runs out
sbatch -c 8 # runs ok.  Show htop, scontrol, sacct,
Htop: H combines threads.  F5 shows tree
sacct -j jobid
scontrol show job jobid
```

7 Data

7.1 Storage Quotas

- most quotas are group quotas
- if exceeded, jobs will die trying to write
- to see all quotas for your group:

```
$ groupquota
$ groupquota -u netid
```

7.2 File Storage

Location	Purpose	Typical Limit (size/files)	Backed Up?
home	scripts, programs, documents, small files	125GB/500K	Y
project	large data, programs	1-4TB/5M	N
scratch60	temp data	10TB/5M	purged
pi_*name*	dedicated	*varies*	N

7.3 File Transfer

- For small transfers:
- Linux/Mac/cygwin: scp, rsync
- Windows: Mobaxterm, winscp
- For larger transfers, or outside Yale, or collaborators without Yale credentials:
- Globus

7.4 Globus

- www.globus.org
- web and command line
- very fast and robust
- transfers between *endpoints*
- endpoints:
- Yale clusters: yale#{cluster}
- Lots of other institutions
- personal endpoint on your own computer
- you can create *shares*, accessible to non-Yale researchers via their globus id

7.5 Example

```
show transfer between farnam and ruddle
show personal endpoint (bjornson_mac)
show creating share and sharing with jason.ignatius@yale.edu
```

7.6 Globus cli

```
$ module load Globus-CLI
$ globus login
(you'll get a url.  Open that in a browser, authenticate, and copy the code.  Enter
$ globus endpoint search bjornson_mac
$ globus endpoint search yale#farnam
(set env vars to BM and F)
$ globus ls -l $BM:/
$ globus transfer -r -s size $BM:Data $F:Data
(returns taskid)
$ globus task show taskid
```

https://docs.globus.org/cli/examples

8 Performance

- When possible, use optimized libraries, rather than roll your own
- Python: numpy or pandas instead of nested lists for matrices
- R, Matlab: vectors instead of loops

8.1 Profiling

- Helps you pinpoint where time is spent
- R: Rprof
- Python: cProfile, kernprof/line_profiler
- C: gprof

8.2 Python: cProfile for function level profiling

```
cd Examples/Profiling
module load Python
(get compute node with srun, module load Python)
python -m cProfile -o bad.prof bad.py
(takes a minute or two)

(in python)
import pstats
p=pstats.Stats('bad.prof')
p.sort_stats('time').print_stats()

or
```

8.3 kernprof for line-by-line

```
module load Python
(decorate functions with @profile)
kernprof -l bad2.py
python -m line_profiler  bad2.py.lprof
```

‘ “ rdb9@c13n08 Profiling]$ python -m line_profiler bad2.py.lprof Timer unit: 1e-06 s
Total time: 114.565 s File: bad.py Function: doit at line 4

9 Line # Hits Time Per Hit % Time Line Contents

```
 4                                           @profile
 5                                           def doit(gf, mf, of):
 6         1          134    134.0      0.0      ofp=open(of, "w")
 7         1            1      1.0      0.0      genenames=[]
 8         1            0      0.0      0.0      geneinfo=[]
 9     80923        44421      0.5      0.0      for l in open(gf):
10     80922       105965      1.3      0.1          gn, chrm, strand, start, end =
11     80922        44241      0.5      0.0          genenames.append(gn)
12     80922        49721      0.6      0.0          geneinfo.append([chrm, strand,
13
14     71521        90557      1.3      0.1      for l in open(mf):
15     71520       126260      1.8      0.1          genename=l.strip().split()[-1]
```

```
16
17     71520    110808166   1549.3     96.7          idx=genenames.index(genename)
18     71520      3295037     46.1      2.9          print(l + str(geneinfo[idx]),
```

'''

9.1 FIX: replace list with dictionary

```
kernprof -l good.py > /dev/null
python -m line_profiler good.py.lprof
```

10 Parallelism

- Sbatch can allocate multiple cores and nodes, but the script runs on one core on one node sequentially.
- *Simply allocating more nodes or cores DOES NOT make jobs faster.*
- How do we use multiple cores to increase speed?
- Two classes of parallelism:
- Lots of independent sequential jobs
- Single job parallelized (somehow, maybe by someone else?)

10.1 First, my "Disclaimer"

- Parallel computing is a complex field with many tools and techniques
- Entire courses: e.g. Yale CS 424/524 "Parallel Programming Techniques" by Dr. Andrew Sherman
- We'll see some simple techniques for easy cases

10.2 Slurm Job Arrays

- Useful when you have many nearly identical, independent jobs to run
- Starts many identical copies of your script, distinguished by $SLURM_ARRAY_TASK_ID

Submit jobs like this:

```
sbatch --array=1-100 script.sh
```

Inside your batch script this environment variable to do something different in each task:

```
./mycommand -i input.${SLURM_ARRAY_TASK_ID} \
    -o output.${SLURM_ARRAY_TASK_ID}
```

10.3 Slurm Job Arrays continued

A few nice features of job arrays: - only one job to keep track of - easy to start or cancel entire set - slurm options, e.g. cpus, memory, time limits, apply to each task, not overall job - your allocation can grow and shrink as conditions change - when using scavenge partition, tasks are killed, but job persists

10.4 Array job example

```
cd Examples/Array
sbatch array.sh
```

10.5 dSQ (aka Dead Simple Queue)

- local tool built on job arrays. Same nice features, but easier to use
- more flexible; tasks can be arbitrarily different
- reporting and error recovery built in

10.6 Using dSQ

- Create file containing list of commands to run (jobs.txt)

  ```
  prog arg1 arg2 -o job1.out
  prog arg1 arg2 -o job2.out
  ...
  ```

- Create batch script

  ```
  module load dSQ
  dSQ --jobfile jobs.txt [slurm args] > run.sh
  ```

  slurm args can specify partion, timelimit, memory, etc. in the usual way.

- Submit launch script

  ```
  sbatch run.sh
  ```

For more info, see http://research.computing.yale.edu/support/hpc/user-guide/dead-simple-queue

11 dSQ Reporting

- When dSQ job is finished, you'll see a file `job_<jobid>_status.tsv`
- Generate report:

```
$ dSQAutopsy jobs.txt job_<jobid>_status.tsv > failedjobs.txt
Autopsy Task Report:
9 succeeded
1 failed
0 didn't run.
```

- If any jobs failed, failedjobs.txt will contain those jobs

11.1 dSQ Example

```
cd Examples/GenomePL
python mktasks.py data output > tasks.sh # create dSQ job file
module load dSQ
dSQ --jobfile tasks.sh > run.sh
sbatch run.sh # will fail, do sacct -j jobid, resubmit --mem=10g
```

12 Running already parallelized applications

12.1 Parallel bootstrap in R using Multicore library

- works on multiple cpus on one node

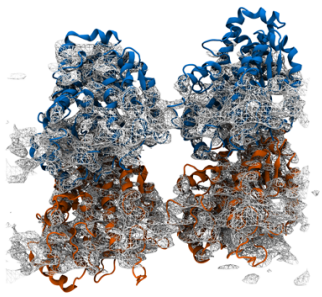
```
boot(data=trees, statistic=volume_estimate, R=50000, parallel="multicore", ncpus=8)
```

```
cd Examples/R-Bootstrap
sbatch -c 8 ... script.sh
```

12.2 Parallel Namd

- molecular dynamics simulation written in C and C++
- Namd is parallelized several ways, which can be combined:
- OpenMP (within one node)
- MPI (across nodes)
- GPU

Example: Satellite Tobacco Mosaic Virus 1,066,628 atoms, 500 time steps



alt text

13 Namd Performance on Grace

startup (s)	simulate	total	step	step s/u	
1 node, 1 cpu	14	5833	5847	11.6	1.0
1 node, 28 cpu	13	218	231	0.44	26.3
1 node, 20 cpu+4 K80 gpus	12	32	44	.056	207
9 nodes, 40 cpus	29.5	212	242	.424	27.4

in /home/fas/lsprog/rdb9/repos/PracticalHPC/Examples/Namd/stmv - gpus slurm-9139008.out - mpi slurm-9152709.out - 28 core MP slurm-9139007.out - 1 core slurm-9140416.out

13.1 Namd Example

```
cd Examples/Namd/stmv
sbatch multi.sh
sbatch gpu.sh
use htop and nvidia-smi on allocated node
```

14 DIY Parallelism

14.1 Parallel Kmeans in R using parallel

- simple idea: reduce computation to applying function to list of values (very R-ish)
- then do that in parallel

sequential

```
starts=100000; tasks=8; nstarts=rep(starts/tasks, tasks)
results<-lapply(nstarts, function(nstart) kmeans(Boston, 4, nstart=nstart)))
```

parallel

```
library(parallel)
...
results<-mclapply(nstarts,
  function(nstart) kmeans(Boston, 4, nstart=nstart),
  mc.cores=cores)
```

14.2 Example Kmeans

```
cd R-Kmeans
sbatch kmeans.sh
sbatch parkmeans.sh
```

14.3 kmeans performance on farnam m610s

runtime	s/u	
1 node, 1 cpu	44.475	-
1 node, 8 cpus	5.96	7.46

14.4 Other thoughts on parallel R

- *Parallel R*, by Steve Weston, O'Reilly Press
- YCRC Bootcamp: Writing Efficient R Code (taught by Steve)
- `foreach` parallel construct (written by Steve)

14.5 Python Multiprocessing

- Support for creating parallel processes and communicating and synchronizing
- Simplest interface is parallel map, very similar to R's mclapply

```
import multiprocessing
p=multiprocessing.Pool(10)
inputs=[...]
results=p.map(f, inputs)
```

- Pool creates set of "workers" that compute f() on inputs
- Workers are forked processes. Have copies of data at point where pool was created
- Function, inputs, and outputs are serialized and passed between master and workers

14.6 Parallel Travelling salesman in Python using multiprocessing

- Given N cities, find shortest route visiting each once
- NP-complete -> very expensive to find optimal solution
- We'll use a heuristic approach, simulated annealing:

```
T = initialT
curr = best = initialsolution
while T > stopT:
    candidate = randomly swap two adjacent cities in curr
    if cost(candidate) < cost(best):
        curr=candidate
        best=candidate
    elif cost(candidate)-cost(best) < T
        curr=candidate
    T = T * alpha
```

14.7 Parallelization strategy

- Run K copies, starting from different initial conditions
- Take different random paths
- Choose the best solution

Example/simulated-annealing-tsp

Steps: - seq.py: explicit for loop over simulations - map.py: convert loop to map - parmap.py: convert map to parallel map

14.8 parmap.py

```
def dotrial(t):
    t.anneal()
    return t

p=multiprocessing.Pool(cores)
trials = [SimAnneal(coords, ...) for i in range(ntrials)]
got=p.map(dotrial, trials, chunksize=5)
```

14.9 Example: TSP

```
cd Examples/TSP
srun -p interactive --pty bash
python seq.py 10
python map.py 10
srun -c 10 -p interactive --pty bash
python parmap.py 100
```

14.10 Subtleties WRT random numbers

- When running in parallel, what random number sequence is seen by each process?
- many random number generators use system clock, which may not be different for each process
- you may want reproduceable results
- R: use RNGkind("L'Ecuyer-CMRG"). Sets up independent, reproducible random streams
- Matlab and Python: Each worker has same seed by default. You should explicitly set it, e.g. rng(workerid)

14.11 C/C++: OpenMP

- popular way to parallelize C programs on single node
- annotate for loop with #pragma omp for

```
   #pragma omp for reduction(+:sum)   for (i=0; i<len*threads; i++)
{       sum += (a[i] * b[i]);       psum = sum;     }
```

14.12 Example: Matrix multiply with OpenMP

```
cd Examples/OpenMP
make mm
sbatch mm.sh
```

14.13 To get help or report problems

- Check our status page: http://research.computing.yale.edu/system-status
- Send an email to our tracking system: hpc@yale.edu
- Read documentation: http://research.computing.yale.edu/hpc-support
- Office hours: http://research.computing.yale.edu/hpc-support/office-hours-support
- Email me directly: Robert.bjornson@yale.edu