ada Cluster Tutorial

Frank Ferraro
UMBC

Materials

Code: https://github.com/fmof/ada-tutorial.git

tl;dr: if you’re comfortable with taki (or after this session)

- taki group storage is available, but you won’t be able to use the conda environments I’ve built. You’ll need to make your own.
- Currently, only one partition and no QOSes.
- Required flags for sbatch or srun
  - --gres=gpu:<num>
    - <num> is int specifying the *number* (not IDs) of GPUs your job needs
  - --time=<wallclock-time>
    - Your job will be killed after <wallclock-time>
  - --mem=<mem-required-in-MB>
    - Your job will be killed if it uses more than <mem> amount of RAM.
- Optional
  - Asking for specific cards: --constraint=<feature>
    - Your job needs nodes that have certain <feature>s
  - Specify your PI group: --account=pi_<your-pi>

General Coding Approach

Task Outline

1. Write your code
2. Perform small-scale testing
3. Perform small-scale testing on the grid (at command line; synchronous)
4. Run the code on the grid at the scale needed (batch; asynchronous)

Outline

- Grid Basics
  - What is a grid? Compute+storage+management
  - High-level: How to use a grid
- Submitting jobs
  - Testing → Submitting “real” jobs
  - Managing jobs
- Requesting resources (gotchas)
  - GPUs
  - Memory
  - Time limits
  - Features

How to Ask for Help

nicely :)

1. Read the error (if any) carefully
2. Check your
   a. Paths (to code, input files, output files)
   b. Missing modules (in your submission script)
   c. Check your resources (# CPUs, # nodes, amount of memory, run time, etc.)
3. Read the man pages
4. Do a quick Google search
5. File a ticket: URL TBD
   a. If you’re working with me (Frank), cc me on all tickets
   b. Currently, contact your PI

What is an HPCC/HPCF?

HPCC/F: High Performance Compute Cluster/Facility

Large collection of connected computers for running experiments (code)

- compute servers (nodes), each often with
  - Many CPUs
  - A lot of RAM
  - Some/many GPUs
- storage (most of the time)
  - Combination of backed-up and non-backed up storage
  - Often *networked*

What is an HPCC/HPCF?

HPCC/F: High Performance Compute Cluster/Facility

Large collection of connected computers for running experiments (code)

- compute servers (nodes), each often with
  - Many CPUs
  - A lot of RAM
  - Some/many GPUs
- storage (most of the time)
  - Combination of backed-up and non-backed up storage
  - Often *networked*

Central scheduling service (job manager) controls when jobs run and on what nodes

What is an HPCC/HPCF?

HPCC/F: High Performance Compute Cluster/Facility

Large collection of connected computers for running experiments (code)

- compute servers (nodes), each often with
  - Many CPUs
  - A lot of RAM
  - Some/many GPUs
- storage (most of the time)
  - Combination of backed-up and non-backed up storage
  - Often *networked*

Each node can access the same files

What is an HPCC/HPCF?

HPCC/F: High Performance Compute Cluster/Facility

Large collection of connected ~~computers for running experiments~~ (code)

Each node can talk to the others

- compute servers (nodes), each often with
  - Many CPUs
  - A lot of RAM
  - Some/many GPUs
- storage (most of the time)
  - Combination of backed-up and non-backed up storage
  - Often *networked*

What is an HPCC/HPCF?

HPCC/F: High Performance Compute Cluster/Facility

Large collection of connected ~~computers for running experiments~~ (code)

- compute servers (nodes), each often with
  - Many CPUs
  - A lot of RAM
  - Some/many GPUs
- storage (most of the time)
  - Combination of backed-up and non-backed up storage
  - Often *networked*

Each node can talk to the others

Each node can access the same files

Central scheduling service (job manager) controls when jobs run and on what nodes

Thinking for a Grid

Single Workstation Workflow

all-in-one:code + data + analysis + computer + writing

Thinking for a Grid

Single Workstation Workflow

all-in-one:code + data + analysis + computer + writing

Pros:

- What you're familiar with
- Can be easy to debug

Cons:

- Bandwidth limited
- Non-dedicated, consumer-grade
- Serial thinking

Overall cost: TIME!

Thinking for a Grid

Single Workstation Workflow

all-in-one:code + data + analysis + computer + writing

Pros:

- What you're familiar with
- Can be easy to debug

Cons:

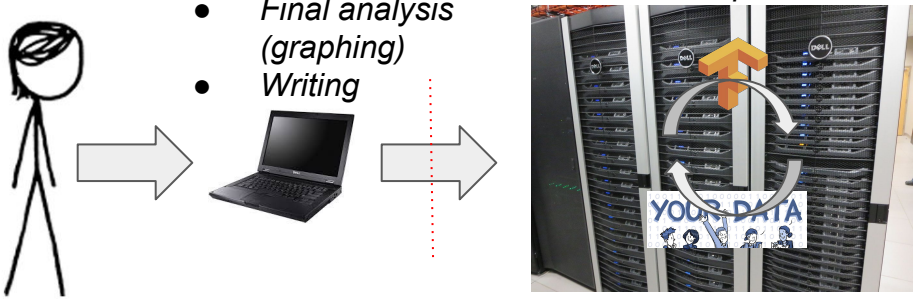
- Bandwidth limited
- Non-dedicated, consumer-grade
- Serial thinking

Overall cost: TIME!

Grid Workflow

- *Initial dev*
- *Final analysis (graphing)*
- *Writing*

code + data + analysis + computer



Thinking for a Grid

Single Workstation Workflow

all-in-one:code + data + analysis + computer + writing

Pros:

- What you're familiar with
- Can be easy to debug

Cons:

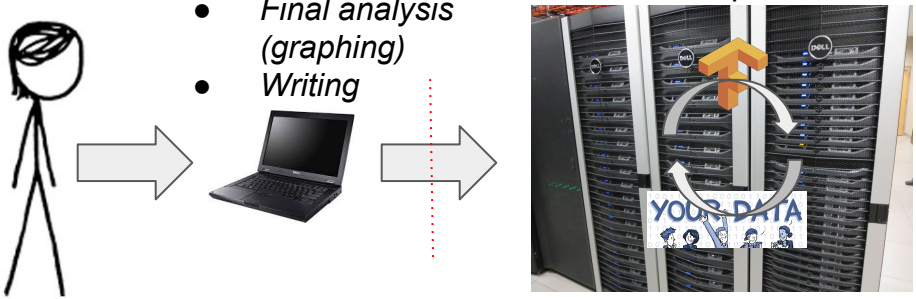
- Bandwidth limited
- Non-dedicated, consumer-grade
- Serial thinking

Overall cost: TIME!

Grid Workflow

- *Initial dev*
- *Final analysis (graphing)*
- *Writing*

code + data + analysis + computer



Pros:

- Dedicated, enterprise-grade hardware
- Many more (& powerful) computers than your laptop
- (Less) bandwidth limited

Cons:

- Learning curve
- Can be harder to debug
- Shared machines
- *You* don't control the machines

Login vs. Compute Nodes

login node

- Access via: `ssh ada[.rs.umbc.edu]`
- Submit jobs from here
- Do NOT run code on this

13 compute nodes

- Access via the grid engine (SLURM), *not* through SSH
- “Run” code on these

“Nodes” vs. “CPUs”

Node: a single server (motherboard)

- Nodes can have many CPUs

“CPU”: a virtual core

- Assume one active process per CPU

Memory on node accessible across all CPUs, but MUST be reserved in advance

“Nodes” vs. “CPUs”

Node: a single server (motherboard)

- Nodes can have many CPUs

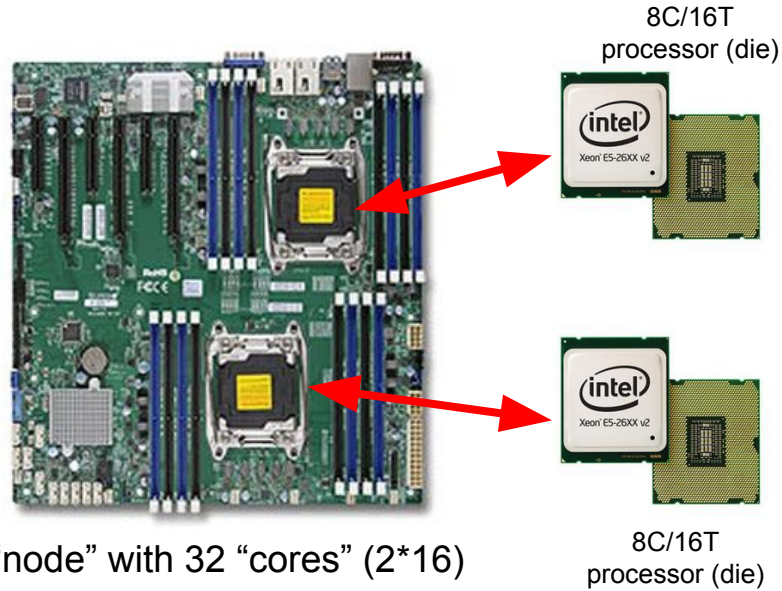
“CPU”: a virtual core

- Assume one active process per CPU

Memory on node accessible across all CPUs, but MUST be reserved in advance

Terminology: Processor vs. CPU

Technically, a processor has many cores, and a node has many processors. Often, the distinction among the different processors, and between processors and cores, is NOT important.



“node” with 32 “cores” (2*16)

Storage on ada

	Path	Properties	How to use	
			Do:	Don’t:
home	• `/home/${USER_ID}` `${HOME}`	• Networked storage • Backed up	• Store code, environment config files • (tip) symlink dot directories to your user workspace	• Store data • Store output files

Storage on ada

	Path	Properties	How to use	
			Do:	Don't:
home	• `/home/${USER_ID}` `${HOME}`	• Networked storage • Backed up	• Store code, environment config files • (tip) symlink dot directories to your user workspace	• Store data • Store output files
HPCF user workspace	• `~/${PI}_user` → `/umbc/xfs1/${PI}/user/${USER_ID}`	Base: `/umbc/xfs1/${PI}` • Networked storage • Not backed up	• Store *your specific* data • Your model files • Experimental output	• Store group relevant code, data • Store anything critical
HPCF group workspace	• `~/${PI}_common` → `/umbc/xfs1/${PI}/common`		• Store group (shared) relevant code, data	• Store items for collaborators • Store anything critical

Storage on ada

	Path	Properties	How to use	
			Do:	Don't:
home	• `/home/${USER_ID}` `${HOME}`	• Networked storage • Backed up	• Store code, environment config files • (tip) symlink dot directories to your user workspace	• Store data • Store output files
HPCF user workspace	• `~/${PI}_user` → `/umbc/xfs1/${PI}/user/${USER_ID}`	Base: `/umbc/xfs1/${PI}` • Networked storage • Not backed up	• Store *your specific* data • Your model files • Experimental output	• Store group relevant code, data • Store anything critical
HPCF group workspace	• `~/${PI}_common` → `/umbc/xfs1/${PI}/common`		• Store group (shared) relevant code, data	• Store items for collaborators • Store anything critical
ada-specific storage	`TBD`	• Additional ~180TB Networked storage • Not backed up	• Store group (shared) relevant code, data	• Store items that collabora • Store anything critical

Storage on ada

	Path	Properties	How to use	
			Do:	Don't:
home	• `/home/${USER_ID}` `${HOME}`	• Networked storage • Backed up	• Store code, environment config files • (tip) symlink dot directories to your user workspace	• Store data • Store output files
HPCF user workspace	• `~/${PI}_user` → `/umbc/xfs1/${PI}/user/${USER_ID}`	Base: `/umbc/xfs1/${PI}` • Networked storage • Not backed up	• Store *your specific* data • Your model files • Experimental output	• Store group relevant code, data • Store anything critical
HPCF group workspace	• `~/${PI}_common` → `/umbc/xfs1/${PI}/common`		• Store group (shared) relevant code, data	• Store items for collaborators • Store anything critical
ada-specific storage	`TBD`	• Additional ~180TB Networked storage • Not backed up	• Store group (shared) relevant code, data	• Store items that collabora • Store anything critical
scratch	`/scratch/${SLURM_JOB_ID}`	• Node-specific • Automatically **created** & **deleted** for each job	• Use for job-specific **intermediate** output files	• Store anything you don't need at the end of a job

Storage on ada

	Path	Properties	How to use	
			Do:	Don't:
home	• `/home/${USER_ID}` `${HOME}`	• Networked storage • Backed up	• Store code, environment config files • (tip) symlink dot directories to your user workspace	• Store data • Store output files
HPCF user workspace	• `~/${PI}_user` → `/umbc/xfs1/${PI}/user/${USER_ID}`	Base: `/umbc/xfs1/${PI}` • Networked storage • Not backed up	• Store *your specific* data • Your model files • Experimental output	• Store group relevant code, data • Store anything critical
HPCF group workspace	• `~/${PI}_common` → `/umbc/xfs1/${PI}/common`		• Store group (shared) relevant code, data	• Store items for collaborators • Store anything critical
ada-specific storage	`TBD`	• Additional ~180TB Networked storage • Not backed up	• Store group (shared) relevant code, data	• Store items that collabora • Store anything critical
scratch	`/scratch/${SLURM_JOB_ID}`	• Node-specific • Automatically **created** & **deleted** for each job	• Use for job-specific **intermediate** output files	• Store anything you don't need at the end of a job
tmp	`/tmp`	• Node-specific	---	• Use, if at all possible

Storage on ada

	Path	Properties	How to use	
			Do:	Don't:
home	•			• Store data • Store output files
HPCF user workspace	• /umbc			• Store group relevant code, data • Store anything critical
HPCF group workspace	• /umbc			• Store items for collaborators • Store anything critical
ada-specific storage				• Store items that collabora • Store anything critical
scratch	/scrat			• Store anything you don't need at the end of a job
tmp	/tmp	• Node-specific	---	• Use, if at all possible

Always backup elsewhere!!!

Partitions vs. Quality of Service (QoS)

“Partitions” group certain nodes together

- Makes scheduling easier
- Clearly state what types of physical resources your jobs need (and what they don’t)
- Partitions do not have to be mutually exclusive

QoS

- High-level binning of how *many* of the different resources you can use

Use both partitions and QoS to effectively manage your jobs

Partitions vs. Quality of Service (QoS)

"Partitions" group certain nodes together

- Makes scheduling easier
- Clearly state what types of physical resources your jobs need (and what they don't)
- Partitions do not have to be mutually exclusive

QoS

- High-level binning of how *many* of the different resources you can use

Use both partitions and QoS to effectively manage your jobs

Partitions (Groupings) of Compute Nodes

	batch	**develop**	**gpu**	**high_mem**	**support**
Basic user access?	Yes	Yes	Yes	No (?) ($$$ from PI)	No (admins only)
# Nodes	100	7	18	42	(full grid)
CPU distribution	35x16, 65x8	3x1, 2x36, 2x8	17x16; 1x36	42x36	
# CPUs total	1080	91	308	1512	
# GPUs per node	0	0	17x2; 1x4	0	
# GPUs total	0	0	38	0	

SLURM: Overview

Is a **job scheduler:**

- Lets a user request a compute node to do an analysis (job)
- Provides a framework (commands) to start, cancel, and monitor a job
- Ensures efficient use of shared computing resources

You submit jobs and their resource needs to slurm, not to machines

- It manages where they go, prioritization, when they start, …

Everything goes through slurm!

- Examples of commands: https://github.com/statgen/SLURM-examples

SLURM: Grid Manager

https://slurm.schedmd.com

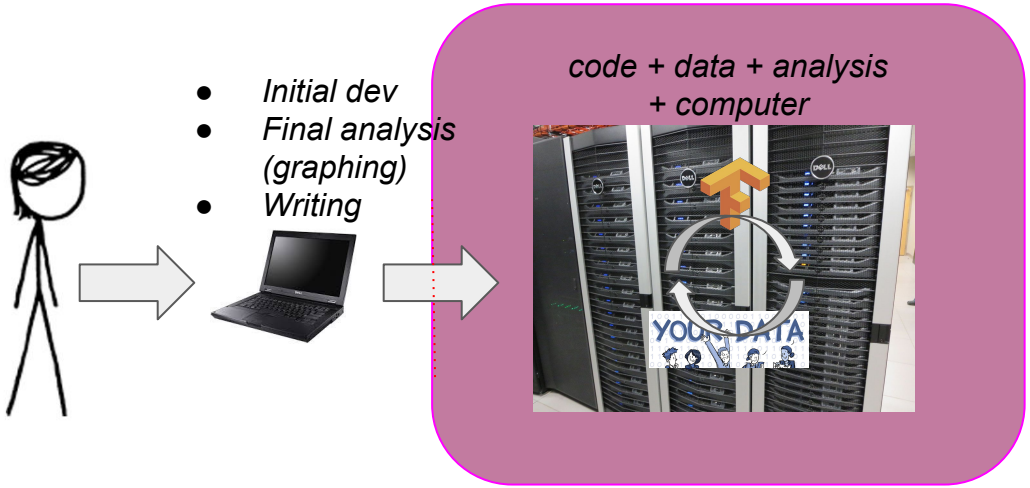
> “Slurm is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters. ... First, it **allocates exclusive and/or non-exclusive access to resources (compute nodes) to users for some duration of time** so they can perform work. Second, it provides a framework for **starting, executing, and monitoring** work (normally a parallel job) on the set of allocated nodes. Finally, it **arbitrates contention for resources** by managing a queue of pending work... accounting, advanced reservation, gang scheduling (time sharing for parallel jobs), backfill scheduling, topology optimized resource selection, resource limits by user or bank account, and sophisticated multifactor job prioritization algorithms.”

Outline

- Grid Basics
  - What is a grid? Compute+storage+management
  - High-level: How to use a grid
- **Submitting jobs**
  - Testing → Submitting “real” jobs
  - Managing jobs
- Requesting resources (gotchas)
  - GPUs
  - Memory
  - Time limits
  - Features

Re-examining the Grid Workflow

Current goal: Perform small-scale testing on the grid (at command line; synchronous)



- *Initial dev*
- *Final analysis (graphing)*
- *Writing*

1. Set up the environment
   a. Transfer code (& data, if not there)
   b. Install packages
2. Test interactively if necessary (should be limited)

Set up the Environment

1. Transfer code

   ```
   scp -R pytorch-examples ada:.
   ```

   Okay, not great

   Or

   ```
   ssh ada
   git clone https://github.com/pytorch/examples.git
   ```

   Much better

2.

Set up the Environment

1. Transfer code

   ```
   scp -R pytorch-examples ada:.
   ```

   Okay, not great

   Or

   ```
   ssh ada
   git clone https://github.com/pytorch/examples.git
   ```

   Much better

2. Install packages & libraries
3.

Setting up the Environment: `modulefiles`

- Standard Linux software to help control possibly conflicting software dependencies
- Encapsulate the environment needed to use each software package in a module so that users of a shared system can use conflicting software packages
- Each modulefile updates the necessary, standard environment variables:
  - Binary path ($PATH)
  - Include paths ($CPATH, $C_INCLUDE_PATH, $CPLUS_INCLUDE_PATH)
  - Linking & runtime paths ($LIBRARY_PATH, $LD_LIBRARY_PATH, $PYTHONPATH, etc.)
  - Any other environment variables

Checking for Loaded & Available `modulefiles`

To see what modules are currently loaded in your session, do:

```
$ module list
```

To see what modules are available to be loaded, do:

```
$ module avail
```

You can use module avail with grep to find certain modules:

```
$ module avail 2>&1 | grep -i conda
   Anaconda2/2018.12
   Anaconda2/2019.10                                          (D)
   Anaconda3/2020.07
   Miniconda3/4.7.10
```

Writing Your Own modulefiles

You can do this, but that’s for a different time.

Loading conda & creating envs (create-env.bash)

```
$ mkdir ~/ferraro_user/.ada_conda

$ ln -s ~/ferraro_user/.ada_conda .conda

$ conda create --prefix=ferraro_user/ada_envs/nlp-env \
      pytorch torchvision torchaudio torchtext cudatoolkit=11.0 -c pytorch

## Do these next two every time you want to load the environment (or memoize it)
$ source /usr/ebuild/software/Anaconda3/2020.07/etc/profile.d/conda.sh

$ conda activate /home/ferraro/ferraro_user/ada_envs/nlp-env
```

Set up the Environment

1. Transfer code

   ```
   scp -R pytorch-examples ada:.
   ```

   Okay, not great

   Or

   ```
   ssh ada
   git clone https://github.com/pytorch/examples.git
   ```

   Much better

2. Install packages & libraries
3. Transfer data

Testing Code on the Grid

DO NOT RUN CODE ON THE LOGIN NODE!!!

Testing Code on the Grid

Recommended steps: You've already tested your code for obvious bugs.

1. If you *must*, do *small-scale* interactive debugging
   a. This will become much less necessary as you become more familiar with using slurm.
   b. For a variety of reasons, it's better to limit interactive sessions.
2. Write a batch script.
3. If necessary, do small-scale tests. Then, run more broadly.

Common SLURM Commands

`sbatch`

`srun`

`scancel`

`sinfo`

`squeue`

Two SLURM commands: `srun` and `sbatch`

- **`sbatch [options*] script`**: Asynchronously
  - Allocate resources from the grid
  - Run `script` on those nodes/CPUs
- **`srun [options*] script`**: Synchronously
  - Sub-allocate resources from a larger allocation
  - Run `script` on those nodes/CPUs
  - (can be run within an **`sbatch`**'s script)

srun vs sbatch: what do you need?

Interactive usage? **srun**

Batch usage? **sbatch** (or **sbatch**+**srun**)

My experiments need...

	1 node	A small number (~ ≤ 10) of independent nodes (e.g., testing a small number of model architectures in an ML project)	Many (> 10) independent nodes (naively distributed code)	Any number of interdependent nodes (distributed code, e.g. MPI)
1 CPU (serial code)				
Multiple CPUs (parallel code, e.g., pthreads, OpenMP, ThreadPools, Python multiprocessing)				
1 GPU				
Multiple GPUs				
Large (32GB+) CPU memory				
Very Large (64GB+) CPU memory				

sbatch

sbatch → **srun**

Running Synchronous Jobs

srun [options*] script: Synchronously

- Allocate resources from the grid
- Run `script` on those nodes

```
$ srun --mem=20000 --gres=gpu:1 --time=1:00:00 --constraint=rtx_2080 \
    --pty --preserve-env $SHELL
```

Running Synchronous Jobs

srun [options*] script: Synchronously

- Allocate resources from the grid
- Run `script` on those nodes

20G max RAM

```
$ srun --mem=20000 --gres=gpu:1 --time=1:00:00 --constraint=rtx_2080 \
    --pty --preserve-env $SHELL
```

Running Synchronous Jobs

srun [options*] script: Synchronously

- Allocate resources from the grid
- Run `script` on those nodes

20G max RAM

Use one GPU

```
$ srun --mem=20000 --gres=gpu:1 --time=1:00:00 --constraint=rtx_2080 \
    --pty --preserve-env $SHELL
```

Running Synchronous Jobs

srun [options*] script: Synchronously

- Allocate resources from the grid
- Run `script` on those nodes

20G max RAM

Use one GPU

Run for 1 hour max

```
$ srun --mem=20000 --gres=gpu:1 --time=1:00:00 --constraint=rtx_2080 \
    --pty --preserve-env $SHELL
```

Running Synchronous Jobs

srun [options*] script: Synchronously

- Allocate resources from the grid
- Run `script` on those nodes

20G max RAM

Use one GPU

Run for 1 hour max

Ask for a node with 2080 TIs on it

```
$ srun --mem=20000 --gres=gpu:1 --time=1:00:00  --constraint=rtx_2080  \
     --pty --preserve-env $SHELL
```

Running Synchronous Jobs

srun [options*] script: Synchronously

- Allocate resources from the grid
- Run `script` on those nodes

20G max RAM

Use one GPU

Run for 1 hour max

Ask for a node with 2080 TIs on it

```
$ srun --mem=20000 --gres=gpu:1 --time=1:00:00 --constraint=rtx_2080 \
    --pty --preserve-env $SHELL
```

Flags for synchronous usage

(handy, but potentially very brittle!!!)

Running Synchronous Jobs

srun [options*] script: Synchronously

- Allocate resources from the grid
- Run `script` on those nodes

20G max RAM

Use one GPU

Run for 1 hour max

Ask for a node with 2080 TIs on it

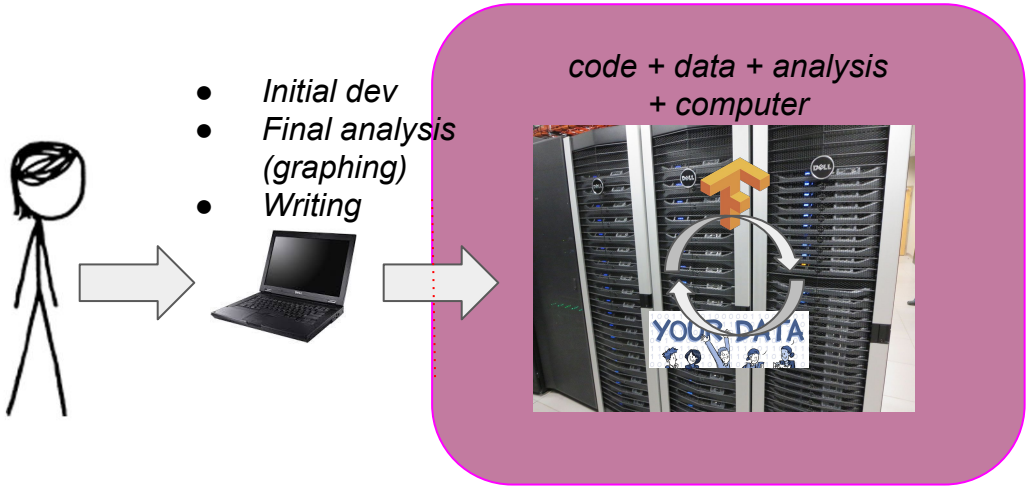
```
$ srun --mem=20000 --gres=gpu:1 --time=1:00:00 --constraint=rtx_2080 \
    --pty --preserve-env $SHELL
```

Flags for synchronous usage

command to run

Re-examining the Grid Workflow

Current goal: Perform larger testing on the grid (batch; asynchronous)



- *Initial dev*
- *Final analysis (graphing)*
- *Writing*

Write an sbatch script that:

1. Requests appropriate resources
2. Sets up the environment
   a. Loads modules, sets variables
3. Runs asynchronously

Perform async on the grid: Write an sbatch script

Write an sbatch script that:

1. Requests appropriate resources
2. Sets up the environment
    a. Loads modules, sets variables
3. Runs asynchronously

Run async.

sbatch [options*] **script**: Asynchronously

- Allocate resources from the grid
- Run `script` on those nodes

run-lm.slurm

```
$ sbatch --mem=20000 --gres=gpu:1 --time=60:00 --constraint=rtx_2080 ./run-lm.slurm
```

Run async.

sbatch [options*] **script**: Asynchronously

- Allocate resources from the grid
- Run `script` on those nodes

run-lm.slurm

```
$ sbatch --mem=20000 --gres=gpu:1 --time=1:00:00 --constraint=rtx_2080  $SHELL
```

Issues:

1. Output is written to `pwd` on submission node (in /home)!
2. Still relying on some default for sbatch options
3. sbatch options are easy to mess up

Run async.

sbatch [options*] **script**: Asynchronously

- Allocate resources from the grid
- Run `script` on those nodes

run-lm-headers.slurm

```
$ sbatch $SHELL
```

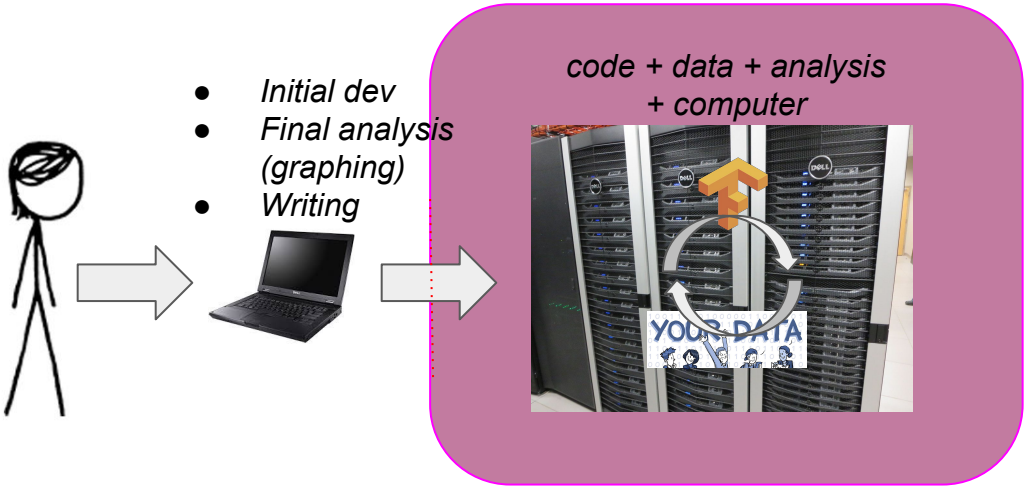
Advantages:

- Less typing at terminal → easier to submit
- More self-documenting (how much memory? time limit?)
- Can be overridden at command line

```
$ head -6 run-lm-headers.slurm
#!/bin/bash
#SBATCH --mem=20000
#SBATCH --gres=gpu:1
#SBATCH --time=1:00:00
#SBATCH --constraint=rtx_2080
```

Re-examining the Grid Workflow

Current goal: Run the code on the grid at scale



- *Initial dev*
- *Final analysis (graphing)*
- *Writing*

Run our existing sbatch script that:

1. Requests appropriate resources (**batch** partition)
2. Sets up the environment
   a. Loads modules, sets variables
3. Runs asynchronously

On an **arbitrary** file, 203 total

Two Ways of Looping

`Development sbatch command: sbatch run-lm-headers.slurm`

Shell `for` loop

1. Loop through all needed files
2. Issue a separate sbatch command for each file

203 files ⇒ 203 jobs (1 task each)

Advantage: minimal changes to the script

Disadvantage: *much* harder to control (delete, hold, release, throttle) jobs

SLURM array job

Two Ways of Looping

```
Development sbatch command: sbatch `pwd`/scripts/submit_pos_count_with_headers.slurm
```

Shell `for` loop

```
for f in $(find /umbc/xfs1/ferraro/common/data/cac/cawiki-en.text_pos -type f); do
  sbatch `pwd`/scripts/submit_pos_count_headers_arg.slurm "${f}"
done
```

Advantage: minimal changes to the script

Disadvantage: *much* harder to control (delete, hold, release, throttle) jobs

203 files ⇒ 203 jobs (1 task each)

SLURM array job

Two Ways of Looping

```
Development sbatch command: sbatch `pwd`/scripts/submit_pos_count_with_headers.slurm
```

Shell `for` loop

```
for f in $(find /umbc/xfs1/ferraro/common/data/cac/cawiki-en.text_pos -type f); do
  sbatch `pwd`/scripts/submit_pos_count_headers_arg.slurm "${f}"
done
```

Advantage: minimal changes to the script

Disadvantage: *much* harder to control (delete, hold, release, throttle) jobs

203 files ⇒ 203 jobs (1 task each)

SLURM array job

1. Add logic to the script to associate task IDs with files
2. Add an `--array=start-end%throttle`flag to the *single* sbatch command

203 files ⇒ 1 job, but with 203 tasks

Advantage: very easy to control (delete, hold, release, throttle) jobs

Disadvantage: must perform integer to configuration mapping

Two Ways of Looping

```
Development sbatch command: sbatch `pwd`/scripts/submit_pos_count_with_headers.slurm
```

Shell `for` loop

```
for f in $(find /umbc/xfs1/ferraro/common/data/cac/cawiki-en.text_pos -type f); do
  sbatch `pwd`/scripts/submit_pos_count_headers_arg.slurm "${f}"
done
```

Needed to update slurm script to use command line arguments

Advantage: minimal changes to the script

Disadvantage: *much* harder to control (delete, hold, release, throttle) jobs

203 files ⇒ 203 jobs (1 task each)

SLURM array job

```
sbatch --array=0-202%40 `pwd`/scripts/submit_pos_count_headers_dirarg.array.slurm \
  /umbc/xfs1/ferraro/common/data/cac/cawiki-en.text_pos
```

Needed to update slurm script to use SLURM array variables

Advantage: very easy to control (delete, hold, release, throttle) jobs

Disadvantage: must perform integer to configuration mapping

203 files ⇒ 1 job, but with 203 tasks

Array Jobs

(syntax is more general:
read `man sbatch`)

Advanced

```
sbatch [--hold] --array=start-end[%throttle][options*] script
```

Asynchronously

- Allocate resources from the grid (as given by **`options*`**)
- Run `script` on **`end`**-**`start`**+1 times
  - Each is a different task ($SLURM_ARRAY_TASK_ID), but under the same, single job ($SLURM_JOB_ID)
  - **`start`**, **`end`** & **`throttle`** are ints
    - **`start`** >= 0, **`end`** < MaxArraySize (SLURM parameter: 20,000)
    - See /etc/slurm/slurm.conf
  - Allow **`throttle`** tasks to run simultaneously (default: run as many as possible)

- A good option is to `--hold` the jobs too (job control)

Array Job Logic

Advanced

Use $SLURM_ARRAY_TASK_ID

```
### get the file
file_index=${SLURM_ARRAY_TASK_ID} ## 0-indexed files
input_directory=/umbc/xfs1/ferraro/common/data/cac/cawiki-en.text_pos
input_file=${input_directory}/cawiki-en.${file_index}text_pos.tsv.gz
```

Job Control

Advanced

Deleting jobs (running or queued)

- If job has ID 10000
  - Non-array jobs (or all tasks in array): `scancel 10000`
  - Tasks (#0-#14) in array job: `scancel 10000_0-14`

hold jobs: Add `--hold` to sbatch command

- Registers job(s) with scheduler but does not queue them to run
- Release jobs with scontrol release. If job has ID 10000
  - Non-array jobs (or all tasks in array): `scontrol release 10000`
  - Tasks (#0-#14) in array job: `scontrol release 10000_0-14`

Job dependencies

- Require other job X to complete before Y will run
- See `--dependency=<dependency_list>` in `man sbatch`

Job Control and Aliases

hpc-tutorial/slurm_aliases.bash

Source this file (`source slurm_aliases.bash`) to get access to some aliases (short-cuts) to SLURM management commands. For example:

- `sinteract`: Get one CPU (with 6GB vMem) on the batch partition (4 hours)
- `my-jobs`: List the jobs you have submitted (running, queued, pending)
- `list-nodes`: Show all nodes in the grid, with various resources available

(and more)

SLURM (Array) Job Environment Variables

From the man page for sbatch:

SLURM_JOB_ID: the job

SLURM_ARRAY_JOB_ID: the job ID of the array

SLURM_ARRAY_TASK_ID: will be set to the job array index value

SLURM_ARRAY_TASK_COUNT: will be set to the number of tasks in the job array

SLURM_ARRAY_TASK_MAX: will be set to the highest job array index value (`end`)

SLURM_ARRAY_TASK_MIN: will be set to the lowest job array index (`start`)

Outline

- Grid Basics
  - What is a grid? Compute+storage+management
  - High-level: How to use a grid
- Submitting jobs
  - Testing → Submitting “real” jobs
  - Managing jobs
- Requesting resources (gotchas)
  - GPUs
  - Memory
  - Time limits
  - Features

Requesting Memory

- Memory is measured in
  - MB
  - virtual memory
- `--mem=<M>` provides an upper-bound on the memory needed per node
- `--mem-per-cpu=<M>` provides a ***lower*-bound** on the memory needed per CPU

Requesting GPUs

- Use the `--gres=gpu:<number_of_devices_per_node>` to request GPUs

- Include `--gres=gpu:<num>`: if you do, `$CUDA_VISIBLE_DEVICES` should be properly set and you shouldn’t clobber anyone else’s jobs

Time Limit

- `--time=<T>` provides an upper-bound on the wallclock time needed

- Formats (see `man sbatch`)
  - "minutes"
  - "minutes:seconds"
  - "hours:minutes:seconds"
  - "days-hours"
  - "days-hours:minutes"
  - "days-hours:minutes:seconds"

Features: --constraint=<feature>

Features of node: currently, how to select which type of card you want

- --constraint=rtx_2080
- --constraint=rtx_6000
- --constraint=rtx_8000

Partitions vs. Quality of Service (QoS)

“Partitions” group certain nodes together

- Makes scheduling easier
- Clearly state what types of physical resources your jobs need (and what they don’t)
- Partitions do not have to be mutually exclusive

QoS

- High-level binning of how *many* of the different resources you can use

Use both partitions and QoS to effectively manage your jobs

Quality-of-Service (QoS)

QOS	Wall time limit per job	CPU time limit per job	Total cores limit for the QOS	Cores limit per user	Total jobs limit per user
short	1 hour	1024 hours	2048	560	—
normal (default)	4 hours	1024 hours	—	256	—
medium	24 hours	1024 hours	2048	256	—
medium_prod	48 hours	2048 hours	2048	768	—
long	5 days	—	256	16	4
long_contrib	5 days	—	768	128	4
long_prod	45 days	—	64	—	—
support	—	—	—	—	—

https://hpcf.umbc.edu/scheduling-rules-on-taki/

(check link for most recent QoS)

Writing Your Own modulefiles

1. **Write a small Tcl file**

   location: `/<path-to-directory-of-my-modules>/<library_name>/<version>`

   Notice that `<version>` is a text (tcl) file with no extension

2. **Tell modulefiles where to find this new module**

   a. Add `/<path-to-directory-of-my-modules>` to a personalized repository for you

      ```
      $ module use "~/ferraro_common/module_files"
      ```

   b. This is specific to each shell session: add it to your .bashrc to make it permanent

3. **Load it**

   ```
   $ module load <library_name>/<version>
   ```

Getting anaconda 2.7 with modulefiles

pi_ferraro conda install

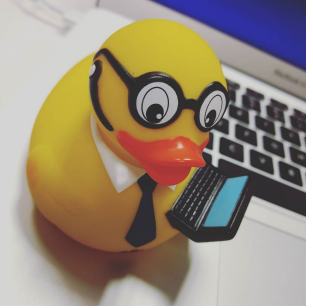
```
$ module load conda/2.7

$ which python
/umbc/xfs1/ferraro/common/anaconda2
/bin/python

$ python --version
Python 2.7.14 :: Anaconda, Inc.
```

```
$ cat ~/ferraro_common/module_files/conda/2.7

#%Module -*- tcl -*-
##
## modulefile
##

proc ModulesHelp { } {
  puts stderr "\tAdds Anaconda 2.7 to your
environment variables,"
}
module-whatis "adds Anaconda 2.7 to your
environment variables"

set                 root
/umbc/xfs1/ferraro/common/anaconda2
prepend-path        PATH                $root/bin
```

Outline

- Grid Basics
  - What is a grid? Compute+storage+management
  - High-level: How to use a grid
- Submitting jobs
  - Testing → Submitting “real” jobs
  - Submitting many jobs
  - Managing jobs
- Requesting resources (gotchas)
  - GPUs
  - Memory
  - Time limits
  - Features

Remember: Ask for Help if Needed



nicely :)

1. Read the error (if any) carefully
2. Check your
   a. Paths (to code, input files, output files)
   b. Missing modules (in your submission script)
   c. Check your resources (# CPUs, # nodes, amount of memory, run time, etc.)
3. Read the man pages
4. Do a quick Google search
5. File a ticket: https://doit.umbc.edu/request-tracker-rt/doit-research-computing/
   a. If you're working with me (Frank), cc me on all tickets

Paraphrased from https://testlio.com/blog/the-ideal-bug-report/

"Think of your bug report like a good tweet: You want it short, sweet, and to the point."

- Subject: very short (< 10 word) summary of what's wrong
- Main body: brief (2-3 sentence) summary of what's wrong
- Steps to reproduce
  - Is there a script you can point to?
  - Code environment: what modules are you trying to use (and where are they)
  - Resources: are you trying to use CPUs, GPUs, multiple nodes, etc.
- Expected Result
- Actual Result: including clipped error messages is okay