MSI/Git Workshop

Tom Kono

2017-11-14

Slides: https://z.umn.edu/msi-git

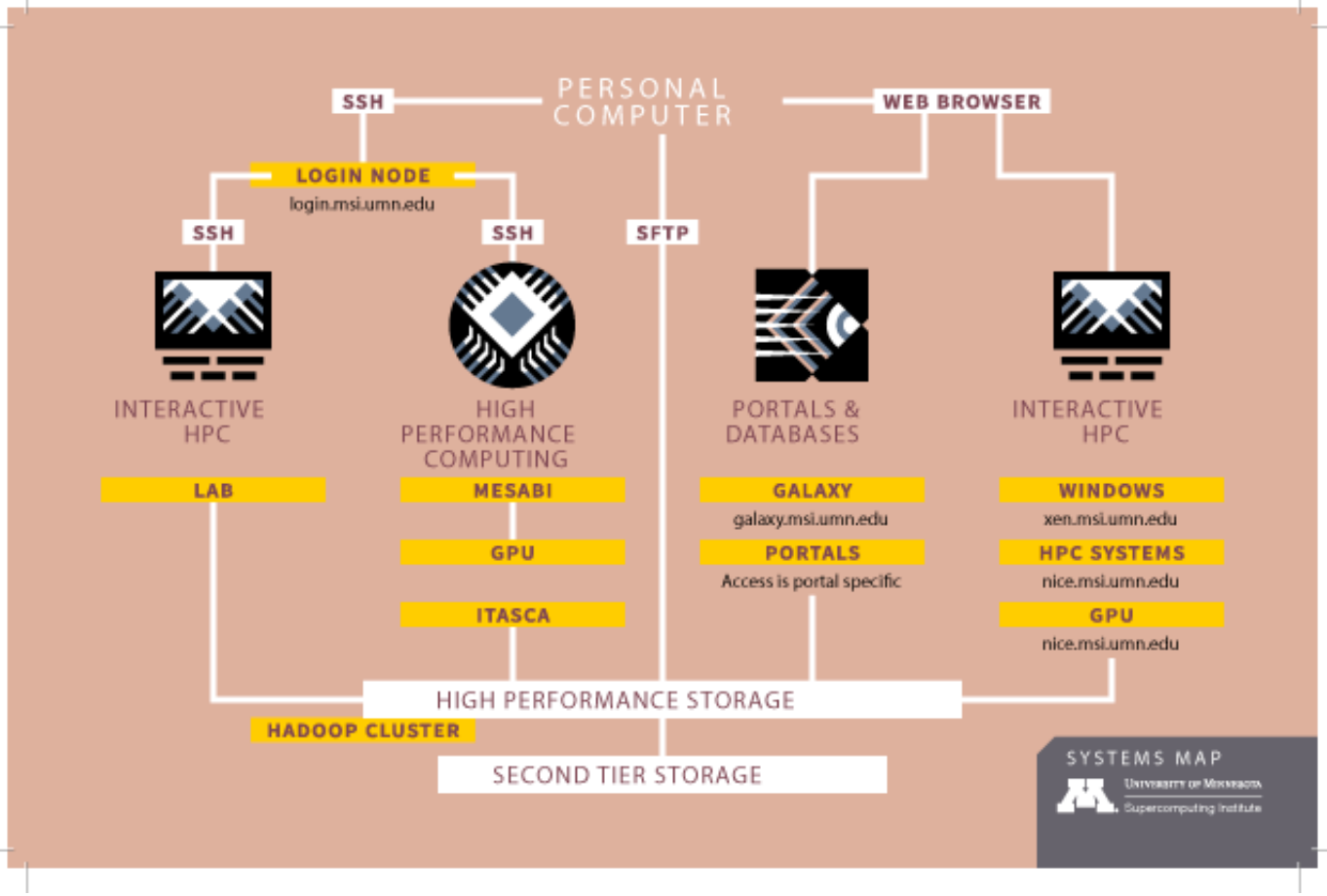
Outline

- Part 1: Intro to MSI Systems
  - HPC systems, Storage, Queues, Modules
- Part 2: Parallelization and Task Arrays
  - GNU Parallel, PBS Task Arrays
- Part 3: Git and GitHub

Part 1: MSI Organization

- Divisions of computing support:
  - **High performance computing (HPC)**
  - **Interactive HPC (Linux)**
  - **Storage systems**
  - GPU nodes

Part 1: MSI Organization



Part 1: HPC Overview

- Non-interactive jobs: all work must be done as shell scripts
- Run CentOS Linux
- Hundreds of cores, hundreds of GB RAM, days to weeks of runtime
- Two clusters: Itasca and Mesabi

Part 1: HPC - Itasca

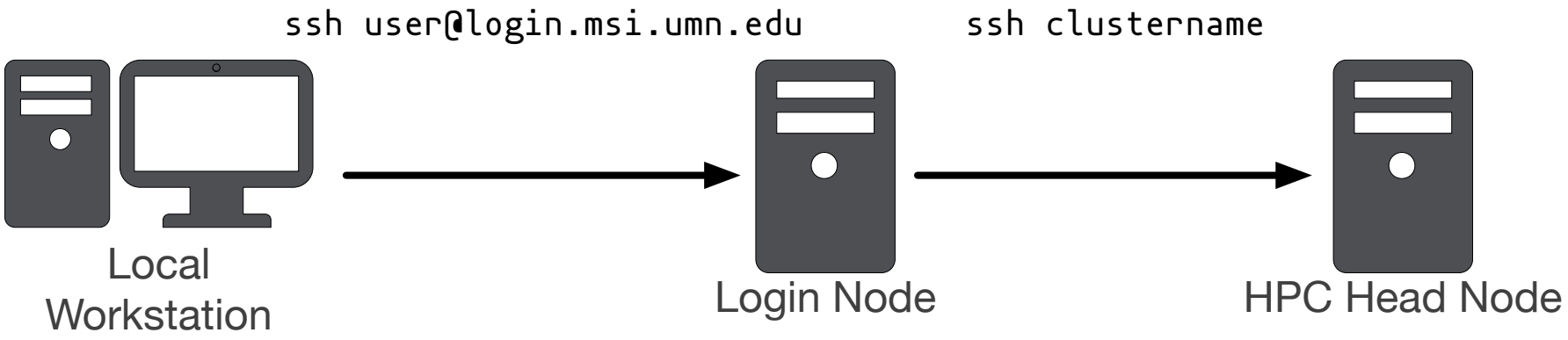
- Many 22 GB RAM, 8 core nodes, no node sharing
- Several 64+ GB RAM, 16 core nodes
- Great for short jobs that can be run in parallel

Part 1: HPC - Mesabi

- Very diverse system: long-running jobs, highly parallel jobs, high-memory jobs
- Has some GPU processing capacity

Part 1: Accessing HPC Systems

- Must be done on-campus, or through university VPN
- Access via terminal and `ssh`:



Part 1: HPC Queues

- Scheduling jobs is done by TORQUE
- Jobs must be submitted from the cluster head node
- You can submit to a variety of job queues, depending on the parameters of the analysis

Part 1: HPC Queues

- Itasca queues:

Queue name	Number of Nodes	Processor cores per node	Wallclock Limit	Total Node Memory Limit	Per-core Memory Limit	Local Scratch (GB/node)	Per User Running Jobs	Per User Idle Jobs (gaining priority in queue)
batch (default)	1086 nodes (8688 cores)	8	24 hours	22gb	2750mb	90 GB	60	8
devel	32 nodes (256 cores)	8	2 hours	22gb	2750mb	90 GB		
long	28 nodes (224 cores)	8	48 hours	22gb	2750mb	90 GB		
sb	35 nodes (560 cores)	16	48 hours	62gb	3875mb	112 GB		
sb128	8 nodes (128 cores)	16	96 hours	126gb	7875mb	534 GB		
sb256	8 nodes (128 cores)	16	96 hours	254gb	15875mb	534 GB		

Service Unit (SU) rate: 1.5 CPU hours / SU

Part 1: HPC Queues

- Mesabi queues:

Queue name	Node Sharing	Max Nodes Per Job	Min Nodes Per Job	Processor cores per node	Wallclock Limit	Total Node Memory Limit	Per-core Memory Limit	Local Scratch (GB/node)	Per User Limits	Per Group Limits
small[€]	Yes	9	None	24	96 hours	62gb	2580mb	390 GB	500 Jobs	1800 total cores[‡]
large	No	48	10	24	24 hours	62gb	2580mb	390 GB	4 Jobs	16 Jobs
widest[§]	No	360	49	24	24 hours	62gb	2580mb	390 GB	4 Jobs	16 Jobs
max	Yes	1 (single core per job)	None	1	696 hours	62gb	62gb	390 GB	4 Jobs	16 Jobs
ram256g	Yes	2	None	24	96 hours	252gb	10580mb	390 GB	2 nodes	1800 total cores[‡]
ram1t	Yes	2	None	32[†]	96 hours	998gb	31180mb	228 GB	2 nodes	1800 total cores[‡]
k40 GPU nodes[*]	No	40	None	24	24 hours	126gb	5290mb	390 GB	None	1800 total cores[‡]
mesabi (default)	The mesabi queue is a meta-queue, which will automatically route jobs to the **small**, **large**, **widest**, or **max** queues, according to where each job will best fit based on the resource request.									

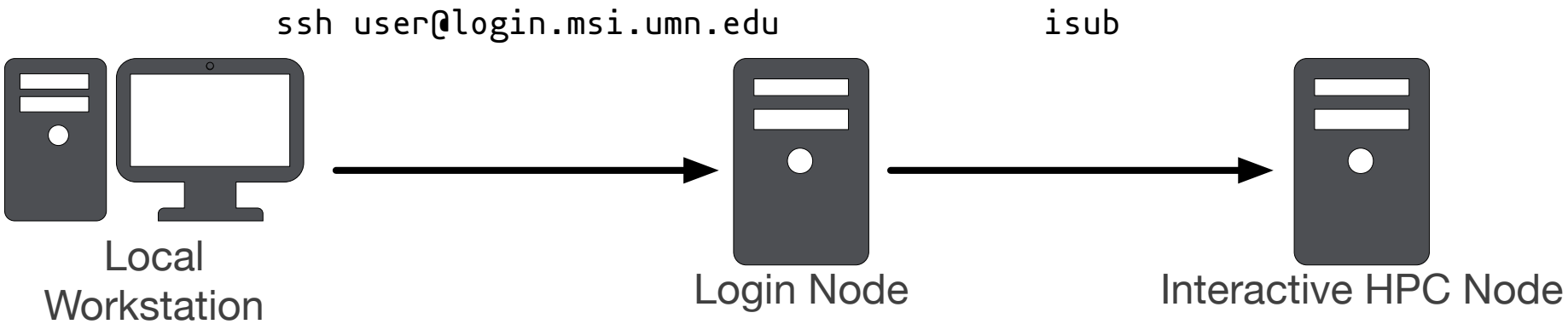
Service Unit (SU) rate: 1.5 CPU hours / SU

Part 1: Interactive HPC

- Small interactive jobs: some GUI applications, R, MATLAB
- 1-16 cores, tens of GB RAM, hours of runtime
- Useful for testing scripts and programs before throwing them onto HPC

Part 1: Accessing Interactive HPC

- Must be done on-campus, or through university VPN
- Terminal, `ssh`, and `isub`:



Part 1: Interactive HPC

- Use interactive HPC if you want to run or test analysis software
  - You can technically run software on the HPC head node, but your processes can (and will) be killed
- Never run analyses on the login node. You can lock up access to the systems and people will yell at you.

Part 1: Storage Systems

- **Primary**: Home directories on SSD. Snapshot backups, quotas enforced
- **Scratch**: Temp storage, quotas not enforced, files kept for at most 30 days
- **Tier 2**: Long-term storage, Amazon S3 system, no snapshots.
- **Archive**: Tape storage, system in beta

Part 1: Important Directories

- Primary storage

  `/home/groupname/username`: Home directory. Scripts/small data files

  `/home/groupname/shared`: Group shared directory. Software, reference sequences, common datasets.

- Scratch storage

  `/panfs/roc/scratch`: Global scratch storage.

  `/scratch.local`: Node local scratch storage. Cleaned out at end of job.

Part 1: Storage Guidelines

- Primary has a group quota. Once quota is met, **no new files can be made**.
- Remove files from scratch when done.
- **Tier 2 is unlimited, but not backed up. Be careful.**
- Archive is limited access, you'll know if you have it

Part 1: Modules

- Software on MSI is loaded using the `module` system
- Many common pieces of bioinformatics software (bowtie2, bwa, samtools, blast, etc.) are available as modules
- `module avail` to list, `module load` to use

Part 1: Modules

- You can download/compile software if there is no module available for it.
  - Mostly unsupported: you should know what you are doing
  - You can request software to be installed on the system; email the help desk.
- **Always specify a version when loading a module, for reproducibility.**

Part 1: Let's Access MSI

1. Open terminal program
   - Mac: Applications **>** Utilities **>** Terminal
   - Windows: Download puTTY from http://www.putty.org/
   - Linux: You should know how to do this.

Part 1: Let's Access MSI

2. Connect to the login node
   - Mac/Linux:
     1. Type `ssh user@login.msi.umn.edu`
     2. Type 'y' when the security alert shows up.
     3. Enter password.
   - Windows:
     1. Type `login.msi.umn.edu` into the Host Name box, click "Open"
     2. Enter username and password as with Mac

Part 1: Let's Access MSI

3. Request an interactive session
   - You should see something like
     `username@login03 [~] %`
     before your cursor
   - Type `isub`
   - You may have to wait a minute or two. Once prompt returns, you are running in an interactive session

Part 1: Let's Access MSI

4. Load the R software module
   - Type `module load R/3.3.3`
   - Type `R`
   - This should work with other modules:
     - Try `module load samtools/1.6`
     - Try `module load ncbi_blast+/2.2.29`
     - Try `module avail bwa` to view BWA versions

Part 1: MSI Organization

System status
https://www.msi.umn.edu/systemstatus

System usage
https://s3.msi.umn.edu/pbsnodes/index.html

Queues
https://www.msi.umn.edu/queues

Writing a job script
https://www.msi.umn.edu/content/job-submission-and-scheduling-pbs-scripts

Outline

- Part 1: Intro to MSI Systems
  - HPC systems, Storage, Queues, Modules
- Part 2: Parallelization and Task Arrays
  - GNU Parallel, PBS Task Arrays
- Part 3: Git and GitHub

Part 2: Parallelization

- Many analyses can be broken down into independent tasks
- Examples:
  - Mapping reads from 60 samples to a reference genome
  - Generating thousands of gene alignments for relative rates calculations

Part 2: Parallel Tools on MSI

- **GNU Parallel**: fancy Perl script that monitors processes and interleaves commands
- **TORQUE task arrays**: Generate an array of jobs, run ranges of them at once as separate PBS jobs

Part 2: Parallel Tools on MSI

- Both are useful and have pros/cons
- GNU Parallel:
  - Somewhat non-intuitive, but very powerful
  - Requires tricks for multi-node jobs
- Task arrays:
  - Easy to use if you understand bash arrays
  - Can use multiple nodes without tricks

Part 2: Which Parallel Tool to Use?

- It's really just personal preference.
- My strategy: use both.
  - Task arrays when there are ≤ dozens of expensive/long tasks, where any of them might fail and need to be re-run.
  - Parallel when there are ≥ hundreds small/cheap tasks

Part 2: Using Task Arrays

- Example: You want to align paired end reads from 60 samples to your reference genome. Each sample should be aligned with identical parameters.

Part 2: Using Task Arrays

1. Organize your data in a way that is easy to programmatically access.
   - Example:

```
/panfs/roc/scratch/konox006/reads/sample1_R1.fastq.gz
/panfs/roc/scratch/konox006/reads/sample1_R2.fastq.gz
/panfs/roc/scratch/konox006/reads/sample2_R1.fastq.gz
/panfs/roc/scratch/konox006/reads/sample2_R2.fastq.gz
…
/panfs/roc/scratch/konox006/reads/sample60_R1.fastq.gz
/panfs/roc/scratch/konox006/reads/sample61_R2.fastq.gz
```

Part 2: Using Task Arrays

2. Inside your job script, generate bash arrays that hold the units of parallelization. In this example, it will be reads files
   - Example:

```
SAMPLES_F=($(find /panfs/roc/scratch/konox006/reads
-type f -name '*R1.fastq.gz' | sort -V))

SAMPLES_R=($(find /panfs/roc/scratch/konox006/reads
-type f -name '*R2.fastq.gz' | sort -V))
```

Part 2: Using Task Arrays

3. Use the `${PBS_ARRAYID}` variable to access an individual element of the arrays. In this case, the forward and reverse reads of a sample.
   - Example:

```
CURRENT_FWD=${SAMPLES_F[${PBS_ARRAYID}]}
CURRENT_REV=${SAMPLES_R[${PBS_ARRAYID}]}
SAMPLE=$(basename ${CURRENT_FWD} | cut -f 1 -d
'_')
```

Part 2: Using Task Arrays

4. Build the command using the element that you fetched from the array.
   - Example:

```
bwa mem \
    -t 8 -k 12 -M \
    /home/group/shared/references/ref.fa \
    ${CURRENT_FWD} ${CURRENT_REV} \
    > /panfs/roc/scratch/konox006/aln/${SAMPLE}.sam
```

Part 2: Using Task Arrays

5. Submit the array to the scheduler with `qsub` and the `-t` option. It can understand both comma separated values and ranges.
   - Example:
     ```
     qsub -t 0-59 bwa.sh
     ```
     (if jobs 33 and 48 fail)
     ```
     qsub -t 33,48 bwa.sh
     ```

Part 2: Task Array Notes

- You will get an email about each job starting and exiting (if you have configured the PBS script that way), so watch out.
- Example script here: https://github.com/TomJKono/Presentations/blob/master/MSI-Git/Task_Array_Example.sh

Part 2: Task Array Notes

- `${PBS_ARRAYID}` is a TORQUE specific variable. It's automatically set by the scheduler when a job starts.
- If `qsub` is called without the `-t` option, the value of `${PBS_ARRAYID}` will be `0`
- Ranges are inclusive. `qsub -t 0-10` will start 11 jobs: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

Part 2: Using GNU Parallel

- Example: You want to generate alignments for relative rates calculations. You have thousands of genes to align, but each alignment should take at most 15 minutes.

Part 2: Using GNU Parallel

1. Organize your data in a way that it is easy to programmatically access, similar to task arrays
   - Example:

```
/panfs/roc/scratch/konox006/genes/gene1_group.fasta
/panfs/roc/scratch/konox006/genes/gene2_group.fasta
/panfs/roc/scratch/konox006/genes/gene3_group.fasta
/panfs/roc/scratch/konox006/genes/gene4_group.fasta
…
/panfs/roc/scratch/konox006/genes/gene7999_group.fasta
/panfs/roc/scratch/konox006/genes/gene8000_group.fasta
```

Part 2: Using GNU Parallel

2. Build the commands, and save them to a text file with `echo`
   - Example:

```
for gene in /panfs/roc/scratch/konox006/genes/*
do
echo "clustalo -v -i ${gene} -t Protein \
     --full-iter --iter=10 --out /scratch/${gene}"
done > commands.txt
```

Part 2: Using GNU Parallel

3. Use `parallel` to dispatch these individual jobs to the multiple cores on the node. Some arithmetic with `ppn` may be necessary.
   - Example:

```
#PBS -l mem=22gb,nodes=1:ppn=24,walltime=24:00:00
…
parallel --jobs 4 < commands.txt
```

Part 2: Using GNU Parallel

4. Some tricks are necessary for **multi-node** jobs
   - See this page: https://www.msi.umn.edu/support/faq/how-can-i-use-gnu-parallel-run-lot-commands-parallel
   - Basically: modify the parallel environment, then use `--sshloginfile` and `${PBS_NODEFILE}` to dispatch jobs across nodes

Part 2: Let's Use Parallel

- Request an interactive session on MSI
  ```
  isub -m 4gb -w 2:00:00 -q lab
  ```
- Load the parallel module
  ```
  module load parallel
  ```

Part 2: Let's Use Parallel

- Make a new text file `commands.txt`

```
echo "a"
echo "b"
echo "c"
echo "d"
echo "this is a longer string"
```

Part 2: Let's Use Parallel

- Run it with bash, in serial

```
bash commands.txt
```

- Output:

```
a
b
c
this is a longer string
```

Part 2: Let's Use Parallel

- Now, run it with parallel

```
parallel < commands.txt
```

- Output (order may vary):

```
c
b
a
this is a longer string
```

Part 2: Parallel Tips

- Command lines are separated by newlines be default. This means you can string commands together to track progress for longer jobs
- You can even use the bash command separators `&&`, `||`, and `;`

Part 2: Parallel Tips

- Example: Keep track of commands that succeed

```
clustalo … && echo “gene1.fasta” >> Pass.txt
```

- Example: Keep track of commands that fail

```
clustalo … || echo “gene1.fasta” >> Fail.txt
```

Part 2: Parallel Tips

- There is a LOT more you can do with `parallel`. See the manual page: https://www.gnu.org/software/parallel/man.html
- Example `parallel` script here: https://github.com/TomJKono/Presentations/blob/master/MSI-Git/Parallel_Example.sh
  With commands here: https://github.com/TomJKono/Presentations/blob/master/MSI-Git/Parallel_Commands.txt

Outline

- Part 1: Intro to MSI Systems
  - HPC systems, Storage, Queues, Modules
- Part 2: Parallelization and Task Arrays
  - GNU Parallel, PBS Task Arrays
- Part 3: Git and GitHub

Part 3: Why Git?

- If anything, to keep a working version of a script, in case you break it
- (Relatively) easy to collaborate with other scientists on the same script
- Great place to host scripts for reproducibility after publication

Part 3: Git and GitHub

- Git is the software the manages versions and allows users to “clone” and modify repositories
- GitHub is a service provider that hosts git servers and manages access to repositories

Part 3: UMN Has GitHub

- Specifically, we have Enterprise GitHub
  - Unlimited private repositories (can control who can see/contribute to a repo)
  - Only available to people with an X500 name and password
- http://github.umn.edu/

Part 3: Installing Git

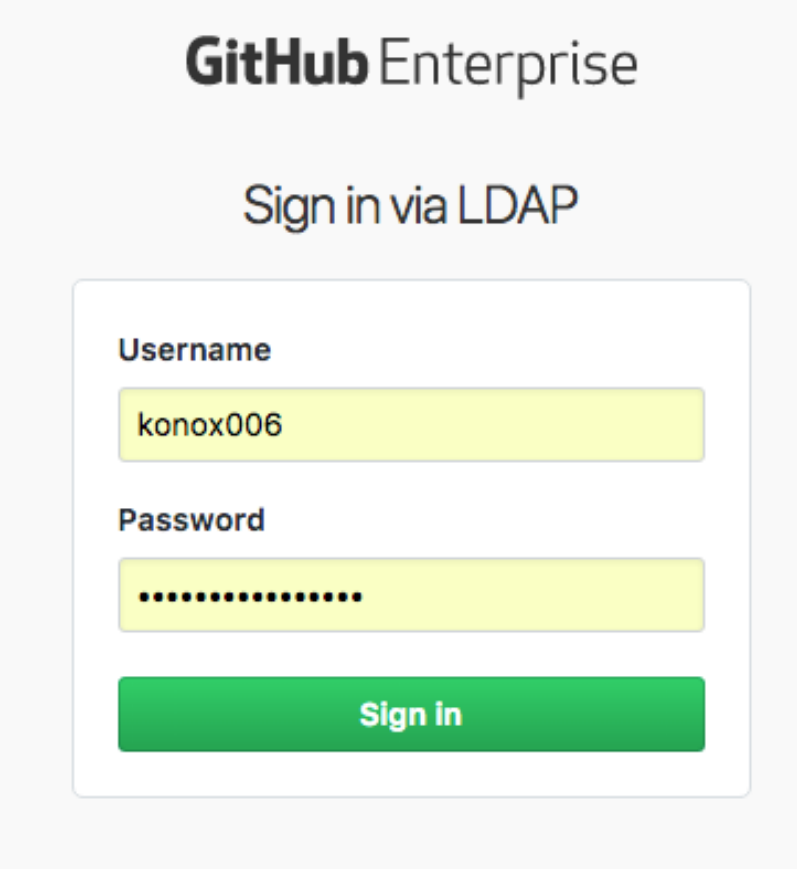
- Download the software
- Mac: https://git-scm.com/download/mac
- Windows: https://git-scm.com/download/win
- Note: I don't know how to use git on windows, sorry

Part 3: Verify Installation

```
[22:09:00] [tomkono@Toms-MacBook-Pro] (~) $ which git
/usr/local/bin/git
[22:09:07] [tomkono@Toms-MacBook-Pro] (~) $ git
usage: git [--version] [--help] [-C <path>] [-c name=value]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
   init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add        Add file contents to the index
   mv         Move or rename a file, a directory, or a symlink
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index
```

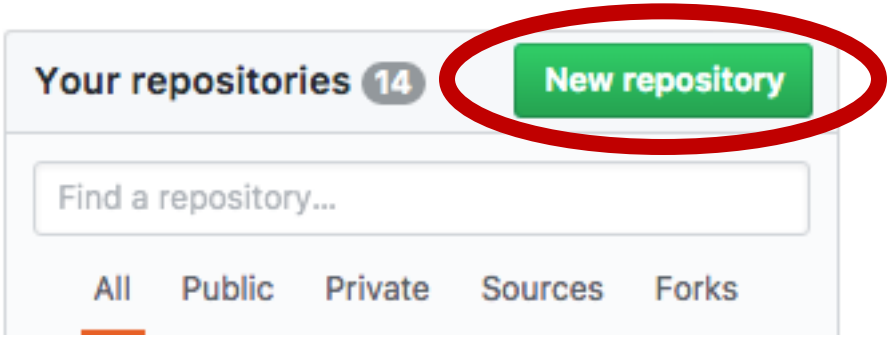
Part 3: Let's Use UMN GitHub

- Sign in to http://github.umn.edu/ to activate your account



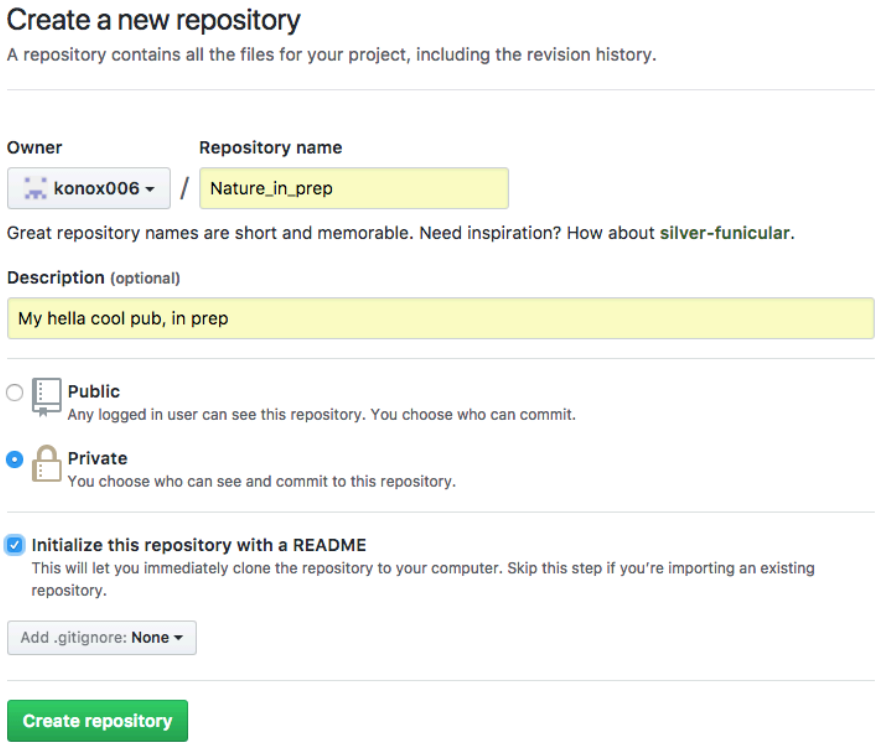
Part 3: Let's Use UMN GitHub

- Start a new repository with this button



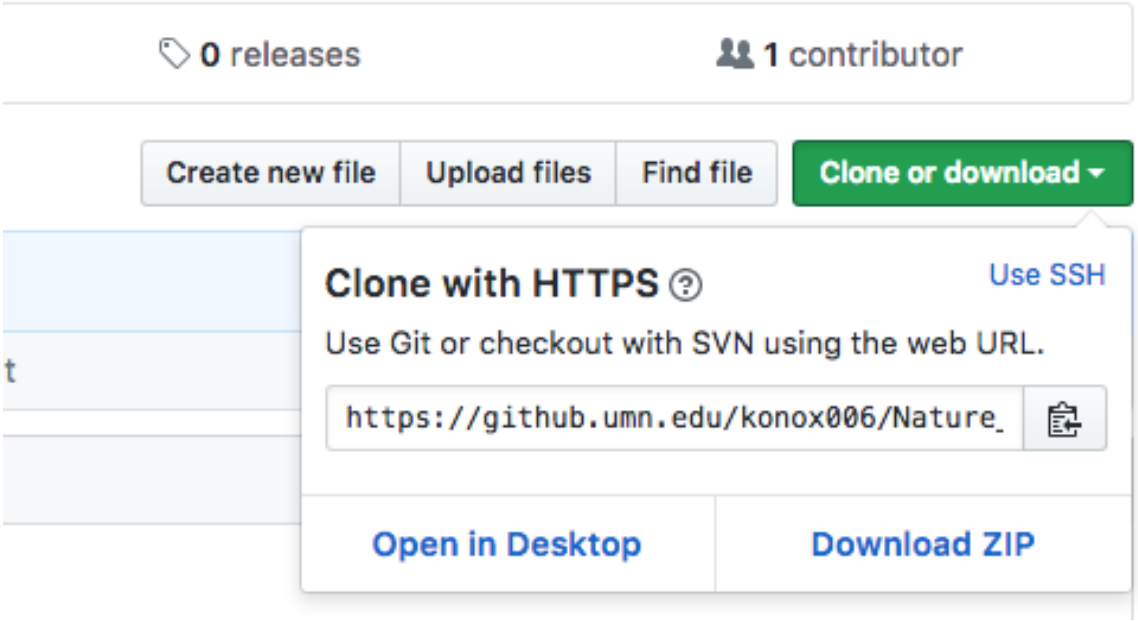
Part 3: Let's Use UMN GitHub

- Fill in the details



Part 3: Let's Use UMN GitHub

- Show the "clone" button to get the clone URL



Part 3: Let's Use UMN GitHub

- Clone the repo in your Terminal window

```
[22:35:51] [tomkono@Toms-MacBook-Pro] (~) $ git clone https://github.umn.edu/konox006/Nature_in_prep.git
Cloning into 'Nature_in_prep'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
[22:36:05] [tomkono@Toms-MacBook-Pro] (~) $ cd Nature_in_prep/
[22:36:15] [tomkono@Toms-MacBook-Pro] (~/Nature_in_prep) $
```

Part 3: Let's Use UMN GitHub

- Edit the README.md file to add some additional detail

```
1 # Nature in prep
2 My hella cool pub, in prep
3
4 This research project is so great, it'll knock your socks off. Trust me, even
5 though I haven't done any actual analyses or collected any data, this stuff
6 will be tip-top.
```

Part 3: Let's Use UMN GitHub

- View the changes with `status`, and `add` them to the repository

```
[22:48:59] [tomkono@Toms-MacBook-Pro] (~/Nature_in_prep) $ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
[22:49:01] [tomkono@Toms-MacBook-Pro] (~/Nature_in_prep) $ git add README.md
```

Part 3: Let's Use UMN GitHub

- Then `commit` them to create a "save point" in the repo history. Use `-m` to add a "commit message" that will display in the repository history. Make it meaningful (more than just "added stuff")

```
[22:49:04] [tomkono@Toms-MacBook-Pro] (~/Nature_in_prep) $ git commit -m 'Put crap in the README'
[master 8e51dab] Put crap in the README
 1 file changed, 4 insertions(+)
```

Part 3: Let's Use UMN GitHub

- Actually, that commit message is not very nice. Let's `amend` it:

```
[22:58:27] [tomkono@Toms-MacBook-Pro] (~/Nature_in_prep) $ git commit --amend
```

```
Put additional text in the README

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Sun Nov 12 22:58:27 2017 -0600
#
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#   (use "git push" to publish your local commits)
#
# Changes to be committed:
#       modified:   README.md
#
```

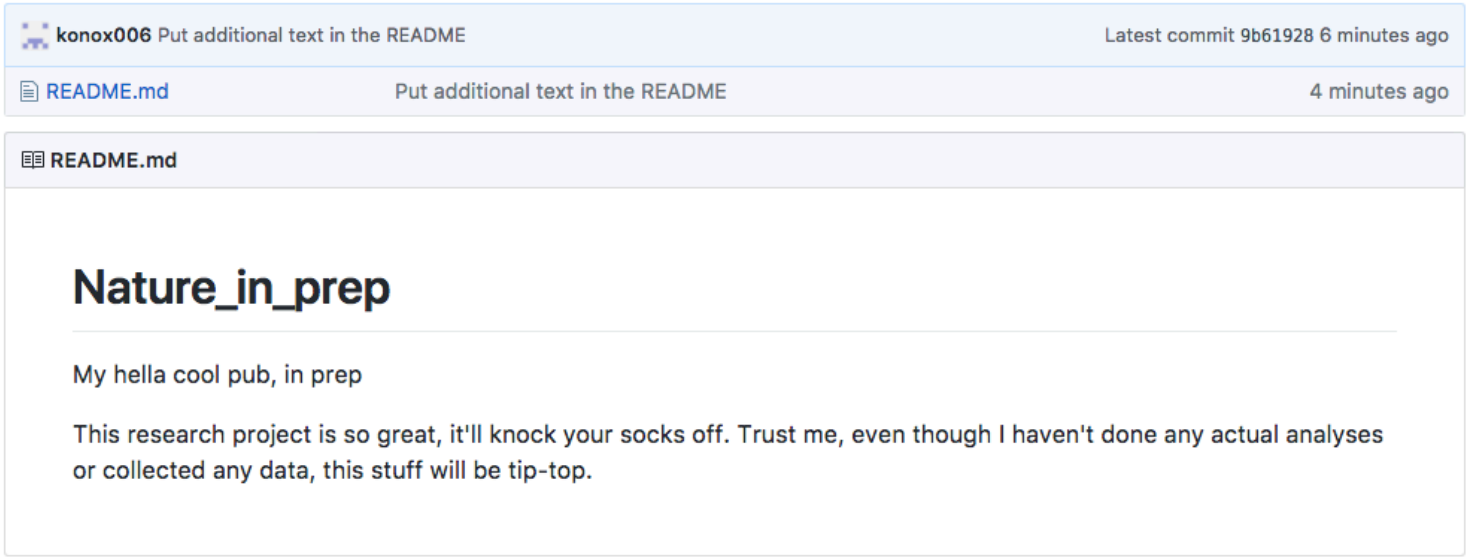
Part 3: Let's Use UMN GitHub

- Now, we are ready to `push` the changes to the repository

```
[23:02:04] [tomkono@Toms-MacBook-Pro] (~/Nature_in_prep) $ git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 422 bytes | 422.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.umn.edu/konox006/Nature_in_prep.git
   2a8a8b0..9b61928  master -> master
[23:02:07] [tomkono@Toms-MacBook-Pro] (~/Nature_in_prep) $
```

Part 3: Let's Use UMN GitHub

- The new commit will appear on the repo website

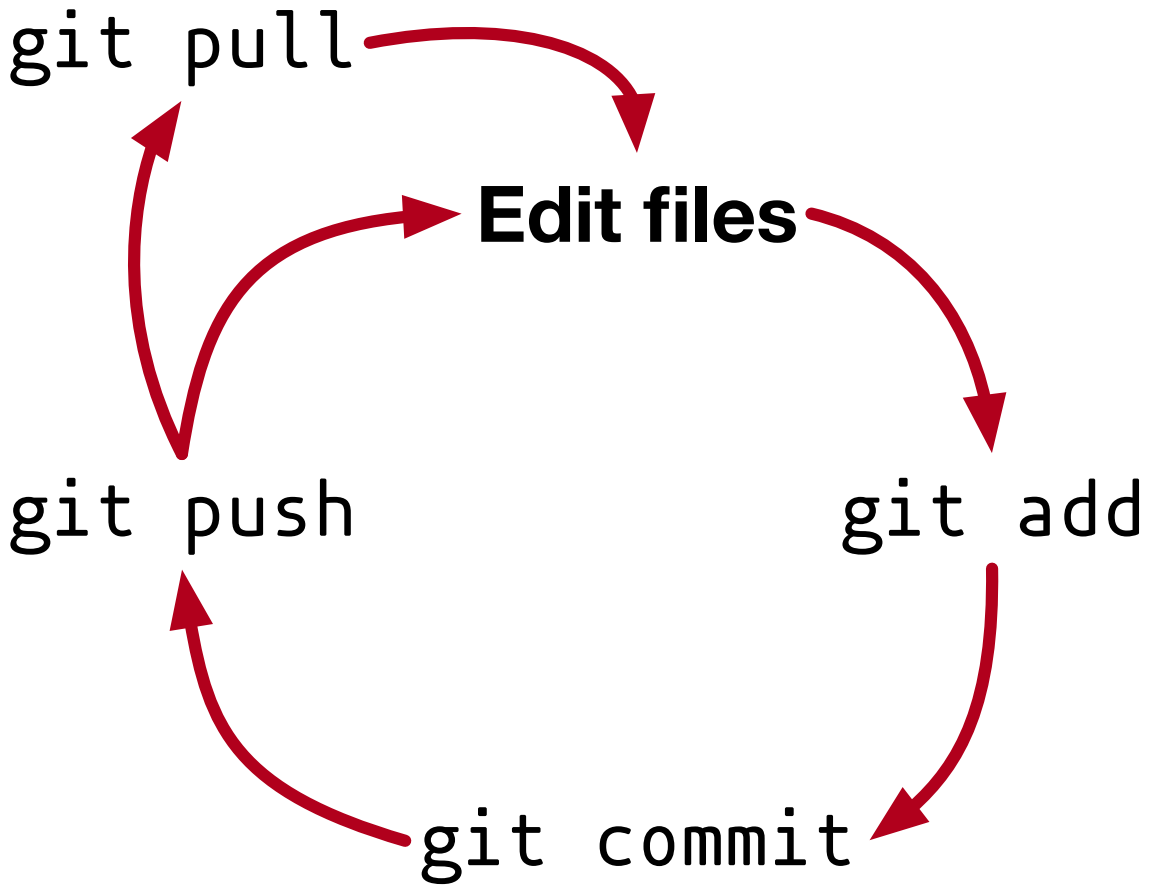


Part 3: Let's Use UMN GitHub

- When working with others, be sure to `pull` remote changes before editing files, else you'll create a "merge conflict" which must be **manually resolved.**
- **This is painful. Avoid it.**
- **I had one with LaTeX and I just burned the repo and re-cloned it**

Part 3: Let's Use UMN GitHub

- Put your own script into the repo, and go through `add`/`commit`/`push`



Part 3: Git Resources

- GitHub cheat sheet
  https://education.github.com/git-cheat-sheet-education.pdf
- Interactive git tutorial
  https://try.github.io/
- Vince Buffalo's book (Chapter 5)
  http://vincebuffalo.org/book/

Thanks! Questions?

- What can I do better?