# COMPUTING SERVERS AND OTHER TOPICS

ANDREW BLANDINO

## 1. Computing Server Basics

The department computing servers are a powerful resource during your tenure here in the statistics program. But it is also important to keep in mind theses words of wisdom concerning great power:

With great power comes great responsibility. – Uncle Ben, *Spiderman*

### 1.1. What are the computing servers?

In overly simplistic terms, one can think of the department servers as very expensive computers with bigger/faster/(stronger?) components. In particular,

(1) Larger Storage space (hard-drive/ long-term memory)
(2) Larger RAM (short term memory)
(3) More Cores / CPUs (processing power)

The specs and details of the computing servers are on the department webpage (c.f. https://statistics.ucdavis.edu/resources/computing-resources/servers)

### 1.2. How do I use the computing servers?

First, you need to get access from Nehad who will setup your account login and password that is the same for all statistic servers. You can then log-in using the instructions specified on the department systems page (c.f. https://anson.ucdavis.edu/systems/).
VPN: if you are on campus then you can just .ssh into any of the department servers. However, if you are off-campus then you will need to be logged onto the UC Davis Library VPN in order to .ssh in. See https://www.library.ucdavis.edu/service/connect-from-off-campus/ for more details.

### 1.3. Why do I use the computing servers?

In principal, anytime you need more Storage Space, RAM, and/or CPU Cores than what your current Laptop/PC provides. The most common bottleneck of a program is not enough CPUs or slow processing power. The server comes in handy here because we can take advantage of the large number of CPU cores available and run jobs in **parallel**.

*Date*: December 2, 2019.

1.3.1. *What does it mean to run a job in **parallel**?* In simple terms, it is running a program such that a large number of cpus can be utilized simultaneously in order to speed up computation time. The most common example is when a loop statement is to be executed a large number of times but each iteration takes a significant amount of time.

- Need to execute an expensive code chunk $N$ times.
- Without parallel: total run-time is $N \times T$, where $T =$ run time of one code chunk.
- Parallel: have $B$ CPU's compute $N/B$ code chunk simultaneously (assuming $B$ divides $N$)
- *new* run-time is $\frac{N}{B} \times T$ (assuming $T$ is the same).
- Obviously, larger $B$ results in shorter run-time.

When each iteration of the loop is independent of each other, we can speed things up by having several CPU's run a subset.

1.4. **Different ways of running code in parallel.** There are many ways of running the same piece of code in parallel, especially if we include with programming language to use. Here's a couple examples to get started, but there are more out there.

1.4.1. *Bash scripting.* This is the scripting language on the command line on the department servers. We can make our code parallel by writing a bash script to execute our script $B$ times (number of workers / parallel processes). There is a lot of power and control with this method but it does require more initial time investment.

1.4.2. *Built-in functionality.* A lof of programming languages have parallel functionality that is easily implemented. Typically, you can specify how many parallel workers/processes you wish to run within the script and it will execute in parallel.

- `R`: a lot of different options via packages e.g. `parallel`, doMC, `foreach`
- `Matlab`: `parfor(...)` attempts to run a for loop in parallel analyzing the code, etc..

This may be the preferred way if it we obtain a run-time fast enough without doing too much extra scripting.

1.5. **Server usage & philosophy.** The stats department servers follow a *laissez-faire* philosophy: anyone can run a job at any time regardless of what other activity there is on the server. This presents two possibilities:

- Anyone can quickly test parallel code and get results (without the overhead of job-scheduling).
- Anyone can **overload** the server, making everyone's job run slower. E.g. Executing a job that uses 50 cores with only 32 total possible.

1.5.1. *How can I run my jobs responsibly?*

- See the current load on the server: the scripts `top` and `nps` (custom script on stats servers) are useful here.

- Manage your jobs: use the previous scripts to see the usage of your current job(s). It is also good practice to `kill` any jobs that take too long, mismanage resources, etc.
- Check the server status page: `https://anson.ucdavis.edu/systems/` (have to log-in CAS), and see what severs are not in-use (small load %)

## 2. Useful extras for using the Server

2.0.1. *screen:* a very useful command that allows separate "screens" while connected to the server. The reason for this is the following scenario: if you log-out from the server with a job running in terminal then the job also terminates, which is useless for long jobs. But if you have a job running on a separate *screen*, you can safely disconnect from the server without canceling your job.

2.1. **File manager (FTP).** In order to get any use out of the servers we will typically need to upload files/data/scripts to the server. This can be done in terminal using `scp` to copy files from the server to your computer, and vice versa. But luckily there is (free) software that streamlines this process into a convenient GUI

- `forklift`: for MAC `https://binarynights.com/`
- `WinSCP`: for Windows `https://winscp.net/eng/download.php`

2.2. **Git.** The computing servers do not provide **any backup** of files stored on it. So it is important to maintain your own back-up of any important files you upload. One option for this is via `git`, which is primarily used for software development / version control. What `git` does is create a repository for your code, including all changes, updates, etc.. Note only does this backup your code in it's current form, but it also keeps a history of all changes to your code.

Some popular websites to host your repositories:

- `github.com`: very popular.
- `gitlab.com`: another option.

To see an example of a git repository see `https://github.com/AndoBlando/STA390`.

## 3. Gauss & other job scheduling computing clusters

3.1. **Logging onto Gauss/Peloton.** There is a little more work involved for logging onto the Gauss computing cluster, but the details of it are in the following link `https://anson.ucdavis.edu/systems/sshkey/`. The main difference is that an encrypted key is your 'password' on your computer, instead of using a personal password each time you login. Both require you to register an account with their respective webforms:

- **Gauss:** `https://wiki.cse.ucdavis.edu/cgi-bin/index2.pl`
- **Peloton:** `https://wiki.cse.ucdavis.edu/cgi-bin/mps.pl`

3.2. **SLURM Job Scheduling.** The Gauss & Peloton servers use the SLURM Job Scheduler to run computing jobs on the Cluster. Essentially, it assigns processes to nodes in the network based on different job criteria. From our point of view as a user of this job scheduler, we only need to focus on a few details to get a job scheduled:

- Job Size / Number of CPU's needed
- RAM/ Memory Needed

There is more we could specify but this is the bare minimum to a get a job up and running with Slurm.

**Note:** Slurm does not check your Jobs for accuracy. Therefore, it is important for you to check that your Job only uses the resources allocated towards it.

For more details on how to submit jobs and more, checkout out the following wiki `https://wiki.cse.ucdavis.edu/support/systems/gauss`.