



F. Wermelinger
Office: Pierce Hall 211

Lab 1

Accessing the cluster, compiling code and submitting jobs

Issued: January 30, 2023 Due: February 10, 2023 11:59pm
--

This lab is a bit lengthy in text. There are only few work items which you can solve as you read along the lab instructions. The main purpose of this lab is to give you an introduction to the Harvard academic cluster maintained at FASRC. The following instructions are further intended to serve as a reference for your work on the cluster. The academic cluster will be our main computing resource for the class. The job scheduling system we use is the same as on the Cannon cluster¹ which translates to an identical workflow if you already have access to Cannon.

Submission instructions: please see the lab submission section in the syllabus.^a

^a<https://harvard-iacs.github.io/2023-CS205/pages/syllabus.html#lab-submission>

Task 1: Accessing the Academic Cluster

In this task we will focus on how to access the compute cluster that we will use throughout the CS205 class. This task explains the core computational resources that are at your disposal. You should be familiar with these as we will need them for the homework, labs and project. A summary of a few frequent keywords can be found here

<https://www.rc.fas.harvard.edu/services/cluster-computing>

The user guide for the FAS OnDemand tool can be found at the following link, please make sure you understand the “First time launch”, “A brief tour” and “Storage” sections in that guide

<https://fasrc.github.io/fas-ondemand-user-guide>

(Most of the content in these sections is covered in task 1a below.)

- a) You can access the compute cluster through the FAS OnDemand tool on Canvas (see navigation panel on the left in the browser) or with the following link: https://canvas.harvard.edu/courses/113711/external_tools/93631?display=borderless. Doing so will open a new tab in your browser that initializes your dashboard. When you do this the first time it may take a few minutes.

¹<https://www.rc.fas.harvard.edu/about/cluster-architecture>

In the top left corner (“Tools” drop-down) or at the very bottom of the dashboard there are a few tools for

- uploading or downloading data to/from your home directory on the cluster.
- composing job submission scripts.
- check your currently active jobs which are either queued or running on the cluster.
- obtain shell access to a login node, your main interface for work on the cluster.

Click now on “FAS-OnDemand Shell Access”. This will bring up a new tab in your browser with a terminal session on one of the *login* nodes (see task 2a for the distinction between login and compute nodes).

The quota in your home directory is 20 GB which should be used to store code and configuration data. There are two default directories in your home directory:

scratch_folder is a directory used to run your simulation and test cases. You should run your code in this directory because it is mounted on a high performance Lustre² file system with a large 10 TB quota that can accommodate the I/O data generated by your code. The data in the scratch file system is considered *temporary* but will be preserved for the duration of the class. (On typical supercomputing systems the files in the scratch folder will be **deleted without notice after 30 days**. On the Cannon cluster at Harvard it is 90 days, see the scratch policy for further information: <https://docs.rc.fas.harvard.edu/kb/policy-scratch>.) The path to your scratch folder is accessible through the MYSCRATCH environment variable.

shared_data is a directory that is shared with the class. The directory may contain programs, libraries or other data associated with homework or labs that you will need. We can also use it to share data with others or for your projects. The path to this directory is accessible through the SHARED_DATA environment variable as well (for scripting).

You should not remove the two directories. You can check your current quota on the scratch file system using the command

```
$ lfs quota -h $MYSCRATCH
```

Note here and in the following, the leading “\$” in commands indicates the prompt, you do not need to type it when you enter the commands.

The environment on the academic cluster is Linux, take a few moments to navigate around in your home directory to get comfortable. You may want to customize your .bashrc file to improve your user experience. Append your customization at the end of the file. Some of the resources listed at

<https://harvard-iacs.github.io/2023-CS205/pages/resources.html#general> might be helpful to refresh some shell and command line concepts.

- b) There is a vast amount of software available on the cluster through *modules* that are managed by the lmod³ system on the cluster. You usually have to load them yourself

²<https://www.lustre.org>

³<https://lmod.readthedocs.io/en/latest/index.html>

and you often want to place these load commands in your job submission scripts as your code might depend on third party libraries (submission scripts are simple bash scripts, see task 3 below). For a modules introduction on the Harvard academic cluster see <https://docs.rc.fas.harvard.edu/kb/modules-intro>.

The modules you will need most for CS205 are a recent compiler and a recent MPI library. Here are a few examples:

- gcc/12.1.0-fasrc01 — GCC compiler suite version 12.1.0
- intel/21.2.0-fasrc01 — Intel compiler suite version 21.2.0
- openmpi/4.1.3-fasrc01 — OpenMPI implementation 4.1.3

Note that there are dependencies among software modules. For example, you can only load openmpi/4.1.3-fasrc01 if you have loaded one of the two listed compilers before.

Whenever you need to search for a module you can use

```
$ module-query <search pattern>
```

This command behaves similar as `module spider` but is faster.

If you want to load the GCC compiler and OpenMPI implementation shown above, simply invoke this command

```
$ module load gcc/12.1.0-fasrc01 openmpi/4.1.3-fasrc01
```

Alternatively you could add this line to your `.bashrc` file if you want to make it persistent (**note:** loading software modules on login nodes takes some time, later we will do this on interactive nodes). Try the above command in the shell session you have started earlier. When you have loaded the two modules, ensure that your `PATH` variable is correctly set by testing

```
$ which gcc
/n/helmod/apps/centos7/Core/gcc/12.1.0-fasrc01/bin/gcc
$ gcc --version
gcc (GCC) 12.1.0
Copyright (C) 2022 Free Software Foundation, Inc.
```

You can list the modules you have currently loaded with

```
$ module list
Currently Loaded Modules:
  1) gmp/6.2.1-fasrc01    3) mpc/1.2.1-fasrc01    5) openmpi/4.1.3-fasrc01
  2) mpfr/4.1.0-fasrc01  4) gcc/12.1.0-fasrc01
```

To purge all your loaded modules and restore the initial environment you can invoke

```
$ module purge
```

- c) It might be a good idea to setup a private git repository for your CS205 class work. It is up to you how you want to structure your workflow. Homework and lab submissions are done through Gradescope^{4,5}. A git repository would be a good place to host your personal work progress. You can then just create a zip archive of the corresponding assignment and submit it on Gradescope before due date. You can create a private repository at <https://code.harvard.edu> in your profile using your HarvardKey.

Please do not create your repository in the CS205 organization you have been added to, we will clean up data/repositories in there after the semester and you would need to setup a new remote (assuming you have a local clone).

In order to access the repository from the compute cluster you need to upload your *public SSH key* from the academic cluster to your remote repository host (<https://code.harvard.edu>). You want to copy the *public* SSH key in your `$HOME/.ssh` directory on the compute cluster to either your account settings on <https://code.harvard.edu> (recommended and also useful if you plan on using multiple repositories) or set it up as a deploy key that is only valid for the repository you have just created above. You can display your public RSA key on the cluster with this command

```
$ cat $HOME/.ssh/id_rsa.pub
```

Copy the output to your clipboard.

Important: Do not modify, overwrite or delete the files in your `$HOME/.ssh` directory (the files are required to access the cluster from Canvas and are created when you login for the first time). *Any modification may lead to login denial on the FASRC resources.*

To setup the SSH key globally on <https://code.harvard.edu> (recommended as we will need it in task 2):

1. Click on your user avatar in the top right of the screen while logged in to <https://code.harvard.edu>, then click on “Settings”.
2. Click on “SSH and GPG keys” in the left panel.
3. Click on the green “New SSH key” button.
4. Give your new key a suitable name and paste the public key you have just copied into the large box then save it.

Next, load a recent git version on the cluster

```
$ module load git/2.17.0-fasrc01
```

Now you can clone your repository from <https://code.harvard.edu> using the SSH protocol and work on homework code or add data from the cluster directly via git.

⁴HW: <https://harvard-iacs.github.io/2023-CS205/pages/syllabus.html#homework-submission>

⁵Lab: <https://harvard-iacs.github.io/2023-CS205/pages/syllabus.html#lab-submission>

Finally, if you need python for your work in CS205 (e. g., post-processing and plotting data) you should make sure to use python3 when you launch your scripts. The default version on the cluster is python2.7.⁶ You can load a more recent python version with

```
$ module load python/3.9.12-fasrc01
```

Task 2: Compute Architecture and Compiling Code

We now learned how to access the cluster, load software modules and have established the credentials to work with `git` within the Harvard computing infrastructure. In this task we will focus on how to compile code on our cluster.

- a) Before we actually compile code, we should first understand the two main node types on the cluster.

Login node: A login node is the landing point when you access the cluster. These nodes are usually lightweight in terms of computing resources but more importantly host all the other users currently working on the cluster. It is a multi-user Linux environment in essence, you can check for users logged in using this command

```
$ users
```

Because compiling code can be a resource hungry task, you should generally avoid compiling code on login nodes as other users will get upset when the response is lagging because you consume all of the CPU. On login nodes you usually perform administrative tasks like writing code in a `vim` instance, commit changes to your `git` repositories or managing your data.

Compute nodes: These are the workhorse nodes where you perform the computation. In the class we will see how we can distribute a program on many such compute nodes. You can checkout compute nodes exclusively for yourself and not worry about other users. Consequently, compilation should be carried out on compute nodes as well. In order to schedule the execution of a compute job, a *scheduler* is required. In task 3 of this lab we are looking at SLURM⁷ which is responsible for the scheduling of jobs on the cluster.

Whenever you work on the cluster, the first thing you should do is checkout a compute node to work on (*unless you are using a VSCode instance from Canvas, then you already are on a compute node*). The login nodes are very slow and should be avoided to do work (you will experience lag). Note that a compute node is just another "computer". You will always "see" the same files, no matter whether you are on a login node or compute node (the latter is just a better and faster "computer" connected to the same file system). You will see in task 2b below how to checkout a compute node.

⁶It is important to use stable operating systems on such platforms and they are usually behind with software releases.

⁷<https://slurm.schedmd.com/overview.html>

Tip

Often you need multiple terminals open at the same time when working at a remote site. For example, you might want to have an open session on a login node and simultaneously checkout an interactive session on a compute node to perform some work. You will need two terminal sessions for this. Instead of opening two browser tabs, a faster way is using `tmux`⁸. `tmux` is a terminal multiplexer which you can start on the login node, detach from it, close the current connection to the cluster and return to it at a later time when you reconnect. `tmux` will keep running on the login node while you are not there (provided the login node does not crash in the meantime and you login to the same login node). This is very useful when you have something running in the shell that you want to keep running while disconnected from the machine. *Most important:* when you get disconnected from your network connection due to various reasons, you will need to create a new shell session in the browser and if you were working with `tmux` you can just attach to it again as if nothing happened. You can load `tmux` with

```
$ module load tmux/2.7-fasrc01
```

- b) It is also important to get an idea of the node architecture you are working on before you run compute jobs on it. This may influence our decision on how we ultimately map our parallel application to the compute nodes. Now is a good time to do a little work and investigate what our compute nodes look like.

The code and documents we are going to work with in the labs are located in the main class git repository at <https://code.harvard.edu/CS205/main/tree/master/lab>. You can clone this repository in your cluster home directory and then easily pull new content for later labs at the given time. In order for this to work you must have setup your SSH key in your <https://code.harvard.edu> account (see task 1c) and you must be a member of the CS205 organization on the same platform, see <https://harvard-iacs.github.io/2023-CS205/#important>. In the following it is assumed you have executed the command below (you may adjust the local repository name according to your liking)

```
$ git clone git@code.harvard.edu:CS205/main.git $HOME/cs205_class
```

It is further assumed here you have created a private git repository in task 1c and a clone of it is located in `$HOME/cs205` (the naming may vary depending on your preferences). You can now copy the documents for this lab into your private repository

```
$ mkdir -p $HOME/cs205/lab
$ cp -r $HOME/cs205_class/lab/lab1 $HOME/cs205/lab/
```

⁸<https://www.ocf.berkeley.edu/~ckuehl/tmux>

Note: Instead of working with two repositories and selectively copying content into your private work repository, you could also create a fork of <https://code.harvard.edu/CS205/main> and work with branches in one repository instead. This is more likely to result in merge conflicts however.

The `cp` command above copied the `lab1` directory from the class repository into your own work repository. You must complete this directory with your solution and submit it on Gradescope as a zip archive.

For the following questions, write your answers below the heading `# Task 2b` in the markdown file

```
$ echo '# Task 2b' >>$HOME/cs205/lab/lab1/answers.md
```

We are now going to request an *interactive* compute node. Interactive means that you will be dropped in a shell on the node which allows you to enter commands interactively. Compare this to a job submission (discussed in task 3) where we are submitting a job script to the scheduler, which then automatically launches the application on the compute nodes when the resources are available. We usually do such job submissions from a login node although they can also be carried out from an interactive compute node. First let us check what is the host name of our current (login) node

```
$ hostname # your output may look different
academic-login07.rc.fas.harvard.edu
```

We now request one (interactive) compute node for one hour

```
$ salloc -N1 -t 1:00:00
```

The option `-N1` requests one node and `-t hours:minutes:seconds` specifies the duration you want to work with the resource. Once the resource is available we will be dropped in a shell (depending on how many users run jobs on the cluster at the time). This is a new instance on another node or “machine” where we will have again control of the prompt. Verify that you are on another host with

```
$ hostname # your output may look different
holy2a04208.rc.fas.harvard.edu
```

You are now on an interactive compute node with the requested resources allocated exclusively for you. Open another shell session with a login node (or try out `tmux`) and check your job queue on the cluster to verify that the compute node allocation went through the job scheduler. Type

```
$ squeue -u $(whoami)
```

JOBID	PARTITION	NAME	USER	ST	TIME	NODES	NODELIST(Reason)
83042	academic	interact	u_399544	R	0:46	1	holy2a04208

Check the JOB STATE CODES and EXAMPLES sections in `man squeue` and take a moment to analyze the output. You can list all your jobs this way (some may be pending others may be running).

Return to your interactive compute node session. We are interested in what type of CPUs are available on our node. You can get information about CPUs on a Linux system by looking at the file `/proc/cpuinfo`. Either open this file in an editor of your choice or run

```
$ cat /proc/cpuinfo | less
```

Alternatively you can examine the command

```
$ lscpu
```

Please work on the following subtasks

- i) What is the manufacturer and model name of the CPU on the compute node? Try to find this information in `/proc/cpuinfo` and use a google search to find additional information if needed. How many cores are available to you on the compute node? Write your answer in `$HOME/cs205/lab/lab1/answers.md`.

Interpreting the `/proc/cpuinfo` text file can be cumbersome and requires some practice. A very useful tool to help identifying a node architecture is the Portable Hardware Locality (`hwloc`) software package⁹. Spend a few moments on the `hwloc` webpage given in the footnote. In essence, the tool can be used to generate a graphical view of the machine architecture, you could also use it on your laptop. You can find a pre-compiled executable in `$SHARED_DATA/local/bin`. There will be other tools made available in this location and you should add the path to your `.bashrc`. To do so, run the following commands

```
$ $SHARED_DATA/local/install.sh
$ source $HOME/.bashrc
```

- ii) Run the following command after you have installed the path into your environment (the backslash “\” means continuation on the next line)

```
$ lstopo --whole-system --no-io --no-icaches \
    $HOME/cs205/lab/lab1/c1.png
```

This will create a PNG file in `$HOME/cs205/lab/lab1/c1.png`. You can open that file in the browser by clicking on “Home Directory” in your tools sections in the Canvas FAS OnDemand dashboard. Navigate to the `c1.png` and click on it. Please answer the following questions and write your answers again in `$HOME/cs205/lab/lab1`

- How many physical cores are on *one* CPU?
- How many CPUs are on your node?

⁹<https://www.open-mpi.org/projects/hwloc>

- How much memory does each core have access to?
- What are the L1, L2 and L3 cache sizes?
- Are there any caches *shared* among CPU cores?
- Some cores are marked red, why do you think that is?

Press Ctrl-D or type exit to leave the interactive compute node. Now we are going to checkout another interactive compute node using the following command

```
$ salloc -N1 -c32 -t 1:00:00
```

This time we specify the option -c32 which tells the scheduler that we are requesting one node with 32 cores instead of just one by default.

Note: the compute cluster has two types of nodes with different hardware configurations. These are the nodes available:

- 32 nodes with 16 core CPUs in dual socket configuration (default node)
- 16 nodes with 14 core CPUs in dual socket configuration

If you request a node with the command `salloc -N1 -t 1:00:00` and all of the 32 default nodes are busy, you may get one of the nodes with 14 core CPUs. If you are more specific and request resources with `salloc -N1 -c32 -t 1:00:00` (one node and 32 cores), this will map exactly to one of the 32 default nodes. If they are all busy and you use the latter resource request, you will have to wait until one of the nodes becomes available.

- iii) When you are given back control of the prompt, run the following command (use the up-arrow key or Ctrl-P to cycle through the command history)

```
$ lstopo --whole-system --no-io --no-icaches \
  $HOME/cs205/lab/lab1/c32.png
```

Look at the c32.png this time. Briefly describe the differences you observe between c1.png and c32.png in your `$HOME/cs205/lab/lab1/answers.md` file.

The compute node you have just investigated is a non-uniform memory access (NUMA) node, something common in high performance computing systems. We will further discuss this type of node in class.

Finally make sure you commit your answers and data for this task

```
$ cd $HOME/cs205
$ git add lab/lab1
$ git commit -vm 'Lab1: completed task 2b'
```

- c) We are now going to compile a small test code that helps us understand the mapping of threads and nodes. It is not important that you fully comprehend the code, we will talk about these concepts in the class. Here we just need a code to play around with. The code we are going to work with is already located in your private `$HOME/cs205` repository that we prepared in task 2b. Change to the following directory in your interactive compute node

```
$ cd $HOME/cs205/lab/lab1/code/t2
```

Spend a few moments to study `hybrid_node.cpp` (do not spend too much time as we have not yet discussed any of the concepts used in the code). The code is a hybrid MPI/OpenMP code that prints the thread ID's on each node it is running on. In order to compile this code we need to load a MPI library. In your *interactive compute node session* load the following modules:

```
$ module load gcc/12.1.0-fasrc01 openmpi/4.1.3-fasrc01
```

Because the code makes use of MPI we need to *link* to the MPI library in the last phase of compilation. The MPI implementation we loaded above provides a convenience compiler wrapper for this task called `mpic++` for C++ code (and `mpicc` for C code or `mpif90` for Fortran90 code). Navigate your interactive compute node to `$HOME/cs205/lab/lab1/code/t2` if you have not done so already.

In this next step we are going to compile the code in the interactive session, make sure you have loaded the modules above in that terminal. Execute the command

```
$ mpic++ -std=c++11 -fopenmp -o hybrid_node hybrid_node.cpp
```

The meaning of the arguments for the compilation are:

- Use the C++11 standard¹⁰ (`-std=c++11`).
- Use the compiler built-in OpenMP library for multi-threaded execution (`-fopenmp`).
- Emit the executable `hybrid_node` (`-o hybrid_node`).
- Use the source code `hybrid_node.cpp` to build the executable.

Run the following command a few times

```
$ srun hybrid_node
```

and spend a moment to interpret the output. Make sure it makes sense; discuss with a TF or a fellow student if you find something strange or unclear.

Of course, if your code does not depend on MPI we could just use `g++` to compile the code. Try this command quickly

```
$ g++ -std=c++11 -fopenmp hybrid_node.cpp
```

Try to understand what went wrong by studying the error messages from the compiler (they are related to MPI). You can check what `mpic++` actually wraps around using the command

```
$ mpic++ -show
```

It is these additional options that allow to compile the MPI code without errors.

¹⁰<https://isocpp.org/wiki/faq/cpp11>

Task 3: SLURM

From the SLURM homepage¹¹:

Slurm is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters. (...) As a cluster workload manager, Slurm has three key functions. 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.

The documentation for SLURM can be found at this link <https://slurm.schedmd.com/documentation.html>.

a) Most often you want to submit a job which can be done in the following way

```
$ sbatch script.sh
```

Here script.sh is a shell script that we are going to look at in task 3b. If you want to test a submission without scheduling the job, you can use the `--test-only` option

```
$ sbatch --test-only script.sh
```

To see what jobs you have queued you can use

```
$ squeue -u $(whoami)
```

You can also add the `-t RUNNING` or `-t PENDING` options to split up the output further. If you want to cancel *all* your jobs (including running jobs) you can use

```
$ scancel -u $(whoami)
```

or

```
$ scancel -u $(whoami) -t PENDING
```

if you only want to cancel jobs that are pending.

You can find more information for any Linux command by visiting the man page. Spend a few moments to look at

```
$ man sbatch
$ man squeue
$ man scancel
```

¹¹<https://slurm.schedmd.com/overview.html>

Some more examples for useful commands can be found at this link <https://docs.rc.fas.harvard.edu/kb/convenient-slurm-commands>.

- b) In task 2c we have compiled our `hybrid_node` and tested it using a single node. We now want to run this application using 4 nodes, each with 32 cores such that our application will run on 128 cores in total (your laptop likely has 4 to 8 cores).

Whenever we are going to run jobs, it is best practice to write the configuration of *how* we ran that job in a script file. This allows us to rerun the job at a later time with ease and we have the job configuration *archived* at the same time. The need to rerun a job occurs more often than you probably think right now, having a submission script is important and good practice. The submission script in this command

```
$ sbatch script.sh
```

is `script.sh`. You can name it however you like of course.

You may perform the following on a login node or continue on your interactive node. Change to

```
$ cd $HOME/cs205/lab/lab1/code/t2
```

where we compiled `hybrid_node` in task 2c. Create a file `job.sh` with the following content (you may omit comments)

```
1  #!/usr/bin/env bash
2  #SBATCH --job-name=hybrid_4nodes
3  #SBATCH --output=hybrid_4nodes_%j.out
4  #SBATCH --error=hybrid_4nodes_%j.err
5  #SBATCH --nodes=4
6  #SBATCH --ntasks-per-node=1
7  #SBATCH --cpus-per-task=32
8  #SBATCH --time=00:30:00
9
10 # set the number of OpenMP threads per MPI process
11 # (more on this in class later)
12 export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
13
14 # here we run our program: see man srun
15 #
16 # * srun is the SLURM runner similar to mpirun (more on this in class)
17 # * we also add "$@" here to forward any arguments to our program that
18 #   were passed to the job script at submission time.
19 # * note that the submission script assumes that the hybrid_node program
20 #   is in the same directory (./)
21 srun ./hybrid_node "$@"
22
```

```
23 # return the exit code of srun above
24 exit $?
```

SLURM allows for a special syntax in your job scripts submitted with sbatch. Every line beginning with #SBATCH is interpreted as an option that is passed to sbatch. This is great for self-containment. Note that the #SBATCH statements must be located at the beginning of the file, after a possible shebang and before any shell command. The scripted options can be overwritten when passed directly in the command line to sbatch. Here is a short description of the options:

- job-name:** a string to specify the name of your job as seen with squeue for example. You are free to choose a meaningful name here.
- output/error:** file names to redirect stdout and stderr streams for the job. These are important files for your job analysis. The %j placeholder will be replaced with the job ID at the time the job starts running. This is useful when you rerun the job such that previous files will not be overwritten. A job ID is always a unique number. If you only specify --output then stderr will be redirected to that file as well.
- nodes:** the number of compute nodes you request.
- ntasks-per-core:** the number of MPI processes per compute core (we do not need it here in this example but it is a useful option you should keep in mind).
- ntasks-per-node:** the number of MPI processes per compute node.
- cpus-per-task:** the number of CPU cores per MPI process. (We have investigated the node architecture in task 2b.)
- time:** the wall time we request the resources for. Be aware that if you estimate too small wall time and your job has not finished by that time, it will be forcefully terminated. This is the same option as we already saw with `salloc -N1 -t 1:00:00`.

If you want to be notified by email when your job ends, you can add the following two lines after line 8 in the `job.sh` script above

```
#SBATCH --mail-user=<your email address>
#SBATCH --mail-type=END
```

You can now submit your job to the queue by running

```
$ sbatch job.sh
```

Your job may not run immediately depending on the availability of the resources. You can always check the status of your jobs with (on a login node)

```
$ squeue -u $(whoami)
```

After your job has completed, have a look at the `hybrid_4nodes_*.out` file. How many lines do you count in the file? How many different host names can you spot? Your `hybrid_4nodes_*.err` file should be empty.

Finally make sure you commit your files and data for this task

```
$ cd $HOME/cs205
$ git add lab/lab1/code
$ git commit -vm 'Lab1: WIP task 3b (*.out, *.err job files missing)'
```

Note that at this point the *.out and *.err log files might be missing because the job has not ran yet. Use the email option above to get notified when the job has completed and you may add these files in a second completion commit. You are asked to submit your job.sh script and all the files that are output in \$HOME/cs205/lab/lab1/code/t2 by your job submission above.

For your submission on Gradescope, you can create a compressed zip archive with

```
$ cd $HOME/cs205/lab
$ zip -r lab1.zip lab1
```

If you do this on the cluster, you can download the archive via the “Home Directory” tool in your dashboard in order to submit on Gradescope.