Introduction to R-based High Performance Computing at MIT for Political Science*

March 23, 2015

1 Introduction

This tutorial introduces ENGAGING1, a computing cluster for high performance computing. ENGAGING1 is a collaborative project run by several universities in the Boston area. The project PI at MIT is Chris Hill. You can find out more about the ENGAGING1 cluster from the Facility Notes, maintained by Chris.[1]

This tutorial walks a user through the steps required to run a simple R script on ENGAGING1. Some of the steps will be familiar to users of computer clusters such as ATHENA. Experienced users may wish to jump ahead to Section 6, which discusses how to interact with the Slurm scheduler.

It will be necessary to have some level of comfort with the command line when using the cluster. This tutorial introduces some basic commands that are commonly used.[2] Readers are assumed to have working knowledge of R.

1.1 Hardware

ENGAGING1 consists of several head nodes, 234 compute nodes and a 350TB central Lustre storage system. Each node has two 8-core (=16 cores), 2GHz Intel Xeon E2650 processors, 64GB of memory and 3.5TB of local disk.

1.2 Getting access to the ENGAGING1 cluster

Please speak to In Song.

2 Running example

We use the following running example in this tutorial. Suppose we want to run an ordered probit model on a dataset. The data are grouped by countries, and we believe that errors are correlated within countries. To get cluster robust standard errors, we can do block bootstrap in R:

```
### Preamble
setwd("~/projects/demo")
require(MASS)

### Analysis
d <- read.csv("mydata.csv")
polr.formula <- factor(redist) ~ finan + incbins + female + age + partyid + educ
```

*Prepared by Weihuang Wong, `wwong@mit.edu`. Portions of the setup script were adapted from Jonathan Olmsted's setup scripts for R-based HPC at Princeton University.

[1] `https://github.com/christophernhill/engaging1/blob/master/facility_notes`

[2] It is easy to find online tutorials by Googling "`linux command line tutorial`".

```
### Bootstrap
dobootstrap <- function(...) {
    id <- unlist(tapply(d$ID, d$ccode, function(i) sample(i, length(i), replace=TRUE)))
    D <- d[match(id, d$ID),]
    fit <- polr(polr.formula, data=D)
    c(fit$coefficients, fit$zeta)
}
```

We run our code:

```
> system.time(polr.out <- sapply(1:1000, dobootstrap))
    user   system  elapsed
1260.257  159.268 1435.657
```

and see that it takes almost 24 minutes, or approximately the running time of a Simpsons episode. Because we can only watch so much Simpsons, it would be ideal to find a faster way to run the code. Parallelizing it on our local machine (or on Athena) can help, but we can do much better if we have access to a HPC cluster, because then we can parallelize our work over many more processors across multiple nodes (machines).

3 Connecting to the engaging1 cluster and initial setup

3.1 Connecting to engaging1

Once you have a username and password, you can login to engaging1 via ssh.[3] Mac or Linux OS users can open up a terminal window:

```
Cueballs-MacBook:~ cueball$
```

where your local username will take the place of cueball.[4] Then, type the following:

```
Cueballs-MacBook:~ cueball$ ssh student001_17806@eofe4.mit.edu
```

where student001_17806 should be replaced by your engaging1 username. eofe4.mit.edu refers to the engaging1 server address.

Windows users will need to install an ssh utility. MIT IST recommends SecureCRT, which is available for download at the IST website.[5] In SecureCRT, start a new session, and type eofe4.mit.edu in the Hostname box.

You will be prompted to enter your password. You will not be able to see the characters you type in (there is no visual feedback), but if you have entered your password correctly, you will be logged in and see the following prompt:

```
[student001_17806@eofe4 ~]$
```

3.2 Initial setup

Your engaging1 account is not automatically configured to work with R, so we need to do some initial setup. At the prompt, type in the following:

```
[student001_17806@eofe4 ~]$ git clone https://github.com/insongkim/eofe-scripts.git
```

This command (git clone ...) downloads a set of setup scripts, as well as example code for use on engaging1, from a GitHub repository, into a directory named eofe-scripts. You can see that the directory has been created by listing the items in your filespace using the ls command:

[3] You can also submit your SSH key to the engaging1 system administrator for password-less access.
[4] http://www.explainxkcd.com/wiki/index.php/Cueball
[5] http://ist.mit.edu/securecrt-fx/

```
[student001_17806@eofe4 ~]$ ls
eofe-scripts
[student001_17806@eofe4 ~]$
```

Now, navigate to the `eofe-scripts` directory using the `cd` command.[6] (A useful trick is to type the first few letters of the directory, e.g. `eo`, and then hit the tab button to "autocomplete" the full directory name.)

```
[student001_17806@eofe4 ~]$ cd eofe-scripts/
[student001_17806@eofe4 eofe-scripts]$
```

We can find out where we are in the directory structure using the `pwd` command:

```
[student001_17806@eofe4 eofe-scripts]$ pwd
/home/student001_17806/eofe-scripts
[student001_17806@eofe4 eofe-scripts]$
```

Note that every time we login to ENGAGING1, our starting directory (a.k.a. our home directory) is always `/home/student001_17806/` (where `student001_17806/` is replaced by our actual username). Also see Footnote 8.

Now, type (`. setup.sh`) to run the setup script (note that there is a period and a space before `setup.sh`).[7]

```
[student001_17806@eofe4 eofe-scripts]$ . setup.sh
```

You will be asked a series of four questions. First-time users should answer yes (`y`) to all four questions. The script does the following:

- *Do you need to update and add modules to .bashrc?* This modifies `.bashrc`, which is a file that is run every time you login to ENGAGING1, so that required modules and programs (such as R) are automatically made available to you. The dot in front of `bashrc` indicates that the file is hidden, and will not be visible when you do `ls`. To see hidden files, use `ls -a`. The `-a` option tells `ls` to show entries starting with a dot.

- *Do you need to download Rmpi?* RMPI is an R package that provides an interface for MPI, which stands for Message Passing Interface. RMPI allows R to use multiple computers in parallel.

- *Do you need to install Rmpi?* Installs RMPI, which is not as trivial as it sounds.

- *Do you need to install misc. HPC R packages?* This does `install.packages` in R for a number of standard packages, such as foreach and snow, that are used for parallel processing.

Once the setup script is done, you are ready to start using R. You can type `R` (capital letter) at the command line to start R. Type `q()` at the R prompt to quit R. We usually **do not** use R interactively in a HPC cluster. Instead, we write an R script that we then send to the cluster scheduler to run. It is good practice to use only computing resources that has been explicitly allocated to us by the scheduler.

3.3 Logging out

To log out of ENGAGING1, type `exit` at the command line, which will return you to your local machine. Note that you can open multiple terminal windows, so that you can work on both your local machine and the remote machine (e.g. ENGAGING1 or ATHENA) in parallel.

[6] To navigate up one level, use `cd ..` (`cd`, space, two dots).

[7] "`. setup.sh`" is equivalent to "`source setup.sh`".

4 Managing files

A common workflow is for us to:

1. Develop our project (i.e. write and test our code) on our local machine on our favorite programs (RStudio, Emacs, etc.).
2. Copy our scripts and data to a remote machine (e.g. ENGAGING1 or ATHENA).
3. Run the scripts on the remote machine.
4. Transfer the output (e.g. a CSV or RData file) back to our local machine for further analysis.

This section describes how to implement steps 2 and 4. Section 5 describes step 1, and Section 6 describes step 3.

We continue with the block bootstrap example from Section 2. The project files are assumed to be saved on our local machine, in the following directory: `/Users/cueball/projects/demo`. There are three files in this project:[8]

```
Cueballs-MacBook-Air:demo cueball$ ls ~/projects/demo
bootstrap_example.r     mydata.csv              run_bootstrap.sh
```

Readers who wish to follow along with this example can download the files from `https://github.com/insongkim/eofe-scripts/tree/master/demo`.

4.1 Creating directories on the remote machine

First, we set up some directories on the remote machine. It is often convenient to have the same directory structure for our project on the remote machine as on our local machine. This way, we can just write `setwd("~/projects/demo/")` in our R code and and have our code work on both local and remote machines.

Login to ENGAGING1 (Section 3.1). Type in the following:

```
[student001_17806@eofe4 ~]$ mkdir projects
[student001_17806@eofe4 ~]$ mkdir projects/demo
[student001_17806@eofe4 ~]$ ls ~/projects/demo
[student001_17806@eofe4 ~]$
```

which creates (`mkdir`) a directory named `projects` on the remote machine, and then creates a directory named `demo` within the `projects` directory. The `ls` command returns nothing, because the `~/projects/demo/` directory on the remote machine is currently empty.

4.2 Transferring files using Rsync

Back on your local machine, type:

```
Cueballs-MacBook-Air:~ cueball$ cd ~/projects
Cueballs-MacBook-Air:projects cueball$ rsync -r ~/projects/demo/ student001_17806@eofe4.mit.edu:/
    home/student001_17806/projects/demo/
```

which navigates to our `projects` directory (if we are not already there), and then runs the rsync program. rsync is a utility that, as its name implies, synchronizes the *destination* directory (in this case, `student001_17806@eofe4.mit.edu:/home/student001_17806/projects/demo/`) with the *source* directory (in this case, `~/projects/demo/`). The source directory is always the first argument, and the destination directory is the second. The `-r` option tells `rsync` to recurse into directories,

[8]On Unix-like operating systems, `~/` represents the user's *home directory*. The home directory could be something like `/Users/cueball` or `/home/cueball`, where `cueball` is replaced by our actual username. `~/projects/demo` thus expands out to `/Users/cueball/projects/demo`. Note that when we type `ls` by itself – without specifying the directory for which we want a listing – it returns the contents of the current directory.

i.e. sync the files in the source directory and all directories within that directory. To find out what other options are available, type `man rsync` at the command line.[9][10]

Now, when we run `ls` on the remote machine,

```
[student001_17806@eofe4 ~]$ ls ~/projects/demo
bootstrap_example.r  mydata.csv  run_bootstrap.sh
```

we can see that the directory on the remote machine has been synchronized with the directory on our local machine. An advantage of rsync is that it compares the source and destination directories, and only transfers from source to destination the files that are different.

4.3 Transferring files using SCP

Suppose we successfully ran our script on ENGAGING1, which produced an RData file named `polr_output.RData`:

```
[student001_17806@eofe4 demo]$ ls
bootstrap_example.r  mydata.csv  polr_output.RData  run_bootstrap.sh
```

We want to copy the output file back to our local machine for further analysis. One way would be to use rsync, setting the source directory as the directory on the remote machine, and the destination directory as the directory on our local machine. Another way would be to use the `scp` command. On our *local* machine, type the following:

```
Cueballs-MacBook-Air:~ cueball$ scp student001_17806@eofe4.mit.edu:/home/student001_17806/
    projects/demo/polr_output.RData ~/projects/demo/
student001_17806@eofe4.mit.edu's password:
polr_output.RData                                                     100%   75KB
    75.2KB/s   00:00
Cueballs-MacBook-Air:~ cueball$ ls -l ~/projects/demo
total 1240
-rw-r--r--  1 cueball  staff    1437 Mar 20 20:21 bootstrap_example.r
-rw-r--r--  1 cueball  staff  548077 Mar 20 08:58 mydata.csv
-rw-r--r--  1 cueball  staff   76972 Mar 21 14:19 polr_output.RData
-rw-r--r--  1 cueball  staff     691 Mar 21 13:50 run_bootstrap.sh
Cueballs-MacBook-Air:~ cueball$
```

We can now load `polr_output.RData` in R and continue our analysis.

4.4 Graphical file management clients

It is also possible to transfer files using file management programs with a graphical interface, such as Fetch (for Mac OS) or SecureFX (for Windows). With these programs, transferring files is as simple as dragging-and-dropping. However, the command line is often more efficient, and the small investment required to learn Rsync and SCP can yield large returns.

5 Writing code for parallelization on a cluster

5.1 R Code using foreach

In Section 2, we ran our block bootstrap with `polr.out <- sapply(1:1000, dobootstrap)`. Running this code in parallel on the cluster is not much more complicated. The code snippet below shows how parallelization on the cluster is done with the `foreach` command, using the same `dobootstrap` function from Section 2:

[9] The `man` command brings up the manual pages for the command that you specify. So, e.g. you can also type `man ls` or even `man man`.

[10] Within the manual pages, type `f` to page-forward, `b` to page-back, and `q` to quit.

```
cl <- makeCluster(np, type="MPI")   # Init MPI backend
registerDoParallel(cl)
polr.out <- foreach(i=1:1000, .combine='cbind', .packages="MASS") %dopar% dobootstrap()
stopCluster(cl)                     # Clean up
```

In this code snippet, we do the following:

- `cl <- makeCluster(np, type="MPI")`: Starts an MPI cluster with `np` number of processors (which we call workers, and the literature sometimes calls cluster nodes or “slaves”). For the choice of `np`, see the discussion of line 17 of our Slurm script in Section 6.1.
- `registerDoParallel(cl)`: Registers the parallel backend with the foreach package.
- `polr.out <- foreach(i=1:1000, .combine='cbind', .packages="MASS") %dopar% dobootstrap()`: Runs `dobootstrap` in parallel 1000 times, loading the MASS package (which we need to run `polr`) in each worker, and combines the output using `cbind`.
- `stopCluster(cl)`: Shuts down cluster before existing R.

In this instance, running our code involved only minor changes to our script, and adding three lines of code to setup and shut down the cluster. As we will see later, the payoff for making these minor changes, in terms of computing time required, is quite large.

Note that each worker starts “clean”, i.e. as if we started a new session of R on the worker. foreach knows which objects are needed to run the code within `%dopar%`, and exports those objects to each worker, but we must remember to specify any packages that our code needs to run via the `.packages` argument (e.g. MASS in the case of `polr`). You can learn more about the foreach package by Googling `R foreach tutorial`.

5.2 R Code using snow

We can use the snow (short for Simple Network of Workstations) package to achieve the same outcome:

```
cl <- makeCluster(np, type="MPI")   # Init MPI backend
clusterEvalQ(cl, require(MASS))     # same as clusterEvalQ(cl, library(MASS))
clusterExport(cl=cl, list=ls())
polr.out <- do.call(cbind, parLapply(cl, 1:1000, dobootstrap))
stopCluster(cl)                     # Clean up
```

The main differences, as compared to foreach, are:

- `clusterEvalQ`, `clusterExport`: `clusterEvalQ` evaluates an expression on each worker. In this case, we run `require(MASS)`, which is equivalent to and can be replaced by `library(MASS)`, to load the MASS package on each worker. `clusterExport` exports the objects in our workspace to each worker.
- `parLapply`: Runs `dobootstrap` in parallel 1000 times and returns a list, which we then combine using `do.call(cbind, ...)`.

5.3 Example R code

The box below presents the R script that we will be running on the cluster. Note the use of `commandArgs` in line 10. This function allows us to feed arguments into our script when we run it through the scheduler (see Section 6.1). For instance, running `Rscript bootstrap_example.r foreach 30` on the command line will pass two arguments to `bootstrap_example.r`, `foreach` and `30`, which we assign to the names `use.method` and `np` respectively (lines 11 and 12). We can then use these two objects in the rest of our code.

bootstrap_example.r

```r
## Example
ptm <- proc.time() # Start-time

## ==========================================================================
## Preamble
setwd("~/projects/demo")

packages <- c("doMPI", "doParallel", "snow", "MASS")
sapply(packages, require, character.only=TRUE)
args <- commandArgs(trailingOnly = TRUE)
use.method <- args[1]
np <- as.numeric(args[2])

## ==========================================================================
## Functions
### Bootstrap
dobootstrap <- function(...) {
    id <- unlist(tapply(d$ID, d$ccode, function(i) sample(i, length(i), replace=TRUE)))
    D <- d[match(id, d$ID),]
    fit <- polr(polr.formula, data=D)
    c(fit$coefficients, fit$zeta)
}

## ==========================================================================
## Analysis
d <- read.csv("mydata.csv")
polr.formula <- factor(redist) ~ finan + incbins + female + age + partyid + educ

if (use.method=="foreach") {
    cl <- makeCluster(np, type="MPI")   # Init MPI backend
    registerDoParallel(cl)
    polr.out <- foreach(i=1:1000, .combine='cbind', .packages="MASS") %dopar% dobootstrap()
    stopCluster(cl)                     # Clean up
}

if (use.method=="snow") {
    cl <- makeCluster(np, type="MPI")   # Init MPI backend
    clusterEvalQ(cl, require(MASS))     # same as clusterEvalQ(cl, library(MASS))
    clusterExport(cl=cl, list=ls())
    polr.out <- do.call(cbind, parLapply(cl, 1:1000, dobootstrap))
    stopCluster(cl)                     # Clean up
}

save(polr.out, file="polr_output.RData")
print(proc.time() - ptm)
mpi.quit()
```

6 Working with the scheduler

We now have to request resources from the scheduler to run our code. When many users are requesting resources, the scheduler puts job requests in a queue. The amount of time a job request spends in the queue before it runs is determined by an algorithm in the backend. What goes into the algorithm is usually quite opaque to the end user, but includes factors like past cluster usage ("fair use"), job size (the amount of time and computing resources requested), queue time, etc.

ENGAGING1 uses Slurm (Simple Linux Utility for Resource Management) to allocate resources. The following sections describe how to write a job request script and submit this job to Slurm.

6.1 Slurm scripts

All jobs are submitted by sending Slurm a batch script that describes the resources that we are requesting, as well as the job that we want to run. A batch script is a simple text file, which we can create and edit in a regular text editor like TextEdit (Mac), Notepad (Windows), or Emacs (more sophisticated and useful). The box below presents the script that we will send to the scheduler for our bootstrap exercise:

run_bootstrap.sh

```
 1  #!/usr/bin/env bash
 2  #SBATCH -p sched_17806           # name of runtime queue
 3  #SBATCH --nodes=2                # number of nodes
 4  #SBATCH --ntasks-per-node=16     # number of cores
 5  #SBATCH -t 10:00                 # wall-time (mm:ss)
 6  #SBATCH -J clustered_bootstrap   # job name
 7  #SBATCH -o log.%j                # output
 8
 9  echo '=========='
10  cd ${SLURM_SUBMIT_DIR}
11  echo ${SLURM_SUBMIT_DIR}
12  echo Running on host $(hostname)
13  echo Time is $(date)
14  echo SLURM_NODES are $(echo ${SLURM_NODELIST})
15  echo '=========='
16
17  mpirun -n 1 Rscript --no-save --no-restore --verbose bootstrap_example.r foreach 30 > bs_output.
        Rout 2>&1
```

The lines prefixed with `SBATCH` (lines 2-7) indicate the resources we are requesting:

- `-p sched_17806`: Tells Slurm to put us into the "queue" named `sched_17806`. Each queue has a specific set of nodes ("machines") set aside for it.
- `--nodes=2`, `--ntasks-per-node=16`: Asks for 2 nodes, and 16 processors per node = 32 processors in total.
- `-t 10:00`: Asks for 10 minutes of computing time. Resources will automatically be withdrawn after this amount of time, and any job that has not completed will be cancelled. To request 1 hour of computing time, use `-t 1:00:00`.
- `-J clustered_bootstrap`: Give the job a name.
- `-o log.%j`: Tells Slurm to send the batch script's standard output to a log file named `log.xxxxxx`, where `xxxxxx` is the job ID.

Lines 9-15 prints out some useful information about the job, but are not necessary for the job submission. We can see this output by using the `more` command to view the log file from the job:

```
[student001_17806@eofe4 demo]$ more log.815349
==========
/home/student001_17806/projects/demo
Running on host node107
Time is Sat Mar 21 16:46:53 EDT 2015
SLURM_NODES are node[107-108]
==========
[student001_17806@eofe4 demo]$
```

Line 17 is the actual command that we want to run. Let's unpack this line:

- `mpirun -n 1 Rscript`: Use `mpirun` to run 1 copy of `Rscript`. If the `-n` option is not specified, `mpirun` runs a copy of the program in every node – which we do not want.
- `--no-save --no-restore --verbose`: Run R (1) without saving output upon exit, (2) without restoring objects from previous sessions, and (3) printing information on progress.
- `bootstrap_example.r`: Filename of the R code that we are sending to `Rscript`. From Section 5.3.
- `foreach 30`: Arguments to be passed to our code (see Section 5.3). **NOTE:** Recall that the second argument is the value of `np` in our code, or the number of processors that we parallelizing across. Recall also that we requested for 2 nodes × 16 processors = 32 processors in total. Why are we telling R to use only 30 processors? Unfortunately, I don't have a good

answer to this question, but (for some reason) the maximum number of processors R can use is capped at number of processors we are assigned (32) minus number of nodes assigned (2) or $32 - 2 = 30$.

- `> bs_output.Rout 2>&1`: Write output data from `R`, as well as any error messages, to a file named `bs_output.Rout`. The output in this file is useful for debugging purposes.

We can view the contents of `bs_output.Rout` using the `more` command:

```
[student001_17806@eofe4 demo]$ more bs_output.Rout
running
  '/cm/shared/engaging/R/3.1.1/lib64/R/bin/R --slave --no-restore --no-save --no-restore --file=
      bootstrap_example.r --
args snow'

Loading required package: doMPI
Loading required package: foreach
Loading required package: iterators
Loading required package: Rmpi
Loading required package: doParallel
Loading required package: parallel
Loading required package: snow
```

⋮

```
        30 slaves are spawned successfully. 0 failed.
[1] 1
   user  system elapsed
 70.915   0.478  72.790
[student001_17806@eofe4 demo]$
```

It took us 73 seconds to run our block bootstrap, or about 1/20-th the time it took originally (Section 2). Why wasn't it 30 times as fast? This is in part because there is some overhead incurred to (1) initialize the parallelization, (2) pass data between the master node and the workers.[11] While optimizing code for parallelization is beyond the scope of this tutorial, it is worth keeping in mind the rule that "communication is much slower than computation."

6.2 Submitting jobs

Once we have written and debugged our code, and our R script, Slurm script, and data have all been uploaded to ENGAGING1, we are ready to run our job, using the following command:

```
[student001_17806@eofe4 demo]$ sbatch run_bootstrap.sh
Submitted batch job 815349
```

6.3 Monitoring jobs

To monitor the progress of our job, use the `squeue` command, giving it the argument `-u username`:

```
[student001_17806@eofe4 demo]$ squeue -u student001_17806
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
            815349 sched_178 clustere student0  R       0:43      2 node[107-108]
```

We can also track our job using the `sacct` command, giving it either a username (e.g. `-u student001_17806`) or a job ID (e.g. `-j 815349`):

```
[student001_17806@eofe4 demo]$ sacct -u student001_17806
       JobID    JobName  Partition    Account  AllocCPUS      State ExitCode
------------ ---------- ---------- ---------- ---------- ---------- --------
815349       clustered+ sched_178+ class_178+         32    RUNNING      0:0
815349.0          orted            class_178+          1    RUNNING      0:0
```

[11]See, for example, the discussion here: `http://stackoverflow.com/questions/14614306/why-is-the-parallel-package-slower-than-just-using-apply`

To check the memory usage of our job, use the `--format` argument. `MaxRSS` and `MaxVMSize` refer to the maximum RAM and virtual memory usage for a job respectively:

```
[student001_17806@eofe4 demo]$ sacct -u student001_17806 --format JobID,NTasks,nodelist,MaxRSS,
     MaxVMSize,AveRSS,AveVMSize
       JobID   NTasks        NodeList     MaxRSS  MaxVMSize     AveRSS  AveVMSize
------------ -------- --------------- ---------- ---------- ---------- ----------
815349                    node[107-108]
815349.batch        1         node107   1573228K    121628K   1573228K    106096K
815349.0            1         node108   1510952K    187676K   1510952K     57220K
```

6.4 Cancelling jobs

To cancel a job, use the `scancel` command, giving it your job ID e.g. `scancel 815349`.

7 Conclusion

In this tutorial, we introduced the ENGAGING1 high performance computing cluster. We covered the following topics:

- Log in, log out, and work with directories
- Synchronize and transfer files between local and remote machines
- Adapt R code for parallelization on a cluster across multiple nodes
- Write Slurm scripts to submit jobs to the scheduler/resource manager
- Submit and monitor job progress on the cluster

Some other resources you may wish to consider:

- To learn more about writing R code for parallelization, Google is almost always the best resource, e.g. "`R parallel computing tutorial`". Adapting your code for parallelization is usually trivially easy if you are able to run your code using `apply`, `lapply`, or a variant.
- To learn more about Slurm commands, try `man <command>` (e.g. `man sbatch`) at the command line after you have logged into ENGAGING1. Also try `http://slurm.schedmd.com/` as well as Google.
- Jonathan Olmsted at Princeton also has a useful tutorial on basic HPC usage, available at `http://q-aps.princeton.edu/files/BasicHPC-Della.pdf`.
- Last but not least, please feel free to get in touch with me (Weihuang) for simple questions, and In Song for more complicated questions!