High Performance Computing: Tools and Applications

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology

Lecture 3

Free Intel compilers and other tools

- Free for students
- Available for Linux, Windows, OS X
- `https://software.intel.com/en-us/articles/free-mkl`

jinx and **deepthought** clusters

- `https://support.cc.gatech.edu/facilities/instructional-labs/jinx-cluster`
- `https://support.cc.gatech.edu/facilities/instructional-labs/deepthought-cluster`

-
  ```
  ssh jdoe1@jinx-login.cc.gatech.edu
  ssh jdoe1@deepthought-login.cc.gatech.edu
  ```
- When you log in, you are on a head node. Do not run jobs on this head node. Instead, submit jobs to the compute nodes or obtain an interactive node.
- Access is possible from outside the GT network.
- The two clusters use a common home directory.
- `git` is available on these clusters.

Job control

```
# show all jobs
qstat -a

# show details about job
qstat -f <jobid>

# list queues and their limits
qstat -q

# check which nodes are down
pbsnodes

# request interactive node
qsub -I -q class -l nodes=1:sixcore -l walltime=30:00
```

Requesting node attributes

```
qstat -l nodes=1

qstat -l nodes=1:sixcore

qstat -l nodes=jinx1
```

Jinx node attributes: - sixcore, fourcore, bigmem, gpu, m2070, m2090

Interactive jobs vs. batch jobs

- Usual practice at supercomputer centers is to submit a batch script
- Interactive jobs are useful for debugging
- Cluster etiquette
  - *Log out of interactive jobs when you are not using them*
  - Use batch jobs if possible

Interactive jobs

- If you are using multithreaded parallelism, you will usually want to request an entire node; otherwise you may be sharing a node with others
- When you are allocated a node, you can also ssh into that node.

Intel Xeon Phi servers: **joker** and **gotham**

- Each server has 8 Intel Xeon Phi (Knights Corner) cards.
- No resource management on these servers. Each card will be assigned to two students, and students coordinate the old-fashioned way.

-
  ```
  ssh jdoe1@joker.cc.gatech.edu
  ssh jdoe1@gotham.cc.gatech.edu
  ```
- Your account name will be your GT login name. For us to create an initial password, provide a throw-away password on the course github repo in the `ex03/accountpass` file.

github

- We will use the Georgia Tech *github* system for distributing course materials and submiting assignments. Go to:

  `https://github.gatech.edu`

  and sign in with your GT credentials. This creates your account if you don't already have one.
- After your account is created, we will give you access to the repository called `chow-courses/cse6230`. You will get an email when you have access to the repository.
- *Fork* this repository. The fork button is in the top-right corner of the screen. This creates a copy of the repository such that the TA and instructor can see your updates to your fork. Note, this fork of the repository stays on the github server.

github

- To make a copy of the fork so that you can make changes to it, you need to clone your fork. To do this, follow the instructions on the web page for your fork, e.g., use:

```
git clone git@github.gatech.edu:jdoe1/cse6230.git
```

- When you submit an assignment, *commit and push* your files in your clone. Commit will update your local copy. Push will update the fork on the server (which the instructor and TA can see).

What is a *fork*?

- A fork is a clone of a repo, but…
- The owner of the original repo can see the files in your fork.

Big picture

```
      Course repo
chow-courses/cse6230
      (upstream)
           |
           | pull updates
           |
           v
    Your local repo
      jinx:cse6230
           |
           | push commits
           |
           v
        Your fork
     jdoe1/cse6230
         (origin)
```

Getting updates into your local repo from the course repo

First set the `upstream` repo. It is only needed once.

```
git remote add upstream \
  git@github.gatech.edu:chow-courses/cse6230.git
```

Then pull from the upstream repo (from the master branch), and merge with your changes (if any):

```
git pull upstream master
```

Finally, push the changes to your fork to also keep it up to date:

```
git push
```

Getting updates into your local repo from your fork

- This is simple:

```
git pull
```

- However, we do not recommend changing files in your fork using github. This just gives you an extra place to make changes which you must keep synchronized.

git – *distributed* version control system

- all repositories are equivalent and contain the entire history (but bare repositories do not have working directories)
- commits are local (do not need to be connected to a central repository)
- but a central repository is still useful. These are implemented as *bare* repositories

Creating a git repository

```
cd project          # change to project directory
git init            # create empty repo
git add filename    # add file to index
git commit          # commit the file to the repo
```

git

Conceptually, there are *four* locations for your files

- working copy
- staging area (a.k.a. index), which is simply a file
- local repository
- remote repository

git cheat sheet (without branches)

```
working copy   <--->   index   <--->   local repo   <--->  remote repo
```

```
git clone                  # clone a git repository
git init                   # initialize git for the current directory
git status                 # status of all files (differences between work
                          # directory, index, and most recent commit)
git log                    # history of commits
git diff                   # show changes between working and local copies
git ls-files               # what files are being managed by git?
                          # (files in index)
git add <file>              # stage a file
git reset HEAD <file>       # unstage a file
git checkout -- <file>      # revert a file to latest version in repo
git checkout <commit> <file> # revert to old version of file
git commit                 # commit files
git fetch                  # fetch from remote repo and update local repo
git merge                  # merge local repo files
git pull                   # fetch and merge changes with your working copy
git push                   # push from local to remote repo
```

`git log` and `git show`

```
# given a file, which commits is it in?
git log <file>

# when was a file added?  (Or just use the above command.)
git log --diff-filter=A -- <file>

# see the changes in a commit
git show <commit>

# list files that changed in a commit
git show --name-only <commit>

# see version of file at a given commit
git show <commit>:<file>    # note colon
```

`git diff`

- `git diff` Show differences between your working directory and the index. Useful when you are editing a file and want to see what changes you have made since the last *add*.
- `git diff -cached` Show differences between the index and the most recent commit. After you do an *add*, you can see what will be committed.
- `git diff HEAD` Show the differences between your working directory and the most recent commit. Usually the most recent commit and what is in the index is the same because you have not yet done an add, so this command may not be so useful (just use git diff by itself). However, consider this: You change a file and do an add. Then you change the file again. Now git diff HEAD will show you the two changes that you have made. This could be useful if you do an add but did not mean to.
- `git diff <commit> <file>` compare workspace file with file in previous commit
- `git diff <commit1> <commit2> <file>` compare file from two different commits

Specifying revisions

- use the hash of the commit
- `HEAD` - most recent commit
- `HEAD~` - parent of most recent commit
- `HEAD~~` or `HEAD~2` - grandparent

There is also `^2`, etc., which can be used to specify which parent of a commit, if there is more than one. If there is only one parent, then `~` and `^` are equivalent.
There is also a `reflog`, a history of where HEAD has been pointing. Reflogs use `@` in its syntax. Since HEAD can be moved without doing commits, this may not necessarily be what you want.

Bare repositories (special property of some repositories)

- repositories that are *shared*
- typical use: this shared repository acts as the *master copy*, so there is typically only this one *centralized* repository
- this shared, centralized repository does not have a working directory, therefore it is *bare*
- the directory names of bare repos end in `.git` by convention

Bare repo and normal repo for a fresh project (single user)

- In the directory where you want the bare repo stored,

```
git init --bare project.git
```

- Then in the directory where you want your normal repo stored,

```
git clone /path_or_url/to/project.git
```

- When you are ready to `push` the first time, use

```
git push origin master
```

  since the bare repo does not yet have any branches on it.
  Subsequently, `git push` is enough.

Bare repo for an existing git project (single user)

- In the directory where you want the bare repo stored,

```
git clone --bare /path_or_url/to/existing_repo
```

- Then in the directory of the existing repo,

```
git remote add origin /path_or_url/to/bare_repo
```

Using git the first time

You may get a message asking you to set some variables for convenience:

```
git config --global user.name "Your Name"
git config --global user.email your@email.com
git config --global core.editor vim
```

Exercise 3

- Due before Wednesday (i.e., do it today): log in to `https://github.gatech.edu`
- Your accounts will be created by Wednesday and you will get an email.
- Due by Friday 11:59 pm: Fork the `chow-courses/cse6230` repository.
  Clone the fork to your `jinx` or `deepthought` account. Edit the file `ex03/accountpass` and provide an initial password for your Intel Xeon Phi account.
  Do whatever is necessary to push your changes to your fork so that the instructor and TA can see it.
- If you have not used `jinx` or `deepthought`, we suggest you also compile and run your Exercise 2 on one of the compute nodes.

Download these slides at

`http://edmondchow.com/hpc/03_github.pdf`

Earlier slides:

`http://edmondchow.com/hpc/intro.pdf`
`http://edmondchow.com/hpc/openmp.pdf`

Future slides will only be on the course github repository in the `lectures` directory.

Special Seminar Today (after class)

- Speaker: Satoshi Matsuoka
- Title: FLOPS to BYTES: Accelerating Beyond Moore's Law
- Location: Klaus 1116 E
- Time: Aug 30, 11 am

The so-called "Moore's Law", by which the performance of the processors will increase exponentially by factor of 4 every 3 years or so, is slated to be ending in 10-15 year timeframe due to the lithography of VLSIs reaching its limits around that time, and combined with other physical factors. This is largely due to the transistor power becoming largely constant, and as a result, means to sustain continuous performance increase must be sought otherwise than increasing the clock rate or the number of floating point units in the chips, i.e., increase in the FLOPS.

The promising new parameter in place of the transistor count is the perceived increase in the capacity and bandwidth of storage, driven by device, architectural, as well as packaging innovations: DRAM-alternative Non-Volatile Memory (NVM) devices, 3-D memory and logic stacking evolving from VIAs to direct silicone stacking, as well as next-generation terabit optics and networks. The overall effect of this is that, the trend to increase the computational intensity as advocated today will no longer result in performance increase, but rather, exploiting the memory and bandwidth capacities will instead be the right methodology. However, such shift in compute-vs-data tradeoffs would not exactly be return to the old vector days, since other physical factors such as latency will not change when spatial communication is involved in X-Y directions. Such conversion of performance metrics from FLOPS to BYTES could lead to disruptive alterations on how the computing system, both hardware and software, would be evolving towards the future.