A Quick-start Guide to Using MTL

How to log in

- Execute the commands:
  `ssh indigo.cs.yorku.ca`
  `ssh -l yufb-s## 192.55.51.81` (provide password when prompted)
  (You must go through indigo, because all other IPs are rejected by MTL)
  - Once you are logged in, you are on the *login* node. You do not run performance testing jobs on this server (it is setup to foil you in any attempt to do this). You submit jobs to a batch server where they are executed, and results are returned back to you on the login node.

How to transfer files

- Advanced: use an SCP client that supports tunneling over SSH, and connect first through indigo.cs.yorku.ca using your CS account, then to MTL using your MTL account, then transfer your files directly from your PC to MTL (and back). On Windows, I highly recommend WinSCP.
- Simple: use the scp command from a Linux box (or an SSH client like Putty) as follows.
  ssh to indigo (you can't access MTL otherwise)
  execute (with the obvious substitutions):
  "`scp path/to/myfile yufb-s##@192.55.51.81:/mtl/path/for/myfile`"
  if you want to copy a whole directory (recursively) to MTL, try:
  "`scp -r path/to/mydir yufb-s##@192.55.51.81:/mtl/path/for/mydir`"
  if you want to copy from MTL to your CS account, just reverse the path arguments:
  "`scp -r yufb-s##@192.55.51.81:/mtl/path/for/mydir path/to/mydir`"

  Example:
  I want to copy file "/home/yufb/yufb-s10/data-32procs.csv" from MTL to
  file "/cs/home/cse63046/mtl/data-32procs.csv" in my CS storage; I execute:
  "`scp yufb-s10@192.55.51.81:/home/yufb/yufb-s10/data-32procs.csv $HOME/mtl/data-32procs.csv`"

How to compile

- The latest java compiler is located at "/opt/java/latest/bin/javac" (on all MTL nodes)
- You compile on the *login* node, in the typical "/opt/java/latest/bin/javac MyClass.java" way.
- When you run your code on the batch node, it will access your files on the login node.
  (technically on network-attached storage)

How to run

- The latest java VM is located at "/opt/java/latest/bin/java" (on all MTL nodes)
- You run a performance test by submitting it as a batch job.
- This consists of writing a *job* file like myjobfile.job, and executing "qsub myjobfile.job" to submit it as a batch job to be executed.
- When you submit it, an ID like "5749.acaad01" will be printed to the terminal. The numeric part of this is the job number (which you can use to kill a bad job, as per below).
- Other people use the system, too, so it may be busy. "`qfree`" will tell you if any nodes are free, and "`qstat`" will show you the queue of jobs waiting to be executed.
- If your job runs wildly out of control, kill it with "`qdel -W force ####`"
  eg. "`qdel -W force 5749`"

- Job files look like this:

```
#!/bin/sh
#PBS -l ncpus=32
#PBS -l mem=12gb
#PBS -l walltime=00:10:00
#PBS -N bbst-trials12
cd /home/yufb/yufb-s10/mtltrials2
./run 12 17 3 data-12procs.csv
```

    1. It is a shell script.
    2. The "PBS tags" names "ncpus" and "mem" are not actually necessary.
    3. The tag "walltime" gives the amount of time (hh:mm:ss) until the job is killed.
    4. The line "#PBS –N bbst-trials12" specifies the name of the job (in output of "qstat")
    5. You have to "cd" to your working directory.

How to get results

- If your program writes to any files, they are written back into your working directory.
  (ie. The same place they would be written if you ran it locally.)
- Any output to standard error or standard output is routed to files "jobname.e####" and "jobname.o####" respectively, where "jobname" is the name specified in the job file and #### is the job number.

Don't crash the server (how to plan so you can recover)

- Any significant number of threads will crash the login node (I crashed it with 64 recently)
- If this happens, you can't login anymore, because the login node cannot spawn a process.
- If you don't want to wait 24 hours, you should "reserve" a process beforehand that you can kill remotely. I use an FTP login. If I happen to crash the login node, I kill my FTP connection, freeing a process. I can then login and use command "`for i in {1..9999}; do kill -9 $i; done`" to kill your processes (you'll see why this is the only way if you find yourself in this situation).

Tips for performance testing (mostly Java)

- Control the cores your application will use with taskset
  eg. "`taskset 1-16 java MyClass`" // MyClass will use only cores 1-16
- Use nanoTime for the most precise timing results

```
long now = System.nanoTime();
/* do stuff */
long elapsedSeconds = (System.nanoTime() - now)/1e9;
```

- If you use a lot of memory, allocate it upfront so that heap resizes don't affect your results
  eg. "`java -Xms 10240m MyClass`"
  or "`java -Xms 10g -Xmx 10g MyClass`"
  (to set a *maximum* if you *really* want to prevent heap resizes)
- Use server mode for more aggressive optimization of code
  eg. "`java -server MyClass`" (you can combine with the above)
- Run garbage collection before each trial by calling System.gc() to restore a "clean slate"
- If you want to conserve memory during thread creation, limit their stack sizes (-Xss 512k)
  eg. "`java –Xss 512k MyClass`" // MyClass' threads will have 512KB stacks
  (You may also have to execute "`ulimit –s 512`" first so that OS hardware threads don't override this.)
- Run many trials and throw out the first few.
  - Java compiles and optimizes on the fly, so your program gets more efficient as time passes. The effect is significant, and most optimizing happens very early. I throw out the first few seconds' data. (For one-second trials, I throw out 3 or 4.)