HPC tutorial

David Chiang

11 Sep 2009

1 Getting started

You access HPC by `ssh` to `hpc-login1.usc.edu` (alias: `hpc.usc.edu`) or `hpc-login2.usc.edu`. These are relatively small nodes. Do not do anything CPU or memory intensive on them. If you do, the Violation Daemon will kill your job and send you a nasty e-mail.

Installed software Some software is installed in `/usr/usc`:

- Matlab: `/usr/usc/matlab/default/`
- Python: `/usr/usc/python/default/`
- GCC: `/usr/usc/gnu/gcc/default/`

You are supposed to run a setup script which will put the executable in your path. For example, in `ksh` or `bash`:

```
$ . /usr/usc/gnu/gcc/default/setup.sh
$ gcc program.c -o program
```

In `csh` or `tcsh`:

```
% source /usr/usc/gnu/gcc/default/setup.csh
% gcc program.c -o program
```

2 Submitting a job

Let's say we have a parser, `parser.py`, which takes two arguments, the input file and the output file. Both have one sentence per line.

Interactive mode The very easiest way to grab a node is:

```
$ qsub -l walltime=1:00:00 -I
```

This requests a single node for interactive use (`-I`) for 1 hour (`-l walltime=1:00:00`). (Don't forget the `-I`: if you do, then `qsub` will wait for you to give it a script on stdin, which is almost never what you want.) When a node becomes available, you will be logged into it, and then you can do

```
$ cd /home/rcf-12/chiangd/hpc-tutorial
$ ./parser.py input output
```

But please don't start an interactive job with a walltime of two weeks and then use the node as your personal computer. It is much greener to use batch mode.

Batch mode Interactive mode is fine for debugging or for various small tasks, but the right way to submit real jobs is batch mode. Write a shell script containing your commands:

Listing 1: run-parser.sh

```
#!/bin/sh

cd /home/rcf-12/chiangd/hpc-tutorial
./parser.py input output
```

and then submit the script:

```
$ qsub -l walltime=1:00:00 run-parser.sh
```

You can also embed `qsub` options in your script:

Listing 2: run-parser.sh

```
#!/bin/sh
#PBS -l walltime=1:00:00

cd /home/rcf-12/chiangd/hpc-tutorial
./parser.py input output
```

Then we don't have to pass the options on the command line:

```
$ qsub run-parser.sh
```

Working directory Note that we had to `cd` into the directory where our files live. Regardless of what directory you do the `qsub` from, the script starts in your home directory. One often wants the script to work in the directory where the `qsub` was issued. That directory is given to you in the `PBS_O_WORKDIR` environment variable. Hence many scripts start off with:

```
cd $PBS_O_WORKDIR
```

Environment variables You can pass additional environment variables to your script using the `-v` option:

```
$ qsub -v FOO=bar,BAR=foo run-parser.sh
```

Arguments You can't use `qsub` to pass any arguments to your script. This is annoying at times, but there is some logic to it: it is better to save complicated command lines in a script instead of typing them in and then forgetting what you did.

Output The stdout and stderr of your job get copied to files called `run-parser.sh.o123456` and `run-parser.sh.e123456`, respectively, in the directory where you did the `qsub`. In order to reduce network traffic, they are first stored *locally* at `/var/spool/torque/spool/123456.hpc.{OU,ER}`, and only after your job finishes will they be copied to their final destination.

- To specify different filenames, use `-o pathname` or `-e pathname`.
- To send stderr to the same place as stdout, use `-j oe`. To do the opposite, use `-j eo`.
- To make the stdout or stderr file appear immediately, use `-k o` or `-k e`, respectively, or `-k oe` for both. But, as if to deter you from committing such a profligate act, the files will appear in your *home* directory and you cannot override the filename with `-o` or `-e`.

An easier way? One might wish for a `qsub` that provided more seamless integration between the command-line on the login node and your job. A script called `qs`, due to Steve DeNeefe and modified by me, takes a command with arguments and runs it in your current working directory:

```
$ qs -l walltime=1:00:00 ./parser.py input output
```

Requesting particular nodes You can submit your job to one of several *queues*:

	maximum...		
queue	nodes/job	walltime	running jobs
main	99	24:00:00	10
quick	4	1:00:00	10
large	256	24:00:00	1
largemem	1	336:00:00	1
isi	∞	336:00:00	∞
hovy	∞	336:00:00	∞

To submit a job to (say) the `isi` queue, use `qsub -q isi`. But you can't submit to main, quick, or large; instead submit your job to default (no `-q`) and it will automatically go to the best queue. In other words: if you use `-l walltime 1:00:00`, your job will probably start sooner.

Here are the specs of the machines in the `largemem`, `isi` and `hovy` queues (the default queue has 1444 machines and is too variegated to list here):

nodes	memory	disk	queue	nodeset
108	8G	60G	isi	V20z
38	16G	60G	isi	pe1950
62	16G	250G	isi	x2200
25	16G	250G	hovy	
5	64G	133G	largemem	

To request nodes. . .

- in (say) the V20z nodeset, use `-l nodes=1:V20z`. (More on the `-l nodes` option below.)
- with at least (say) 10G of memory, use `-l pmem=10g`.

Other important commands Getting information about jobs:

- `qstat -u username` to see the jobs owned by `username`. Our `qstatme` shows you the jobs owned by yourself.
- `qstat -a queue` to see all the jobs on queue `queue`

This won't tell you, however, what kinds of nodes are being used and how many nodes are free. That information comes from another (unwieldy) command, `pbsnodes`. We use a script called `newqf` that fuses the output of the two together:

- `newqf queue` to see the nodes in queue `queue` (default `isi`)
- `newqf -s queue` for the same in summary format

Manipulating individual jobs:

- `qstat -f 123456` to get detailed information about an individual job
- our script `jobsh 123456` to log in to the node the job is running on
- `qdel 123456` to delete a job
- Sometimes a job doesn't die even after you `qdel` it. Doing `qsig -s0 123456` may encourage it to depart from this world.
- `qmove` to move a job to another queue or `qalter` to change a job's submission options (read the man pages). But I find that using `qalter` to decrease a job's walltime is not as effective as deleting and resubmitting it with a lower walltime.

3 Multi-node jobs

Submitting a multi-node job is easy:

```
$ qsub -l nodes=100 run-parser.sh
```

But with our script as-is, one node (the "mother superior") will do all the parsing and the other 99 will be idle. How do we use all these nodes?

- You can get a list of the hostnames of the nodes in your job in the file `$PBS_NODEFILE`, and then use `ssh` to run commands on them.
- This can be kind of tedious. There is a command `pbsdsh` that executes a command on every node which is convenient but isn't very stable.
- The program `giraffe` by Alex Fraser is a better solution.

  Listing 3: run-parser-parallel.sh

  ```
  #!/bin/sh
  #PBS -l nodes=100

  cd /home/rcf-12/chiangd/hpc-tutorial

  PARSER=/home/rcf-12/chiangd/hpc-tutorial/parser.py

  # split input into 1000-line chunks named input.aa, input.ab, etc.
  split input /scratch/input.

  # dynamically generate commands
  for FILE in /scratch/input.*; do
    echo "$PARSER $FILE $FILE.parsed"
  done > $TMPDIR/commands

  # run commands in parallel
  giraffe $TMPDIR/commands

  # reassemble outputs
  cat /scratch/*.parsed > output
  ```

  It reads the commands from `commands` (one per line, no comments), and executes them in parallel, one per node, until they are all done.
- MPI is a library for running a program on multiple intercommunicating nodes.
- Hadoop MapReduce is good for performing lots of independent tasks on large amounts of data.

You can also configure your job so that multiple processes run on a single machine:

```
$ qsub -l nodes=100:ppn=4 run-parser-parallel.sh
```

This will make your job run on 400 virtual nodes, 4 per machine.

- Note that the `-l pmem` option specifies the amount of memory per *virtual* node. So if you do `-l pmem=6g,nodes=100:ppn=4`, it will only run on machines with 24G of memory.
- Don't expect `ppn=4` to give you a fourfold speedup. In general it's not a win to use all the available cores on a machine.

4 Multi-job workflows

You can orchestrate jobs so that one job starts only when another job finishes. This is handy if, for example, one stage of your workflow needs lots of nodes but a later stage only needs one or two.

- `qsub -W depend=afterok:123456` will start the new job only if job 123456 is successful.
- `qsub -W depend=afterany:123456` will start the new job after job 123456 finishes under any circumstances.
- You can specify multiple job ids separated by a colon.

Since the `qsub` command outputs a job id on stdout, the typical usage is to run a meta-script like this:

Listing 4: qsub-parser.sh

```
#!/bin/sh

PARSER=`qsub run-parser.sh`
qsub postprocess.sh -W depend=afterok:$PARSER
```

Then:

```
$ ./qsub-parser.sh
```

Anything more complicated than a few stages, though, and this gets unwieldy. Consider using something more advanced like Condor.

5 Working with files

You have access to several filesystems:

- NFS: shared across nodes, persists across jobs
  - `/home/nlg-01`...`/home/nlg-05`
  - `/home/nlg-tmp`: a little faster, but old files are automatically deleted

    - NFS is very, very slow. You can speed up transfers a little by raising the block size to 1M (e.g., use `dd bs=1M` instead of `cp`).

- PVFS2 in `/scratch`: shared across nodes, not persistent
    - This is a virtual filesystem that is stored across the `/tmp` directories of all the nodes in your job.
    - The capacity can be pretty large (the more nodes, the better)
    - The speed is okay (it operates over a special high-speed network). There are some tricks to speed it up, e.g.:
        - Special-purpose commands like `pvfs2-cp` instead of `cp`
        - Normally a file is distributed uniformly across all nodes. If you have a lot of small files in a directory `/scratch/subdir`, you can `setfattr -n user.pvfs2.num_dfiles -v 1 /scratch/subdir` to make each file live on a single node.[1]
    - In a job with only one node, `/scratch` is not a PVFS, but a symlink to `$TMPDIR`.

- local space in `$TMPDIR`: not shared, not persistent
    - By far the fastest.
    - Do use `$TMPDIR` and not `/tmp`. They go to the same disk, but the former is (theoretically) automatically cleaned up while the latter is not. Please keep the disk clean for the next user.

Always use the fastest storage possible given your storage requirements. For example, if you plan to read a file a lot (= anything other than a single sequential read), copy it from NFS to PVFS (or from NFS to PVFS to local) when your job starts. The reason for this is not only to make your own job finish faster, but to reduce the load on the NFS server for everyone else.

The HPC staff also advise not putting a lot of directories in your `PATH` and not to use an `LD_LIBRARY_PATH` at all. The reason is that every time you run a command, the system must make many small reads all over the NFS filesystems, which apparently increases the load a lot. The same logic applies to `PERL5LIB`, `PYTHONPATH`, `CLASSPATH`, etc.

6 Dealing with problems

Why isn't my job starting?

- Do `checkjob 123456` to get more information. Or `checkjob -v 123456` will tell you node-by-node why your job can't run on that node.

[1]See `http://www.pvfs.org/cvs/pvfs-2-8-branch.build/doc/pvfs2-faq/pvfs2-faq.php` for more information.

- Sometimes it's because you accidentally specified an impossible combination of requirements (e.g., you asked for more memory than any machine has).
- Sometimes a very large or long-running job will have to wait for smaller jobs to go first, even if they were queued later.
- Sometimes there is a future scheduled downtime and your requested walltime would cut into it. So it's important to set your walltime as low as possible (but not too low).
- The `main` queue has a limit of 10 running jobs at a time. Even if plenty of nodes are available, your 11th job will not start.

Why does my job run slow or die?

- You're using NFS too much, or someone else is using NFS too much. Writing is worse than reading; lots of small files is worse than one big file.
- You're using too much memory and the machine is swapping.
- When a job becomes too overloaded, the processes (called MOMs) that monitor your job may lose contact, which causes the job to be killed. In some cases, this is not your fault: if you see the same node giving a MOM-related error message over and over, ask HPC to look at it.

Why don't HPC answer my e-mails? Probably because you aren't providing enough information for them to go on.

- Provide instructions for how to reproduce your problem.
- Provide a specific job id or hostname that is exhibiting the problem. Don't kill the job until they've had a chance to look at it.
- Try to dig as deeply as you can into the underlying cause of the problem.