# HPC

HPC is an acronym for "high-performance computing," and it generally means using a cluster of computers. Our students have access to the Ocelote cluster at the University of Arizona, and most anyone is welcome to use the clusters at TACC. To use a cluster, it's necessary to submit a batch job along with a description of the resources you need (e.g., memory, number of CPUs, number of nodes) to a scheduler that will start your job when the resources become available. We will discuss schedulers "PBS" used at UA and "SLURM" used at TACC.

To interact PBS and SLURM, you must log in to the "head" node(s). Often you will be placed on a random nodes such as "login1." YOU ARE NOT ALLOWED TO DO HEAVY LIFTING ON THE HEAD NODE. For our class, you can write files, interact with the Python RELP, run small scripts, etc., but you should never run BLAST or launch long-running jobs on these machines. They are intended to be used to submit jobs to the queue.

## Handy aliases

To make it easier to go back and forth between PBS and SLURM, I create aliases so that I can execute the same command on both systems:

```
alias qstat="/usr/local/bin/qstat_local"
ME="kyclark"
alias qs="qstat -u $ME"
alias qt="qstat -Jtu $ME"
function qkill() {
  if [[ "${#1}" -eq 0 ]]; then
    echo "Now I crush you!"
    OUT=$(qstat -u $ME | grep $ME | cut -f 1 -d ' ' | sed 's/\[\]\..*/[]/' | xargs qdel)

    if [[ $? -eq 0 ]]; then
        echo "Jobs killed"
    else
        echo -e "\nError submitting job\n$OUT\n"
    fi
  else
    echo Argument = \"$1\"
    echo "This isn't the command you're looking for. I don't take arguments"
  fi
}

function qr() {
  WHO=${1:-$ME}
```

```
  echo "qstat for \"$WHO\""
  OUT=`qstat -Jtu $WHO | tail -n +6 | awk '{print $10}' | sort | uniq -c`
  if [ -n "$OUT" ]; then
      echo "$OUT"
  else
      echo No jobs currently running.
  fi
}
```

## Parallel

One of the easiest ways to boost the performance of a simple bash or Python program is to run many copies of it in parallel. The GNU Parallel program (https://www.gnu.org/software/parallel/) will read all the jobs you wish to run and will distribute them over all the available CPUs, starting new jobs as others finish, and exiting if there is a problem (one of your programs exits with an error code). There is a similar implementation written in the Rust language you may also wish to use (https://github.com/mmstick/parallel).

Here is a bash program to illustrate how to use GNU Parallel. We read a dictionary file and print the first 25 words with their order to demonstrate that the order in which the jobs are executed is not necessarily the way they are written to the jobs file.

```
$ cat -n parallel.sh
     1    #!/usr/bin/env bash
     2
     3    set -u
     4
     5    MAX=25
     6    WORDS=/usr/share/dict/words
     7    CORES=4
     8
     9    if [[ ! -f "$WORDS" ]]; then
    10        echo "WORDS \"$WORDS\" is not a file"
    11        exit 1
    12    fi
    13
    14    TMP=$(mktemp)
    15    i=0
    16    while read -r WORD; do
    17        i=$((i+1))
    18        echo "echo \"$i $WORD\"" >> "$TMP"
    19        if [[ $i -eq $MAX ]]; then
    20            break
```

```
    21        fi
    22    done < "$WORDS"
    23
    24    echo "Starting parallel on $CORES cores"
    25    parallel -j $CORES --halt soon,fail=1 < "$TMP"
    26    echo "Finished parallel"
$ ./parallel.sh
Starting parallel on 4 cores
3 aa
4 aal
5 aalii
6 aam
7 Aani
8 aardvark
9 aardwolf
10 Aaron
11 Aaronic
12 Aaronical
13 Aaronite
14 Aaronitic
15 Aaru
16 Ab
17 aba
18 Ababdeh
19 Ababua
20 abac
21 abaca
22 abacate
23 abacay
24 abacinate
2 a
25 abacination
1 A
Finished parallel
```

And here is a Python implementation of the same idea:

```
$ cat -n parallel.py
     1    #!/usr/bin/env python3
     2    """
     3    Author : kyclark
     4    Date   : 2019-02-25
     5    Purpose: Demonstrate GNU Parallel
     6    """
     7
     8    import argparse
     9    import os
```

```
10    import sys
11    import subprocess
12    import tempfile as tmp
13
14
15    # --------------------------------------------------
16    def get_args():
17        """get command-line arguments"""
18        parser = argparse.ArgumentParser(
19            description='Demonstrate GNU Parallel',
20            formatter_class=argparse.ArgumentDefaultsHelpFormatter)
21
22        parser.add_argument(
23            '-f',
24            '--file',
25            metavar='FILE',
26            help='A positional argument',
27            default='/usr/share/dict/words')
28
29        parser.add_argument(
30            '-c',
31            '--cores',
32            help='Number of cores',
33            metavar='INT',
34            type=int,
35            default=4)
36
37        parser.add_argument(
38            '-m',
39            '--max_lines',
40            help='Maximum number of input lines',
41            metavar='INT',
42            type=int,
43            default=25)
44
45        return parser.parse_args()
46
47
48    # --------------------------------------------------
49    def warn(msg):
50        """Print a message to STDERR"""
51        print(msg, file=sys.stderr)
52
53
54    # --------------------------------------------------
55    def die(msg='Something bad happened'):
```

```
    """warn() and exit with error"""
    warn(msg)
    sys.exit(1)


# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()
    in_file = args.file
    max_lines = args.max_lines
    num_cores = args.cores

    if not os.path.isfile(in_file):
        die('"{}" is not a file'.format(in_file))

    jobfile = tmp.NamedTemporaryFile(delete=False, mode='wt')
    for i, line in enumerate(open(in_file), start=1):
        jobfile.write('echo "{} {}"\n'.format(i, line.rstrip()))
        if i == max_lines: break

    jobfile.close()

    print('Starting parallel on {} cores'.format(num_cores))
    cmd = 'parallel -j {} --halt soon,fail=1 < {}'.format(
        num_cores, jobfile.name)

    try:
        subprocess.run(cmd, shell=True, check=True)
    except subprocess.CalledProcessError as err:
        die('Error:\n{}\n{}\n'.format(err.stderr, err.stdout))
    finally:
        os.remove(jobfile.name)

    print('Finished parallel')


# --------------------------------------------------
if __name__ == '__main__':
    main()
```

```
$ ./parallel.py
Starting parallel on 4 cores
3 aa
4 aal
5 aalii
6 aam
```

```
7 Aani
8 aardvark
9 aardwolf
10 Aaron
11 Aaronic
12 Aaronical
13 Aaronite
14 Aaronitic
15 Aaru
16 Ab
17 aba
18 Ababdeh
19 Ababua
20 abac
21 abaca
22 abacate
23 abacay
24 abacinate
2 a
25 abacination
1 A
Finished parallel
```

The key to being able to use this idea to your advantage is to see when you can split and process your input files without affecting the outcome of your analysis. For instance, if you have 100K sequences you wish to BLAST against a database, perhaps you can split sequences into 10 files of 1K sequences (see "fa_split.py" in the Python Parsing chapter), write out a job file with 10 BLAST jobs, and have `parallel` process them as quickly as possible. Keep in mind, however, that the number of cores may not be your limiting factor. If, for instance, the BLAST database is large and requires a significant amount of your RAM (https://www.youtube.com/watch?v=NdREEcfaihg), then running several jobs in parallel is likely to consume all available memory causing your OS to start swapping to disk ("thrashing") and performance will nosedive. If you have need of multiple, high-memory jobs, then it's necessary to move from parallelizing over all the cores on one machine to parallelizing over many machines (and possibly over all their cores).

## Stampede/SLURM

```
ME="kyclark"
alias qs='squeue -u $ME | column -t'
```

## PBS

The University of Arizona's HPC cluster uses the "PBS Pro" (portable batch system) scheduler.

Here are some important links:

- hpc-consult@list.arizona.edu (email list for help)
- https://confluence.arizona.edu/display/UAHPC/HPC+Documentation
- http://rc.arizona.edu/hpc-htc/high-performance-computing-high-throughput-computing
- http://rc.arizona.edu/hpc-htc/using-systems/pbs-example

## Allocations

Your "allocation" is how much compute time you are allowed on the cluster. Use the command `va` to view your allocation of compute hours, e.g.:

```
$ va
kyclark current allocation (remaining/encumbered/total):
-------------------------------------------
Group               standard                 qualified
bhurwitz            17215:23/00:00/108000:00    99310:56/72:00/100000:00
bh_admin            00:00/00:00/00:00    00:00/00:00/00:00
bh_dev              00:00/00:00/00:00    00:00/00:00/00:00
gwatts              12000:00/00:00/24000:00    00:00/00:00/00:00
mbsulli             228000:00/00:00/228000:00    00:00/00:00/00:00
```

The UA has three queues: high-priority, normal, and windfall. If you exhaust your normal hours in a month, then your jobs must run under "windfall" (catch as catch can) until your hours are replenished.

## Job submission

The PBS command for submitting to the queue is `qsub`. Since this command takes many arguments, I usually write a small script to gather all the arguments and execute the command so it's documented how I ran the job. Most of the time I call this "submit.sh" it basically does `qsub $ARGS run.sh`. To view your queue, use `qstat -u $USER`.

## Hello

Here is a "hello" script:

```
$ cat -n hello.sh
     1     #!/bin/bash
     2
     3     #PBS -W group_list=bhurwitz
     4     #PBS -q standard
     5     #PBS -l jobtype=cluster_only
     6     #PBS -l select=1:ncpus=1:mem=1gb
     7     #PBS -l walltime=01:00:00
     8     #PBS -l cput=01:00:00
     9
    10     echo "Hello from sunny \"$(hostname)\"!"
```

The `#PBS` lines almost look like comments, but they are directives to PBS to describe your job. Lines 6-7 says that we require a very small machine with just one CPU and 1G of memory and that we only want it for 1 hour. The less you request, the more likely you are to get a machine meeting (or exceeding) your needs. On line 10, we are including the `hostname` of the compute node so that we can see that, though we submit the job from a head node (e.g., "login1"), the job is run on a different machine.

Here is a Makefile to submit it:

```
$ cat -n Makefile
     1     submit: clean
     2         qsub hello.sh
     3
     4     clean:
     5         find . -name hello.sh.[eo]* -exec rm {} \;
```

Just typing `make` will run the "clean" command to remove any previous out/error files, and this it will `qsub` our "hello.sh" script:

```
$ make
find . -name hello.sh.[eo]* -exec rm {} \;
qsub hello.sh
818089.service0
$ type qs
qs is aliased to `qstat -u kyclark'
$ qs

service0:
                                                            Req'd  Req'd   Elap
Job ID          Username Queue    Jobname    SessID NDS TSK Memory Time  S Time
--------------- -------- -------- ---------- ------ --- --- ------ ----- - -----
818089.service0 kyclark  clu_stan hello.sh      --    1   1    1gb 01:00 Q   --
```

Until the job is picked up, the "S" (status) column will show "Q" for "queued," then it will change to "R" for "running," "E" for "error," or "X" for "exited." When `qstat` returns nothing, then the job has finished. You should see files like

"hello.sh.o[jobid]" for the output and "hello.sh.e[jobid]" for the errors (which we hope are none):

```
$ ls -lh
total 128K
-rw-rw-r-- 1 kyclark staff     210 Aug 30 10:09 hello.sh
-rw------- 1 kyclark bhurwitz    0 Aug 30 10:19 hello.sh.e818089
-rw------- 1 kyclark bhurwitz  181 Aug 30 10:19 hello.sh.o818089
-rw-rw-r-- 1 kyclark staff      81 Aug 30 10:08 Makefile
-rw-rw-r-- 1 kyclark staff    1.3K Aug 26 08:12 README.md
$ cat hello.sh.o818089
Hello from sunny "r1i3n10"!
Your group bhurwitz has been charged 00:00:01 for 1 cpus.
You previously had 42575:01:07. You now have 42575:01:06 remaining for the queue clu_standar
```

## FTP

While Makefiles can be a great way to document for myself (and others) how I submitted and ran a job, I will often write a "submit.sh" script to check input, decide on resources, etc. Here is a more complicated submission for retrieving data from an FTP server:

```
$ cat -n submit.sh
     1     #!/bin/bash
     2
     3     set -u
     4
     5     export FTP_LIST=get-me
     6     export OUT_DIR=$PWD
     7     export NCFTPGET=/rsgrps/bhurwitz/hurwitzlab/bin/ncftpget
     8     export PBSDIR=pbs
     9
    10     if [[ -d $PBSDIR ]]; then
    11       rm -rf $DIR/*
    12     else
    13       mkdir $DIR
    14     fi
    15
    16     NUM_FILES=$(wc -l $FTP_LIST | cut -d ' ' -f 1)
    17
    18     if [[ $NUM_FILES -gt 0 ]]; then
    19       JOB_ID=$(qsub -N ftp -v OUT_DIR,FTP_LIST,NCFTPGET -j oe -o $PBSDIR ftp-get.sh)
    20       echo "Submitted \"$FILE\" files to job \"$JOB_ID\""
    21     else
    22       echo "Can't find any files in \"$FTP_LIST\""
    23     fi
```

```
$ cat get-me
ftp://ftp.imicrobe.us/projects/33/CAM_PROJ_HumanGut.asm.fa.gz
ftp://ftp.imicrobe.us/projects/121/CAM_P_0001134.csv.gz
ftp://ftp.imicrobe.us/projects/66/CAM_PROJ_TwinStudy.csv.gz
```

Here I have a file "get-me" with a few files on an FTP server that I want to download using the program "ncftpget" (http://ncftp.com/) which is installed in our shared "bin" directory. Since I don't like having the output files from PBS scattered about my working directory, I like to make a place ("pbs") to put them (lines 8-14), and then I include the "-j oe" flag to "join output/error" files together and "-o" to put the output files in $PBSDIR. On line 16, I check that there is legitimate input from the user. Line 19 captures the output from the `qsub` command to report on the submission.

One way to pass arguments to the compute node is by `export`ing variables (lines 5-8) and then using the "-v" option to send those parts of the environment with the job. If you ever get an error on `qsub` that say it can't send the environment, it's because you failed to `export` the variable.

Here is the script that actually downloads the files:

```
$ cat -n ftp-get.sh
     1     #!/bin/bash
     2
     3     #PBS -W group_list=bhurwitz
     4     #PBS -q standard
     5     #PBS -l jobtype=serial
     6     #PBS -l select=1:ncpus=2:mem=4gb
     7     #PBS -l place=pack:shared
     8     #PBS -l walltime=24:00:00
     9     #PBS -l cput=24:00:00
    10
    11     set -u
    12
    13     cd $OUT_DIR
    14
    15     echo "Started $(date)"
    16
    17     i=0
    18     while read FTP; do
    19       let i++
    20       printf "\%3d: \%s\n" \$i \$FTP
    21       $NCFTPGET $FTP
    22     done < $FTP_LIST
    23
    24     echo "Ended $(date)"
```

All of the `#PBS` directives in this script could also have been specified as options

to the `qsub` command in the submit script. Even though I have `set -u` on and have not declared `OUT_DIR`, I can `cd` to it because it was exported from the submit script. When your job is placed on the compute node, it will be placed into your `$HOME` directory, so it's important to have your job place its output files into the correct location. The rest of the script is fairly self-explanatory, reading the `$FTP_LIST` one line at a time, using `ncftpget` to fetch it (`wget` would work just fine, too).

## Job Arrays

Downloading files doesn't usually take a long time, but for our purposes let's pretend each file would take upwards of 10 hours (and we don't know about or have access to GNU Parallel). We are only allowed 24 hours on a compute node, so we think we can fetch at most two files for each job. If we have 200 files, then we need 100 jobs which exceeds the polite and allowed number of jobs we can put into queue at any one time. This is when we would use a job array to submit just one job that will be turned into the required 100 jobs to handle the 200 files:

```
$ cat -n submit.sh
     1     #!/usr/bin/env bash
     2
     3     set -u
     4
     5     export FTP_LIST=get-me
     6     export OUT_DIR=$PWD
     7     export NCFTPGET=/rsgrps/bhurwitz/hurwitzlab/bin/ncftpget
     8     export PBSDIR=pbs
     9     export STEP_SIZE=2
    10
    11     if [[ -d $PBSDIR ]]; then
    12       rm -rf $DIR/*
    13     else
    14       mkdir $DIR
    15     fi
    16
    17     NUM_FILES=$(wc -l $FTP_LIST | cut -d ' ' -f 1)
    18
    19     if [[ $NUM_FILES -lt 1 ]]; then
    20       echo "Can\'t find any files in \"$FTP_LIST\""
    21       exit 1
    22     fi
    23
    24     JOBS=""
    25     if [[ $NUM_FILES -gt 1 ]]; then
    26       JOBS="-J $NUM_FILES"
```

```
27       if [[ $STEP_SIZE -gt 1 ]]; then
28         JOBS="$JOBS:$STEP_SIZE"
29       fi
30     fi
31
32     JOB_ID=$(qsub $JOBS -N ftp -v OUT_DIR,STEP_SIZE,FTP_LIST,NCFTPGET -j oe -o $PBSDI
33
34     echo "Submitted $FILE files to job $JOB_ID"
```

The only difference from the previous version is that we have a new `STEP_SIZE` variable set to "2" meaning we want to handle 2 jobs per node. Lines 24-30 build up the a string to describe the job array which will only be needed if there is more than 1 job.

The FTP downloading now needs to take into account which files to download from the `FTP_LIST`:

```
$ cat -n ftp-get.sh
     1     #!/usr/bin/env bash
     2
     3     #PBS -W group_list=bhurwitz
     4     #PBS -q standard
     5     #PBS -l jobtype=serial
     6     #PBS -l select=1:ncpus=1:mem=1gb
     7     #PBS -l place=pack:shared
     8     #PBS -l walltime=24:00:00
     9     #PBS -l cput=24:00:00
    10
    11     set -u
    12
    13     echo "Started $(date)"
    14
    15     cd $OUT_DIR
    16
    17     TMP_FILES=$(mktemp)
    18     sed -n "${PBS_ARRAY_INDEX:-1},${STEP_SIZE:-1}" $FTP_FILES > $TMP_FILES
    19     NUM_FILES=$(wc -l $TMP_FILES | cut -d ' ' -f 1)
    20
    21     if [[ $NUM_FILES -lt 1 ]]; then
    22       echo "Failed to fetch files"
    23       exit 1
    24     fi
    25
    26     echo "Will fetch $NUM_FILES"
    27
    28     i=0
    29     while read FTP; do
```

```
30        let i++
31        printf "\%3d: \%s\\n" \$i \$FTP
32        $NCFTPGET $FTP
33      done < $TMP_FILES
34
35      rm $TMP_FILES
36
37      echo "Ended $(date)"
```

To extract the files for the given compute node, we use the `PBS_ARRAY_INDEX` variable created by PBS along with the `STEP_SIZE` variable as arguments to a `sed` command, redirecting that output into a temporary file. From there, the script proceeds as before only reading from the `TMP_FILES` and removing it when the job is done.

### Interactive job

You can use `qsub -I` flag to be placed onto a compute node to run your job interactively. This is a good way to debug your script in the actual runtime environment. TACC has a nifty alias called `idev` that will fire up an interactive node for you to play with, so here is a PBS version to do the same. Place this line in your `~/.bashrc` (be sure to `source` the file afterwards):

```
alias idev="qsub -I -N idev -W group_list=bhurwitz -q standard -l walltime=01:00:00 -l selec
```

Then from a login node (here "service2") I can type `idev` to get a compute node. When I'm finished, I can CTRL-D or type `exit` or `logout` to go back to the login node:

```
$ hostname
service2
$ idev
qsub: waiting for job 652560.service2 to start
qsub: job 652560.service2 ready

$ hostname
htc50
$ logout

qsub: job 652560.service2 completed
```

## SLURM

SLURM's command for queue submission is `sbatch` and `showq` will show you your queue. Compute nodes are shared by default. You must request exclusive

access if you need.

- hpc-consult@list.arizona.edu is the help account

## TACC/Stampede

- TACC is part of the XSEDE (xsede.org) project
- TACC does not allow the use of job arrays on their clusters. Instead, they have written their "parametric launcher" (https://www.tacc.utexas.edu/research-development/tacc-software/the-launcher)
- Your three important directories are `$HOME`, `$WORK`, and `$SCRATCH`, and they can be accessed with `cd`, `cdw`, and `cds`, respectively
- Compute nodes are not shared

## SLURM Hello

Here is our "hello" script modified from PBS to SLURM:

```
$ cat -n hello.sh
     1     #!/bin/bash
     2
     3     #SBATCH -A iPlant-Collabs
     4     #SBATCH -p development # or "normal"
     5     #SBATCH -t 01:00:00
     6     #SBATCH -N 1
     7     #SBATCH -n 1
     8     #SBATCH -J hello
     9     #SBATCH --mail-user=kyclark@email.arizona.edu
    10     #SBATCH --mail-type=BEGIN,END,FAIL
    11
    12     echo "Hello from sunny \"$(hostname)\"!"
```

As with the `#PBS` directives, we have `#SBATCH` to describe the job and resources. Most important for TACC is the `-A` allocation argument that decides which account will be charge for the compute time. For the `-p` partition, I can choose either "normal" or "development," the latter of which allows me a maximum of two hours. The idea is that your job gets picked up relatively quickly, which makes it much faster to test new code. The `-J` here is not "job array" (those are not allowed on stampede) but the job name, and I also threw in the options to email me when the job starts and stops.

The Makefile is pretty similar to before. The command `make` will run "clean" and then "sbatch" for us. The "qs" alias shows the job in "R" running state and the "CG" for "completing." The standard error and output go into "slurm-[jobid]" files.

```
$ cat -n Makefile
     1     submit: clean
     2         sbatch hello.sh
     3
     4     clean:
     5         find . -name slurm-\* -exec rm {} \;
$ make
find . -name slurm-\* -exec rm {} \;
sbatch hello.sh
------------------------------------------------------------------
              Welcome to the Stampede Supercomputer
------------------------------------------------------------------

No reservation for this job
--> Verifying valid submit host (login4)...OK
--> Verifying valid jobname...OK
--> Enforcing max jobs per user...OK
--> Verifying availability of your home dir (/home1/03137/kyclark)...OK
--> Verifying availability of your work dir (/work/03137/kyclark)...OK
--> Verifying valid ssh keys...OK
--> Verifying access to desired queue (development)...OK
--> Verifying job request is within current queue limits...OK
--> Checking available allocation (iPlant-Collabs)...OK
Submitted batch job 7560143
$ type qs
qs is aliased to `squeue -u kyclark | column -t'
$ qs
JOBID    PARTITION    NAME   USER     ST  TIME  NODES  NODELIST(REASON)
7560143  development  hello  kyclark  R   0:00  1      c557-904
$ qs
JOBID    PARTITION    NAME   USER     ST  TIME  NODES  NODELIST(REASON)
7560143  development  hello  kyclark  CG  0:08  1      c557-904
$ ls -l
total 16
-rw------- 1 kyclark G-814141  262 Aug 30 12:56 hello.sh
-rw------- 1 kyclark G-814141   77 Aug 30 13:01 Makefile
-rw------- 1 kyclark G-814141 1245 Aug 30 12:52 README.md
-rw------- 1 kyclark G-814141   54 Aug 30 13:09 slurm-7560143.out
[tacc:login4@/work/03137/kyclark/metagenomics-book/hpc/slurm/hello]$ cat slurm-7560143.out
Hello from sunny "c557-904.stampede.tacc.utexas.edu"!
```