BioHPC workshop "Parallelization and load balancing": Exercises Part 1

Exercise 0: Log in to a Linux workstation using an ssh client

If you are on Ithaca campus or have the Ithaca NetID

If you have the Cornell-Ithaca NetID but are currently not on campus, launch the VPN connection on your local machine (laptop) using the CIT-provided **Cisco AnyConnect Secure Mobility Client**. This will make your laptop effectively a part of Ithaca campus network.

If you have a Windows laptop

If not yet done, download the PuTTy ssh client: https://the.earth.li/~sgtatham/putty/latest/w32/putty.exe. Save the exe file anywhere on your laptop (e.g., on the Desktop for access).

Double-click on the PuTTy icon. In the '**Host Name**' field, enter the full name of your assigned machine (e.g., `cbsum1c1b002.biohpc.cornell.edu`). Make sure that '**Port**' is set to '**22**' and '**Connection type**' to '**ssh**'. Click '**Open**'. A terminal window will open with the login prompt. At the prompt, type your BioHPC user ID and hit ENTER. Then enter your BioHPC password and hit ENTER (NOTE: as you type the password - nothing will be happening on the screen - this is on purpose).

Since you will be accessing your assigned machine often during the workshop, it makes sense to create and save a customized profile for it in PuTTy. To do this, open the PuTTy client and enter the full name of the workstation in the '**Host Name**' field and make sure the '**Connection** Type' is '**ssh**' and '**Port**' is '**22**'. Then under 'Saved Session', enter a short nickname for the machine (*e.g.*, the first part of the name, like `cbsum1c1b002`). Expand the '**SSH**' tab in the left panel and click '**X11**' in the left panel, check the box '**Enable X11 forwarding**'. If you prefer the black text on white background, you can change the color settings. Click '**Colours**' in the left panel, set '**Default Foreground**' and '**Default Bold Foreground**' to '**0 0 0**', '**Default Background**' and '**Default Bold Background**' to '**255 255 255**'. Once the customization is complete, click '**Session**' in the left panel, and then click '**Save**'. This will save the machine's profile under a nickname you specified, and it will appear on the list of saved profiles. To connect to a machine with the saved profile, just double-click on the nickname displayed in the '**Saved Sessions**' section.

If you have a Mac (or Linux) laptop

Launch the terminal window. Type (replacing `cbsum1c1b002` with the name of your assigned machine and `your_id` with your own BioHPC user ID)

```
ssh -Y your_id@cbsum1c1b002.biohpc.cornell.edu
```

Enter user your BioHPC password when prompted.

If you are outside off Ithaca campus and do not nave the Cornell-Ithaca NetID

First, you will need to `ssh` to one of our login nodes, and from there - `ssh` further to your assigned machine. To do this, follow the instructions above for your type of laptop, replacing the name of your assigned machine with either of the login nodes: `cbsulogin`, `cbsulogin2`, or `cbsulogin3` (all with the `.biohpc.cornell.edu` suffix). In the terminal which opens on the login node (you will notice the name of that node at the prompt), `ssh` further to you assigned machine, *e.g.*,:

```
ssh cbsum1c1b002
```

Notice that the part `your_id@` and the domain `.biohpc.cornell.edu` have been be omitted from the `ssh` command above. This is possible because your user ID on the login node is the same as on the assigned machine, and all BioHPC machines share the same domain.

It will be convenient to have two or three terminal windows open. If you are familiar with the **screen** program, open a screen session by entering the command `screen`, then create 2 or 3 shell windows (press `Ctrl-a c` a few times). Otherwise use your ssh client to log in two or three times.

Configure passwordless ssh between BioHPC machines

For the sake of this exercise as well as the one next week, it will be convenient to enable passwordless access between BioHPC machines. First, check whether you have such access configured already by attempting to run a simple command (here: `hostname`) on some remote host, for example, the login node:

```
ssh cbsulogin3 uname -a
```

If you are **not** asked for your password, you are already good to go and you may skip this step. Otherwise, run the following sequence of commands:

```
cd
ssh-keygen -t rsa      # press enter a few times to skip over questions
cat .ssh/id_rsa.pub >> .ssh/authorized_keys
echo Host \* >> .ssh/config
echo StrictHostKeyChecking no >> .ssh/config
chmod 700 .ssh
chmod 600 .ssh/authorized_keys .ssh/config
```

Test your configuration by running `uname` on the login node as shown above. Also, you should be able to log in via `ssh` to any of the machines you have reservations on (`e.g`., all workshop machines) from your assigned without having to type the password.

Exercise 1: Process monitoring using `htop`

Since this tool displays a lot of information, it is best viewed when your terminal window is maximized and occupies your whole screen. To start, simply type

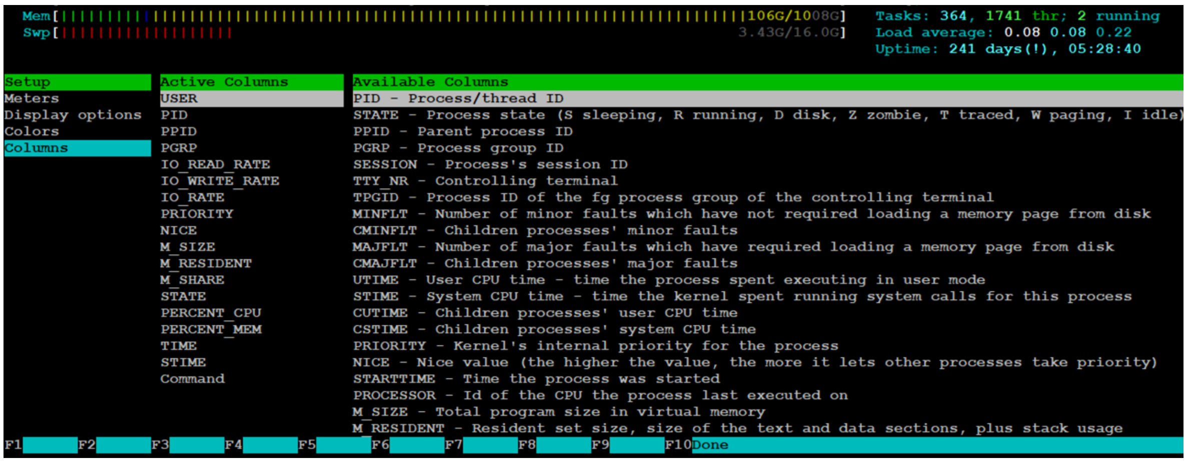
```
htop
```

or

```
htop -u your_id
```

in one of your terminals. The second version (replace `your_id` with your actual BioHPC user ID) will only display information about your own processes. The top display shows CPU cores activity, memory status, and system load information. The bottom part lists all processes (or threads) sorted according to some selectable criteria.

You probably want to customize the display to show certain interesting columns that may not be shown by default. Notice the bar at the bottom - this makes different functions available by pressing the 'F' buttons or other keys on your keyboard. Press '**F1**' or '**h**' to invoke a short help screen (press any key to exit). Press '**F2**' or '**S**' (make sure it's capital) for setup screen:



```
  Mem[|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||106G/1008G]   Tasks: 364, 1741 thr; 2 running
  Swp[||||||||||||||||||                                                  3.43G/16.0G]   Load average: 0.08 0.08 0.22
                                                                                         Uptime: 241 days(!), 05:28:40

Setup            Active Columns      Available Columns
Meters           USER                PID - Process/thread ID
Display options  PID                 STATE - Process state (S sleeping, R running, D disk, Z zombie, T traced, W paging, I idle)
Colors           PPID                PPID - Parent process ID
Columns          PGRP                PGRP - Process group ID
                 IO_READ_RATE        SESSION - Process's session ID
                 IO_WRITE_RATE       TTY_NR - Controlling terminal
                 IO_RATE             TPGID - Process ID of the fg process group of the controlling terminal
                 PRIORITY            MINFLT - Number of minor faults which have not required loading a memory page from disk
                 NICE                CMINFLT - Children processes' minor faults
                 M_SIZE              MAJFLT - Number of major faults which have required loading a memory page from disk
                 M_RESIDENT          CMAJFLT - Children processes' major faults
                 M_SHARE             UTIME - User CPU time - time the process spent executing in user mode
                 STATE               STIME - System CPU time - time the kernel spent running system calls for this process
                 PERCENT_CPU         CUTIME - Children processes' user CPU time
                 PERCENT_MEM         CSTIME - Children processes' system CPU time
                 TIME                PRIORITY - Kernel's internal priority for the process
                 STIME               NICE - Nice value (the higher the value, the more it lets other processes take priority)
                 Command             STARTTIME - Time the process was started
                                     PROCESSOR - Id of the CPU the process last executed on
                                     M_SIZE - Total program size in virtual memory
                                     M_RESIDENT - Resident set size, size of the text and data sections, plus stack usage
F1      F2      F3      F4      F5      F6      F7      F8      F9      F10Done
```

Use **arrow keys** to navigate rows and columns. In the '**Column**' screen, set the '**Active Columns**' as shown, by selectin items from the '**Available Columns**', then use **F7** and **F8** to move the items up or down the selected list. Pres **F10** when done to return to the main screen.

You can dynamically format the process information an many ways. For example, pressing '**H**' will toggle between the process and thread display. '**K**' will show/hide kernel threads (these won't be visible if you started the tool with the `-u` option). Pressing 't' will show processes in a tree view, where it will be easy to figure out the process 'genealogy'. These switches and options may seem not to do much yet, but they will prove more useful once some jobs are started on the machine.

Leave the `htop` running in one of the windows - you will be frequently coming back to look at it.

Exercise 2: Monitoring BLAST search using increasing numbers of threads

If you have not yet done so, prepare your scratch directory on your assigned machine, and create a subfolder blast in it from which this exercise will be run (as always, replace `your_id` with your actual BioHPC user ID):

```
mkdir -p /workdir/your_id/blast
```

Now `cd` to the new directory and copy the query file (20 randomly selected human cDNA sequences) there

```
cd /workdir/your_id/blast
cp /shared_data/Parallel_workshop/cDNA_subset.fa  .
```

Now start the BLAST search with a <u>single thread</u> using the following command (to be written all in one line)

```
/usr/bin/time -v  blastx  -db ../../parworkshop/databases/swissprot  -
num_alignments 1   -num_threads 1 -query cDNA_subset.fa  -out hits.txt.1 >&
run.log.1 &
```

The meaning of the command is as follows: **blastx** search will be performed for each DNA sequence in the query file `cDNA_subset.fa` against **swissprot** amino acid sequence database (stored in files `/workdir/parworkshop/databases/swissprot.*`). The output from the search will be written to the (text) file `hits.txt.1` and will contain 'pictorial' representation of a single (top) hit. The run will use **one** thread, run in the background, and stats from it will be computed by the `/usr/bin/time` tool and reported as STDOUT, redirected to a file `run.log.1`. The elapsed time of this run, when executed using one thread, will be slightly over 10 minutes.

Since you will be running this command a few times with varying number of threads (as given by `-num_threads` option), you may find it useful to create a simple shell script with the following content:

```
#!/bin/bash

NCPU=$1
QFILE=cDNA_subset.fa
OFILE=hits.txt.$NCPU

/usr/bin/time -v \
blastx \
-db ../../parworkshop/databases/swissprot \
-num_alignments 1 \
-num_threads $NCPU \
-query  $QFILE \
-out $OFILE \
&> run.log.$NCPU
```

Once the script is saved (for example, in a file called `run_blast.sh`) and made executable

```
chmod u+x run_blast.sh
```

it can be run simply as follows

```
./run_blast.sh 1 &
```

which is equivalent to the lengthy `blastx` command presented above. Note that the number of threads to run on is passed on as an argument `$1` and represented by a variable `NCPU`. The names of the query file and output file are also assigned variables and are therefore easy to change. The long command has been broken into several separate lines to improve script readability (make sure that the backslash `\` character is the very last one in each of the broken lines, `i.e.`, it is not followed by any empty spaces). Running a search using number of threads different than 1 is simple, `e.g.`:

```
./run_blast.sh 8 &
```

will start the run using 8 threads.

Run the search using varying number of threads, for example, 1, 2, 4, 8. During each run, observe the output from `htop` (which you should have running in a separate window or terminal). Press '**H**' and '**t**' to see the processes and threads in a tree view. Are all threads running at full capacity (100% CPU)? How much real memory (**RES**) does each process take (note that all threads from a given process share the same memory)? Is the number of threads shown equal to the number requested by `-num_threads` option? How many of these threads are 'running' (are is '**R**' state) and how many are sleeping (are is '**S**' state)? Are all `blastx` threads running continuously or do they disappear and are then 're-born'? What does it suggest about the parallelization scheme inside `blastx`?

The statistics produced by the `time` command are recorded in files `run.log.*`. For each run, collect information about the elapsed time of the run and peak memory usage. Does peak memory grow with the number of threads used? Given the speedup achieved, is it worth using all 8 cores to run a single `blastx` search on this machine?

Exercise 3: BWA alignment using varying numbers of threads

This exercise is similar to Exercise 2, except this time the task will be to align a bunch (1,683,041) paired-end reads from *Drosophila melanogaster* to a reference genome using `bwa mem` aligner whose output (in SAM format) is piped into `samtools` for 'on the fly' conversion to BAM format. Here again it will be convenient to use a shell script which will accept the number of threads as an argument. Create a scratch folder for the exercise and make it your current directory:

```
mkdir -p /workdir/your_id/bwa
cd /workdir/your_id/bwa
```

Copy the example input files `example_1.fastq.gz` ad `example_2.fastq.gz` with the reads to align:

```
cp /shared_data/Parallel_workshop/example_?.fastq.gz  .
```

Then use a text editor of your choice to create the following script (call it `bwa_aln.sh`):

```
#!/bin/bash

# Reference genome we will be aligninh to
REFFASTA=../../parworkshop/genome/genome.fa

# intercept and save the requested number of thread in variable NCPU
NCPU=$1

echo Alignment started
date

# ... and this is the proper command

/usr/bin/time -v \
bwa mem -M -t $NCPU \
$REFFASTA \
example_1.fastq.gz  example_2.fastq.gz \
| /usr/bin/time -v \
samtools view -Sb - -o example.bam
```

```
echo Alignment finished
date
```

Note that `bwa mem` (the aligner) is run on the number of threads given by `-t` option, and its output is piped into another program, `samtools`, which converts its format from SAM (text and big) to BAM (binary and small). Both the `bwa` and `samtools` executables are run through the tool `/usr/bin/time` in verbose mode to collect various run-time counters and statistics. This has to be done this way, since this tool only works with one executable at a time. Note also, that while the `bwa` aligner is parallelized, `samtools` works as a relatively faster but single-threaded process.

After saving the script file, make it executable

```
chmod u+x bwa_aln.sh
```

and run it a few times, varying the number of threads. For example, you would start the run on 4 threads with the command

```
./bwa_aln.sh 4 >& run.log.4 &
```

which will save any screen output (including the output from the time commands) in the log file `run.log.4.` Run only one instance of such script at a time. Between the runs, you may want to delete the result of the alignment, file `example.bam`, which will be re-created each time.

During each run, monitor the processes, threads, and resources they take using `htop`, which you should have open in a separate window. Observe the behavior of processes and threads and try to answer questions asked in Exercise 2. What can you say about relative 'business' of `bwa` and `samtools` threads as the number of the latter increases?

Exercise 3a (optional)

As shown (hopefully) in Exercise 3 above, the sequential `samtools` step of SAM to BAM conversion may be hindering parallel performance of the whole alignment pipeline. One can imagine a different scheme, where `bwa mem` is run on its own, saving an intermediate (big) SAM file to disk, and then `samtools` is used, in a separate step, to convert that file to BAM format. Thus, instead of the pipe construct involving "`|`", the following separate commands would be used:

```
/usr/bin/time -v \
bwa mem -M -t $NCPU \
$REFFASTA \
example_1.fastq.gz  example_2.fastq.gz > example.sam
```

and then

```
/usr/bin/time -v \
samtools view -Sb example.sam -o example.bam
rm example.sam
```

The downside of this strategy is the creation of a large intermediate file `example.sam`, which not only takes space on disk, but potentially generates a lot of I/O traffic. The hope is, however, that thanks to the decoupling of the pipe, the speedup of the `bwa` aligner will scale better with the number of threads, so that the total execution time (the sum of `bwa mem` and the `samtools` times) may be reduced compared to the 'piped' version. Test this hypothesis.

Exercise 4: using GNU `parallel`

GNU `parallel` is a standard tool for easy generation of commands representing multiple independent tasks with and running these commands concurrently with some load balancing mechanisms built in.

Before using parallel for the first time it is good to silence the 'citation request', which would otherwise appear every time you run the tool. To get rid of the request, run the following:

```
parallel --citation
```

then type `will cite` and hit ENTER.

Create a separate scratch directory for this exercise and `cd` to it:

```
mkdir -p /workdir/your_id/parallel
cd /workdir/your_id/parallel
```

Test basic functionality of the tool as shown in workshop slides 44-52. As command referred to in the slides as `someprog`, you can use the shell command `echo`, whose sole purpose is to echo the arguments to the screen, e.g.,

`echo a3`

will simply print `a3` in your terminal. Try running `echo` through `parallel` with arguments supplied both as a list after `:::` and as a list in a file.

To see the usefulness of parallel in a more realistic situation, you can try `gzip`'ping a few files in parallel. Copy three example files `BBB_1`, `BBB_2`, and `BBB_3`

```
cp /shared_data/Parallel_workshop/BBB_?  .
```

and compress them using parallel in several ways, as suggested in the workshop slides.

Option 1: Compress the three files, running no more than 2 processes at a time:

```
parallel -j 2 gzip ::: BBB_1 BBB_2 BBB_3 &
```

The result, which should appear after a few moments, will be three compressed files `BBB_1.gz`, `BBB_2.gz`, and `BBB_3.gz`. Before you try other tricks with parallel, restore these files to their original, un-compressed form. This, in fact, can also be done using parallel, for example,

```
parallel gunzip ::: BBB_1.gz BBB_2.gz BBB_3.gz &
```

This time, since there is no `-j` option, all three `gunzip` processes will run concurrently.

A different but equivalent equivalent forms of the two commands above would use the reference to the argument, `{}`

```
parallel -j 2 gzip BBB_{} ::: 1 2 3 &
```

```
parallel gunzip BBB_{}.gz ::: 1 2 3 &
```

While running the example above, observe the output from the `htop` command running in the other window. You should see no more than 2 of your `gzip` processes and 3 `gunzip` processes running simultaneously.

Option 2: Prepare a file, call it `FileFile` , containing the list of files to process:

```
BBB_1
BBB_2
BBB_3
```

Then use one of the three equivalent commands to compress the files (with no more than 2 concurrent processes)

```
parallel -j 2 -a FileFile gzip &
```

```
parallel -j 2 gzip :::: FileFile &
```

```
cat FileFile | parallel -j 2 gzip &
```

Again, un-compress all files before trying the next Option. You can use the same list of files:

```
parallel gunzip {}.gz :::: FileFile
```

Option 3: Prepare a file, call it `TaskFile` , containing the list of commands to process:

```
gzip BBB_1
gzip BBB_2
gzip BBB_3
```

The run these commands through parallel making sure that no more than 2 processes are running at any given time:

```
parallel -j 2 < TaskFile  &
```

Equivalent form of this command would be to pipe the content of the `TaskFile` into parallel

```
cat TaskFile | parallel -j 2
```

or use the `::::` separator to provide input from a file:

```
parallel -j 2 :::: TaskFile
```

Exercise 5: Using `parallel` with BLAST

In this exercise we will revisit the **blastx** search from Exercise 2, but this time, we will split the query file `cDNA_subset.fa` into 4 smaller files, containing 8, 8, 7, and 7 sequences, respectively. Each of these small files will be fed into a separate **blastx** search run on 2 threads, and all these 4 searches will run simultaneously using `parallel`, *i.e.*, using the total of 8 threads. The objective is to compare the timing of this run to that of a single run on 8 threads from Exercise 2.

Copy the four sub-parts of the query , `cDNA_subset_1.fa` through `cDNA_subset_4.fa` from the shared workshop directory to your blast scratch folder:

```
cd /workdir/your_id/blast
cp /shared_data/Parallel_workshop/cDNA_subset_?.fa  .
```

Modify the script `run_blast.sh` to parametrize it with the part of the query (passed to the script as argument $1) rather than number of threads:

```
#!/bin/bash

NCPU=2
QFILE=cDNA_subset_${1}.fa
OFILE=hits.txt.part$1

/usr/bin/time -v \
blastx \
-db ../../parworkshop/databases/swissprot \
-num_alignments 1 \
-num_threads $NCPU \
-query  $QFILE \
-out $OFILE \
&> run.log.part$1
```

Then execute 4 such runs concurrently, each on a different part of the query, using `parallel`:

```
parallel --joblog LOG ./run_blast.sh cDNA_subset_{}.fa ::: 1 2 3 4 &>
overall.log &
```

Option `--joblog` will create a timing summary of all the runs, saved in file `LOG`. In addition, detailed output from the time tool will be saved for all runs in files `run.log.part1` - `run.log.part4`.

Compare this timing information to results of the 8-thread run on the full query, as done in Exercise 2. Which is faster?