Efficient use of CPUs/cores on BioHPC Lab machines

All BioHPC machines have multiple CPUs (cores) available. Modern processors are faster than the past ones, but the biggest difference between old and new servers is in the amount of parallel processing power available – multiple CPUs and cores. Without using this power, i.e. running programs in parallel, there will be little speedup and lots of resources will be wasted. From practical point of view the parallel processing power of a server is measured in ***cores***, which are processing units capable of executing code independently. ***CPU*s** are separate microprocessors on separate chips, and each of them usually host multiple cores.

There 4 general ways to run programs in parallel: (a) using a given program's built-in parallelization, (b) executing a bunch of programs in the background in parallel, (c) using a driver program to execute multiple programs, or (d) using a job scheduler (SLURM). Each of these methods will be discussed below.

In order to illustrate how important it is to run programs in parallel, the table below shows two real world examples.

(a) Using BLAST to search Swissprot database for matches of 10,000 randomly chosen human cDNA sequences. Swissprot is a good example of a small memory footprint.

Machine	CPU available	cores available	cores used	time (hrs)	speedup (in machine)
cbsulm10	4	64	64	0.931	27.506
cbsulm10	4	64	16	1.962	13.056
cbsulm10	4	64	1	25.619	1.000
cbsumm15	2	24	24	2.058	12.117
cbsumm15	2	24	12	2.593	9.616
cbsumm15	2	24	1	24.930	1.000
cbsum1c2b008	2	8	8	4.193	6.717
cbsum1c2b008	2	8	1	28.161	1.000

(b) Using BLAST to search nr database for matches of 2,000 randomly chosen human cDNA sequences. Nr is a good example of a large memory footprint.

Machine	CPU available	cores available	cores used	time (hrs)	speedup (in machine)
cbsulm10	4	64	64	10.97	2.222
cbsulm10	4	64	16	24.37	1.000
cbsumm15	2	24	24	26.10	2.140
cbsumm15	2	24	12	55.85	1.000

The above examples highlight several important points of parallel execution.

- ✓ First – it is VERY important to use multiple cores. BLAST on 64 cores takes only 0.931 hours (2K cDNA vs swissprot), the same run on a single core takes over 25 hours!
- ✓ Speedup is not directly proportional to the number of cores. Sometimes it is linear, but most often it is not, usually it is somewhat less than expected, but still sufficiently large to justify the effort. 64 cores compared to 1 core in example (a) have 27.5 speedup rate, much less than 64 expected from linear trend, but still large!
- ✓ Speedup depends not only on the machine (hardware), but also program (algorithm) and parameters (nr vs swissport). When using nr database (example b) on cbsumm15 the speedup between 12 and 24 cores is 2.14, for swissprot in the same situation (example a) it is 12.117/9.616=1.26. It is often a good idea to run a short example first (if possible) on a subset of data to figure out the optimal number of cores.

(a)Using a given program's built-in parallelization

Many programs have built-in parallelization options. You will need to read the appropriate documentation to find out what is the name of this option; usually it is described as "number of threads" or "number of processes". Typical examples are blast (option is '-a') and blast+ ('-num_threads').

```
blastall -a 8 [other options]
```

```
blast+ -num_threads 8 [other options]
```

Other examples are tophat ('-p'), cuffdiff ('-p'), bwa ('-t') and bowtie ('-p').

```
tophat -p 8 [other options]
```

```
cuffdiff -p 8 [other options]
```

```
bwa -t 8 [other options]
```

```
bowtie -p 8 [other options]
```

Most of the common CPU-intensive programs do have multithreading options, if their algorithms permit it. It is usually important to research how many cores are reasonable; sometimes using more cores will not give any measurable speedup (diminishing returns). It may not be a big problem, since you can always run multiple programs in the background, each on multiple cores.

(b) Executing a bunch of programs in the background in parallel

If the number of programs to run is less or equal to the number of cores available you can run them all in the background in parallel. To do so, you will need to prepare a file containing all the commands necessary, with each program output redirected to a different file. Here is a real world example of running several tophat commands, each using multiple cores on a 64 core machine. The file prepared contains 9 commands since I had 9 tophat jobs to run for this particular RNA-seq pipeline. To use all cores on a 64 core machine I used 7 cores per tophat command.

```
tophat -p 7 -o B_L1-1 --transcriptome-index genome/transcriptome/ZmB73_5a_WGS \
   --no-novel-juncs genome/maize \
   fastq/2284_6063_7073_C3AR7ACXX_B_L1-1_ATCACG_R1.fastq.gz \
   fastq/2284_6063_7073_C3AR7ACXX_B_L1-1_ATCACG_R2.fastq.gz >& B_L1-1.log &
tophat -p 7 -o B_L1-2 --transcriptome-index genome/transcriptome/ZmB73_5a_WGS \
   --no-novel-juncs genome/maize \
   fastq/2284_6063_7076_C3AR7ACXX_B_L1-2_TGACCA_R1.fastq.gz  \
   fastq/2284_6063_7076_C3AR7ACXX_B_L1-2_TGACCA_R2.fastq.gz >& B_L1-2.log &
tophat -p 7 -o B_L1-3 --transcriptome-index genome/transcriptome/ZmB73_5a_WGS \
   --no-novel-juncs genome/maize \
   fastq/2284_6063_7079_C3AR7ACXX_B_L1-3_CAGATC_R1.fastq.gz  \
   fastq/2284_6063_7079_C3AR7ACXX_B_L1-3_CAGATC_R2.fastq.gz >& B_L1-3.log &
tophat -p 7 -o L_L1-1 --transcriptome-index genome/transcriptome/ZmB73_5a_WGS \
   --no-novel-juncs genome/maize \
   fastq/2284_6063_7074_C3AR7ACXX_L_L1-1_CGATGT_R1.fastq.gz \
   fastq/2284_6063_7074_C3AR7ACXX_L_L1-1_CGATGT_R2.fastq.gz >& L_L1-1.log &
tophat -p 7 -o L_L1-2 --transcriptome-index genome/transcriptome/ZmB73_5a_WGS \
   --no-novel-juncs genome/maize \
   fastq/2284_6063_7077_C3AR7ACXX_L_L1-2_ACAGTG_R1.fastq.gz  \
   fastq/2284_6063_7077_C3AR7ACXX_L_L1-2_ACAGTG_R2.fastq.gz >& L_L1-2.log &
tophat -p 7 -o L_L1-3 --transcriptome-index genome/transcriptome/ZmB73_5a_WGS \
   --no-novel-juncs genome/maize \
   fastq/2284_6063_7080_C3AR7ACXX_L_L1-3_ACTTGA_R1.fastq.gz \
   fastq/2284_6063_7080_C3AR7ACXX_L_L1-3_ACTTGA_R2.fastq.gz >& L_L1-3.log &
tophat -p 7 -o S_L1-1 --transcriptome-index genome/transcriptome/ZmB73_5a_WGS \
   --no-novel-juncs genome/maize \
   fastq/2284_6063_7075_C3AR7ACXX_S_L1-1_TTAGGC_R1.fastq.gz \
   fastq/2284_6063_7075_C3AR7ACXX_S_L1-1_TTAGGC_R2.fastq.gz >& S_L1-1.log &
tophat -p 7 -o S_L1-2 --transcriptome-index genome/transcriptome/ZmB73_5a_WGS \
   --no-novel-juncs genome/maize \
   fastq/2284_6063_7078_C3AR7ACXX_S_L1-2_GCCAAT_R1.fastq.gz \
   fastq/2284_6063_7078_C3AR7ACXX_S_L1-2_GCCAAT_R2.fastq.gz >& S_L1-2.log &
tophat -p 7 -o S_L1-3 --transcriptome-index genome/transcriptome/ZmB73_5a_WGS \
   --no-novel-juncs genome/maize \
   fastq/2284_6063_7081_C3AR7ACXX_S_L1-3_GATCAG_R1.fastq.gz \
   fastq/2284_6063_7081_C3AR7ACXX_S_L1-3_GATCAG_R2.fastq.gz >& S_L1-3.log &
```

In principle, the file should look like the template below

```
command1 [parameters] >& log1 &
command2 [parameters] >& log2 &
command3 [parameters] >& log3 &
command4 [parameters] >& log4 &
…
commandN [parameters] >& logN &
```

Such a script can be executed by typing

```
bash script_name
```

in the directory where the script is. You will need to examine log files in order to find out if the programs finished. Another way of monitoring them is to execute 'ps' command and filter out the name of your program. In my case, it was tophat, and the command was

```
ps -ef | grep tophat
```

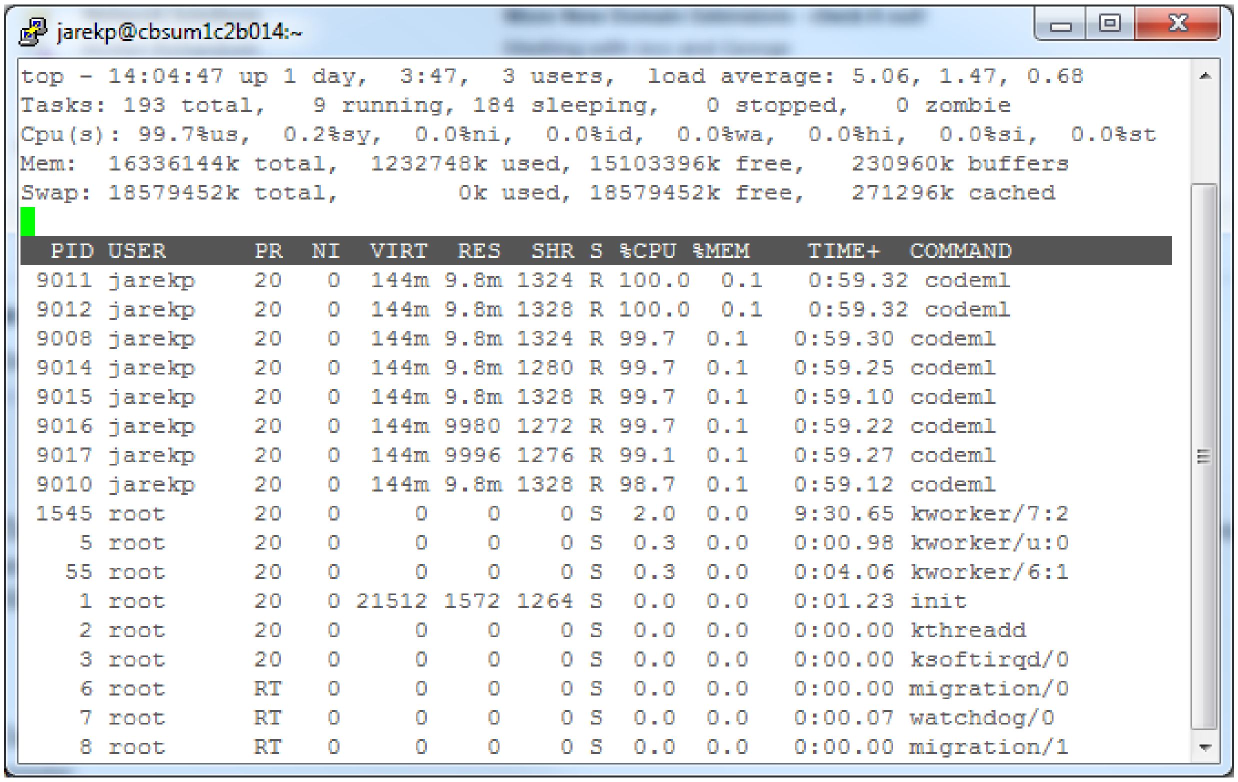
For more information about running programs, dealing with files and the Linux computing environment in general please refer to our "Linux for Biologists" workshop slides available online

http://cbsu.tc.cornell.edu/lab/doc/Linux_workshop_Part1.pdf

http://cbsu.tc.cornell.edu/lab/doc/Linux_workshop_Part2.pdf

When running multiple programs in parallel it is important to check the memory requirements – how much memory (RAM) does a single program need? The sum of these memory requirements cannot exceed the total memory available on the server. For example, if a single program needs 3GB of RAM, then you can run easily 24 of them on medium memory machine (total 72GB needed and 128GB available), but only 5 on general machine (16GB available). The easiest way to check the memory requirements (besides reading program's manual) is to run one task and check its memory usage with 'top' command. It displays % of memory used for each process, knowing the total RAM on the machine it is easy to compute programs memory footprint.

The output of the top command is showed below. Memory usage is reported in "%MEM" column, in this case I am running 8 codeml programs in parallel, each using 0.1% of memory (0.001*16GB is 0.016GB, which in turn is 16MB).



```
top - 14:04:47 up 1 day,  3:47,  3 users,  load average: 5.06, 1.47, 0.68
Tasks: 193 total,   9 running, 184 sleeping,   0 stopped,   0 zombie
Cpu(s): 99.7%us,  0.2%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  16336144k total,  1232748k used, 15103396k free,   230960k buffers
Swap: 18579452k total,        0k used, 18579452k free,   271296k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 9011 jarekp    20   0  144m 9.8m 1324 R 100.0  0.1   0:59.32 codeml
 9012 jarekp    20   0  144m 9.8m 1328 R 100.0  0.1   0:59.32 codeml
 9008 jarekp    20   0  144m 9.8m 1324 R 99.7  0.1   0:59.30 codeml
 9014 jarekp    20   0  144m 9.8m 1280 R 99.7  0.1   0:59.25 codeml
 9015 jarekp    20   0  144m 9.8m 1328 R 99.7  0.1   0:59.10 codeml
 9016 jarekp    20   0  144m 9980 1272 R 99.7  0.1   0:59.22 codeml
 9017 jarekp    20   0  144m 9996 1276 R 99.1  0.1   0:59.27 codeml
 9010 jarekp    20   0  144m 9.8m 1328 R 98.7  0.1   0:59.12 codeml
 1545 root      20   0     0    0    0 S  2.0  0.0   9:30.65 kworker/7:2
    5 root      20   0     0    0    0 S  0.3  0.0   0:00.98 kworker/u:0
   55 root      20   0     0    0    0 S  0.3  0.0   0:04.06 kworker/6:1
    1 root      20   0 21512 1572 1264 S  0.0  0.0   0:01.23 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/0
    6 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    7 root      RT   0     0    0    0 S  0.0  0.0   0:00.07 watchdog/0
    8 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/1
```

(c) Using a parallel driver program to execute multiple programs

If the number of programs to run is greater than the number of cores available you need another program to control proper utilization of the cores (load balance). The goal is to have all cores used, but the number of cores requested at a given time should not be larger than the number of cores available. We have written a Perl program (/programs/bin/perlscripts/perl_fork_univ.pl) to be used in this case.

This Perl program takes 2 arguments:

```
/programs/bin/perlscripts/perl_fork_univ.pl JobListFile ProcessNumber
```

JobListFile is a file containing all the commands to execute – one per line. *ProcessNumber* is the number of processes to execute in parallel, which is equal to the number of cores to be used.

Typical examples of parallel Perl driver use are cases when the number of tasks exceeds the number of cores. For example, when the number of libraries in RAN-seq project is large (say 50), you can prepare a file with all tophat tasks needed (50 lines, no '&' at the end of lines!, each of them on 7 cores) and then run 9 of them at a time on 64 core machine (using total 63 cores – 9 instances at a time using 7 cores each). Another example is when you need to compress large number of files, then you can prepare a list of 'gzip filename' commands in a file and then run 10 of them a time (*ProcessNumber*=10).

Another example is PAML simulation on 110 genes. The example input data is in /programs/paml.example.tar. If you would like to try it you need to unpack data into your /workdir subdirectory (substitute you own lab id for 'jarekp')

```
cd /workdir
mkdir jarekp
cd jarekp
tar -xf /programs/paml.example.tar
```

Task list is stored in file 'tasklist', which begins as follows (total 110 lines). Note no '&' at line ends, which is different from point (b).

```
cd results/984 ; /workdir/jarekp/paml4.7/bin/codeml my.control >& log
cd results/916 ; /workdir/jarekp/paml4.7/bin/codeml my.control >& log
cd results/935 ; /workdir/jarekp/paml4.7/bin/codeml my.control >& log
cd results/937 ; /workdir/jarekp/paml4.7/bin/codeml my.control >& log
cd results/922 ; /workdir/jarekp/paml4.7/bin/codeml my.control >& log
cd results/978 ; /workdir/jarekp/paml4.7/bin/codeml my.control >& log
```

In order to run PAML simulations in parallel you need to execute Perl parallel driver

```
/programs/bin/perlscripts/perl_fork_univ.pl tasklist 8
```

I used 8 cores (tasks) since the example was run on one of general machines.

Same as in point (b) you need to make sure total amount of memory needed by concurrently run programs does not exceed total amount of RAM available.

Similarly, it is possible to run programs in parallel on multiple workstations using a modified version of this Perl program: /programs/bin/perlscripts/perl_fork_univ_mn.pl . More details are available in

https://cbsu.tc.cornell.edu/lab/doc/using_perl_fork_univ_mn.pdf

(d) Using a job scheduler (SLURM) to execute multiple programs

SLURM is a workload manager (SLURM = Simple Linux Utility for Resource Management). It has much more flexibility than the perl_fork_univ.pl function. Some benefits include: being able to run multiple jobs in parallel with different processor/memory requirements; adding more jobs to the queue at any time; sharing cores between users; distributing jobs across several machines.

Some machines at BioHPC are already running SLURM; to see clusters that are available to you, use the command:

```
manage_slurm list
```

If there are no results on the machine where you want to run jobs, you can create a cluster with the command “manage_slurm machine1,machine2,…”, where the argument is a comma-delimited list of machine(s) where you want to create a single SLURM cluster (to distribute jobs across all these machines). For example:

```
manage_slurm new cbsum1c1b002,cbsum1c1b003
```

will create a new SLURM cluster across these two machines; the first machine is the “master node” of the cluster, and the cluster is named after it. In order for this command to succeed, you need to have an active reservation on all machines. Any user who has access to the full set of machines can submit jobs to this cluster, but the user who created the cluster becomes the “head” user and is the only one who can manage it (see “manage_slurm --help” for options). The cluster will automatically be killed when head user’s reservation to the master node ends, and other nodes will be removed from the cluster when the head user’s reservation ends on those nodes.

We can now see the cluster we created in the example above using “manage_slurm list”:

```
CLUSTER: cbsum1c1b002
  Machine  CPUs  memory (GB)
  cbsum1c1b002  8    16
  cbsum1c1b003  8    16
  Users: mt269
```

We see that the cluster’s name is cbsum1c1002, and consists of two machines, each with 8 CPUs and 16Gb of memory. It currently has only one user who can submit jobs (mt269). If a user is added to the reservations for these two machines, they will automatically be added to the user list (within 5 minutes of reservation start).

Jobs can now be submitted to this cluster using the “sbatch” command, which takes as an argument the name of a script to execute. In the simplest form, you can just do “sbatch script.sh”. If you have access to more than one cluster, you should specify which cluster to submit to using the “--cluster” option (for example,
“sbatch --cluster cbsum1c1b002 script.sh”). By default, each job runs on a single core, and if there is an available core on the cluster, the job should start running immediately. You can submit many jobs, and they will wait in a queue until a core becomes available.

Many options are available to sbatch, and they can be specified on the command-line or using a special tag in the script. For example, to give a name to your job, you can use the option

"—job-name=<jobname>". This option can be given on the command line, or within the script on a line with the form:

```
#SBATCH --job-name=myJobName
```

Once the job is run, the output (by default) will be in a file with the name <jobname>-<jobID>.out. The job ID is a unique identifier assigned to the job when it is submitted.

Other useful options to sbatch include:

```
#SBATCH --job-name=jobname
#SBATCH --ntasks=1
#SBATCH --mem=8000
#SBATCH --output=jobname.out.%j
#SBATCH --mail-user=<email_address>
#SBATCH --mail-type=ALL
```

To describe the options above:

- --ntasks specifies how many cores the job should take.
- --mem can be used to specify how much memory a job needs (in Mb; so the above requests 8Gb).
- --output can be used to change the default output file; the %j will be automatically replaced by the job ID.
- --mail-user can be used to have SLURM send you emails.
- --mail-type=ALL says to send emails at job start, end, and crash. Other useful options to --mail-type are END, FAIL, BEGIN.

There are many other options to sbatch; a quick summary is available here: https://slurm.schedmd.com/pdfs/summary.pdf, and extensive documentation is available online.

Once jobs are running, you can monitor them with the command "squeue". You may want to see only your jobs, using "squeue -u $USER", or specify which cluster's queue to see with the --cluster option. You can cancel jobs with "scancel <jobID>".

Another useful feature is to get an interactive job through SLURM; this can be done by typing the command "salloc". This is particularly useful for testing a script before submitting a large number of jobs.