Linux for Biologists – Part 3

Robert Bukowski
Institute of Biotechnology
Bioinformatics Facility
(aka Computational Biology Service Unit - CBSU)

http://cbsu.tc.cornell.edu/lab/doc/Linux_workshop_Part3.pdf
Running applications
Running applications

- Very much like running system commands
- (Very) general syntax

\[
\text{<path_to_application_executable>} \ \text{<options>}
\]

- A few quick examples:

```
blastall -p blastx -b 1 -d ./databases/swissprot -i seq_tst.fa
```

```
samtools flagstat alignments.bam
```

```
tophat -p 7 -o B_L1-1 --transcriptome-index ZmB73_5a_WGS \ 
    --no-novel-juncs genome/maize reads_R1.fastq.gz reads_R2.fastq.gz
```
Why can we call, say, `samtools` by just typing `samtools` rather than the full path (in this case, `/programs/bin/samtools/samtools`)?

- Because of the `search path` environment variable which is defined for everybody. When you type `samtools`, the system tries each directory on the search path one by one until it finds the corresponding executable.

  - `which samtools` *(tells us where on disk the command bwa is located)*

  - `echo $PATH` *(displays the search path)*

- **Note**: the current directory `./` is **NOT** in the search path. If you need to run a program located, say in your home directory, you need to precede it with `./`, for example, `./my_program`

- **Note**: majority of executables are **NOT in search path** – they need to be launched using **full path**.
  - Visit [https://cbsu.tc.cornell.edu/lab/labsoftware.aspx](https://cbsu.tc.cornell.edu/lab/labsoftware.aspx) to find out the path to your application
Running applications

- How to run Java applications?
- Java programs usually come packaged in so-called jars
- Java program is launched by running the java virtual machine with the appropriate jar as an argument
- Example:

```
java -Xmx6g -jar GenomeAnalysisTK.jar -T UnifiedGenotyper \
-R genome.fa -i aln.bam -o variants.vcf
```

Launch Java with 6GB of RAM
Run program from this jar
Program options
Running applications

- Need to know what program(s) are relevant for your particular problem
- Need to know what a given program does and how to use it
  
  - Visit our software page [http://cbsu.tc.cornell.edu/lab/labsoftware.aspx](http://cbsu.tc.cornell.edu/lab/labsoftware.aspx)
  
  - Links to manuals (all options explained, examples given, test data available)
  
  - Specific hints on running in BioHPC Lab environment

- Getting quick help – run **command without any options**, or sometimes with `-h` or `--help`
  
  - Should print a list of options with very short descriptions
Running applications example: BLAST

- **Input:**
  - **FASTA file** with query sequences
    - We will use 9 random human cDNA sequences
  - **Database** of known sequences with which the query is to be compared
    - We will use *Swissprot* set of amino acid sequences
    - Need to translate each cDNA query in 6 frames and align to Swissprot templates

- **Output**
  - Text file describing hits

- **Program to run:** `blastall`
Running applications example: BLAST
prepare input

- Create your local scratch directory (if not yet done) and a sub-directory `blast_test` where this exercise will be run:

  ```
  mkdir /workdir/bukowski
  cd /workdir/bukowski
  mkdir blast_test
  cd blast_test
  ```

- Copy file with query sequences to the exercise directory:

  ```
  cp /shared_data/Linux_workshop/seq_tst.fa .
  ```

- Copy Swissprot BLAST database (we’ll make a separate directory for it):

  ```
  mkdir databases
  cp /shared_data/Linux_workshop/databases/swissprot* ./databases
  ```

- Verify that the files have been copied (use `ls` command)
Files frequently read and/or written (like input and output from an application being run) must be located on **local directories** (on BioHPC Lab machines: `/workdir`)
Running applications example: BLAST
run the program

Very general syntax for launching applications:

```
<path_to_application_executable>  [options] >& log
```

In our specific case:

```
blastall  -p blastx  -b 1  -d ./databases/swissprot  -i seq_tst.fa >& run.log
```

Options used:
- **-p:** type of search (blastx: compare 6-frame translations of DNA to AA sequences)
- **-b:** number of database sequences to show alignments for
- **-d:** path to database files
- **-i:** input file (with query sequences in fasta format)

For full set of options, run

```
blastall  | more
```
Running applications example: BLAST
running the program

```
blastall -p blastx -b 1 -d ./databases/swissprot -i seq_tst.fa >& run.log
```

- The program will run for about 1 minute and then write the output to the file `run.log` (STDOUT and STDERR streams combined)
  - Often output will appear in `run.log` gradually as a program is running

- For larger queries, the run will take (much) longer and produce more output...
  - 10,000 similar query sequences run using a similar command would take about 24 hours
Running a program, cnt.

- Running a program in the background
  - Normally, the program will run to completion (or crash), blocking the terminal window
  - By putting an “&” at the end of command, we can send the program to the background
    - Terminal prompt will return immediately – you will be able to continue working
    - Good for long-running programs (most programs of interest...)
    - Can run multiple programs simultaneously if more than 1 processor available on a machine (more about it later)
    - If all screen output redirected to disk, you may log out and leave the program running (to make sure, use nohup before the command)

```
blastall [options] >& run.log &
```

```
nohup blastall [options] >& run.log &
```
# Running applications

## Checking on your application: the `top` command

To exit – just type `q`.

<table>
<thead>
<tr>
<th>PID</th>
<th>USER</th>
<th>PR</th>
<th>NI</th>
<th>VIRT</th>
<th>RES</th>
<th>SHR</th>
<th>S</th>
<th>%CPU</th>
<th>%MEM</th>
<th>TIME+</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>18803</td>
<td>bukowski</td>
<td>20</td>
<td>0</td>
<td>288m</td>
<td>49m</td>
<td>45m</td>
<td>R</td>
<td>99.9</td>
<td>0.1</td>
<td>0:13.14</td>
<td>blastall</td>
</tr>
<tr>
<td>82</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:09.80</td>
<td>events/15</td>
</tr>
<tr>
<td>18804</td>
<td>bukowski</td>
<td>20</td>
<td>0</td>
<td>17424</td>
<td>1584</td>
<td>984</td>
<td>R</td>
<td>0.3</td>
<td>0.0</td>
<td>0:00.04</td>
<td>top</td>
</tr>
<tr>
<td>1</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>21454</td>
<td>1596</td>
<td>1284</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:01.51</td>
<td>init</td>
</tr>
<tr>
<td>2</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>kthreadd</td>
</tr>
<tr>
<td>3</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>migration/0</td>
</tr>
<tr>
<td>4</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>ksoftirqd/0</td>
</tr>
<tr>
<td>5</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>migration/0</td>
</tr>
<tr>
<td>6</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>watchdog/0</td>
</tr>
<tr>
<td>7</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>migration/1</td>
</tr>
<tr>
<td>8</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>migration/1</td>
</tr>
<tr>
<td>9</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.01</td>
<td>ksoftirqd/1</td>
</tr>
<tr>
<td>10</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.01</td>
<td>watchdog/1</td>
</tr>
<tr>
<td>11</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>migration/2</td>
</tr>
<tr>
<td>12</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>migration/2</td>
</tr>
<tr>
<td>13</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0.00.03</td>
<td>ksoftirqd/2</td>
</tr>
<tr>
<td>14</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>watchdog/2</td>
</tr>
<tr>
<td>15</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>migration/3</td>
</tr>
<tr>
<td>16</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>migration/3</td>
</tr>
<tr>
<td>17</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>ksoftirqd/3</td>
</tr>
<tr>
<td>18</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>watchdog/3</td>
</tr>
<tr>
<td>19</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>migration/4</td>
</tr>
<tr>
<td>20</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>migration/4</td>
</tr>
<tr>
<td>21</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>ksoftirqd/4</td>
</tr>
<tr>
<td>22</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.04</td>
<td>watchdog/4</td>
</tr>
<tr>
<td>23</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>migration/5</td>
</tr>
<tr>
<td>24</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.00</td>
<td>migration/5</td>
</tr>
</tbody>
</table>
## Running applications, cnt.

### Checking on your application:
the `ps` command – display info about all your processes – one of them should be blastall

```
ps -ef | grep bukowski
```

<table>
<thead>
<tr>
<th>Process ID (PID)</th>
<th>Running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>root 8263</td>
<td>09:00:00 sshd: bukowski [priv]</td>
</tr>
<tr>
<td>bukowski 8266</td>
<td>09:00:02 sshd: bukowski@pts/0</td>
</tr>
<tr>
<td>bukowski 8267</td>
<td>09:00:00 -bash</td>
</tr>
<tr>
<td>bukowski 9258</td>
<td>09:00:00 screen</td>
</tr>
<tr>
<td>bukowski 9259</td>
<td>09:00:01 SCREEN</td>
</tr>
<tr>
<td>bukowski 9260</td>
<td>09:00:00 /bin/bash</td>
</tr>
<tr>
<td>bukowski 9284</td>
<td>09:00:00 /bin/bash</td>
</tr>
<tr>
<td>bukowski 9307</td>
<td>09:00:00 /bin/bash</td>
</tr>
<tr>
<td>bukowski 18815</td>
<td>09:00:00 /bin/bash /run.sh</td>
</tr>
<tr>
<td>bukowski 18817</td>
<td>09:00:08 /programs/bin/blast/blastall -p blastx -b 1-d ./database/swissprot -i seq_tst.fa</td>
</tr>
<tr>
<td>bukowski 18818</td>
<td>09:00:00 ps -ef</td>
</tr>
<tr>
<td>bukowski 18819</td>
<td>09:00:00 grep bukowski</td>
</tr>
</tbody>
</table>

Try `man ps` for more info about the `ps` command.
Running applications

- Stopping applications

  - If the application is running in the foreground (i.e., without “&”), it can be stopped with Ctrl-C (press and hold the Ctrl key, then press the “C” key) issued from the window (terminal) it is running in.

  - If the application is running in the background (i.e., with “&”), it can be stopped with the `kill` command

    \[
    \text{kill } -9 \text{ <PID>}
    \]

    Where <PID> is the process id obtained from the `ps` command. For example, to terminate the `blastall` process from the previous slide, we would use

    \[
    \text{kill } -9 \text{ 18817}
    \]

    Try `man kill` for more info about the `kill` command.
Keeping a program running in the background after you log out or disconnect

**Option 1:** Use `nohup` (as on previous slide). Of course, you can use this also with options 1 and 2.

**Option 2:** Start a program in a terminal within a **VNC session**
- the session keeps running after VNC connection is killed
- you can reconnect to VNC session later

**Option 3:** Start a program within a **screen** window
- all such windows keep running after you disconnect using “Ctrl-a d” or by killing terminal window
- you can reconnect to the whole session later
Shell scripting

Example we already talked about: Downloading Illumina sequencing results

Script **download.sh** is sent as attachment to notification e-mail from the sequencing facility.

```bash
#!/bin/bash

# Collect all the wget commands from notification e-mails
# (make sure there are no blank spaces after "\")

wget -c -q -O 783_3_3944_N_PhiX_R1.fastq.gz \ 
wget -c -q -O 783_3_3944_N_PhiX_R2.fastq.gz \ 
wget -c -q -O 783_3_3944_N_Sample_R1.fastq.gz \ 
wget -c -q -O 783_3_3944_N_Sample_R2.fastq.gz \ 
```

Copy **download.sh** to your Linux machine and run as a script

```
sh ./download.sh
```
Example: given Illumina reads (in FASTQ format) and reference genome (FASTA), call SNPs

1. Index genome (bwa)
2. Reference BWA index files
3. Align reads to reference (bwa)
4. Alignment (aln.sam)
5. Convert to BAM format (samtools)
6. Alignment (aln.bam)
7. Sort alignments (samtools)
8. Sorted alignments (aln_srt.bam)
9. Index BAM file (samtools)
10. Index genome (bwa)
11. Index genome (bwa)
12. Align reads to reference (bwa)
13. Alignment (aln.sam)
14. Convert to BAM format (samtools)
15. Alignment (aln.bam)
16. Sort alignments (samtools)
17. Sorted alignments (aln_srt.bam)
18. Index BAM file (samtools)
19. Get genotype lkhds (samtools, bcftools)
20. Raw genotyping result (var.raw.vcf)
21. SNP filtering (bcftools)
22. Final SNPs (var.flt.vcf)
Scripts: tools for executing complex tasks

- Sequence of steps on previous slide is an example of a pipeline

- Each step corresponds to (typically) one instance of a program or command

- Input files used in a step are (typically) generated in preceding steps

- Some steps may run quite long (depends on amount of input data and size of reference)

- Executing each step in a terminal as a command is possible, but tedious and hard to repeat (for example, with a new input data)

- Remedy: write a shell script – a text file with commands
Shell script: a set of commands (and comments) in a text file

This is a fragment of an actual script implementing the SNP-calling pipeline.

Run the whole script as homework – see the end of this presentation.
Shell scripts

- First line should be `#!/bin/bash` (indicates the shell used to interpret the script)
  - If absent, default shell will be used (bash)

- Everything in a line following “#” is a comment

- May include system commands (like `cp`, `mv`, `mkdir`, …) and commands launching programs (`blastall`, `bwa`, `samtools`, …)

- Commands will be executed “in the order of appearance”

- Long lines can be broken with “\” character
  - The “\” character must be the last one in a line (no blank spaces after it)

- Script (e.g., `my_script.sh`, in the current directory) can be run as in the following:

  ```bash
  bash ./my_script.sh >& my_script.log &
  ./my_script.sh >& my_script.log  &
  ```

- The second command will work if the file `my_script.sh` is made executable with the command

  ```bash
  chmod u+x my_script.sh
  ```
Shell scripts: conditionals and loops

```bash
#!/bin/bash

# Example of a conditional statement
if [ -e file*.txt ]
then
  echo File file.txt exists
else
  echo File file.txt does not exist
fi

#!/bin/bash

# Example of a loop

# For each file with name ending with "*.txt"
# count the files and compress the file
for i in *.txt
do
  wc ${i}
gzip ${i}
done

# Another loop example:
# Create 10 directories called dir1, dir2, ..., dir10

for i in {1..10}
do
  mkdir dir${i}
done
```
Exercise
(see end of slide deck)

simple SNP-calling pipeline

Objective: align (simulated) Illumina reads to D. Melanogaster genome using **BWA** aligner and call variants using **samtools**
More about scripting

Multiple scripting tools available

- **shell** (bash, tcsh – good for stitching together shell commands)
- **perl** (very popular in biology, due to BioPerl module package)
- **python** (good numerical analysis tools – NumPy, SciPy packages)
- **awk** (mostly text parsing and processing)
- **sed** (mostly text parsing and processing)
- **R** (rich library of numerical analysis and statistical functions)
Using multiple processors

Recommended reading:
Efficient use of CPUs/cores on BioHPC Lab machines
http://cbsu.tc.cornell.edu/lab/doc/using_BioHPC_CPUs.pdf
Multiple processors

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).

<table>
<thead>
<tr>
<th>machine</th>
<th>CPU available</th>
<th>cores available</th>
<th>cores used</th>
<th>time (hrs)</th>
<th>speedup (in machine)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbsulm10</td>
<td>4</td>
<td>64</td>
<td>64</td>
<td>0.931</td>
<td>27.506</td>
</tr>
<tr>
<td>cbsulm10</td>
<td>4</td>
<td>64</td>
<td>16</td>
<td>1.962</td>
<td>13.056</td>
</tr>
<tr>
<td>cbsulm10</td>
<td>4</td>
<td>64</td>
<td>1</td>
<td>25.619</td>
<td>1.000</td>
</tr>
<tr>
<td>cbsumm15</td>
<td>2</td>
<td>24</td>
<td>24</td>
<td>2.058</td>
<td>12.117</td>
</tr>
<tr>
<td>cbsumm15</td>
<td>2</td>
<td>24</td>
<td>12</td>
<td>2.593</td>
<td>9.616</td>
</tr>
<tr>
<td>cbsumm15</td>
<td>2</td>
<td>24</td>
<td>1</td>
<td>24.930</td>
<td>1.000</td>
</tr>
<tr>
<td>cbsum1c2b008</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>4.193</td>
<td>6.717</td>
</tr>
<tr>
<td>cbsum1c2b008</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>28.161</td>
<td>1.000</td>
</tr>
</tbody>
</table>

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).

<table>
<thead>
<tr>
<th>machine</th>
<th>CPU available</th>
<th>cores available</th>
<th>cores used</th>
<th>time (hrs)</th>
<th>speedup (in machine)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbsulm10</td>
<td>4</td>
<td>64</td>
<td>64</td>
<td>10.97</td>
<td>2.222</td>
</tr>
<tr>
<td>cbsulm10</td>
<td>4</td>
<td>64</td>
<td>16</td>
<td>24.37</td>
<td>1.000</td>
</tr>
<tr>
<td>cbsumm15</td>
<td>2</td>
<td>24</td>
<td>24</td>
<td>26.10</td>
<td>2.140</td>
</tr>
<tr>
<td>cbsumm15</td>
<td>2</td>
<td>24</td>
<td>12</td>
<td>55.85</td>
<td>1.000</td>
</tr>
</tbody>
</table>
Multiple processors

- 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. Most often it is less than expected, but still sufficiently large to justify the effort. 64 cores compared to 1 core in swissprot example give 27.5 speedup rate, much less than 64-fold, but still large!

- Speedup depends on the machine (hardware), program (algorithm), and parameters (e.g., nr vs swissport). When using **nr** database on cbsumm15 the speedup between 12 and 24 cores is 2.14. For **swissprot** on the same machine it is only 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.
Multiple processors

Three ways to utilize multiple CPU cores on a machine:

- Using a given program’s built-in parallelization

- Simultaneously executing several programs in the background

- Using a “driver” program to execute multiple tasks in parallel
Multiple processors

- Take advantage of a program’s built-in parallelism **invoked with an option**
  - read documentation to find out if your program has this feature
  - Look for keywords like “multithreading”, “parallel execution”, “multiple processors”, etc.

A few examples:

- blastall \(-a\ 8\) [other options]
- blast+ \(-\text{num\_threads}\ 8\) [other options]
- tophat \(-p\ 8\) [other options]
- cuffdiff \(-p\ 8\) [other options]
- bwa \(-t\ 8\) [other options]
- bowtie \(-p\ 8\) [other options]

Remember speedup is not perfect, so optimal number of threads needs to be optimized by trial and error using subset of input data
Multiple processors

blastall -a 2 -p blastx -b 1 -d ./databases/swissprot -i seq_tst.fa

- >100% CPU indicates the program is **multithreaded**
  - Multiple threads within a single process rather than multiple processes
Multiple processors

- Simultaneously executing several programs in the background

Example: suppose we have to compress (gzip) several files. We can simply launch multiple `gzip` commands in the background, without waiting for previous ones to finish:

```bash
gzip file1 &
gzip file2 &
gzip file3 &
```

Multiple processes
(1 thread in each)
Multiple processors

What if in the previous example, we had, say, 3000 files instead of just 3, but still only a few processors?

Submitting all 3000 commands simultaneously in the background (in principle, it could be done painlessly using a script) would not work too well, because:

- Each processor would have to switch between many processes – possible, but inefficient
- With large number (and/or size) of files being processed, access to disk would become a bottleneck (i.e., processes would spend most of their time competing for access to disk)
  - Disk access (referred to as I/O – input/output) is always an issue for programs which do a lot of reading/writing (like gzip)
- As a result, we would get no speedup, or (more likely) processing of all files in parallel would take longer than processing them one by one

In situations like this (many short tasks and a few processors), we need a special “driver” tool to efficiently distribute the tasks.
Multiple processors

- Using a “driver” program to execute multiple tasks in parallel

Example: create a file called (for example) **TaskFile**
(This is **NOT** a script, although it could be executed as such...)

This long file can be created, for example, using the following shell script:

```bash
#!/bin/bash
rm -f TaskFile
for i in {1..3000}
  do
echo gzip file$i >> TaskFile
done
```

..... (up to **file3000**)
Multiple processors

Then run the command (assuming the TaskFile and all file* files are in the current dir)

```
/perlfork_univ.pl TaskFile NP >& log &
```

where NP is the number of processors to use (e.g., 10)

- perl_fork_univ.pl is an CBSU in-house “driver” script (written in perl)
- It will execute tasks listed in TaskFile using up to NP processors
  - The first NP tasks will be launched simultaneously
  - The (NP+1)th task will be launched right after one of the initial ones completes and a “slot” becomes available
  - The (NP+2)nd task will be launched right after another slot becomes available
  - ...... etc., until all tasks are distributed
- Only up to NP tasks are running at a time (less at the end)
- All NP processors always kept busy (except near the end of task list) – Load Balancing
Mixed parallelization: running several simultaneous multi-threaded tasks (each processing different data) on a large machine (here: 64-core)

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 &
Multiple processors

General guidelines

- Do not run more processes/threads than CPU cores available on the machine
  - For large number of tasks, use script `perl_fork_univ.pl`

- Run only as many simultaneous processes as will fit in memory (RAM)
  - when in doubt, run a single process first and check its memory requirement (for example, using `top`)

- Programs heavy on I/O will compete for disk access if run in parallel – running too many simultaneously is not a good idea

- If available, use program’s own multithreading options

- Using subset of input data, try to determine number of CPU cores which (for a given machine, input, and program options) gives the optimal speedup.
Exercises
Exercise: simple SNP-calling pipeline

Objective: align (simulated) Illumina reads to D. Melanogaster genome using BWA aligner and call variants using samtools

1. Copy the input data and shell script to your local working directory (replace my_id with your login ID):

   ```
   mkdir /workdir/my_id
   cd /workdir/my_id
   cp /shared_data/Linux_workshop/pipeline_example.tgz .
   tar -xzvf pipeline_example.tgz
   ```

2. Using commands like more, tail, head, wc,... to examine the sequence files (genome.fa – this is the reference genome; reads.fastq – these are the simulated Illumina reads), e.g.,

   - `grep "">" genome.fa | wc` (will count chromosomes in genome)
   - `wc reads.fastq` (the first number divided by 4 is the number of reads)
Exercise: simple SNP-calling pipeline

3. Open the file `pipeline.sh` in a text editor of your choice. Examine the structure of this file. Based on comments, identify commands corresponding to steps from slide “Complex task example: SNP-calling”

4. Run the pipeline in the background, saving any screen output to a log file. The run should take about 15 minutes.

   ```
   cd /workdir/my_id
   ./pipeline.sh >& pipeline.log &
   ```

5. Use the `top`, `ps`, and `ls` commands to monitor the progress of the pipeline (processes and files).

6. List the generated output files and confront with script `pipeline.sh`

7. Using a text editor, examine the log file `pipeline.log`. Can you identify messages from individual commands in the script?

8. Using a text editor or text browsing commands (more, head, tail, etc) examine the alignment file (`aln.sam`) and final variant output file `var.flt.vcf`. You may want to look up the SAM and VCF format specifications (see [http://samtools.sourceforge.net](http://samtools.sourceforge.net/) for quick reference).
Exercise: connect to your assigned workstation using VNC

• Go to “My Reservations” page
  http://cbsu.tc.cornell.edu/lab/lab.aspx, log in, click on “My Reservations” menu link

• Choose resolution (depends on your monitor)

• Click on “Connect VNC”

• Follow prompts to connect your VNC client to your VNC session

• Open terminal window in the VNC desktop by right-click on the desktop background and choosing “Open Terminal”.

• Disconnect (close VNC window) and then reconnect. Is the session still alive?