Parallel Processing and Load Balancing

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


Contact: brc_bioinformatics@cornell.edu
Motivation

- Can you solve a ‘big task’ on your laptop? Not really...
  - too small: not enough memory, not enough disk to store big data
  - too slow: analysis would take forever

- You need a more powerful resource
  - bigger: more memory, more disk
  - FASTER!!!

- What does FASTER mean?
  - faster processor (and other hardware) – yes, but first of all....
  - MORE processors !!!
  - knowledge how to use it all
## BioHPC renatal resources

<table>
<thead>
<tr>
<th>Server type</th>
<th>#servers</th>
<th>#cores</th>
<th>RAM [GB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>interactive</td>
<td>4</td>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td>general</td>
<td>32</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>medium gen1</td>
<td>1</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>24</td>
<td>128</td>
</tr>
<tr>
<td>medium gen2</td>
<td>12</td>
<td>40</td>
<td>256</td>
</tr>
<tr>
<td>large gen1</td>
<td>8</td>
<td>64</td>
<td>512</td>
</tr>
<tr>
<td>large gen2</td>
<td>2</td>
<td>96</td>
<td>512</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>112</td>
<td>512</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>80</td>
<td>512</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>88</td>
<td>512</td>
</tr>
<tr>
<td>extra-large</td>
<td>1</td>
<td>64</td>
<td>1,024</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>112</td>
<td>1,024</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>88</td>
<td>1,024</td>
</tr>
<tr>
<td>GPU gen2</td>
<td>2</td>
<td>32</td>
<td>256</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>89</strong></td>
<td><strong>3,056</strong></td>
<td><strong>19,104</strong></td>
</tr>
</tbody>
</table>

*your workshop machines*
Given:

- ‘big task’ at hand
- multiple CPUs, RAM, and disk storage, possibly scattered across multiple networked computers

Objective:

- Parallelize: solve the ‘big task’ in time shorter than it would take using a single CPU on a single computer
- Balance load: keep resources busy, but not overloaded
Some basic hardware facts

Some basic software facts

Parallelization: problems and tools

Monitoring and timing Linux processes

Multiple independent tasks
  - Load balancing

Next week:
  - Advanced load balancing using job scheduler (SLURM = Simple Linux Utility for Resource Management)

**Hands-on exercises:** will introduce some tools and techniques (although quantitative conclusions doubtful in shared environment... )
**Resources on a single machine (here: cbsumm12)**

**CPU**: an integrated circuit (a “chip”) containing computational hardware. May be more than one per server, typically 2-4.

**Core**: a subunit of CPU capable of executing an independent sequence of instructions (a **thread**). Shares communication infrastructure and internal memory with other cores on the CPU.

\[
\text{# threads possible to run at the same time} = \text{# cores}
\]

**Hyperthreading (HT)**: technology to simultaneously run several (typically – two) independent sequences of instructions (**threads**) on each core, sharing the core’s hardware; may be **disabled** or **enabled**.

- If HT enabled, **core** is understood as **hyperthreaded core**
- In this example, with HT enabled, **# cores =24**

**RAM**: memory. All accessible to all cores (but easier to access CPU’s ‘own’ portion); **Cache**: fast-access (but small) memory ‘close’ to CPU.

---

**Diagram**:
- **CPU 1 (in ‘slot’ or ‘socket’)**
  - RAM 64GB
  - Cache 15MB
  - 4 cores
- **CPU 2 (in ‘slot’ or ‘socket’)**
  - RAM 64GB
  - Cache 15MB
  - 4 cores
- **Motherboard**
- **Local disk storage**
  - /workdir
  - fast
- **Network disk storage**
  - /home
  - slow
Check on CPU configuration with `lscpu`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture:</td>
<td>x86_64</td>
</tr>
<tr>
<td>CPU op-mode(s):</td>
<td>32-bit, 64-bit</td>
</tr>
<tr>
<td>Byte Order:</td>
<td>Little Endian</td>
</tr>
<tr>
<td>CPU(s):</td>
<td>24</td>
</tr>
<tr>
<td>On-line CPU(s) list:</td>
<td>0-23</td>
</tr>
<tr>
<td>Thread(s) per core:</td>
<td>2</td>
</tr>
<tr>
<td>Core(s) per socket:</td>
<td>6</td>
</tr>
<tr>
<td>Socket(s):</td>
<td>2</td>
</tr>
<tr>
<td>NUMA node(s):</td>
<td>2</td>
</tr>
<tr>
<td>Vendor ID:</td>
<td>GenuineIntel</td>
</tr>
<tr>
<td>CPU family:</td>
<td>6</td>
</tr>
<tr>
<td>Model:</td>
<td>45</td>
</tr>
<tr>
<td>Model name:</td>
<td>Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz</td>
</tr>
<tr>
<td>Stepping:</td>
<td>7</td>
</tr>
<tr>
<td>CPU MHz:</td>
<td>2500.122</td>
</tr>
<tr>
<td>CPU max MHz:</td>
<td>2500.0000</td>
</tr>
<tr>
<td>CPU min MHz:</td>
<td>1200.0000</td>
</tr>
<tr>
<td>BogoMIPS:</td>
<td>4000.35</td>
</tr>
<tr>
<td>Virtualization:</td>
<td>VT-x</td>
</tr>
<tr>
<td>L1d cache:</td>
<td>32K</td>
</tr>
<tr>
<td>L1i cache:</td>
<td>32K</td>
</tr>
<tr>
<td>L2 cache:</td>
<td>256K</td>
</tr>
<tr>
<td>L3 cache:</td>
<td>15360K</td>
</tr>
<tr>
<td>NUMA node0 CPU(s):</td>
<td>0-5,12-17</td>
</tr>
<tr>
<td>NUMA node1 CPU(s):</td>
<td>6-11,18-23</td>
</tr>
<tr>
<td>Flags:</td>
<td>fpu vme de pse tsc</td>
</tr>
</tbody>
</table>

- Hyper-threaded cores
- Hyperthreading ON
- CPUs
Check memory using `free`

```
[bukowski@cbsumm12 ~]$ free -m

            total    used    free     shared    buff/cache  available
Mem:  128738   1772  123220      1550       3746       124575
Swap:   4095    836   3259
```

show in Mbytes
(other options: `-k -g`)

Not always correct, unfortunately...
Check disk storage using `df`

```bash
[root@cbsumm12 ~]# df -h

Filesystem                  Size  Used  Avail Use% Mounted on
/dev/mapper/rhel_cbsumm12-root   300G   32G   269G  11%  /
devtmpfs                       63G     0   63G   0%  /dev
tmpfs                           63G     0   63G   0%  /dev/shm
tmpfs                           63G   2.1G   61G   4%  /run
 tmpfs                           63G     0   63G   0%  /sys/fs/cgroup
/sys/fs/cgroup                 871G   72M   827G   1%  /SSD
/dev/md126                     3.4T  130G   3.3T   4%  /local
/dev/sda2                      494M  144M   351M  30%  /boot
tmpfs                           13G   16K   13G   1%  /run/user/42
tmpfs                           13G     0   13G   0%  /run/user/0
128.84.180.177@tcp1:128.84.180.176@tcp1:/lustrel 1.3P 1003T  300T  78%  /home
cbsugfs1:/home                 233T  135T   99T  58%  /home
/glusterfs/home                13G     0   13G   0%
```

Local scratch space (fast, temporary)

Network-mounted (slow, permanent).

NO I/O-intensive computations there!
Check other hardware using `lspci`

PCI = **Peripheral Component Interconnect**

Most devices are attached this way

`lspci` produces long output, better paginate or filter, e.g.,

```
[root@cbsumm12 ~]# lspci | grep -i raid
00:1f.2 RAID bus controller: Intel Corporation C600/X79 series chipset SATA RAID Controller (rev 06)

[cryosparc_user@cbsugpu03 ~]$ lspci | grep -i nvidia
02:00.0 3D controller: NVIDIA Corporation GP100GL [Tesla P100 PCIe 16GB] (rev a1)
83:00.0 3D controller: NVIDIA Corporation GP100GL [Tesla P100 PCIe 16GB] (rev a1)
```
What’s running on a machine

- Everything ‘running on a machine’ (apps run by users, OS tasks) does this by means of **processes**

- **Process**: instruction sequence loaded into memory, to be executed by CPU cores, using some memory to store code (text) and data, communicating with peripherals (disk storage, network, …)

- Templates for processes stored on disk as **executable files**

- Process may contain **one or more threads** (multithreading), all with **access to the same data** (but not to data of other processes)

- Each process has a unique **process ID** (and so do individual threads)

- Each process is created by another process - its **parent** process (thus, there is a **process tree**)

- Each process (with all its threads) runs on a **single machine**
Cores and processes: mixing it all together

- At any given time, a **core** can be
  - executing one thread
  - idle

- At any given time, a **thread** can be
  - running on one of the cores
  - waiting off-core (for input or data from memory or disk, or for an available core)
  - stopped on purpose

- **Load**: number of threads running + waiting for a core to run on (should not exceed number of cores!)

- **Context switches**
  - a core executes one thread for a while, then switches to another (state of the previous one is saved to be resumed later)
  - some threads have higher priority (like quick housekeeping tasks by OS)
  - threads are only allowed to run for some time without being switched out
  - frequent context switches **not good for performance** (occur at high load)

- **Scheduler** (part of **Linux kernel**) takes care of distributing **threads** over **cores**
  - (not to be confused with **SLURM job scheduler** discussed in Part 2)
Software structure

**Kernel mode**

- **Linux kernel**
  - system call interface (SCI) – used by processes
  - process scheduling
  - inter-process communication tools (IPC)
  - memory management
  - interface to hardware (drivers)

**User mode**

- **User applications** (bash, python, bwa, FireFox, ssh, VNC, blast+...)
- **Low-level system components** (init, services, logind, networkd, X11,...)
- **C standard library** (processes communicating with kernel)

**Hardware:** CPUs, memory, disk storage, other peripherals
Cores and processes: mixing it all together

- Typically, there are many more threads than cores:

Example: empty (i.e., no users) machine **cbsumm12** (24 cores), some time last Saturday:

```plaintext
ps -ef | wc -l : 596 (all processes)
ps -efL | wc -l : 919 (all threads)
```

these are processes that keep the OS running
- mostly waiting for stuff to help with, clean up, running only when needed
- consume very few CPU cycles and little RAM

- Despite large number of threads, the **load** on the machine was very low, and most memory was available:

```plaintext
uptime
10:37:32 up 265 days, 14:57, 3 users, load average: 0.08, 0.21, 1.11
```

```plaintext
free -m
Mem: 128738 1772 123220 1550 3746 124575
```

Almost all CPU and memory resources up for grabs by users’ programs
Given a ‘big task’ at hand, make multiple CPU cores work in parallel to achieve the solution in time shorter than what would be needed if only a single core were used.

Constraints:

- CPUs and memory possibly scattered over multiple networked machines
- Core number and memory limits on individual machines
- A process (with all its threads and memory) can only run on one machine
- No direct data sharing between processes

Parallelize the problem!
Parallelizing a problem: a silly (but complex) example

Sum up a bunch of numbers (here: from 1 to 8) and calculate the exponential of the sum using 4 threads.

Sequential (i.e., non-parallelizable) part
# Programmer’s perspective: planning complex parallelization

## Algorithm design
- Identify parallelizable portions of the problem
- Minimize the sequential (non-parallelizable) part
- Consider/minimize synchronization and inter-thread communication
- Avoid race conditions
- Avoid simultaneous I/O by multiple threads
- Threads organization
  - Single process with multiple threads
  - Multiple single-threaded processes
  - Multiple multi-threaded processes

## Constraints
- CPUs and memory possibly scattered over multiple networked machines
- \#threads <= \#cores (on each machine)
- Combined memory taken up by all processes not to exceed total machine’s memory
- Storage capacity and access
- A process (with all its threads and memory) can only run on one machine
- No direct data sharing between processes
Programmer’s perspective: tools

For complicated algorithms with varying levels of parallelism and communication, programs are typically written using appropriate parallelization tools (libraries of functions). By design, these programs fall into one of the following categories:

- Single multi-threaded process (by far the largest class)
  - Sometimes called shared memory model
  - Tools: pthreads, OpenMP
  - Advantage: all threads have access to same memory – no or easy communication
  - Disadvantage: can only run on one machine (but really no problem if machine huge)

- Multiple single-thread processes
  - Sometimes called distributed memory model
  - Tools: Message-Passing Interface (MPI) (Implementations: OpenMPI, mpich2)
  - Advantage: can run on a single machine and/or across multiple machines
  - Disadvantage: no direct access to process memory by other processes – data must be passed using messages – costly, especially between machines

- Multiple multi-threaded processes
  - Tools: combination of OpenMP, pthreads, MPI
  - Advantages: optimized, high-level parallelism possible
  - Advantage: can run on a single machine and/or across multiple machines
Find out how a program is parallelized
(easy only for executables using shared libraries)

[root@cbsuxm01 ~]# ldd /programs/bin/blast+/blastx

    linux-vdso.so.1 =>  (0x00007ffd0f79b000)
    libpthread.so.0 => /lib64/libpthread.so.0 (0x00007facee470000)
    libz.so.1 => /lib64/libz.so.1 (0x00007facee25a000)
    libbz2.so.1 => /lib64/libbz2.so.1 (0x00007facee04a000)
    libdl.so.2 => /lib64/libdl.so.2 (0x00007facede46000)
    libnsl.so.1 => /lib64/libnsl.so.1 (0x00007facedc2c000)
    libm.so.6 => /lib64/libm.so.6 (0x00007faced92a000)
    libc.so.6 => /lib64/libc.so.6 (0x00007faced55c000)
    libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007faced346000)
    /lib64/ld-linux-x86-64.so.2 (0x00007facee68c000)

[root@cbsuxm01 ~]# ldd /programs/discovar/bin/Discovar

    linux-vdso.so.1 =>  (0x00007fff1cd8a000)
    libstdc++.so.6 => /lib64/libstdc++.so.6 (0x00007fc9ba791000)
    libm.so.6 => /lib64/libm.so.6 (0x00007fc9ba48f000)
    libgomp.so.1 => /lib64/libgomp.so.1 (0x00007fc9ba269000)
    libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007fc9ba053000)
    libpthread.so.0 => /lib64/libpthread.so.0 (0x00007fc9b9e37000)
    libc.so.6 => /lib64/libc.so.6 (0x00007fc9b9a6a000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fc9b98000)
Find out how a program is parallelized
(easy only for executables using shared libraries)

```bash
[root@cbsuxm01 ~]# ldd /programs/ima2p/bin/IMa2p

linux-vdso.so.1 => (0x00007fffd843b9000)
libm.so.6 => /lib64/libm.so.6 (0x00007f5100e44000)
libmpi_cxx.so.1 => /usr/lib64/openmpi/lib/libmpi_cxx.so.1 (0x00007f5100c29000)
libmpi.so.12 => /usr/lib64/openmpi/lib/libmpi.so.12 (0x00007f5100945000)
libstdc++.so.6 => /lib64/libstdc++.so.6 (0x00007f510063e000)
libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007f5100428000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f51000428000)
libc.so.6 => /lib64/libc.so.6 (0x00007f50fe1a46000)
/lib64/ld-linux-x86-64.so.2 (0x00007f50fe1a46000)
libopen-rte.so.12 => /usr/lib64/openmpi/lib/libopen-rte.so.12 (0x00007f50ffbc3000)
libopen-pal.so.13 => /usr/lib64/openmpi/lib/libopen-pal.so.13 (0x00007f50ff91f000)
libdl.so.2 => /lib64/libdl.so.2 (0x00007f50fee7b000)
librt.so.1 => /lib64/librt.so.1 (0x00007f50ff513000)
libutil.so.1 => /lib64/libutil.so.1 (0x00007f50ff310000)
libhwloc.so.5 => /lib64/libhwloc.so.5 (0x00007f50ff0d3000)
libnuma.so.1 => /lib64/libnuma.so.1 (0x00007f50feec7000)
libldl.so.7 => /lib64/libldl.so.7 (0x00007f50fecd000)
```
Amdahl’s Law: More threads not always better

Suppose the total execution time of a program consists of non-parallelizable part \( t_{seq} \) and a part that can be parallelized, \( t_{par} \). Then for number of threads \( N \) we have

Time on a single thread: \( T_1 = t_{seq} + t_{par} \)

Time on \( N \) threads: \( T_N = t_{seq} + \frac{t_{par}}{N} \) (assuming no communication or other delays)

Speedup on \( N \) threads: \( S_N = \frac{T_1}{T_N} \rightarrow 1 + \frac{t_{par}}{t_{seq}} \)

Ideal: linear scaling

Performance deterioration possible due to sync/communication/IO
Example: speedup in BLAST

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>
Parallelizing a problem: ‘embarrassingly parallel’ case

Simple! No communication, no sync, no sequential part – least susceptible to Amdahl’s law

BUT

Threads compete for disk access if too many!
Parallelizing a problem: ‘not so embarrassingly parallel’ case

Simple! No communication **between processes**, no sync

**BUT**

Processes compete form disk access if too many!

Complex multi-threaded processes, independent from one another
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 \
   tophat -p 7 -o B_L1-2 --transcriptome-index genome/transcriptome/ZmB73_5a_WGS \
   --no-novel-juncs genome/maize \
   fastq/2284_6063_7073_C3AR7ACXX_B_L1-1_ATCACG_R2.fastq.gz >& B_L1-1.log &
   faster than tophat -p 63!
```

```
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 \
   tophat -p 7 -o L_L1-2 --transcriptome-index genome/transcriptome/ZmB73_5a_WGS \
   --no-novel-juncs genome/maize \
   fastq/2284_6063_7074_C3AR7ACXX_L_L1-1_CGATGT_R2.fastq.gz >& L_L1-1.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 \
   tophat -p 7 -o S_L1-2 --transcriptome-index genome/transcriptome/ZmB73_5a_WGS \
   --no-novel-juncs genome/maize \
   fastq/2284_6063_7075_C3AR7ACXX_S_L1-1_TTAGGC_R2.fastq.gz >& S_L1-1.log &
```

```
tophat -p 7 -o S_L1-3 --transcriptome-index genome/transcriptome/ZmB73_5a_WGS \
   --no-novel-juncs genome/maize \
   fastq/2284_6063_7078_C3AR7ACXX_S_L1-3_GCCAAT_R1.fastq.gz \
   tophat -p 7 -o S_L1-4 --transcriptome-index genome/transcriptome/ZmB73_5a_WGS \
   --no-novel-juncs genome/maize \
   fastq/2284_6063_7078_C3AR7ACXX_S_L1-3_GCCAAT_R2.fastq.gz >& S_L1-2.log &
```

```
tophat -p 7 -o S_L1-5 --transcriptome-index genome/transcriptome/ZmB73_5a_WGS \
   --no-novel-juncs genome/maize \
   fastq/2284_6063_7081_C3AR7ACXX_S_L1-5_GATCAG_R1.fastq.gz \
   tophat -p 7 -o S_L1-6 --transcriptome-index genome/transcriptome/ZmB73_5a_WGS \
   --no-novel-juncs genome/maize \
   fastq/2284_6063_7081_C3AR7ACXX_S_L1-5_GATCAG_R2.fastq.gz >& S_L1-3.log &
```
Common situation in ‘end user’ bioinformatics

- Instances of complex, multi-threaded applications run concurrently on distinct sets of input data
  - Examples: BLAST, bwa, tophat, STAR, Trinity, .....  
  - applications ‘pre-programmed’ for us by software developers

- What we need to know about each instance of the application
  - how to run the application, know/control number of threads it uses
  - memory, disk, disk I/O, time requirements of the application (may depend on number of threads)
  - optimal number of threads for given input data, machine
  - Run, monitor, observe, extrapolate...

- Load balancing: How to manage multiple instances subject to resource constraints
  - (#instances) X (#threads_per_instance) < #cores on each machine
  - (memory_per_instance) X (#instances) < total_machine_memory
  - competition for I/O bandwidth
  - sufficient scratch disk storage
Running multi-threaded applications

Parallelism is typically controlled by a program 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]
- blastx -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.
Running multi-threaded applications

blastx -num_threads 2 -db ./databases/swissprot -query seq_tst.fa

- >100% CPU indicates the program is **multithreaded**
  - Multiple **threads** within a **single process** rather than multiple processes
What if the number of threads is not specified?

Default number of threads for a multi-threaded programs

- Depends on the program’s author(s)
- Sometimes 1
- Sometimes equal to the number of cores found on machine (rather nasty in shared environment)
- Programs parallelized with OpenMP ‘obey’ environment variable $\texttt{OMP\_NUM\_THREADS}$

```
export OMP_NUM_THREADS=10
```

will make such program use up to 10 threads (BioHPC default: 1)

- Programs parallelized with Intel’s Math Kernel Library (MKL) require variable $\texttt{MKL\_NUM\_THREADS}$ (BioHPC default: 1) in addition to $\texttt{OMP\_NUM\_THREADS}$

- Programs parallelized with pthreads: you are at the developer’s mercy....
Running MPI applications

Message-Passing Interface (MPI)

- Used to create programs running as multiple interacting processes
- May run across multiple machines (Distributed Memory) – may use huge number of cores (in principle)
- Interaction between processes by sending/receiving messages
  - mechanism dependent on where processes are running (one or multiple machines), but generally costly...
- Each MPI process may be multithreaded (i.e., use pthreads and/or OpenMP)
- Various implementations (OpenMPI and mpich2 most popular – both available on BioHPC cloud)
Running MPI programs

Programs using MPI are started using a **launcher** program `mpirun` (some variations on that name are possible, depending on MPI implementation)

Run using 10 processes on the **local machine** (the one the command is run on)

```
mpirun -np 10 myprogram >& somefile.log &
```

To run on **multiple machines**, construct a file with a list of machines, `mymachines`, possibly specifying some limits on number of **processes** to be allowed

```
cbsum1c1b001 slots=4 max_slots=4
cbsum1c2b003 max_slots=4
cbsum1c2b002 slots=4
```

NOTE: each MPI process may be multi-threaded!

Then, for example, the command

```
mpirun -hostfile mymachines -np 14 myprogram >& somefile.log &
```

will launch 4 processes on `cbsum1c1b001`, 4 more on `cbsum1c2b003`, and 6 on `cbsum1c2b002` (oversubscription possible on this node)

Plenty of other options for distributed processes on nodes.
Killing parallel tasks may be tricky

- If the application is running in the **background** (i.e., with “&”), it can be stopped with the `kill` command
  
  \[
  \text{kill -9 <PID>}
  \]
  
  Where `<PID>` is the process id obtained from the `ps` command. For example,

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

- To kill a parallel application consisting of **multiple processes**, use the PID of the top parent process, preceded by a dash
  
  \[
  \text{kill -9 -18817}
  \]
  
  (technically, this kills all processes in the **process group** 18817)

- If some processes, still left over, you may have to track them down (with `ps`) and kill individually
Monitoring a running task using `top`

```bash
blastx -num_threads 2 -db ./databases/swissprot -query seq_tst.fa
```

- >100% CPU indicates the program is **multithreaded**
  - Multiple threads within a single process rather than multiple processes
Monitoring a running task using `htop`
Monitoring a single task using /usr/bin/time tool

Command being timed: "blastx -db ./databases/swissprot -num_alignments 1 -num_threads 3 -query seq_tst.fa -out seq_tst.fa.hits.txt" > & run.log

User time (seconds): 35.86
System time (seconds): 0.15
Percent of CPU this job got: 292%

Elapsed (wall clock) time (h:mm:ss or m:ss): 0:12.31

Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0

Maximum resident set size (kbytes): 208488

Content of run.log

Max memory the process used in its lifetime

Shows ‘user’ time combined over all threads
Assess I/O activity using iostat

No significant I/O
Monitoring I/O using \texttt{io\_top} tool

```
/programs/bin/labutils/io\_top -o -u bukowski
```

No I/O-intensive processes running

<table>
<thead>
<tr>
<th>TID</th>
<th>PRI</th>
<th>USER</th>
<th>DISK READ</th>
<th>DISK WRITE</th>
<th>SWAPIN</th>
<th>IO &gt;</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Three gzip processes running

<table>
<thead>
<tr>
<th>TID</th>
<th>PRI</th>
<th>USER</th>
<th>DISK READ</th>
<th>DISK WRITE</th>
<th>SWAPIN</th>
<th>IO &gt;</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>81177</td>
<td>be/4</td>
<td>bukowski</td>
<td>0.00 B/s</td>
<td>1395.44 K/s</td>
<td>0.00 %</td>
<td>0.00 %</td>
<td>gzip BBB 1</td>
</tr>
<tr>
<td>81178</td>
<td>be/4</td>
<td>bukowski</td>
<td>0.00 B/s</td>
<td>1440.46 K/s</td>
<td>0.00 %</td>
<td>0.00 %</td>
<td>gzip BBB 2</td>
</tr>
<tr>
<td>81179</td>
<td>be/4</td>
<td>bukowski</td>
<td>0.00 B/s</td>
<td>1395.44 K/s</td>
<td>0.00 %</td>
<td>0.00 %</td>
<td>gzip BBB 3</td>
</tr>
</tbody>
</table>

Read/write rate % time spent waiting for I/O
Monitoring I/O using htop tool
Monitoring I/O

- Notoriously hard, because
  - most I/O operations are buffered and cached, i.e., go through memory if enough available
  - I/O behavior of a single task not always representative of that of concurrent tasks
  - performance dependent on disk hardware
    - slow on cbsumlc* machines
    - very fast on the newest machines with NVMEs (SSDs with fast connect)
  - performance dependent on data structure (a lot of small files vs few large files)

- Indications of heavy I/O problem:
  - small %CPU compared to number of threads in top or htop report
  - large %IO in iotop output (% of time the process spends waiting for I/O operation)
  - continuously high Read-Write rates in iotop or htop report

**Ultimate test:** monitor performance as a function of number of concurrent tasks
Balancing the load: multiple independent tasks

- Suppose we monitored/profiled our application and we already know
  - memory needed per instance
  - optimal number of threads per instance
  - at least a vague idea about I/O needs per instance
  - \( N \) - number of instances to be run concurrently

What if the total number of tasks we have is >> \( N \) ?

Example: compress 9 files, running at most 3 instances of gzip at a time
# Balancing the load: pedestrian way

Example: 9 tasks, 3 at a time

```bash
#!/bin/bash

gzip [options] file1 &
gzip [options] file2 &
gzip [options] file3 &

wait

gzip [options] file4 &
gzip [options] file5 &
gzip [options] file6 &

wait

gzip [options] file7 &
gzip [options] file8 &
gzip [options] file9 &
```

Not too efficient, if compressing different file* takes different amounts of time

`wait` needs to wait for the slowest of the three instances

(Note: `wait` – makes the script wait for everything before it to finish before proceeding)
Using a text editor, create a file called (for example) **TaskFile**  
(This is **NOT** a script, just a list of commands to run)

```plaintext
# A longer file could be created, for example, using a shell script similar to:

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

```plaintext
| gzip file1 |
gzip file2  
gzip file3  
gzip file4  
gzip file5  
gzip file6  
gzip file7  
gzip file8  
gzip file9  
```
Load balancing using GNU parallel tool

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

```
parallel -j NP < TaskFile >& log &
```

where \textbf{NP} is the number of instances to use (e.g., 3)

\begin{itemize}
  \item \texttt{parallel} will execute tasks listed in \texttt{TaskFile} using up to \textbf{NP} instances at a time
  \begin{itemize}
    \item The first \textbf{NP} tasks will be launched simultaneously
    \item The \((\textbf{NP}+1)\)th task will be launched right after one of the initial ones completes and a core becomes available
    \item The \((\textbf{NP}+2)\)nd task will be launched right after another core becomes available
    \item ...... etc., until all tasks are distributed
  \end{itemize}
  \item Only up to \textbf{NP} tasks are running at a time (less at the end)
  \item All \textbf{NP} cores always kept (on average) busy (except near the end of task list) – \textbf{Load Balancing}
\end{itemize}
GNU parallel: general idea and syntax

Suppose `someprog` is a program taking one argument, and we want to run it \( N \) times with \( N \) values of that argument:

```
someprog a1
someprog a2
someprog a3
...
someprog aN
```

GNU parallel can help:

```
parallel [options] someprog ::: a1 a2 a3 ... aN
```

will start these commands running **concurrently**

[options] are there to control things (examples later)

(so, in essence, parallel just concatenates `someprog` with each of \( a_i \) and treats those as commands to run)
GNU parallel: general idea and syntax

Instead of listing arguments, we can put them in a file, say `argfile`, listing one argument per line like this:

```
a1
a2
a3
...
aN
```

Then run `parallel` like this (note the four colons `:::`):

```
parallel [options] someprog :::: argfile
```

Equivalent forms:

```
parallel [options] -a argfile someprog

cat argfile | parallel [options] someprog

parallel [options] someprog < argfile
```
GNU parallel: general idea and syntax

Remember the ‘original’ command we introduced `parallel` with?

```
parallel -j 10 < TaskFile
```

where `TaskFile` was

```
gzip file1
gzip file2
...
gzip file3000
```

This is like running

```
parallel -j 10 someprog :::: TaskFile
```

with empty `someprog` and ‘arguments’ in the form `gzip file1`
What is the `someprog` command needs more than 1 argument?

```
parallel -N2 someprog ::: a1 a2 a3 a4 a5 a6
```

will produce the following commands:

```
someprog a1 a2
someprog a3 a4
someprog a5 a6
```
GNU parallel: general idea and syntax

What if we need to run a not one, but a few commands?

```
parallel someprog1 {}\; someprog2 {} :::: a1 a2 a3
```

( {} represents the argument, if only one )

will result in

```
someprog1 a1; someprog2 a1     # run one after the other, but concurrently with other such pairs
someprog1 a2; someprog2 a2
someprog1 a3; someprog2 a3
```

Another example: `someprog1` and `someprog2` run on different arguments

```
parallel -N2 someprog1 {1}\; someprog2 {2} :::: a1 a2 a3 a4
```

( {1},{2} represent individual arguments, if multiple )

will result in

```
someprog1 a1; someprog2 a2
someprog1 a3; someprog2 a4
```
GNU parallel: more control through options

```
parallel -j 4 --delay 5 --load 200% --memfree 2G someprog :::: argfile
```

- `-j 4` run up to 4 commands concurrently
- `--delay 5` start each command 5 seconds after previous one
- `--load 200%` start command only if load on the machine is not more than 2 threads
- `--memfree 2G` start command only if there is at least 2G of RAM available

Caution:

If `someprog` is multi-threaded, it will ‘occupy’ not 4, but (4 x number_threads_per_task) CPU cores !!!
GNU parallel: remote execution (and more options)

```bash
parallel -j 2 \
-S machine1 -S machine2 \
--transferfile BBB_{} \
--return BBB_{}.gz \
--workdir /workdir/bukowski \
--cleanup \
--joblog run.log \
gzip ::: 1 2 3
```

What will happen here:

- Commands `gzip BBB_1`, `gzip BBB_2`, and `gzip BBB_3` will be run, at most 2 at a time, using machines `machine1`, `machine2`, accessed via `ssh`.
- Files `BBB_1`, `BBB_2`, and `BBB_3` will be transferred from the current directory to the relevant machine to directory `/workdir/bukowski`, and the ‘gzipping’ will take place there.
- Upon completion, compressed files `BBB_1.gz`, `BBB_2.gz`, and `BBB_3.gz` will be transferred back to the current directory.
- Files on the remote machines will be cleaned up.
- Log of the entire operation, with some useful timing information, will be saved in file `run.log` (in the current directory on the current machine, from which `parallel` was submitted).

NOTE: user should have passwordless `ssh` access set up between the machines to avoid being asked for password...
GNU parallel: killing tasks

Find the process ID (PID) of the parallel process

```
ps -ef | grep parallel
bukowski 28310 1710 1 13:50 pts/13 00:00:00 perl /programs/parallel/bin/parallel -j 2 gzip BBB_{} ::: 1 2 3
bukowski 28558 1710 0 13:50 pts/13 00:00:00 grep --color=auto parallel
```

Now send the SIGTERM signal to the process c- this will ‘drain the queue’ (allow tasks already running to finish)

```
kill -15 28310
parallel: SIGTERM received. No new jobs will be started.
parallel: Waiting for these 2 jobs to finish. Send SIGTERM again to stop now.
```

Send the SIGTERM signal again to kill off the remaining running processes

```
kill -15 28310
```
**xargs – ‘older brother’ of GNU parallel**

Functionality of **xargs** similar (but more limited) than that of **parallel**

some options of **parallel** designed to mimic those of **xargs**

Example:

Let **TaskFile** contain a list of files

```
file1
defile2
file3
```

```
cat TaskFile | xargs gzip
```

will construct (and run) the following, using a single process (i.e., 3 **gzip** operations one after another)

```
gzip file1 file2 file3
```

```
cat TaskFile | xargs -n 1 -P 2 gzip
```

will construct (and run) the following, using **up to 2** processes at a time

```
gzip file1
gzip file2
gzip file3
```
Exercise 3: timing *bwa mem* alignment

**Example:**
- `example_1.fastq.gz`
- `example_2.fastq.gz`
- Reference genome

**Process:**
1. Run `bwa mem` with `-M` and `-t 8` flags to align the input files to the reference genome.
2. Pipe the output to `samtools` with `-Sb` and `-o` flags to create the `example.bam` file.

**Objective:**
- Run this on varying numbers of threads.
- Measure time, memory, I/O as functions of that number.
GNU parallel limitations

- Parallel is a clever tool for submitting multiple commands in the background, possibly on multiple machines.

- Very useful extra options, such as (and there are many more):
  
  - \(-j \ 4\)  
  - \(--\text{delay} \ 5\)  
  - \(--\text{load} \ 200\%\)  
  - \(--\text{memfree} \ 2G\)  
  - \(--\text{timeout} \ 60\)  
  
  - Limit the number of commands run concurrently (here: 4).
  - Start each command some time (here: 5 seconds) after previous one.
  - Start command only if load on the machine is not more than some number (here: 2) of threads.
  - Start command only if there is at least some mount (here: 2G) of RAM is available.
  - Impose timeout (here: 60 s) on each command.

- But there are limitations to what parallel can do for you:
  
  - Once a command is running, no control over how many cores or how much memory it uses (may overwhelm machine).
  - Can’t control individual commands.
  - No way to enforce fair sharing of resources among multiple users and/or user groups.

**Need a SCHEDULER to deal with these!**
<table>
<thead>
<tr>
<th>Functionality</th>
<th>Parallel</th>
<th>Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start multiple jobs on limited resources</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Terminate individual jobs</td>
<td>no (or hard)</td>
<td>yes</td>
</tr>
<tr>
<td>Control #cores and memory of running jobs</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Prioritize jobs of different users, groups</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Control job timeout</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Streamline submission based on job requirements</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>File pre-staging</td>
<td>yes</td>
<td>yes (sort of)</td>
</tr>
<tr>
<td>Job staggering</td>
<td>yes</td>
<td>yes ?</td>
</tr>
<tr>
<td>Job accounting</td>
<td>no (or limited)</td>
<td>yes</td>
</tr>
</tbody>
</table>
Some popular schedulers

- Obsolete or commercialized
  - **PBS**: Portable Batch System
  - **SGE**: Sun Grid Engine
  - **LFS**: Load Sharing Facility
  - **Lava**: light version of LSF
  - **TORQUE** (version) of PBS
  - **UNIVA**: commercial fork of SGE

- Free, modern, and actively developed
  - **SLURM**: Simple Linux Utility for Resource Management
Structure of SLURM cluster(s)

Execution nodes
- machine1
- machine2
- machine3

Control node
- slurmd
- slurmd
- slurmd
- slurmd

Services (daemons)
- **slumrd**: starts and manages jobs on a compute node
- **slurmctld**: decides where and when to dispatch jobs, tracks jobs
- **slurmdbd**: logs jobs information into database, maintains user and account information

Accounting node
- mysqld
SLURM setup is an admin task

- Non-trivial setup and maintenance
  - Require extra pieces of software installed running on all machines involved
  - To be started, configured, and maintained by an administrator (users generally cannot do it)
  - Takes significant know-how and work to set up and configure
  - Configuration typically tailored specifically to a particular cluster/lab/group/institution

- Users need to follow usage guidelines for the specific scheduler configuration
  - Learning curve involved – different for each cluster
Configuration of resources in SLURM

- Nodes (machines) grouped into partitions (queues)
  - typically collect similar nodes, or nodes with similar function
  - each node may belong to multiple partitions
  - partition may have per job limits and defaults (run time, memory, max #cores, etc)
  - User needs to specify which partition their job is to be submitted to
  - One partition is ‘default’

- Cluster may be configured to grant jobs either whole nodes, or node ‘slices’ (i.e., some #cores + some memory)
  - jobs are restricted to #cores and RAM requested at submission – will not use more (may crash on attempt!)
  - #cores and RAM allocated to a running job are subtracted from the node’s totals – only available resources are offered to new jobs

- Users organized in (trees of) accounts (e.g., lab groups), with defined shares, determining usage priorities

- Per user and/or per group limits or privileges may be defined (QOS – quality of service)
SLURM at BioHPC

- **Permanent clusters**, made up of Lab- or Department-owned machines, customized to serve those Labs or Departments. Access for lab members only.
  - BSCB cluster (csubscb): 15 nodes, 1136 CPU cores, 5.8 TB RAM
  - cbsuxu
  - cbsuorm
  - cbsugaurav
  - others welcome – contact us to discuss/set up

- 'SLURM on demand' clusters:
  - possible to spin up by any user on their reserved or Lab-owned machine(s)
  - access for all users with reservations on these machines
  - temporary – will disappear upon the end of reservation
  - not configurable (at present, only a single configuration is offered)

- What are they good for:
  - load balancing of single or multiple users’ jobs (like `parallel`, but with more control)
  - re-using SLURM scripts brought from elsewhere (some customization typically be required)
  - running pipelines which require SLURM for load balancing (and some do)
‘SLURM on demand’ at BioHPC

- Reserve one or more machines
- Log in to one of the reserved machines
- Use `manage_slurm` tool to spin up and control (some aspects of) the cluster

```
[bukowski@cbsum1c2b005 ~]$ manage_slurm
Usage: manage_slurm <action> [args]
manage_slurm new machine1,machine2,...
    • to create a SLURM cluster on the named machines (need an active reservation on all machines). The first node will be the "master node". All users with active reservations on the full set of machines will be given access to the cluster, and will automatically be added or removed as their reservation status changes.
manage_slurm kill masterNode
    • to end the slurm cluster identified by the master node
manage_slurm addNode masterNode machine
    • adds the machine to the cluster identified by the masterNode. Need an active reservation on this machine for all current cluster users.
manage_slurm add Node-force masterNode machine
    • Like addNode, but will remove cluster access for any users necessary to add the node (including deleting any of their submitted jobs). Try addNode first to get a list of users that will be removed
manage_slurm removeNode masterNode machine
    • Remove a node from the cluster identified by masterNode. The machine should not be the masterNode; removing the masterNode will kill entire cluster, so use "kill" command instead.
manage_slurm list
    • List and describe all clusters that you have access to. Reports the list of machines, number of CPUs/memory, and list of authorized users
```

More info: [https://biohpc.cornell.edu/lab/userguide.aspx?a=software&i=689#c](https://biohpc.cornell.edu/lab/userguide.aspx?a=software&i=689#c)
At present, only one configuration offered:

- One partition (‘regular’)
  - contains all nodes
  - no per job time limit
  - no per job CPU core limits
  - 4 GB RAM per job default

- One ‘account’, containing all users having reservations on all machines of the ‘cluster’
  - Fairshare scheduling policy with all users ‘equal’ (more details later)
  - No per user limits
## SLURM: know your cluster – partitions, nodes summary

Info on ‘current cluster’, which the node belongs to

```
[bukowski@cbsum1c2b003 ~]$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE  NODELIST
regular*     up  infinite      4  idle  cbsum1c2b[003-004,006-007]
```

Info on other clusters

```
[bukowski@cbsum1c2b003 slurm]$ sinfo --cluster=cbsubscb
CLUSTER:  cbsubscb
PARTITION AVAIL  TIMELIMIT  NODES  STATE  NODELIST
short*           up   4:00:00     16  mix  cbsubscb[01-15],cbsubscbgpu01
regular          up 1-00:00:00     16  mix  cbsubscb[01-15],cbsubscbgpu01
long7            up 7-00:00:00     15  mix  cbsubscb[01-15]
long30           up 30-00:00:00   15  mix  cbsubscb[01-15]
gpu              up 3-00:00:00     1  mix  cbsubscbgpu01
```
NodeName=cbsum1c2b006 Arch=x86_64 CoresPerSocket=1
   CPUAlloc=0 CPUTot=8 CPULoad=0.01
AvailableFeatures=(null)
ActiveFeatures=(null)
Gres=(null)
NodeAddr=128.84.181.157 NodeHostName=cbsum1c2b006
OS=Linux 3.10.0-957.10.1.el7.x86_64 #1 SMP Mon Mar 18 15:06:45 UTC 2019
RealMemory=15883 AllocMem=0 FreeMem=3254 Sockets=8 Boards=1
State=IDLE ThreadsPerCore=1 TmpDisk=0 Weight=1 Owner=N/A MCS_label=N/A
CfgTRES=cpu=8,mem=15883M,billing=8 AllocTRES=
CapWatts=n/a CurrentWatts=0 AveWatts=0
ExtSensorsJoules=n/s ExtSensorsWatts=0 ExtSensorsTemp=n/s
### SLURM: know your cluster – partition details

```
[bukowski@cbsum1c2b003 slurm]$ scontrol show partitions

PartitionName=regular
  AllowGroups=ALL  AllowAccounts=ALL  AllowQos=ALL
  AllocNodes=ALL  Default=YES  QoS=N/A
  DefaultTime=NONE  DisableRootJobs=NO  ExclusiveUser=NO  GraceTime=0  Hidden=NO
  MaxNodes=UNLIMITED  MaxTime=UNLIMITED  MinNodes=0  LLN=NO
  MaxCPUsPerNode=UNLIMITED
    Nodes=cbsum1c2b003,cbsum1c2b004,cbsum1c2b006,cbsum1c2b007
  PriorityJobFactor=1  PriorityTier=1  RootOnly=NO  ReqResv=NO  OverSubscribe=NO
  OverTimeLimit=NONE  PreemptMode=OFF
  State=UP  TotalCPUs=32  TotalNodes=4  SelectTypeParameters=None
  JobDefaults=(null)
  DefMemPerNode=4096  MaxMemPerNode=UNLIMITED
```
SLURM and you – typical scenario

- Determine job’s CPU-cores and RAM requirements

- Write a shell script that will
  - create a job directory on local scratch file system
  - prepare (copy) input files to job’s scratch
  - launch the application (output to be written to job’s scratch)
  - copy output files back to permanent storage (e.g., home directory)

- Submit script using `sbatch` command with desired options (#cores, RAM, partition, nodes, ...)
  - may embed SLURM options in the script header
  - interactive session may be requested using `srun` command
  - submit as many jobs as you need

- Jobs are queued up and wait for resources and their turn to start on some node (competing with other jobs)

- Check on your jobs using `squeue`, state of cluster using `sinfo, scontrol`

- Control/cancel your jobs (`scontrol update, scancel`)

- Get information about finished jobs using `sacct`

SLURM: typical shell script

Typical shell script, call it **my_script.sh**

```bash
#!/bin/bash

# Create a scratch directory for this job
WDIR=/workdir/bukowski/${SLURM_JOB_ID}
mkdir -p $WDIR
cd $WDIR

#Copy input to the scratch directory
cp /home/bukowski/inputs/input.file .

# Run the computation
/home/bukowski/programs/my_program < input.file >& output.file.${SLURM_JOB_ID}

# Copy results back to permanent storage
cp output.file.${SLURM_JOB_ID} /home/bukowski/outputs

# Clean up after job
cd /workdir/bukowski
rm -Rf ${SLURM_JOB_ID}
```

Make job inherit your login environment

Integer unique for every SLURM job
SLURM: submitting a script

```
sbatch [options] my_script.sh [arguments]
```

Here are some more important [options] with examples:

- `--nodes=1` (number of nodes, must be 1 for all non-MPI jobs)
- `--ntasks=8` (number of tasks; task=1 slot=1 thread; default: 1)
- `--mem=8000` (request 8 GB of memory for this job; default: 4 GB)
- `--time=1-20:00:00` (wall-time limit for job; here: 1 day and 20 hours)
- `--partition=regular,long30` (request partition(s) a job can run in; here: regular and long30)
- `--account=bscb09` (project to charge the job to)
- `--chdir=/home/bukowski/slurm` (start job in specified directory; default is the directory in which sbatch was invoked)
- `--job-name=jobname` (name of job)
- `--output=jobname.out.%j` (write stdout+stderr to this file; %j will be replaced by job ID)
- `--mail-user=email@address.com` (set your email address)
- `--mail-type=ALL` (send email at job start, end or crash - do not use if this is going to generate thousands of e-mails!)

So, the submission command could look like

```
sbatch --nodes=1 --ntasks=6 --mem=4000 my_script.sh
```

Many options have shorthand notation, e.g.,

```
sbatch -N 1 -n 6 --mem=4000 my_script.sh
```
SLURM: specifying option in script header

Typical script header may specify submission options after #SBATCH keyword

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=8
#SBATCH --mem=8000
#SBATCH --time=1-20:00:00
#SBATCH --partition=regular,long30
#SBATCH --account=bscb09
#SBATCH --chdir=/home/bukowski/slurm
#SBATCH --job-name=jobname
#SBATCH --output=jobname.out.%j
#SBATCH --mail-user=email@address.com
#SBATCH --mail-type=ALL

# Rest of the script goes here...
```

Options given directly on `sbatch` command line supersede the ones in header
SLURM: other interesting options/comments

--ntasks
used to request the number of threads (cores) for a job (also, set `OMP_NUM_THREADS` accordingly)

-N 1
non-MPI jobs **must** run on a single machine (otherwise some cores may be allocated on other nodes and won’t be useful)

--nodelist=cbsubscb11
run on a specific node (otherwise it will start on some node within requested partitions – unknown in advance!)

--exclude=cbsubscb10,cbsubscbgpu01
exclude specified nodes

--chdir and --output
directories specified in these options must be present on all nodes where a job can start (e.g., `$HOME`)

Alternative core and memory specifications

```bash
sbatch -N 1 -n 8 --cpus-per-task=3 --mem-per-cpu=2G my_script.sh

sbatch -n 24 --mem=48G my_script.sh
```
SLURM: where to submit jobs from?

This is configurable, but at BioHPC

- Any node of the cluster (if you have ssh login access to it)

  `sbatch [options] my_script.sh [arguments]`

- Any machine configured to use the same slurmdbd (database) service as the cluster in question, e.g., the login node `cbsulogin.biohpc.cornell.edu`. Use the `--clusters` option to indicate the cluster to submit to:

  `sbatch --clusters=cbsumlc2b003 [options] my_script.sh [arguments]`

- Directory a job is submitted from becomes the job’s ‘startup directory’ and so it must exist on all nodes the job may start on. `$HOME` is a good choice
  - job’s ‘startup directory’ may be changed using option `--chdir`
  - jobs should use local scratch storage (rather than ‘startup directory’) for I/O-intensive computations
(some of) Environment Variables available within a SLURM job

**SLURM_JOB_CPUS_PER_NODE**: number of CPU cores (threads) allocated to this job

**SLURM_NTASKS**: number of tasks, or slots, for this job (as given by `--ntasks` option)

**SLURM_MEM_PER_NODE**: memory requested with `--mem` option

**SLURM_CPUS_ON_NODE**: total number of CPUs on the node (not only the allocated ones)

**SLURM_JOB_ID**: job ID of this job; may be used, for example, to name a scratch directory (subdirectory of `/workdir`, or output files) for the job. For array jobs, each array element will have a separate `SLURM_JOB_ID`

**SLURM_ARRAY_JOB_ID**: job ID of the array master job

**SLURM_ARRAY_TASK_ID**: task index of a task within a job array

**SLURM_ARRAY_TASK_MIN, SLURM_ARRAY_TASK_MAX**: minimum and maximum index of jobs within the array

Complete list – in section 'OUTPUT ENVIRONMENT VARIABLES' of [https://slurm.schedmd.com/sbatch.html](https://slurm.schedmd.com/sbatch.html).
SLURM: Job arrays (array jobs?)

Consider script `my_array_script.sh` (objective: compress files BBB_1, BBB_2, BBB_3)

```bash
#!/bin/bash

# Prepare the scratch directory for the job and 'cd' to it
WDIR=/workdir/SLURM_JOB_ID
mkdir -p $WDIR
cd $WDIR

# Copy the file to gzip from network-mounted directory
cp /shared_data/Parallel_workshop/BBB_$SLURM_ARRAY_TASK_ID

# Run the compression
gzip BBB_$SLURM_ARRAY_TASK_ID

# Copy the result back into the result directory (here: same as submission dir)
cp BBB_$SLURM_ARRAY_TASK_ID.gz $SLURM_SUBMIT_DIR

# Clean up
cd $SLURM_SUBMIT_DIR; rm -Rf $WDIR
```

Now submit this script as `job array`:

```
sbatch --array=1-3 [other_options] my_array_script.sh
```

This single command will submit three separate jobs, each running `my_array_script.sh`, but in each the value of `SLURM_ARRAY_TASK_ID` will be different (one of 1, 2, or 3). This variable is provided as a result of the `--array` option.
SLURM: interactive jobs

```
srun -n 2 -N 1 --mem 2G --pty --preserve-env --cpu-bind=no /bin/bash
```

This will create a job (can check with `squeue`) and open an interactive `bash` shell on a machine picked by SLURM.

This shell will be constrained to the number of cores and memory requested.

After interactive work finished, exit the shell (Ctrl-D or ‘exit’).

On some clusters (cbsubscb), `salloc` may be configured to automatically execute the `srun` command as above.

`srun`: very much like `sbatch`, except it works in real time rather than batch mode.
SLURM: checking on jobs using `squeue`

```
[slurm]$ squeue

<table>
<thead>
<tr>
<th>JOBID</th>
<th>PARTITION</th>
<th>NAME</th>
<th>USER</th>
<th>ST</th>
<th>TIME</th>
<th>NODES</th>
<th>NODELIST(REASON)</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>regular</td>
<td>blast_te</td>
<td>bukowski</td>
<td>PD</td>
<td>0:00</td>
<td>1</td>
<td>(Resources)</td>
</tr>
<tr>
<td>18</td>
<td>regular</td>
<td>bwa_test</td>
<td>bukowski</td>
<td>R</td>
<td>0:48</td>
<td>1</td>
<td>cbsum1c2b003</td>
</tr>
<tr>
<td>20_1</td>
<td>regular</td>
<td>array_te</td>
<td>bukowski</td>
<td>R</td>
<td>0:10</td>
<td>1</td>
<td>cbsum1c2b004</td>
</tr>
<tr>
<td>20_2</td>
<td>regular</td>
<td>array_te</td>
<td>bukowski</td>
<td>R</td>
<td>0:10</td>
<td>1</td>
<td>cbsum1c2b004</td>
</tr>
<tr>
<td>20_3</td>
<td>regular</td>
<td>array_te</td>
<td>bukowski</td>
<td>R</td>
<td>0:10</td>
<td>1</td>
<td>cbsum1c2b004</td>
</tr>
</tbody>
</table>
```
SLURM: checking on jobs using `squeue`

Default output from `squeue` is not that informative. It is better to use a more detailed format

```
squeue -o '%.18i %.10n %s %.30P %.8j %.8u %.8T %.10M %.91 %.6D %C %R %p %S'
```

or simply use the wrapper script (which we wrote)

```
squeue_l
```
**SLURM: retrieving job accounting information using `sacct`**

```
[bukowski@cbsum1c2b003 slurm] $ sacct
```

<table>
<thead>
<tr>
<th>JobID</th>
<th>JobName</th>
<th>Partition</th>
<th>Account</th>
<th>AllocCPUS</th>
<th>State</th>
<th>ExitCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>bwa_test</td>
<td>regular</td>
<td>acc_cbsum+</td>
<td>4</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>3.batch</td>
<td>batch</td>
<td></td>
<td>acc_cbsum+</td>
<td>4</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>4</td>
<td>bwa_test</td>
<td>regular</td>
<td>acc_cbsum+</td>
<td>1</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>4.batch</td>
<td>batch</td>
<td></td>
<td>acc_cbsum+</td>
<td>1</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>5</td>
<td>bwa_test</td>
<td>regular</td>
<td>acc_cbsum+</td>
<td>1</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>5.batch</td>
<td>batch</td>
<td></td>
<td>acc_cbsum+</td>
<td>1</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
</tbody>
</table>

For each job, all job steps are listed (job itself + the batch) – hence two lines per job.
SLURM: more job accounting information using `sacct_l`

`sacct_l`: a simple wrapper around `sacct`, giving more useful information

Produces rather long lines. In the example below, the lines are broken in half

Left-most columns of `sacct_l` output for two jobs

<table>
<thead>
<tr>
<th>JobID</th>
<th>JobIDRaw</th>
<th>JobName</th>
<th>User</th>
<th>NodeList</th>
<th>ReqMem</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>15</td>
<td>bwa_test</td>
<td>bukowski</td>
<td>cbsum1c2b004</td>
<td>4Gn</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>batch</td>
<td></td>
<td>cbsum1c2b004</td>
<td>4Gn</td>
</tr>
<tr>
<td>16_1</td>
<td>17</td>
<td>array_test</td>
<td>bukowski</td>
<td>cbsum1c2b005</td>
<td>4Gn</td>
</tr>
<tr>
<td>16_1</td>
<td>17</td>
<td>batch</td>
<td></td>
<td>cbsum1c2b005</td>
<td>4Gn</td>
</tr>
</tbody>
</table>

Right-most columns of `sacct_l` output for same two jobs

<table>
<thead>
<tr>
<th>MaxRSS</th>
<th>MaxVMSIZE</th>
<th>NCPUS</th>
<th>Start</th>
<th>CPUPtime</th>
<th>TotalCPU</th>
<th>UserCPU</th>
<th>Elapsed</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980K</td>
<td>132900K</td>
<td>1</td>
<td>2020-05-10T11:30:12</td>
<td>00:01:08</td>
<td>01:03.112</td>
<td>01:02.051</td>
<td>00:01:08</td>
<td>COMPLETED</td>
</tr>
</tbody>
</table>
SLURM: use accounting info to evaluate performance

Signature of an efficient job:

\[ \text{CPUTime} \ (\text{= NCPU*Elapsed}) \approx \text{TotalCPU} \approx \text{UserCPU} \]
\[ \text{MaxRSS} < \text{ReqMem} \]

Signs of inefficiency:

\[ \text{TotalCPU} < \text{CPUTime} \]  inefficient parallelization (e.g., due to sequential part of job, like file copying)
\[ \text{UserCPU} < \text{TotalCPU} \]  lot of time spent is system calls (typically due to inefficient I/O)
\[ \text{MaxRSS} \approx \text{ReqMem} \]  job may have not enough memory, may crash or swap; increase \text{--mem}

\textbf{MaxRSS}: use this to estimate memory needs of your job

- Run test requesting a lot of memory, then for production runs set \text{--mem} to a value \text{slightly} larger (1.2 times?) than \text{MaxRSS} obtained from the test run
What happens to queued jobs?

- Scheduler (part of `slurmctld` service daemon) runs periodically (once in about 1 minute)
  - keep track of running jobs and their allocated resources
  - keep track of available resources
  - compute job priorities
  - examine each waiting job, check if requested resources available
  - if multiple jobs compete, submit the ones with highest priorities
  - backfilling: ‘small’ jobs with lower priority may get ahead of ‘big’ jobs with higher priority if it does not affect the start time of the latter
    - accurate timing requests are necessary effective backfilling
Multi-factor job priority

Job_priority =
  site_factor +
  (PriorityWeightAge) * (age_factor) +
  (PriorityWeightAssoc) * (assoc_factor) +
  (PriorityWeightFairshare) * (fair-share_factor) +
  (PriorityWeightJobSize) * (job_size_factor) +
  (PriorityWeightPartition) * (partition_factor) +
  (PriorityWeightQOS) * (QOS_factor) +
  SUM(TRES_weight_cpu * TRES_factor_cpu,
      TRES_weight_<type> * TRES_factor_<type>,
      ...)
  - nice_factor

**Weights**: large integer numbers

**Factors**: numbers between 0 and 1

**Association** = (user, account, cluster, partition)

**QOS**: Quality of Service (set of limits or privileges)

**TRES**: trackable resource

**JobSize**: related to #cores requested

**Fairshare**: reflects proportion of resources consumed by user to user’s ‘share’ in the cluster

Probably the most important factor in multi-user, multi-group clusters
Fair Tree Fairshare example

AccountA (10 shares)
- User1 (40 shares), usage=20
- User2 (30 shares), usage=10

AccountB (20 shares)
- User3 (50 shares), usage=30
- User4 (60 shares), usage=40
- User5 (10 shares), usage=0

Relative share S:
- AccountA: $10/(10+20) = 1/3$
- AccountB: $20/(10+20) = 2/3$
- User1: $40/(30+40) = 4/7$
- User2: $30/(30+40) = 3/7$
- User3: $50/(50+60+10) = 5/12$
- User4: $60/(50+60+10) = 6/12$
- User5: $10/(50+60+10) = 1/12$

Relative usage U:
- AccountA: $30/(30+70) = 3/10$
- AccountB: $70/(30+70) = 7/10$
- User1: $20/(10+20) = 2/3$
- User2: $10/(10+20) = 1/3$
- User3: $30/(30+40) = 3/7$
- User4: $40/(30+40) = 4/7$
- User5: $0/(30+40) = 0$

Level Fairshare LF = S/U:
- AccountA: $10/9$
- AccountB: $20/21$
- User1: $6/7$
- User2: $8/7$
- User3: $35/36$
- User4: $42/48$
- User5: Infinity

First sort accounts according to their LF, then sort users within accounts according to their LF:

AccountA (LF=10/9)
- User1 (LF=6/7) Priority: 4/5
- User2 (LF=8/7) Priority: 5/5

AccountB (LF=20/21)
- User5 (LF=Infinity) Priority: 3/5
- User3 (LF=35/36) Priority: 2/5
- User4 (LF=42/48) Priority: 1/5
What is ‘Usage’ anyway?

Usage for a running job during time period $Dt$

$$U_{\text{job}} = [#\text{cores} \times \text{core\_weigth} + \text{RAM} \times \text{RAM\_weigth}] \times Dt$$

(defaults: $\text{core\_weigth}=1, \text{RAM\_weigth}=0, Dt=5\ \text{min}$)

Total usage $U_{\text{user}}$ for a user: sum of $U_{\text{job}}$ over all user’s jobs

Taking into account past usage:

$$U_{\text{user}} = U_{\text{now}} + d \times U_{\text{now\_1}} + d \times U_{\text{now\_2}} + d \times d \times U_{\text{now\_3}} + \ldots$$

Where

$U_{\text{now\_N}}$ is the user’s usage $U_{\text{user}}$ from time period $N \times Dt$ before the present one

$d$ is set based on the assumed usage half-life time, $T_{\text{half}}$ (e.g., 1 week), i.e.,

$$d = (1/2)^{(Dt/T_{\text{half}})} < 1$$
SLURM documentation

- Version installed on BioHPC: [https://slurm.schedmd.com/archive/slurm-19.05.2/](https://slurm.schedmd.com/archive/slurm-19.05.2/)
  - upgrade needed soon...

- Man pages for individual commands: [https://slurm.schedmd.com/archive/slurm-19.05.2/man_index.html](https://slurm.schedmd.com/archive/slurm-19.05.2/man_index.html)


- Formal documentation very thorough, but rather formal, with few specific examples
  - often ‘googling’ a specific subject or command will yield more clear info from ‘SLURM practitioners’