Parallel Processing and Load Balancing

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


Contact: [brc_bioinformatics@cornell.edu](mailto:brc_bioinformatics@cornell.edu)
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></td>
<td></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>Total</td>
<td>89</td>
<td>3,056</td>
<td>19,104</td>
</tr>
</tbody>
</table>

*your workshop machines*
Big picture

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
Synopsis

- Some basics 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 (but quantitative conclusions doubtful in shared environment... )
**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.

# threads possible to **run** at the same time = # 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
Check on CPU configuration with `lscpu`

```
[root@cbsumml2 ~]# lscpu

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

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

[bukowski@cbsumm12 ~]$ free -m

<table>
<thead>
<tr>
<th></th>
<th>total</th>
<th>used</th>
<th>free</th>
<th>shared</th>
<th>buff/cache</th>
<th>available</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem:</td>
<td>128738</td>
<td>1772</td>
<td>123220</td>
<td>1550</td>
<td>3746</td>
<td>124575</td>
</tr>
<tr>
<td>Swap:</td>
<td>4095</td>
<td>836</td>
<td>3259</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

```
[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
/dev/md126             871G   72M  827G   1%  /SSD
/dev/mapper/rhel_cbsumm12-home 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:/lustre 1.3P 1003T  300T  78%  /home
cbsugfs1:/home
/glusterfs/home
tmpfs                  13G     0   13G   0%
/run/user/4857
```

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 house-keeping 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**

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

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

**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:

```
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:

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

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

Almost all CPU and memory resources up for grabs by users’ programs
Big picture

- 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 Exp of the sum using 4 threads

Thread 1
Thread 2
Thread 3
Thread 4

Synchronize/communicate
Synchronize/communicate

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
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: `pthread`, `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`, `pthread`, `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)
Find out how a program is parallelized
(easy only for executables using shared libraries)

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

linux-vdso.so.1 => (0x00007ffd843b9000)
libm.so.6 => /lib64/libm.so.6 (0x00007f5100e44000)
libmpi_cxx.so.1 => /usr/lib64/openmpi/lib/libmpi_cxx.so.1 (0x00007f5100c9000)
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 (0x00007f510020c000)
libc.so.6 => /lib64/libc.so.6 (0x00007f50ff7fe3f000)
/lib64/ld-linux-x86-64.so.2 (0x00007f5101146000)
libopen-rte.so.12 => /usr/lib64/openmpi/lib/libopen-rte.so.12 (0x00007f50ff71f000)
libopen-pal.so.13 => /usr/lib64/openmpi/lib/libopen-pal.so.13 (0x00007f50ff91f000)
libdl.so.2 => /lib64/libdl.so.2 (0x00007f50ff513000)
librt.so.1 => /lib64/librt.so.1 (0x00007f50ff513000)
libutil.so.1 => /lib64/libutil.so.1 (0x00007f50ff3f0000)
libhwloc.so.5 => /lib64/libhwloc.so.5 (0x00007f50fe0d3000)
libnuma.so.1 => /lib64/libnuma.so.1 (0x00007f50feec7000)
libltdl.so.7 => /lib64/libltdl.so.7 (0x00007f50f0d3000)
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: $S_N = \frac{T_1}{T_N} \xrightarrow{large N} 1 + \frac{t_{par}}{t_{seq}}$

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

File1 & File2 & File3 & File4

gzip

File1.gz & File2.gz & File3.gz & File4.gz

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

BUT

Threads compete for disk access if too many!

Single-threaded processes run simultaneously
Parallelizing a problem: ‘not so embarrassingly parallel’ case

Simple! No communication between processes, no sync

BUT

Processes compete for disk access if too many!
Mixed parallelization: running several simultaneous multi-threaded tasks (each processing different data) on a large machine (here: 64-core)

```bash

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 &
```
Common situation in ‘end user’ bioinformatics

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 as ‘end users’

- How to run each instance of the application, control number of threads
- CPU, memory, disk, time requirements of the application (may depend on number of threads)
- Optimal number of threads for given input data, machine

- How many instances can be run concurrently on a single machine
  - (#instances) X (#threads_per_instance) < #cores on each machine
  - (memory_per_instance) X (#instances) < total_machine_memory
  - Sufficient scratch disk storage

Load balancing

- How to manage multiple instances on available resources (machines, cores, memory, 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:

\texttt{blastall \textasciitilde a 8 [other options]}
\texttt{blastx \textasciitilde \texttt{num\_threads} 8 [other options]}
\texttt{tophat \textasciitilde p 8 [other options]}
\texttt{cuffdiff \textasciitilde p 8 [other options]}
\texttt{bwa \textasciitilde t 8 [other options]}
\texttt{bowtie \textasciitilde 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 `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 `MKL_NUM_THREADS` (BioHPC default: 1) in addition to `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

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

  Where `<PID>` is the process id obtained from the `ps` command. For example,

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

- To kill a parallel application consisting of **multiple processes**, use the PID of the top parent process, preceded by a dash

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

  (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
Multiple independent tasks

As simple as 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:

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

Multiple processes
(1 thread in each)
Multiple independent tasks: load balancing

Seemingly simple, but there are factors that limit how many tasks can be run concurrently:

- If each task multi-threaded, how many threads to run it on (watch for Amdahl’s law)?

- Avoid overloading: # threads <= # cores
  - If tasks multithreaded, # threads = # instances x # thread per instance
  - Watch for other processes (maybe someone else’s) – they also use cores!

- Combined memory of all tasks not to exceed 90% of total memory on the machine
  - Watch for other processes (maybe someone else’s) – they also use memory!

- If tasks do a lot of I/O – how many can coexist before clogging I/O channels?

Some of these questions may be addressed by monitoring resources consumed by a single task run on a representative subset of input data and extrapolating to multiple tasks

Let’s look at some monitoring tools!
Monitoring a running task using `top`

```
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

```
/usr/bin/time -v blastx -db ./databases/swissprot -num_alignments 1 -num_threads 3 -query seq_tst.fa -out seq_tst.fa.hits.txt >& run.log
```

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

- **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
- **Average resident set size (kbytes):** 0
- **Major (requiring I/O) page faults:** 0
- **Minor (reclaiming a frame) page faults:** 59067
- **Voluntary context switches:** 51
- **Involuntary context switches:** 147
- **Swaps:** 0
- **File system inputs:** 0
- **File system outputs:** 312
- **Socket messages sent:** 0
- **Socket messages received:** 0
- **Signals delivered:** 0
- **Page size (bytes):** 4096
- **Exit status:** 0

**Content of run.log**

**Shows ‘user’ time combined over all threads**

**Max memory the process used in its lifetime**
Assess I/O activity using `iostat`

No significant I/O
Monitoring I/O using `iotop` tool

```bash
/programs/bin/labutils/iotop -o -u bukowski
```

<table>
<thead>
<tr>
<th>TID</th>
<th>PRIORITY</th>
<th>USER</th>
<th>DISK READ</th>
<th>DISK WRITE</th>
<th>SWAPIN</th>
<th>IO</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.00 B/s</td>
<td>0.00 B/s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.00 B/s</td>
<td>19.70 K/s</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No I/O-intensive processes running

<table>
<thead>
<tr>
<th>TID</th>
<th>PRIORITY</th>
<th>USER</th>
<th>DISK READ</th>
<th>DISK WRITE</th>
<th>SWAPIN</th>
<th>IO</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>81177</td>
<td>be/4</td>
<td>bukowski</td>
<td>1395.44 K/s</td>
<td>71.27 K/s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>81178</td>
<td>be/4</td>
<td>bukowski</td>
<td>1440.46 K/s</td>
<td>71.27 K/s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>81179</td>
<td>be/4</td>
<td>bukowski</td>
<td>1395.44 K/s</td>
<td>71.27 K/s</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Three gzip processes running

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 `cbsuml1c*` 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 `iostat` output (% of time the process spends waiting for I/O operation)
  - continuously high Read-Write rates in `iostat` or `htop` report

Ultimate test: monitor performance as a function of number of concurrent 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 \( \gg N \) ?

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)
Load balancing using GNU parallel

Using a text editor, create a file called (for example) **TaskFile**
(This is **NOT** a script, just a list of commands to run)

```
gzip file1
gzip file2
gzip file3
gzip file4
gzip file5
gzip file6
gzip file7
gzip file8
gzip file9
```

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
```
Load balancing using GNU parallel tool

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

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

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

- parallel will execute tasks listed in TaskFile using up to NP instances at a time
  - 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 core becomes available
  - The (NP+2)nd task will be launched right after another core becomes available
  - ...... etc., until all tasks are distributed

- Only up to NP tasks are running at a time (less at the end)

- All NP cores always kept (on average) busy (except near the end of task list) – Load Balancing
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 `ai` 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
```
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)

`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 - this will ‘drain the queue’ (allow tasks already running to finish)

```
kil -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

```
kil -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
file2
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
```