Intro to Supercomputing

Bruno C. Mundim

March 21, 2022

About this course

We'll introduce basic concepts of "supercomputing", a.k.a. high performance computing

It is intended to be a high level primer for those largely new to HPC.

Topics will include motivation for HPC, essential issues, problem characteristics as they apply to parallelism and a high level overview of parallel computation models.

Some familiarity with the Linux command line and editing text files is preferred.

What do you need for the course?

- A computer with browser and internet connection to attend the lectures.
- A Zoom client to connect to the office hours.
- An ssh client to connect to the SciNet Teach cluster.
  - Linux and MaxOS: Use the ssh command in the terminal.
  - Windows: Use MobaXTerm https://mobaxterm.mobatek.net.

Make sure you can login to the website https://scinet.courses/1226 !

Course structure

- MONDAY: A first online lecture over Zoom (you're here!).

  An assignment will be given at the end of the lecture.

  You can ask questions:

  - in the chat during and at the end of the lecture.
  - in the forum on the course site.
  - and also during:
- WEDNESDAY: Zoom office hours.

  Submit a solution for the assignment on the course website (deadline is midnight Thursday).
- FRIDAY: A last online lecture on Zoom that will address the solution, common mistakes, and wrap-up.

Introduction

What is Supercomputing?

Supercomputing, a.k.a. High Performance Computing, is leveraging larger and/or multiple computers to solve computations in parallel.

What does it involve?

- hardware - instruction pipelines and sets, multi-processors, inter-connects.
- algorithms - concurrency, efficiency, communications.
- software - parallel approaches, compilers, optimization, libraries.

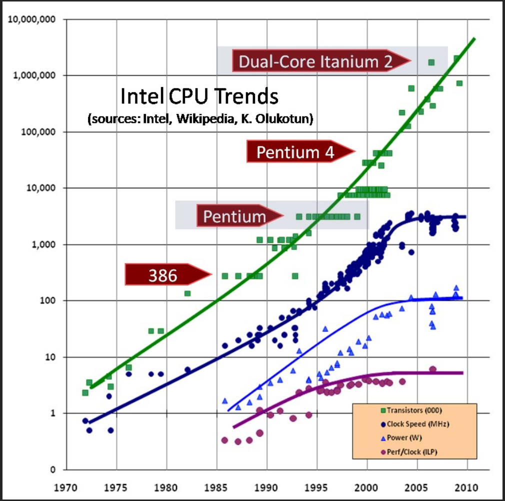
When do I need Supercomputing/HPC?

- My problem takes too long $\longrightarrow$ more/faster computation
- My problem is too big $\longrightarrow$ more memory
- My data is too big $\longrightarrow$ more storage

Examples where supercomputing is needed

- Computational Fluid Dynamics
- Molecular Dynamics and N-Body Simulations
- Smooth Particle Hydrodynamics
- Monte Carlo Simulations
- Computational Quantum Chemistry
- Bioinformatics
- Data Science and Machine Learning

The “free lunch” is over

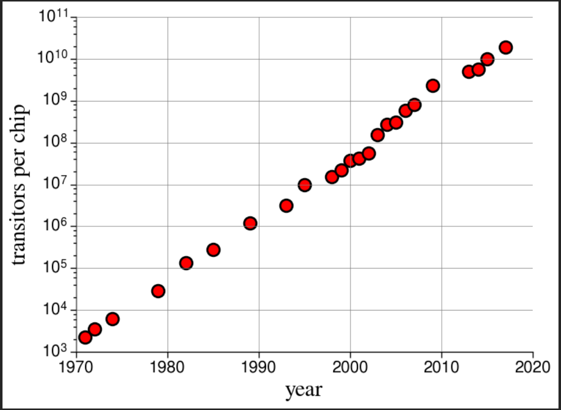


- There once was a time in which computer processor speeds steadily increased in newer generations.
- Due to physical limitations, this trend stopped around 2005, and advances in the speed of processors, memory, and storage, have plateaued.

So:

- Modern HPC means **more** hardware, not faster hardware.
- Thus **parallel** programming and computing is required.

Wait, what about Moore's Law?



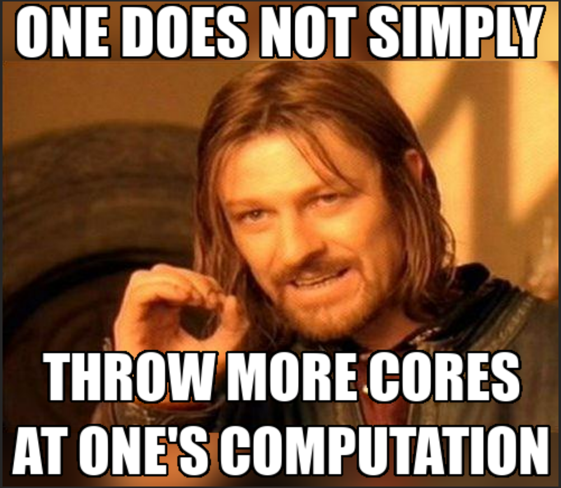
Moore's law...

describes a long-term trend in the history of computing hardware. The number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years.

But...

- Moore's Law didn't promise us clock speed.
- More transistors but getting hard to push clock speed up.
- Power density is limiting factor.
- So more cores at fixed clock speed.

All good, more cores = faster, right?



More cores is like having more workers.

HR Dilemma

Problem: job needs to get done faster

- can't hire substantially faster people
- can hire more people
- must alter workflow from a one-person job

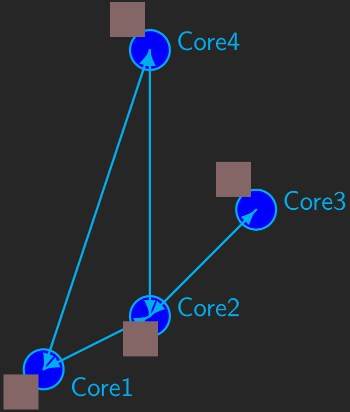
Solution:

- split work up between people (divide and conquer)
- requires rethinking the workflow process
- requires administration overhead
- eventually administration larger than actual work

Supercomputer Architectures

Clusters

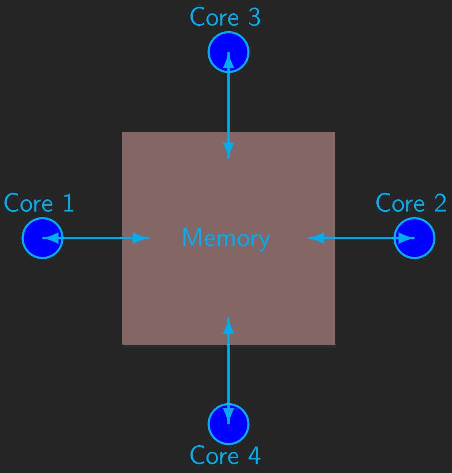
- Take existing powerful standalone computers (called a "node"),
- Link them together through a network (or "*interconnect*").
- Easy to build and easy to expand.
- Because each nodes has its own memory that the other nodes cannot see, these are called distributed memory systems.
- Nodes communicate and transfer data through messages.
- *Programming Model:*
  Message Passing Interface (MPI)



Multi-core Computers

- A collection of processors that can see and use the same memory.
- Limited number of cores, and much more expensive when the number of cores is large.
- Coordination/communication done through memory.
- Also known as shared-memory systems.
- *Programming model:* Threads (e.g. OpenMP)

Your desktop, laptop and cell phone likely use this kind of architecture.



Accelerators

- Systems with accelerators are machines which contain an “off-host” accelerator, such as a GPU or Xeon Phi.
- These accelerator devices are very fast and good at massively parallel processing (having 500-2000+ cores).
- Complicated to program.
- Programming: CUDA, OpenACC, OpenMP, and OpenCL.
- Or implicit programming using frameworks like Tensorflow.
- Needs to be combine with at least some ‘host’ code: heterogeous computing

Examples of Supercomputers

#1[a] Fugaku (at RIKEN in Japan)

Fugaku has 158,976 nodes, each with 48 cores and 32GB of memory, no GPUs, with a "Tofu" network.

#2[a] Summit (at Oak Ridge National Lab)

Summit has 4,608 Infiniband-connected nodes, each with 44 CPU cores, 6 GPUs and 600GB of memory.

#127[a] Niagara (at SciNet/UofT)

Niagara is currenty the second fastest[a] supercomputer in Canada.

It has 2,016 Infiniband-connected nodes, each with 40 CPU cores and 192GB RAM. The nodes are connected with a very fast Infiniband network in a Dragonfly+ topology.

Its GPU expansion, Mist, is like Summit, but 50x smaller. More GPUs are on Cedar, Graham, Beluga & Narval.

Teach Cluster (at SciNet)

This cluster is composed of 42 Infiniband-connected nodes, each of which has 16 cores and 32GB RAM.

[a] According to *https://www.top500.org*, a ranking based on the *HPL* benchmark.

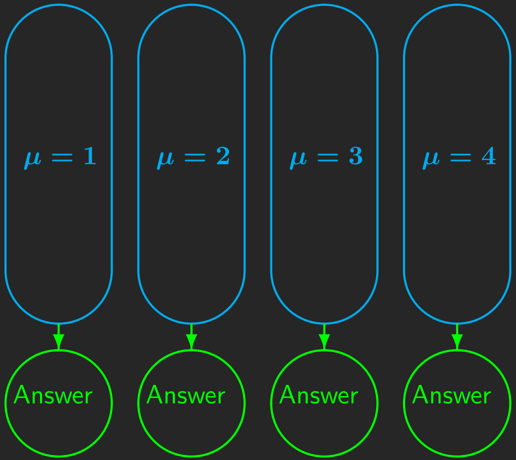
Parallel processing

Parallelism

- So: not faster compute cores, but more.
- Must have something to do for all these cores.
- Find parts of the program that can done independently, and therefore in parallel or at the same physical time.
- There must be many such parts.
- Their order of execution should not matter either.
- Data dependencies limit parallelism.

Parameter "sweep": best case scenario

- Aim is to get results from a model as a parameter varies.
- Can run the serial program on each processor at the same time.
- Get *more* done.

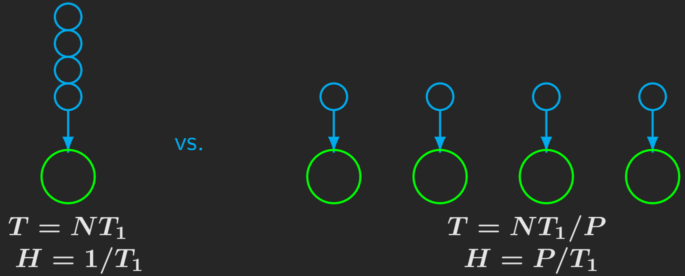


Throughput

- How many tasks can you do per unit time?

$$\text{throughput} = H = \frac{N}{T}$$

- Maximizing H means that you can do as much as possible.
- Independent tasks: using P processors increases H by a factor P



$T = NT_1$
$H = 1/T_1$

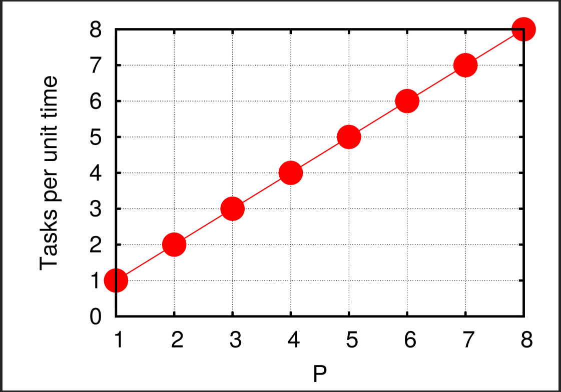
$T = NT_1/P$
$H = P/T_1$

Scaling — Throughput

- How a problem's throughput scales as processor number increases ("strong scaling").
- In this case, linear scaling:

$$H \propto P$$

- This is Perfect scaling.

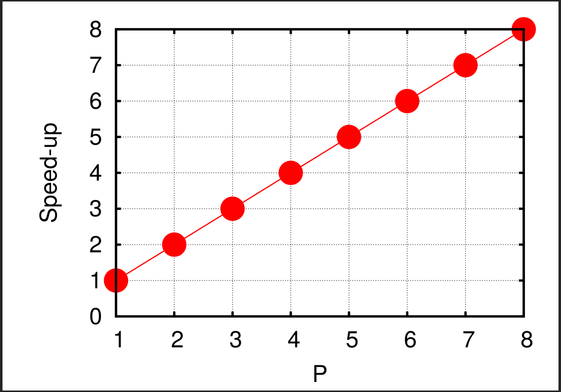


Scaling – Speedup

- How much faster the problem is solved as processor number increases.
- Measured by the serial time divided by the parallel time

$$S = \frac{T_{serial}}{T(P)} \propto P$$

- For embarrassingly parallel applications: Linear speed up.

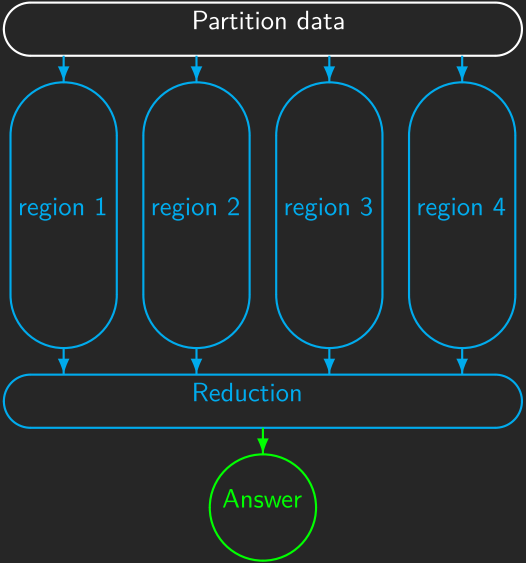


Non-Ideal Parallel Computations

Non-ideal case #1: Non-parallelizable algorithms

- Say we want to integrate some tabulated experimental data.
- Integration can be split up, so different regions are summed by each processor.
- Non-parallelizable parts of the algorithm:
  - First need to get data to processor
  - And at the end bring together all the sums: reduction

$T_s \equiv$ time for serial part (Partition+Reduction)
$T_p \equiv$ time for parallelizable part (for $P = 1$, so, the sum of all the regions on the right)



Deriving Amdahl's law

Speed-up (without parallel overhead):

$$S = \frac{T_p + T_s}{T_p/P + T_s}$$

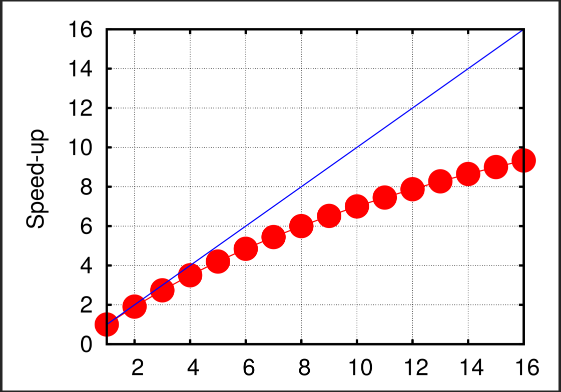
or, with $f \equiv T_s/(T_s + T_p)$ the serial fraction,

$$S = \frac{1}{f + (1-f)/P}$$

Note that

$$\lim_{P\to\infty} S = \frac{1}{f}$$

- Serial part dominates asymptotically.
- Speed-up limited, no matter size of P.



(example for $f = 5\%$)

Beating Amdahl's law

Scale up!

The larger the system size N, the smaller the serial fraction:

$$f(N) \sim \frac{1}{N}$$

Weak scaling

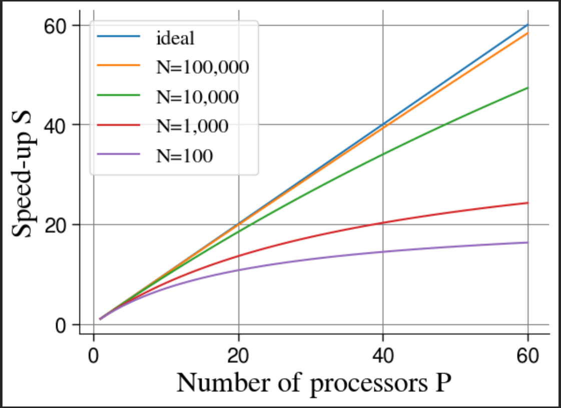
Increase problem size while increasing P

$$Time_{weak}(P) = Time(N = n \times P, P)$$

Weak scaling

Increase problem size while increasing P

$$Time_{weak}(P) = Time(N = n \times P, P)$$



Non-ideal case #2: Non-locality

- Moving data around slows things down because communication is slower than computing.
- Not computing where the data resides or was generated, requires data movement and wastes time.
- Many memory and storage systems hide locality, using caches or pulling data automatically.
- To influence the locality, you need to change the data access pattern.

Communication and data motion can rarely be completely avoided, but can be minimized.

- Shared memory systems: having data processed by the core that generated it improves locality.
- Distributed systems: not having data in the process that needs it, means more communications.
- File systems and memory: accessing data contiguously helps.

  E.g. using many separately files is not contiguous, and also requires additional I/O operations.
- Reusing data helps.

Non-ideal case #3: Load imbalance

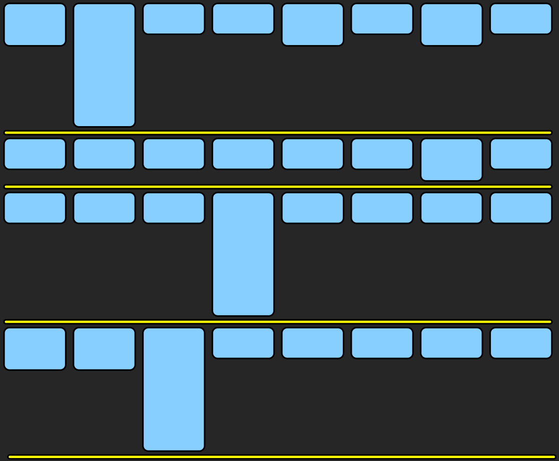
- Suppose you have 32 computations to do, and they are all independent.
- That would scale perfectly, but this time there is a catch:

  The different computations takes very different times.

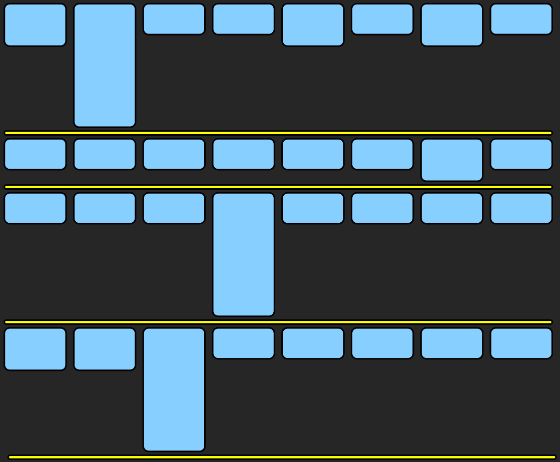
  And we can't know how long a computation will take before we run it.

- Let's say we want to run these computations on 8 cores.

Easy, right? We'll just run 4 sets of 8!

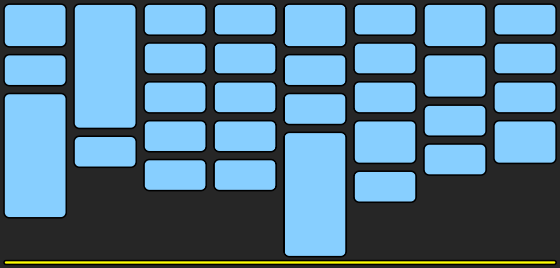


Easy, right? We'll just run 4 sets of 8!



Due to the load imbalance, only 42% used.
Speedup: $S = 3.4$ **We can do better!**

Let's give a new task as soon as a core is done:



Much better: 72% is used.
Speedup: $S = \mathbf{5.8}$

Could try to code this ourselves, but there's a tool that implements that: GNU Parallel^{+}

Before explaining this utility, let's discuss why we care so much about efficiency.

$^{+}$ O. Tange (2011): GNU Parallel - The Command-Line Power Tool, *;login: The USENIX Magazine*, February 2011:42-47.

Working on Shared, Remote Resources

May I have a supercomputer, please?

- If you need a supercomputer, your computation has outgrown your computer or laptop.
- Very few people can afford their own supercomputer: you will need to use a supercomputer that you share with potentially hundreds or thousands of other users.
- Due to the internet, resources can be used remotely, allowing for more sharing and larger systems.
- Sharing is good. Shared resources get better utilization.

  No cores need to wait e.g. because the code isn't ready or the new postdoc hasn't arrived. Someone else can use the computing time in the meantime.

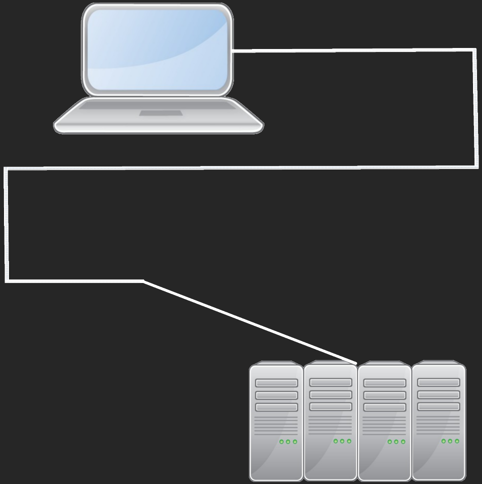
  E.g., Niagara's utilization typically lies between 94% and 98%, with jobs always in the queue.

Fact

Because supercomputers are remote and shared resources, these machines need to be used quite differently from how you use your own computer.

It's Remote

- You're at your computer ("terminal")
- The supercomputer is in a data centre somewhere ("server").
- You must connect remotely using `ssh` ("secure shell").
- You must interact with the supercomputer using the command line.
- Yes, you read that right, ***the command line!***
- Yes, you read that right, ***the command line!***
- Nobody uses a GUI in HPC. Nobody.



Hands-on: Access to SciNet's Teach cluster

Logging in

- Look up your *lcl_uothpc101sNNNN* account on the course website under the "Log In Info" section.
- Click on the password reset link, and finish that process.
- To log in, type on the command line (could be in a local terminal in MobaXTerm in Windows):

```
$ ssh lcl_uothpc101sNNNN@teach.scinet.utoronto.ca
```

  and type the password you just set.

Transfering files

- To download files from the internet to Teach when logged in, use

```
$ wget URL
```

- To copy files from your computer to Teach, or vice-versa (must not be logged in on Teach)

```
$ scp filename USERNAME@teach.scinet.utoronto.ca:path/filename
$ scp USERNAME@teach.scinet.utoronto.ca:path/filename filename
```

It's shared

- You're on the login node **together with all other folks** in this course.
- You have a home directory, called $HOME, and a scratch directory $SCRATCH (in fact, these are variables containing their true location).
- All other nodes of the Teach Cluster are compute nodes.
- To run on compute nodes, you need to create a job script that contains a request for specific resources for a specific time.
- You pass this job script to the scheduler using the sbatch command.
  The scheduler used on Teach, and on many other supercomputers, is called SLURM.
- The scheduler allocates compute resources to your job.

Hands-on: Submit a job on the Teach cluster

Follow along, if you can.

- Log into the teach cluster.

```
$ ssh USERNAME@teach.scinet.utoronto.ca
```

- Change directory to your scratch folder.

```
$ cd $SCRATCH
```

- Copy the material needed for this course.

```
$ cp -r /scinet/course/introhpc $SCRATCH
```

- Change to the newly created directory.

```
$ cd $SCRATCH/introhpc
```

- Submit the job 'sweep_bondbreak.sh'.

```
$ sbatch sweep_bondbreak.sh
```

- Check the status of your job in the queue.

```
$ squeue --me
```

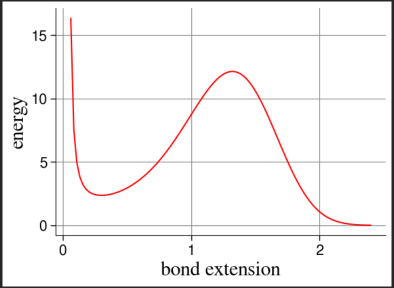
- Once completed, check the output in the file `slurm-<JobID>.out`

```
$ less slurm-*.out
```

Wait, what did we compute exactly, and how?

Simulation of chemical bond breakage

- `sweep_bondbreak.sh` runs a parameter sweep for an app called `bondbreak`.
- `bondbreak` performs a MC simulation.
- The model is for a bond between two atoms, that can break due to thermal fluctuations.
- When the bond breaks, the simulation stops and prints the breakage time.



Model parameters

- the initial bond extension
- the temperature
  (sets strength of thermal fluctuations)

These go into the job script. . .

Simulation parameters

- the timestep
- maximum time to simulate
- random seed
- name of file to write data to
- interval at which write out data
- name of file to write log messages to

How to use bondbreak

```
$ ./bondbreak --help

bondbreak - compute time to break a hypothetical chemical bond
            using a stochastics simulation

Usage:

 bondbreak [OPTION]...

  -d, --delta=DELTA       time step (default: 0.0003)
  -f, --filename=FILENAME output filename (default: output.dat)
  -h, --help              print help
  -i, --initial=INITIAL   initial bond extension (default: 1.15)
  -l, --logfile=LOGFILE   filename for log messages (default: -)
  -o, --outtime=OUTTIME   interval at which to write to file (default: 1.0)
  -r, --runtime=RUNTIME   max. time to simulate (default: 400.0)
  -s, --seed=SEED         random seed (default: 13)
  -t, --temp=TEMP         temperature (default: 1.2)

Note: Give arguments to options in long form as '--xyz ARG' or '--xyz=ARG'
or in short form as '-x ARG' or '-xARG' (but not as '-x=ARG').
```

Job script (sweep_bondbreak.sh)

```
#!/bin/bash
#SBATCH --ntasks=1
#SBATCH --time=00:20:00

module load python/3

# temperature
T=2.2

# Run multiple cases with different random seeds
for S in {1..96} ; do
  echo "Simulation $S of 96"
 ./bondbreak --temp $T --seed $S                       \
 --filename out/$T-$S.dat --logfile out/$T-$S.log
done

# Extract the breakage times from the logs
awk '/BREAKAGE DETECTED/{print $8}' out/$T-*.log
```

⟵ First line makes this a bash shell script

} #SBATCH lines request 1 core for 20 minutes

Rest runs on allocated compute node.

⟵ Most software requires module commands

⟵ Setup up parameters

⟵ A loop in the bash shell

⟵ Pass parameter to bondbreak app (filenames depending on T and S)

⟵ Collect breakage times (awk out of scope)

The SLURM Scheduler

Why a scheduler?

The compute nodes/cores need to be fairly shared among all users.

- You can't just reserve cores for particular users, or at least some of them wouldn't be utilized all the time (which is a waste, as other users could have used them).
- So instead of having fixed reservations, users must submit jobs.
- Each job must specify the resources it needs (time/cpus/gpus).
- A program called the scheduler takes those resource requests and finds a time slot and (set of) compute node(s) to allocate for the job.

On a busy system, the allocated time is usually in the future, and often unknown.

I.e., you have to wait.

Scheduling for a whole cluster is hard and takes time, therefore there are limits to how many jobs you can submit as well as a minimum size. If you have many small jobs to do, bunch them up and use GNU Parallel (more on that later).

Scheduling Factors

- Priority Allocations

  After an annual competition, yearly priority allocations are given to selected groups.
  These priorities are set to hit a target usage of a certain number of cores for a certain time.

- Past usage

  If a research group has recently used a lot of resources, their priority goes down.

- Time

  The longer a job is in the queue, the more priority it accrues.

- Available resources and job sizes

  Requests for scarce resources or for many nodes/memory can lead to much longer wait times.

  Requests for moderate resources (e.g. a single node for 30 minutes) can lead to shorter wait times if there are ‘holes’ in the schedule that it can fill.

The scheduler has to sort all jobs using these criteria & give resources to the jobs with the most priority.

Using the Scheduler

There are different schedulers, but the SciNet clusters use SLURM (as do all Compute Canada clusters).

Some of the most common parameters are:

`-t`	`--time`	amount of time
`-N`	`--nodes`	number of nodes
`-n`	`--ntasks`	number of tasks
	`--ntasks-per-node`	number of tasks per node
`-c`	`--cpus-per-task`	number of threads per task
	`--gres=...`	special requests, e.g. GPUs
	`--mem=...`	amount of memory

Commands to interact with the scheduler

`sbatch`	submit job
`squeue`	see queued jobs and their status
`scancel`	cancel a job
`seff`	see job stats after completion
`salloc/debugjob`	get short interactive job on a compute node

How to program a supercomputer

The job script is in the bash programming language, but often starts one particular application at its core.

Core applications are written in a programming languages that need to be translated into machine code that the computer can understand.

- For compiled languages, like C, C++, Fortran, or CUDA/HIP, the translation is done ahead of the computation for the application as a whole. The compiler analyzes the entire code and optimizer the resulting machine code. Some compilers can create applications that run on GPUs (CUDA/HIP and certain OpenMP/OpenACC compilers).
- For scripted applications, like bash, Python and R, translation is done while the application runs, one line at a time. This tends to be much more inefficient than compiled language, but scripted languages tend to be easier to learn and more flexible.

Scripted languages may use packages that are themselves written in a compiled languages (e.g. Numpy and SciPy), or frameworks that compile on the fly (e.g. Tensorflow) to alleviate the inefficiencies associated with scripting languages.

GNU Parallel

Managing subjobs of different durations.

- Surprisingly versatile (perl) script, especially for text input.
- Gets your many cases assigned to different cores and on different nodes without much hassle.
- Invoked using the `parallel` command, after doing:

```
module load gnu-parallel
```

- O. Tange, *GNU Parallel - The Command-Line Power Tool* ;login: **36** (1), 42-47 (2011)
- http://www.gnu.org/software/parallel/parallel_tutorial.html

GNU Parallel example

- Load the gnu-parallel module in your script.
- The "`-j ...`" flag indicates you wish GNU parallel to run 16 subjobs at a time.
- The "`--nodes`" parameter is important here to make sure all allocated cores are on the same node.

  (Running GNU Parallel across nodes is quite possible, but requires extra flags.)
- If you can't fit as many subjobs onto a node as there are cores due to memory constraints, specify a different value for the "-j" flag.
- Put all the commands for a given subjob onto a single line.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=16
#SBATCH --time=1:00:00
#SBATCH --job-name=gnuparallelx16

module load intel/2018.2 gsl/2.4
module load gnu-parallel

# Run the code on 16 cores.
parallel -j $SLURM_TASKS_PER_NODE <<EOF
cd jobdir1; ../app; echo "job 1 done"
cd jobdir2; ../app; echo "job 2 done"
...
cd jobdir200; ../app; echo "job 200 done"
EOF
```

What does GNU parallel do?

- GNU parallel assigns subjobs to the processors.
  - As subjobs finish it assigns new subjobs to the free processors.
  - It continues to assign subjobs until all subjobs in the subjob list are assigned.
- Consequently there is built-in load balancing!
- You can use GNU parallel across multiple nodes as well.
- It can also log a record of each subjob, including information about subjob duration, exit status, *etc.*

GNU Parallel syntax

Some commonly used arguments for GNU parallel:

- `--jobs NUM`, sets the number of simultaneous subjobs.

  By default, parallel uses the maximum number of cores (16/80 on Teach/Niagara nodes). Same as `-j N`.
- `--joblog LOGFILE`, causes parallel to output a record for each completed subjob. The records contain information about subjob duration, exit status, and other goodies.
- `--resume`, when combined with `--joblog`, continues a GNU parallel job that was killed prematurely or did not finish all subjobs.
- `--pipe`, splits stdin into chunks given to the stdin of each subjob.
- `--memfree SIZE`, sets minimum memory free when starting another subjob.

GNU Parallel - replacement strings

- In the previous example, the commands GNU Parallel is to run were read from standard input.
- A lot of those commands contain the same parts.
- Instead, we can specify on the command line that part of the commands that is the same, with values for placeholders to be read as from standard input.

For instance, this:

```
parallel <<EOF
cd jobdir1;../app;echo "job 1 done"
cd jobdir2;../app;echo "job 2 done"
...
cd jobdir200;../app;echo "job 200 done"
EOF
```

is equivalent to

```
parallel 'cd jobdir{};../app;echo "job{} done"'<<EOF
1
2
...
200
EOF
```

The replacement string here is {}.
There are other replacement strings that can remove extensions, paths, etc.

GNU Parallel - command line argument data

We can also give the arguments on the command line instead of as standard input.

```
parallel 'cd jobdir{};../app;echo "job{} done"' <<EOF
1
2
...
200
EOF
```

is equivalent to

```
parallel 'cd jobdir{};../app;echo "job{} done"' ::: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

But bash has sequence expressions, so we can generate a list of 200 lines with `{1..200}`, and write:

```
parallel 'cd jobdir{};../app;echo "job{} done"' ::: {1..200}
```

Multiple placeholders are also possible using this technique, e.g.

```
parallel 'echo {2} {1}' ::: {1..10} ::: {0..200..50}
```

which prints out all combinations of the elements of each set.

Hands-on Assignment

Assignment Task

- The script `sweep_bondbreak.sh` executes 96 repeats of the computation of the bond breakage time, one by one.
- These could all run in parallel.
- How long did this script take?
- Create a modified version of `sweep_bondbreak.sh`, `sweep_bondbreak_parallel.sh`, that uses GNU Parallel to parallelize the computation using 16 cores on a single compute node of the Teach cluster.
- Submit this new script to the scheduler. How long did the new script take?
- Questions? Ask them in the forum, and/or attend the hands-on session.
- The script and its output should be submitted to the course website by Thursday, March 24, 11:55 PM (EDT).