



# Advanced Research Computing Training for EDIA Champions

Ramses van Zon  
December 5, 2024



Digital Research  
Alliance of Canada

Alliance de recherche  
numérique du Canada





## Topics in this session:

- 1 What is Advanced Research Computing?**
- 2 ARC Resources**
- 3 Accessing and Operating ARC Resources**
- 4 Programming a Supercomputer**
- 5 Final Tips and Conclusions**





# What is Advanced Research Computing?





## I.e., what are we talking about here?

**Research Computing** Whenever you're doing research that depends on computer.

**Advanced Research Computing (ARC)** Whenever your own computer no longer suffices; you may need expert advice.

**Supercomputing** When the computer that suffices is a large, shared, custom system or cluster.

**High-Performance Computing (HPC)** When the speed of your computation matters, and you need parallel processing. I.e., nearly always.





## So, ARC:

### is needed when:

- My problem takes too long → more/faster computation
- My problem is too big → more memory
- My data is too big → more storage

### involves:

- hardware - cpus, multi-processors, network
- algorithms - parallelism, efficiency
- software - parallel programming, compilers, optimization, libraries, apps
- data management - RDM plan, data transfer, storage





## Examples where ARC is needed



*(the Niagara supercomputer)*

- Computational Fluid Dynamics
- Molecular Dynamics and N-Body Simulations
- Smooth Particle Hydrodynamics
- Monte Carlo Simulations
- Computational Quantum Chemistry
- Bioinformatics
- Digital Humanities
- Data Science and Machine Learning
- (insert your research area here)

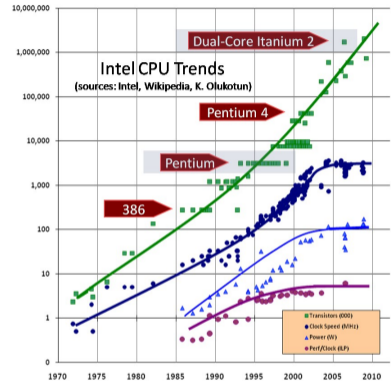


# The free lunch is over

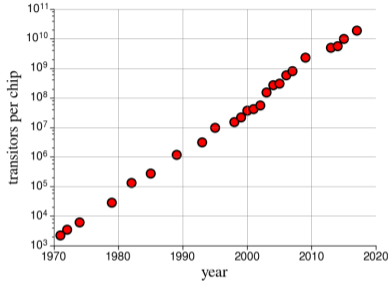
- There once was a time in which computer processor speeds steadily increased in newer generations.
- Due to physical limitations, this trend stopped around 2005, and advances in the speed of processors, memory, and storage, have plateaued.

So:

- Modern HPC means **more** hardware, **not faster** hardware.
- Thus **parallel** 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 speed.
- More transistors but getting hard to push clock speed up.
- Power density is limiting factor.
- Instead: more cores at fixed clock speed.



## All good, more cores = faster, right?



Solution:

- split work up between people
- requires rethinking the workflow process
- requires administrative overhead

More cores is like having more workers.

### ***Human Resources Analogy***

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



# ARC Resources





# This is not your laptop!

The architecture of supercomputers is different than that of your own computer, and this matters.

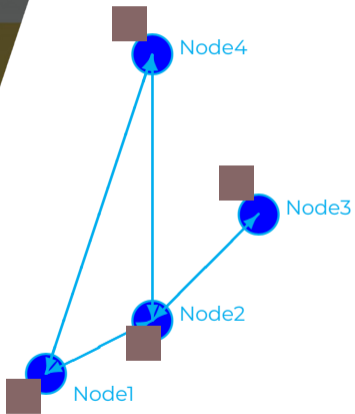
There are a few prototypical architectures you should be aware of:

- Clusters
- Multi-core computers
- Accelerators





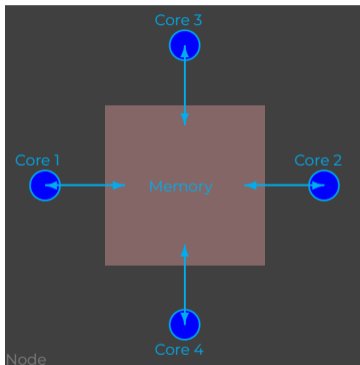
# Clusters



- Take existing powerful standalone computers, called **nodes**.
- Link them together through a **network** (a.k.a an “*interconnect*”).
- Easy to build and easy to expand.
- Because each node ● has its **own memory** a.k.a. **RAM** ■, these are called **distributed memory systems**.
- Nodes communicate and transfer data through messages.



# Multi-core computers



- A collection of processors on one node 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**.

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



# Accelerators



- Systems with accelerators have nodes which contain a device like a **GPU**.
- Accelerators are very fast and good at massively parallel processing (having 500-2000+ GPU cores).
- More complicated to program.
- Implicit programming using frameworks like **Tensorflow** or **PyTorch**.
- Needs to be combined with at least some 'host' code on the CPU cores: **heterogeneous computing**





## Top 5 supercomputers (November 2024)

### #1 El Capitan (at Lawrence Livermore National Labs in the USA)

11,136 nodes, each with 96 cores and 4 GPUs, 512GB of memory, with a “Slingshot” network.

### #2 Frontier (at Oak Ridge National Labs in the USA)

9,472 nodes, each with 64 cores and 8 GPUs, 512GB of memory, with a “Slingshot” network.

### #3 Aurora (at Argonne National Labs in the USA)

10,624 nodes, each with 104 cores and 6 GPUs, 1TB of memory with a “Slingshot” network.

### #4 Eagle (in Microsoft Azure in the USA)

1,800 nodes, each with 48 cores, 8 GPUs, with an “Infiniband” network.

### #5 HPC6 (at Eni in Italy)

3,472 nodes, each with 64 cores and 4 GPUs, with a “Slingshot” network.

See <https://www.top500.org>, a ranking based on the *HPL* benchmark.



# Available supercomputers in Canada

## #190 Narval (Calcul Québec)

<https://docs.alliancecan.ca/wiki/Narval>

1,181 nodes with 64 cores, 256+GB RAM, infiniband network.

159 nodes with 48 cores and 4 GPUs, 512 GB RAM, same network.

## #282 Niagara (SciNet/U.Toronto)

<https://docs.alliancecan.ca/wiki/Niagara>

2,024 nodes with 40 cores, 192GB RAM, with a dragonfly infiniband network.

*Its GPU expansion, Mist, is a separate cluster, like a 50x smaller Summit.*

## #301 Cedar (at SFU)

<https://docs.alliancecan.ca/wiki/Cedar>

724 nodes with 32 cores, 128+GB RAM, OmniPath network.

1,408 nodes with 48 codes, 187+GB RAM, same network.

338 nodes with 24 cores and 4 GPUs, 128+GB RAM, same network.

## Béluga (Calcul Québec)

<https://docs.alliancecan.ca/wiki/Beluga>

802 nodes with 40 cores, 92+GB RAM, Infiniband network.

172 nodes with 40 cores and 4 GPUs, same network.

## Graham (SHARCNET/U.Waterloo)

<https://docs.alliancecan.ca/wiki/Graham>

1119 nodes with 32+ cores, 125+GB RAM, 160 nodes with 32 cores and 2 GPUS, 124GB RAM, ...





## Working on shared, remote resources

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

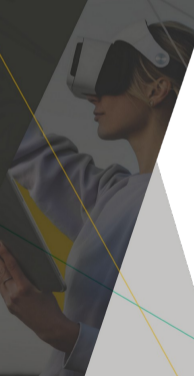
---

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



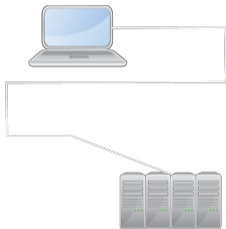


# Accessing and Operating ARC Resources





## ARC systems are 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.





# Demonstration: access to Narval

## Logging in

**Narval** is one of the Alliance's ARC clusters.

- If you do not have an account, we'll get to that.
- If you have an account, you can follow along.
- To log in, type on the command line (could be in a local terminal in MobaXTerm in Windows):

```
$ ssh USERNAME@narval.alliancecan.ca
```

and type the password.

- You'd be asked for your Second Factor (MFA is mandatory).



# Transferring files

- To download files when logged in to an ARC cluster, use

```
$ wget URL
```

- To copy files to or from your computer:

```
$ scp filename USERNAME@narval.alliancecan.ca:path/filename  
$ scp USERNAME@narval.alliancecan.ca:path/filename filename
```

But these commands must be used on your terminal, and not when already logged into the ARC cluster.

- You have a home directory and a scratch directory.  
You may also have a group-based project directory.

The home directory is stored in the environment variable `$HOME`, and is backed up.

The scratch directory may be stored in `$SCRATCH`, or present as a link in your `$HOME`. It is not backed up, but much larger than `$HOME`.



## Getting access

- 1** Register with the Alliance CCDB:  
[https://ccdb.alliancecan.ca/account\\_application](https://ccdb.alliancecan.ca/account_application)  
PIs must get an account one first, so they can sponsor your account at no cost.  
The approval process typically takes 1-2 business days.
- 2** Setup Multi-factor Authentication  
[https://docs.alliancecan.ca/wiki/Multifactor\\_authentication](https://docs.alliancecan.ca/wiki/Multifactor_authentication)
- 3** Recommended: Setup SSH keys

### For Niagara (and likely other clusters in the future)

- 4** Go to  
[https://ccdb.alliancecan.ca/services/opt\\_in](https://ccdb.alliancecan.ca/services/opt_in)  
and click on the “Join” button next to Niagara and Mist.
- 5** After a business day or two, you get an email confirming your access to Niagara and Mist.



## ARC systems are shared

- We're now on a **login node** together with all other folks working on this Narval instance. **Login nodes are not for computing!**
- All other nodes of a cluster like *Narval* 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**.  
The scheduler used on *Narval* is called **SLURM**.  
To submit a job, use the **sbatch** command.
- The scheduler allocates compute resources to your job and runs it in due time.





## Demonstration #2: running jobs

- Log in to *Narval*
- Change directory to scratch
- Download code
- Change to the new directory
- Submit the job 'sweep\_bondbreak.sh'
- Check the status of your job(s)
- Once completed:

```
ssh USERNAME@narval.alliancecan.ca
```

```
cd scratch
```

```
wget https://tinyurl.com/iatgz -O ia.tgz  
tar xzvf ia.tgz
```

```
cd ia
```

```
sbatch sweep_bondbreak.sh
```

```
squeue --me
```

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





## Job script: sweep\_bondbreak.sh

```
#!/bin/bash
#SBATCH --ntasks=1
#SBATCH --time=01:00:00
#SBATCH --mem=1000M

# This file runs a parameter sweep for
# an application called 'bondbreak'.

module load StdEnv/2023 python/3.10.13 scipy-stack/2024b

T=2.2      # temperature value
NUMSEEDS=500 # number of seeds
OUT=output-$TEMP-$SLURM_JOB_ID
mkdir -p $OUT

# Run multiple cases with different random seeds
for S in $(seq $NUMSEEDS) ; do
    echo "Simulation $$ of $NUMSEEDS"
    ./bondbreak -t $T -s $$ -f $OUT/$T-$S.dat -l $OUT/$T-$S.log
done

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



## 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, priority allocations are given to selected groups.

- **Past usage**

If a 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 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

Some of the most common sbatch parameters are:

---

amount of time	-t	--time
number of nodes	-N	--nodes
number of tasks	-n	--ntasks
number of tasks per node		--ntasks-per-node
number of threads per task	-c	--cpus-per-task
number of gpus per node	-G	--gpus-per-node
amount of memory		--mem
scheduler account to use	-A	--account
use reserved nodes		--reservation

---





## Using the scheduler

Commands to interact with the scheduler:

---

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

---





# Programming a Supercomputer



# 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.



# Parallelism and concurrency

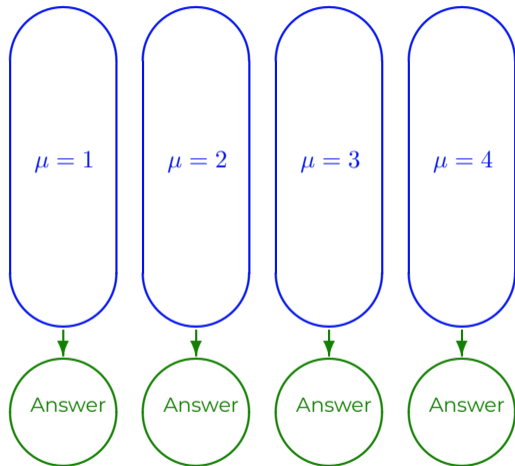
- Remember: there are no faster compute cores, just more.
- Speedup must come from having something to do for all these cores.
- Find parts of the program that can be done independently, and therefore concurrently.
- There must be many such parts.
- Their order of execution should not matter either.
- Data dependencies limit concurrency.





## 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.

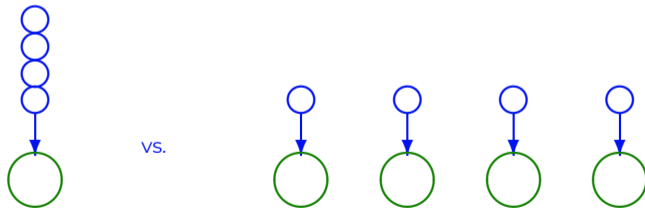


# Throughput

- 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  $P$



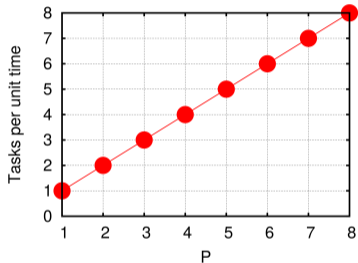


## Scaling --- throughput

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

$$H \propto P$$

- This is **Perfect scaling**.

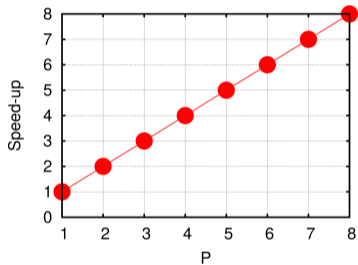


## Scaling -- speed-up

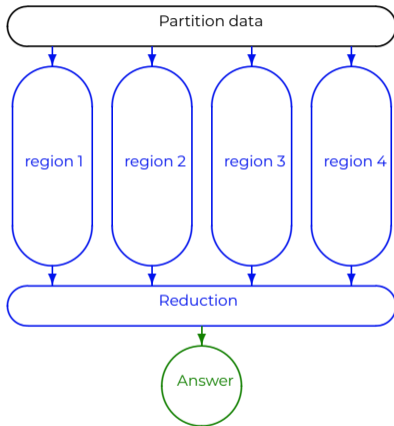
- How much faster the problem is solved as processor number increases.
- Measured by the serial time divided by the parallel time

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

- For embarrassingly parallel applications: Linear speed up.



## Non-ideal case 1: non-parallelizable work



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

# Amdahl's law

$T_s \equiv$  time for serial part (Partition+Reduction)

$T_p \equiv$  time for parallelizable parts combined

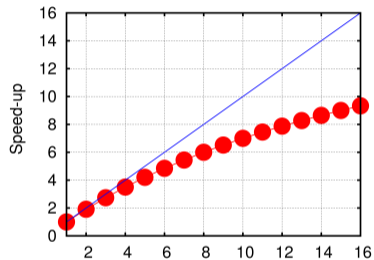
Speed-up, with  $f \equiv T_s / (T_s + T_p)$  the serial fraction,

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

Note that

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

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



(example for  $f = 5\%$ )



## 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.**



## Non-ideal case #3: load imbalance

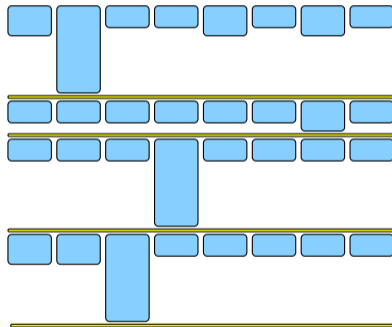
- Suppose you have 32 computations to do, and they are all independent.
- That would scale perfectly, but this time there is a catch:

**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.

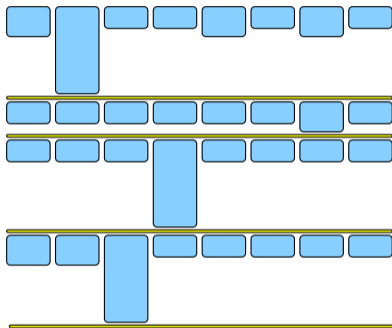
Easy, right? We'll just run 4 sets of 8!





## Non-ideal case #3: balance the load

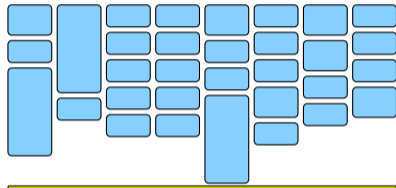
Easy, right? We'll just run 4 sets of 8!



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:



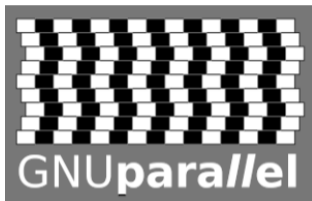
Much better: 72% is used.

Speedup:  $S = 5.8$

*Could try to code this ourselves, but there's a tool that implements that: **GNU Parallel**<sup>+</sup>*

<sup>+</sup> O. Tange (2018): GNU Parallel 2018, March 2018, <https://doi.org/10.5281/zenodo.1146014>.

# GNU Parallel: managing subjobs of different durations



- Versatile tool, especially for text input.
- Gets your many cases assigned to different cores without much hassle.
- Invoked using the `parallel` command.

- O. Tange (2018): GNU Parallel 2018, March 2018, <https://doi.org/10.5281/zenodo.1146014>.
- <https://www.gnu.org/software/parallel>

# GNU Parallel example

- Load the required modules
- The “-j ...” flag indicates for GNU parallel to run 8 subjobs at a time.
- The “--nodes” parameter makes sure all allocated cores are on the same node.
- If you can't fit as many subjobs onto a node as there are cores due to memory constraints, specify a different “-j” value.
- Put the commands for a given subjob separate lines.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH --time=1:00:00
#SBATCH --job-name=gnumparallelx8
#SBATCH --mem=4000M
module load python/3.10.13
module load scipy-stack/2023a
P=$SLURM_CPUS_PER_TASK
# Distribute 32 runs over 8 cores.
parallel -j $P <<EOF
cd jobdir1; ../app; echo "1 done"
...
cd jobdir32; ../app;echo "32 done"
EOF
```



## How does this balance the load?

- 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.





## Demonstration #3: compute in parallel

- The script `sweep_bondbreak.sh` executes 500 repeats of the computation of the bond breakage time, one by one.
- How long did `sweep_bondbreak.sh` take? Too long!
- These 500 cases could run in parallel.
- Now let's use the modified version of `par_sweep_bondbreak.sh` that uses GNU Parallel to parallelize the computation using 8 cores on a single compute node of *Narval*.
- After submitting this modified script to the scheduler, let's see how it took now.





# Final Tips and Conclusions



## No magic speedups

- Just asking for more resources is not enough.
- Conversely, not asking for resources (e.g. leaving out `--cpus-per-task`) kills any parallelization.
- Note that output can be in any order when things run in parallel.
- Executing scripts on the command line makes them run on the login node. Not what you want!





## Do not trust code you do not understand

- Beware of copy-paste: Don't include lines you do not understand.
- Related: Be wary of recipes you find online. Even if they are for Linux, and all supercomputers run Linux, they are often written for a machine that you own and have exclusive administrative rights to.







## Use best practices with your code

- There is no such thing as one-off codes or scripts. If it was worth writing, it's worth running again, and someone (could be you) could use it.
- If it's a big, ununderstandable mess, no one wants to touch it and all the effort will need to be repeated. Plus, do you really trust its results if you do not understand it?
- Professional software developers deal with this all the time, but computational researchers are focused on results.
- Putting in a bit of effort in writing maintainable, reproducible code, however, can pay off big in future projects.
- So let's consider some common, useful best practices in research computing.





# Automate everything



- Write scripts for any computation or data processing.
- This means you have a precise record, you can rerun them, and you can adapt them.
- If GUIs are part of your workflow, try to replace it with commands.

GUIs are particularly awkward for record keeping: instead of a script, you would need to keep a written log of every mouse movement, click and keystroke.

- Use makefiles or cmake for automating building software.



# Use version control



- Version Control is a tool for managing changes in a set of files.
- Keeps historical versions for easy tracking.
- It essentially takes a snapshot of the files (code) at a given moment in time.

## Why use it?

- Makes collaborating on code easier/possible/less violent.
- Helps you stay organized.
- Allows you to track changes in the code.
- Allows reproducibility in the code.
- And when something goes wrong, you can back up to the last working version.



# Comment and document everything

You must document your code. Must. Not optional. Documentation of code comes in many forms:

- sensible variable, function, class, and module names.
- comments in the code.
- help commands, doc-strings, or other built-in feedback.

## But why?

Six months from now you're not going to remember the motivation for writing that function.

So write it down in the code somewhere.

Write a README or a full manual if you expect others to use the code without reading all of it.

*Note: there is no such thing as self-documenting code.*





## Make your code and scripts portable

- Try it on a **different computer**.
- **Don't hard-code** paths, use variables like `$HOME`.
- Use `#!/usr/bin/env` in **shebangs** for scripts.  
(and always use shebangs!)
- Make as few assumption about the (super)computer the code will run on as possible.  
Add an **explicit requirements** file with your code.
- Stick to programming language **standards**.  
This avoid the “but it works on my computer” bug.





# Checkpoint

- Things can go wrong while your job is running.
  - Could be due to your job (e.g. takes too long or too much memory and crashes)
  - Could be due to your account (e.g. out of space).
  - Could be due to the system (e.g. file system issues)
  - Could be due to the external infrastructure (e.g. power outage)
- To make sure not to have wasted computation time in these cases, you **checkpoint**
- Checkpointing is writing out enough information about the state of the system to allow the computation to restart at the checkpoint.
  - Checkpoint could be implemented in the operating system and scheduler, but this is so inefficient that is rarely is.
  - So your application needs to include a feature to just write and read the checkpoint data.
- GNU Parallel has this feature, using `--joblog` and `--resume`. It won't checkpoint within sub-jobs, though.






# Optimize I/O

- Home, scratch, and project all use a parallel file system.
- Your files can be seen on all login and compute nodes.
- These are high-performance file systems which provides rapid reads and writes to large data sets in parallel from many nodes.
- But accessing data sets which consist of many, small files leads to poor performance.

## Do's and don'ts

- Avoid reading and writing lots of small amounts of data to disk.
  - Many small files on the system would waste space and would be slower to access, read and write.
  - Write numerical data out in binary. Faster and takes less space.
  - Some file systems are better than others I/O heavy jobs and checkpoints.
  - When your files fit in memory, use ramdisk, which lives in memory.
- 



## Want to learn more?

about Linux, scheduling, programming, research data management, etc... ?

Check the following resources:

### Wiki

<https://docs.alliancecan.ca>

### Training Calendar

<https://alliancecan.ca/en/services/advanced-research-computing/technical-support/training-calendar>

### Regional LMSs and training:

<https://scinet.courses>

<https://training.sharcnet.ca/courses>

<https://www.acenet.training>

<https://www.eventbrite.com/o/calcul-quebec-8295332683>

<https://training.westdri.ca/events/upcoming-training-fall-2024/>







## Conclusions

I hope we've enlightened you about the business of advanced research computing.

Common traits in using ARC include:

- Shared resource, multiple users
- Linux command line
- Batch computing
- Parallel processing
- Shared, parallel file systems
- Scheduling resources for fair sharing
- Remote connections
- Graphical interfaces are rare



***Thank you for your attention!***

