Introduction to High-Performance Computing

Using clusters to speed up your research

ALEX RAZOUMOV
alex.razoumov@westgrid.ca

✔ slides and data files at `http://bit.ly/introhpc`

- the link will download a file `introHPC.zip` (∼3 MB)
- unpack it to find `codes/` and `slides.pdf`

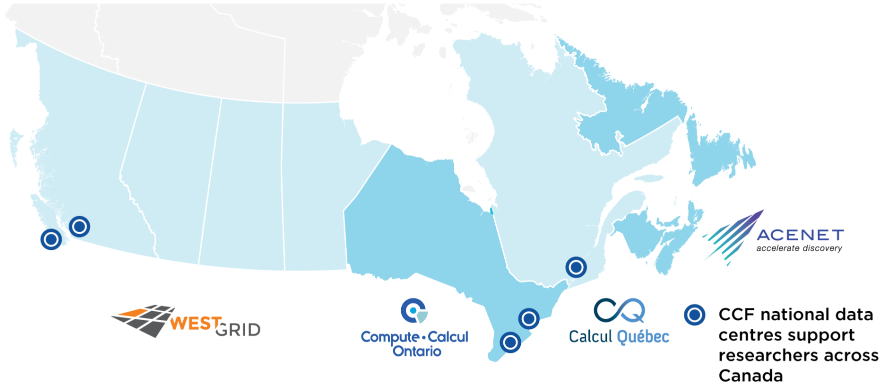
Workshop outline

- To work on the remote cluster today, you will need
  1. Wi-Fi access (Eduroam? Campus guest account?)
  2. an SSH client
     - pre-installed on Linux/Mac
     - `http://mobaxterm.mobatek.net` Home Edition on Windows
  3. only on the production clusters:
     - a Compute Canada account `http://bit.ly/2K0ol4S` (approved by the PI) or a guest account
     - be added to the Slurm reservation
- Cluster hardware overview
- Basic tools for cluster computing
  - logging in, transferring files
  - software environment, modules
  - Linux command line, editing remote files
- Programming languages and tools
  - overview of languages from HPC standpoint
  - parallel programming environments
  - compilers
  - quick look at OpenMP, MPI, Chapel, make
- Working with Slurm scheduler
- Debugging and very briefly on profiling
- Best practices (common mistakes)

Hardware overview

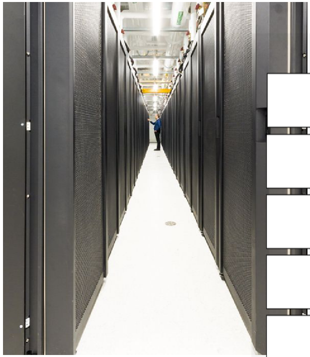
Added to CC's national infrastructure in the past several years

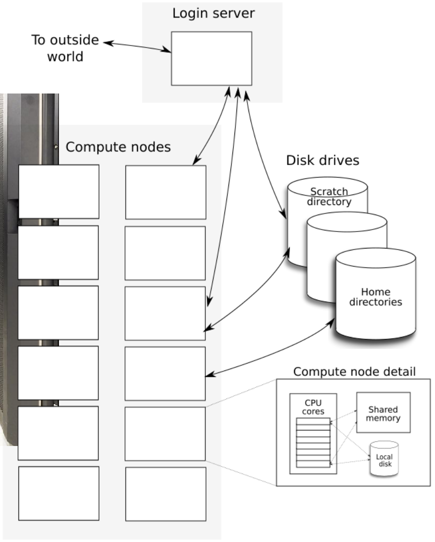
- **Arbutus** @UVic is an extension to West Cloud (in production since September 2016)
- General-purpose clusters: **Cedar** @SFU and **Graham** @UofWaterloo (both in production since June 2017), **Béluga** @McGill (in production since April 2019)
- Large parallel cluster: **Niagara** @UofToronto (in production since April 2018)



HPC cluster overview

- Mostly off-the-shelf components for individual nodes, everything rack-mounted
- Typically hundreds of nodes, wired by fast interconnect
- Shared vs. distributed memory
- Login vs. compute nodes
- Compute nodes: CPU-only, GPU nodes (accelerators)
- Job scheduler
- Development/visualization nodes





	cedar.computecanada.ca	*graham.computecanada.ca*	*beluga.computecanada.ca*
purpose	general-purpose cluster for a variety of workloads		
specs	`https://docs.computecanada.ca/wiki/Cedar`	`https://docs.computecanada.ca/wiki/Graham`	`https://docs.computecanada.ca/wiki/Beluga`
processor count	58,416 CPUs and 584 GPUs	35,520 CPUs and 320 GPUs	34,880 CPUs and 688 GPUs
interconnect	100Gbps Intel OmniPath, non-blocking to 1024 cores	100Gbps Mellanox InfiniBand, non-blocking to 1024 cores	InfiniBand 100Gbps or better
base nodes	**576 nodes**: 128GB, 32 cores **640** *SL* **nodes**: 192GB, 48 cores	**864 nodes**: 128GB, 32 cores	**172** *SL* **nodes**: 96GB, 40 cores **516** *SL* **nodes**: 192GB, 40 cores
large-memory nodes	**128 nodes**: 256GB, 32 cores **24 nodes**: 512GB, 32 cores **24 nodes**: 1.5TB, 32 cores **4 nodes**: 3TB, 32 cores	**56 nodes**: 256GB, 32 cores **24 nodes**: 512GB, 32 cores **3 nodes**: 3TB, 64 cores	**12** *SL* **nodes**: 768GB, 40 cores
GPU base	**114 nodes**: 128GB, 24 cores, 4 NVIDIA P100 Pascal 12GB GPUs **32 nodes**: 256GB, 24 cores, 4 NVIDIA P100 Pascal 16GB GPUs	**160 nodes**: 128GB, 32 cores, 2 NVIDIA P100 Pascal 12GB GPUs	**172 nodes**: 192GB, 40 cores, 4 NVIDIA Volta V100 16GB GPUs

➥ All nodes have on-node SSD storage ➥ *SL* = Skylake ➥ On Cedar can use `--constraint=broadwell` or `--constraint=skylake` to specify CPU

	niagara.computecanada.ca
purpose	for large parallel jobs, ideally ≥1,000 cores with an allocation
specs	`https://docs.computecanada.ca/wiki/Niagara` and `https://docs.scinet.utoronto.ca`
processor count	60,000 CPUs and no GPUs
interconnect	EDR Infiniband (Dragonfly+, completely connected topology, dynamic routing), 1:1 to 432 nodes, effectively 2:1 beyond that
base nodes	**1,500 *SL* nodes**: 192GB, 40 cores

- No local disk, nodes booting off the network, small RAM filesystem
- All cores are Intel Skylake (2.4 GHz, AVX512) ➥ *SL* = Skylake
- Authentication via CC accounts; **for now, need to request a SciNet account**; long-term regular access for all CC account holders
- **Scheduling is by node** (in multiples of 40 cores)
- Users with an allocation: job sizes up to 1000 nodes and 24h max runtime
- Users without an allocation: job sizes up to 20 nodes and 12h max runtime
- Maximum number of jobs per user: running 50, queued 150

Accessing resources: RAS vs. RAC

- ~15% of compute cycles available via the Rapid Access Service (RAS)
  - available to all CC users via default queues
  - you can start using it as soon as you have a CC account
  - shared pool with resources allocated via "fair share" mechanism
  - will be sufficient to meet computing needs of many research groups
- ~85% of compute cycles allocated via annual Resource Allocation Competitions (RAC)
  - apply if you need >50 CPU-years or >10 GPU-years
  - only PIs can apply, allocation per research group
  - announcement in the fall of each year via email to all users
  - 2019 RAC: 507 applications, success rate (awarded vs. requested): 41% of CPUs, 20% of GPUs, 86% of storage, 95% of virtual CPUs

File systems

Details at `https://docs.computecanada.ca/wiki/Storage_and_file_management`

filesystem	quotas	backed up?	purged?	performance	mounted on compute nodes?
$HOME	**50GB, 5e5 files per user, 100GB Niagara**	nightly, latest snapshot	no	medium	yes
$SCRATCH	**20TB Cedar, 100TB Graham, 25TB Niagara, 1e6 files per user** except when full	no	yes	high for large files	yes
`/project` (long-term disk storage)	**1TB, 5e5 files per user** could be increased via RAC, Niagara only via RAC	nightly	no	medium	yes
`/nearline` (tape archive)	**5TB per group** via RAC	no	no	medium to low	no
`/localscratch`	**none**	no	maybe	very high	local

- Wide range of options from high-speed temporary storage to different kinds of long-term storage
- On Cedar no longer allowed to submit jobs from `$HOME`
- For frequent I/O *use on-node SSD* (Slurm-generated directory `$SLURM_TMPDIR` → `/localscratch/${USER}.${SLURM_JOBID}.0`) or *RAM disk* (`$TMPDIR` → `/tmp`) ✍ don't forget to move files out before your job terminates !!!
- To check disk usage: `quota` command (aliased to `diskusage_report`)
- To request more storage: **support@computecanada.ca** for small increases, RAC for large requests

Basic tools for working with a cluster

Logging into the systems

- On Mac or Linux in terminal:

```
$ ssh userXXX@206.12.90.125   # guest account on our training cluster (where XX=01..60)
$ ssh yourUsername@cedar.computecanada.ca    # use your CC account on Cedar
$ ssh yourUsername@graham.computecanada.ca   # use your CC account on Graham
```

- On Windows many options:
  - MobaXTerm `https://docs.computecanada.ca/wiki/Connecting_with_MobaXTerm`
  - PuTTY `https://docs.computecanada.ca/wiki/Connecting_with_PuTTY`
  - Secure Shell Extension `http://bit.ly/2FeLp0R` in Chrome browser
  - bash from the Windows Subsystem for Linux (WSL) in Win10 – need to enable developer mode and then WSL
- SSH key pairs are very handy, save you from typing passwords
  - implies secure handling of private keys, non-empty passphrases
  - `https://docs.computecanada.ca/wiki/SSH_Keys`
  - `https://docs.computecanada.ca/wiki/Using_SSH_keys_in_Linux`
  - `https://docs.computecanada.ca/wiki/Generating_SSH_keys_in_Windows`
- GUI connection: X11 forwarding (through ssh), VNC, x2go
- Client-server workflow in selected applications, both on login and compute nodes

Linux command line

- All our systems run Linux (CentOS 7) ⇒ you need to know basic command line
  - separate 3-hour *"Bash Scripting and Tools"* session
  - attend a Software Carpentry *bash* session `https://software-carpentry.org/workshops`
  - lots of tutorials online, e.g. tutorials 1 – 4 at `http://bit.ly/2vH3j8v`
- Much typing can be avoided by using bash aliases, functions, ~/.bashrc, hitting TAB

File commands

ls	directory listing
ls -alF	pass command arguments
cd dir	change directory to dir
cd	change to home
pwd	show current directory
mkdir dir	create a directory
rm file	delete file
rm -r dir	delete directory
rm -f file	force remove file
rm -rf dir	force remove directory
cp file target	copy file to target
mv file target	rename or move file to target
ln -s file link	create symbolic link to file
touch file	create or update file

Paths

relative vs. absolute paths
meaning of **~ . ..**

File commands

command > file	redirect command output to file
command >> file	append command output to file
more file	page through contents of file
cat file	print all contents of file
head -n file	output the first n lines of file
tail -n file	output the last n lines of file
tail -f file	output the contents of file as it grows

Process management

top	display your currently active processes
ps	display all running processes
kill pid	kill process ID pid

File permissions

chmod -R u+rw,g-rwx,o-rwx file	set permissions

Environment variables and aliases

export VAR='value'	set a variable
echo $VAR	print a variable
alias ls='ls -aFh'	set an alias command

Searching

grep pattern files	search for pattern in files
command | grep pattern	example of a pipe
find . -name '*.txt' | wc -l	another pipe

Other tools

man command	show the manual for command
command --help	get quick help on command
df -kh .	show disk usage
du -kh .	show directory space usage

Compression and archiving

tar cvf file.tar files	create a tar file
tar xvf file.tar	extract from file.tar
gzip file	compress file
gunzip file.gz	uncompress file.gz

Loops

for i in *tex; do wc -l $i; done	loop example

Bash walk-through on the cluster

1. Connect to the cluster
2. Can you tell the difference between local and remote shells?
3. Run `whoami` and `hostname`
4. Bring up a manual page on some command, also try `--help` flag
5. Check files in your home directory: are there any hidden files? lists files by time of last change
6. Check file permissions
7. Check out different filesystems – `$HOME`, `$SCRATCH`, `$PROJECT` – what are their paths?
8. Play with paths, try to use both absolute and relative paths, use special characters . .. ~ /
9. Create a directory, put a file with some contents into it (with a text editor), look at this file from the command line with `more` and `cat`
10. Copy this file into another file, try moving files, delete a file, delete a directory
11. Create several files, put them into a *gzipped tar archive*
12. Move this archive into another directory, unpack it
13. Create a new directory, download `http://hpc-carpentry.github.io/hpc-intro/files/bash-lesson.tar.gz` into it and unpack it there
14. How much space do the unpacked files take?
15. Count the number of lines in all `*.fastq` files
16. Try to redirect output from the last command into a file
17. Try pipes: construct a one-line command to display the name of the longest (by the number of lines) file
18. Search inside files with `grep`
19. Find files with `find`
20. Write and run a bash script (need to start with #!/bin/bash – called *shebang*) making it executable
21. Print out some shell variables
22. Write a quick loop to display each `*.fastq` file's name, its number of lines, and then its first two lines, separating individual files with an empty line
23. Write a quick loop to remove `SRR` from each `*.fastq` file's name (use `${name:3:14}` syntax)

Editing remote files from the command line

- For novice users: `nano` (easiest option)
- For power users: `emacs -nw` (super configurable, lots of shortcuts)
- For die-hard fans: `vi` and `vim` (basic, difficult to use)
- Remote graphical `emacs` not recommended
  - you would connect via ssh with an X11 forwarding flag (-X, -Y)

Editing remote files (cont.)

- My favourite option: local `emacs` on my laptop editing remote files via ssh with emacs's built-in package `tramp`
  - need to add to your ~/.emacs

```
(require 'tramp)
(setq tramp-default-method "ssh")
```

  - only makes sense with a working ssh-key pair

```
--- on your laptop
$ chmod go-rwx ~/.ssh
$ /bin/rm -rf ~/.ssh/id_rsa*
$ ssh-keygen -b 2048 -t rsa -f ~/.ssh/id_rsa   # enter a non-empty passphrase
$ cat ~/.ssh/id_rsa.pub | ssh yourUsername@cedar.computecanada.ca \
      'cat >>.ssh/authorized_keys'
--- on the cluster
$ chmod 700 ~/.ssh
$ chmod 640 ~/.ssh/authorized_keys
```

  - your private key is your key to the cluster, so don't share it!

Cluster software environment at a glance

- **Programming languages**: **C/C++, Fortran 90**, Python, R, Java, Matlab, **Chapel –** several different versions and flavours for most of these
- **CPU parallel development support**: MPI, OpenMP, Chapel
- **GPU parallel development support**: CUDA, OpenCL, OpenACC
- **Job scheduler**: Slurm open-source scheduler and resource manager
- **Popular software**: installed by staff, listed at `https://docs.computecanada.ca/wiki/Available_software`
  - lower-level, not performance sensitive packages installed via Nix package manager
  - general packages installed via EasyBuild framework
  - everything located under `/cvmfs`, loaded via modules (next slide)
- Other software
  - email *support@computecanada.ca* with your request, or
  - can compile in your own space (feel free to ask staff for help)

Software modules

- Use appropriate modules to load centrally-installed software (might have to select the right version)

```
$ module avail <name>          # search for a module (if listed)
$ module spider <name>         # will give a little bit more info
$ module list                  # show currently loaded modules
$ module load moduleName
$ module unload moduleName
$ module show moduleName       # show commands in the module
```

- All associated prerequisite modules will be automatically loaded as well
- Modules must be loaded before a job using them is submitted
  - alternatively, can load a module from the job submission script

File transfer

In Mac/Linux terminal or in Windows MobaXterm or bash/WSL you have two good options:

1. use `scp` to copy individual files and directories

```
$ scp filename yourUsername@cedar.computecanada.ca:/path/to
$ scp yourUsername@cedar.computecanada.ca:/path/to/filename localPath
```

2. use interactive `sftp`
3. use `rsync` to sync files or directories

```
$ flags='-av --progress --delete'
$ rsync $flags localPath/*pattern* yourUsername@cedar.computecanada.ca:/path/to
$ rsync $flags yourUsername@cedar.computecanada.ca:/path/to/*pattern* localPath
```

Windows MobaXTerm allows GUI drag-and-drop
Windows PuTTY uses `pscp` command for secure file transfer

Globus file transfer

Details at `https://docs.computecanada.ca/wiki/Globus`

- The CC Globus Portal `https://globus.computecanada.ca` is a fast, reliable, and secure service for big data transfer (log in with your CC account)
- Easy-to-use web interface to automate file transfers between any two *endpoints*
  - an *endpoint* could be a CC system, another supercomputing facility, a campus cluster, a lab server, a personal laptop (requires Globus Connect Personal app)
  - runs in the background: initialize transfer and close the browser, it'll email status
- Uses GridFTP transfer protocol: much better performance than scp, rsync
  - achieves better use of bandwidth with multiple simultaneous TCP streams
  - some ISPs and Eduroam might not always play well with the protocol (throttling)
- Automatically restarts interrupted transfers, retries failures, checks file integrity, handles recovery from faults
- Command-line interface available as well

Programming languages and tools

High-level overview of programming models

- Installed serial compilers and interpreters: C, C++, Fortran, Python, R, Java
- For more details see `https://docs.computecanada.ca/wiki/Programming_Guide`

In HPC speed matters. Not all languages are built equal in terms of performance!

- Native loops and arithmetic in Python are 80-200X slower than optimized compiled C/C++/Fortran
  - Python compilers and accelerators (Cython, Numba, Nuitka, etc.) try to improve things to some extent
  - use precompiled numerical libraries such as numpy and scipy
  - call C/C++/Fortran functions from Python
- Native R is even slower ... designed for desktop-scale statistical computation and graphics
  - very popular in engineering, mathematics, statistics, bioinformatics
  - there are some ways to accelerate R to an “acceptable balance” of coding time investment vs. performance
- Matlab, Java are also slowish (3-10X compared to optimized compiled C/C++/Fortran)
- Later (when submitting serial jobs) we'll do an exercise to time `pi.c` vs. `pi.py`
  - you can find both codes in `codes/` directory
  - make sure to use the same value of n

Parallel programming environment

- CPU parallel development support: OpenMP (since 1997), MPI (since 1994)
  - OpenMP is a language extension for C/C++/Fortran provided by compilers, implements shared-memory parallel programming
  - MPI is a library with implementations for C/C++/Fortran/Python/R/etc, designed for distributed-memory parallel environments, also works for CPU cores with access to common shared memory
  - industry standards for the past 20+ years
- Chapel is a open-source parallel programming language
  - ease-of-use of Python + performance of a traditional compiled language
  - combines shared- and distributed-memory models; data and task parallelism for both
  - multi-resolution: high-level parallel abstractions + low-level controls
  - in my opinion, by far the best language to learn parallel programming $\Rightarrow$ we teach it as part of HPC Carpentry, in summer schools and full-day Chapel workshops
  - experimental support for GPUs
  - relative newcomer to HPC, unfortunately still rarely used outside its small/passionate community
- GPU parallel development support: CUDA, OpenCL, OpenACC

Installed compilers

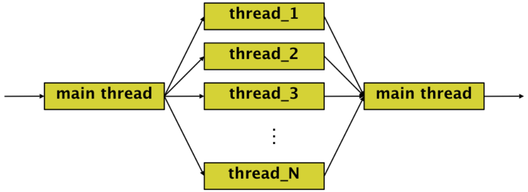
	Intel intel/2016.4 and openmpi/2.1.1 loaded by default		**GNU** module load gcc/5.4.0 (*)		**PGI** module load pgi/17.3 (*)	
C	icc	mpicc	gcc -O2	mpicc	pgcc	mpicc
Fortran 90	ifort	mpifort	gfortran -O2	mpifort	pgfortran	mpifort
C++	icpc	mpiCC	g++ -O2	mpiCC	pgc++	mpiCC
OpenMP flag	-qopenmp		-fopenmp		-mp	

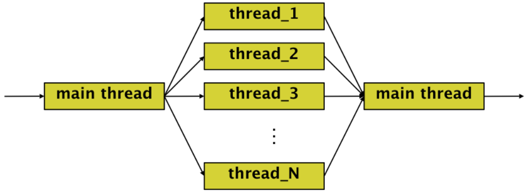
(*) in both cases intel/2016.4 will be unloaded and openmpi/2.1.1 reloaded automatically

- mpiXX scripts invoke the right compiler and link your code to the correct MPI library
- use `mpiXX --show` to view the commands they use to compile and link

OpenMP quick look

- OpenMP is a language extension (C, C++, Fortran) for parallel programming in a SMP environment $\Rightarrow$ pure OpenMP is always limited to a single node
- Programmer uses compiler directives to define "parallel regions" in code which are executed in separate threads
  1. runs the master thread until the first parallel region is encountered
  2. creates a team of parallel threads
  3. when the team threads complete all commands in the parallel region, they synchronize and terminate, leaving only the master thread





MPI quick look

- MPI library available for all popular programming languages (C, C++, Fortran, Python, R, Java, ...)
- Each processor runs exactly the same copy of the code
- Pseudo-code to exchange variables between two processors (point-to-point operation), starting with A on proc0 and B on proc1:

```
rank <- MPI function to find the current task
if rank == 0
send A to 1
receive B from 1
else if rank == 1
receive A from 0
send B to 0
endif
```

MPI quick look (cont.)

- Point-to-point or collective communications
- Each processor runs exactly the same copy of the code
- Pseudo-code to calculate a sum using collective reduce operation:

```
sum = 0, partialSum = 0
np <- MPI function to find the total number of tasks
rank <- MPI function to find the current task
decide if I am MASTER (rank=0) or WORKER (rank=1, ..., np-1)
compute partialSum: 1/np-th of the total work based on rank
if I am MASTER
receive from WORKERS their partialSum
compute sum from all partialSum's
print sum
else if I am WORKER
send to MASTER partialSum
endif
```

MPI quick look (cont.)

- More complex MPI operations
  1. collective communication routines
  2. derived data types
  3. communicators and virtual topologies

command	send buffer				receive buffer			
	P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_4
MPI_Send()+ MPI_Recv()		A					A	
MPI_Sendrecv		A	B			B	A	
MPI_Bcast		A			A	A	A	A
MPI_Gather	A	B	C	D		A,B,C,D		
MPI_Scatter		A,B,C,D			A	B	C	D
MPI_Allgather	A	B	C	D	A,B,C,D	A,B,C,D	A,B,C,D	A,B,C,D
MPI_Reduce	A	B	C	D		r(A,B,C,D)		
MPI_Allreduce	A	B	C	D	r(A,B,C,D)	r(A,B,C,D)	r(A,B,C,D)	r(A,B,C,D)

Chapel quick look

High-level abstractions for task and data parallelism

	single locale shared memory parallelism	multiple locales distributed memory parallelism likely shared memory parallelism

task parallel	`config var numtasks = 2;`
	`coforall taskid in 1..numtasks do`
	`  writeln("this is task ", taskid);`
		`forall loc in Locales do`
		`  on loc do`
		`    writeln("this locale is named ", here.name);`

data parallel	`var A, B, C: [1..1000] real;`
	`forall (a,b,c) in zip(A,B,C) do`
	`  c = a + b;`
		`use BlockDist;`
		`const mesh = {1..100,1..100} dmapped`
		`  Block(boundingBox={1..100,1..100});`
		`var T: [mesh] real;`
		`forall (i,j) in T.domain do`
		`  T[i,j] = i + j;`

Locality and parallelism are orthogonal concepts in Chapel: can even have serial execution on mutiple locales

Build tools: make

- Tool for automating builds, typically in a workflow with multiple dependencies
- Most frequent usage: source files changed ⇒ recompile parts of the code
  - consider a large software project with hundreds of source code files, e.g., *Enzo*: 426 C++ files, 6 C files, 121 fortran77 files, 10 fortran90 files, 48 header files
  - typically work on a small section of the program, e.g., debugging a single function, with much of the rest of the program unchanged ⇒ would be a waste of time to recompile everything (with *Enzo* typically 30-40 mins with heavy optimization) every time you want to compile/run the code
- Another example: updated data files ⇒ redraw the figure ⇒ rebuild the paper
- Hard or impossible to keep track of:
  - what depends on what
  - what's up-to-date and what isn't (don't want to redo everything from scratch every time)

Build tools: make (cont.)

- Idea: use a build manager to automate the process
- Need to describe the following in a build file (often called a *makefile*)
  - dependencies for each target, e.g. an executable depends on source code files
  - commands used to update targets
- The manager program will aid you in your large workflow
  - checks whether sources are older than targets
  - if not $\Rightarrow$ rebuild
- Most widely used build manager is *make*
  - invented in 1975, evolved into a programming language of its own
  - `https://docs.computecanada.ca/wiki/Make`
  - for large projects there are even pre-processor/build tools for make (CMake, etc.)

Make: very simple example with three source files

main.f90

```
program main
  implicit none
  real*8 :: a, b, add, sub
  a = 4.
  b = 1.
  print*, add(a,b)
  print*, sub(a,b)
end program main
```

add.f90

```
function add(a,b)
  implicit none
  real*8, intent(in) :: a, b
  real*8 :: add
  add =  a + b
  return
end function add
```

sub.f90

```
function sub(a,b)
  implicit none
  real*8, intent(in) :: a, b
  real*8 :: sub
  sub =  a - b
  return
end function sub
```

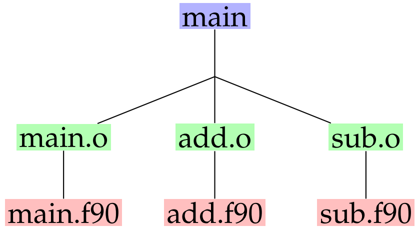
Make: very simple example with three source files (cont.)

1. Compiler stage: each .f90 file is converted into an object code (.o) which the computer understands directly

```
gfortran -c add.f90
gfortran -c sub.f90
gfortran -c main.f90
```

2. Linker stage: linking all object codes to optional libraries to produce an executable program `main`

```
gfortran main.o add.o sub.o -o main
```



Make: first take (long version)

- Start with three files `main.f90`, `add.f90`, `sub.f90`
- Create a file called `Makefile`, put the following in it:

```
main: main.o add.o sub.o
<TAB> gfortran main.o add.o sub.o -o main
main.o: main.f90
<TAB> gfortran -c main.f90
add.o: add.f90
<TAB> gfortran -c add.f90
sub.o: sub.f90
<TAB> gfortran -c sub.f90
clean:
<TAB> /bin/rm -rf *.o main
```

- The format is:

```
target: prerequisites
<TAB> rule to make the target
```

- Compile by typing `make` or `make main`

Make: first take (shorter version)

- Let's remove redundancy using wildcards (%) and predefined makefile variables

```
main: main.o add.o sub.o
<TAB> gfortran $^ -o $@
%.o: %.f90
<TAB> gfortran -c $^
clean:
<TAB> @/bin/rm -rf *.o main
```

- Predefined makefile variables:
  - `$@` is "the target of this rule"
  - `$^` is "all prerequisites of this rule"
  - `$<` is "the first prerequisite of this rule"
  - `$?` is "all out-of-date prerequisites of this rule"
- "@" means silent run (without echoing the command)

Make: exercise

- Let's create a makefile for compiling π to replace the following lines:

initial command	replacement
`gcc -O2 pi.c -o serial`	`make serial`
`gcc -O2 -fopenmp sharedPi.c -o openmp`	`make openmp`
`mpicc distributedPi.c -o mpi`	`make mpi`

- On Cedar no need to load any modules before compiling (defaults to Intel compilers)

Other essential tools

- Version control (*git* or *mercurial*) – normally taught as a 3-hour Software Carpentry course
- Terminal multiplexer (*screen* or *tmux*)
  - share a physical terminal between several interactive shells
  - access the same interactive shells from many different terminals
  - very useful for persistent sessions, e.g., for compiling large codes
- *VNC* and *x2go* clients for remote interactive GUI work
  - on Cedar in `$HOME/.vnc/xstartup` can switch from *twm* to *mwm*/etc. as your default window manager
  - can run VNC server on compute nodes
- Understand client-server workflows in other, non-VNC applications: Jupyter notebook, ParaView, VisIt, etc.
  - we'll take a look at running a Python notebook shortly

Python

Details at `https://docs.computecanada.ca/wiki/Python`

- Initial setup:

```
module avail python      # several versions available
module load python/3.7.0
virtualenv astro    # install Python tools in your $HOME/astro
source ~/astro/bin/activate
pip install numpy jupyter pandas   # all these will go into your $HOME/astro
...
```

- Usual workflow:

```
source ~/astro/bin/activate   # load the environment
python
...
deactivate
```

- On the same machine create as many virtual environments as you want, switch between them easily
- Ideally you would work on the command line (graphics would be saved to a file) and *eventually* script all your workflows, and then run them as *offscreen batch jobs* on compute nodes
  - few interactive nodes; interactive workflows might not be fully utilizing resources
  - for those (hopefully) rare occasions when you need interactivity and on-screen graphics, you can start a remote Jupyter notebook (next slide) – you can run it on a *login* (poor security + login node policies and limitations) or *compute* (better) node

Jupyter notebook inside a scheduled job

Details at `https://docs.computecanada.ca/wiki/Jupyter`

- Before you do this
  ... stop and think whether you should be running a Jupyter notebook on a cluster
- One-time setup:

```
source ~/astro/bin/activate    # load the environment
pip install jupyter    # if you have not done so already
echo -e '#!/bin/bash\nunset XDG_RUNTIME_DIR\njupyter notebook --ip $(hostname -f) --no-browser' \
     > $VIRTUAL_ENV/bin/notebook.sh
chmod u+x $VIRTUAL_ENV/bin/notebook.sh
```

- Run the Jupyter notebook server on a compute node:

```
source ~/astro/bin/activate    # load the environment
salloc --time=1:0:0 --ntasks=1 --cpus-per-task=1 --mem-per-cpu=1024M
~/astro/bin/notebook.sh
```

- Write down the node's name, the Jupyter notebook server listening port, and the token. Next, use them to connect your laptop's browser to this notebook:

```
ssh userXXX@206.12.90.125 -L 8888:node1:8888
open http://localhost:8888/?token=308cbf318cda9880829445f9baae225ea3d85169d98e43b0
```

Jupyter notebook inside a scheduled job (cont.)

- Connecting to a remote Jupyter notebook from Windows
  `https://docs.computecanada.ca/wiki/Jupyter#From_Windows`
- A Jupyter notebook can start an RStudio session via RStudio Launcher
  - adds an RStudio Session option to the Jupyter notebook's `New` dropdown list
  - uses Jupyter notebook's token authentication system
  - details at
    `https://docs.computecanada.ca/wiki/Jupyter#RStudio_Launcher`

R on cluster’s command line

Details at `https://docs.computecanada.ca/wiki/R`

```
$ module spider r      # several versions available
$ module load r/3.4.3
$ R
> install.packages("sp")    # install packages from cran.r-project.org; it’ll suggest
# installing into your personal library $HOME/R/
$ R CMD INSTALL -l $HOME/myRLibPath package.tgz    # install non-CRAN packages
```

- Running R scripts: `Rscript script.R`
- Installing and running Rmpi: see our documentation
- pbdR (*Programming with Big Data in R*): high-performance, high-level interfaces to MPI, ZeroMQ, ScaLAPACK, NetCDF4, PAPI, etc. `http://r-pbd.org`
- Launching multiple serial R calculations via *array jobs* (details in Scheduling)
  - inside the *job submission script* use something like

```
Rscript script${SLURM_ARRAY_TASK_ID}.R
```

or

```
export params=${SLURM_ARRAY_TASK_ID}
Rscript script.R
```

and then inside `script.R`:

```
s <- Sys.getenv('params')
filename <- paste('/path/to/input', s, '.csv', sep='')
```

Scheduling and job management

figures and some material in this section borrowed from Kamil Marcinkowski

Frequently asked cluster questions

- Why does my job take such a long time to start?
- Is there anything that can be done to make my job start more quickly?
- Why does my job's start time estimate keep moving into the future?

Why job scheduler?

- Tens of thousands of CPUs, many thousands of simultaneous jobs $\Rightarrow$ need an automated solution to manage a queue of pending jobs, allocate resources to users, start/stop/monitor jobs $\Rightarrow$ we use Slurm open-source scheduler/resource manager
  - efficiency and utilization: we would like all resources (CPUs, GPUs, memory, disk, bandwidth) to be all used as much as possible, and minimize gaps in scheduling between jobs
  - minimize turnaround for your jobs
- Submit jobs to the scheduler when you have a calculation to run; can specify:
  - walltime: maximum length of time your job will take to run
  - number of CPU cores, perhaps distribution across nodes
  - memory (per core or total)
  - if applicable, number of GPUs
  - Slurm partition, reservation, software licenses, ...
- Your job is automatically started by the scheduler when enough resources are available
  - standard output and error go to file(s)

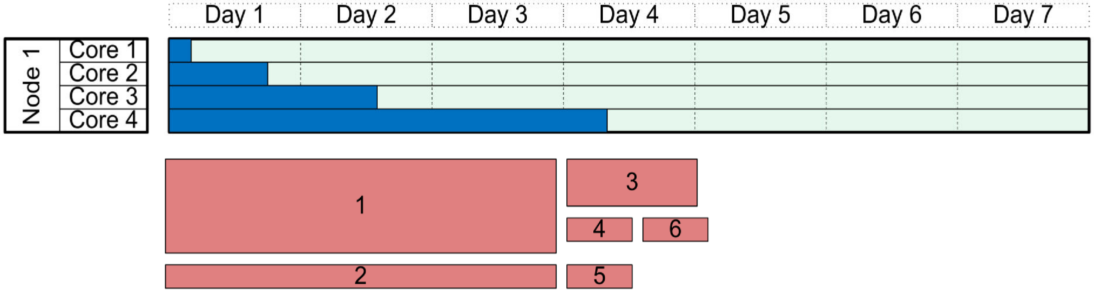
Fairshare mechanism

Allocation based on your previous usage and your *"share"* of the cluster

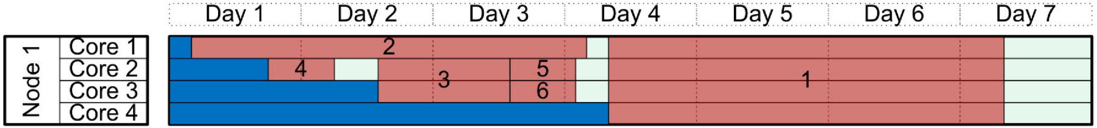
- *Priority*: one per research group (not per user!), ranges from 6 (= high, default) to 0 (= low)
- Each group has a *share target*
  - for regular queues: *share* $\propto$ the number of group members
  - in RAC: *share* $\propto$ the awarded allocation (important projects get a larger allocation)
- If a research group has used more than its *share* during a specific interval (typically 7-10 days) $\Rightarrow$ its priority will go down, and vice versa
  - the exact formula for computing priority is quite complex and includes some adjustable weights and optionally other factors, e.g., how long a job has been sitting in the queue
  - no usage during during the current fairshare interval $\Rightarrow$ recover back to level 6
- Higher priority level can be used to create short-term bursts
- Reservations (specific nodes) typically only for special events

Job packing: simplified view

- Consider a cluster with 4 running jobs and 6 newly submitted jobs
- Scheduled jobs are arranged in order of their users' priority, starting from the top of the priority list (jobs from users with the highest priority)
- Consider 2D view: cores and time
- In reality a multi-dimensional rectangle to fit on a cluster partition: add memory (3rd dimension), perhaps GPUs (4th dimension), and so on, but let's ignore these for simplicity



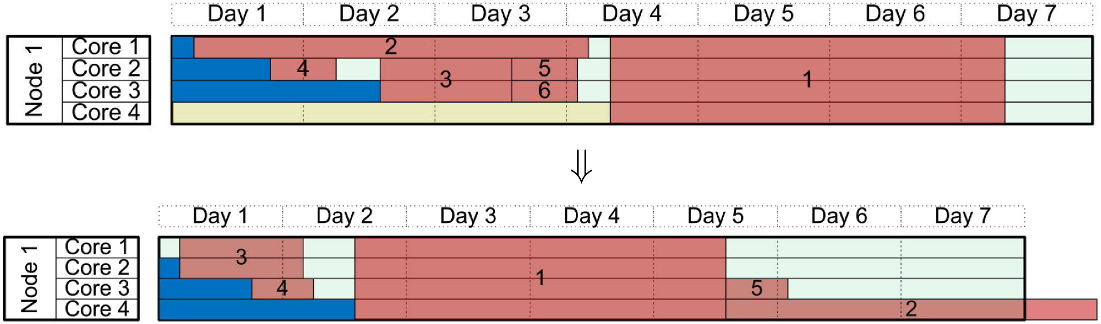
Jobs are scheduled in order of their priority. Highest-priority job may not run first!



- *Backfill*: small lower-priority jobs can run on processors reserved for larger higher-priority jobs (that are still accumulating resources), if they can complete before the higher-priority job begins

Why does my job's start time estimate keep moving into the future?

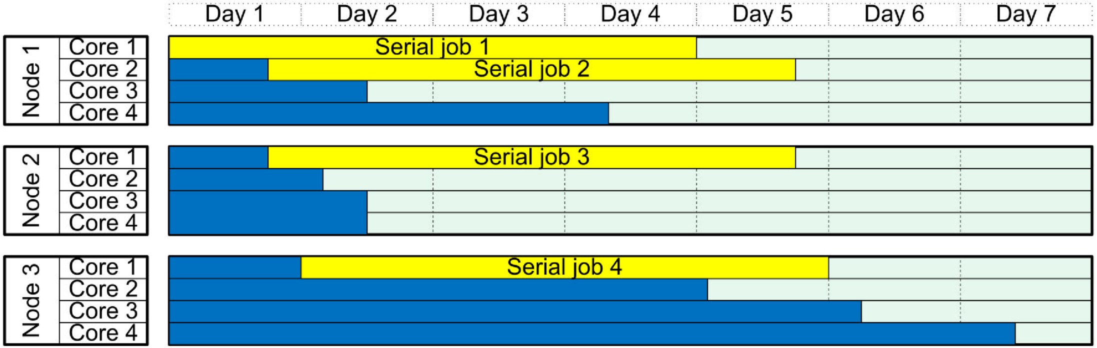
- If a running job finishes early, or a waiting job is canceled, or a new higher priority job is added to the queue $\Rightarrow$ all waiting jobs are rescheduled from scratch in the next cycle, again in the order of their priority
- This will change the estimated start time, and not always in the obvious direction (notice what happens to job #2 in the graph below!)



Job billing: goes into determining your priority

- Recall: base nodes have 128GB and 32 cores per node $\Rightarrow$ effectively 4GB per core
- Job billing is by core and memory (via core-equivalents: 4GB = 1 core), whichever is larger
  - this is fair: large-memory jobs use more resources
  - a 8GB serial job running for one full day will be billed 48 core-hours

Many serial jobs on a cluster



- Each job uses a single CPU: easiest and most efficient to schedule, excellent scaling linear speedup
- Submitting many serial jobs is called “serial farming” (perfect for filling in the parameter space, running Monte Carlo ensembles, etc.)
- In your job script you can ask for a serial job with `#SBATCH --ntasks=1` (this is the default if not specified)

Scheduler: submitting serial jobs

```
$ icc pi.c -o serial
$ sbatch [other flags] job_serial.sh
$ squeue -u username [-t RUNNING] [-t PENDING] # list all current jobs
$ sacct -j jobID [--format=jobid,maxrss,elapsed] # list resources used by completed job
```

```
#!/bin/bash
#SBATCH --time=00:05:00    # walltime in d-hh:mm or hh:mm:ss format
#SBATCH --job-name="quick test"
#SBATCH --mem=100      # 100M
./serial
```

- `--account=...` needed only on production systems if you have more than one allocation (RAS / RAC / reservations), used for "billing" purposes (not the same as your cluster account!)
- `--reservation=...` used only for special events
- On Cedar/Graham in this school we will use `--account=ucss19-wa_cpu` and `--reservation=ucss19-wr_cpu`
- On the training cluster no need to specify `--account`, and there are no reservations
- It is good practice to put all flags into a job script (and not the command line)
- Could specify number of other flags (more on these later)

Exercises: simple serial jobs

1. Submit a serial job that:
   - runs the `hostname` command
2. Monitor your job with `squeue -u $USER`, check your email, print output file
3. How much memory did the job use?
4. Now try timing `pi.c` vs. `pi.py` on a compute node, using either:
   - `time <command>`, or
   - Slurm's walltime reporting

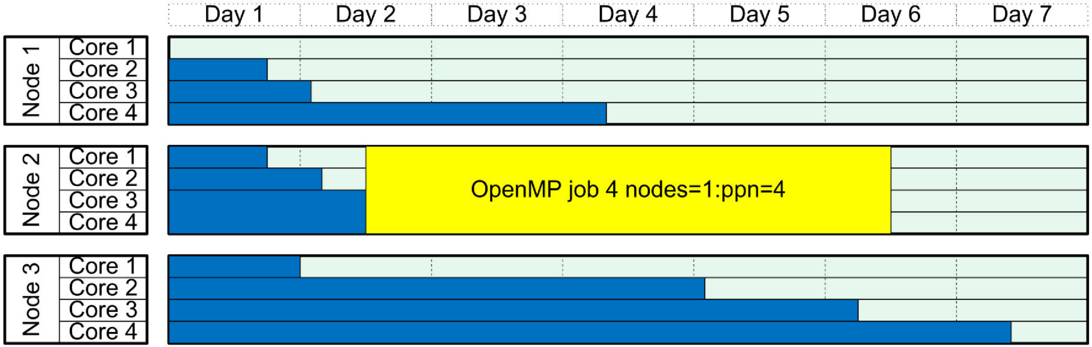
Scheduler: submitting array jobs

- Job arrays are a handy tool for submitting many serial jobs that have the same executable and might differ only by the input they are receiving through a file
- Job arrays are preferred as they don't require as much computation by the scheduling system to schedule, since they are evaluated as a group instead of individually
- In the example below we want to run 30 times the executable "myprogram" that requires an input file; these files are called input1.dat, input2.dat, ..., input30.dat, respectively

```
$ sbatch job_array.sh [other flags]
```

```
#!/bin/bash
#SBATCH --array=1-30          # 30 jobs
#SBATCH --job-name=myprog     # single job name for the array
#SBATCH --time=02:00:00       # maximum walltime per job
#SBATCH --mem=100             # maximum 100M per job
#SBATCH --output=myprog%A%a.out   # standard output
#SBATCH --error=myprog%A%a.err   # standard error
# in the previous two lines %A" is replaced by jobID and "%a" with the array index
./myprogram input$SLURM_ARRAY_TASK_ID.dat
```

Single-node, muti-core job



- All threads are part of the same process, share single memory address space
- OpenMP is one of the easiest methods of parallel programming
- Always limited to a single node
- Does not have to occupy an entire node

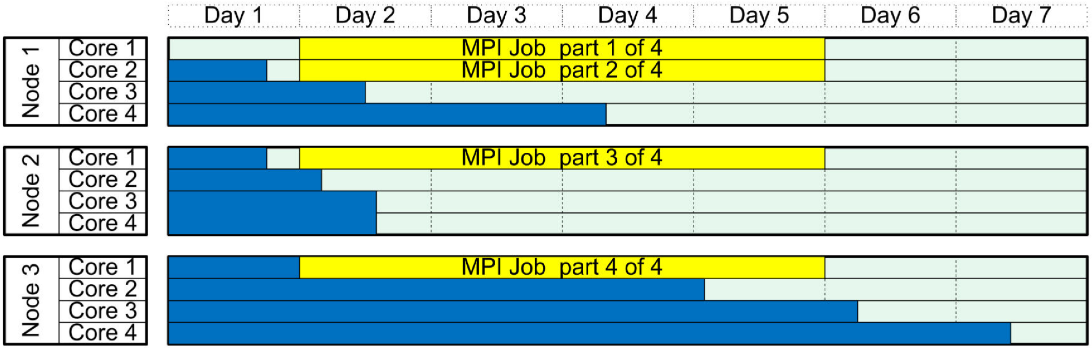
Scheduler: submitting OpenMP or threaded jobs

```
$ icc -qopenmp sharedPi.c -o openmp
$ sbatch job_openmp.sh [other flags]
$ squeue -u username [-t RUNNING] [-t PENDING] # list all current jobs
$ sacct -j jobID [--format=jobid,maxrss,elapsed] # list resources used by
                                                  # your completed job
```

```
#!/bin/bash
#SBATCH --cpus-per-task=2    # number of cores
#SBATCH --time=0-00:05       # walltime in d-hh:mm or hh:mm:ss format
#SBATCH --mem=100            # 100M for the whole job (all threads)
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK    # passed to the program
echo running on $SLURM_CPUS_PER_TASK cores
./openmp
```

- Did you get any speedup running this calculation on four cores?

MPI job



- Distributed memory: each process uses a different memory address space
- Communication via messages
- More difficult to write MPI-parallel code than OpenMP
- Can scale to much larger number of processors (clusters are larger than SMP machines)

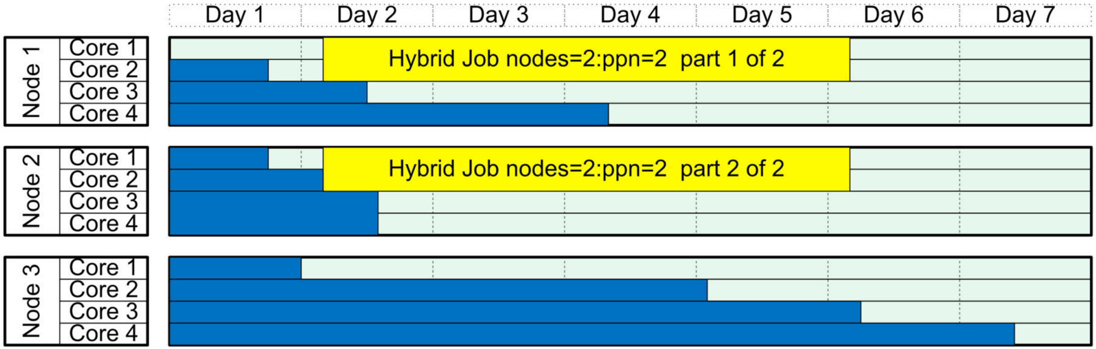
Scheduler: submitting MPI jobs

```
$ mpicc distributedPi.c -o mpi
$ sbatch job_mpi.sh [other flags]
$ squeue -u username [-t RUNNING] [-t PENDING] # list all current jobs
$ sacct -j jobID [--format=jobid,maxrss,elapsed] # list resources used by completed job
```

```
#!/bin/bash
#SBATCH --ntasks=4           # number of MPI processes
#SBATCH --time=0-00:05       # walltime in d-hh:mm or hh:mm:ss format
#SBATCH --mem-per-cpu=100 # in MB
mpirun -np $SLURM_NTASKS ./mpi                # or 'srun ./mpi'
```

- Did you get any speedup running this calculation on four processors?
- What is the code's *parallel efficiency*? Why is it not 100%?

Hybrid job



- Combining OpenMP and MPI for running on clusters of SMP machines
  - inside each node: shared-memory communication without MPI programming/overhead
  - scaling to larger resources (CPUs, memory) on the cluster
- In practice might be easier to code with MPI only
- New advanced languages such as Chapel combine shared- and distributed-memory models

Scheduler: submitting hybrid jobs

```
#!/bin/bash
#SBATCH --ntasks=4    # number of MPI tasks
#SBATCH --cpus-per-task=12    # number of cores per task
#SBATCH --time=12:00:00    # maximum walltime
if [ -n "$SLURM_CPUS_PER_TASK" ]; then
  omp_threads=$SLURM_CPUS_PER_TASK
else
  omp_threads=1
fi
export OMP_NUM_THREADS=$omp_threads
srun ./mpi_openmp_program
```

- More fine-grained control with `#SBATCH --ntasks-per-node=...`
- Do not run this script: only an example

Scheduler: submitting GPU jobs

```
#!/bin/bash
#SBATCH --nodes=3            # number of nodes
#SBATCH --gres=gpu:1         # GPUs per node
#SBATCH --mem=4000M          # memory per node
#SBATCH --time=0-05:00       # walltime in d-hh:mm or hh:mm:ss format
#SBATCH --output=%N-%j.out     # %N for node name, %j for jobID
srun ./gpu_program
```

- Do not run this script: only an example

Scheduler: interactive jobs

```
$ salloc --time=1:0:0 --ntasks=2   # submit a 2-core interactive job for 1h
$ echo $SLURM_...   # can check out Slurm environment variables
$ ./serial     # this would be a waste: we have allocated 2 cores
$ srun ./mpi   # run an MPI code, could also use mpirun/mpiexec
$ exit   # terminate the job
```

- Should automatically go to one of the Slurm interactive partitions

```
$ sinfo -a | grep interac
```

- Useful for debugging or for any interactive work, e.g., GUI visualization
    - interactive CPU-based ParaView client-server visualization on Cedar and Graham `https://docs.computecanada.ca/wiki/Visualization`
    - we use `salloc` in our hands-on Chapel course
- Make sure to only run the job on the processors assigned to your job – this will happen automatically if you use srun, but not if you just ssh from the headnode

Slurm environment variables

- Available inside running jobs
- You can start an interactive job, type `echo $SLURM` and then hit Tab to see all defined variables inside your job
- Useful to pass parameters to your program at runtime or to print out job information

Environment variable	Description
SLURM_JOB_ID	unique slurm jobID
SLURM_NNODES	number of nodes allocated to the job
SLURM_NTASKS	number of tasks allocated to the job
SLURM_NPROCS	number of tasks allocated to the job
SLURM_MEM_PER_CPU	memory allocated per CPU
SLURM_JOB_NODELIST	list of nodes on which resources are allocated
SLURM_JOB_CPUS_PER_NODE	number of CPUs allocated per node
SLURM_JOB_ACCOUNT	account under which this job is run

Exercise: simple interactive job

- Try to print some Slurm environment variables from an interactive job
- Run your "`pi.c` vs. `pi.py`" timing exercise as an interactive job

Exercise: Jupyter notebook server (more advanced interactive)

- Start a Jupyter notebook server on training VM's compute node
  - use Python virtual environment from `/home/centos/astro`
  - hopefully, not too many people run it: the cluster has only 6 nodes and 2 cores / 3GB per node
- Or you can do the same on Cedar or Graham
  - there you will have to roll out your own Python virtual environment

Slurm script flags

- Use them inside a job script (with `#SBATCH`) or in the command line (after `sbatch`)
- We already saw many examples in previous slides
- For full list of flags, type `man sbatch`

Slurm script command	Description
#SBATCH --ntasks=X	request X tasks; with cpus-per-task=1 (the default) this requests X cores
#SBATCH --nodes=X	request a minimum of X nodes
#SBATCH --nodes=X-Y	request a minimum of X nodes and a maximum of Y nodes
#SBATCH --cpus-per-task=X	request a minimum of X CPUs per task
#SBATCH --tasks-per-node=X	request a minimum of X tasks be allocated per node
#SBATCH --mail-type=ALL	notify via email about ALL, NONE, BEGIN, END, FAIL, REQUEUE
#SBATCH --mail-user=...	set email address

Slurm script flags (cont.)

Slurm script command	Description
#SBATCH --output=name%j.out	standard output and error log
#SBATCH --error=name.err	standard error log
#SBATCH --mem=2000	request 2000 MB of memory in total
#SBATCH --mem-per-cpu=2000	request 2000 MB of memory per CPU
#SBATCH --gres=gpu:1	request 1 GPU per node
#SBATCH --exclusive	request node(s) with no other running job(s)
#SBATCH --dependency=after:jobID1	request that the job starts after jobID1 has started
#SBATCH --dependency=afterok:jobID1	request that the job starts after jobID1 has finished successfully
#SBATCH --array=0-4	request job array of 5 jobs with indices 0-4
#SBATCH --array=0-4%2	array of 5 jobs with a maximum of 2 jobs running at the same time
#SBATCH --array=1,3,5,9,51	request job array of 5 jobs with indexes 1, 3, 5, 9, 51

Slurm jobs and memory

It is very important to specify memory correctly!

- If you don't ask for enough, and your job uses more, your job will be killed
- If you ask for too much, it will take a much longer time to schedule a job, and you will be wasting resources
- If you ask for more memory than is available on the cluster your job will never run; the scheduling system will not stop you from submitting such a job or even warn you
  - always ask for slightly less than total memory on a node as some memory is used for the OS, and your job will not start until enough memory is available
- Can use either `#SBATCH --mem=4000` or `#SBATCH --mem-per-cpu=2000`
- What's the best way to find your code's memory usage?

Slurm jobs and memory (cont.)

- Second best way: use Slurm command to estimate your completed code's memory usage

```
$ sacct -j jobID [--format=jobid,maxrss,elapsed]
                 # list resources used by a completed job
```

- Use the measured value with a bit of a cushion, maybe 15-20%
- Be aware of the discrete polling nature of Slurm's measurements
  - sampling at equal time intervals might not always catch spikes in memory usage
  - sometimes you'll see that your running process is killed by the Linux kernel (via kernel's *cgroups* `https://en.wikipedia.org/wiki/Cgroups`) since it has exceeded its memory limit but Slurm did not poll the process at the right time to see the spike in usage that caused the kernel to kill the process, and reports lower memory usage
  - sometimes `sacct` output in the memory field will be empty, as Slurm has not had time to poll the job (job ran too fast)

Getting information and other Slurm commands

```
$ squeue -u username [-t RUNNING] [-t PENDING]         # list all current jobs
$ squeue -p partitionName                              # list all jobs in a partition
$ squeue -P --sort=-p,i --states=PD --format="%.10A␣%.18a␣%.12P␣%.8C␣%.12m␣%.15l␣%.25S"
                        # show all queued (=PD) jobs sorted by their current priority
$ scancel [-t PENDING] [-u username] [jobID]           # kill/cancel jobs

$ sinfo                                  # view information about Slurm partitions
$ partition-stats                        # similar in a tabular format
$ scontrol show partition                # similar with more details
$ sinfo --states=idle                    # show idle node(s) on cluster
$ sinfo -n gra10 -o "%n␣%c␣%m"           # list node's name, core count and memory

$ sacct -j jobID --format=jobid,maxrss,elapsed   # resources used by a completed job
$ sacct -u username --format=jobid,jobname,avecpu,maxrss,maxvmsize,elapsed
                                               # show details of all your jobs
$ sacct -aX -S 2018-04-25 --format=account%20,partition%20,jobid%12,submit%22,\
  start%22,end%15,timelimit%15,reqmem,ncpus%8,nnodes%8,state | more
              # show all jobs (all users) on the system since a specific date

$ scontrol show job jobID      # produce a very detailed report for the job
$ sprio [-j jobID1,jobID2] [-u username]     # list job priority information
```

Common job states: R = running, PD = pending, CG = completing right now, F = failed

Slurm partitions

- Idea: restrict jobs of specific shapes to node sets
  - obvious for large-memory / GPU / interactive jobs
- By-node vs. by-core
  - *full-node* (or *by-node*) jobs can run on most nodes
  - *partial-node* (or *by-core*) jobs have access to fewer nodes
    - by-core jobs can fit into little, sparse places $\Rightarrow$ too many of them will easily fill the cluster, leaving no resources for "denser" jobs $\Rightarrow$ hence separate cluster partitions
- Short vs. long jobs
  - jobs 3 hours and shorter can run on most nodes
  - longer jobs have access to fewer nodes
- Backfill jobs can run on most nodes

Why is my job *not* running?

In no particular order:

1. Other jobs have greater priority
2. There is a problem with the job / resources are not available
   - resources do not exist on the cluster?
   - did not allow for OS memory (whole-node runs)?
3. The job is blocked
   - disk quota / other policy violation?
   - dependency not met?
4. There is a problem with the scheduling system or cluster
   - most frequent: someone just submitted 10,000 jobs via a script, then cancelled them, then resubmitted them, rendering Slurm unresponsive for a while, even if resources are available for your job right now

Debugging and profiling

Identifying bugs and errors in your code

- Methodical process of finding and fixing flaws in software
- Typical signs that your program is buggy include:
  - it fails to complete (crashes), usually with an error message in the output file ("Segmentation fault", "Floating point exception", etc) or with a numeric *job exit code*
  - it produces incorrect output (NaNs)
  - it fails to progress (hangs), often showing ∼ 100% CPU usage

signal name	OS signal #	OS signal name	description
floating point exception	8	SIGFPE	the program attempted an arithmetic operation with values that do not make sense (e.g., divide by zero)
segmentation fault	11	SIGSEGV	the program accessed memory incorrectly (e.g., accessing an array beyond its declared bounds, using incorrect pointers
aborted	6	SIGABRT	generated by the runtime library of the program or a library it uses, after having detected a failure condition

Common bugs

- Arithmetic: infinities (division by zero), out of range
- Memory access: index out of range, uninitialized pointers
- Logic: infinite loop, corner cases (sloppy condition evaluation)
  - example: try evaluating `0.1 + 0.2 == 0.3` in Python or R
- Misuse: wrong initial conditions, variable initialization (forgot to set to zero?), implicit variable declarations ($\Rightarrow$ wrong type)
- Syntax: wrong operator, function arguments (variable number/types must match)
- Resource starvation: memory leak
- Parallel: race conditions, deadlock, nonmatching send/receive

Debugging

- Debugger is a program to manipulate and inspect your program as it is running
  - not a magic bullet – you are the real debugger!
- Write better code, use existing libraries instead of your own code
- Test individual parts of your code
- Once you are convinced there is a problem
  - try to reproduce the problem in the simplest situation possible
  - try to reverse your steps to a working state (version control!) and then make one change at a time
  - try smaller problem size
  - use compiler flags to turn off floating point exception masking (the code will stop)
  - turn on compiler warnings (GNU: `-Wall`)
  - (mostly Fortran) use compiler flags to enable runtime checking for various conditions (array indices within bounds, uninitialized variables, proper pointer usage)
  - ensure that variables are defined with sufficient precision (overflow/underflow)
  - use a debugger
  - use print statements? ... not a good strategy

Debuggers for compiled languages

- Command-line serial debugger *gdb* – standard on Linux systems
- Commercial parallel GUI debuggers: DDT, TotalView
  - we have DDT on Graham (installed under `\cvmfs` on Cedar but no license ...)
- Prepare your code for debugging
  - compile your program with "-g" flag to include a symbol table, if you are going to run it in a debugger
  - disable all processor optimizations — these might produce misleading debugger behaviour
  - turn off floating point masking behaviour for the program to stop when a NaN or an Inf is computed
    - Intel C/C++ compiler: need explicit signal handling in the code

Buggy code example

bugs1.c

```c
#include <stdio.h>
void divide(float e, float d) {
  printf("%f\n", e/d);
}
void array(float f[], int index) {
  printf("%f\n", f[index]);
}
int main(int argc, char **argv) {
  float a = 0., b = 1., c[10];
  int i;
  for (i = 0; i < 10; i++)
    c[i] = (float)i;
  divide(b, a);
  array(c, 1200);
  return(0);
}
```

bugs2.c (same with floating point exception handling)

```c
#include <fenv.h>
#include <signal.h>
#include <stdio.h>
void divide(float e, float d) {
  printf("%f\n", e/d);
}
void array(float f[], int index) {
  printf("%f\n", f[index]);
}
void fpehandler(int sig_num) {
  signal(SIGFPE, fpehandler);
  printf("floating point exception, exiting\n");
  abort();
}
int main(int argc, char **argv) {
  int feenableexcept();
  feenableexcept(FE_ALL_EXCEPT);
  signal(SIGFPE, fpehandler);
  float a = 0., b = 1., c[10];
  int i;
  for (i = 0; i < 10; i++)
    c[i] = (float)i;
  divide(b, a);
  array(c, 1200);
  return(0);
}
```

Typical gdb session

```
$ icc bugs1.c -o bugs
$ ./bugs
inf
70310426758547368093419569152.000000

$ icc bugs2.c -o bugs
$ ./bugs
floating point exception, exiting
Aborted

$ icc -g bugs2.c -o bugs
$ gdb bugs
(gdb) r
Starting program: bugs
Program received signal SIGFPE, Arithmetic exception.
0x00000000004006a5 in divide (e=1, d=0) at bugs2.c:5
5          printf("%f\n",e/d);

(gdb) where
#0  0x00000000004006a5 in divide (e=1, d=0) at bugs2.c:5
#1  0x00000000004007d0 in main (argc=1,
          argv=0x7fffffffc978) at bugs2.c:23
```

```
(gdb) l 5
1       #include <fenv.h>
2       #include <signal.h>
3       #include <stdio.h>
4       void divide(float e, float d) {
5         printf("%f\n",e/d);
6       }
7       void array(float f[], int index) {
8         printf("%f\n",f[index]);
9       }
10      void fpehandler(int sig_num) {

(gdb) p e
$1 = 1

(gdb) p d
$2 = 0
```

Continue debugging

- Resolve the bug, and then repeat
- The next bug (index=1200) might or might not produce a segmentation fault (that memory address is used by something else inside the code at runtime?)
  - if it does, gdb will track it

```
(gdb) r
Starting program: /home/razoumov/introToHPC/scripts/bugs
Program received signal SIGSEGV, Segmentation fault.
0x00000000004005a1 in array (f=0x7fffffffc84c, index=1200000) at bugs1.c:6
6          printf("%f\n", f[index]);
```

  - if it does not, need a more powerful memory debugger (*valgrind*)
- Other gdb commands:
  - insert breakpoints at certain lines or at the beginning of certain functions, then run until the next breakpoint
  - commands to show and delete breakpoints
  - run until the breakpoint
  - step one line at a time, step into a function or out of a function
  - run `help` and `help className` to get more info

Using core files

- Often it takes a long time before the program reaches the error condition $\Rightarrow$ cannot debug interactively
- In this case submit the job (debugging-instrumented "-g" executable) to the scheduler, tell the OS to produce a **core file** when it crashes
  - need to set up the Linux environment to produce core files (set core limit to be non-zero):
    `ulimit -c unlimited` $\leftarrow$ **do this on compute node as part of your running job!** (put it into your job script)
- A core file contains the state of the program at the time it crashed $\Rightarrow$ can load this file into the debugger to inspect the state and determine what caused the problem
- Once a core is produced, load it into gdb

```
$ gdb ./programName core.jobID
```

Debugging summary

- Prepare the code:
  - compile with “-g”
  - turn off optimization
  - turn off floating point masking behaviour for the program to stop when a NaN or an Inf is computed (Intel C/C++ compiler: need explicit signal handling in the code)
- gdb for *serial debugging*: will show the function and line in which the error occurred, the reason behind the error, can print variables, step outside the function
  - interactive debugging: `gdb ./programName`
  - post-processing core files: `gdb ./programName core.jobID`
- GUI and/or parallel debugging: use a commercial debugger (DDT, TotalView)
  - for MPI or threaded codes; can debug individual threads or processes
  - again, use “-g” to include a symbol table
  - parallel bugs: race conditions, deadlock, nonmatching send/receive

Code profiling

- Profiling tools perform analysis of actual program execution providing very fine-grained information regarding program operation
  - number of times a function is called
  - amount of time spent in each function $\Rightarrow$ identify functions that use most CPU time and try to optimize them
- Tools for code profiling
  1. inline timing
     - C: `time` (seconds only), `gettimeofday` (seconds and microseconds separately)
     - Fortran 90: `date_and_time` (ms granularity)
     - MPI library: `MPI_Wtime` (use `MPI_Wtick` to check accuracy, usually μs)
  2. *gprof* (GNU command line **serial** code profiler)
  3. commercial profilers for parallel MPI codes, e.g., *OPT*
  4. *IPM*, open-source profiler for parallel MPI codes, and other tools from HPC centers
  5. Compilers often come with basic profilers too

Code profiling (cont.)

Using a profiler typically involves three steps:

(1) instrumenting the source code to collect data: extra compile flags and/or linking to special libraries

(2) running the instrumented binary

(3) performing analysis of the collected data after program execution using special GUI tools

Best practices

Best practices: computing

- Production runs: only on compute nodes via the scheduler
  - do not run anything intensive on login nodes or directly on compute nodes
- Only request resources (memory, running time) needed
  - with a bit of a cushion, maybe 115-120% of the measured values
  - use Slurm command to estimate your completed code's memory usage
- Test before scaling, especially parallel scaling
  - small tests Ok on login nodes
  - larger tests on "interactive" nodes (requested with `salloc`)
  - test parallel scaling (frequently big surprises!)
- For faster turnaround, if applicable, request whole nodes (`--nodes=...`) and short runtimes (`--time=...`)
  - by-node jobs can run on the "entire cluster" partitions, as opposed to smaller partitions for longer and "by-core" jobs
- Do not run unoptimized codes (use compilation flags -O2 or -O3 if needed)
- Be smart in your programming language choice, use precompiled libraries

Best practices: file systems

Filesystems in CC are a shared resource and should be used responsibly!

- Do not store millions of small files
  - organize your code's output
  - use **tar** or even better **dar** (`https://docs.computecanada.ca/wiki/Dar`, supports indexing, per-file compression, differential/incremental backups, encryption, more resilient to data corruption)
- Do not store large data as ASCII (anything bigger than a few MB): waste of disk space and bandwidth
  - use a binary format
  - use scientific data formats (NetCDF, HDF5, etc.): portability, headers, binary, compression, parallel
  - compress your files
- Use the right filesystem
- Learn and use parallel I/O
- If searching inside a file, might want to read it first
- Have a backup plan
- Regularly clean up your data in `$SCRATCH`, `$PROJECT`, possibly archive elsewhere

Summary

Documentation and getting help

- Official documentation `https://docs.computecanada.ca/wiki`
- WestGrid training materials
  `https://westgrid.github.io/trainingMaterials`
- Compute Canada *Getting started* videos `http://bit.ly/2sxGO33`
- Compute Canada YouTube channel `http://bit.ly/2ws0JDC`
- Email **support@computecanada.ca** (goes into the ticketing system)
  - try to include your full name, CC username, institution, the cluster name, copy and paste as much detail as you can (error messages, jobID, job script, software version)
- Email me **alex.razoumov@westgrid.ca** for questions on any of today's topics
- Please get to know your local support
  - difficult problems are best dealt face-to-face
  - might be best for new users