Using High-Performance, Advanced, and Cloud Computing Resources for Research

Paul Preney, OCT, M.Sc., B.Ed., B.Sc.
preney@sharcnet.ca / preney@uwindsor.ca

School of Computer Science / Office of Research and Innovation Services (ORIS)
University of Windsor, Windsor, Ontario, Canada

Sept. 29, 2023

Table of Contents

What is SHARCNET?

SHARCNET:

- **S**hared **H**ierarchical **A**cademic **R**esearch **C**omputing **Net**work.
- provides free access to high performance, advanced, and cloud computing and corresponding **compute, cloud, and storage** resources and services to researchers
- weekly new user seminars, biweekly webinars, summer school courses, and other training activities, e.g., see `https://training.sharcnet.ca`

What is SHARCNET? (con’t)

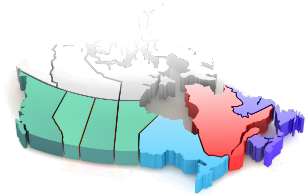
- URL: `https://www.sharcnet.ca/`
- Started in 2001.
- Founding members:
  - *Fanshawe* College, University of *Guelph*, *McMaster* University, *Sheridan* College, *Western* University, *Wilfred Laurier* University, University of *Windsor*
- Today’s members include:
  - … Brock, Conestoga, Durham, Lakehead, Laurentian, Nipissing, Ontario College of Art and Design University, Ontario Tech University, Perimeter Institute, Trent, Waterloo, York
- Part of *Compute Ontario* which is part of the *Digital Research Alliance of Canada*.

What is Compute Ontario?

Compute Ontario, `https://computeontario.ca`:

- Plays a key role in coordinating Ontario's advanced research computing and big data focus
- Has these four partners:
  - SHARCNET
  - SciNet
  - Centre for Advanced Computing
  - HPC4Health

What is the Digital Research Alliance of Canada?



The Alliance, `https://www.alliancecan.ca` has these regional partners:

- ACENET (New Brunswick, Nova Scotia, P.E.I., Newfoundland and Labrador)
- Calcul Québec (Québec)
- Compute Ontario (Ontario)
- Prairies DRI (Alberta, Saskatchewan, Manitoba)
- BC DRI (British Columbia)

Some Important Details

- Cost:
  - Access is **free** to researcher faculty at Canadian academic research institutions.
  - We are funded directly and indirectly through various federal and provincial grants.
- Limited to **research**.
  - e.g., undergraduate course work is not permitted
- Faculty members (PIs) **sponsor** some number of users.
- Jobs/activities are associated with a **sponsor PI** who supervises that research.
- Users can have more than one **sponsor PI**.
- All accounts must be renewed **every year** to remain active.
- New accounts are not made later, to **re-activate an inactive account log in to the CCDB web site** and apply for a new role: `https://ccdb.alliancecan.ca/`.

Acquiring an Account

To obtain access, apply for an account:

- `https://alliancecan.ca/en/services/advanced-research-computing/account-management/apply-account`
- Or equivalently, go to `https://docs.alliancecan.ca` and click on **Getting an Account** in the left-hand side menu.

Acquiring an Account (con’t)

If you are a student / doing work under a PI:

- It is important that your supervisor **first** acquires/activates their account **before** you can apply for your own account.
- You will **need to know** your supervisor’s **CCRI** number (which PIs can get by logging in to the CCDB) before you can apply for your own account.

Table of Contents

What a Server Room Looks Like

Early picture of Graham server room (at UWaterloo) under construction:

Hardware Resources

General-Purpose Clusters:

- **beluga**.alliancecan.ca
  - 28 PB disk; 974 nodes: 196 TB RAM, 38,960 CPUs, 688 GPUs
    - Intel Xeon Sky Lake
    - NVIDIA V100SXM2 (16 GB RAM)
- **cedar**.alliancecan.ca
  - 29 PB disk; 2,470 nodes: 478 TB RAM, 100,400 CPUs, 1,352 GPUs
    - Intel Xeon: Broadwell, Sky Lake; Cascade Lake
    - NVIDIA P100 (12GB & 16GB HBM2 RAM), V100 (32G HBM2 RAM)
- **graham**.alliancecan.ca
  - 20 PB disk; 1,327 nodes: 202 TB RAM, 44,444 CPUs, 534 GPUs
    - Intel Xeon: Broadwell, Sky Lake, Cascade Lake
    - NVIDIA P100 (12GB HBM2 RAM), V100 (16GB HBM2 RAM), T4 (16GB GDDR6 RAM)
- **narval**.alliancecan.ca
  - 25 PB disk; 1,340 nodes: 443 TB RAM, 83,216 CPUs, 636 GPUs
    - AMD Rome and Milan
    - NVIDIA A100 (40 GB RAM)

Hardware Resources(con't)

Large-Parallel Job Cluster:

- **niagara**.alliancecan.ca
  - 37 PB disk; 1,728 nodes: 349 TB RAM, 69,120 CPUs
    - Intel Xeon Sky Lake
  - Scheduling by node only.

Cloud Systems (details omitted):

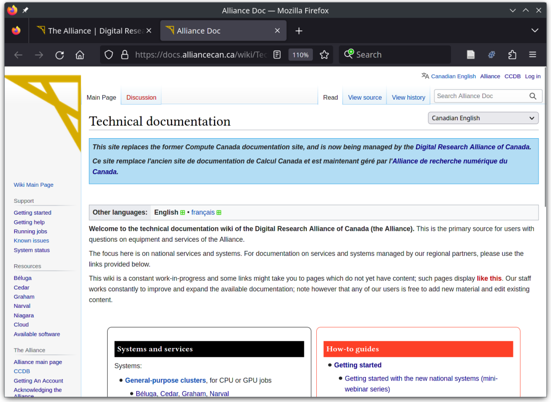
- **arbutus.cloud**.alliancecan.ca
- **east.cloud**.alliancecan.ca
- **graham.cloud**.alliancecan.ca
- **cedar.cloud**.alliancecan.ca

Storage space:

- **`/home`**: 50 GB per user, backed up
- **`/project`**:
  - 1 TB per PI group
  - up to 40 TB by request
  - backed up
- **`/scratch`**:
  - 20 TB per user
  - up to 200 TB by request
  - old files removed after 60 days
- **`/nearline`**: (tape)

Other Resources

A wiki with lots of useful technical documentation (https://docs.alliancecan.ca/):



Technical documentation

This site replaces the former Compute Canada documentation site, and is now being managed by the Digital Research Alliance of Canada.

Ce site remplace l'ancien site de documentation de Calcul Canada et est maintenant géré par l'Alliance de recherche numérique du Canada.

Wiki Main Page

Support
- Getting started
- Getting help
- Running jobs
- Known issues
- System status

Resources
- Béluga
- Cedar
- Graham
- Narval
- Niagara
- Cloud
- Available software

The Alliance
- Alliance main page
- CCDB
- Getting An Account
- Acknowledging the Alliance

Other languages: **English** • français

Welcome to the technical documentation wiki of the Digital Research Alliance of Canada (the Alliance). This is the primary source for users with questions on equipment and services of the Alliance.

The focus here is on national services and systems. For documentation on services and systems managed by our regional partners, please use the links provided below.

This wiki is a constant work-in-progress and some links might take you to pages which do not yet have content; such pages display like this. Our staff works constantly to improve and expand the available documentation; note however that any of our users is free to add new material and edit existing content.

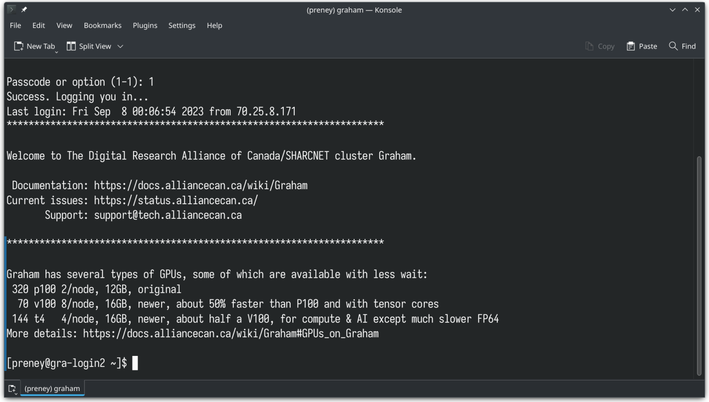
Systems and services

Systems:
- **General-purpose clusters**, for CPU or GPU jobs
  - Béluga, Cedar, Graham, Narval

How-to guides
- **Getting started**
  - Getting started with the new national systems (mini-webinar series)

Other Resources (con't)

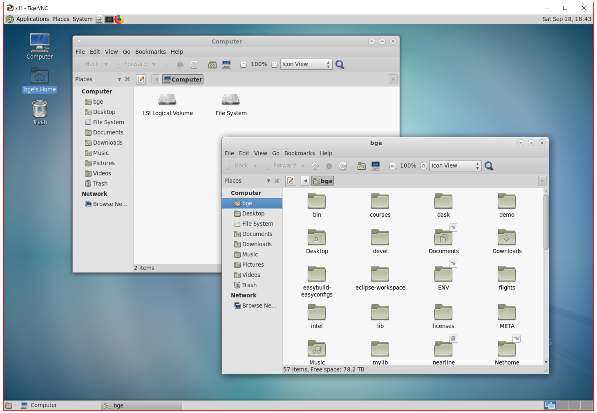
Compute cluster login node access via secure shell (SSH):


(preney) graham — Konsole
File Edit View Bookmarks Plugins Settings Help
New Tab Split View
Copy Paste Find
(preney) graham

```
Passcode or option (1-1): 1
Success. Logging you in...
Last login: Fri Sep  8 00:06:54 2023 from 70.25.8.171
**********************************************************************

Welcome to The Digital Research Alliance of Canada/SHARCNET cluster Graham.

 Documentation: https://docs.alliancecan.ca/wiki/Graham
Current issues: https://status.alliancecan.ca/
       Support: support@tech.alliancecan.ca

**********************************************************************

Graham has several types of GPUs, some of which are available with less wait:
 320 p100 2/node, 12GB, original
  70 v100 8/node, 16GB, newer, about 50% faster than P100 and with tensor cores
 144 t4   4/node, 16GB, newer, about half a V100, for compute & AI except much slower FP64
More details: https://docs.alliancecan.ca/wiki/Graham#GPUs_on_Graham

[preney@gra-login2 ~]$ 
```

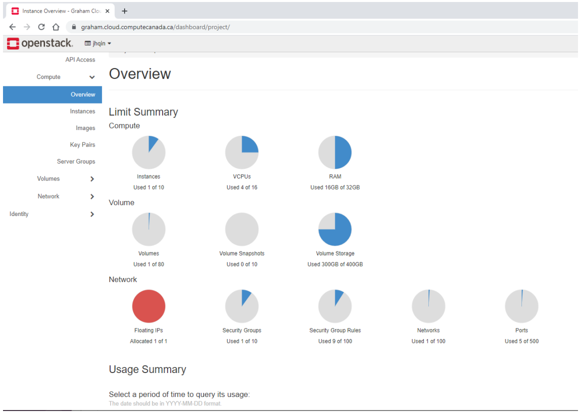
Other Resources (con't)

3D-accelerated Graham-cluster-only graphical "gra-vdi" login nodes (using TigerVNC; `https://docs.alliancecan.ca/wiki/VNC`):



Other Resources (con’t)

Cloud node management access portals using OpenStack (`https://docs.alliancecan.ca/wiki/Cloud_Quick_Start`):



Overview

Limit Summary

Compute

Instances	VCPUs	RAM
Used 1 of 10	Used 4 of 16	Used 16GB of 32GB

Volume

Volumes	Volume Snapshots	Volume Storage
Used 1 of 80	Used 0 of 10	Used 300GB of 400GB

Network

Floating IPs	Security Groups	Security Group Rules	Networks	Ports
Allocated 1 of 1	Used 1 of 10	Used 9 of 100	Used 1 of 100	Used 5 of 500

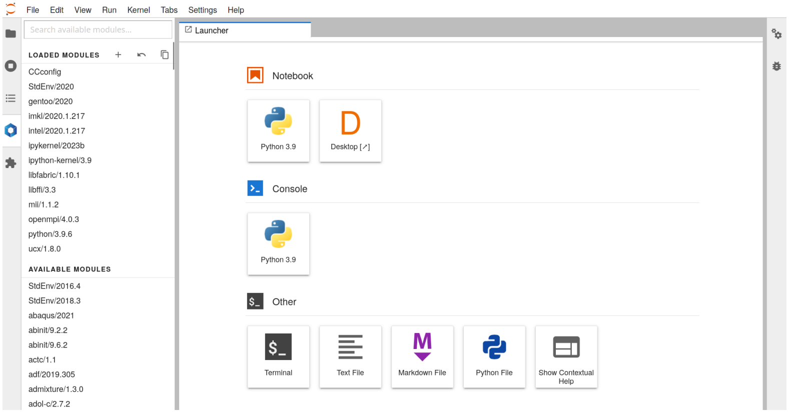
Usage Summary

Select a period of time to query its usage:

The date should be in YYYY-MM-DD format.

Other Resources (con't)

JupyterHub Web Portals (`https://docs.alliancecan.ca/wiki/JupyterHub`):



LOADED MODULES

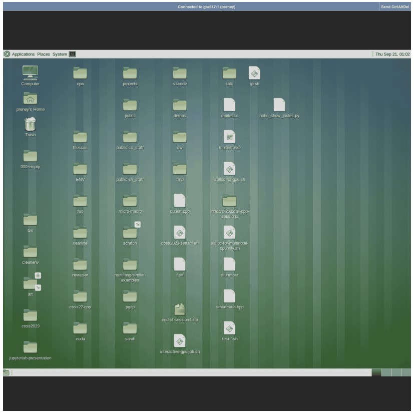
CCconfig
StdEnv/2020
gentoo/2020
imkl/2020.1.217
intel/2020.1.217
ipykernel/2023b
ipython-kernel/3.9
libfabric/1.10.1
libffi/3.3
mii/1.1.2
openmpi/4.0.3
python/3.9.6
ucx/1.8.0

AVAILABLE MODULES

StdEnv/2016.4
StdEnv/2018.3
abaqus/2021
abinit/9.2.2
abinit/9.6.2
actc/1.1
adf/2019.305
admixture/1.3.0
adol-c/2.7.2

Other Resources (con't)

JupyterHub Web Portals also support graphical desktops:



NOTE: Use gra-vdi if interactive 3D hardware acceleration is needed.

Other Resources (con't)

Online training materials/courses:

- SHARCNET: `https://training.sharcnet.ca`
- Compute Ontario Summer Schools: `https://training.computeontario.ca`

Weekly (online) webinars and their recordings:

- `https://www.computeontario.ca/training-colloquia`

Weekly (online) new user / refresher webinar:

- Tuesdays 2pm to 3pm: `https://training.sharcnet.ca/courses/course/view.php?id=34`

Other Resources (con't)

Support through (email) tickets, i.e., support@tech.alliancecan.ca

Support from expert staff via:

- ticketing system
- video conferencing
- direct email (ticketing system is preferred)
- in-person (if at same institution)
- telephone (if needed)

Table of Contents

Cluster Computing Environment

- OS: 64-bit Linux
- Supported programming languages include:
  - C, C++, Fortran, Java, Julia, MATLAB, Octave, Python, R, etc.
- A large variety of open source (and some commercial) software packages.
- Parallel development, e.g.,
  - **C**: multithreading since 2011 standard
  - **C++**: multithreading since 2011 standard
  - **Fortran**: since 2003 standard
  - **Julia**: shared and distributed memory
  - **MPI**, **Chapel**: shared and distributed memory systems
  - **OpenMP**, **pthreads**: shared memory (single node)
  - **CUDA**, **OpenACC**, **OpenCL**: GPUs
- Data science support:
  - DASK, Julia, Jupyter, Python, R, etc.

Cluster Computing Environment (con’t)

- Container technology: Apptainer (Docker cannot be used.)
- Compiler toolchains:
    - GNU Compiler Collection (GCC)
    - Clang/LLVM
    - Intel OneAPI (and legacy)
    - NVIDIA nvhpc

+ one can install additional softwares in one’s own account.

Table of Contents

Clusters Are Different Than Your Computer

On your own computer:

- You typically run programs/jobs **on-demand** since you are the **only** user.
- Programs/jobs typically have access to all resources (CPUs, GPUs, disk space, etc.).

Clusters Are Different Than Your Computer (con’t)

On clusters:

- There are many users able to submit jobs to run at any time.
  - Unlike your personal computer where there is one user.
- Each submitted job needs to provide:
  - how much **RAM (at most)** the job requires,
  - how many **CPUs** are required,
  - if needed, how many **GPUs** are required,
  - how much **time (at most)** the job will run for, and,
  - which PI **account** the job is to be run under.
- The scheduler then determines **when** the submitted job can be run **with its requirements** in a **fair manner**.
- Jobs run **non-interactively** with no access to a keyboard, mouse, screen, etc.
- Job **input and output** needs to be from/to **files**.

Development and Testing

- Generally each cluster has a **login node** and a set of **compute nodes**.
- Program development, debugging, and testing can be done:
  - within an interactive scheduler (Slurm) job,
  - on login node *if and only if* such does not significantly use resources (time, RAM, or CPUs), or,
  - using your own computer.
- Jobs are typically submitted from a login node.
- Login node access is via SSH.

Maximizing Job Throughput

All jobs are submitted to a queue to run:

- Your research team ideally wants to always have jobs **in the queue** waiting to be run to maximize throughput.
- Every individual and every **team** has fair-share priority.

Supported Software and Systems

Available software on our clusters:

- `https://docs.alliancecan.ca/wiki/Available_software`

Upon request, we work with researchers to help install and use software on our systems.

Table of Contents

Why Use Supercomputing Resources?

- You do not have sufficient CPU/GPU cores.
- You do not have sufficient memory (RAM or storage).
- You need a lot of disk space, e.g., hundreds of TBs.
- You need to run a lot of simulations needing hundreds of cores.
- You need to run a large number of simulations in parallel.
- You need to run web services.
- You need to run an SQL database to service compute jobs / cloud resources.
- You need to make use of cloud resources.

Table of Contents

2024 Resource Allocation Competition (RAC)

Each year there is a Resource Allocation Competition (RAC):

- Peer-reviewed competition
- For research projects that need resources significantly beyond what is normally available.

2024 Resource Allocation Competition (RAC) (con’t)

There are two competitions:

- **Research Platforms and Portals (RPP)** which primarily involves using cloud(s).
- **Resources for Research Groups (RRG)** which primarily involves using compute cluster(s).

2024 Resource Allocation Competition (RAC) (con't)

Key Dates for RAC:

- RRG & RPP applications: Sept. 26 to Nov. 7, 2023
- General information session: Sept. 28, 2023 (English), Sept. 29, 2023 (French)
- GPU, etc. information session: Oct. 3, 2023 (English), Oct. 4, 2023 (French)
- Results announcement: Late March 2024
- Start of RAC allocations: Early April 2024

For detailed information see:

- `https://alliancecan.ca/en/services/advanced-research-computing/accessing-resources/resource-allocation-competition/resource-allocation-competition-application-guide`
- `https://alliancecan.ca/en/services/advanced-research-computing/accessing-resources/resource-allocation-competition`

2024 Resource Allocation Competition (RAC) (con't)

Each 2024 RRG (compute) RAC application must meet at least one of these minimum requirements:

- CPU core years or equivalent: 200
- GPU core years or equivalent: 25
- Storage in TB: 41
- Nearline storage (tape) in TB: 101
- A submission is required if any dCache storage is needed.

2024 Resource Allocation Competition (RAC) (con't)

Each 2024 RPP (cloud) RAC application must meet at least one of these minimum requirements:

- Virtual CPU years (VCPU): 81
- Virtual GPU years (VGPU): 1.3
- Persistent virtual CPU years (VCPU): 26
- Volume and snapshot storage in TB: 11
- Shared filesystem storage in TB: 11
- Object storage in TB: 11

2024 Resource Allocation Competition (RAC) (con't)

All information, links to information sessions, etc. can be found at: `https://alliancecan.ca/en/services/advanced-research-computing/accessing-resources/resource-allocation-competition` and in the "RAC Competition Guide".

There is a technical glossary of terms for RACs: `https://docs.alliancecan.ca/wiki/Technical_glossary_for_the_resource_allocation_competitions`.

One can request staff interaction for RAC applications by:

- send an email to support@tech.alliancecan.ca requesting such to open a ticket.

Table of Contents

Questions, Answers, and Discussion

Thank you!

Questions, answers, and discussion.

Table of Contents

Appendix Overview

These appendices are extra slides provided should such need referring to in the presentation and/or should one be viewing these slides at a later date.

Table of Contents

Software/Environment Modules on Clusters

- A lot of software programs as well as many versions of those softwares are available on clusters.
- These can be utilized by using the `module` command.

Software/Environment Modules on Clusters (con’t)

Some examples:

- **`module reset`**: reset loaded modules back to defaults upon login
- **`module avail`**: shows names of software packages that are available with the currently loaded modules
- **`module spider PKGNAME`**: shows information about PKGNAME and its versions
- **`module spider PKGNAME/VERSION`**: shows information about version VERSION of PKGNAME and how it must be loaded
- **`module load PKGNAME`**: loads the default version of PKGNAME
- **`module load PKGNAME/VERSION`**: loads version VERSION of PKGNAME
- **`module unload PKGNAME`**: unloads PKGNAME

Table of Contents

Slurm Scheduler Use

The Slurm scheduler allows one to control and/or query information about non-interactive and interactive jobs with these commands:

- **`sbatch`**: submits a job into the scheduler's job queue
- **`salloc`**: requests an interactive job (max: 3h)
- **`scancel`**: cancels a job
- **`squeue`**: see jobs currently in job queue on cluster
- **`sacct`**: see history of jobs run

sbatch

The **sbatch** command is typically invoked like this:

- `sbatch --account ACCTNAME jobscript.sh`

where:

- `ACCTNAME` is the name of the PI account the job will run under
- `jobscript.sh` is the name of the job script to run

sbatch Sequential Examples

An example `jobscript.sh` to run an instance of a sequential program:

sequential.sh

```
#!/bin/bash
#SBATCH --time=0-05:00          # D-HH::MM, i.e., 5 hours (max)
#SBATCH --mem=4000M             # i.e., ~4 GB RAM

./sequential-program.exe
```

sbatch Sequential Examples (con't)

An example `jobscript.sh` to run 100 instances of a sequential program:

sequential-array.sh

```
#!/bin/bash
#SBATCH --time=0-05:00          # D-HH::MM, i.e., 5 hours (max)
#SBATCH --mem=4000M             # i.e., ~4 GB RAM
#SBATCH --array=1-100

# Pass $SLURM_ARRAY_TASK_ID to program so it knows which one it is...
./sequential-program.exe $SLURM_ARRAY_TASK_ID
```

sbatch Sequential Examples (con't)

An example `jobscript.sh` to run a single CPU, single GPU program:

gpu-seq.sh

```
#!/bin/bash
#SBATCH --time=0-11:00
#SBATCH --mem=4000M
#SBATCH --gpus-per-node=1

./gpu-program.exe
```

sbatch Multithreaded Examples

An example `jobscript.sh` to run a multithreaded program using 10 threads (limited to a single node):

mt.sh

```
#!/bin/bash
#SBATCH --time=0-05:00          # D-HH::MM, i.e., 5 hours (max)
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=10
#SBATCH --mem-per-cpu=1024M     # i.e., ~1 GB RAM per CPU

./multithreaded-program.exe
```

sbatch Multithreaded Examples (con’t)

An example `jobscript.sh` to run an OpenMP program using 10 threads (limited to a single node):

openmp.sh

```
#!/bin/bash
#SBATCH --time=0-05:00          # D-HH::MM, i.e., 5 hours (max)
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=10
#SBATCH --mem-per-cpu=1024M     # i.e., ~1 GB RAM per CPU

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
./openmp-program.exe
```

sbatch Multithreaded Examples (con't)

An example `jobscript.sh` to run a six CPU, single GPU OpenMP program:

gpu-openmp.sh

```
#!/bin/bash
#SBATCH --time=0-11:00
#SBATCH --mem=4000M
#SBATCH --gpus-per-node=1
#SBATCH --cpus-per-task=6
#SBATCH --mem=4000M           # i.e., total memory needed

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
./gpu-program.exe
```

sbatch MPI Examples

An example jobscript.sh to run an MPI program using 8 MPI processes:

mpi.sh

```bash
#!/bin/bash
#SBATCH --time=0-05:00          # D-HH::MM, i.e., 5 hours (max)
#SBATCH --ntasks=8
#SBATCH --mem-per-cpu=1024M     # i.e., ~1 GB RAM per CPU

srun ./mpi-program.exe
```

sbatch MPI Examples (con't)

An example `jobscript.sh` to run an MPI program using 8 GPUs with 6 CPUs per GPU:

gpu-mpi.sh

```bash
#!/bin/bash
#SBATCH --time=0-11:00
#SBATCH --gpus=8
#SBATCH --ntasks-per-gpu=1
#SBATCH --cpus-per-task=6
#SBATCH --mem-per-cpu=5G

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
srun --cpus-per-task=$SLURM_CPUS_PER_TASK ./mpi-gpu-program.exe
```

sbatch Whole Node Example

An example `jobscript.sh` to run a program exclusively on an entire node:

whole-node.sh

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --gpus-per-node=p100:2  # i.e., 2 NVIDIA Pascal GPUs
#SBATCH --ntasks-per-node=32    # i.e., node needs at least 32 CPUs
#SBATCH --mem=0                 # i.e., use all memory on node
#SBATCH --time=0:03:00

./whole-node-gpu-program.exe
```

salloc

The **salloc** command is invoked by passing on a single command line all of the SBATCH options needed for the job, e.g.,

- `salloc --account ACCTNAME --time=0-03:00 --cpus-per-task=10 --mem-per-cpu=1024M`

After a wait for an available machine with those resources, you will be logged in to such.

scancel

The **`scancel`** command is invoked by passing the job number one wishes to cancel, e.g.,

- `scancel 32385923`

Table of Contents

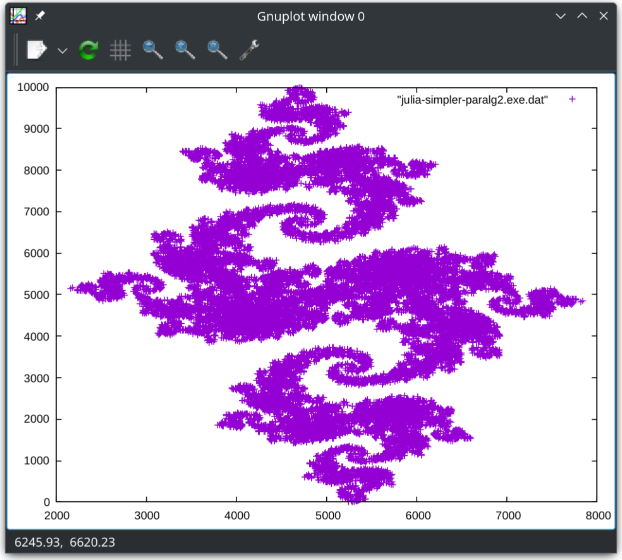
Julia Set Fractal Example

Let's consider a program that computes the Julia Set, `https://en.wikipedia.org/wiki/Julia_set`, fractal.

To determine whether or not a particular complex value is in the Julia set, one can map a 2D point, (x, y) to a complex value, z, and then repeatedly compute $z = z^2 + c$, where c is fixed, and check whether or not z escapes some threshold value.

- if the threshold is not exceeded after a maximum number of iterations, then it is in the Julia Set,
- otherwise it is considered not to be in the Julia Set.

Julia Set Fractal Example (con't)



Some C Code…

C has built-in complex numbers, e.g., `float _Complex`, which makes this easier to write:

julia.c
```c
#include <complex.h>
#include <stdlib.h>
#include <stdio.h>

typedef unsigned short pixel_type;
typedef unsigned short coord_type;

#define DIM 10000

inline float cnormf(float _Complex const x)
{
  return crealf(x) * crealf(x) + cimagf(x) * cimagf(x);
}

pixel_type julia(coord_type x, coord_type y)
{
  float const scaling = 1.5;
  float _Complex const c = -0.8f + 0.156f * _Complex_I;

  float const scaled_x = scaling * ((float)(DIM/2) -
      x)/(DIM/2);
  float const scaled_y = scaling * ((float)(DIM/2) -
      y)/(DIM/2);
  float _Complex z = scaled_x + scaled_y * _Complex_I;

  for(unsigned short iter = 0; iter < 1000; ++iter)
  {
    z = z * z + c;
    if (cnormf(z) > 1000)
      return 0;
  }
  return 1;
}
```

Some C++ Code...

C++ code equivalent to the C code presented:

julia.cxx

```cpp
#include <atomic>
#include <chrono>
#include <complex>
#include <fstream>
#include <iostream>
#include <vector>
#include "now.hxx"

using namespace std;

using pixel_type = unsigned short;
using coord_type = unsigned short;

constexpr coord_type DIM = 10000;

pixel_type julia(coord_type const x, coord_type const y)
{
  constexpr float scaling = 1.5f;
  constexpr complex<float> c{-0.8f, 0.156f};
```

Some C++ Code… (con’t)

```cpp

  float scaled_x = scaling * (float(DIM/2) - x)/(DIM/2);
  float scaled_y = scaling * (float(DIM/2) - y)/(DIM/2);
  complex<float> z{scaled_x, scaled_y};

  for (unsigned short i=0; i != 1000; ++i)
  {
    z = z*z + c;
    if (norm(z) > 1000)
      return 0;
  }
  return 1;
}
```

- Subsequent presented code will be C++ code.

Some C++ Code… (con't)

We will not parallelize the `julia(x,y)` function:

- its code is simple, and,
- the number of loop iterations required for any point (x, y) is unknown.

Instead we will focus on calling `julia(x,y)` in parallel.

First, let's look at what needs to be done sequentially…

C++ Code for Reliable Timings

It is important to have a way to benchmark function calls:

now.hxx

```cpp
#ifndef now_hxx_
#define now_hxx_

#include <atomic>
#include <chrono>

// Include Google Benchmark (https://github.com/google/benchmark)...
#include <benchmark/benchmark.h>

inline void dno() noexcept { }

template <typename Arg, typename... Args>
inline void dno(Arg arg, Args&&... args)
{
  benchmark::DoNotOptimize(arg);
  dno(std::forward<Args>(args)...);
```

C++ Code for Reliable Timings (con't)

```
}

template <typename... Args>
inline auto now(Args&&... args) ->
  std::chrono::time_point<std::chrono::steady_clock>
{
  using namespace std;
  dno(std::forward<Args>(args)...);
  auto tp = chrono::steady_clock::now();
  dno(tp);
  return tp;
}

#endif
```

main()

The rest of the program is in main():

julia.cxx

```cpp
int main()
{
  vector<pixel_type> v(DIM*DIM);

  // "kernel" code to compute the Julia set...
  auto t0 = now(v.data());  // get start time_point
  for (coord_type y = 0; y != DIM; ++y)
    for (coord_type x = 0; x != DIM; ++x)
      v[y*DIM+x] = julia(x,y);
  auto t1 = now(v.data(),t0);  // get stop time_point

  // output elapsed time in seconds...
  cout << "elapsed time: " << chrono::duration<double>(t1-t0).count() << " seconds\n";

```

main() (con't)

```
  // write output to file...
  ofstream out("julia.dat");
  for (coord_type y = 0; y != DIM; ++y)
    for (coord_type x = 0; x != DIM; ++x)
    {
      if (v[y*DIM+x] == 1)
        out << x << ' ' << y << '\n';
    }
}
```

Parallelizing with OpenMP

OpenMP code:

julia-openmp.cxx

```cpp
int main()
{
  vector<pixel_type> v(DIM*DIM);

  // "kernel" code to compute the Julia set...
  auto t0 = now(v.data());  // get start time_point
  for (coord_type y = 0; y != DIM; ++y)
    #pragma omp parallel for
    for (coord_type x = 0; x < DIM; ++x)
      v[y*DIM+x] = julia(x,y);
  auto t1 = now(v.data(),t0);  // get stop time_point

  // output elapsed time in seconds...
  cout << "elapsed time: " << chrono::duration<double>(t1-t0).count() << " seconds\n";

  // write output to file...
  ofstream out("julia.dat");
  for (coord_type y = 0; y != DIM; ++y)
    for (coord_type x = 0; x != DIM; ++x)
```

Parallelizing with OpenMP (con't)

```
    {
      if (v[y*DIM+x] == 1)
        out << x << ' ' << y << '\n';
    }
}
```

C++17 Parallel Algorithms v1

C++17 parallel algorithms v1 code:

julia-paralg1.cxx

```
int main()
{
  vector<pixel_type> v(DIM*DIM);

  // run kernel function to compute the Julia set...
  vector<coord_type> xs(DIM);
  iota(xs.begin(), xs.end(), 0);

  auto t0 = now(xs.data());  // get start time_point
  for (coord_type y = 0; y != DIM; ++y)
  {
    transform(
      execution::par_unseq,
      xs.begin(), xs.end(),
      v.begin()+y*DIM,
      [y](coord_type const x) { return julia(x,y); }
    );
  }
  auto t1 = now(v.data(),t0);  // get stop time_point
```

C++17 Parallel Algorithms v1 (con't)

```cpp

  // output elapsed time in seconds...
  cout << "elapsed time: " << chrono::duration<double>(t1-t0).count() << " seconds\n";

  // write output to file...
  ofstream out("julia.dat");
  for (coord_type y = 0; y != DIM; ++y)
    for (coord_type x = 0; x != DIM; ++x)
    {
      if (v[y*DIM+x] == 1)
        out << x << ' ' << y << '\n';
    }
}
```

C++17 Parallel Algorithms v2

C++17 parallel algorithms v2 code:

julia-paralg2.cxx

```
int main()
{
  vector<pixel_type> v(DIM*DIM);

  // "kernel" code to compute the Julia set...
  vector<pair<coord_type,coord_type>> indices;
  indices.reserve(DIM*DIM);
  for (coord_type i{}; i != DIM; ++i)
    for (coord_type j{}; j != DIM; ++j)
      indices.emplace_back(pair{i,j});

  auto t0 = now(v.data(),indices);  // get start time_point
  transform(
    execution::par_unseq,
    indices.begin(), indices.end(),
    v.begin(),
    [](auto const& p) constexpr { return julia(p.first,p.second); }
  );
  auto t1 = now(v.data(),t0);  // get stop time_point
```

C++17 Parallel Algorithms v2 (con't)

```cpp

 // output elapsed time in seconds...
 cout << "elapsed time: " << chrono::duration<double>(t1-t0).count() << " seconds\n";

 // write output to file...
 ofstream out("julia.dat");
 for (coord_type y = 0; y != DIM; ++y)
  for (coord_type x = 0; x != DIM; ++x)
  {
   if (v[y*DIM+x] == 1)
    out << x << ' ' << y << '\n';
  }
}
```

Compiling and Linking Steps

On our clusters, load one of these module sets:

- GCC: `module load googlebenchmark gcc tbb`
- Clang: `module load googlebenchmark clang tbb`
- Intel: `module load googlebenchmark intel tbb`
- NVHPC (CPU): `module load googlebenchmark nvhpc`
- NVHPC (GPU): `module load googlebenchmark nvhpc cuda`

Compiling and Linking Steps (con't)

Compiling and linking C++ code with GCC:

- Serial: `g++ -O3`
- OpenMP: `g++ -O3 -fopenmp`
- ParAlg: `g++ -std=c++17 -O3`
- ParAlgLink: `-ltbb -lbenchmark`

Compiling and Linking Steps (con't)

Compiling and linking C++ code with Clang:

- Serial: `clang++ -O3`
- OpenMP: `clang++ -O3 -fopenmp`
- ParAlg: `clang++ -std=c++17 -O3`
- ParAlgLink: `-ltbb -lbenchmark`

Compiling and Linking Steps (con't)

Compiling and linking C++ code with Intel OneAPI:

- Serial: `icpx -O3 -fp-model precise`
- OpenMP: `icpx -O3 -fp-model precise -fopenmp`
- ParAlg: `icpx -O3 -std=c++17 -fp-model precise`
- ParAlgLink: `-ltbb -lbenchmark`

Compiling and Linking Steps (con't)

Compiling and linking C++ code with NVIDIA's nvhpc for CPU only:

- Serial: `nvc++ -O3`
- OpenMP: `nvc++ -O3 -fopenmp`
- ParAlg: `nvc++ -O3 -std=c++17 -stdpar=multicore`
- ParAlgLink: `-lbenchmark`

Compiling and Linking Steps (con't)

Compiling and linking C++ code with NVIDIA's nvhpc for CPU & GPU:

- Serial: `nvc++ -O3`
- OpenMP: `nvc++ -O3 -fopenmp`
- ParAlg: `nvc++ -std=c++17 -O3 -stdpar=gpu -gpu=managed`
- ParAlgLink: `-lbenchmark`

To run on a GPU NVHPC's `-stdpar=gpu` requires:

- using dynamically allocated memory, e.g., `std::vector`
- using lambda functions and/or templated function objects that do not capture host state / call stack variables

Compiling and Linking Steps (con't)

To generate compiler reports on missed vectorization, add these options when compiling:

- GCC: `-fopt-info-vec-missed`
- Clang: `-Rpass-missed=loop-vectorize`
- Intel OneAPI: `-qopt-report`
- NVHPC: `-Minfo`

Some Example Timings

On a 16-core AMD Threadripper with two hyperthreads per core using GCC and `-O3 -march=native`:

- Serial: 8.25s
- OpenMP: 1.14s
- ParAlg: 0.3s