Mastering HPC

Jaime Pinto James Willis

October 28, 2024

Outline

- Motivation
- Introduction
- Identifying your Application
- Code Compilation
- Interactive Sessions
- Performance Tuning
- Batch Submission
  - Script Examples
- Checkpointing
- Job Monitoring
- Debugging
- Hands-on

Motivation

- High Performance Computing (HPC) resources are expensive to run and maintain
- Utilising as much of the resources as possible is very important
- Running jobs on HPC resources can be complex and there are many ways to do it
- This course will help you to use HPC resources more efficiently
- This course is a result of repeated users' tickets of all levels of expertise
- Observing these steps will help you achieve your goals quicker

Introduction

- When running any program or application on Niagara, you need to make sure you are using the resources efficiently
- This means ensuring that:
    - As many cores on a node are running as close to a 100% load for as long as possible
    - Your problem will fit in the available amount of memory on a node
    - Your job will finish in the time you have requested or will be able to checkpoint and restart
    - You are performing IO correctly and not overloading the file system
- We will now go through a set of recommendations to best help you achieve this

Identifying your Application

Identifying your Application

- Before you start running your jobs on Niagara, you need to identify the type of application you have. This will help you determine the best way to run your job
- Some questions you should ask yourself are:
    - Is your code single-threaded or multi-threaded?
    - Is your code multi-node (i.e. MPI) or strictly serial?
    - Is your code a hybrid of both? (e.g. MPI + OpenMP)
    - Are you developing your own code or using 3rd party code?
    - Is your code containerized?
    - Are you using commercial code?
    - Is your code suitable to run on Linux?
    - Can your code run from the command line?
- Once you have identified the type of application you have, you can start to develop a plan to run your job

Definition and Terminology

- In the context of Slurm and HPC, the definitions of CPUs, cores, threads, sockets, and tasks are:
  - Node: A single computational machine
  - Socket: A physical socket on a motherboard that accepts a CPU part
  - Core: A physical CPU core that can execute instructions independently
  - Thread: A logical software unit that can run on a single core
  - CPU: Depending on the system configuration, this can be either a core or a thread
  - Task: An instance of an executed command
- Slurm uses the terms "core" and "CPU" interchangeably depending on the context and command. For example, the –cpus-per-task option specifies the number of cores per task.

Serial Applications

- Run on a single core on a single node
- If ran unmodified, they will only utilise 2.5% of the CPU wasting 39/40 cores on a node
- Multiple independent jobs can be bundled together on a single node to maximise CPU usage
- For more details please see: [https://docs.scinet.utoronto.ca/index.php/Running_Serial_Jobs_on_Niagara] (https://docs.scinet.utoronto.ca/index.php/Running_Serial_Jobs_on_Niagara)

Multi-threaded Applications

- Run on a single node using OpenMP or equivalent
- Can run on multiple cores on a single node utilising shared memory
- `OMP_NUM_THREADS` environment variable is used to set the number of threads

Multi-node Applications

- Run on multiple nodes using MPI
- Each node has its own memory and communicates with other nodes over the network
- MPI libraries are used to manage the communication between nodes
- The problem is divided into smaller parts and each part is run on a separate node

Containerized Applications

- Containerized applications are created using a software deployment process called containerization, which bundles an application's code with its required files and libraries.
- They run in isolated packages of code called containers.
- Containers include all the dependencies that an application might need to run on any host operating system, such as libraries, binaries, configuration files, and frameworks, into a single lightweight executable.
- On the Alliance systems the preferend method of containerization is via Apptainer: https://docs.alliancecan.ca/wiki/Apptainer
- Note that singularity has been deprecated in favor of Apptainer
- Docker files can be converted to Apptainer: https://docs.alliancecan.ca/wiki/Apptainer#Creating_an_Apptainer_container_from_a_Dockerfile

Hybrid Applications

- Runs on multiple nodes using MPI + OpenMP
- MPI manages communication between nodes
- OpenMP performs the computations in shared memory within a node

Commercial Code

- Commercial code is only available if you have a license
- Even with a license, additional setup may be required to run on Niagara (e.g. ssh tunnel to external license server)
- Here are some commercial codes that some users have been able to run on Niagara:
    - ANSYS
    - COMSOL
    - VASP

Code Compilation

Code Compilation 1

- Before you can run your code on Niagara, you need to compile it
- You should always compile your code on the login nodes as they have internet access and are the same architecture as the compute nodes
- Depending on the software you are using, you may need to load a set of modules to gain access to right compilers and libraries
- There are two software stacks available on Niagara: `NiaEnv` (Default) and `CCEnv`
- They can be switched using the `module load` command. For example to load the latest `NiaEnv` and `CCEnv` stacks use:

```
module load NiaEnv/2022a
```

```
module load CCEnv StdEnv/2023
```

Code Compilation 2

- The code you require may be available as a module on Niagara already
- To check if a specific module is present, you can run: `module spider <module_name>`
- Find what software dependencies your code has and load the appropriate modules (e.g. documentation, README, etc.)

Code Compilation 3

- For example, to see what OpenMPI libraries are available, you can run: `module spider openmpi`:

```
user@nia-login03:~$ ml spider openmpi

----------------------------------------------------------------------------
  openmpi:
----------------------------------------------------------------------------
    Description:
      An open source Message Passing Interface implementation

     Versions:
        openmpi/4.1.2+ucx-1.11.2
        openmpi/4.1.4+ucx-1.11.2
        openmpi/5.0.2+ucx-1.15.0
```

Code Compilation 4

- Then to see what dependencies a specific version has run: `module spider openmpi/5.0.2+ucx-1.15.0`:

```
user@nia-login03:~$ module spider openmpi/5.0.2+ucx-1.15.0

-----------------------------------------------------------------------
  openmpi: openmpi/5.0.2+ucx-1.15.0
-----------------------------------------------------------------------
    Description:
      An open source Message Passing Interface implementation



    You will need to load all module(s) on any one of the lines below before
    the "openmpi/5.0.2+ucx-1.15.0" module is available to load.

      gcc/11.3.0
```

Code Compilation 5

- Finally, to load the module, you can run:

```
module load gcc/11.3.0 openmpi/5.0.2+ucx-1.15.0
```

- Once you have loaded the correct modules, you can compile your code using the appropriate compiler
- You are now ready to run the executable on the login nodes

Preliminary Checks

- You should make sure your executable runs on a login node
- Start by running your executable on 1 core, then 2 cores, and finally 4 cores
- Ensuring it doesn't fail and runs for no more than 15 minutes as the login nodes are shared resources
- PLEASE, REFRAIN FROM SUBMITTING A JOB FOR THE FIRST TIME ASKING ALREADY FOR THE MAX NUMBER OF NODES AND THE MAX AMOUNT OF TIME, IF YOU ARE NOT REASONABLY SURE IT WILL WORK, OR YOU WILL BE WASTING A LOT OF RESOURCES.
- These are only preliminary checks and not production runs

Interactive Sessions

Interactive Session 1

- Once you have confirmed your executable runs on the login nodes, the next step is to run on a node that you can scale up to 40 cores
- Also ensure that you have no dependency on internet access:
  - For your code
  - Datasets
  - Software licenses
- If your code needs external data then you have to break the execution into 2 steps:
  1. Download the data from a login node first (if small amounts) or the datamovers (`nia-dm1`/`nia-dm2`)
  2. Then run the code offline on the debug/compute nodes

Interactive Session 2

- We recommend that you request an interactive session on the *debug queue* (at a higher priority)
- This will allow you to run your job on a **single node** for up to 40 cores and up to 1 hour on a dedicated node
- You can request an interactive session using the `debugjob` or `salloc` command:

  ```
  debugjob 1
  ```

  or

  ```
  salloc --time=1:00:00 -N 1 --account=def-user -p debug
  ```

- In principle, any request you would submit with a script to the scheduler you could request on an interactive job with `salloc` as well

Interactive Session 3

- There are more details on interactive jobs here:
  - https://docs.scinet.utoronto.ca/index.php/Niagara_Quickstart#Testing_and_Debugging
  - https://docs.alliancecan.ca/wiki/Running_jobs#Interactive_jobs
  - https://slurm.schedmd.com/salloc.html
- You can now try and scale up your job to use all 40 cores on the node
- Making sure you don't run out of memory or encounter any other single node issues
- Pay close attention to any error messages or notifications in the standard output
- Fix any bugs detected at this stage, if any

Multiple Nodes (if it applies to you)

- Once you have confirmed your job runs on a single node, you can request an interactive session on the debug queue for **two nodes** (`debugjob 2`)
- This will give you an interactive session on two nodes for up to **30 minutes**
- Start adjusting your scripts to run over multiple nodes

Batch Submission

Batch Submission 1

- Now that your job has shown it can run on multiple nodes in an interactive session, you can focus on the submission script itself using batch mode
- Make sure you are submitting your jobs from a path under `$SCRATCH`, this will be known as your **working directory**
- Keep all your scripts, executables, input files, and output files in this directory. This helps support staff in case you need assistance
- By default SLURM will generate output and error files into your working directory unless you specify otherwise. Do not modify the default location until your workflow is fully developed

Batch Submission 2

- Submit your job to the debug queue for **one node** for 15 minutes using the `sbatch` command:

```
sbatch -N 1 --time=0:15:00 submit.sh -p debug
```

- If that runs successfully, submit your job to the debug queue for **two nodes** for 30 minutes:

```
sbatch -N 2 --time=0:30:00 submit.sh -p debug
```

- Pay close attention to any error messages or notifications in the output logs. As a general rule, there are always hints and/or suggestions in the log files. Sometimes the error message may appear in the middle of the log, so please read the whole log.

Batch Submission 3

- Login to the compute node (headnode) from another shell to check what is going on via `ssh`:
  - Check which nodes your job is running on using `squeue --me` or `sq` command
  - Then login to the node using `ssh <node_name>`
  - Try running the `htop` command to monitor CPU and memory usage
  - Track memory utilisation with `free -g`
- `htop` is a useful tool for monitoring CPU and memory usage on a node
- `htop` demonstration (time permitting)
- If your job fails and you need to debug it further, you can run always revert to running the script interactively with `bash submit.sh` from an interactive session

Batch Submission 4

- You should be confident now that your application scales efficiently up to 2 nodes at least
- From this point on, and only then, submit your jobs to the normal batch queue (`sbatch submit.sh`)
- Start with **one hour** at first on **two nodes** only, then proceed to two hours and ten nodes etc. Scale up gradually and verify that your code's performance doesn't degrade or decay with larger numbers of nodes
- PLEASE, REFRAIN FROM SUBMITTING A JOB FOR THE FIRST TIME ASKING ALREADY FOR THE MAX NUMBER OF NODES AND THE MAX AMOUNT OF TIME, IF YOU ARE NOT REASONABLY SURE IT WILL WORK, OR YOU WILL BE WASTING A LOT OF RESOURCES.
- We will now go through some templates of how to run different types of jobs

Serial Applications

- This can be achieved using **GNU Parallel**, for example:

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=40
#SBATCH --time=12:00:00
#SBATCH --job-name gnu-parallel-example

# Turn off implicit threading in Python, R
export OMP_NUM_THREADS=1
module load NiaEnv/2019b gnu-parallel

parallel -j $SLURM_TASKS_PER_NODE <<EOF
  cd serialjobdir01 && ./doserialjob01 && echo "job 01 finished"
  cd serialjobdir02 && ./doserialjob02 && echo "job 02 finished"
  ...
  cd serialjobdir80 && ./doserialjob80 && echo "job 80 finished"
EOF
```

t

Multi-threaded Applications

- Example SLURM submission script:

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --cpus-per-task=40
#SBATCH --time=1:00:00
#SBATCH --job-name openmp_job
#SBATCH --output=openmp_output_%j.txt
#SBATCH --mail-type=FAIL

module load NiaEnv/2022a
module load intel/2022u2

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

./openmp_example
# or "srun ./openmp_example".
```

t

Multi-node Applications

- Example SLURM submission script:

```
#!/bin/bash
#SBATCH --nodes=8
#SBATCH --ntasks-per-node=40
#SBATCH --time=1:00:00
#SBATCH --job-name mpi_job
#SBATCH --output=mpi_output_%j.txt
#SBATCH --mail-type=FAIL

module load NiaEnv/2022a
module load intel/2022u2
module load intelmpi/2022u2+ucx-1.11.2

mpirun ./mpi_example
# or "srun ./mpi_example"
```

Hybrid Applications

- Example SLURM submission script:

```
#!/bin/bash
#SBATCH --nodes=8
#SBATCH --ntasks-per-node=10
#SBATCH --cpus-per-task=4
#SBATCH --time=1:00:00
#SBATCH --job-name hybrid_test

module load NiaEnv/2022a
module load intel/2022u2
module load intelmpi/2022u2+ucx-1.11.2

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

mpirun ./hybrid_example
```

Performance Tuning

Performance tuning for the watchmaker

- If you suspect that your code is not running optimally, you can try experimenting with different module stacks, compilers etc.
- For example try the `CCEnv` stack instead of `NiaEnv` or vice versa
- Potentially try *hyperthreading*:
  - Niagara has 2-way hyperthreading enabled on its Intel CPUs, so each core has two threads
  - You can run up to 80 threads on a single node
  - Some applications see a small performance increase of 5-10%
  - Your application may benefit from hyperthreading, but not always
  - So please experiment and see what works best for your application
  - Please see: https://docs.scinet.utoronto.ca/index.php/Slurm#Hyperthreading:_Logical_CPUs_vs._cores

Checkpointing

Checkpointing 1

- Checkpointing is the process of saving the state of your application at regular intervals
- It is a useful technique to add to your workflow in case of disruptions to the execution, i.e. power outages, hardware failures, etc.
- This allows you to restart your application from the last checkpoint in case of a failure, saving you time, lost progress and wasted compute resources
- Niagara limits jobs to a maximum wallclock time of 24 hours, so checkpointing is also useful if your job runs for longer than this

Checkpointing 2

- Checkpointing is achieved by:
  - Writing the entire state of your application to the file system
  - Performed every X time steps or every Y minutes so as not to overload the file system or impact the performance of your application
  - Then subsequently reading the state back in when you restart your application from the checkpoint file
  - Checkpointing instructions have to be given and executed prior to the requested time has elapsed. You don't want your job to timeout before it has a chance to checkpoint
- You should checkpoint your application at regular intervals to avoid losing too much work in case of a failure
- You can find more information on checkpointing here:
  - https://docs.scinet.utoronto.ca/index.php/Checkpoints

Job Monitoring

Job Monitoring 1

- An important resource to help you keep track of your jobs and see how efficiently they are running is the `my.scinet` portal: https://my.scinet.utoronto.ca

- You can login to the `my.scinet` portal using the same credentials you use to login to CCDB

Features:

- Niagara CPU and storage utilisation
- Status of the login nodes
- History of your jobs on Niagara and Mist
- Per job:
  - Job script
  - Job environment
  - CPU and memory usage updated every 10 minutes
  - GFLOPS and disk IO rates updated every 10 minutes
  - Lots of other useful metrics

Job Monitoring 2

- `my.scinet` demonstration (time permitting)
- There is also the recently created Alliance Portal, which complements the `my.scinet` portal very well: https://portal.alliancecan.ca
- The portal focuses on what resources are being used, when and by whom
- Alliance Portal demonstration (time permitting)
- For real time monitoring logging into the compute nodes and running `htop` is a good way to see what is going on
- To get a quick snapshot of the performance of a **running** job there is also the `jobperf <job_id>` command

Job Monitoring 3

- This will give an overview of the performance of your job, i.e. CPU and memory usage

```
user@nia-login06:~$ jobperf 13753632
--------------------------------------------------------------------------------
                    RUNNING          USER        MEMORY(MB)
HOSTNAME    #THD  CPUUSE  %MEM       NAME        USED     AVAIL   PROCESS NAMES
--------------------------------------------------------------------------------
nia0001     40    97%      44.4%     willis2     107889   85178   mpirun 2*srun 40*xhpl
nia0002     40    98%      43.4%     willis2     103190   89863   40*xhpl
--------------------------------------------------------------------------------
```

Job Arrays

- Job arrays are a way to submit multiple jobs with a single submission script
- This is useful if you have a large number of jobs that are similar and can be run **independently**
- However, you should NOT use them to run serial jobs on Niagara
- If you have a set of jobs that you know scale well to at least one node, then you can use job arrays to submit them all at once
- Note that a job array is like a knife that cuts both ways: if your jobs are not efficient they will consume/waste the allocation of your group very quickly
- You can find more information on job arrays here: https://docs.alliancecan.ca/wiki/Job_arrays/en

`mpirun` vs `srun` vs `mpiexec`

- Use `mpirun` if it doesn't work use `srun` and if that doesn't work use `mpiexec`

Debugging

Debugging Errors

- If your job fails, you can check the output file for any error messages
- You can also check the SLURM logs for more information on why your job failed
- SLURM output and error files can be renamed with the `--output` and `--error` options in your SLURM script
- Give your log files meaningful names, for example:

```
#SBATCH --job-name=gromacs
#SBATCH --output=%x_%j.out
#SBATCH --error=%x_%j.err
```

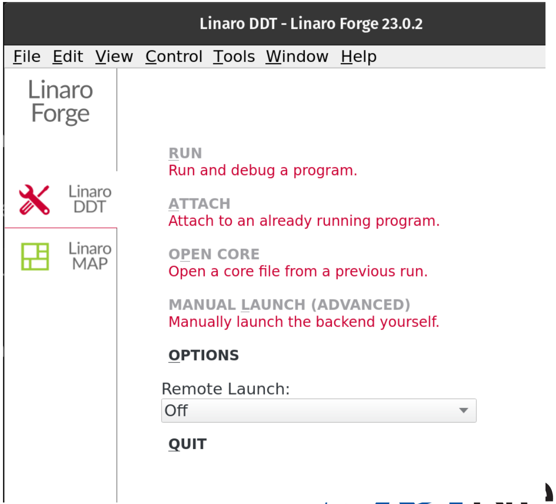
- This will create files like `gromacs_12345.out` and `gromacs_12345.err` for job ID 12345, where `%x` is the job name and `%j` is the job ID

Debugging Errors

- If you fail to identify the error from the log files, you will need to look at using a debugger
- There are many debuggers available on Niagara, including `gdb`, `valgrind`, and `DDT`
- Let’s take a quick look at DDT

What is DDT?

- **DDT** stands for **Distributed Debugging Tool**
- Powerful GUI-based commercial debugger by *Linaro*
- Developed for debugging parallel, multi-threaded, and distributed applications
- Widely used in high-performance computing environments
- Available on Niagara and other Alliance systems (*Note: license only allows debugging up to 64 processes*)



Run and debug a program.

Attach to an already running program.

Open a core file from a previous run.

Manually launch the backend yourself.

Remote Launch: Off

DDT Features

- **Key Features:**
  - Parallel and distributed debugging capabilities
  - Graphical user interface for intuitive navigation
  - Support for multiple programming languages (e.g. C, C++, Fortran, Python)
  - Supports MPI, OpenMP, threads, CUDA, ROCm and more
  - Integrated performance analysis tools - *MAP*
  - Memory debugging functionalities
- We have a full course on DDT, happening April 28th. Please sign up if you are interested: https://education.scinet.utoronto.ca/course/view.php?id=1373

Hands-on

- With the remaining time, we can help you with any issues you are having with your jobs

Summary

- We have covered a lot of information in this course:
    - Identifying your application
    - Code compilation
    - Interactive sessions
    - Performance tuning
    - Batch submission
    - Checkpointing
    - Job monitoring
    - Debugging

Further information

Useful sites

- Niagara: https://docs.alliancecan.ca/wiki/Niagara_Quickstart
- Mist: https://docs.scinet.utoronto.ca/index.php/Mist
- Other Alliance clusters or general topics: https://docs.alliancecan.ca
- System Status: https://docs.scinet.utoronto.ca
- Training: https://education.scinet.utoronto.ca/

Support

- Email to **niagara@tech.alliancecan.ca** or **support@scinet.utoronto.ca**
- Still need help? Request a one-to-one consultation (request via email).