Research Computing at Mines Workshop
Serial and Parallel Computing

October 2023

Presented by:
Nicholas A. Danes, PhD
Computational Scientist
Cyber Infrastructure & Advanced Research Computing (ITS)
Recap of Day 1

• Overview of the world of Cyberinfrastructure & Research Computing
• HPC options for Mines Researchers
• Overview of skills needed to be a successful researcher on HPC
  • Linux/Bash
  • Slurm/Job Scheduler
  • Parallel Computing
  • Computational Notebook Practices
• Intro to Linux/Bash Lab
• Overview of Job Schedulers, SLURM and Python
• Intro to Slurm & Python Lab
Goals for Day 2

• Overview of Serial vs Parallel Computing
• A case study using a Python serial code (Lab)
• Parallel Programming Overview
  • Shared vs Distributed Memory
  • MPI
  • OpenMP
• Lab on using parallelized software: GROMACS
HPC Resource Usage

• How do I use them?
  • Most programs spawn 1 process (“task” in Slurm) and use one thread (“cpu” in Slurm)
    • On a desktop, some programs can see how many CPU cores you have and request that many threads for the process and use them
      • Examples: Some MATLAB functions, Games using DX12, Chrome/Firefox
  • Slurm does not know how your program will use the resources you give it
    • If you give it 12 cores (“cpus”) but program only works with 1 core, those 11 cores will idle and do nothing
  • To think about how to utilize HPC resources, we need to learn how parallel programming/processing is implemented.
Serial vs Parallel Computing

• When a program uses a single process ("task") with 1 core ("cpu"), we say it is a **serial computing** program.
• When a program uses multiple cores, we say it is a **parallel computing** program.
• Before thinking about parallel computing, we need to focus on how well the program performs with serial computing.
Note: Optimization before Parallelization!

• “Premature optimization is the root of all evil” – Donald Knuth

• Often, writing your code to run as fast as possible (within reason) with a single core is necessary before thinking about parallelization.

• We will explore optimization with a simple Python code for a single core next!
How to profile Python code

• cProfile
  • Gives you a breakdown of all functions’ runtime in a code
  • Multiple ways to use it:
    • Call it in the command line:
      $ python -m cProfile myscript.py
    • Call it in another script:
      ```python
      import cProfile
      cProfile.run("mycode.main()")
      ```

• Other options: lineprofiler, timeit, pstats

References: https://towardsdatascience.com/how-to-profile-your-code-in-python-e70c834fad89
https://github.com/pyutils/line_profiler
A starting point for optimization: Writing an ODE solver

Consider the initial value problem of the form:

\[ y'(t) = f(t, y) \]
\[ y(t_0) = y_0 \]

which can numerically solved using Heun’s Method:

\[ \hat{y}[i+1] = y[i] + h f(t[i], y[i]) \]
\[ y[i+1] = y[i] + \frac{h}{2} \left( f(t[i], y[i]) + f(t[i+1], \hat{y}[i+1]) \right) \]

Where \( h \) is the time step size, \( i \) is the time step index, and \( \hat{y} \) denotes the intermediate solution. Let’s use this problem to see how to optimize writing scientific code for Python!
Lab #1: Serial Python Optimization

Copy the workshop materials using the following command:

```
cp /sw/BUILD/src/workshop/Workshop_Fall2023_day2.tar.gz ~/scratch
```

And untar it and go to the directory:

```
cd ~scratch && tar -xf Workshop_Fall2023_day2.tar.gz
cd Workshop_Fall2023/rk2_python && ls
```

Using Open OnDemand Interface

Go To: https://wendian-ondemand.mines.edu
Lab #1 Summary: Profiling multiple versions of our ODE code

• Pure Python
  • **Surprisingly Performant!**

• NumPy only
  • Performs poorly due to lack of vectorization

• NumPy + Cython
  • [https://cython.org/](https://cython.org/)
  • Cython effectively allows one to write static-typed code in Python/“Cython”, which is parsed into C and compiled into a Python module.
Parallel Programming

- Shared vs Distributed Memory Programming
  - Shared (e.g. OpenMP)
    - All CPU cores have access to the same pool of memory
    - Typically, all CPU cores are on the same CPU node
    - Ideal for multi-threaded loops
  - Distributed-memory program (e.g. MPI)
    - Each CPU core is given access to a specific pool of memory, which may or may not be shared
    - A “communicator” designates how each CPU core can talk to another CPU core
    - CPU cores do not have to live on the same CPU node

**Shared Memory Parallelism:**
1 task, 4 threads

**Distributed Memory Parallelism:**
4 tasks, 1 thread per task
Shared Memory Parallelism: OpenMP

- OpenMP is a portable, high-level API that is used to write multithreaded applications
  - It provides a set of directives that can be used to parallelize loops, regions of code, and entire functions.
  - Supported by a wide range of compilers and hardware platforms (e.g. C/C++, Fortran, Python, etc)
  - For loops typically a compiler directive is added before a loop to tell the compiler that OpenMP is being used:
    - `#pragma omp parallel`
  - The environment variable OMP_NUM_TASKS will tell the operating system how many OpenMP threads to use in the program.
A note on Python and the GIL: A constraint on shared memory programming

- Python Global Interpreter Lock (GIL)
  - A mechanism with Python which allows only one CPU thread to use the Python interpreter
  - The GIL addressed the problem of memory management for Python programs.
  - Releasing the GIL can cause memory leaks if not managed correctly.

- Solutions:
  - Use multiprocessing instead of multithreading
    - Each process gets its own Python interpreter and memory space
    - Module options: mpi4py, multiprocessing
  - Use a different interpreter
  - Use Cython to release the GIL to allow multithreading within subroutines

Reference: https://realpython.com/python-gil/
Distributed Memory Parallelism: MPI

- MPI stands for message-passing interface, standard provided as a library for exchanging data (called messages) between objects.
- Different libraries have implemented the MPI standard:
  - OpenMPI
  - MPICH
  - Intel MPI
- Objects that can be used to send messages are separated by memory
  - Can be entire CPU nodes, or CPU cores, called ranks.
  - By breaking up by memory of each tasks, a rank can send messages theoretically anywhere as long as there is another layer of network communication
    - MPI most commonly uses Infiniband for node-to-node communication
    - Intra-node communication uses CPU architecture
Distributed Memory Parallelism: MPI

- It provides a set of functions for sending and receiving messages, as well as for synchronizing the execution of different processes. Common functions:
  - Send a message from one rank to another: MPI_SEND
  - Receive a message from a rank: MPI_RECV
  - Broadcast the same message to all ranks from a particular rank: MPI_BCAST
  - Take a message from multiple ranks to a single rank: MPI_GATHER
  - Block messages from continuing until all ranks have finished: MPI_BARRIER
Choosing between Shared vs Distributed Memory Parallelism

• Shared Memory Parallel is ideal for:
  • Single computer/node workloads
  • Speeding up for-loops
    • By splitting up the work across loop iterations

• Distributed Memory Parallel works best for
  • Large memory workloads that require multiple compute nodes

• Shared and distributed memory parallel programming can sometimes be combined
  • Called **hybrid** parallel programming
    • Combining MPI and OpenMP
Lab #2: Running Parallel Code: GROMACS

And exploration of SLURM commands
Summary of Slurm commands to monitor jobs

• squeue – View job queue
• squeue $USER – View job queue for your jobs
• sacct –j <jobid> – Get info on a particular job ID
• sinfo – Show info on nodes
• scontrol show node <name> – show info on a particular node
GPU: Further Accelerate computational workloads

• Use graphical processing units (GPUs)
  • NVIDIA
    • The CUDA library is by far the most popular GPU computing language
      • Provides an API to use NVIDIA CUDA codes
    • Popular CUDA uses:
      • AI/ML: PyTorch, Tensorflow
      • Molecular Dynamics: LAMMPS, GROMACS
      • Scientific Visualization: Paraview
  • OpenCL
    • Open source alternative to CUDA
    • Works on AMD, NVIDIA, and Intel GPUs
  • HIP (Heterogeneous-Compute Interface)
    • GPU acceleration library developed by AMD
Limitations to GPUs

- Hardware is more expensive and less widely available
- Programming for GPUs requires more setup
- Not all workloads can be easily ported to GPUs
Final Takeaways

• When developing your own scientific programs, get the serial case working as efficiently as possible (within time constraints)
  • Try different libraries, compiler options, etc.
  • Run benchmarks to compare setups

• When jumping to parallel programming
  • Understand the differences between shared and distributed memory parallel programming
  • Leverage established libraries to implement parallel programming methods
  • Use software with these libraries already in use (e.g. GROMACS)
Further Resources

Python Parallel Processing:
https://wiki.python.org/moin/ParallelProcessing

Parallel Programming with MP implementations for Python:

Intro to F2Py:
https://www2.atmos.umd.edu/~dkleist/docs/pythonTraining/Slides/F2Py_SSSO.pdf

OpenMP: Open MP specifications
(open-mpi.org)

OpenMP: Specifications – OpenMP

NVIDIA CUDA Toolkit: CUDA Toolkit - Free Tools and Training | NVIDIA Developer