Supercomputing @ MPIA

Data Science Workshop - March 2024

Chapter 1: Parallel Processing

Terminology

- **Storage:** Long-term, Large, Slow: HDD (~0.1 GB/s), SDD (~1 GB/s)
- **Memory:** Volatile, Fast(er) (~10 GB/s)
- **Node:** Component Computer
  - **CPU:** Physical Chip: Contains Cache Memory + Compute Unit(s).
  - **Core:** A single compute unit capable of billions of actions /s.
  - **Thread (logical CPU):** Number of concurrent threads supported.
- **Thread:** A context of execution where computation occurs.
- **Process:** Instance of a program in memory. Contains the resources necessary for computation (memory, sockets, security). Can spawn (child) processes and will use thread(s) to accomplish computation.

Parallel–Processing

Multi-Threading:

- **Not** to be confused with Simultaneous Multi-Threading or Hyperthreading
- Multiple concurrent threads access the same shared memory.
- Less overhead, one copy of process.
- Limited to a single Node.
- Can lead to nastiness (race conditions).
- Python can't multi-thread (GIL).
- *Many* low-level libraries are MTed.

Multi-Processing:

- Multiple processes working together.
- Can be distributed over multiple nodes.
- Can even be distributed over networks.
- Each process has its own memory, though it's possible to share.
- Much more overhead incurred from spawn, copy, and communication.
- Usually, if we want to do this on large scale, we need **MPI** (or similar).

Exercise #0: Multi-Threading

HPCWorkshop/Chapter1/multithreading:

- **C:**
  - Native: multithread.c (cumbersome)
  - OpenMP: multithread_omp.c (simpler but not race-safe)
- **Python:**
  - multithread.py (offers little benefit due to GIL)
- **Julia:**
  - multithread.jl (slow due to overhead and race-safe)

Exercise #0: Takeaways

1. Parallelism benefits can be lost quickly do to overheads.
2. Use low-level libraries (or packages that call them) to write fast code.
3. When possible, work smarter, not harder.
4. Writing **fast/efficient** multithreaded low-level code is hard.
5. In general, unless we are writing/optimizing an algorithm, we want…

Exercise #1: Multi-Processing

HPCWorkshop/Chapter1/multiprocessing:

- **C:**
  - Native: multiprocess.c (can only be run one one machine)
  - MPI: multiprocess_mpi.c (can be scaled to an HPC/network)
- **Python:**
  - multiprocess.py (can only be run one one machine)
  - MPI: multiprocess_mpi.py (can be scaled to an HPC/network)
- **Julia:**
  - multiprocess.jl (easy to set up and distribute)
  - multiprocess_mpi.jl (possible but required configuring)

Exercise #1: Takeaways

1. MPI is our friend for parallel computing!
    - Can be used with many languages across multiple nodes.

2. Chunking is incredibly important to reduce overheard.
    - Splitting up our work too finely will remove our benefits!

3. Many codes out there will use OpenMPI or MPICH.
    - But these will usually be available to use on MPCDF/astro-nodes.

Chapter 2: Batch Processing

Compute Facilities Available to YOU

	Nodes	CPUs/Cores/Threads	Memory	Storage
MPIA Laptop	1	1/10/10	16GB	~1TB
Astro-nodes	~25	1/32/32 - 2/72/144	250GB - 1TB	~1PB
MPCDF Cobra	~3000	1/40/80	250GB - 1TB	~1PB
MPCDF Raven	~1600	2/72/144	250GB - 1TB	~1PB
MPCDF Vera	~100	2/72/72	250GB - 1TB	~1PB

MPCDF

Max Planck Computing and Data Facility

- In general we don't access the compute nodes directly.
- First, we access a Bastion Host with **password and OTP**:
  - *ssh ***@gate.mpcdf.mpg.de*
  - Purely for getting elsewhere.
- Then, access the cluster login node:
  - *ssh ***@vera01.bc.rzg.mpg.de*
  - *ssh raven* or *ssh cobra*
  - Login nodes are used for:
    - Editing code.
    - Compiling binaries.
    - Submitting jobs.

MPCDF

Max Planck Computing and Data Facility

- See ssh_config file to make this "slightly" less painful.

- How do I transfer data to MPCDF?
  - Small (< 1GB): scp
  - Medium (<250GB): rsync
  - Large (>250GB): Globus

- File Systems & Storage:
  - Home directory (~1TB)
    - For Results and Code
  - Scratch (~1PB or 5TB)
    - NOT backed up, for heavy I/O tasks.
  - Long-term Tape Storage

Scheduler/System

- Responsible for taking requests and allocating computational time.
- Designed for HPC systems: monitoring, fault tolerance, etc.
- **Required information:**
  - Number of Threads and Cores (CPUs+GPUs)
  - Amount of Memory
  - Necessary Packages/Modules
  - Maximum Runtime
- There are a few common ones, but we will focus on Slurm:
  - Currently the one used by the MPCDF.

Anatomy of a SLURM Script

Preamble:

- Meta-information about the job.
  - stdout, stderr
  - Job Name
  - Working Directory
- Requests resources for the job.
  - Nodes
  - Memory
  - Core
  - SMTs
- Array Jobs.
- Conditional Jobs.

Main Body:

- Loads relevant modules.
  - Conda
  - OpenMPI
- Runs your code!

SLURM Commands

- **sbatch:** Submit a job.
- **srun:** Run a parallel command.
- **squeue:** Monitor the queue
- **scancel:** Cancel a job

Exercise #2: Slurm Scripts

Log-in to cobra:

- Helloworld.slurm
- Helloarray.slurm

Log-in to raven-i (for interactive example) or raven:

- Square_mpi.slurm (copy over python script from Chapter 1)
  - module load anaconda/3/2023.03 gcc/13 openmpi/4.1
  - conda create --name fast-mpi4py python=3.8 -y && conda init
  - source ~/.bashrc
  - conda activate fast-mpi4py && pip install mpi4py --no-cache-dir
  - srun -n 2 -p interactive multiprocess_mpi.py

Exercise #2: Takeaways

1. Parallel code is (mostly) straightforward to run with SLURM.
2. Batch scripting can make iterating over large samples efficient.
3. Understand the available modules.
4. Interactive nodes can be used for testing purposes.
5. In general, queue times are long. Ensure your code runs first!

Chapter 3: Workflow Managers

Automate Building Scripts

Make:

- Reads a Makefile to execute.
- Broken down into:
    - target
    - dependencies
    - commands
- Trivially parallelizable.
- Designed for automating software compilation.

Snakemake:

- Reads a Snakefile
- Broken down
    - Rules
    - Targets
    - Outputs
- Built with Python.
- Designed to automate data processing tasks.

Exercise #3: Snakemake

Run a Snakemake Workflow

- conda env create -f environment.yaml
- conda activate snakemake
- snakemake --dryrun
- snakemake --process 3
  - Snakemake will run until all outputs are created
  - -F to force scripts to run again.
- snakemake --dag | dot -Tsvg > dag.svg
  - With graphviz we can visualize the dependency structure.

Exercise #3: Challenge

Complete this Snakemake workflow

- We’re going to download SDSS spectra and plot them in Fnu.
- Remember to think about the workflow!
- Look in the Answer file if you get stuck.
- We can then run this on the HPC in ONE line!
    - --cluster "sbatch --cpus-per-task=2 --mem=2000 -t 1" --jobs 3

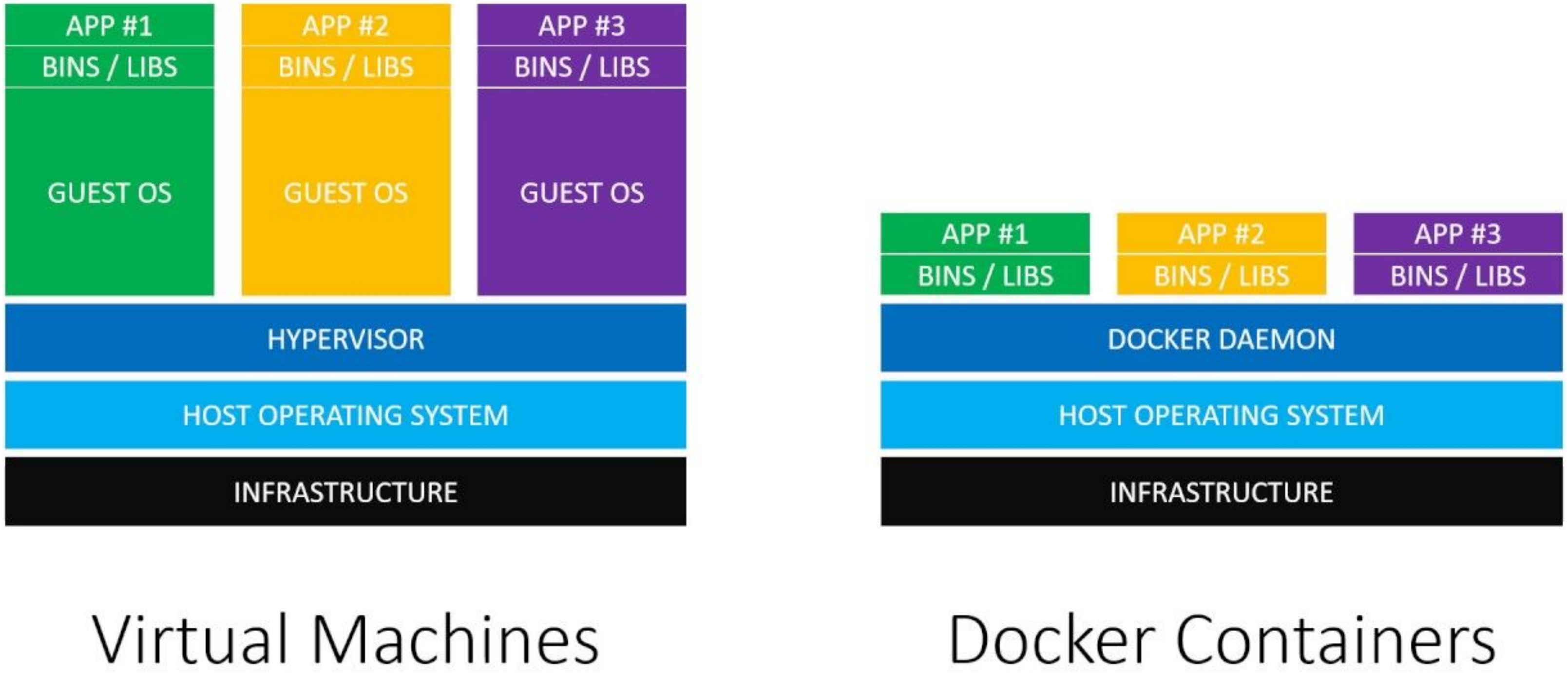
Exercise #3: Takeaways

1. Snakemake rules are focused on creating products (files).
2. Snakemake can handle the relationship between rules.
3. Resource limits and batching make snakemake efficient.
4. Can help encapsulate entire project workflows.
5. Visualizing your workflow and dry-run for project management.
6. Incredibly extensible with conda and containers and HPC systems!

Chapter 4: Containers

- Solves the age-old issue: “But it runs fine on MY machine!”
- Containers are a standard unit of computation:
  - Contains *everything* you need to run code.
- Industry Standards:
  - Docker: Typically not used on HPC systems due to security.
  - Apptainer (nee Singularity)
  - Charlie Cloud (optimized for efficiency)
- NOT a virtual machine, no hardware virtualization!
  - This means a smaller hit to performance!

Containers


APP #1
BINS / LIBS
GUEST OS
APP #2
BINS / LIBS
GUEST OS
APP #3
BINS / LIBS
GUEST OS
HYPERVISOR
HOST OPERATING SYSTEM
INFRASTRUCTURE
Virtual Machines
APP #1
BINS / LIBS
APP #2
BINS / LIBS
APP #3
BINS / LIBS
DOCKER DAEMON
HOST OPERATING SYSTEM
INFRASTRUCTURE
Docker Containers

Containers

- Images are built from definition files.
- **Note:** Images always require **root** to build.
- However, Apptainer are executed w/o root, unlike Docker.
- Once we have a Container running, we can execute code within it!
- In general, we can pull images made by anyone in the community!
  - Ideally we can find a pre-made image of software that we want.
  - We can also take an existing image, and add on top of it.
- They can be surprisingly lightweight even with an entire OS inside

Apptainer

Exercise #4: Containers

Download The Docker App

- We will use a Dockerfile to specify a simple environment.
- docker build buildx --tag python:latest .
  - Docker hashes intermediate steps to keep things fast.
- docker run -v ~path:/app -w /app --name python -dit python:latest
  - -v Mounts a local directory.
  - -dit Keeps the container open and runs in the background.
- docker exec python ./dockertest.py

Exercise #4: Takeaways

1. Containers are effective ways of sharing reproducible code.
2. They are easy to make and share, there might already be one for you!
3. Container systems are ideal for deploying at scale.
4. Want your new student/postdoc to run code Day 1? Use a container!
5. Have a short term student on a project? Use a container!

Chapter 5: Putting it all Together

Exercise #5: Gadget-4

- Let’s run a test simulation using Gadget!
- Together we’re going to:
  a. Build a Docker Image
  b. Build the Software with OpenMPI Multiprocessing
  c. Use Apptainer to Run it on the HPC
  d. Write a SLURM script to submit the job
  e. Write a SnakeMake file to automate this process?