Introduction to High Performance Computing and the Linux Command Line

Ian Percel

University of Calgary, Research Computing Services

September 14, 2021

Who is here?

- Who has programmed before?
- Who has used Linux before?
- Who has used a Job Scheduler before?

Some simplifications and complexities

- We will focus on multi-user, Linux-based computing clusters with job schedulers
- We will not discuss virtual machines or graphical desktop based interfaces
- We will not discuss commercial cloud computing services

Outline

Where we are going

- Introduce some of the basic ideas of high-performance computing and how it differs from personal computing
- Discuss computing at the command line
- Develop ideas about communicating with a job scheduler

Where we are going

By the end of this training, you should be able to

- Login to the cluster
- Transfer data to and from the cluster
- Navigate your files on the cluster
- Write shell scripts
- Communicate with the job scheduler

A Simple Model for Computing

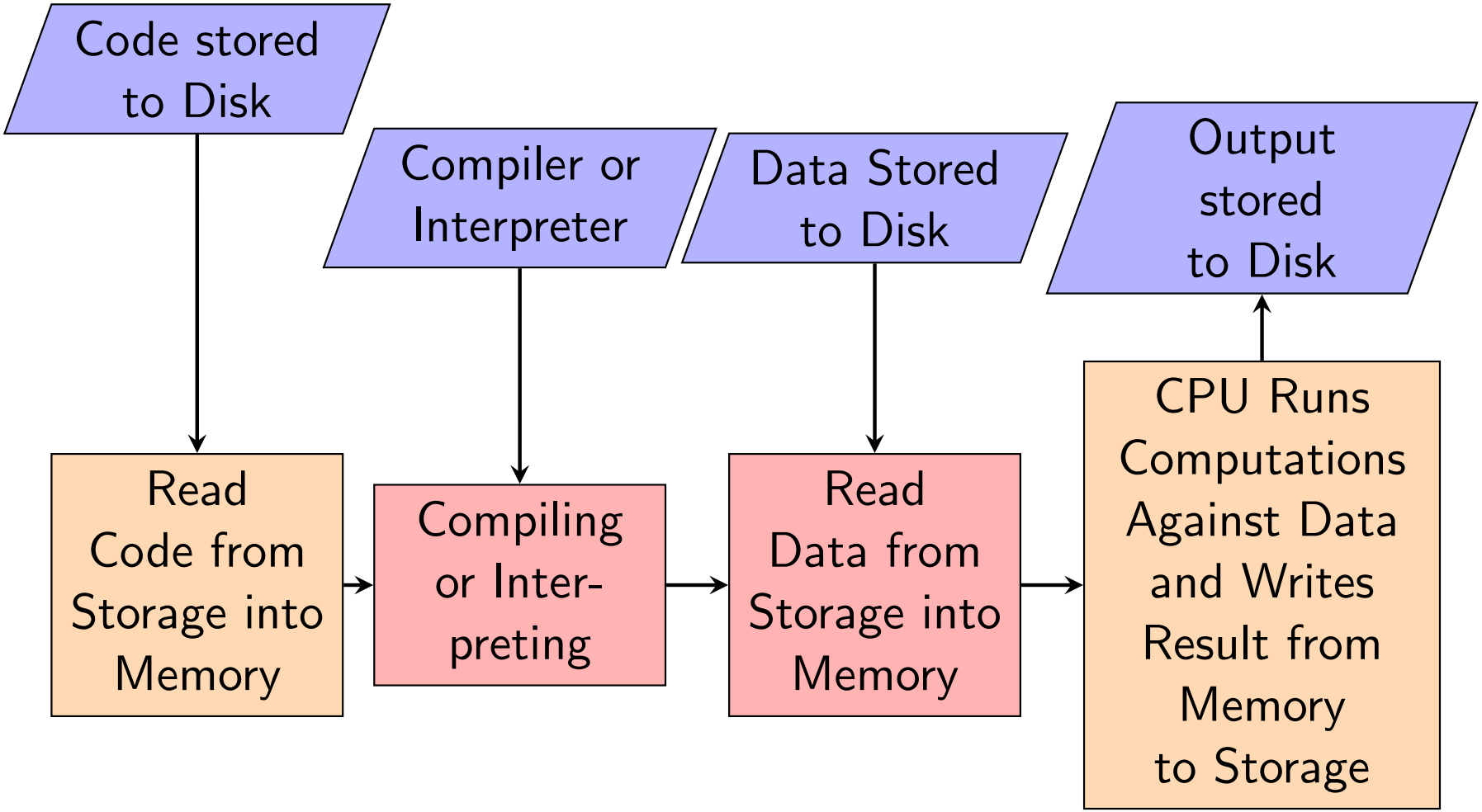


Figure 1: Computing Diagram

A Simple Python Program

```
$cat myCode.py

import pandas as pd

newdf=pd.read_csv("~/myData.txt")
output=newdf.mean()
output.to_csv("~/myOutput.txt")
```

To execute from command line:

```
$ /anaconda3/bin/python myCode.py
```

A Simple Model for Computing Applied to Python

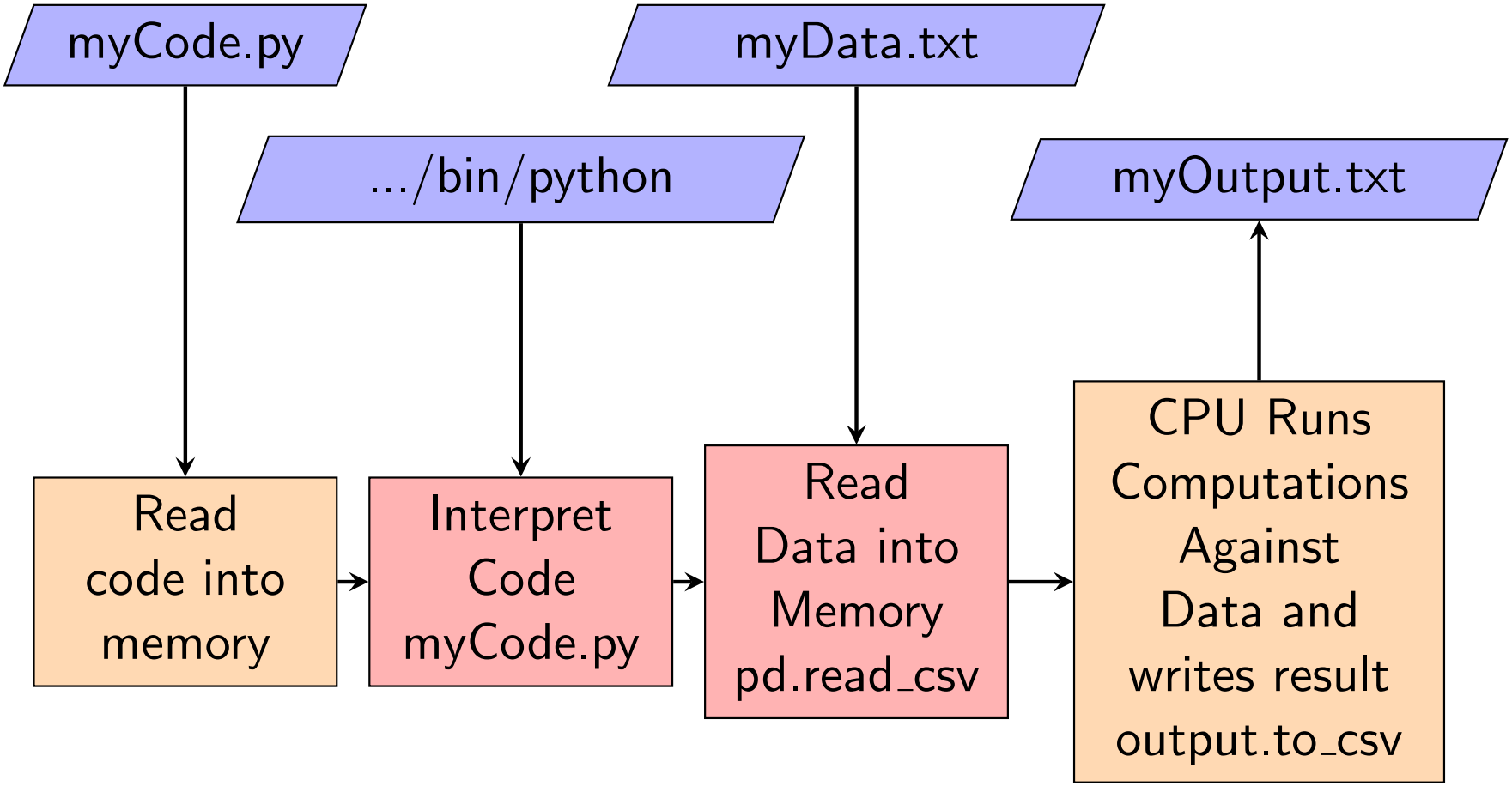


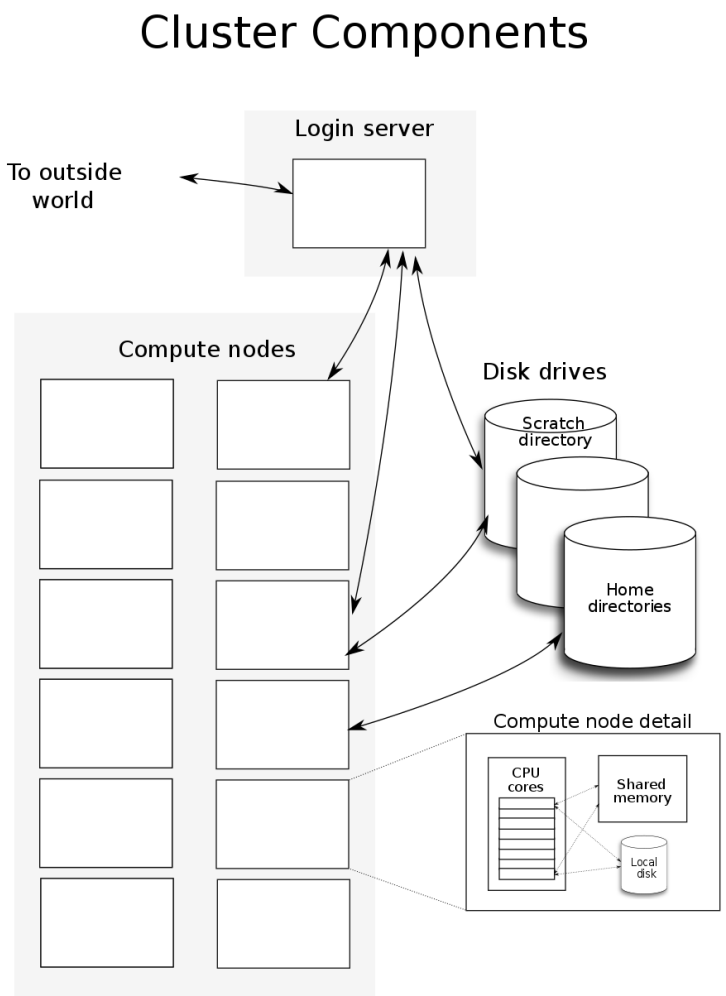
Figure 2: Computing Diagram for Running Python Code

Implications of this model

When you start work on an HPC system, there are several things to keep in mind

- What storage will your data live on and how will you get it there?
- Who will have access to your data? What kind of data organization system will you use to keep your work organized?
- What storage will the software that includes instructions for your computations live on?
- What kind of computational resources memory and processing will be required for your computation?
- If multiple computing resources will be working in parallel how will they communicate?
- What storage system will the output of your computation be written to? Where will it go in the file system?

Cluster Architecture



How are Beowulf-type clusters different from personal computing?

- Moving from a personal computer to a high-performance computing cluster is akin to moving from a small optical telescope in your backyard to a radio telescope array. The methods and workflow are drastically different and not merely bigger.
- Different techniques for structuring a problem (decomposing it into sub-problems)
- Different techniques for organizing data and computing work
- Different techniques for integrating sub-problems back into a satisfactory whole
- Different techniques for performance analysis

HPC Workflow Diagram

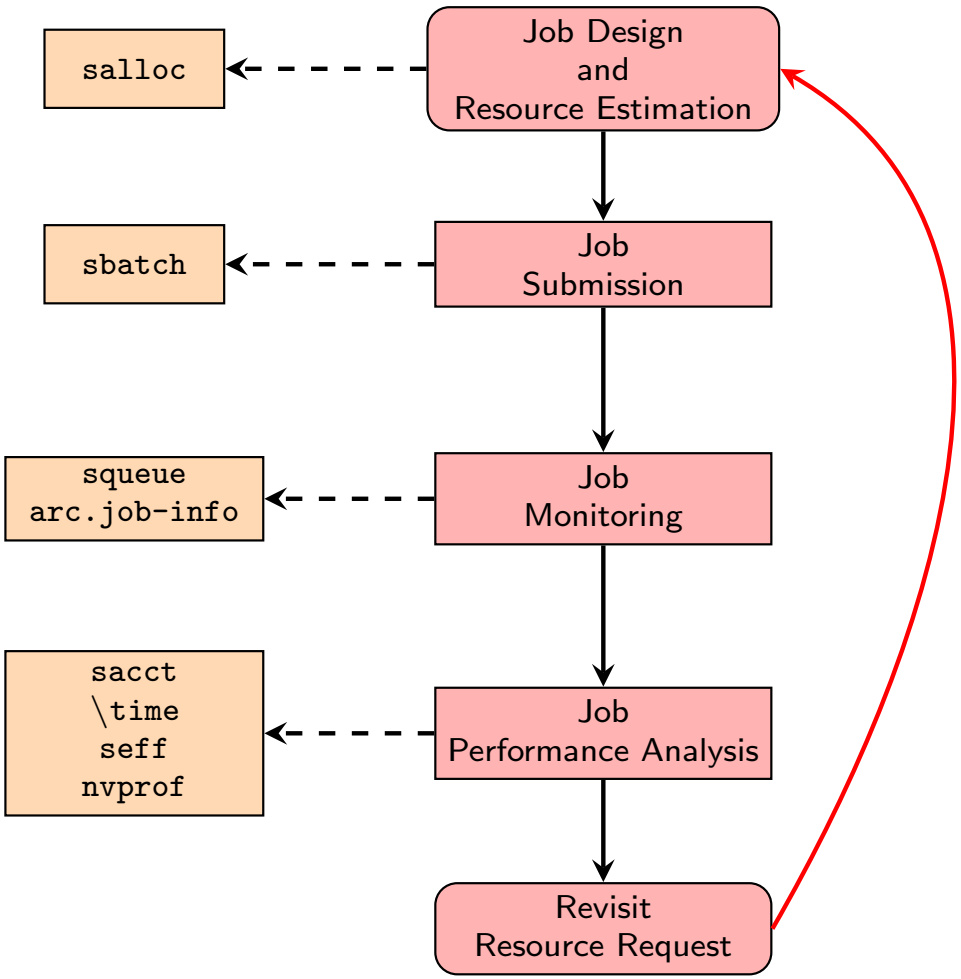


Figure 4: Compute Cluster Workflow with Job Scheduler

The Linux Command Line

Most HPC systems use some flavour of Linux as an operating system. At the University of Calgary, our clusters are currently running on CentOS 8. Computing on a cluster can involve some limited graphical components in script writing and postprocessing, but most of the work will require a basic understanding of the Linux Command Line.

- Text-only interface
- Command prompt usually takes the form of a snippet of information (for example, about the user and the computer they are working on) followed by a $
- Interacting with the system involves typing commands at the command prompt and then pressing the return key to submit them to the system
- There is an interpreter program (the default on arc is bash) that turns the commands that you submit into system processes and returns the result to your screen

Sample Command Line Interaction

```
[ian.percel@arc ~]$ ls
scripts miniconda3 data
[ian.percel@arc ~]$ cd data
[ian.percel@arc data]$pwd
/home/ian.percel/data
```

Frequently Used Commands

Most commands are calls to small programs that do limited tasks. Among these there are a handful that are required for day to day work. A solid foundation for users of such a system can come from reading the first 4 chapters of "The Linux Command Line" or a similar text. [1] The most common commands for everyday work on ARC fall into 4 categories:

- Filesystem navigation and file management
- Text processing and editing
- Configuring the environment
- Job submission and management

Structure of Commands

Most commands take a standard form

```
command [subcommand] [-option argument | --option=argument]
```

- The command will be the name of a program
- Sometimes there is a subcommand or argument that specifies a subset of functionality or tells what files to apply the command to
- Sometimes there are options with possible arguments that change the behaviour
- Bash parses on spaces, so be careful to quote text with whitespace if you need to use them in an argument
- `man command` will generally tell you everything you need to know about options, subcommands, and arguments

Filesystem Model

- A given session has a location in the file system at any given time
- This location can be changed
- The filesystem is a tree (parent $\rightarrow$ children)
- The root of the tree is /
- Locations in the file system are named by paths = set of coordinates to the location
- Absolute paths are unique and independent of your location
- Relative paths are possibly non-unique and depend on your location

Filesystem Commands

Filesystem commands are used to find and manage files and directories

- `ls` : list files and directories
- `cd` : change your directory in the file system
- `mkdir` : make a new directory in the file system
- `cp` : copy a file or directory to a new location in the file system (leaving the old one intact)
- `mv` : move a file or directory to a new location in the file system (without leaving it in the old location) or rename it
- `find` : search for files or directories or their contents by name or name pattern (very versatile)
- `rm` : remove a file or directory

Filesystem Practice

Problems

1. inspect the contents of your home directory
2. create a new directory in your home directory called `test`
3. create a new directory in `test` called `test2`
4. using `touch` create a new file in `test` named `example.slurm`
5. copy the file into `test2`
6. rename the file in test to `example2.slurm`
7. move the newly renamed file into `test2`
8. from your home directory, inspect the contents of test2 using both relative and absolute paths (how are they different?)
9. change your location to inside `test` and repeat the previous problem (what has changed in the commands?)

Text Processing Commands

Text processing commands are used to example, search, parse, replace, and edit text. Since all commands are submitted to the interpreter as text, this is an extremely important class of tools

- `less` : inspect but do not edit a text file
- `head`,`tail` : send the first (last) n (default 5) lines of a text file to the screen
- `cat` : with single file name, print contents to the screen; many more advanced uses are possible
- `vim`, `emacs`, `nano` : command line text editors
- `sed`, `awk` : efficient text processing scripting languages (advanced technique)
- `grep` : filter lines in a file using regular expression

Text Processing Practice

Problems

1. examine the contents of the file `example.slurm`
2. open the file `example.slurm` that you just created using a text editor
3. add the lines `#!/bin/bash` and `echo $(date)`
4. examine the contents of the file using `less`
5. print the first line of the file to the screen using `head` and an option
6. print the last line of the file to the screen using `tail` and an option

Commands, Software, and the Environment

The use of commands at the linux command line (excepting bash builtins) is simply a reference to a piece of software on the system. New commands can be made available to you simply by installing new software or accessing software not currently findable by your shell. What does this involve?

1. compile or download software for linux_x86-64
2. move the resulting binary files somewhere that is easy to find and give them executable permissions
3. change your environment variables to make it possible for your terminal to find them

The name of the file is the command that you will use at the command line in the future. Environment variables are the key to accessing software on the cluster and to modifying the default behaviour of your terminal session.

Commands for Configuring the Environment

Most configuration options in the terminal (such as number of cores to use in parallel, locations to look for software, and locations to find libraries) are set using environment variables.

- `printenv` : show the values of all environment variables
- `echo $VARIABLE_NAME` : display the value of a specific variable
- `VARIABLE_NAME=value` : set a variable to a text value
- `export VARIABLE_NAME` : make a variable available to child processes
- `VARIABLE_NAME=value$VARIABLE_NAME` : prepend a value to the current value of a variable
- `unset VARIABLE_NAME` : remove the current value from the variable so it is like it was never set

Commands for Configuring the Environment

Important Variables:

- `PATH` : colon delimited list of locations to search for software
- `LD_LIBRARY_PATH` : colon delimited list of locations to search for shared libraries used at runtime
- `OMP_NUM_THREADS` : number of threads to use in a shared memory parallel process
- `USER` : your username

Related Commands:

- `which programName` : search `PATH` from left to right and report the location that `programName` would run from
- `export PATH = value:$PATH` : manually force a location to be searched first for software
- `echo $PATH` : check the current value of PATH

Commands for Configuring the Environment

There are some special commands on HPC systems that auto-populate some special environment variables

- `module avail` : list available software modules
- `module load moduleName` : load a specific module
- `module list` : list loaded modules
- `module purge` : clear loaded modules
- `module unload moduleName` : clear a specific module

Environment Setup Practice

Problems

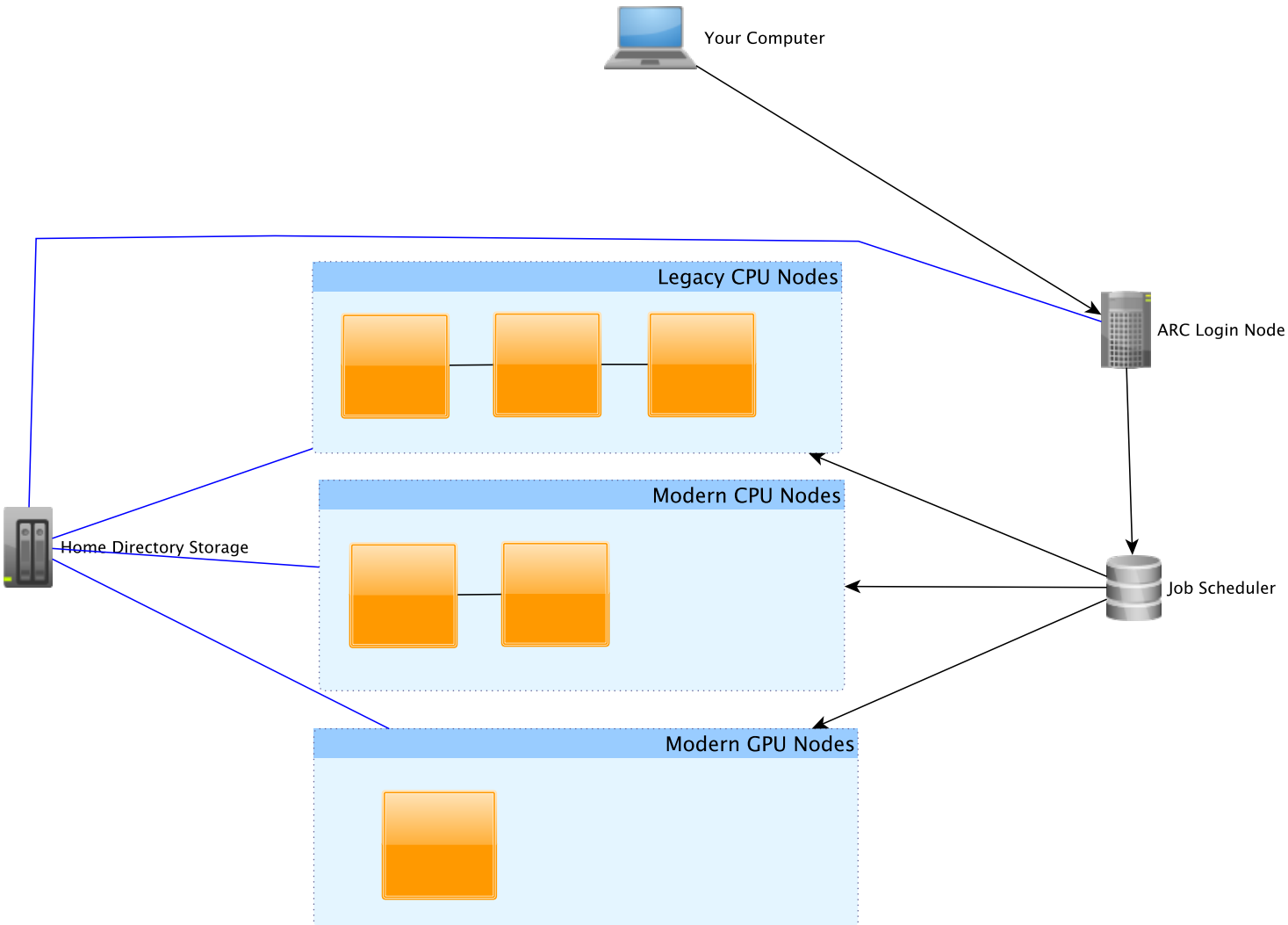
1. check the current state of all set environment variables
2. check the available modules
3. check where `python` would currently be found using `which`
4. check where `python3` would currently be found
5. load a python module
6. check where `python` would currently be found
7. check where `python3` would currently be found
8. compare the old value of the PATH variable with its current value and confirm the presence of a `python` binary in the previously reported location and not in any location to its left in the list

What is a Job Scheduler?

A job scheduler, like SLURM, is a long-running process that you can communicate with to schedule access to resources. It performs the following services for you

- Coordinates your use of resources with other users to avoid collisions
- Identifies the most appropriate resources for a job given a specific resource request
- Enforces resource requests and ensures fair access for all users
- Manages processes and the sets up the environment required by a job for you
- Tracks resource usage to help in estimating requirements for future work

Job Scheduler Diagram



Communicating with SLURM

Jobs are submitted to a job scheduler (SLURM) and managed from the command line using a small number of commands with a large variety of options

- `sinfo` , `arc.nodes` : show the current state of the cluster
- `sbatch jobscript.slurm` : submit a batch job to the scheduler
- `salloc [resource request]` : request an interactive job for light-weight debugging or postprocessing
- `squeue -u $USER` : check the status of your queued jobs

Communicating with SLURM

Jobs are submitted to a job scheduler (SLURM) and managed from the command line using a small number of commands with a large variety of options

- `sacct -j jobID -o JobID,MaxRSS,Elapsed` : check the total wall time and memory used by a given job (without -o option get basic information about job completion)
- `arc.job-info jobID` : check the real time resource utilization of a job
- `seff jobID`: check resource utilization by the job
- `nvprof`, `time` : used inside a slurm script to capture resource usage of a particular command for GPUs or CPUs on a single node respectively
- `scancel` : safely kill a job or remove it from the queue

HPC Workflow Diagram

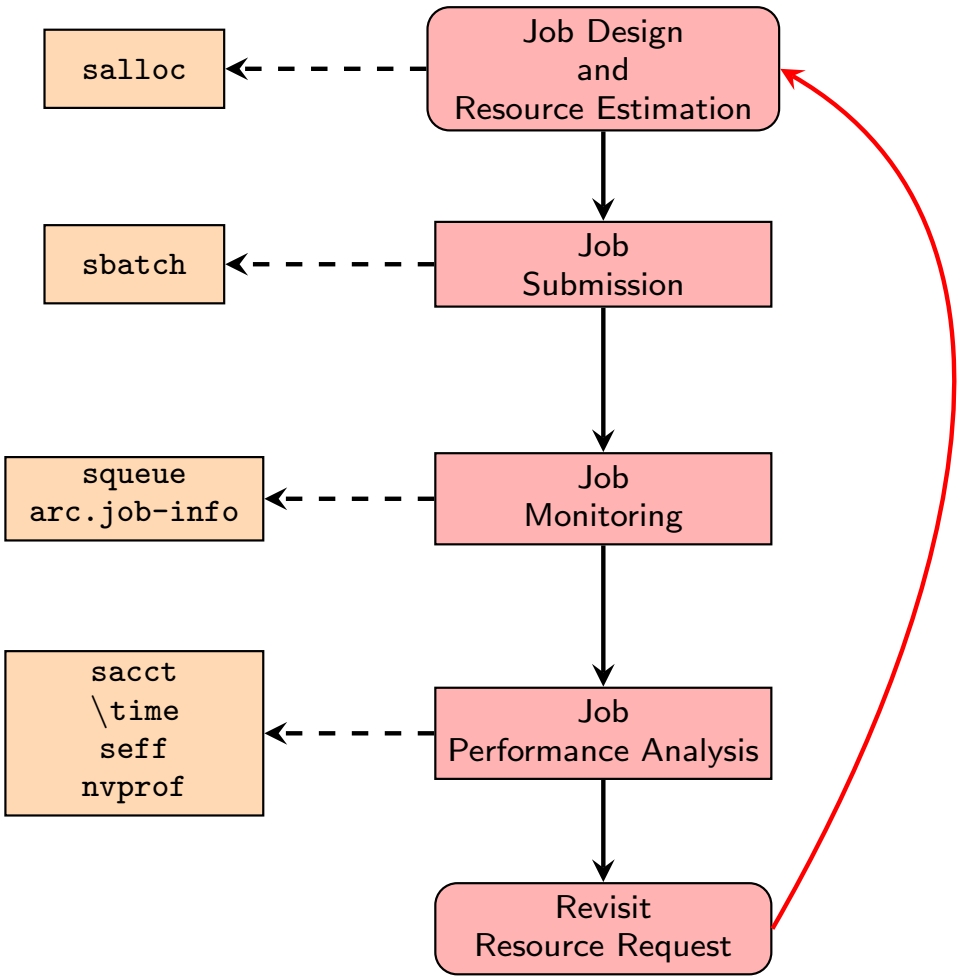


Figure 4: Compute Cluster Workflow with Job Scheduler

Sample SLURM Interaction

```
$vim myJob.slurm
$sbatch myJob.slurm
Submitted batch job 7853383
$squeue -j 7853383
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           7853383 cpu24 example1 ian.perc PD       0:00      1 (Priority)
           ...
$squeue -j 7853383
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           7853383 cpu24 example1 ian.perc R       1:01      1 b14
           ...
$squeue -j 7853383
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
$sacct -j 7853383
       JobID    JobName  Partition    Account  AllocCPUS      State ExitCode
------------ ---------- ---------- ---------- ---------- ---------- --------
7853383         example1      cpu24       ian.perc         24  COMPLETED      0:0
7853383.batch        batch                  ian.perc         24  COMPLETED      0:0
7853383.extern      extern                  ian.perc         24  COMPLETED      0:0
```

Understanding a Job Script

A Job script has three required components

1. `#!/bin/bash` or an equivalent indication of where to find the bash interpreter
2. a resource request (i.e. your contract with the scheduler)
3. a shell script (i.e. instructions in the form of bash commands for what to do when the job starts)

Sample Job Script: myJob.slurm

```
#!/bin/bash
#SBATCH --job-name=example1
#SBATCH --partition=cpu24
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=24
#SBATCH --mem=255000
#SBATCH --time=5:0:0

SCRIPTDIR=/home/ian.percel/scripts/

echo "Code starting at:"$(date)
export PATH=/home/ian.percel/anaconda3/bin:$PATH
echo "Path:"$PATH
echo "Script Directory:"$SCRIPTDIR
echo "scriptPath:"$SCRIPTDIR"mySamplePython.py"

python $SCRIPTDIR"mySamplePython.py"
```

[1] William E. Shotts Jr. *The Linux Command Line: A Complete Introduction*. No Starch Press, 2012.