High Throughput Workflow Tools and Strategies

User Training Seminar
May 30, 2025

Bill Arndt
Data Science Engagement Group
warndt@lbl.gov

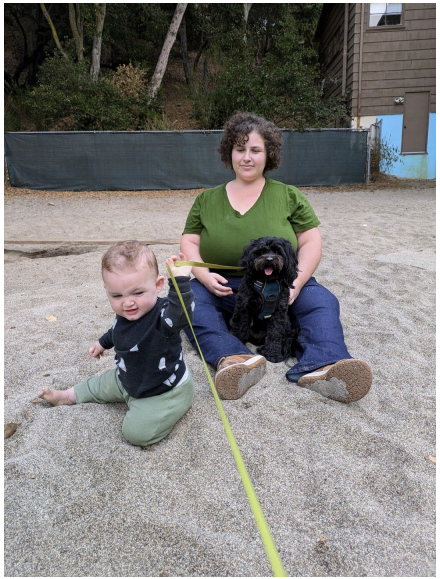
Agenda

- Introductions and Context
- GNU Parallel
- Snakemake
- Hypershell

Bill Arndt

UNIVERSITY OF South Carolina

hhmi | janelia Research Campus



- Arrived at NERSC, January 2016
  - Optimized HMMER3 for HPC
  - NESAP - E3SM Ocean Core
  - JGI Consulting Team
  - Urgent and Interactive HPC Workshop Series, Organizing Committee

Geoffrey Lentner, introduce yourself

Expected User Environment

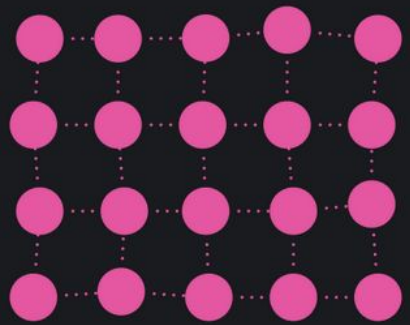
- This training assumes the audience…
  - Are competent users of a Linux terminal
  - Have reason and opportunity to access and use a HPC Linux cluster of fair to large size
  - obtain resources on that cluster via a batch allocation system such as Slurm, PBS, Torque, Flux, etc

What is HTC?

High-Performance Computing

Coherent Coupled Tasks with Communication

Monolithic simulation where tasks run simultaneously and define some physical or abstract space, working together to solve one problem.



High-Throughput, *Many-Task* Computing

Independent Tasks without Communication

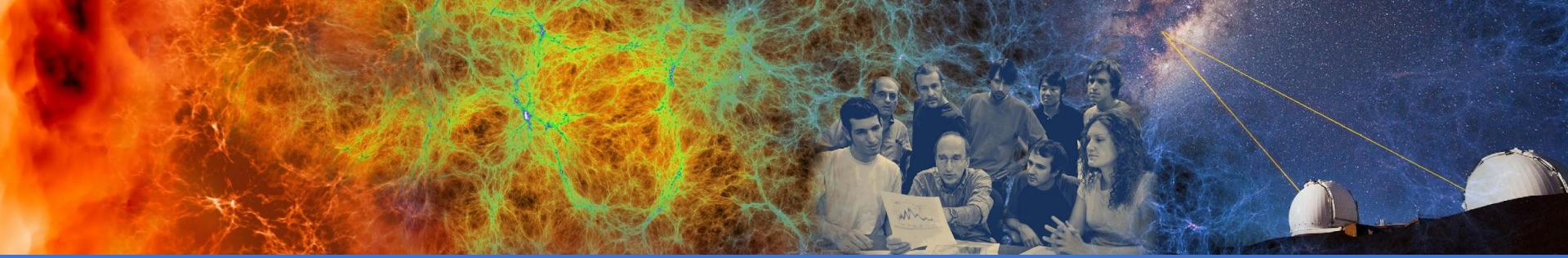
Defined by the number of tasks completed or volume of data processed. Elements of the workflow are entirely independent and may run anywhere, even across administrative boundaries.

Misalignment Between HPC and HTC

- Job scheduling infrastructure designed a small number of large HPC jobs can be overwhelmed by large numbers of small jobs.
  - Some things that feel like should help actually don't; i.e. each element of a job array uses the same resources as a regular job
- Queue policies and wait times tend to favor fewer, larger jobs
- System defaults chosen for different workload types
- "The cluster spanning high speed network is cool and expensive, and you're not using it!"

Fitting HTC on HPC

- Strategy: Use workflow management tools to change many small jobs into fewer large jobs.
- Different tools trade off between ease of use and power.
  - This training begins with the most accessible tool, then focuses on the capabilities gained by using other methods which require more setup and overhead.
- Workflow tools are composable
  - Choosing the right tool and features for each role in your workflow



GNU Parallel

Installation

- It’s simply a 14k line Perl script.
  - https://www.gnu.org/software/parallel/
  - Download, decompress, place in a $PATH location
  - Official packages available for at least 24 platforms
  - Your cluster (such as Perlmutter) may already have it installed or in a module

What Is GNU Parallel

- Use a template to assemble shell commands and run them in parallel.
- It does not:
  - Take responsibility for the ordering the commands run.
  - Interact with multiple nodes, schedulers, or MPI.

Example

```
parallel echo "three colon argument {}" ::: 4 5 6 7 8
three colon argument 4
three colon argument 5
three colon argument 6
three colon argument 7
three colon argument 8
```

Example

```
parallel echo "combine {}" ::: 1 2 ::: a b
combine 1 a
combine 1 b
combine 2 a
combine 2 b

parallel echo "combine {}" ::: 1 2 :::+ a b
combine 1 a
combine 2 b
```

Parallel One Liners

- This is the most accessible workflow tool; useful tasks can be accomplished with single commands and no other preparation.
  - Preparing lots of directories for data
  - Do a command to every file in a directory

```
~/tutorial/htc_training/parallel/00_command_only> parallel echo {} ::: $(ls)
blah.txt
data
example_one_liners.sh
expected_output.txt
README.md
```

Task List from File

```
:~/tutorial/htc_training/parallel/01_file_inputs> cat tasks.txt
6
7
8
parallel echo "mixing types and + argument {}" ::: $(seq 1 100)::::+ tasks.txt
mixing types and + argument 1 6
mixing types and + argument 2 7
mixing types and + argument 3 8
```

- Can also deliver a task list from:
  - stdin redirect
  - pipe operator
  - giving a file path with wildcards to a ::: operator

Substitution Modifiers

```
parallel "cat {} > {//}/{/.}.output" :::: list_of_file_paths
```

- Annotate inside the brackets to modify the task substitution:
  - `{.}` Remove file extension
  - `{/}` Remove directory path
  - `{//}` Remove filename and extension
  - `{./}` Filename only
  - `{#}` Task number
  - `{number}` If column separator flag is used, substitute content of corresponding column. Also works with multiple input lists.

Tasks from .csv

```
cat input.csv
"Bob","18","Cherry"
"Alice","9","Apple"
"Bort","1","Pipevine"

parallel --colsep="," echo {1} {3} :::: input.csv
"Bob" "Cherry"
"Alice" "Apple"
"Bort" "Pipevine"
```

- Pitfall: parallel has a --csv flag, but it depends on a Perl module which is not installed on Perlmutter

Many Tasks Inside a Single Node Job

```
#!/bin/bash
#SBATCH --qos=debug
#SBATCH --nodes=1
#SBATCH --constraint=cpu
#SBATCH --time=00:02:00

srun parallel --jobs 6 ./payload.sh argument_{} :::: input.txt
```

- Replace the example payload.sh with your application

Many MPI Tasks and Multiple Node Job

```
#!/bin/bash
#SBATCH --qos=debug
#SBATCH --nodes=1
#SBATCH --constraint=cpu
#SBATCH --time=00:02:00

srun parallel --jobs 6 ./payload.sh argument_{} :::: input.txt
```

- Replace the example payload.sh with your application

Many Tasks Inside a Single Node Job

```
~/tutorial/htc_training/parallel/05_many_mpi_tasks> cat driver.sh
#!/bin/bash
parallel --delay 3 --jobs 4 srun -c 64 -n 2 --distribution=block,pack \
--network=no_vni ./payload.sh {} ::: $(seq 1 4)
```

- srun is our MPI launcher, parallel uses one per each of many MPI applications
- the --delay 3 prevents flooding the Slurm controller
- the --distribution flag stops the default behavior of separating the MPI procs to different nodes, even if they can all fit on the same one
- the --network flag stops slurm from counting network adapters as a limiting resource, which on Perlmutter would prevent more than 4 applications per node
- “Why not use a loop?”
  - Backgrounding multiple sruns to run in parallel is some work
  - You’re getting value from the parallel substitution brackets

What Goes Wrong

- GNU Parallel tasks use some temporary file handles; it's possible to hit the ulimit by running *a lot* of tasks at the same time.
- Some situations (ex. file path wildcards) can lead to "Argument list too long" errors.
  - The -X flag can help if the payload is flexible enough, or find a way to filter or subdivide the wildcard expansion

“Traps”

- Like all workflow management tools, GNU has some features which are probably best avoided.
  - --ssh can also utilize multiple nodes, but the narrow bandwidth limits scaling to 10’s of concurrent tasks
  - Perl regular expressions can be used to format and filter tasks. They’re cool and powerful but not readable; anyone inheriting your script will probably be annoyed.
  - --sql master and worker don’t store job locks or state information so they don’t add any capabilities beyond a simple task list file

Many Tasks Inside a Many Node Job

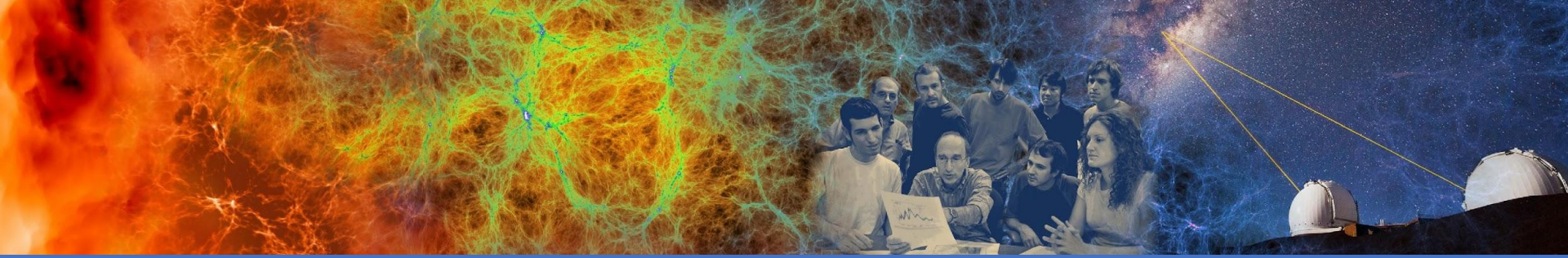
```
#!/bin/bash
#SBATCH --qos=debug
#SBATCH --nodes=2
#SBATCH --constraint=cpu
#SBATCH --ntasks-per-node 1
srun --no-kill --ntasks-per-node=1 --wait=0 driver.sh $1
```

- The first step is to use srun to fan out one instance of GNU parallel to each node in the job
- The srun default is to assume this is an MPI job.
  - If one process ends it assumes they are all finished and ends the job unless --wait=0 is used.
  - If one node fails, it assumes the entire job is ruined and ends it, unless --no-kill is used.

Many Tasks Inside a Many Node Job

```
cat $1 |                                                    \
awk -v NNODE="$SLURM_NNODES" -v NODEID="$SLURM_NODEID"   \
'NR % NNODE == NODEID' |                                    \
parallel ./payload.sh argument_{}
```

- The file listing all job tasks is passed to an awk script that assigns each task to a single node using round robin distribution.
- The parallel instance running on each node receives a pipe with tasks assigned to it, and runs them.
- This works, but demonstrates pushing a tool outside of its strength
  - The task assignments are static; load cannot be balanced between nodes.
  - Using the srun on the outer script means the payloads cannot easily use MPI
  - It's less intuitive and readable; someone inheriting this workflow may also be annoyed



Snakemake

Installation

- Python based tool
  - https://snakemake.readthedocs.io/en/stable/getting_started/installation.html
  - Simplest method is via Anaconda
    - `conda create -c conda-forge -c bioconda --name snakemake_env snakemake`
  - Perlmutter provides the Anaconda infrastructure via
    - `module load python`
  - Pip is also an option
    - `pip install snakemake`

What is Snakemake

- GNU Make counts as workflow management tool.
- Many projects have expanded its model to be more useful for data processing. Snakemake is one of them.
- The primary new capability provided by Snakemake is management of ordering and dependencies between tasks.
- The additional cost is more effort needed to create workflow scripts and a heavy reliance on file systems to control running workflows and store state.

Snakemake Details

- All data and workflow state are files.
- The user writes rules in Python which describe how to transform input files into output files.
- Run the workflow by telling it what file you want; snakemake evaluates the rules and works backwards to figure out how to make the wanted output with the files that are available.
- All logs get saved in a hidden .snakemake directory

Snakemake Hello World

```
:~/tutorial/htc_training/snakemake/00_hello_world> cat Snakefile
rule all:
    input: "hello_world.txt"

rule hello_world:
    output: "hello_world.txt"
    shell: "echo Hello World! > hello_world.txt"
```

- The first rule in a Snakefile is the target output file.
- The command to run this workflow is `snakemake --cores 1`
  - If there is a file named Snakemake in the current directory it will be used by default
  - the cores flag sets how many cores to use

Simplest Dependency

```
rule all:
     input: "cool_hello_world.txt"

rule make_cool:
     input: "hello_world.txt"
     output: "cool_hello_world.txt"
     shell: "echo xXxXx $(cat hello_world.txt) xXxXx > cool_hello_world.txt"

rule hello_world:
     output: "hello_world.txt"
     shell: "echo Hello World! > hello_world.txt"
```

Wildcards

```
from pathlib import Path
name_list = Path("data").iterdir()
name_list = [file.with_suffix("").name for file in name_list]

rule all:
    input: expand("output/cool_{name}.txt", name=name_list)

rule make_cool:
    input: "data/{name}.txt"
    output: "output/cool_{name}.txt"
    shell: "echo xXxXx $(cat data/ {wildcards.name} .txt) xXxXx > output/cool_ {wildcards.name} .txt"
```

Parallel + Snakemake

```
#!/bin/bash
parallel "snakemake --quiet --cores 1 output/cool_{/.}.txt" ::: $(ls data | grep txt)

rule make_cool:
     input: "data/{name}.txt"
     output: "output/cool_{name}.txt"
     shell: "echo xXxXx $(cat data/{wildcards.name}.txt) xXxXx > output/cool_{wildcards.name}.txt"
```

- Workflow tools are composable, which grants opportunities to pick and combine the strengths of each
  - Adding parallel enables the removal of the Pathlib code, and even the base rule
  - Snakemake handles dependency resolution, the modification commands, and logging