High Performance Python

Improving code efficiency and performance

TJ Langford - Yale Center for Research Computing

2024-12-12

Outline

- **Part 0: A word about Python packages and environments**
- **Part 1: Why is my code slow?**
- **Part 2: Resource Management and Monitoring**
- **Part 3: Parallelism locally and on the clusters**
- **Part 4: Accelerating Python with advanced packages**

Python Packages and Environments

Conda, venv+pip, and software-modules

Conda (https://conda-forge.org/)

- Python package manager with massive number of available tools (including non-python dependencies)
- Simple to install, no need for admin privileges
- Compartmentalization of “environments” for simple project management

Virtual Environments and pip

- Use system python as a base to create a “venv”
- Install new python packages with pip
- Can be tucked directly inside project code-spaces

Software modules

- Optimized installations on the clusters built from source
- Less flexible, but can be more performant

Python Packages and Environments

Conda, venv+pip, and software-modules

Conda (https://conda-forge.org/)

- Python package manager with massive number of available tools (including non-python dependencies)
- Simple to install, no need for admin privileges
- Compartmentalization of “environments” for simple project management

Virtual Environments and pip

- Use system python as a base to create a “venv”
- Install new python packages with pip
- Can be tucked directly inside project code-spaces

Software modules

- Optimized installations on the clusters built from source
- Less flexible, but can be more performant

Python Packages and Environments

Build an environment for this workshop

```
conda create --name hpp python jupyter
matplotlib numpy scipy pandas dask distributed
dask-jobqueue numba cupy scalene
```

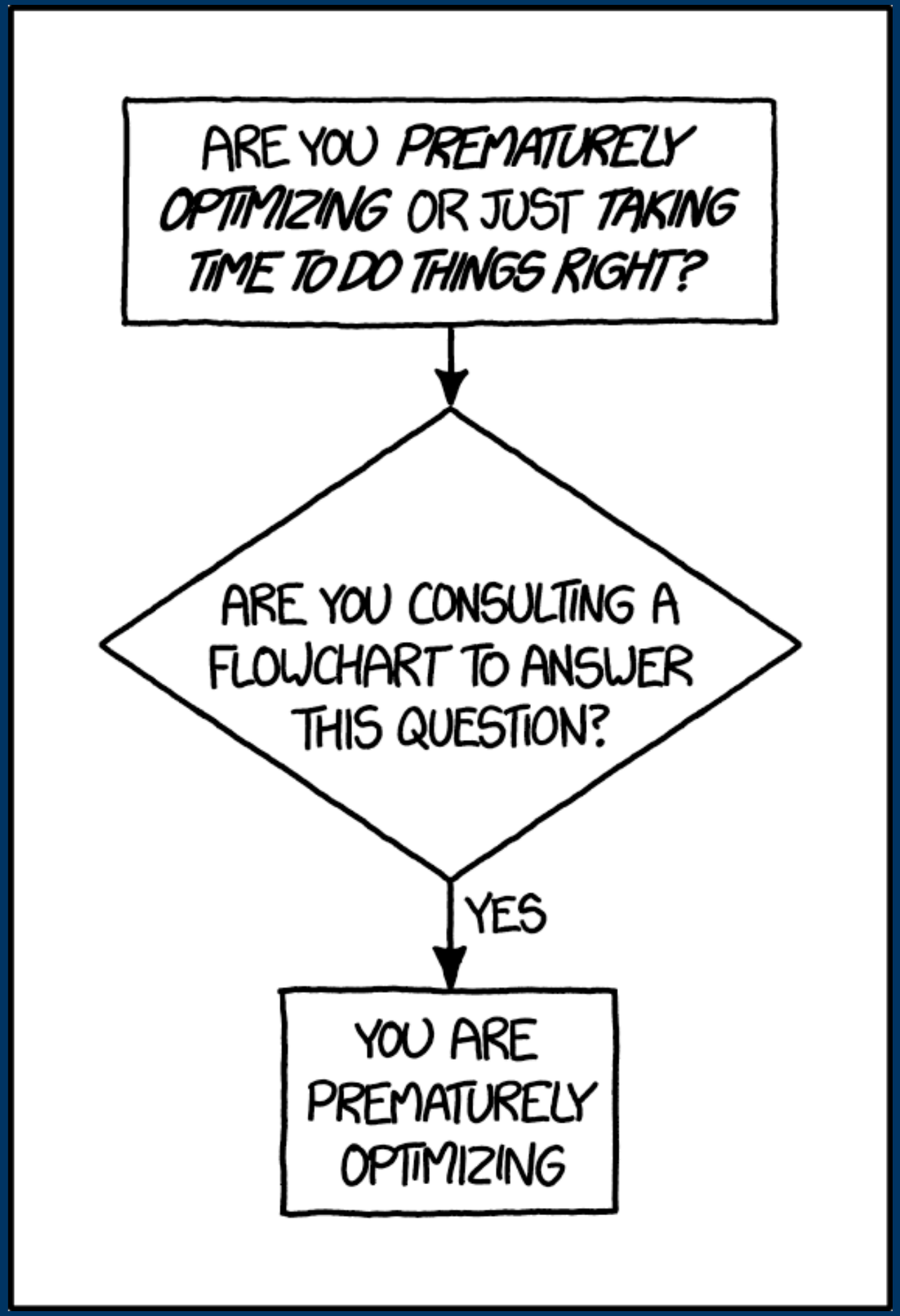
- On personal machine, install miniconda from conda-forge
- On the clusters, `module load miniconda` and run install command on compute node

Presentation Materials

github.com/ycrc/high_performance_python

Part 1:

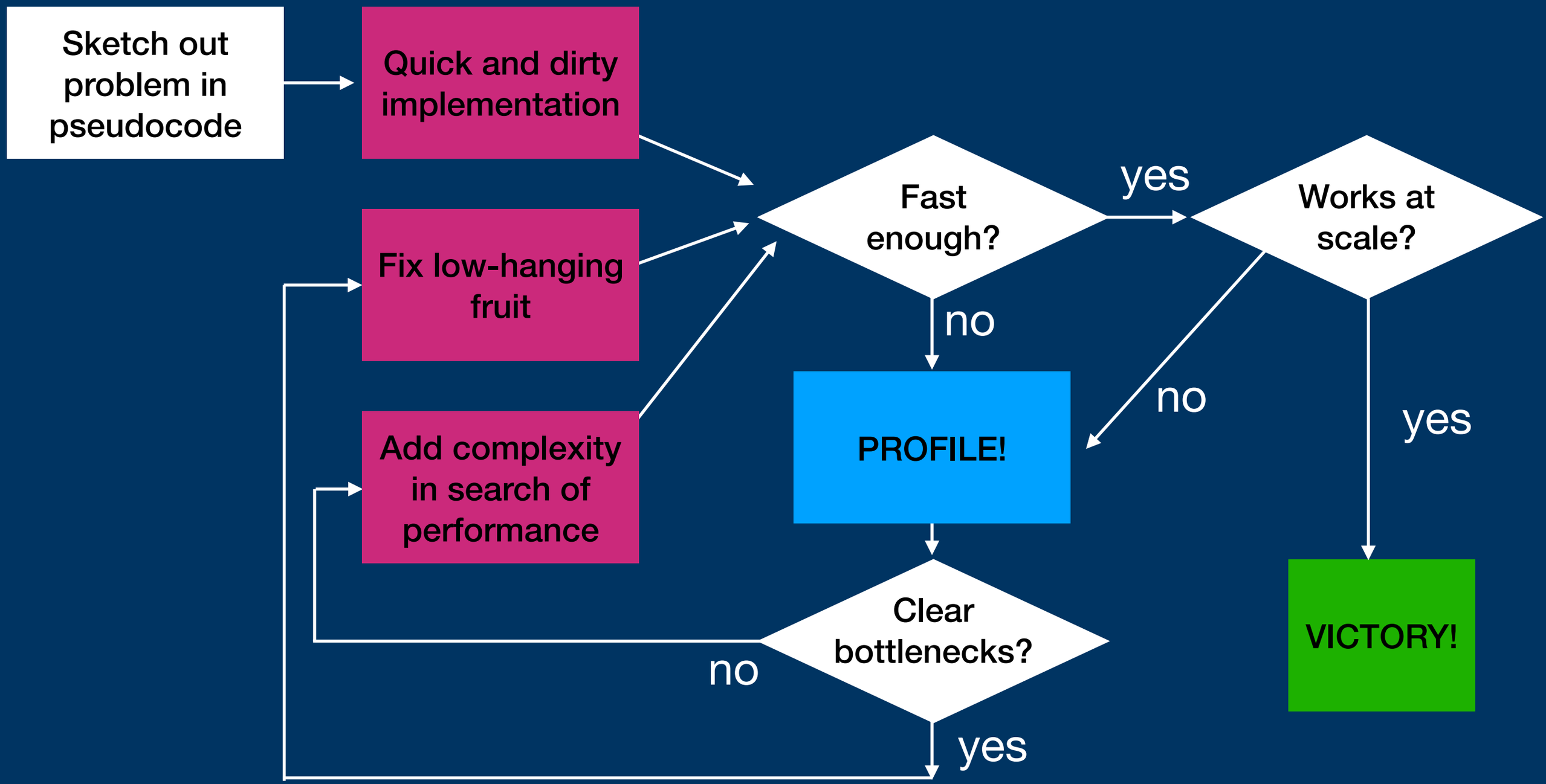
Why is my code slow?



https://xkcd.com/1691/

Better flow-diagram

Programmer time is expensive…



Code Profiling

“~~Why~~ *Where* is my code slow”

- Before any optimization is done, first identify **where** your code is spending all its time
- Only improve sections that contribute significantly to the overall run-time
  - Don’t focus on a slow function that’s only called once
  - Instead improve the slightly inefficient loop that’s called 1M times
- Several “profilers” available for Python
  - kernprof, cprofiler, scalene

Code Profiling

“~~Why~~ *Where* is my code slow”

- Before any optimization is done, first identify where your code is spending all its time
- Only improve sections that contribute significantly to the overall run-time
  - Don’t focus on a slow function that’s only called once
  - Instead improve the slightly inefficient loop that’s called 1M times
- Several “profilers” available for Python
  - kernprof, cprofiler, scalene

Code Profiling

Scalene: https://github.com/plasma-umass/scalene

- CPU, Memory, and GPU profiler for Python
- Fast and low-overhead sampling profiler
- Easy to install on any platform (conda install scalene)
- Can be run inside jupyter notebooks or as a standalone tool
- Generates a CLI report or a json that can be uploaded to a web-GUI (http://plasma-umass.org/scalene-gui/) for more interactive inspection
- Given an OpenAI key, will generate suggestions to improve performance

Profiling Example

Lists vs Numpy arrays

File: **list_example.py**

```python

def main(n):

    print(f'{n} elememement list')

    list_1 = []
    list_2 = []

    for i in range(n):
        list_1.append(i)

    for i in list_1:
        list_2.append(i * i)

    return list_2


if __name__ == "__main__":
    from sys import argv
    main(int(argv[1]))
```

File: **array_example.py**

```python
import numpy as np

def main(n):

    print(f'{n} elememement list')

    array_1 = np.arange(0,n)

    return np.pow(array_1, 2)


if __name__ == "__main__":
    from sys import argv
    main(int(argv[1]))
```

Profiling Example

Lists vs Numpy arrays

```
(hpp)[tl397@r807u31n01.grace numpy]$ scalene list_example.py 10000000
10000000 elemement list
                    Memory usage:                      (max: 777.406 MB, growth rate:   0%)
   /vast/palmer/home.grace/tl397/ycrc/workshops/high_performance_python/numpy/list_example.py: % of time = 100.00% (2.402s) out of 2.402s.

     | Time   |        |        | Memory |      |            | Copy   |
Line | Python | native | system | Python | peak | timeline/% | (MB/s) | /vast/palmer/home.grace/tl397/ycrc/workshops/high_performance_python/numpy/list…
-------------------------------------------------------------------------------------
   1 |        |        |        |        |      |            |        |
   2 |        |        |        |        |      |            |        | def main(n):
   3 |        |        |        |        |      |            |        |
   4 |        |        |        |        |      |            |        |     print(f'{n} elemement list')
   5 |        |        |        |        |      |            |        |
   6 |        |        |        |        |      |            |        |     list_1 = []
   7 |        |        |        |        |      |            |        |     list_2 = []
   8 |        |        |        |        |      |            |        |
   9 |        |        |        |  100%  | 210M |        27% |        |     for i in range(n):
  10 |   26%  |    7%  |   17%  |  100%  | 174M |        23% |        |         list_1.append(i)
  11 |        |        |        |        |      |            |        |
  12 |        |        |        |        |      |            |        |     for i in list_1:
  13 |   33%  |    4%  |   13%  |  100%  | 387M |        50% |        |         list_2.append(i * i)
  14 |        |        |        |        |      |            |        |
  15 |        |        |        |        |      |            |        |     return list_2
  16 |        |        |        |        |      |            |        |
  17 |        |        |        |        |      |            |        |
  18 |        |        |        |        |      |            |        | if __name__ == "__main__":
  19 |        |        |        |        |      |            |        |     from sys import argv
  20 |        |        |        |        |      |            |        |     main(int(argv[1]))
  21 |        |        |        |        |      |            |        |
-------------------------------------------------------------------------------------
     |        |        |        |        |      |            |        | function summary for /vast/palmer/home.grace/tl397/ycrc/workshops/high_performa…
   2 |   59%  |   11%  |   29%  |  100%  | 387M |        73% |        | main

Top AVERAGE memory consumption, by line:
(1)     13:    387 MB
Top PEAK memory consumption, by line:
(1)     13:    387 MB
(2)      9:    210 MB
(3)     10:    174 MB
```

Profiling Example

Lists vs Numpy arrays

```
(hpp)[tl397@r807u31n01.grace numpy]$ scalene array_example.py 10000000
10000000 elemement list
                              Memory usage: ▁▁▂▂▃▃▁▁ (max: 158.773 MB, growth rate:   4%)
  /vast/palmer/home.grace/tl397/ycrc/workshops/high_performance_python/numpy/array_example.py: % of time = 100.00% (399.670ms) out of 399.670ms.

      │Time   │------ │------ │Memory │------ │------------ │Copy  │
 Line │Python │native │system │Python │peak   │timeline/%   │(MB/s)│/vast/palmer/home.grace/tl397/ycrc/workshops/high_performance_python/numpy/arra…
 ---------------------------------------------------------------------------------------------------------
    1 │    5% │   76% │    8% │  100% │   10M │ ▁      6%   │   87 │import numpy as np
    2 │       │       │       │       │       │             │      │
    3 │       │       │       │       │       │             │      │def main(n):
    4 │       │       │       │       │       │             │      │
    5 │       │       │       │       │       │             │      │    print(f'{n} elemement list')
    6 │       │       │       │       │       │             │      │
    7 │       │       │    3% │   90% │   72M │ ▂     46%   │      │    array_1 = np.arange(0,n)
    8 │       │       │       │       │       │             │      │
    9 │       │    4% │    2% │   18% │   76M │ ▃     48%   │      │    return np.pow(array_1, 2)
   10 │       │       │       │       │       │             │      │
   11 │       │       │       │       │       │             │      │
   12 │       │       │       │       │       │             │      │if __name__ == "__main__":
   13 │       │       │       │       │       │             │      │    from sys import argv
   14 │       │       │       │       │       │             │      │    main(int(argv[1]))
   15 │       │       │       │       │       │             │      │
 ---------------------------------------------------------------------------------------------------------
      │       │       │       │       │       │             │      │function summary for /vast/palmer/home.grace/tl397/ycrc/workshops/high_performa…
    3 │    1% │    4% │    6% │   53% │   76M │ █     94%   │      │main

Top AVERAGE memory consumption, by line:
(1)     9:    76 MB
Top PEAK memory consumption, by line:
(1)     9:    76 MB
(2)     7:    72 MB
(3)     1:    10 MB
```

Python Best Practices

Leverage packages like numpy and scipy

- Many common data-focused tasks have already been implemented and optimized in packages like numpy and scipy
  - C/Fortran wrapped with python, so they're *very fast*
- Numpy arrays are good alternatives to native lists
- Allocated empty arrays (with appropriate data types) and then fill with values rather than appending lists
- Use vectorized operations to transform entire arrays at once (often with under-the-hood parallelism) instead of loops
- *Don't reinvent the wheel, use built-in functionalities*

Python Best Practices

File Input and Output

- Working with files is part of many workflows
- Avoid reading small (few-byte) bits of a large file, especially one on a networked filesystem
- Stage data files to local HDDs (/tmp on cluster nodes) to avoid network congestion
- Chunk IO operations to minimize overhead
- Transform into optimized file formats for files that need repeated IO
  - pandas, numpy for reading tabular or binary data
  - PyArrow for columnar datatypes (Feather/Parquet optimized formats)

Part 2:
Resource Management and Monitoring

Resource Management

Request Appropriate Resources

- Clusters are not like workstations, resource requests matter
- Ask for correct number of CPUs, memory, etc. and then monitor usage
- Watch for over-subscription when working with parallel libraries like multiprocessing or joblib (more info in part 2)
- Make sure every requested CPU has exactly one active thread that should be running at 100%
- Watch for memory growth over time
- Consider the whole job, not just the single task, when optimizing

Resource Management

htop and nvidia-smi

- To determine where your job is running:

```
[tl397@r803u11n04.grace ~]$ squeue -u $USER --Format jobid,name,state,nodelist
JOBID               NAME                STATE               NODELIST
9654910             /gpfs/gibbs/public/gPENDING
9654909             waste-seeker.py     PENDING
10715785            ood-desktop         RUNNING             r807u31n01
10717698            interactive         RUNNING             r807u31n01
10718005            node_test           RUNNING             r803u11n04
```

- ssh into the node where your job is running to monitor processes
- `htop`: akin to Activity Monitor or Task Manager for CPU/Memory
- `nvidia-smi`: GPU monitoring
- *Especially important for new workflows or for jobs that feel slow or underperforming*

Resource Management

Jobstats - Realtime resource monitoring

```
[tl397@login1.grace ~]$ jobstats 10717369

================================================================================
                                Slurm Job Statistics
================================================================================
         Job ID: 10717369
  NetID/Account: tl397/admins
       Job Name: node_test
          State: RUNNING
          Nodes: 1
      CPU Cores: 32
     CPU Memory: 181GB (5.7GB per CPU-core)
           GPUs: 4
  QOS/Partition: nothrottle/admintest
        Cluster: grace
     Start Time: Thu Dec 12, 2024 at 9:34 AM
       Run Time: 00:30:42 (in progress)
     Time Limit: 02:00:00

                              Overall Utilization
================================================================================
  CPU utilization  [|||||||||||||||||||||                                47%]
  CPU memory usage [|||||||                                              15%]
  GPU utilization  [|||||||                                              15%]
  GPU memory usage [||||||||||||||||||||||||||||||||||||||||||||||||||||  90%]

                              Detailed Utilization
================================================================================
  CPU utilization per node (CPU time used/run time)
      r902u04n01: 07:44:40/16:22:25 (efficiency=47.3%)

  CPU memory usage per node - used/allocated
      r902u04n01: 31.9GB/206.5GB (1019.5MB/6.5GB per core of 32)

  GPU utilization per node
      r902u04n01 (GPU 0): 14.8%
      r902u04n01 (GPU 1): 14.8%
      r902u04n01 (GPU 2): 13.9%
      r902u04n01 (GPU 3): 14.8%

  GPU memory usage per node - maximum used/total
      r902u04n01 (GPU 0): 21.6GB/24.0GB (90.1%)
      r902u04n01 (GPU 1): 21.6GB/24.0GB (90.1%)
      r902u04n01 (GPU 2): 21.6GB/24.0GB (90.1%)
      r902u04n01 (GPU 3): 21.6GB/24.0GB (90.1%)
```

Resource Management

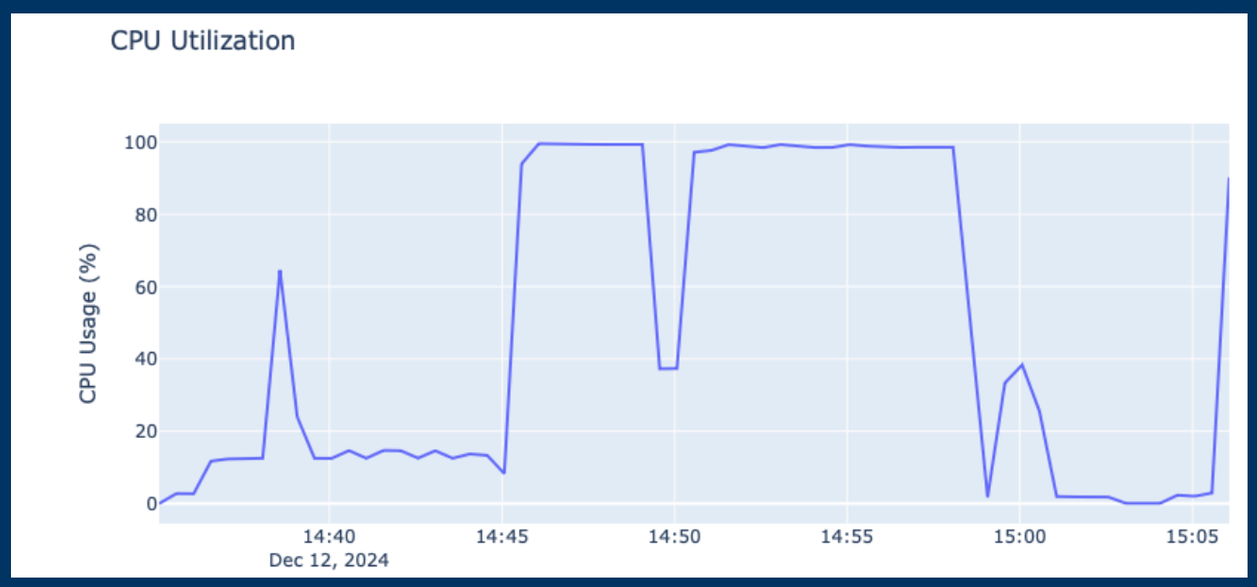
Jobstats-web - Realtime resource monitoring

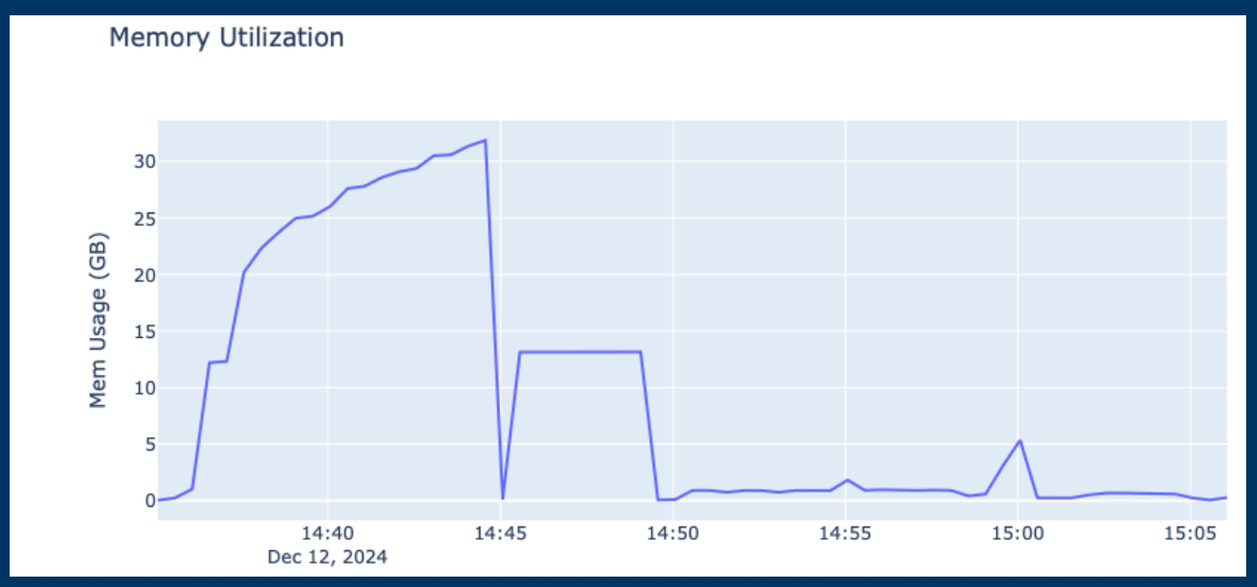
Stats for Job ID: 10717369

Job Info

User	tl397
Account	admins
Partition	admintest
State	RUNNING
JobName	node_test
Start	2024-12-12T09:34:34

Node Statistics	**r902u04n01**
Average CPU Usage (%)	47.53
Maximum Mem Usage (GB)	31.86
Allocated Mem (GB)	206.5
Average GPU Usage (%)	14.29
Maximum GPU Mem Usage (GB)	21.62
Available GPU Mem (GB)	24





Resource Management

Jobstats-web - Realtime resource monitoring

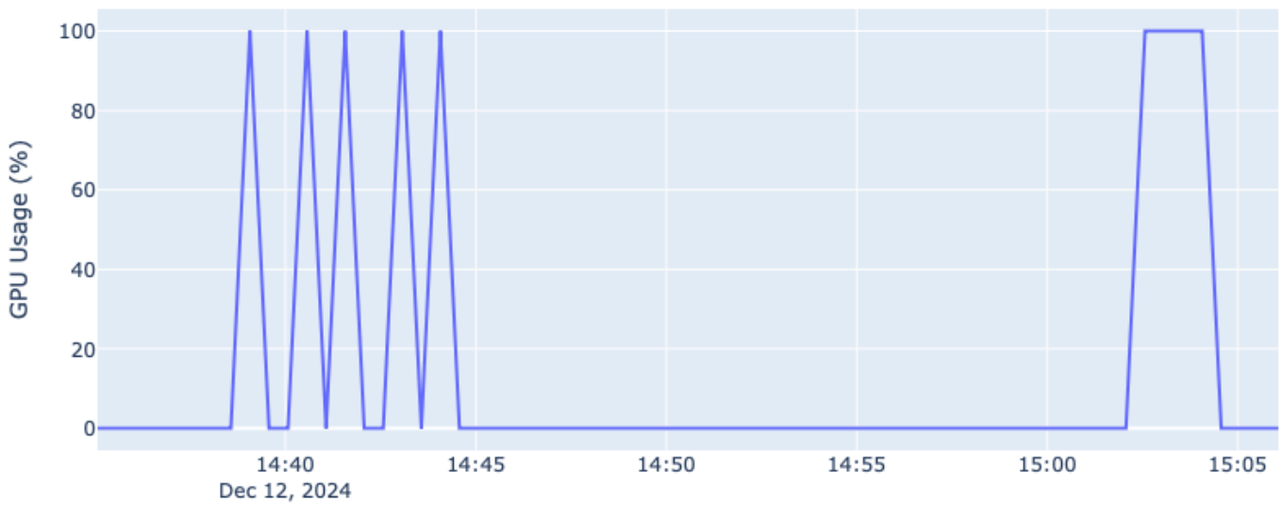
Stats for Job ID: 10717369

Job Info

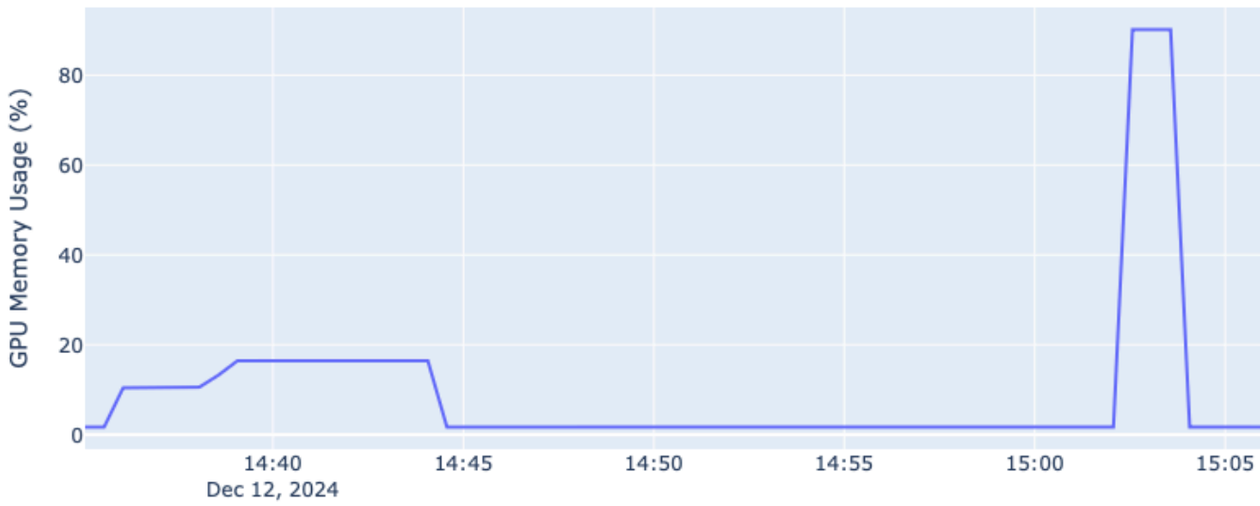
User	tl397
Account	admins
Partition	admintest
State	RUNNING
JobName	node_test
Start	2024-12-12T09:34:34

Node Statistics	**r902u04n01**
Average CPU Usage (%)	47.53
Maximum Mem Usage (GB)	31.86
Allocated Mem (GB)	206.5
Average GPU Usage (%)	14.29
Maximum GPU Mem Usage (GB)	21.62
Available GPU Mem (GB)	24

GPU Utilization



GPU Utilization



Resource Management

Nicely laid-out job

tl397@r803u11n04.grace:~ ⌥⌘1

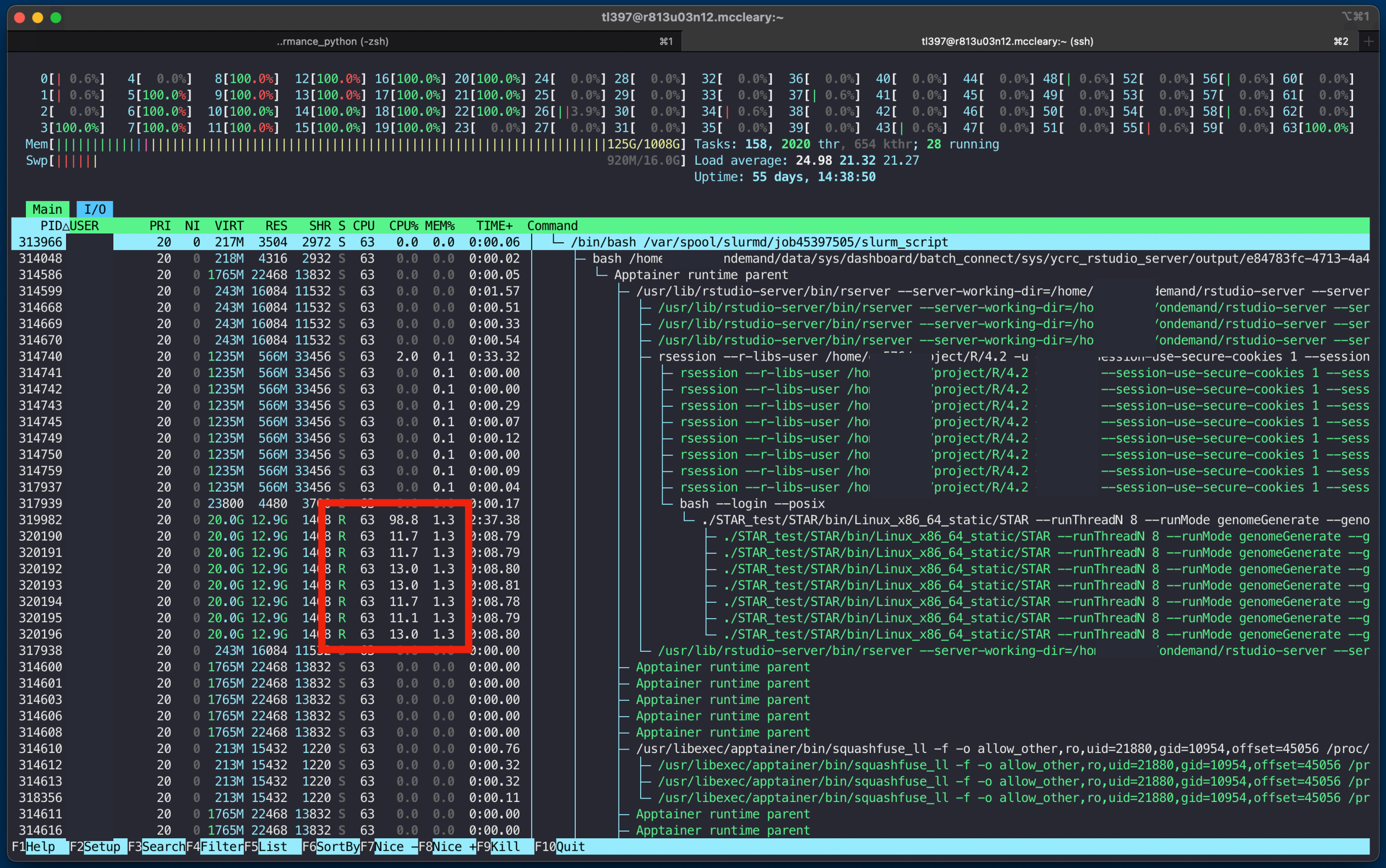
tl397@login1.grace:/vast/palmer/home.grace/tl397/ycrc/workshops/parallel_python (ssh) ⌘1 | tl397@r803u11n04.grace:~ (ssh) ⌘2

```
  0[|||||||||100.0%]  3[|||||||||100.0%]  6[|||||||||100.0%]  9[|||||||||100.0%]  12[|||||||||100.0%] 15[|||||||||100.0%] 18[|||||||||100.0%] 21[|||||||||100.0%]
  1[|||||||||100.0%]  4[|||||||||100.0%]  7[|||||||||100.0%] 10[|||||||||100.0%]  13[|||||||||100.0%] 16[|||||||||100.0%] 19[|||||||||100.0%] 22[|||||||||100.0%]
  2[|||||||||100.0%]  5[|||||||||100.0%]  8[|||||||||100.0%] 11[|||||||||100.0%]  14[|||||||||100.0%] 17[|||||||||100.0%] 20[|||||||||100.0%] 23[|||||||||100.0%]
Mem[|||||||||||||||                                                13.1G/93.0G] Tasks: 188, 739 thr, 336 kthr; 24 running
Swp[                                                                 0K/16.0G] Load average: 9.59 2.32 0.93
                                                                               Uptime: 00:24:05

  Main  I/O
    PID USER       PRI  NI  VIRT   RES   SHR S CPU  CPU%▽MEM%   TIME+  Command
  25282 tl397       20   0  516M  421M 32844 R  14 100.0  0.4  0:21.41 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25287 tl397       20   0  516M  423M 33228 R  12 100.0  0.4  0:21.39 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25292 tl397       20   0  516M  423M 33096 R  21 100.0  0.4  0:21.42 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25312 tl397       20   0  516M  421M 32900 R   8 100.0  0.4  0:21.39 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25432 tl397       20   0  516M  421M 33016 R   6 100.0  0.4  0:21.38 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25456 tl397       20   0  516M  421M 32988 R  16 100.0  0.4  0:21.30 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25283 tl397       20   0  516M  421M 33032 R   5  99.3  0.4  0:21.37 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25289 tl397       20   0  516M  421M 33040 R   4  99.3  0.4  0:21.38 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25290 tl397       20   0  516M  421M 33036 R  23  99.3  0.4  0:21.40 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25294 tl397       20   0  516M  421M 33016 R  18  99.3  0.4  0:21.37 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25295 tl397       20   0  516M  422M 33372 R   2  99.3  0.4  0:21.40 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25297 tl397       20   0  516M  421M 32864 R  13  99.3  0.4  0:21.42 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25306 tl397       20   0  516M  421M 33080 R   0  99.3  0.4  0:21.37 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25319 tl397       20   0  516M  421M 33136 R  17  99.3  0.4  0:21.43 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25336 tl397       20   0  516M  421M 32900 R  11  99.3  0.4  0:21.43 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25337 tl397       20   0  516M  421M 32884 R   3  99.3  0.4  0:21.42 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25340 tl397       20   0  516M  421M 32716 R   1  99.3  0.4  0:21.39 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25362 tl397       20   0  516M  421M 32980 R  19  99.3  0.4  0:21.44 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25370 tl397       20   0  516M  422M 33436 R   7  99.3  0.4  0:21.43 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25403 tl397       20   0  516M  421M 32756 R  15  99.3  0.4  0:21.40 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25429 tl397       20   0  516M  422M 33332 R  20  99.3  0.4  0:21.39 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25457 tl397       20   0  516M  421M 32712 R   9  99.3  0.4  0:21.41 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25463 tl397       20   0  516M  421M 33124 R  22  99.3  0.4  0:21.41 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  25296 tl397       20   0  516M  422M 33436 R  10  98.0  0.4  0:21.28 /usr/local/bin/DMX /vast/palmer/project/support/benchmark/pytest/node_tests/cpu_burn//benchmar
  24715 tl397       20   0  237M  6384  3528 R  10   1.3  0.0  0:00.41 htop
  24798 tl397       20   0 3694M 68112 24808 S  18   0.7  0.1  0:00.02 python -u -m pytest -s -v node_tests/
  24605 tl397       20   0  217M  3464  3036 S   9   0.0  0.0  0:00.00 /bin/bash /var/spool/slurmd/job10718005/slurm_script
  24626 tl397       20   0  163M  5396  4004 S   6   0.0  0.0  0:00.00 sshd: tl397@pts/0
  24627 tl397       20   0  241M 14960  3496 S   9   0.0  0.0  0:00.05 -bash
  24741 tl397       20   0 3694M 68112 24808 S   3   0.0  0.1  0:02.66 python -u -m pytest -s -v node_tests/
  24774 tl397       20   0 3502M 47112  3856 S  14   0.0  0.0  0:00.00 python -u -m pytest -s -v node_tests/
  24775 tl397       20   0 3502M 47116  3856 S  17   0.0  0.0  0:00.00 python -u -m pytest -s -v node_tests/
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Resource Management

Multiple tasks on one CPU



Resource Management

Nicely performing GPU job

```
[tl397@r908u24n01.grace ~]$ nvidia-smi
Thu Dec 12 11:04:53 2024
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 555.42.06              Driver Version: 555.42.06      CUDA Version: 12.5     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  Tesla V100-PCIE-16GB           On  |   00000000:58:00.0 Off |                    0 |
| N/A   41C    P0             89W /  250W |     352MiB /  16384MiB |     94%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
|   1  Tesla V100-PCIE-16GB           On  |   00000000:D8:00.0 Off |                    0 |
| N/A   38C    P0             91W /  250W |     352MiB /  16384MiB |     92%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A    886723      C   ./project/code                                348MiB |
|    1   N/A  N/A    886879      C   ./project/code                                348MiB |
```

Resource Management

CPU-bottleneck

```
                                Overall Utilization
================================================================================
CPU utilization  [|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||99%]
CPU memory usage [|||||||||||||||||||||||||||||||||                        67%]
GPU utilization  [||                                                        5%]
GPU memory usage [||                                                        4%]

                              Detailed Utilization
================================================================================
CPU utilization per node (CPU time used/run time)
    r907u34n01: 19:19:42/19:28:40 (efficiency=99.2%)

CPU memory usage per node - used/allocated
    r907u34n01: 120.0GB/180.0GB (120.0GB/180.0GB per core of 1)

GPU utilization per node
    r907u34n01 (GPU 0): 6.5%
    r907u34n01 (GPU 2): 3.9%
    r907u34n01 (GPU 3): 3.8%

GPU memory usage per node - maximum used/total
    r907u34n01 (GPU 0): 1.7GB/40.0GB (4.3%)
    r907u34n01 (GPU 2): 1.7GB/40.0GB (4.2%)
    r907u34n01 (GPU 3): 1.6GB/40.0GB (4.1%)

                                    Notes
```

Resource Management

CPU-bottleneck

```
[tl397@r907u34n01.grace ~]$ nvidia-smi
Thu Dec 12 11:07:42 2024
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 555.42.06              Driver Version: 555.42.06      CUDA Version: 12.5     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA A100-PCIE-40GB          On  |   00000000:3B:00.0 Off |                    0 |
| N/A   27C    P0              33W /  250W |    1267MiB /  40960MiB |      0%      Default |
|                                         |                        |             Disabled |
+-----------------------------------------+------------------------+----------------------+
|   1  NVIDIA A100-PCIE-40GB          On  |   00000000:5E:00.0 Off |                    0 |
| N/A   27C    P0              43W /  250W |       1MiB /  40960MiB |      0%      Default |
|                                         |                        |             Disabled |
+-----------------------------------------+------------------------+----------------------+
|   2  NVIDIA A100-PCIE-40GB          On  |   00000000:AF:00.0 Off |                    0 |
| N/A   28C    P0              46W /  250W |    1189MiB /  40960MiB |      0%      Default |
|                                         |                        |             Disabled |
+-----------------------------------------+------------------------+----------------------+
|   3  NVIDIA A100-PCIE-40GB          On  |   00000000:D8:00.0 Off |                    0 |
| N/A   27C    P0              45W /  250W |    1163MiB /  40960MiB |      0%      Default |
|                                         |                        |             Disabled |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A   1112659      C   python                                     1258MiB |
|    2   N/A  N/A   1112659      C   python                                     1180MiB |
|    3   N/A  N/A   1112659      C   python                                     1154MiB |
```

```
  0[|||||||100.0%]   5[                0.0%]   10[                0.0%]   15[                0.0%]   18[
  1[                0.0%]   6[                0.0%]   11[                0.0%]   16[                0.0%]   19[
  2[                0.0%]   7[                0.0%]   12[                0.0%]   17[                0.0%]   20[
  3[                0.0%]   8[                0.0%]   13[                0.0%]                           21[
  4[                0.0%]   9[                0.0%]   14[                0.0%]                           22[
Mem[|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||116G/377G] Tasks: 52, 
Swp[||                                                                          78.4M/16.0G] Load average
                                                                                             Uptime: 8 da

  Main   I/O
    PID△USER       PRI  NI  VIRT   RES    SHR S CPU  CPU% MEM%   TIME+  Command
1112623              20   0  217M  3300   2840 S   0   0.0  0.0  0:00.00      └─ /bin/bash /va
1112659              20   0  130G   109G   943M R   0  90.5 29.0 19h11:00          └─ python De
1112670              20   0  130G   109G   943M S   0   0.0 29.0  0:00.00             ├─ python
1112696              20   0 2054M   140M 38872 S   0   8.6  0.0  8:21.78             ├─ wandb-
1112700              20   0 2054M   140M 38872 S   0   0.0  0.0  0:00.00             │  ├─ wan
1112702              20   0 2054M   140M 38872 S   0   0.0  0.0  0:07.15             │  ├─ wan
1112703              20   0 2054M   140M 38872 S   0   0.0  0.0  0:02.09             │  ├─ wan
1112704              20   0 2054M   140M 38872 S   0   0.0  0.0  0:01.94             │  ├─ wan
1112705              20   0 2054M   140M 38872 S   0   0.0  0.0  0:26.38             │  ├─ wan
1112706              20   0 2054M   140M 38872 S   0   0.0  0.0  0:06.57             │  ├─ wan
1112707              20   0 2054M   140M 38872 S   0   0.0  0.0  0:23.34             │  ├─ wan
1112708              20   0 2054M   140M 38872 S   0   0.0  0.0  0:03.18             │  ├─ wan
1112711              20   0 2054M   140M 38872 S   0   0.0  0.0  0:03.97             │  ├─ wan
1112712              20   0 2054M   140M 38872 S   0   0.0  0.0  0:00.00             │  ├─ wan
1112713              20   0 2054M   140M 38872 S   0   0.0  0.0  0:00.00             │  ├─ wan
1112714              20   0 2054M   140M 38872 S   0   0.7  0.0  0:19.08             │  ├─ wan
1112715              20   0 2054M   140M 38872 S   0   0.0  0.0  0:03.04             │  ├─ wan
1112719              20   0 2054M   140M 38872 S   0   0.0  0.0  0:01.86             │  ├─ wan
1112720              20   0 2054M   140M 38872 S   0   0.0  0.0  0:22.02             │  ├─ wan
1112721              20   0 2054M   140M 38872 S   0   0.0  0.0  0:15.06             │  ├─ wan
1112722              20   0 2054M   140M 38872 S   0   8.6  0.0  5:33.62             │  ├─ wan
1112723              20   0 2054M   140M 38872 S   0   0.0  0.0  0:08.55             │  ├─ wan
1112724              20   0 2054M   140M 38872 S   0   0.0  0.0  0:10.08             │  ├─ wan
1112726              20   0 24532  5152  4656 S   0   0.0  0.0  0:00.00             │  ├─ git
1112731              20   0 2054M   140M 38872 S   0   0.0  0.0  0:00.00             │  ├─ wan
1112735              20   0 2054M   140M 38872 S   0   0.0  0.0  0:03.17             │  ├─ wan
1112736              20   0 2054M   140M 38872 S   0   0.0  0.0  0:04.42             │  └─ wan
1112709              20   0  130G   109G   943M S   0   0.0 29.0  0:09.00             ├─ python
```

Part 2:
Parallelism

Explicit and Implicit Parallelism

Combine at your own risk…

- Parallelism is, broadly speaking, running pieces of code at the same time
- Some libraries do this on their own in the background
    - eg: Numpy uses OpenMP to parallelize certain vector operations
- Some libraries give you the tools to explicitly run things in parallel
    - multiprocessing, or joblib
- These can be very useful and powerful, but we have to be careful about how they can collide…

Implicit Parallelism

Environment variables

- Numpy (and libraries which call numpy) looks for available CPUs and attempts to use any/all that it finds
- Can be controlled with a variety of environment variables (depending on what library numpy is built against)
  - `OMP_NUM_THREADS` (for standard OpenMP)
  - `MKL_NUM_THREADS` (if numpy is built against MKL libraries)
- If unset, numpy often attempts to use all available CPUs
- If you wish to do explicit parallelism (next slide) make sure to disable this `export OMP_NUM_THREADS=1`

Explicit Parallelism

multiprocessing Pools

- The “standard” first step into parallelism with python is multiprocessing’s Pool object
  - tasks assigned to “pool” of workers
  - “map” a function to a list of inputs
- **Only works on a single computer**
- Match number of workers to available CPUs (on your local machine, leave one or two CPUs free)

```
# serial:
[sqrt(i ** 2) for i in range(10)]

# multiprocessing.Pool with 2 CPUs:
with multiprocessing.Pool(processes=2) as pool:
    pool.map(sqrt, range(10))
```

Exercise 1: Multiprocessing

Cluster-based Parallelism

Think wide…

- On a workstation you have to manage all infrastructure yourself
  - eg: Using multiprocessing to run O(10) threads
- Clusters offer a scheduler that can launch *O(1000)* of jobs and a shared filesystem so you can run jobs on any node
- When migrating workflows from local machines to HPC, great value in leveraging cluster infrastructure:
  1. Break jobs down to the “smallest independent components” that can be run in parallel
  2. Use command-line arguments to pass values into code
  3. Use job-arrays to run these asynchronously on dozens of nodes

Cluster-based Parallelism

Job Arrays and dSQ

- Slurm has a powerful concept of a collection of jobs that:
  - all require the same resources
  - are provided unique indices
  - can run independently and asynchronously
- Can be constructed directly in a batch script or with the help of our dSQ utility (https://docs.ycrc.yale.edu/clusters-at-yale/job-scheduling/dsq/)
- Use-case examples:
  - Pipeline processing of independent data files
  - Monte Carlo simulation with definable random seeds
  - Grid-search (minimization/maximization) for complicated function with various inputs

Cluster-based Parallelism

Simple Job Array Example

```python
import numpy as np
import pandas as pd
from numba import jit, prange
import random

def numpy_mc(n_samples = 10000000, seed=42):

    np.random.seed(seed)

    x = np.random.random(size=n_samples)

    y = np.random.random(size=n_samples)

    r = np.power(np.power(x,2) + np.power(y,2), 0.5)

    counter = r[r<1].size

    return 4 * counter / n_samples

if __name__ == '__main__':

    from sys import argv
    import time
    import os

    seed = int(argv[1])

    pi_calc = numpy_mc(n_samples = 100000000, seed=seed)

    print(f'{os.getenv("SLURM_ARRAY_TASK_ID")}: {pi_calc:.10f}')
```

```
#!/bin/bash

#SBATCH -J mc_pi
#SBATCH -c 1
#SBATCH --mem-per-cpu=10G
#SBATCH --partition admintest
#SBATCH --array=0-10

module purge
module load miniconda
conda activate jupyter
python ./mc_job_array.py $SLURM_ARRAY_TASK_ID
```

Cluster-based Parallelism

Simple Job Array Example

```
slurm-10628141_0.out:0: 3.1417891600
slurm-10628141_10.out:10: 3.1414101200
slurm-10628141_1.out:1: 3.1413615600
slurm-10628141_2.out:2: 3.1415220400
slurm-10628141_3.out:3: 3.1414847600
slurm-10628141_4.out:4: 3.1416750400
slurm-10628141_5.out:5: 3.1414482000
slurm-10628141_6.out:6: 3.1414481600
slurm-10628141_7.out:7: 3.1415387600
slurm-10628141_8.out:8: 3.1415581200
slurm-10628141_9.out:9: 3.1413055600
```

Note: Each output is different because of random seed.
Can be averaged to get improved result

- Each job reports independent result asynchronously (in text file or data file)
- Write a simple “clean-up” script to combine results

Slurm Environment Variables

https://slurm.schedmd.com/sbatch.html

- `SLURM_CPUS_ON_NODE`: number of CPUs available to this job on this node
- `SLURM_ARRAY_JOB_ID`: shared jobid for entire array
- `SLURM_ARRAY_TASK_ID`: index for a specific array element
- `SLURM_ARRAY_TASK_MIN`: minimum of the specified indices
- `SLURM_ARRAY_TASK_MAX`: maximum of the specified indices
- `SLURM_ARRAY_TASK_COUNT`: total number of array elements
- Use these in your batch scripts!

dSQ: High-level job array tool

Job Script Generation and Submission

- Manually or programmatically generate a “jobfile” with the commands that need to be run

```
dsq -p day -c 1 -t 1:00:00 --mem=2G --job-file jobfile.txt
Batch script generated. To submit your jobs, run:
 sbatch dsq-jobfile-2024-12-12.sh
```

File: **jobfile.txt**

```
python image_flipper.py ./data/waterloo.jpg
python image_flipper.py ./data/victoria.jpg
python image_flipper.py ./data/paddington.jpg
python image_flipper.py ./data/charing_cross.jpg
python image_flipper.py ./data/euston.jpg
python image_flipper.py ./data/kings_cross.jpg
```

File: **dsq-jobfile-2024-12-12.sh**

```
#!/bin/bash
#SBATCH --output dsq-jobfile-%A_%1a-%N.out
#SBATCH --array 0-9
#SBATCH --job-name dsq-jobfile
#SBATCH -p day -c 1 -t 1:00:00 --mem-per-cpus=2G

# DO NOT EDIT LINE BELOW
/vast/palmer/apps/avx2/software/dSQ/1.05/dSQBatch.py --job-file /vast/palmer/home.grace/tl397/ycrc/workshops/high_performance_python/jobfile.txt --status-dir /vast/palmer/home.grace/tl397/ycrc/workshops/high_performance_python

```

dSQ: High-level job array tool

Job Autopsy and Resubmission

- Sometimes jobs fail (insufficient memory requested for example)
- dSQ provides an “autopsy” report to show any jobs which failed and need to be rerun:
- If you pass the job-file, it generates a reduced list of jobs that can be resubmitted via dsq

```
[tl397@login1.grace tl397]$ dsqa -j 10717418
State Summary for Array 10717418
State       Num_Jobs Indices
_____       ________ _______
RUNNING           74 1-37,57-93
FAILED            13 44-56
COMPLETED          6 38-43
```

Cluster-based Parallelism

Think wide… and think about the full job

- Migrating from workstations or laptops to the cluster often requires complete rethinking of code structure
- Need to think “wide” and about optimizing the whole job, not just one iteration or piece
- Linear scaling is hard to achieve. If you have lots of jobs to run, it’s often better to let each job run longer with fewer CPUs so that you can run more jobs in parallel
- *Let the cluster manage parallelism instead of doing it manually!*

Part 3:
Accelerating Python

Accelerating Python

numba, Dask, Cupy

- Here we're going to take a step beyond the "standard" tools of numpy/scipy/pandas and leverage some "advanced" concepts for improving speed and throughput

- **numba:** just-in-time compiling of python code

- **Dask:** intelligently distribute pieces of calculations across multiple CPUs (and also compute nodes)

- **Cupy:** use GPUs to off-load expensive pieces of calculations with minimal API changes from numpy or pandas

Numba

https://numba.pydata.org/

- A compiler is a tool that converts human-readable code into machine readable instructions
- Numba is an open-source “Just-in-time” (JIT) compiler for python
  - Compiles only what’s required while running
- Minimal code-modification required, uses “decorators” to denote functions to be compiled
- Can yield O(10)-O(100) speed up in certain situations
- Can be expanded to CUDA, but that gets complicated *fast*

Numba

Monte Carlo Pi Calculation

Numpy

```
def numpy_mc(n_samples = 1000000):

    np.random.seed(42)

    x = np.random.random(size=n_samples)
    y = np.random.random(size=n_samples)

    r = np.power(np.power(x,2) + np.power(y,2), 0.5)

    counter = r[r<1].size

    return 4 * counter / n_samples
```

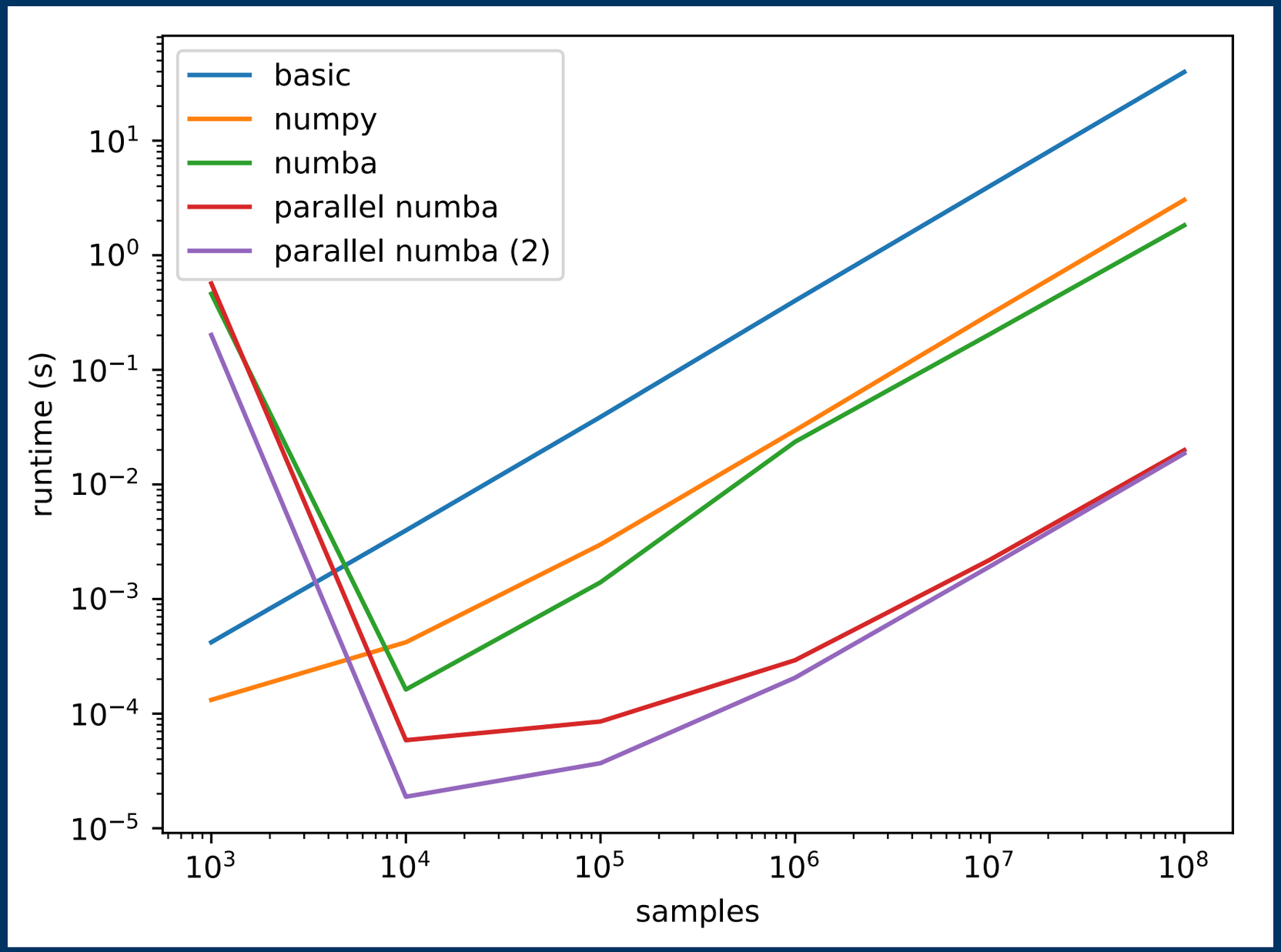
Numba

```
@jit(nopython=True, parallel = True)
def numba_par_mc_loop(n_samples = 1000000):

    counter = 0
    for i in prange(n_samples):
        x = random.random()
        y = random.random()
        if x**2 + y**2 < 1:
            counter += 1
    return 4 * counter / n_samples
```

Numba

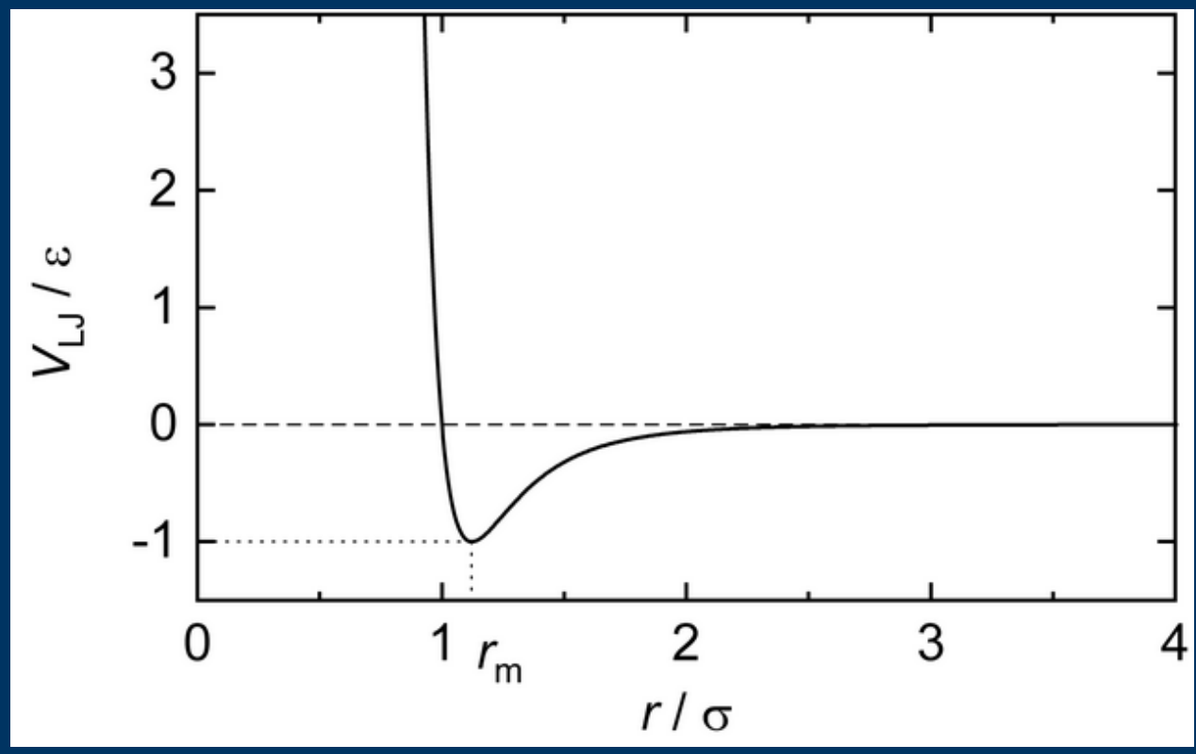
Monte Carlo Pi Calculation

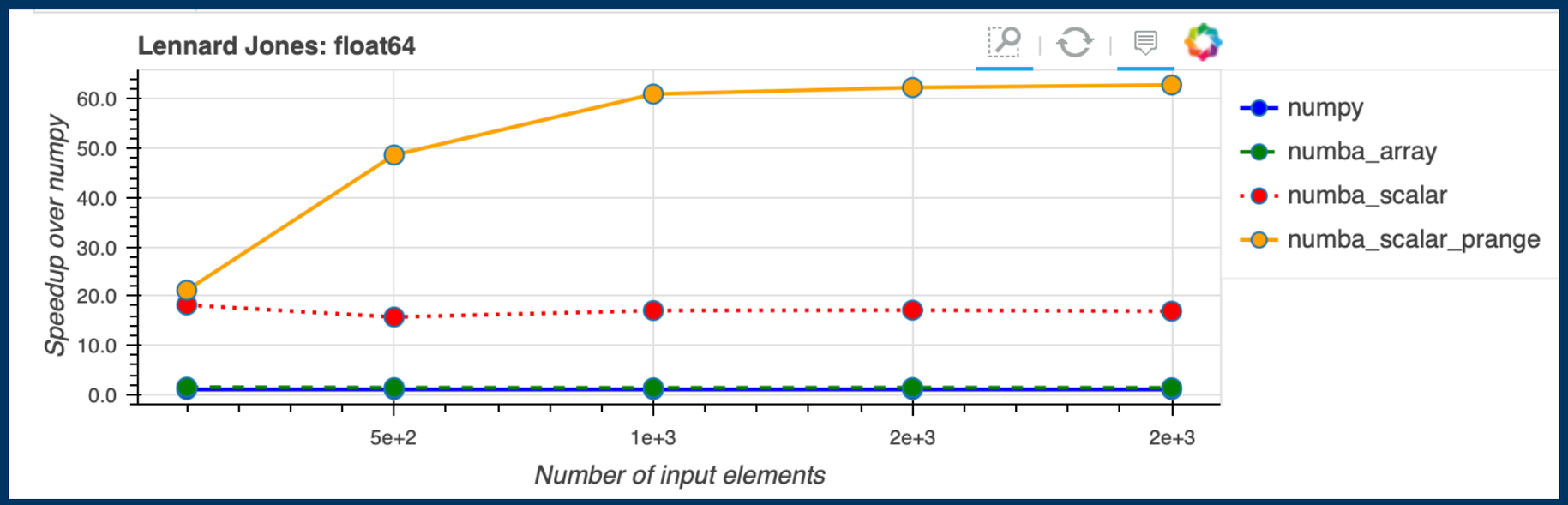


Numba

Non-vectorizable codes

- Some algorithms cannot be easily vectorized to use numpy
- Example: Calculate the behavior of particles in a box with a Lennard Jones potential
- Each particle needs information from all other particles
- https://numba.pydata.org/numba-examples/examples/physics/lennard_jones/results.html





Numba

Take aways

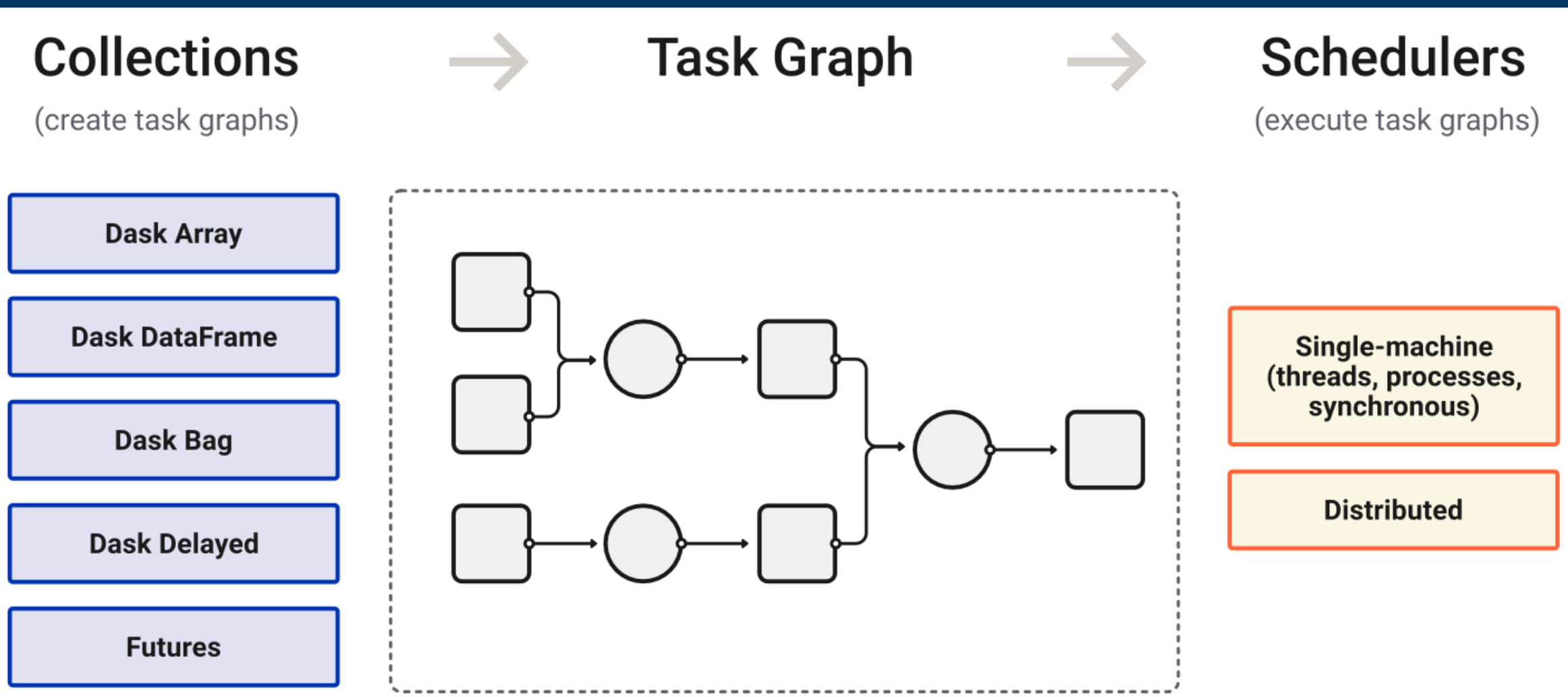
- Numba can take simple python code and dramatically improve speed
- This is especially useful for problems that cannot be easily vectorized to take advantage of numpy
- Numba benefits from having defined datatypes
- For small jobs, the overhead of compiling can slow things down
- For frequently executed sections of code, can have major impact

Dask

https://docs.dask.org/en/stable/

- Dask is a high-level framework designed to behave like numpy or pandas but run computations across multiple CPUs/Nodes/ Systems
- Dask takes several “standard” Python concepts and expands on them:
  - Dask Arrays: numpy-like syntax but for larger-than-memory data, with parallel execution out-of-the-box
  - Dask DataFrames: same thing but for tabular data
- *“Lazy execution” of code which identifies what can be run in parallel*

Dask



High level collections are used to generate task graphs which can be executed by schedulers on a single machine or a cluster.

Exercise: Dask

Dask

Takeaways

- In this era of big data, it's more and more common to interact with larger-than-memory datasets
- Dask provides a simple and effective way to orchestrate large data workflows
- On a single node, this can simplify chunked operations and perform aggregations in parallel
- With Dask-Distributed and Dask-joblib multiple workers can be controlled without complicated code

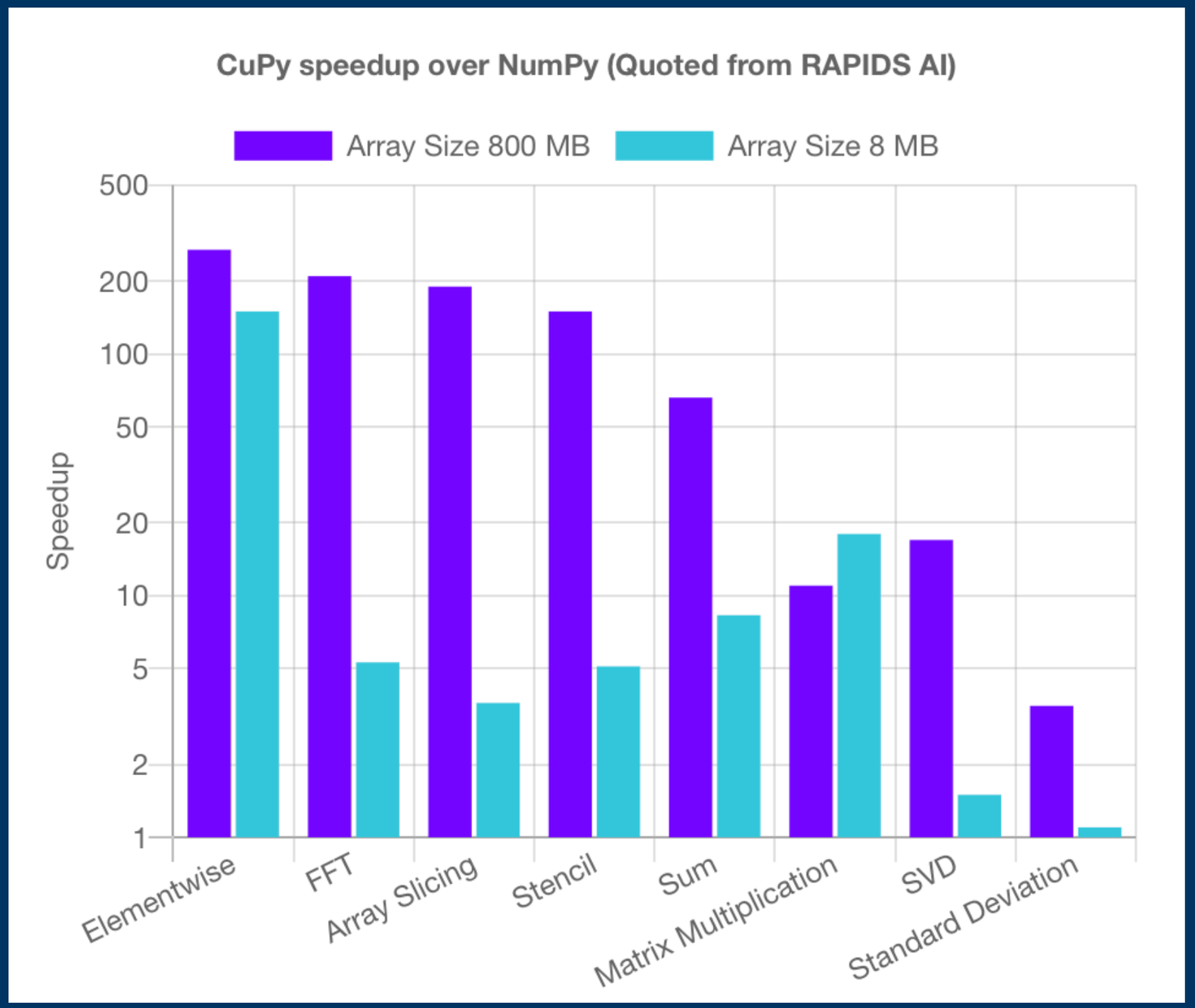
CuPy

cupy.dev GPU accelerated Numpy

- CuPy: provide Python users GPU acceleration capabilities, without the in-depth knowledge of underlying GPU technologies
- Works with both NVIDIA and AMD GPUs (via ROCm)
- Drop-in replacement for many (and increasing with each release) numpy and scipy APIs
  - often this “just works”: `import cupy as np`
- Can even handle multiple GPUs

CuPy

cupy.dev GPU accelerated Numpy



Exercise: CuPy

Summing up…

Things to remember going forward

- **Human time is expensive and CPU time is cheap**
- Profile (and study scaling!) before any optimization
- When running on the cluster, look for ways to break jobs up and run independently
- Many ways to solve every puzzle, stay curious and try new things

Better flow-diagram

Programmer time is expensive…