

High Performance Python

Improving code efficiency and performance

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

UV (<https://docs.astral.sh/uv/>)

- Use uv module to install python and required packages (via pyproject.toml files)

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

UV (<https://docs.astral.sh/uv/>)

- Use uv module to install python and required packages (via pyproject.toml files)

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 joblib
```

- On personal machine, install miniconda from conda-forge
- On the clusters, `module load miniconda` and run install command on compute node
- NB: CUDA13 is incompatible with CUDA on the clusters, so be sure to restrict the installation to CUDA<=12

Presentation Materials

[github.com/ycrc/high performance python](https://github.com/ycrc/high_performance_python)



Part 1: Why is my code slow?

ARE YOU PREMATURELY
OPTIMIZING OR JUST TAKING
TIME TO DO THINGS RIGHT?



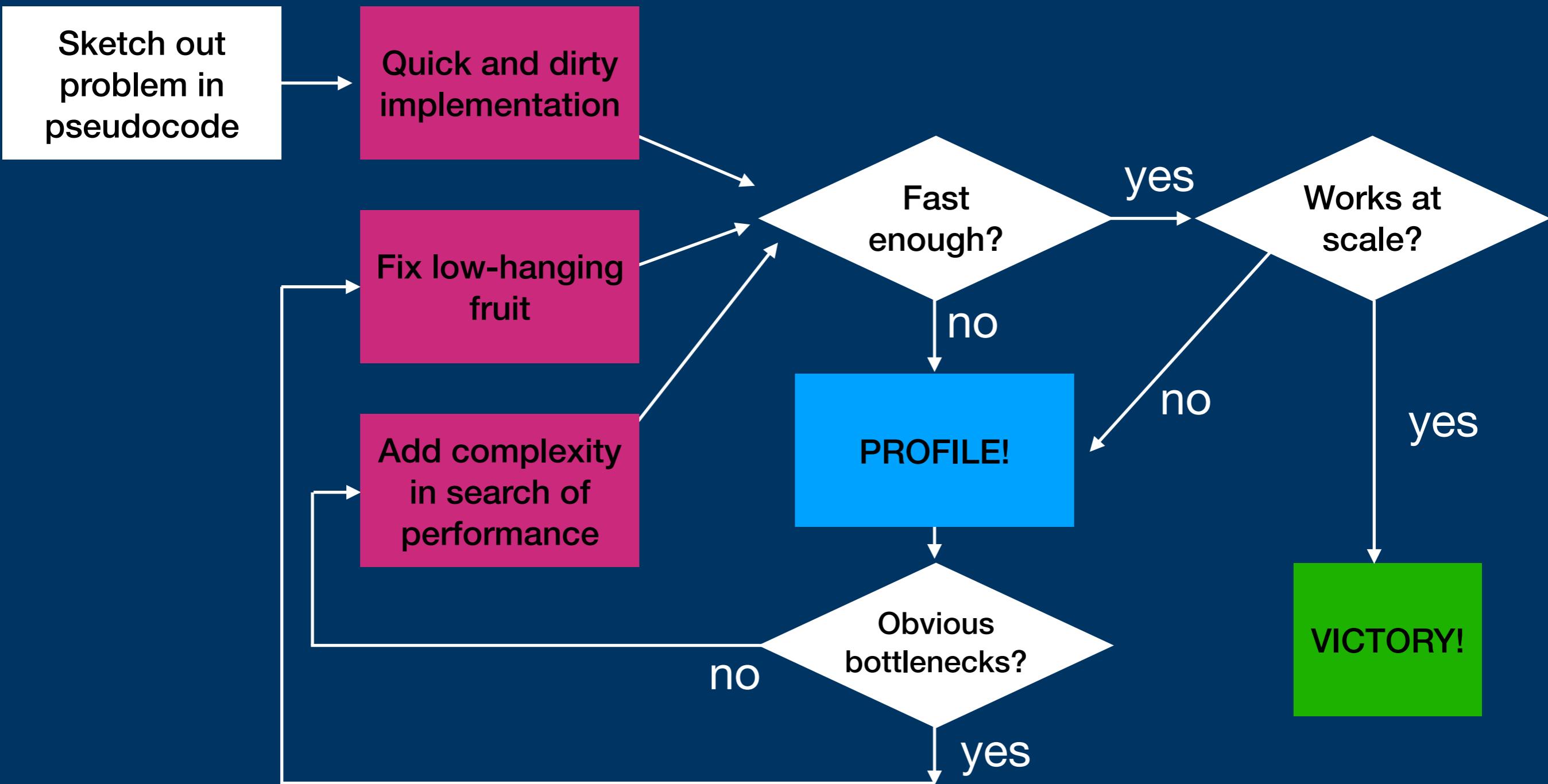
ARE YOU CONSULTING A
FLOWCHART TO ANSWER
THIS QUESTION?

YES

YOU ARE
PREMATURELY
OPTIMIZING

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

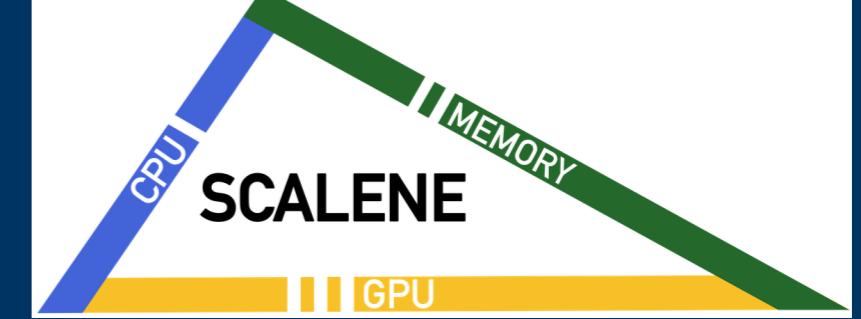
Code Profiling

“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

```
1  def main(n):  
2  
3      print(f'{n} element list')  
4  
5      list_1 = []  
6      list_2 = []  
7  
8      for i in range(n):  
9          list_1.append(i)  
10  
11     for i in list_1:  
12         list_2.append(i * i)  
13  
14     return list_2  
15  
16  
17  
18     if __name__ == "__main__":  
19         from sys import argv  
20         main(int(argv[1]))
```

File: array_example.py

```
1  import numpy as np  
2  
3  def main(n):  
4  
5      print(f'{n} element list')  
6  
7      array_1 = np.arange(0,n)  
8  
9      return np.pow(array_1, 2)  
10  
11  
12     if __name__ == "__main__":  
13         from sys import argv  
14         main(int(argv[1]))
```

Profiling Example

Lists vs Numpy arrays

```
(hpp) [tl397@r807u31n01.grace numpy]$ scalene list_example.py 10000000  
10000000 element 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.

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

Line	Time Python	native	system	Memory Python	peak	timeline/%	Copy (MB/s)	/vast/palmer/home.grace/tl397/ycrc/workshops/high_performance_python/numpy/arra...
1	5%	76%	8%	100%	18M	6%	87	<pre>import numpy as np def main(n): print(f'{n} element list') array_1 = np.arange(0,n) return np.pow(array_1, 2) if __name__ == "__main__": from sys import argv main(int(argv[1]))</pre>
2								
3								
4								
5								
6								
7								
8								
9								
10								
11								
12								
13								
14								
15								
3	1%	4%	6%	53%	76M	94%		<i>function summary for /vast/palmer/home.grace/tl397/ycrc/workshops/high_perfor...</i>
								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
 - numpy, pandas for reading tabular or binary data
 - polars/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

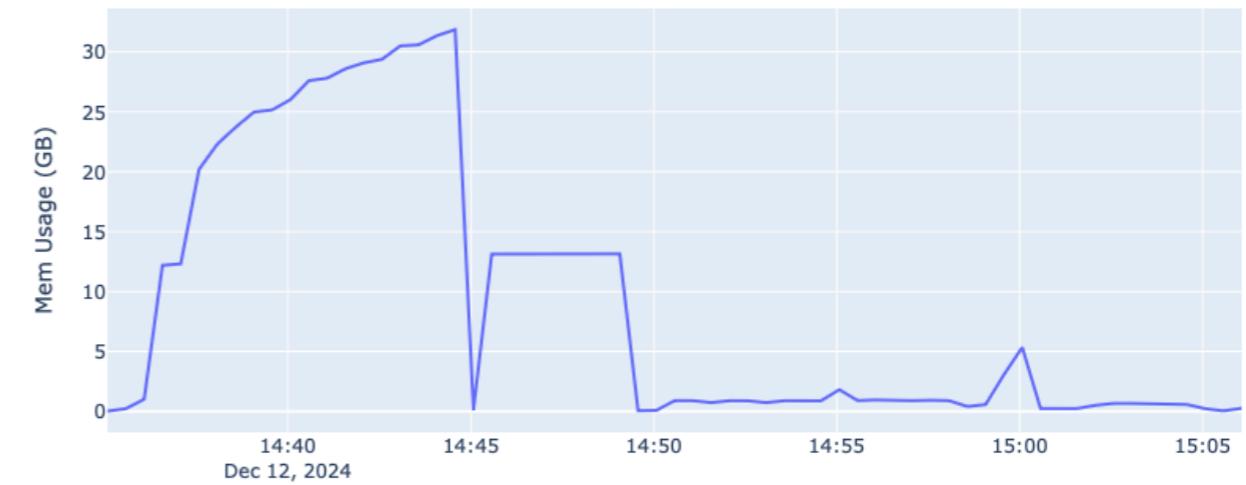
Jobstats-web - Realtime resource monitoring

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

CPU Utilization



Memory Utilization

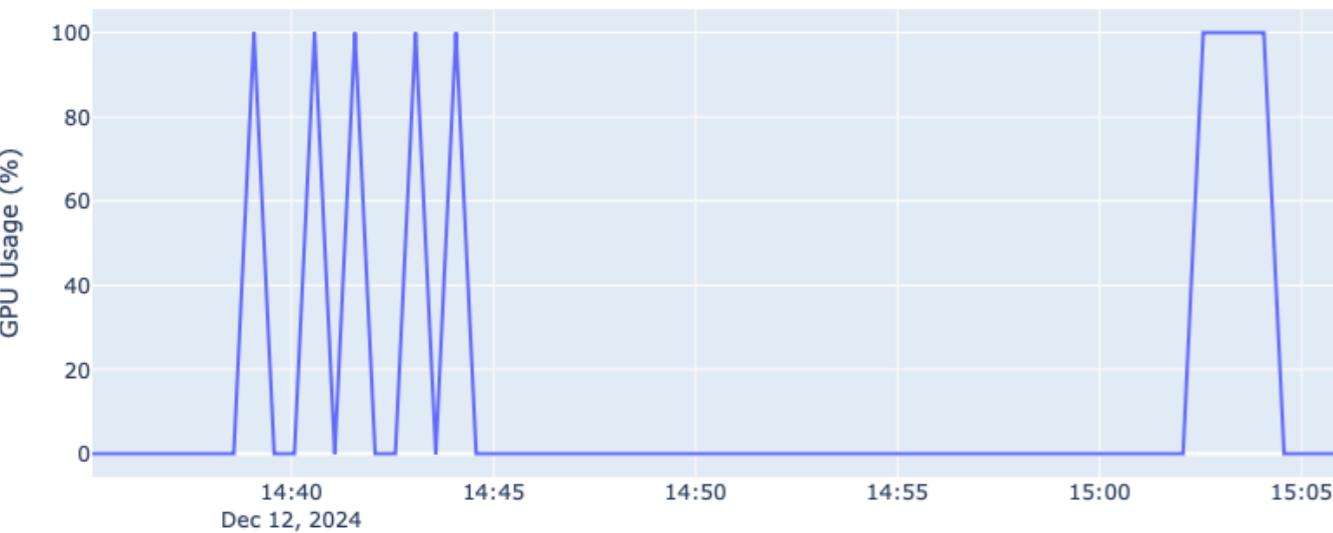


Resource Management

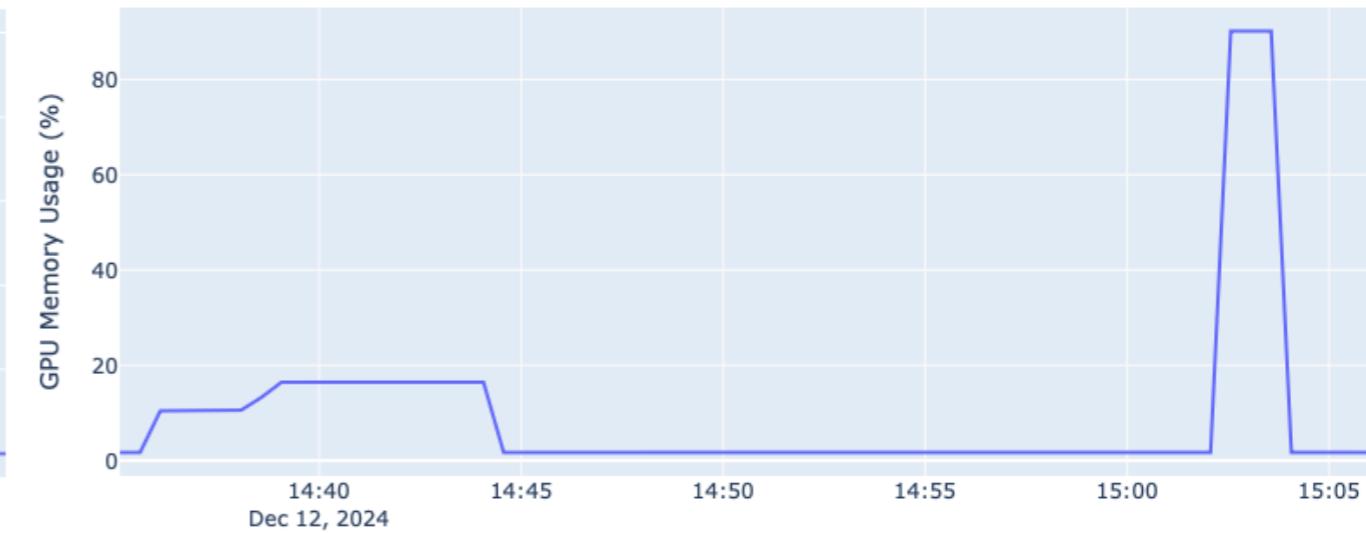
Jobstats-web - Realtime resource monitoring

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

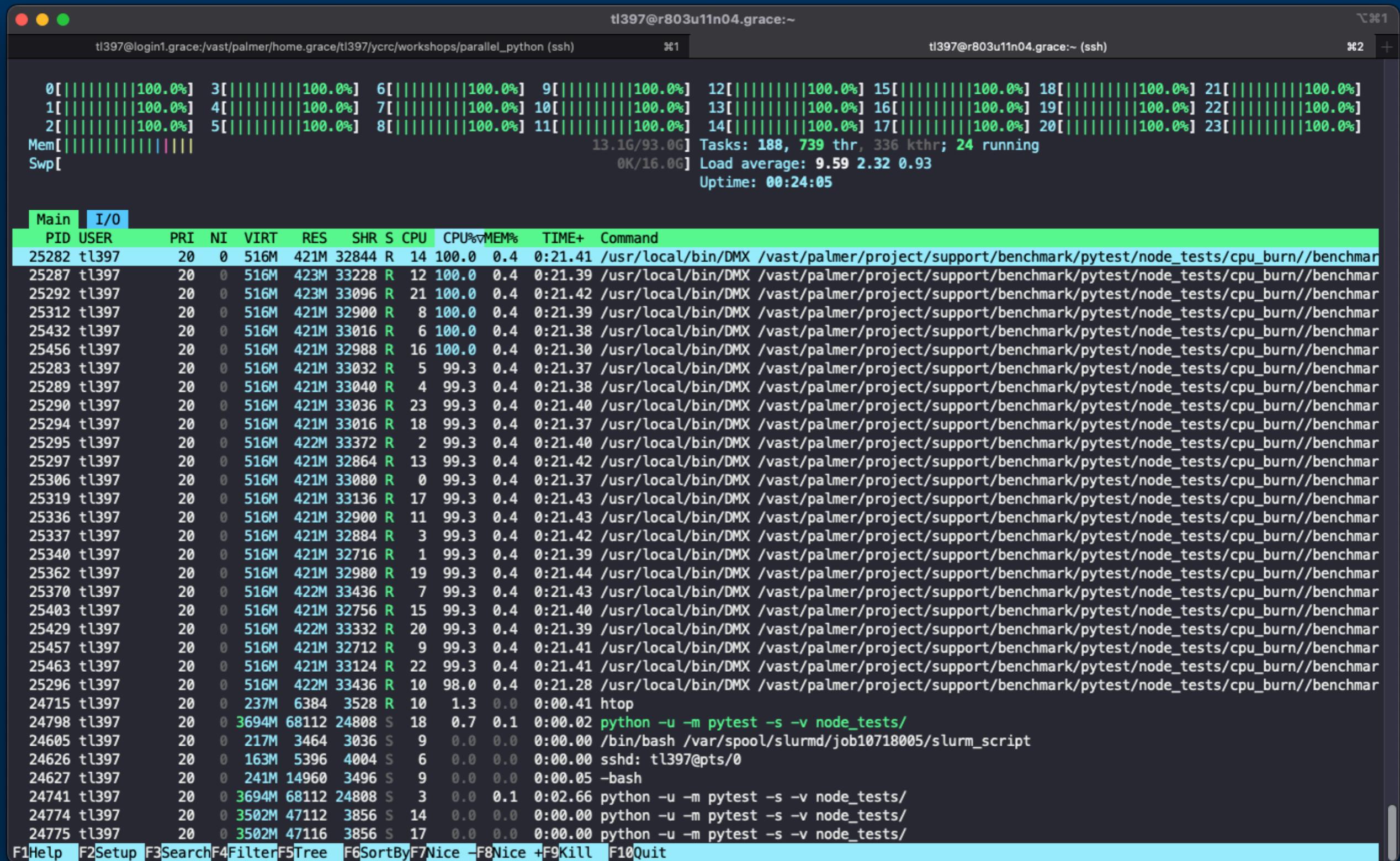


GPU Utilization



Resource Management

Nicely laid-out job



Resource Management

Multiple tasks on one CPU

tl397@r813u03n12.mccleary:~

tl397@r813u03n12.mccleary:~ (ssh) #2

```
0[| 0.6%] 4[ 0.0%] 8[100.0%] 12[100.0%] 16[100.0%] 20[100.0%] 24[ 0.0%] 28[ 0.0%] 32[ 0.0%] 36[ 0.0%] 40[ 0.0%] 44[ 0.0%] 48[| 0.6%] 52[ 0.0%] 56[| 0.6%] 60[ 0.0%]
1[| 0.6%] 5[100.0%] 9[100.0%] 13[100.0%] 17[100.0%] 21[100.0%] 25[ 0.0%] 29[ 0.0%] 33[ 0.0%] 37[| 0.6%] 41[ 0.0%] 45[ 0.0%] 49[ 0.0%] 53[ 0.0%] 57[ 0.0%] 61[ 0.0%]
2[ 0.0%] 6[100.0%] 10[100.0%] 14[100.0%] 18[100.0%] 22[100.0%] 26[| 3.9%] 30[ 0.0%] 34[| 0.6%] 38[ 0.0%] 42[ 0.0%] 46[ 0.0%] 50[ 0.0%] 54[ 0.0%] 58[| 0.6%] 62[ 0.0%]
3[100.0%] 7[100.0%] 11[100.0%] 15[100.0%] 19[100.0%] 23[ 0.0%] 27[ 0.0%] 31[ 0.0%] 35[ 0.0%] 39[ 0.0%] 43[| 0.6%] 47[ 0.0%] 51[ 0.0%] 55[| 0.6%] 59[ 0.0%] 63[100.0%]
Mem[|||||] 125G/1008G Tasks: 158, 2020 thr, 654 kthr; 28 running
Swp[|||||] 920M/16.0G Load average: 24.98 21.32 21.27
Uptime: 55 days, 14:38:50
```

Main I/O

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU	CPU%	MEM%	TIME+	Command
313966		20	0	217M	3504	2972	S	63	0.0	0.0	0:00:06	└ /bin/bash /var/spool/slurmd/job45397505/slurm_script
314048		20	0	218M	4316	2932	S	63	0.0	0.0	0:00:02	└ bash /home/ndemand/data/sys/dashboard/batch_connect/sys/ycrc_rstudio_server/output/e84783fc-4713-4a4
314586		20	0	1765M	22468	13832	S	63	0.0	0.0	0:00:05	└ Apptainer runtime parent
314599		20	0	243M	16084	11532	S	63	0.0	0.0	0:01:57	└ /usr/lib/rstudio-server/bin/rserver --server-working-dir=/home/ndemand/rstudio-server --server
314668		20	0	243M	16084	11532	S	63	0.0	0.0	0:00:51	└ /usr/lib/rstudio-server/bin/rserver --server-working-dir=/home/ondemand/rstudio-server --ser
314669		20	0	243M	16084	11532	S	63	0.0	0.0	0:00:33	└ /usr/lib/rstudio-server/bin/rserver --server-working-dir=/home/ondemand/rstudio-server --ser
314670		20	0	243M	16084	11532	S	63	0.0	0.0	0:00:54	└ /usr/lib/rstudio-server/bin/rserver --server-working-dir=/home/ondemand/rstudio-server --ser
314740		20	0	1235M	566M	33456	S	63	2.0	0.1	0:33.32	└ rsession --r-libs-user /home/ndemand/rstudio-server --session
314741		20	0	1235M	566M	33456	S	63	0.0	0.1	0:00:00	└ rsession --r-libs-user /home/ondemand/rstudio-server --sess
314742		20	0	1235M	566M	33456	S	63	0.0	0.1	0:00:00	└ rsession --r-libs-user /home/ondemand/rstudio-server --sess
314743		20	0	1235M	566M	33456	S	63	0.0	0.1	0:00:29	└ rsession --r-libs-user /home/ondemand/rstudio-server --sess
314745		20	0	1235M	566M	33456	S	63	0.0	0.1	0:00:07	└ rsession --r-libs-user /home/ondemand/rstudio-server --sess
314749		20	0	1235M	566M	33456	S	63	0.0	0.1	0:00:12	└ rsession --r-libs-user /home/ondemand/rstudio-server --sess
314750		20	0	1235M	566M	33456	S	63	0.0	0.1	0:00:00	└ rsession --r-libs-user /home/ondemand/rstudio-server --sess
314759		20	0	1235M	566M	33456	S	63	0.0	0.1	0:00:09	└ rsession --r-libs-user /home/ondemand/rstudio-server --sess
317937		20	0	1235M	566M	33456	S	63	0.0	0.1	0:00:04	└ rsession --r-libs-user /home/ondemand/rstudio-server --sess
317939		20	0	23800	4480	3710	S	63	0.0	0.0	0:00:17	└ bash --login --posix
319982		20	0	20.0G	12.9G	1408	R	63	98.8	1.3	2:37.38	└ ./STAR_test/STAR/bin/Linux_x86_64_static/STAR --runMode genomeGenerate --geno
320190		20	0	20.0G	12.9G	1408	R	63	11.7	1.3	0:08.79	└ ./STAR_test/STAR/bin/Linux_x86_64_static/STAR --runMode genomeGenerate --g
320191		20	0	20.0G	12.9G	1408	R	63	11.7	1.3	0:08.79	└ ./STAR_test/STAR/bin/Linux_x86_64_static/STAR --runMode genomeGenerate --g
320192		20	0	20.0G	12.9G	1408	R	63	13.0	1.3	0:08.80	└ ./STAR_test/STAR/bin/Linux_x86_64_static/STAR --runMode genomeGenerate --g
320193		20	0	20.0G	12.9G	1408	R	63	13.0	1.3	0:08.81	└ ./STAR_test/STAR/bin/Linux_x86_64_static/STAR --runMode genomeGenerate --g
320194		20	0	20.0G	12.9G	1408	R	63	11.7	1.3	0:08.78	└ ./STAR_test/STAR/bin/Linux_x86_64_static/STAR --runMode genomeGenerate --g
320195		20	0	20.0G	12.9G	1408	R	63	11.1	1.3	0:08.79	└ ./STAR_test/STAR/bin/Linux_x86_64_static/STAR --runMode genomeGenerate --g
320196		20	0	20.0G	12.9G	1408	R	63	13.0	1.3	0:08.80	└ ./STAR_test/STAR/bin/Linux_x86_64_static/STAR --runMode genomeGenerate --g
317938		20	0	243M	16084	11532	S	63	0.0	0.0	0:00:00	└ ./STAR_test/STAR/bin/Linux_x86_64_static/STAR --runMode genomeGenerate --g
314600		20	0	1765M	22468	13832	S	63	0.0	0.0	0:00:00	└ Apptainer runtime parent
314601		20	0	1765M	22468	13832	S	63	0.0	0.0	0:00:00	└ Apptainer runtime parent
314603		20	0	1765M	22468	13832	S	63	0.0	0.0	0:00:00	└ Apptainer runtime parent
314606		20	0	1765M	22468	13832	S	63	0.0	0.0	0:00:00	└ Apptainer runtime parent
314608		20	0	1765M	22468	13832	S	63	0.0	0.0	0:00:00	└ Apptainer runtime parent
314610		20	0	213M	15432	1220	S	63	0.0	0.0	0:00:76	└ /usr/libexec/apptainer/bin/squashfuse_ll -f -o allow_other,ro,uid=21880,gid=10954,offset=45056 /proc/
314612		20	0	213M	15432	1220	S	63	0.0	0.0	0:00:32	└ /usr/libexec/apptainer/bin/squashfuse_ll -f -o allow_other,ro,uid=21880,gid=10954,offset=45056 /pr
314613		20	0	213M	15432	1220	S	63	0.0	0.0	0:00:32	└ /usr/libexec/apptainer/bin/squashfuse_ll -f -o allow_other,ro,uid=21880,gid=10954,offset=45056 /pr
318356		20	0	213M	15432	1220	S	63	0.0	0.0	0:00:11	└ /usr/libexec/apptainer/bin/squashfuse_ll -f -o allow_other,ro,uid=21880,gid=10954,offset=45056 /pr
314611		20	0	1765M	22468	13832	S	63	0.0	0.0	0:00:00	└ Apptainer runtime parent
314616		20	0	1765M	22468	13832	S	63	0.0	0.0	0:00:00	└ Apptainer runtime parent

F1Help F2Setup F3Search F4Filter F5List F6SortBy F7Nice -F8Nice +F9Kill F10Quit

Resource Management

Nicely performing GPU job

NVIDIA-SMI 555.42.06			Driver Version: 555.42.06		CUDA Version: 12.5		
GPU			Persistence-M		CUDA Version: 12.5		
Fan	Name	Persistence-M	Bus-Id	Disp.A	Volatile	Incorr. ECC	
		Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute M.	
==	==	==	==	==	==	==	==
0	Tesla V100-PCIE-16GB	On	00000000:58:00.0	Off	94%	0	
N/A	41C P0	89W / 250W	352MiB / 16384MiB			Default	
						N/A	
1	Tesla V100-PCIE-16GB	On	00000000:D8:00.0	Off	92%	0	
N/A	38C P0	91W / 250W	352MiB / 16384MiB			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%)

=====

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

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 /v	
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	python	
1112702		20	0	2054M	140M	38872	S	0	0.0	0.0	0:07.15	wandb	
1112703		20	0	2054M	140M	38872	S	0	0.0	0.0	0:02.09	wandb	
1112704		20	0	2054M	140M	38872	S	0	0.0	0.0	0:01.94	wandb	
1112705		20	0	2054M	140M	38872	S	0	0.0	0.0	0:26.38	wandb	
1112706		20	0	2054M	140M	38872	S	0	0.0	0.0	0:06.57	wandb	
1112707		20	0	2054M	140M	38872	S	0	0.0	0.0	0:23.34	wandb	
1112708		20	0	2054M	140M	38872	S	0	0.0	0.0	0:03.18	wandb	
1112711		20	0	2054M	140M	38872	S	0	0.0	0.0	0:03.97	wandb	
1112712		20	0	2054M	140M	38872	S	0	0.0	0.0	0:00.00	wandb	
1112713		20	0	2054M	140M	38872	S	0	0.0	0.0	0:00.00	wandb	
1112714		20	0	2054M	140M	38872	S	0	0.7	0.0	0:19.08	wandb	
1112715		20	0	2054M	140M	38872	S	0	0.0	0.0	0:03.04	wandb	
1112719		20	0	2054M	140M	38872	S	0	0.0	0.0	0:01.86	wandb	
1112720		20	0	2054M	140M	38872	S	0	0.0	0.0	0:22.02	wandb	
1112721		20	0	2054M	140M	38872	S	0	0.0	0.0	0:15.06	wandb	
1112722		20	0	2054M	140M	38872	S	0	8.6	0.0	5:33.62	wandb	
1112723		20	0	2054M	140M	38872	S	0	0.0	0.0	0:08.55	wandb	
1112724		20	0	2054M	140M	38872	S	0	0.0	0.0	0:10.08	wandb	
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	wandb	
1112735		20	0	2054M	140M	38872	S	0	0.0	0.0	0:03.17	wandb	
1112736		20	0	2054M	140M	38872	S	0	0.0	0.0	0:04.42	wandb	
1112709		20	0	130G	109G	943M	S	0	0.0	29.0	0:09.00	python	

Resource Management

Summary and Best Practices

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

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

```
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 = 10000000, seed=seed)

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

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	
1	python image_flipper.py ./data/waterloo.jpg
2	python image_flipper.py ./data/victoria.jpg
3	python image_flipper.py ./data/paddington.jpg
4	python image_flipper.py ./data/charing_cross.jpg
5	python image_flipper.py ./data/euston.jpg
6	python image_flipper.py ./data/kings_cross.jpg

File: dsq-jobfile-2024-12-12.sh	
1	#!/bin/bash
2	#SBATCH --output dsq-jobfile-%A_%la-%N.out
3	#SBATCH --array 0-9
4	#SBATCH --job-name dsq-jobfile
5	#SBATCH -p day -c 1 -t 1:00:00 --mem-per-cpus=2G
6	
7	# DO NOT EDIT LINE BELOW
8	/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
9	

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

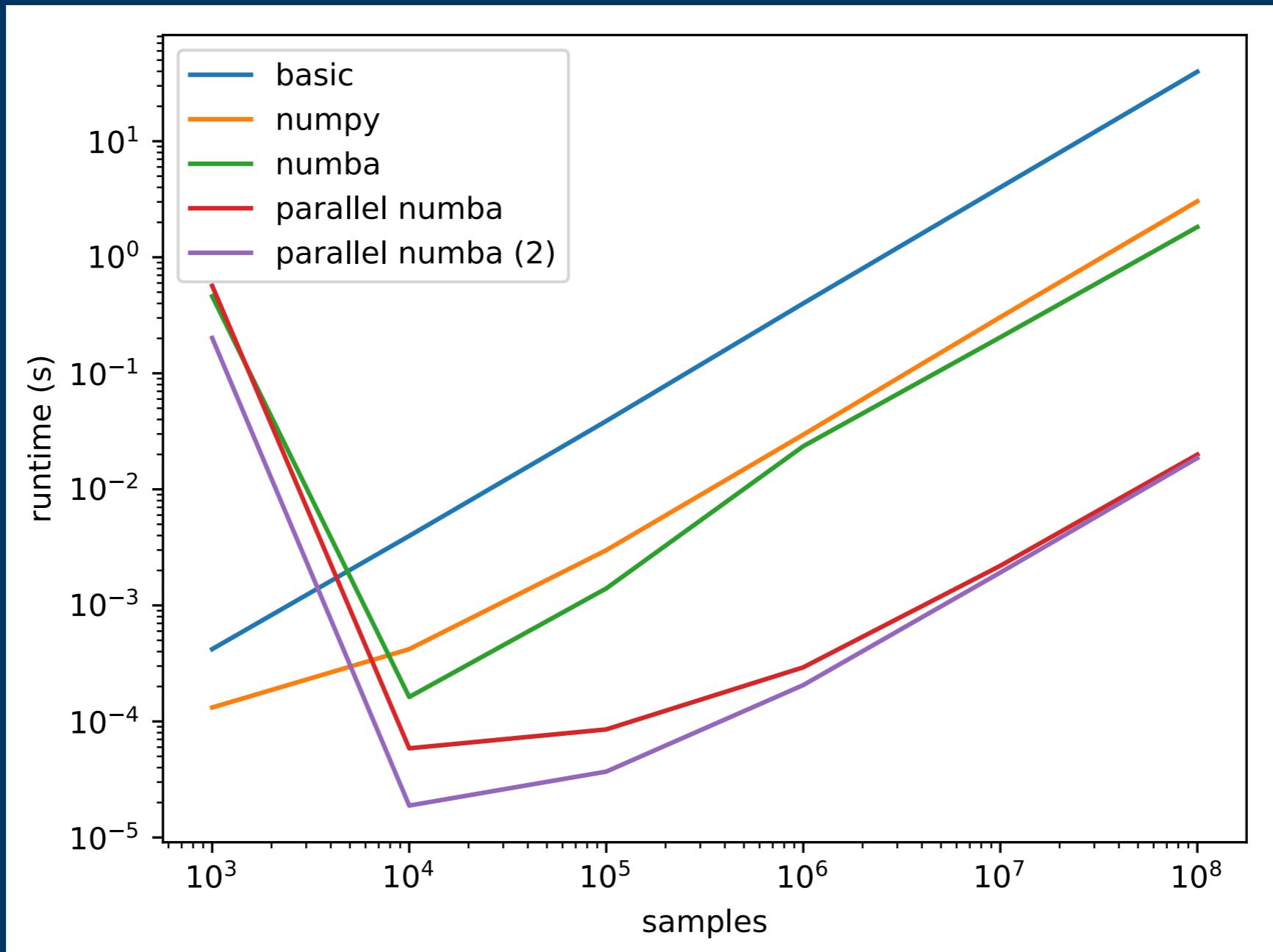
```
def numpy_mc(n_samples = 1000000):  
    np.random.seed(42)  
  
    x = np.random.random(size=n_samples)  
    y = np.random.random(size=n_samples)  
  
    r = np.power(np.power(x,2) + np.power(y,2), 0.5)  
    counter = r[r<1].size  
  
    return 4 * counter / n_samples
```

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

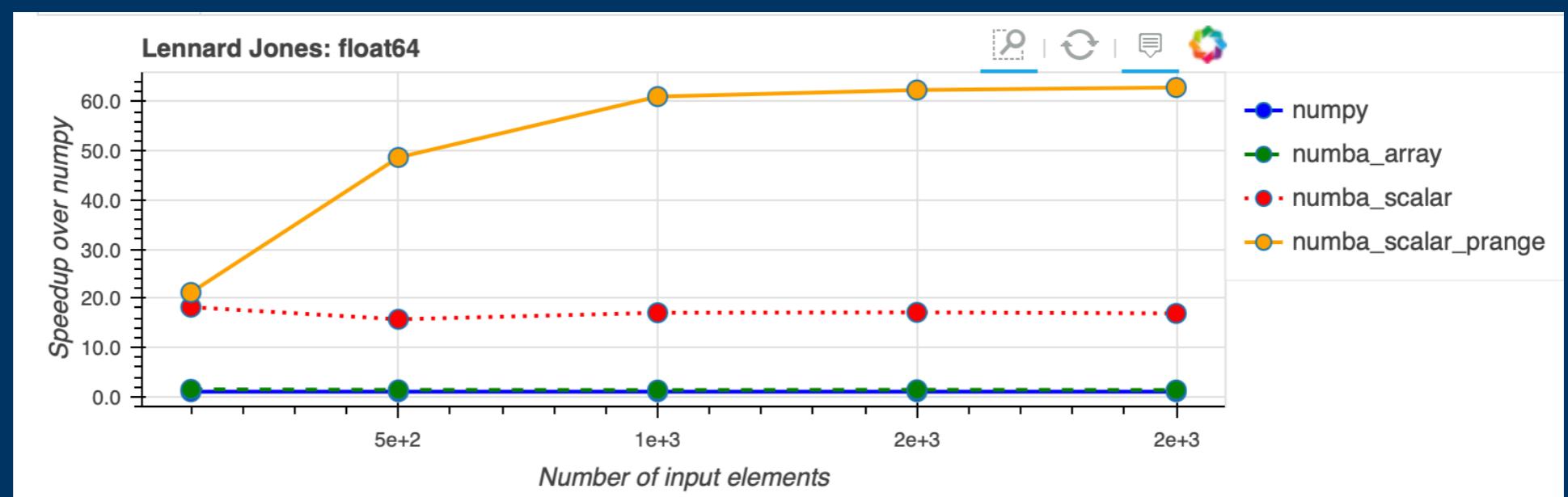
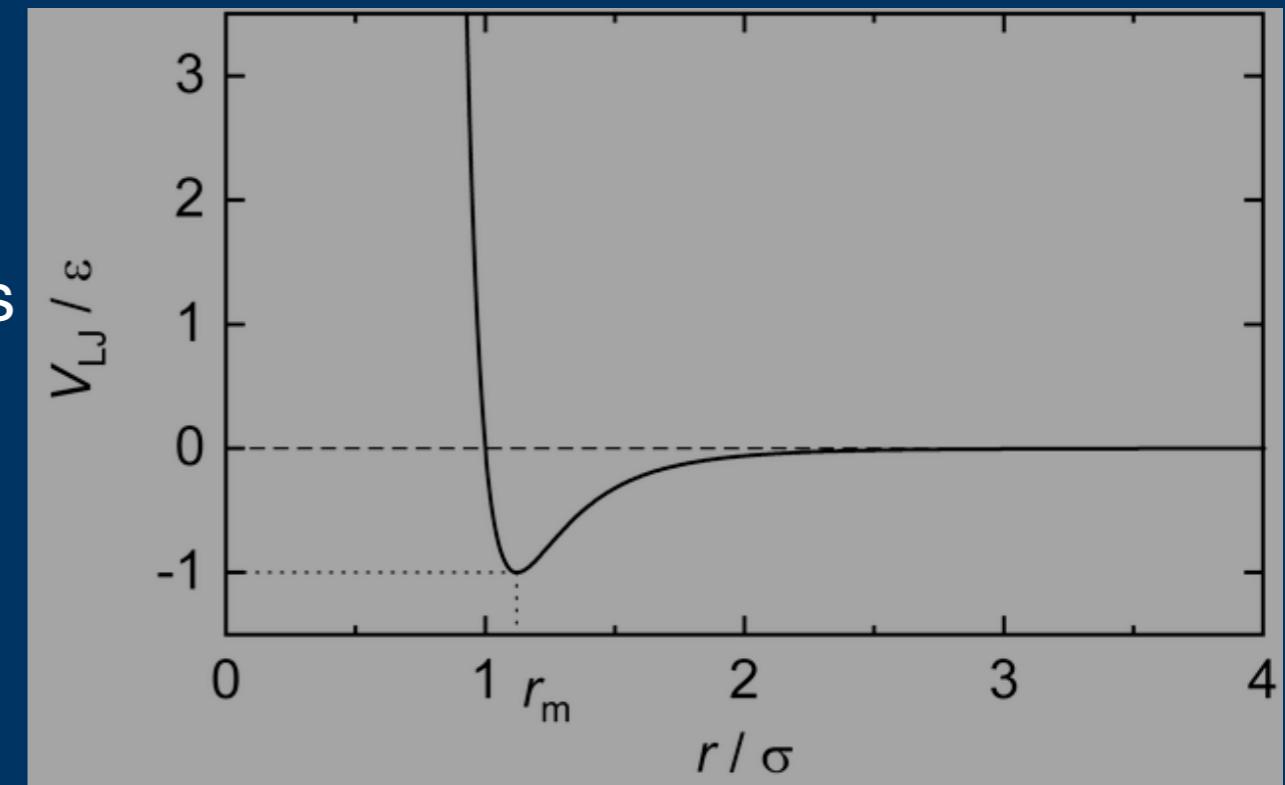
Monte Carlo Pi Calculation



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

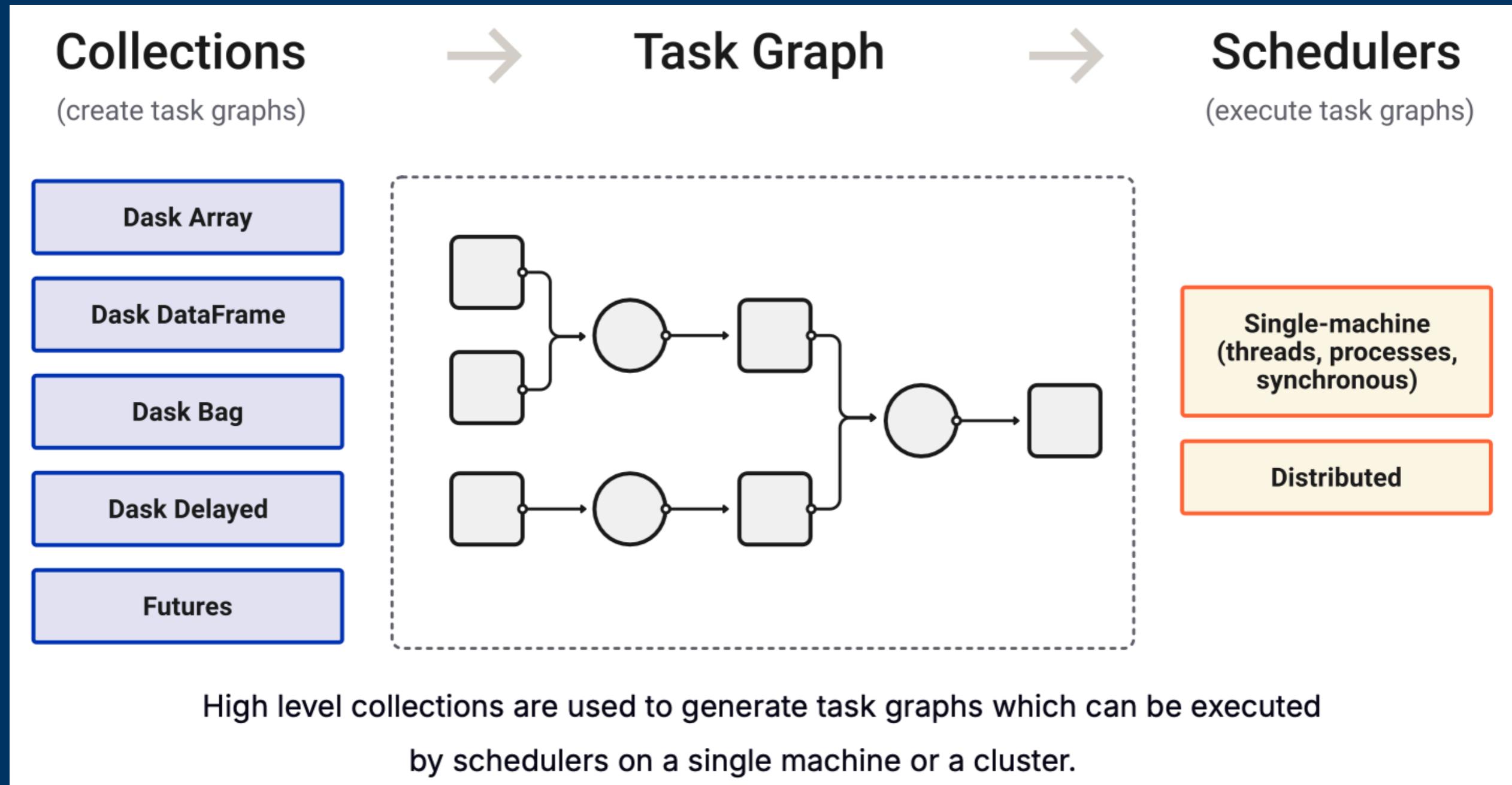
- Numba can take simple python code and dramatically improve speed
- This is especially useful for problems that cannot be easily vectorized to take advantage of numpy
- Numba benefits *greatly* from having defined datatypes
- For small jobs, the overhead of compiling can slow things down
- For frequently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





Exercise: Dask

Dask

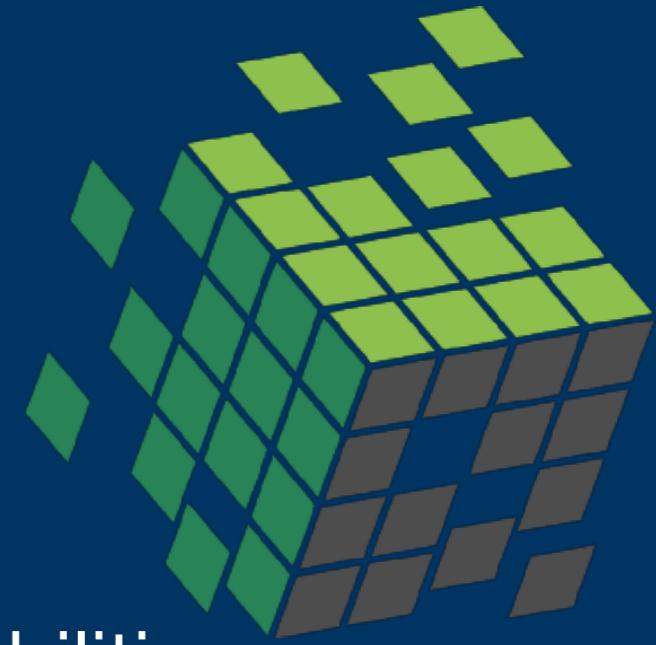
Takeaways



- 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

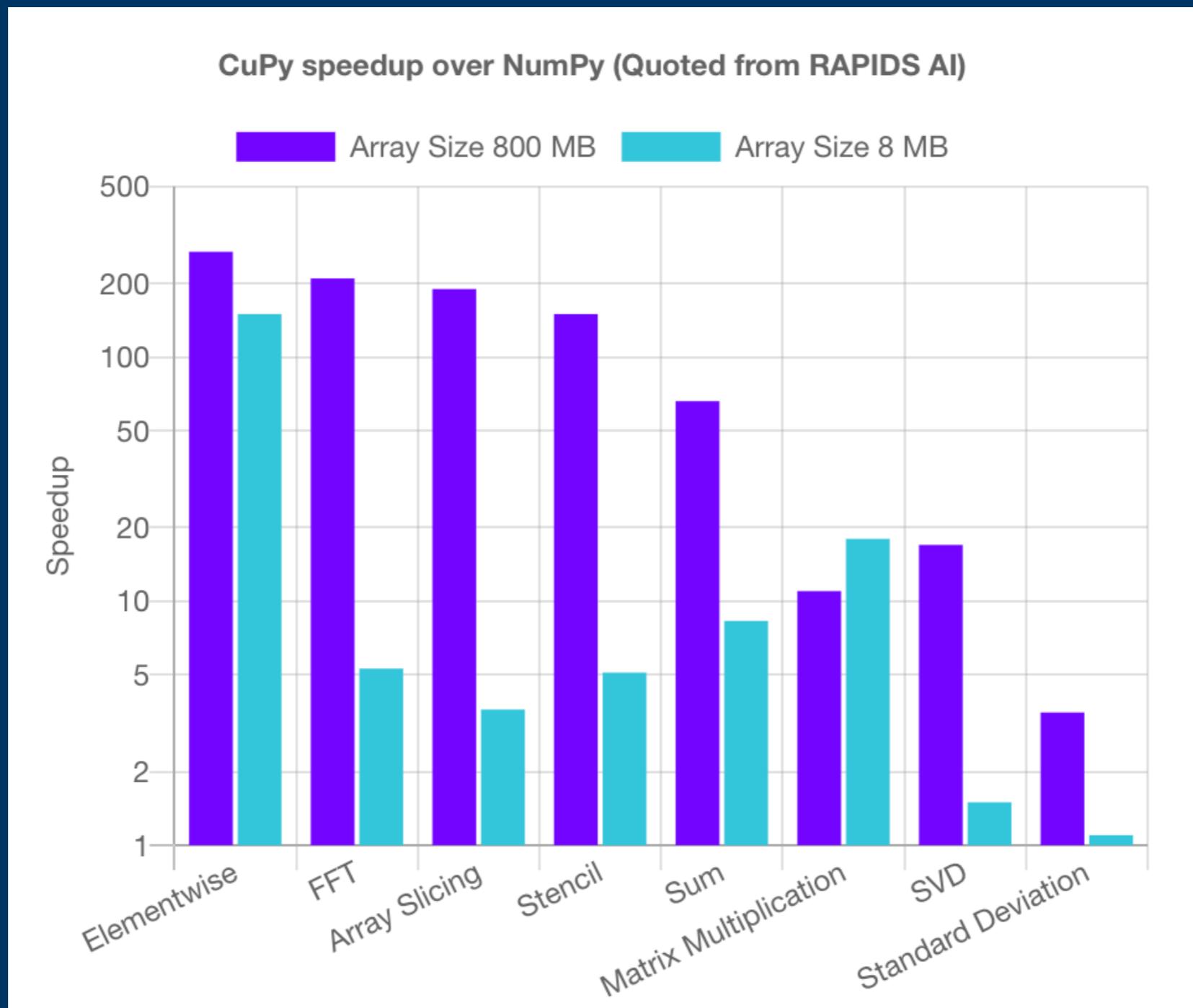
cupy.dev GPU accelerated Numpy



- 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...

