Lecture 14: Slurm (User)

Job Arrays, MPI, and CUDA

```
[vanilla@easley ~]$ git clone https://lobogit.unm.edu/CARC/workshops.git
Cloning into 'workshops'...
remote: Enumerating objects: 132, done.
remote: Counting objects: 100% (75/75), done.
remote: Compressing objects: 100% (43/43), done.
remote: Total 132 (delta 33), reused 74 (delta 32), pack-reused 57
Receiving objects: 100% (132/132), 57.58 KiB | 3.60 MiB/s, done.
Resolving deltas: 100% (51/51), done.
```

Rather than make you write shell scripts lets just download some we wrote for this workshop…

```
[vanilla@easley ~]$ tree workshops
workshops/
├── intro_workshop
│   ├── code
│   │   ├── calcPiMPI.py
│   │   ├── calcPiSerial.py
│   │   └── vecadd
│   │       ├── Makefile
│   │       ├── vecadd_gpu.cu
│   │       ├── vecadd_mpi_cpu
│   │       ├── vecadd_mpi_cpu.c
│   │       ├── vecaddmpi_cpu.sh
│   │       └── vecadd_mpi_gpu.c
│   ├── data
│   │   ├── H2O.gjf
│   │   └── step_sizes.txt
│   └── slurm
│       ├── calc_pi_array.sh
│       ├── calc_pi_mpi.sh
│       ├── calc_pi_parallel.sh
│       ├── calc_pi_serial.sh
│       ├── gaussian.sh
│       ├── hostname_mpi.sh
│       ├── vecadd_hopper.sh
│       ├── vecadd_xena.sh
│       ├── workshop_example2.sh
│       ├── workshop_example3.sh
│       └── workshop_example.sh
└── README.md
```

Run tree to see how the workshops directories are organized…

```
[vanilla@easley ~]$ tree workshops
workshops/
├── intro_workshop
│   ├── code
│   │   ├── calcPiMPI.py
│   │   ├── calcPiSerial.py
│   │   └── vecadd
│   │       ├── Makefile
│   │       ├── vecadd_gpu.cu
│   │       ├── vecadd_mpi_cpu
│   │       ├── vecadd_mpi_cpu.c
│   │       ├── vecaddmpi_cpu.sh
│   │       └── vecadd_mpi_gpu.c
│   ├── data
│   │   ├── H2O.gjf
│   │   └── step_sizes.txt
│   └── slurm
│       ├── calc_pi_array.sh
│       ├── calc_pi_mpi.sh
│       ├── calc_pi_parallel.sh
│       ├── calc_pi_serial.sh
│       ├── gaussian.sh
│       ├── hostname_mpi.sh
│       ├── vecadd_hopper.sh
│       ├── vecadd_xena.sh
│       ├── workshop_example2.sh
│       ├── workshop_example3.sh
│       └── workshop_example.sh
└── README.md
```

Run tree to see how the workshops directories are organized…

The workshop files are divided into “code”, “slurm”, and “data” directories.

```
[vanilla@easley intro_workshop]$ pwd
/users/vanilla/workshops/intro_workshop
[vanilla@easley intro_workshop]$ cat slurm/workshop_example1.sh
#!/bin/bash
#SBATCH --partition debug
#SBATCH --ntasks 4
#SBATCH --time 00:05:00
#SBATCH --job-name ws_example
#SBATCH --mail-user your_username@unm.edu
#SBATCH --mail-type ALL

srun hostname
```

Let's take a look at the workshop_example.sh script in the slurm directory…

```
[vanilla@easley intro_workshop]$ sbatch slurm/workshop_example1.sh
sbatch: Account not specified in script or ~/.default_slurm_account,
using latest project
Submitted batch job 5252
[vanilla@easley intro_workshop]$
```

We submit our slurm shell script with the sbatch command.

```
[vanilla@easley intro_workshop]$ sbatch slurm/workshop_example1.sh
sbatch: Account not specified in script or ~/.default_slurm_account,
using latest project
Submitted batch job 5252
[vanilla@easley intro_workshop]$
```

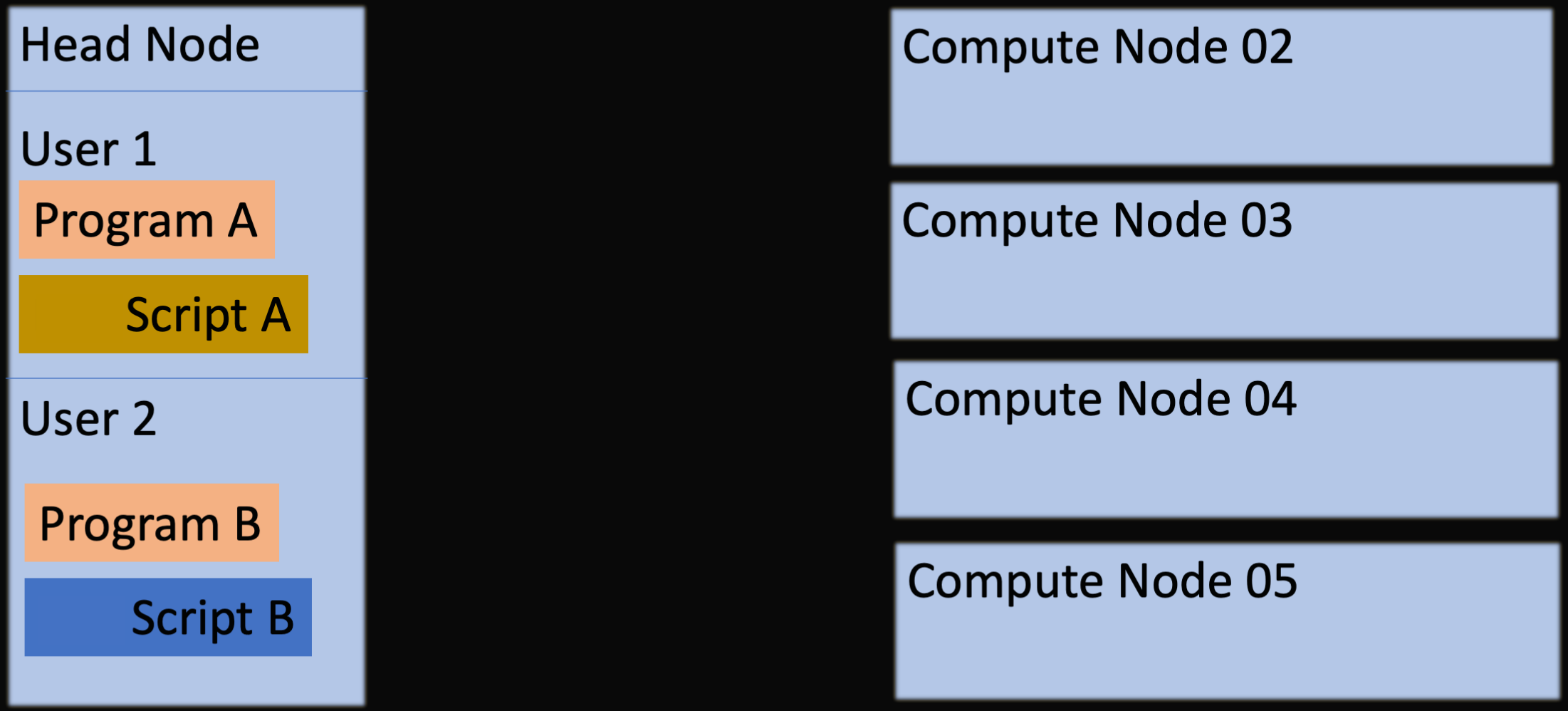
We submit our slurm shell script with the sbatch command.

Notice that the only output we get is a job id.

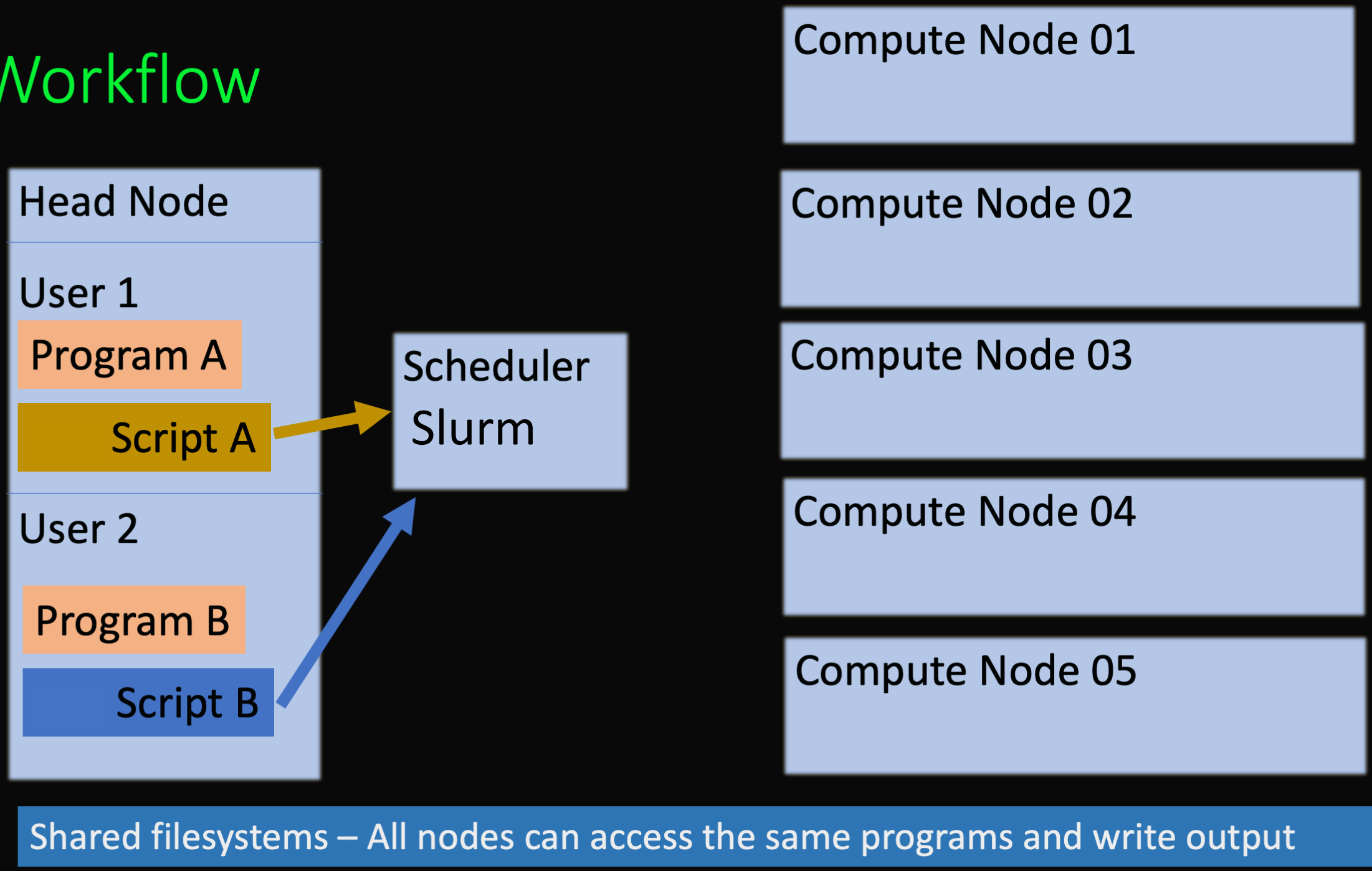
This indicates that the script was successfully sent to the scheduler.

The commands in the script will run as soon as the hardware requested is available.

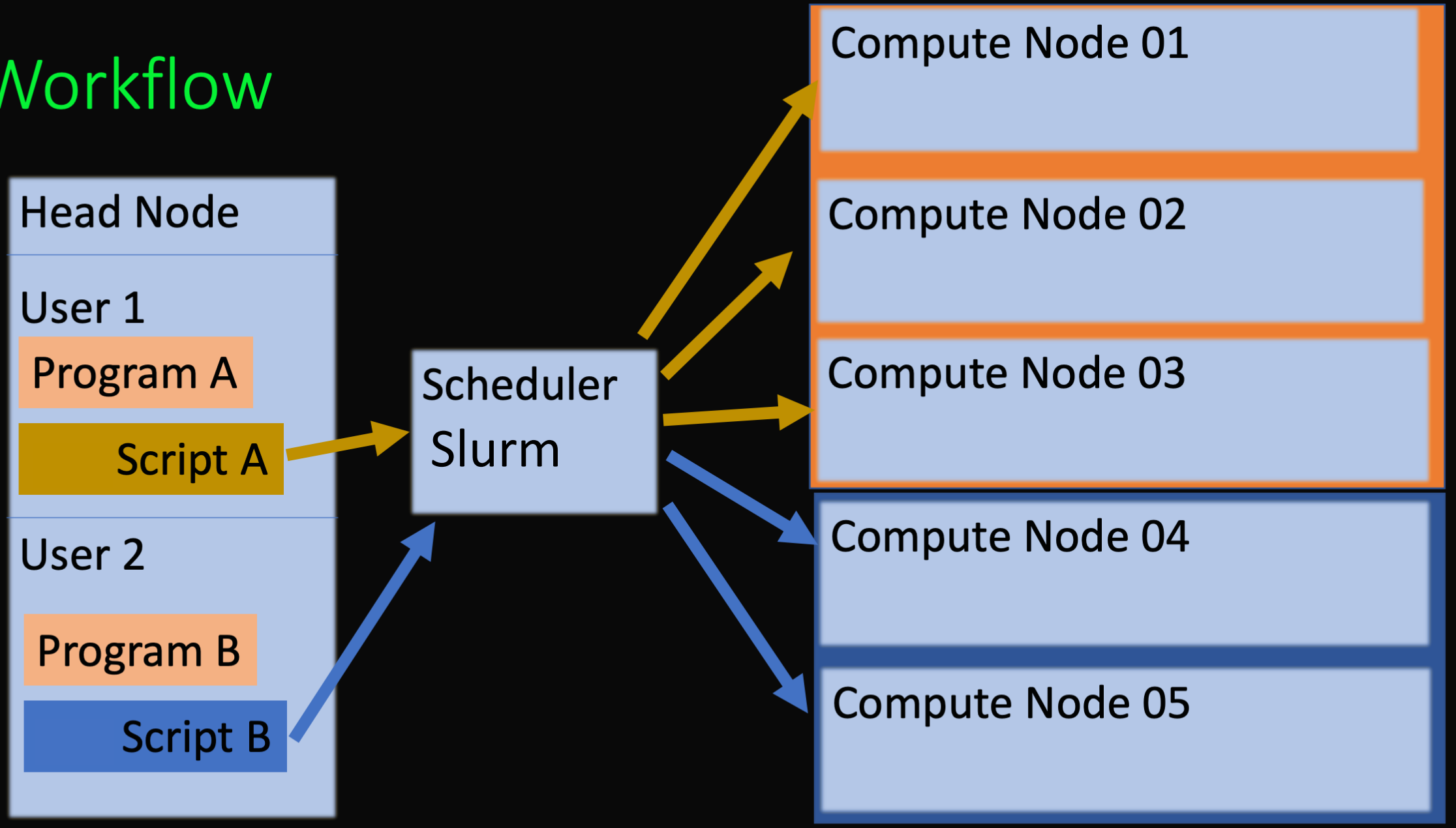
Workflow


Head Node
User 1
Program A
Script A
User 2
Program B
Script B
Compute Node 01
Compute Node 02
Compute Node 03
Compute Node 04
Compute Node 05
Shared filesystems – All nodes can access the same programs and write output

Workflow

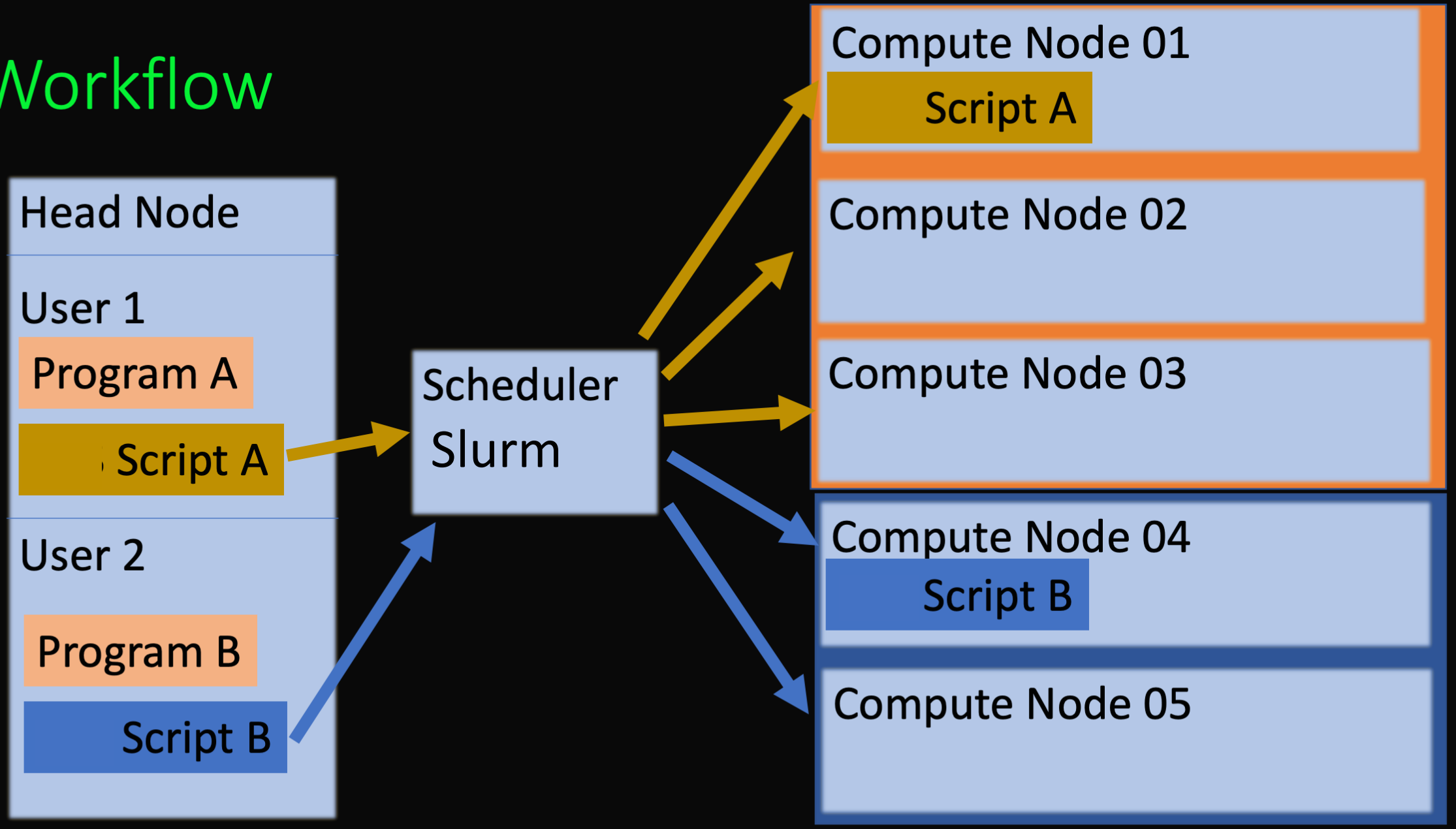

Head Node
User 1
Program A
Script A
User 2
Program B
Script B
Scheduler
Slurm
Compute Node 01
Compute Node 02
Compute Node 03
Compute Node 04
Compute Node 05
Shared filesystems – All nodes can access the same programs and write output

Workflow



Shared filesystems – All nodes can access the same programs and write output

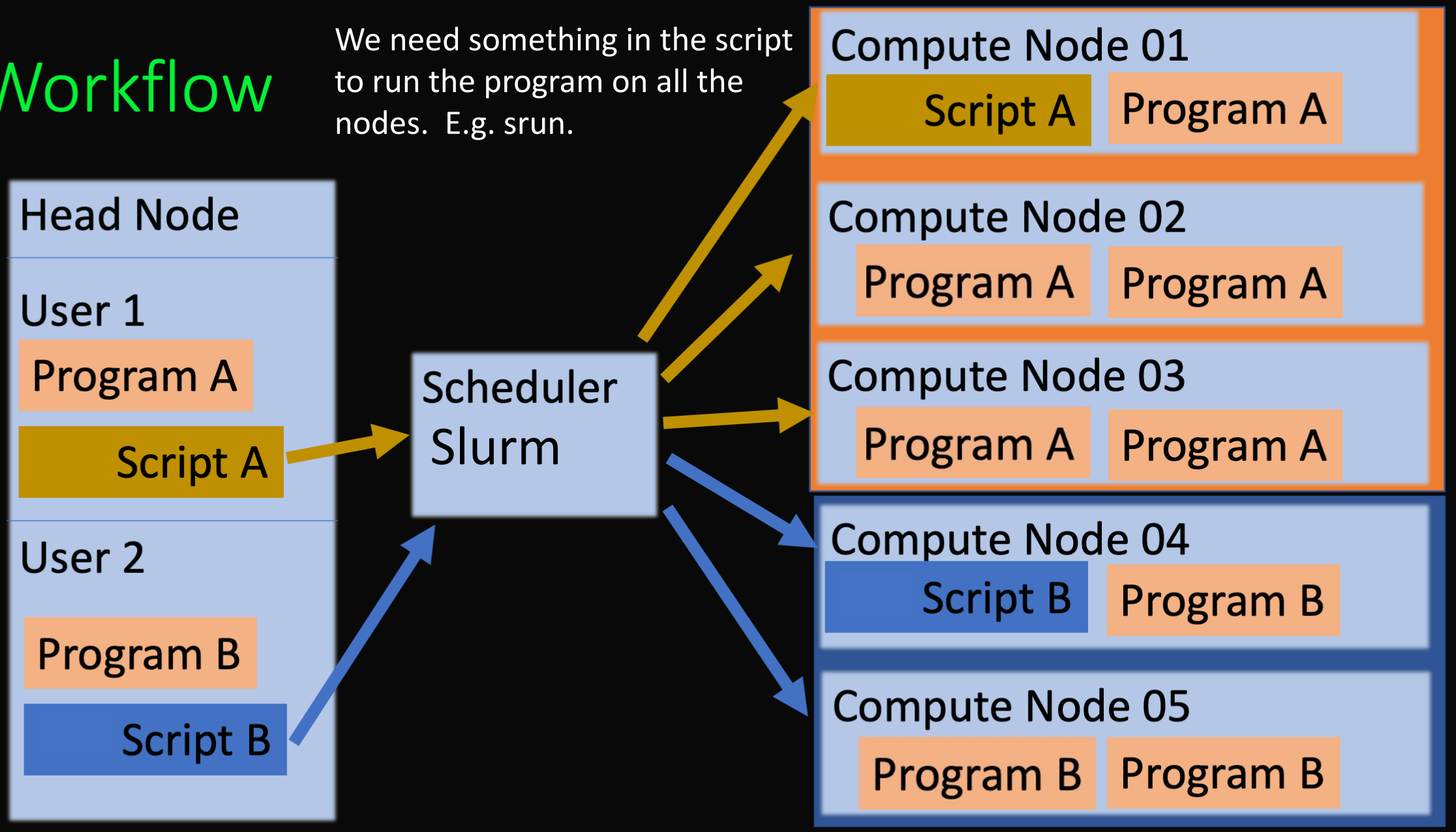
Workflow



Shared filesystems – All nodes can access the same programs and write output

Workflow

We need something in the script to run the program on all the nodes. E.g. srun.



Shared filesystems – All nodes can access the same programs and write output

```
[vanilla@easley intro_workshop]$ ls
code  data  pbs  slurm  slurm-5252.out
```

The **hostname** command is very fast so everyone's job should finish in a few seconds.

When it is finished you will have a new file named slurm-{your job id}.out.

```
[vanilla@easley intro_workshop]$ ls
code  data  pbs  slurm  slurm-5252.out
```


```
[vanilla@easley intro_workshop]$ cat slurm-5252.out
easley011
```

```
[vanilla@easley intro_workshop]$ ls
code  data  pbs  slurm  slurm-5252.out
```

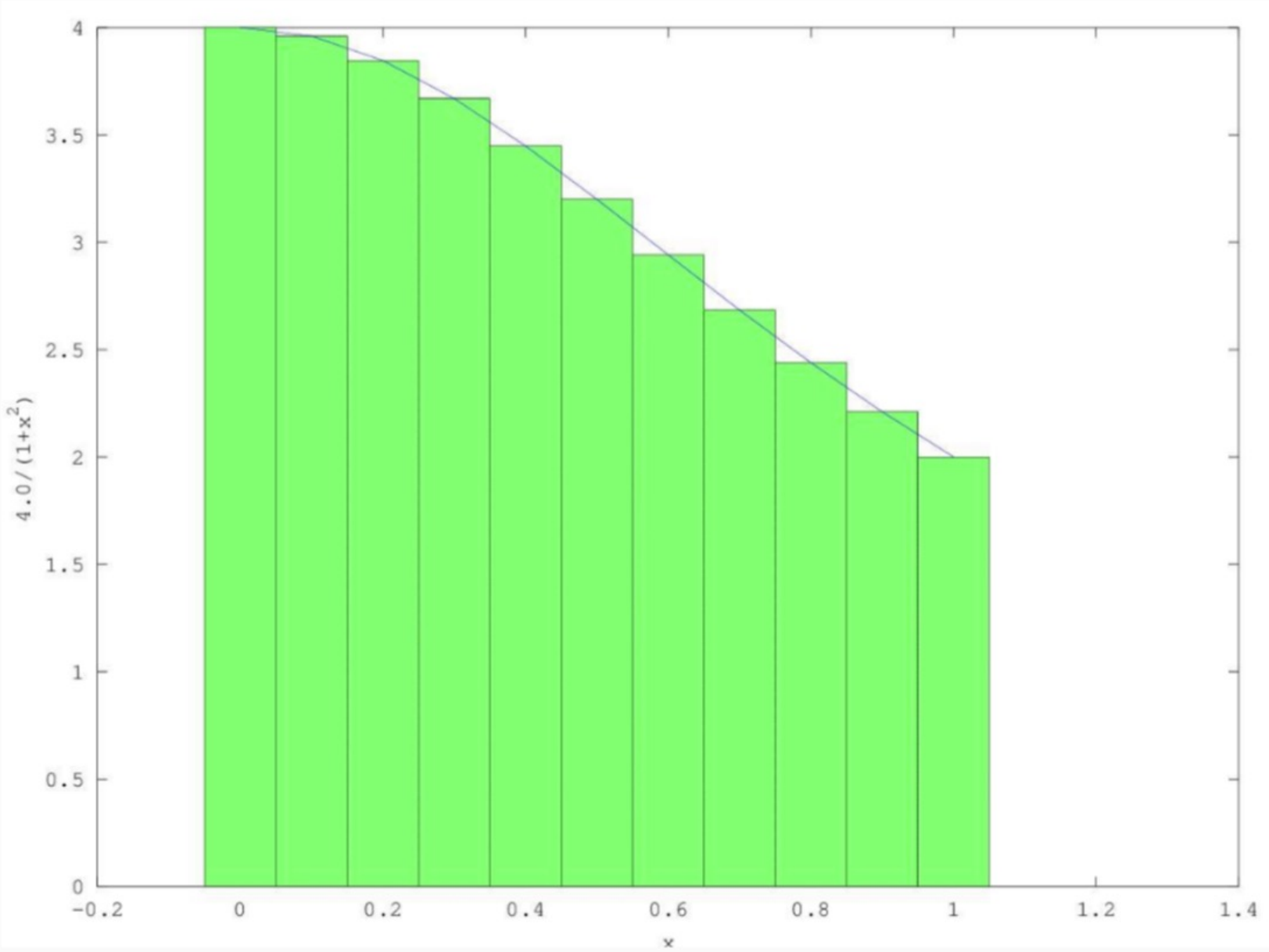

```
[vanilla@easley intro_workshop]$ cat slurm-5252.out
easley011
```

Why did it only run the program once instead of 4 times?

```
[vanilla@easley intro_workshop]$ sbatch slurm/workshop_example1.sh
sbatch: Account not specified in script or ~/.default_slurm_account,
using latest project
Submitted batch job 5252
[vanilla@easley intro_workshop]$
```

Take a look at the output file.

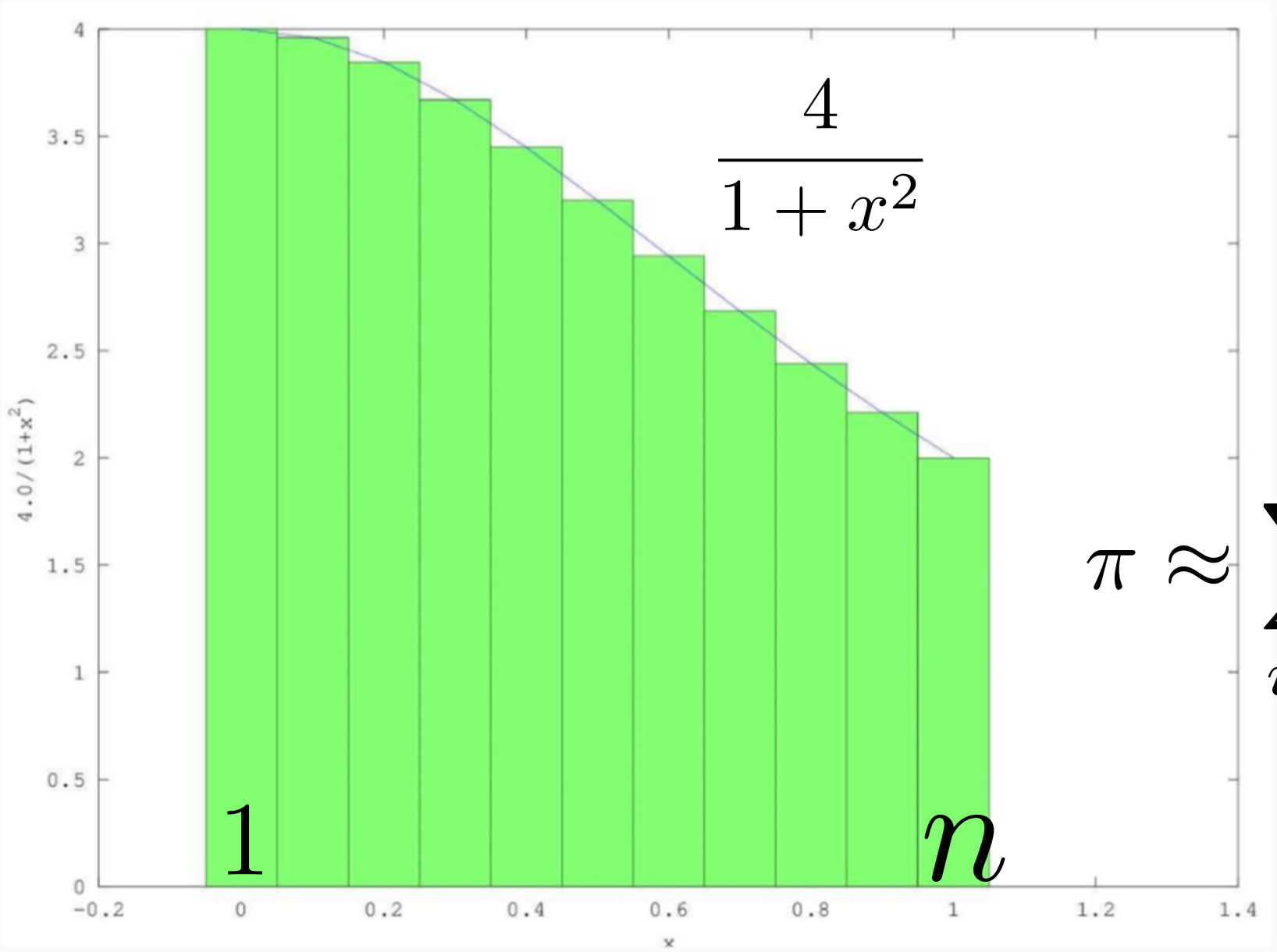
Serial Program to Calculate π



$$\int \frac{4}{1+x^2} = \pi$$

$$\Sigma_{i=1}^{N} \frac{4}{1+\left(\frac{i+\frac{1}{2}}{N}\right)^2} \approx \pi$$

Serial Program to Calculate π



$$\frac{4}{1+x^2}$$

$$w = \frac{1}{n}$$

$$\pi \approx \sum_{i=0}^{n} \frac{4}{1+(i+\frac{w}{2})^2}$$

```
# A program that calculates pi using the area under a curve
# The program checks the value of pi calculated against the
# value provided by numpy
import time
import sys
import numpy as np # Value of PI to compare to


def Pi(num_steps): #Function to calculate pi
      step = 1.0 / num_steps
      sum = 0
        for i in range(num_steps):
                  x = (i + 0.5) * step
                  sum = sum + 4.0 / (1.0 + x * x)
         pi = step * sum
         return pi
```

```
# Check that the caller gave us the number of steps to use
if len(sys.argv) != 2:
        print("Usage: ", sys.argv[0], " <number of steps>")
        sys.exit(1)

num_steps = int(sys.argv[1],10);

# Call function to calculate pi
start = time.time() #Start timing
pi = Pi(num_steps)
end = time.time() # End timing

# Print our estimation of pi, the difference from numpy's value,
and how long it took
print("Pi = %.20f, (Diff=%.20f) (calculated in %f secs with %d
steps)" %(pi, pi-np.pi, end - start, num_steps))
sys.exit(0)
```

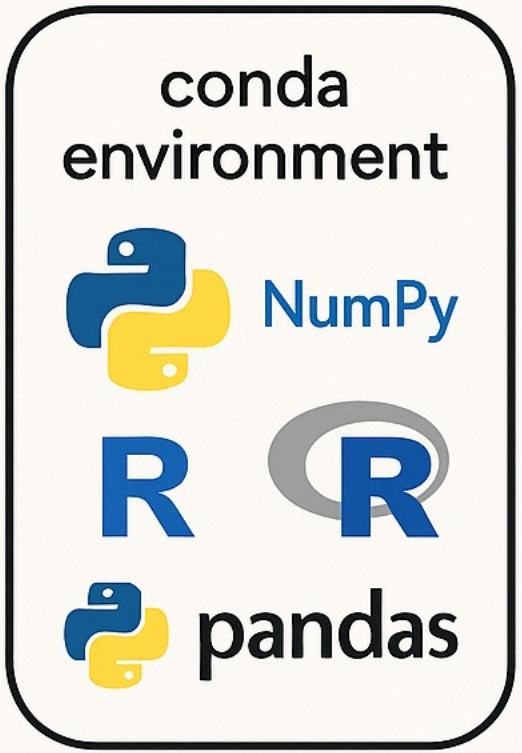
•**Pip**
•Installs packages from the Python Package Index (PyPI)
•Only manages Python packages
•Does *not* handle non-Python dependencies (like C libraries)

•**Conda**
•Installs packages from the Conda repository
•Can manage *both* Python and other language files
•Also manages environments (isolated directories for dependencies, Python versions, etc.)

Package Sources
•**Pip:** Downloads from PyPI
•**Conda:** Downloads precompiled binaries from Anaconda channels

CONDA

- Open source package management system
- Conda environments: Isolated, reproducible directories
- Manage dependencies across
  - Projects
  - Languages



```
[vanilla@easley intro_workshop]$ module load miniconda3
[vanilla@easley intro_workshop]$ conda create -n numpy numpy
```

Wait a while – introduce yourselves to your neighbor…

Conda allows you to install software into your home directory. In this case we need the numerical python libraries for calcPiSerial.py

To save space I've created a single environment for everyone to use.

You don't need to create your own.

Let's experiment with a program that does slightly more than print the hostname.

```
[vanilla@easley intro_workshop]$ source activate intro_workshop
[vanilla@easley intro_workshop]$ srun --partition debug python
code/calcPiSerial.py 10
srun: Using account 2016199 from ~/.default_slurm_account
You have not been allocated GPUs. To request GPUs, use the -G
option in your submission script.
Pi = 3.142425985001098709 40, (Diff=0.0008333314113055934 1)
(calculated in 0.000005 secs with 10 steps)
```

Activate the numpy environment and
Run calcPiSerial.py on a compute node.

For our example program the more steps it takes the more accurate it is, but the longer it takes.

```
[vanilla@easley intro_workshop]$ nano slurm/calc_pi_serial.sh

        #!/bin/bash

        #SBATCH --partition debug
        #SBATCH --ntasks 1
        #SBATCH --time 00:05:00
        #SBATCH --job-name calc_pi_serial
        #SBATCH --mail-user your_username@unm.edu
        #SBATCH --mail-type ALL

        module load miniconda3
        source activate intro_workshop

        srun python code/calcPiSerial.py 1000000000
```

```
[vanilla@easley intro_workshop]$ source deactivate
[vanilla@easley intro_workshop]$ sbatch slurm/calc_pi_serial.sh
 sbatch: Using account 2016199 from ~/.default_slurm_account
Submitted batch job 5263
vanilla@easley:~/workshops/intro_workshop$ squeue --me
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
5263     debug calc_pi_  vanilla  R       0:44      1 easley011
```

Edit slurm/calc_pi_serial.sh.
Change the email address to your address and submit the script.

Then enter squeue --me to see the job status.

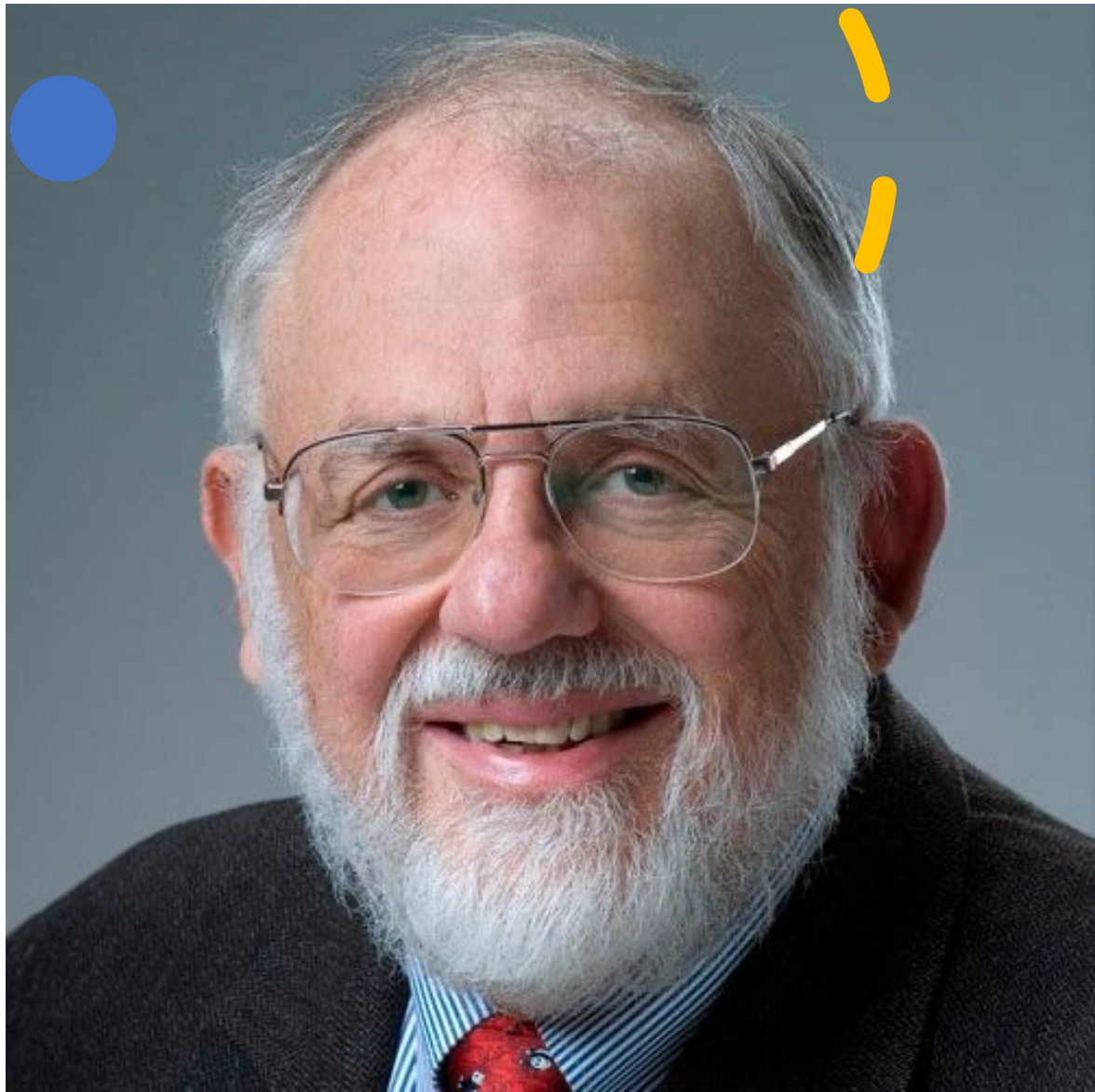
Take a look at the job output. Tell me how long it took to calculate.

Parallelism – Embarrassingly Parallel

- Embarrassingly parallel (Cleve Moler) are problems that are really really easy to speed up with mode CPUs.

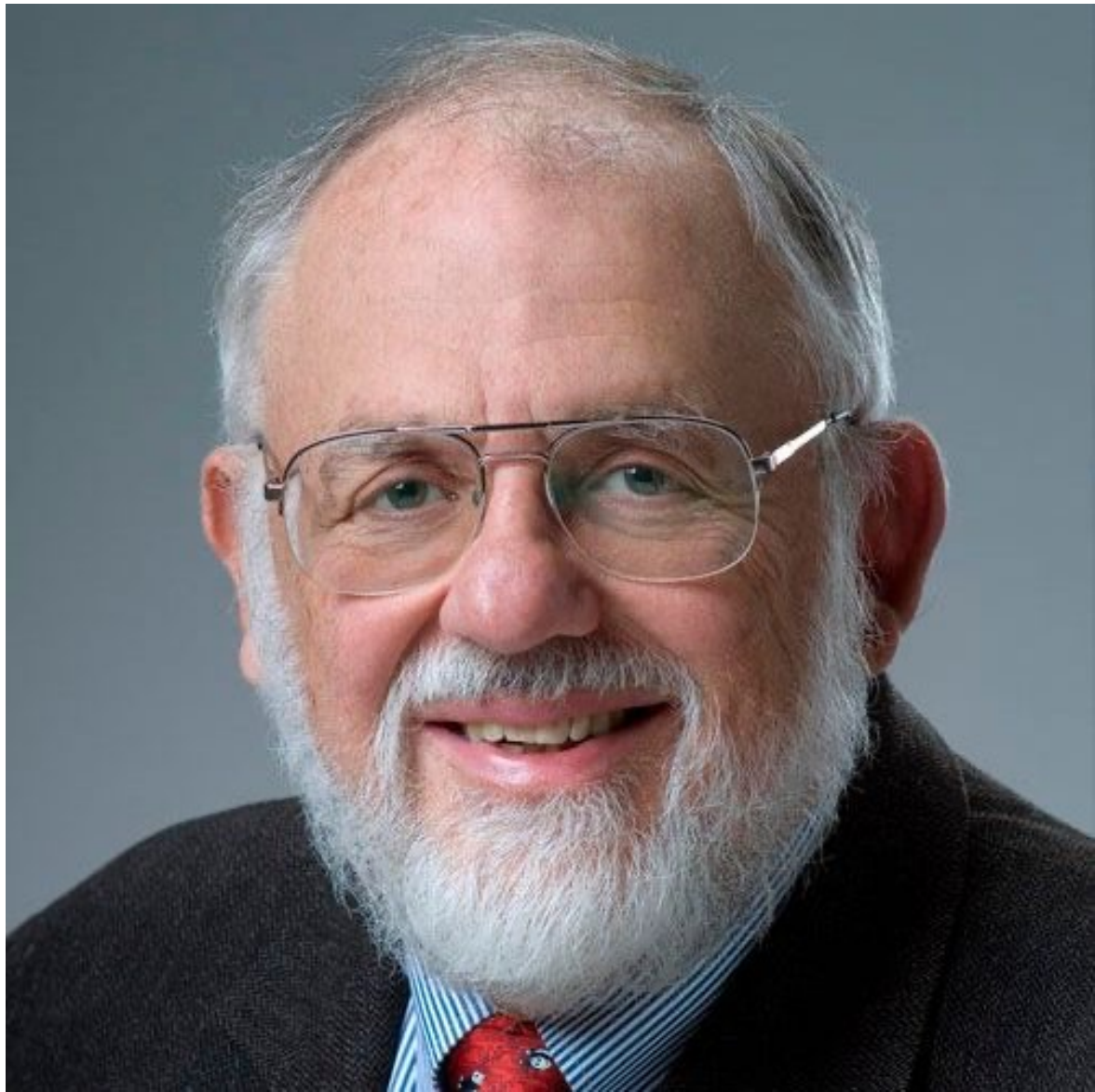
- The most common example is that you have a program that runs in serial and takes some input file, processes it, and produces some output.

- The problem is that you have 1,000 of the input files and want to run your program on each one.



Parallelism – Embarrassingly Parallel

This is ”embarrassing” because all you have to do is run 1,000 copies of your program on 1,000 CPUs each with a different input file and you are done.



SLURM ARRAYS

- One way to run the 1,000 copies of your program on 1,000 different inputs would be to write 1,000 slurm scripts each specifying a different input to your program and then sbatch submit them all. (this would work but there are better ways).

- SLURM arrays are used to schedule a lot of jobs with one slurm script.

```
[vanilla@easley intro_workshop]$ nano slurm/calc_pi_array.sh
#!/bin/bash
#SBATCH --partition debug
#SBATCH --ntasks 1
#SBATCH --time 00:05:00
#SBATCH --job-name calc_pi_array
#SBATCH --mail-user your_username@unm.edu
#SBATCH --mail-type ALL
#SBATCH --array=1-12%3

echo "$HOSTNAME - $SLURM_ARRAY_TASK_ID"

module load miniconda3
source activate intro_workshop

NUM_STEPS="${SLURM_ARRAY_TASK_ID}0000"
echo "Calculating pi with $NUM_STEPS..."
cd $SLURM_SUBMIT_DIR
python code/calcPiSerial.py $NUM_STEPS
```

Requires some annoying bash scripting.

$something means get the value of the variable “something”

```
[vanilla@easley intro_workshop]$ nano slurm/calc_pi_array.sh
#!/bin/bash
#SBATCH --partition debug
#SBATCH --ntasks 1
#SBATCH --cpus-per-task 3
#SBATCH --time 00:05:00
#SBATCH --job-name calc_pi_array
#SBATCH --mail-user your_username@unm.edu
#SBATCH --mail-type ALL
#SBATCH --array=1-12%3

echo "$HOSTNAME - $SLURM_ARRAY_TASK_ID"

module load miniconda3
source activate intro_workshop

NUM_STEPS="${SLURM_ARRAY_TASK_ID}0000"
echo "Calculating pi with $NUM_STEPS..."
cd $SLURM_SUBMIT_DIR
python code/calcPiSerial.py $NUM_STEPS
```

--array says

1) **run 12 separate jobs**
2) **Store the count of the job in the variable SLURM_ARRAY_TASK_ID**

Notice there is no srun.
Why not?

```
[vanilla@easley intro_workshop]$ sbatch slurm/calc_pi_array.sh
 sbatch: Using account 2016199 from ~/.default_slurm_account
Submitted batch job 5263
vanilla@easley:~/workshops/intro_workshop$ squeue -me
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
5263     debug calc_pi_  vanilla  R       0:44      1 easley011
```

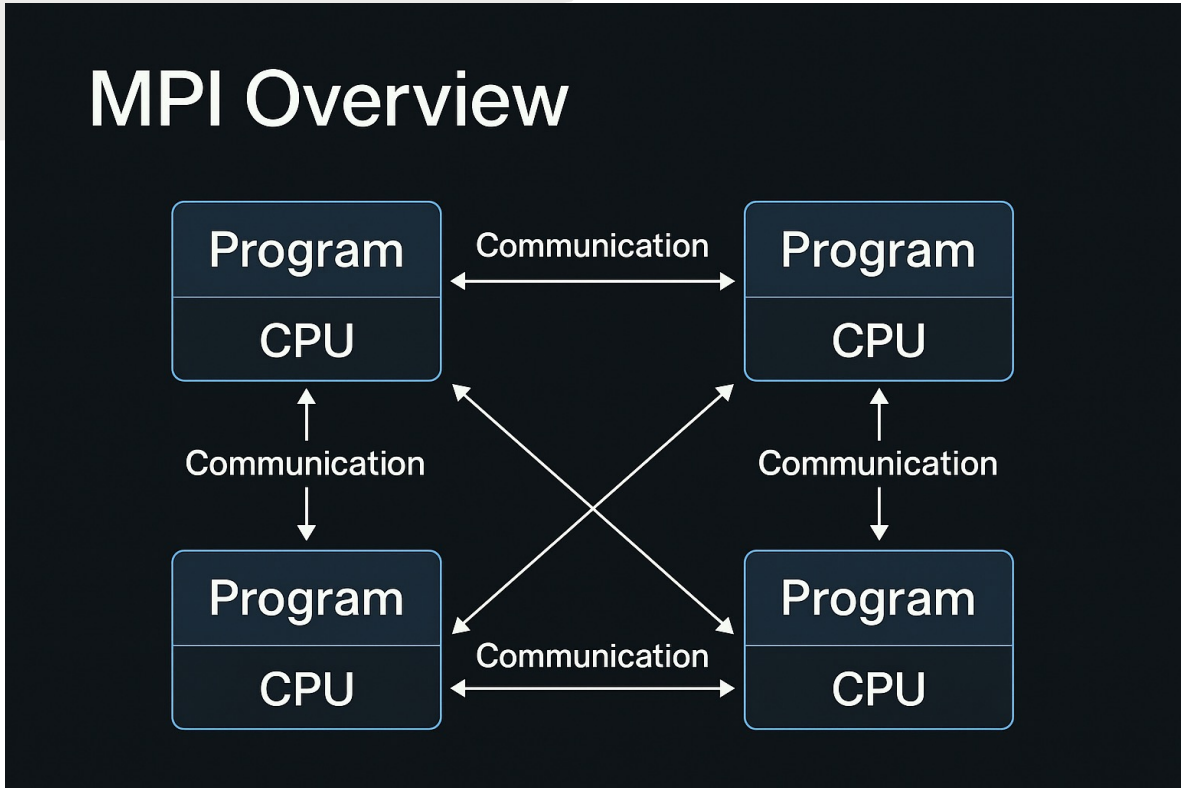
Submit the array script.

Then enter squeue --me to see the job status.

Take a look at the job output. (How many output files do you have?)

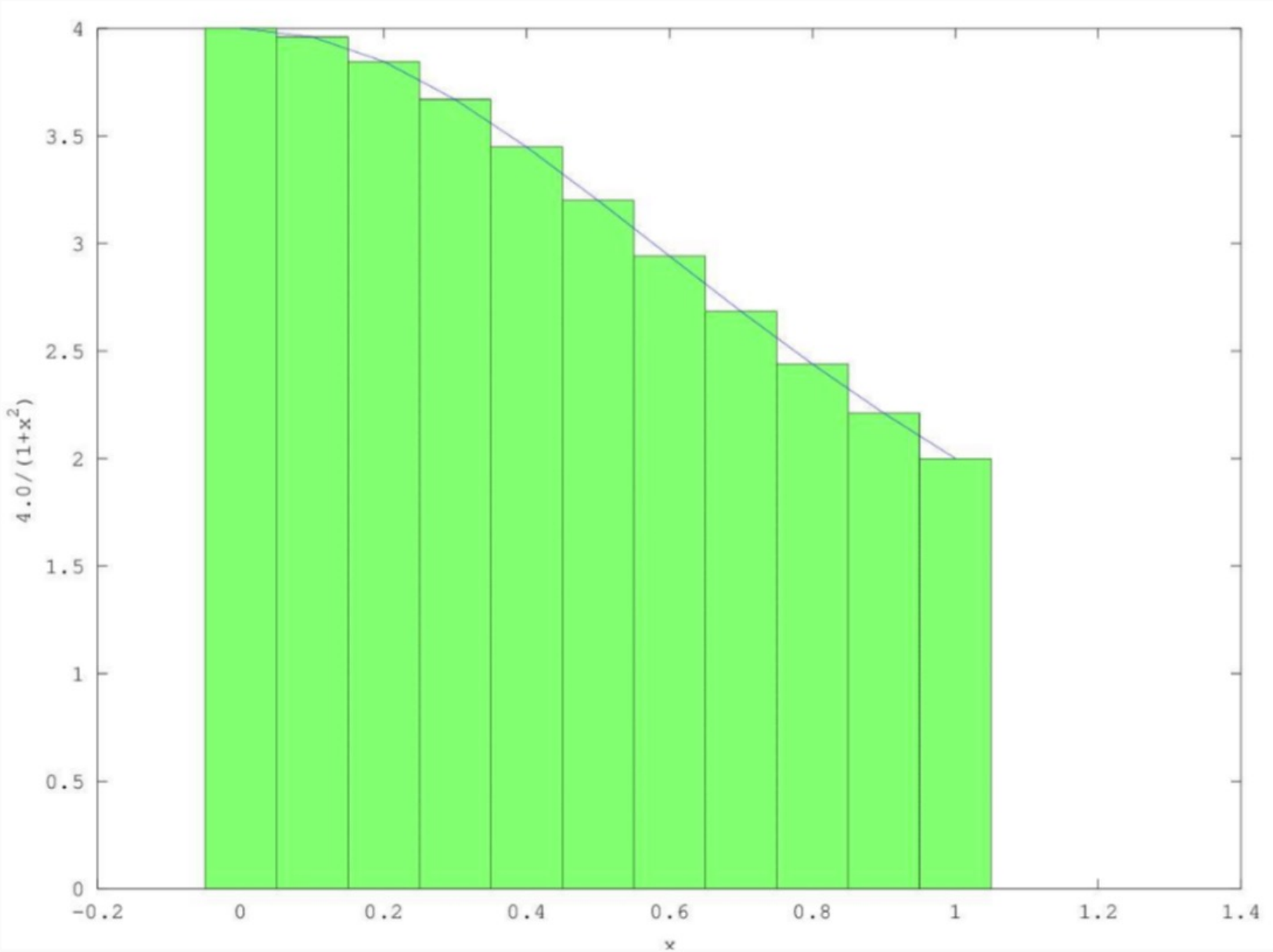
Coupled Parallelism

- Message Passing Interface (MPI)
- Coupled problems are those where the CPUs need to work together to solve a problem by communicating with each other.
- Many commercial and research programs designed to run on HPC systems like CARC use a library called the message passing interface (MPI) to do this.

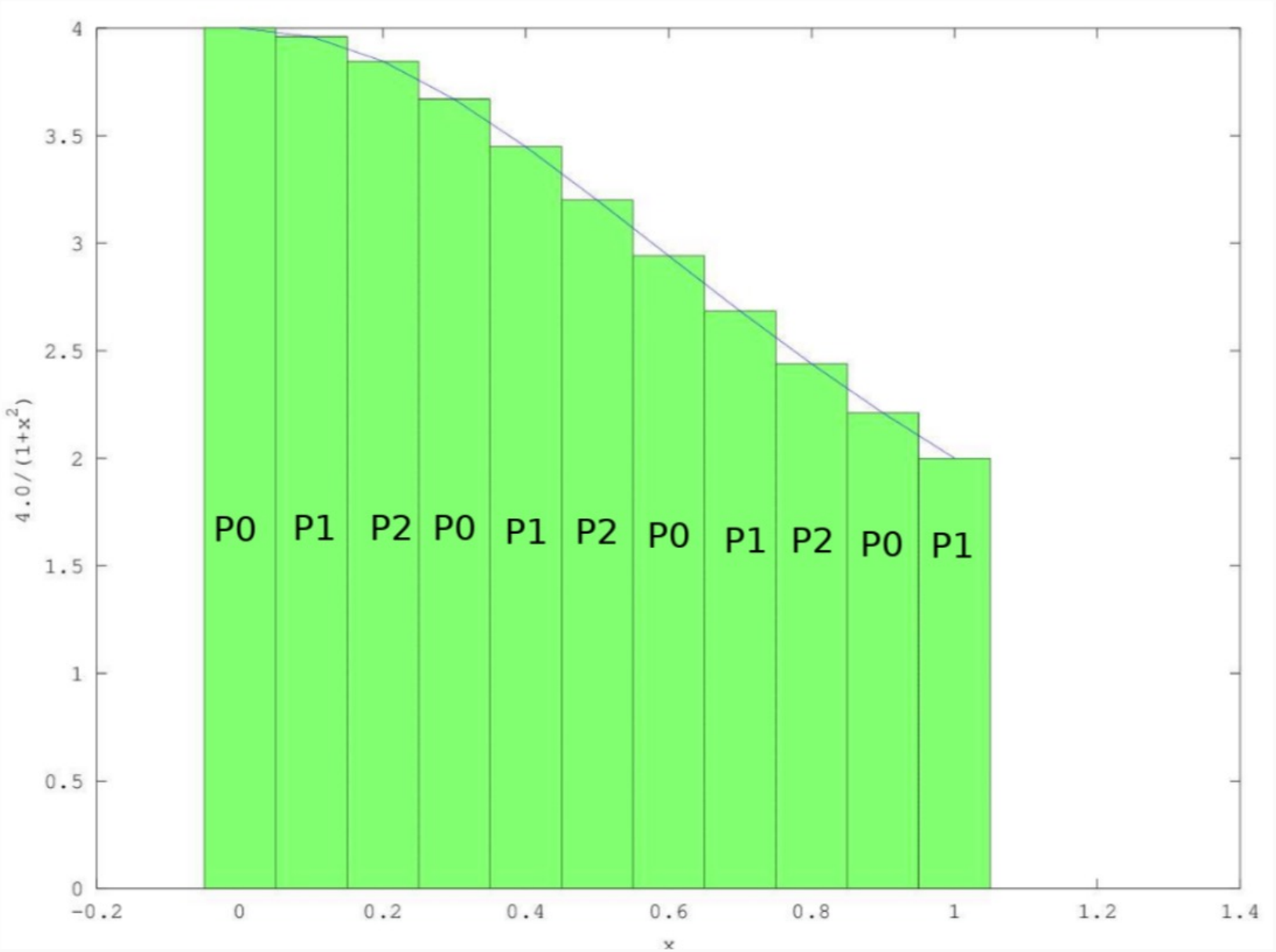


- We have written an MPI version of our python pi calculator to demonstrate.

Serial Program to Calculate π



Parallel Program to Calculate π



MPI: Message Passing Interface

When programs need to run on many processors but also communicate with one another.

Here the parallel version of calcPi needs to communicate the partial sums computed by each process so they can all be added up.
To communicate we will use the MPI library.

Make sure you are in the ~/workshops/intro_workshops directory.

Conda environment has the mpi libraries we need

```
[vanilla@easley intro_workshop]$ conda info --envs | grep intro_workshop
intro_workshop        * /projects/shared/conda/envs/intro_workshop

[vanilla@easley intro_workshop]$ conda activate intro_workshop
(intro_workshop) [vanilla@easley intro_workshop]$ conda list | grep mpi
mpi                       1.0.1                       openmpi    conda-forge
mpi4py                    4.1.1           py310h42af892_100    conda-forge
openmpi                   5.0.8              h2fe1745_108    conda-forge
```

```
import time
import sys
import numpy as np # Value of PI to compare to

##################### SETUP MPI - START #####################
from mpi4py import MPI        #Import the MPI library
comm = MPI.COMM_WORLD         #Communication framework
root = 0                      #Root process
rank = comm.Get_rank()        #Rank of this process
num_procs = comm.Get_size()   #Total number of processes
############################## END ##############################

#Distributed function to calculate pi
def Pi(num_steps):
    step = 1.0 / num_steps
    sum = 0
    for i in range(rank, num_steps, num_procs): # Divide sum among processes
        x = (i + 0.5) * step
        sum = sum + 4.0 / (1.0 + x * x)
    mypi = step * sum

    # Get that partial sums from all the processes, add them up, and give to the root process
    pi = comm.reduce(mypi, MPI.SUM, root)
    return pi
```

```
#Main function
# Check that the caller gave us the number of steps to use
if len(sys.argv) != 2:
    print("Usage: ", sys.argv[0], " <number of steps>")
    sys.exit(1)

num_steps = int(sys.argv[1],10);

#Broadcast number of steps to use to the other processes
comm.bcast(num_steps, root)

# Call function to calculate pi
start = time.time() #Start timing
pi = Pi(num_steps) # Call the function that calculates pi
end = time.time() # End timing

# If we are the root process then print our estimation of pi,
# the difference from numpy's value, and how long it took
print("Pi = %.20f, (Diff=%.20f) (calculated in %f secs with %d steps)" %(pi, pi-np.pi, end - start, num_steps))
```

```
[vanilla@easley intro_workshop]$ cat slurm/calc_pi_mpi.sh
```

```
#!/bin/bash
#SBATCH --partition debug
#SBATCH --nodes 2
#SBATCH --ntasks-per-node 4
#SBATCH --time 00:05:00
#SBATCH --job-name calc_pi_mpi
#SBATCH --mail-user your_username@unm.edu
#SBATCH --mail-type ALL

module load miniconda3
source activate intro_workshop

mpirun -np $SLURM_NTASKS python code/calcPiMPI.py 1000000000
```

```
[vanilla@easley intro_workshop]$ sbatch slurm/calc_pi_mpi.sh
```

```
srun --mpi=pmi2 python code/calcPiMPI.py 1000000000
```

srun understands MPI programs!

If you ever used mpirun or mpiexec you had to provide a lot of parameters to describe how many compute nodes you had and what their names are, etc.

But srun is part of SLURM so it already knows all that.

The only thing you have to specify is the communication library to use. In our case “pmi2”.

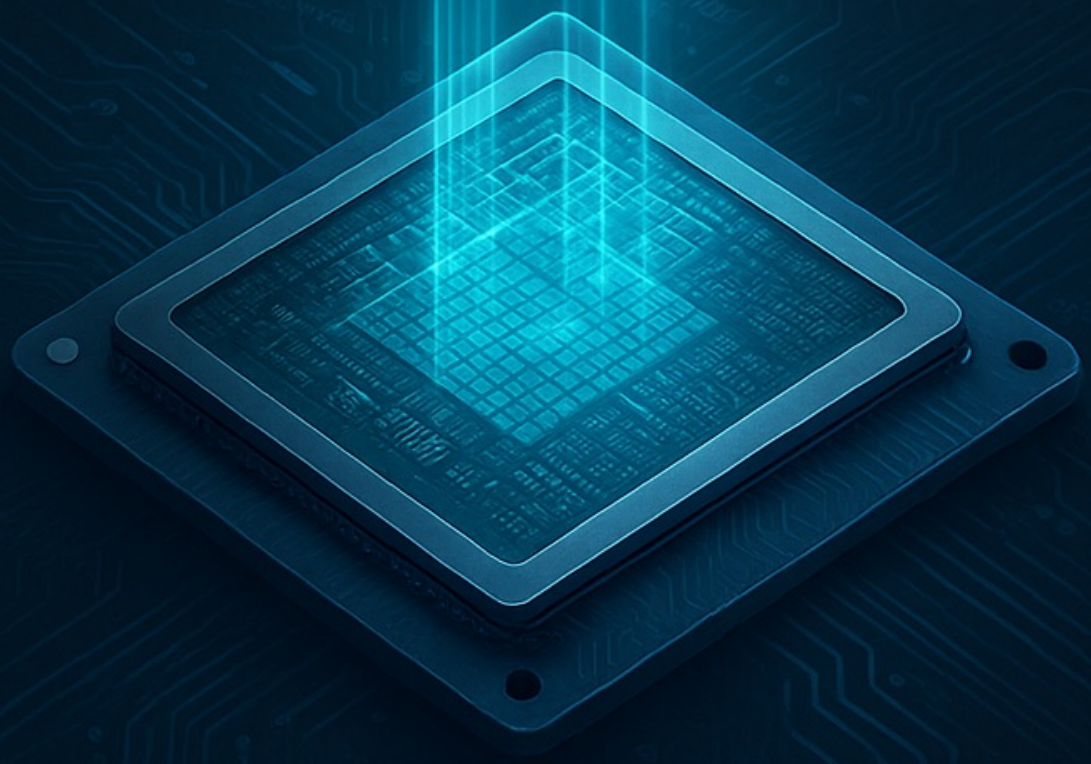
Make a note of the time it took to calculate.

Introduction to GPU Programming

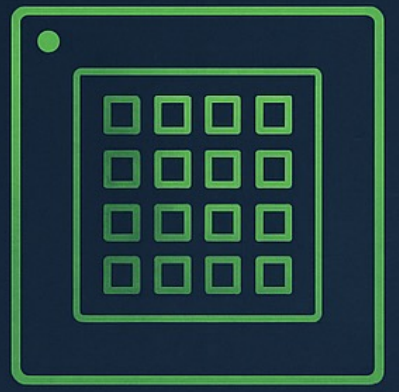
NVIDIA CUDA

Why GPUs?

- Massive parallelism = excellent for scientific computing (e.g. simulations, linear algebra)
- Accelerators for AI / deep learning use cases (training, inference)
- High memory bandwidth & many cores = good throughput for large workloads



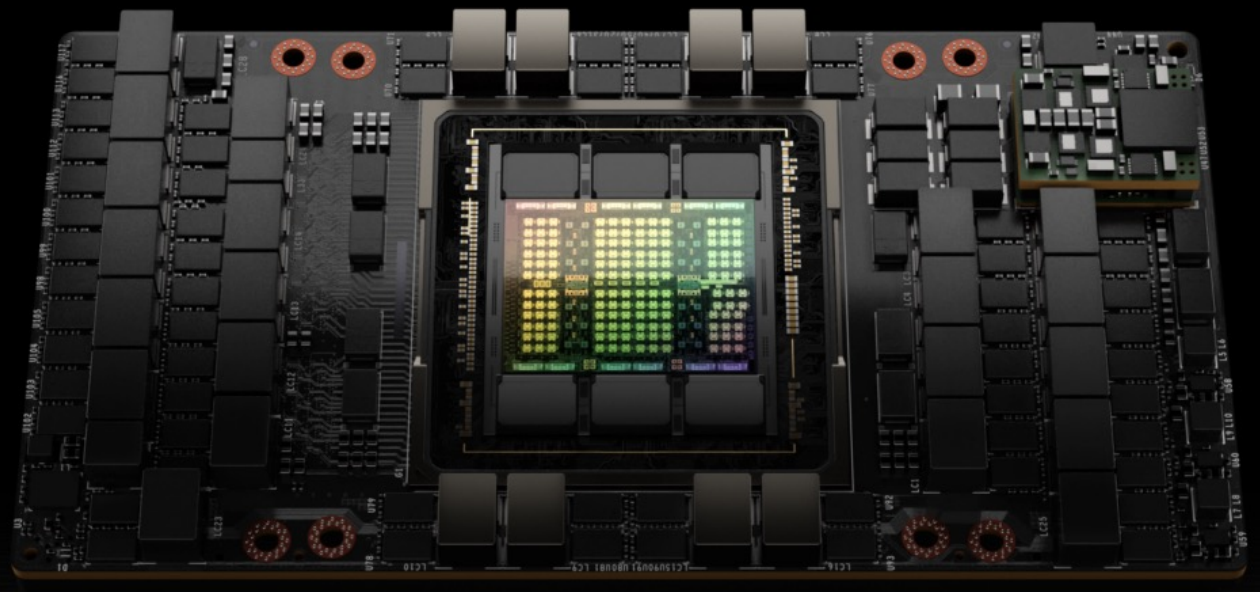
Strategy: Accelerating Pi Computation with GPU Paralelism



How it works:

- **Parallel Area Integration** Divides the unit interval into N steps and computes the integral of $4\ (1 + x^2)$ using numerical integration (Riemann sum)
- **GPU Offloading with Numba CUDA** Uses `@cuda.jit` to run the computation across 128×128 threads on the GPU
- **Device Memory Management** Transfers input arrays (`step_vec`, `sum_vec`) to the GPU, performs computation, and copies results back to the CPU
- **Double Precision Support** Implements calculations using `float64` for high accuracy, comparing results to NumPy's `np.pi`
- **Performance Insight** Execution time is measured to higlight speedup from parallel execution

Arrays of SMs (Streaming Multiprocessors)



nVidia H100 Graphics Processing Unit

Serial CPU version:

```
for i in range(num_steps):
    x = (i + 0.5) * step
    sum = sum + 4.0 / (1.0 + x * x)
    pi = step * sum
```

Unrolling the Loop onto CUDA Cores

```
for i in range(N):
    body of loop
```

CUDA core

CUDA core

CUDA core

CUDA core

CUDA core

CUDA core

CUDA core

Conda environment has the CUDA libraries we need

```
(intro_workshop) [vanilla@easley intro_workshop]$ conda list | grep cuda
cuda-bindings             12.9.3          py310hc395c1c_0    nvidia
cuda-cccl_linux-64        12.8.90                         0    nvidia
cuda-command-line-tools   12.8.1                          0    nvidia
cuda-compiler             12.8.1                          0    nvidia
cuda-crt-dev_linux-64     12.8.93                         0    nvidia
cuda-crt-tools            12.8.93                         0    nvidia
cuda-cudart               12.8.90                         0    nvidia
cuda-cudart-dev           12.8.90                         0    nvidia
cuda-cudart-dev_linux-64  12.8.90                         0    nvidia
cuda-cudart-static        12.8.90                         0    nvidia

...
```

```
(intro_workshop) [vanilla@easley intro_workshop]$ nano code/calcPiCudaFloat64.py
```

```
import time
import sys
import numpy as np
import numba.cuda as cuda
from numba import vectorize
import math

# Calculate pi by assessing the area under the curve using the GPU (float64
precision)
# Then we check the value against what is calculated in numpy
@cuda.jit
def Pi(step_vec, sum_vec, step_size):
    tx =  cuda.threadIdx.x # Unique thread ID within 1 block
    ty = cuda.blockIdx.x # Unique block ID
    block_size = cuda.blockDim.x # Number of threads per block
    grid_size = cuda.gridDim.x # Size of the grid

    # We define the grid size:
    startX = cuda.grid(1) # Starting point on the Grid
    gridX = cuda.gridDim.x * cuda.blockDim.x # stride in x
    stride = cuda.gridsize(1)
```

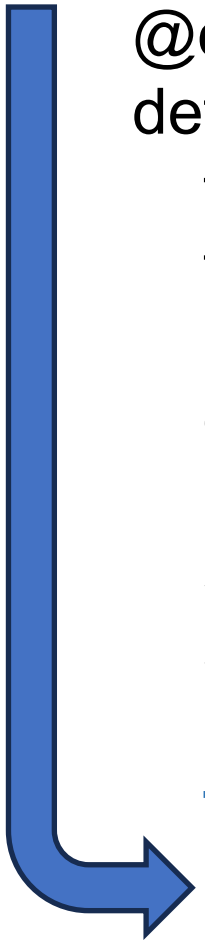
Numba
Developed by **Anaconda, Inc.**
A **Just-In-Time (JIT) compiler** for Python, using **LLVM** (Low-Level Virtual Machine)
Designed to accelerate **numerical Python code**
Integrates tightly with **NumPy** and **CUDA**

`@cuda.jit` is a **decorator** in Numba that tells Python:

Compile this function to run **on the GPU** using CUDA.

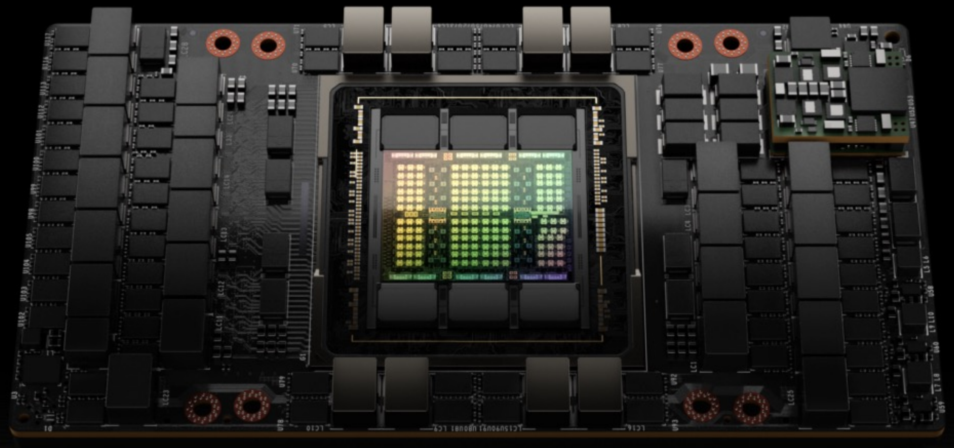
```
for i in range(num_steps):
        x = (i + 0.5) * step
        sum = sum + 4.0 / (1.0 + x * x)
        pi = step * sum
```

nano code/calcPiCudaFloat64.py



```
@cuda.jit
def Pi(step_vec, sum_vec, step_size):
    tx = cuda.threadIdx.x
    ty = cuda.blockIdx.x
    block_size = cuda.blockDim.x
    grid_size = cuda.gridDim.x

    startX = cuda.grid(1)
    stride = cuda.gridsize(1)

    for i in range(startX, step_vec.shape[0], stride):
        step_vec[i] = (step_vec[i] + 0.5) * step_size[0]
        sum_vec[i] += 4.0 / (1.0 + step_vec[i] * step_vec[i])
```

Blocks and Threads



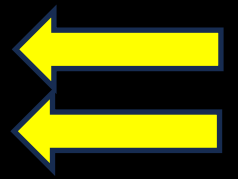
Block

Shared memory area corresponding to a streaming multiprocessor

Thread $\frac{4.0}{1,0+x^2}$	Thread $\frac{4.0}{1,0+x^2}$	Thread $\frac{4.0}{1,0+x^2}$	Thread $\frac{4.0}{1,0+x^2}$
Thread $\frac{4.0}{1,0+x^2}$	Thread $\frac{4.0}{1,0+x^2}$	Thread $\frac{4.0}{1,0+x^2}$	Thread $\frac{4.0}{1,0+x^2}$
Thread $\frac{4.0}{1,0+x^2}$	Thread $\frac{4.0}{1,0+x^2}$	Thread $\frac{4.0}{1,0+x^2}$	Thread $\frac{4.0}{1,0+x^2}$
Thread	Thread	Thread	Thread

```
step_vec = np.arange(int(sys.argv[1],10), dtype=np.float64)
sum_vec = np.zeros(int(sys.argv[1],10), dtype=np.float64)
pi = np.empty(1).astype(np.float64)
N = int(sys.argv[1],10)
step_size = np.ones(1, dtype=np.float64) / N

# We convert the variables to device
step_device = cuda.to_device(step_vec)
sum_device = cuda.to_device(sum_vec)
pi_device = cuda.to_device(pi)
step_size_device = cuda.to_device(step_size)

# Use 128 blocks, each containing 128 threads
blocks_per_grid = 128
thread_per_block = 128

# Call the function to calculate pi
start = time.time() #Start
Pi[blocks_per_grid, thread_per_block](step_device, sum_device, step_size_device)

# Copy data from device (GPU) to host (CPU) and calculate the final pi version
sum_vec = sum_device.copy_to_host()
pi = float(step_size[0] * sum_vec.sum())
end = time.time() # End
```

Read number of steps from command line (N)

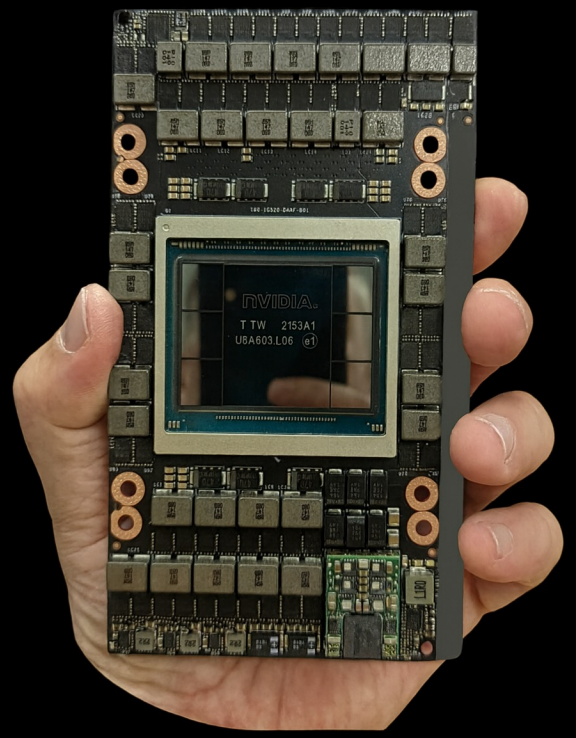


A block is a Streaming Processor

A thread is a SIMD Computation

```
[vanilla@easlet intro_workshop]$ nano slurm/calc_pi_gpu.sh
```

```
#!/bin/bash
#SBATCH --partition=l40s
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --gpus=1
#SBATCH --time=00:05:00
#SBATCH --job-name=calc_pi_gpu64
#SBATCH --mail-user=your_username@unm.edu
#SBATCH --mail-type=ALL
#SBATCH --output=calc_pi_gpu64_%j.out

# Load Conda environment
module load miniconda3
source activate intro_workshop

# Run the single-precision (float64) GPU code
srun python code/calcPiCudaFloat64.py 1000000000
```



NVIDIA L40S

```
[vanilla@easley intro_workshop]$ sbatch slurm/calc_pi_gpu.sh
Submitted batch job 72683
[vanilla@easley intro_workshop]$ squeue --me
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
             72683      l40s calc_pi_  mfricke  R       0:02      1 easley057
[vanilla@easley intro_workshop]$ cat calc_pi_gpu64_72094.out
This Miniconda3 installation uses the conda-forge channel. Conda-forge packages
are generally open-source and free for research and educational use. Be sure to
review the license agreements of software packages you install and additional
channels you enable.
You have been allocated one or more GPUs.
Job 72094 running on easley056
Pi = 3.141592653589801553 69, (Diff=0.00000000000000843769) (calculated in 2.709788 secs
with 1000000000 steps)
```

Compare the serial, MPI, and GPU times