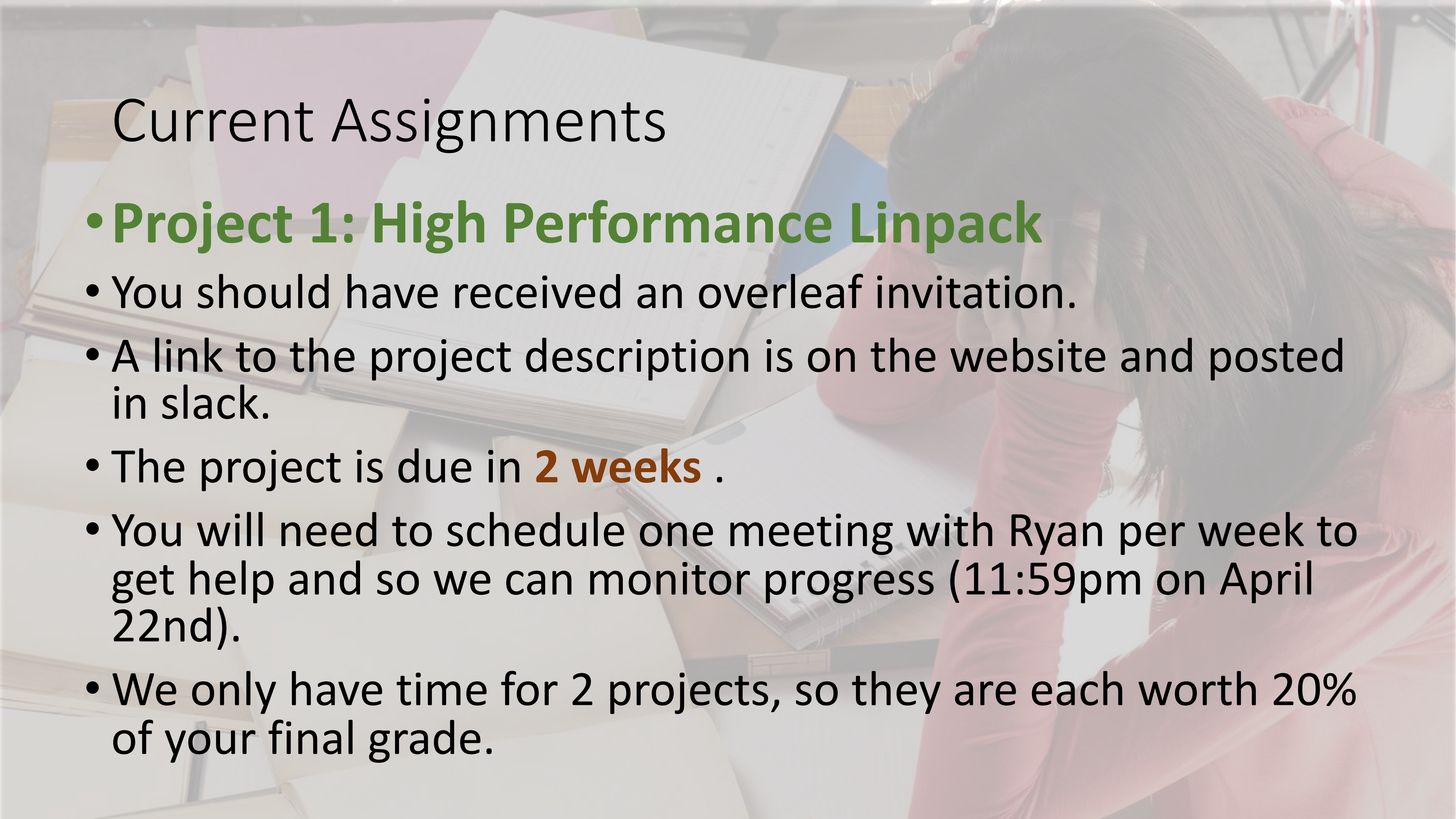
Lecture 18: Distributed Computing

Job Arrays and MPI



Current Assignments

- **Project 1: High Performance Linpack**
- You should have received an overleaf invitation.
- A link to the project description is on the website and posted in slack.
- The project is due in **2 weeks** .
- You will need to schedule one meeting with Ryan per week to get help and so we can monitor progress (11:59pm on April 22nd).
- We only have time for 2 projects, so they are each worth 20% of your final grade.

- Embarrassingly Parallel
  - SLURM Arrays
  - GNU Parallel
- Coupled Parallelism
  with MPI

We will have one 15 minute break. Opportunity to see the machine room.

```
[vanilla@hopper ~]$ 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@hopper ~]$ 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@hopper ~]$ 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@hopper intro_workshop]$ pwd
/users/vanilla/workshops/intro_workshop
[vanilla@hopper intro_workshop]$ cat slurm/workshop_example.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

hostname
```

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

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

We submit our slurm shell script with the sbatch command.

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

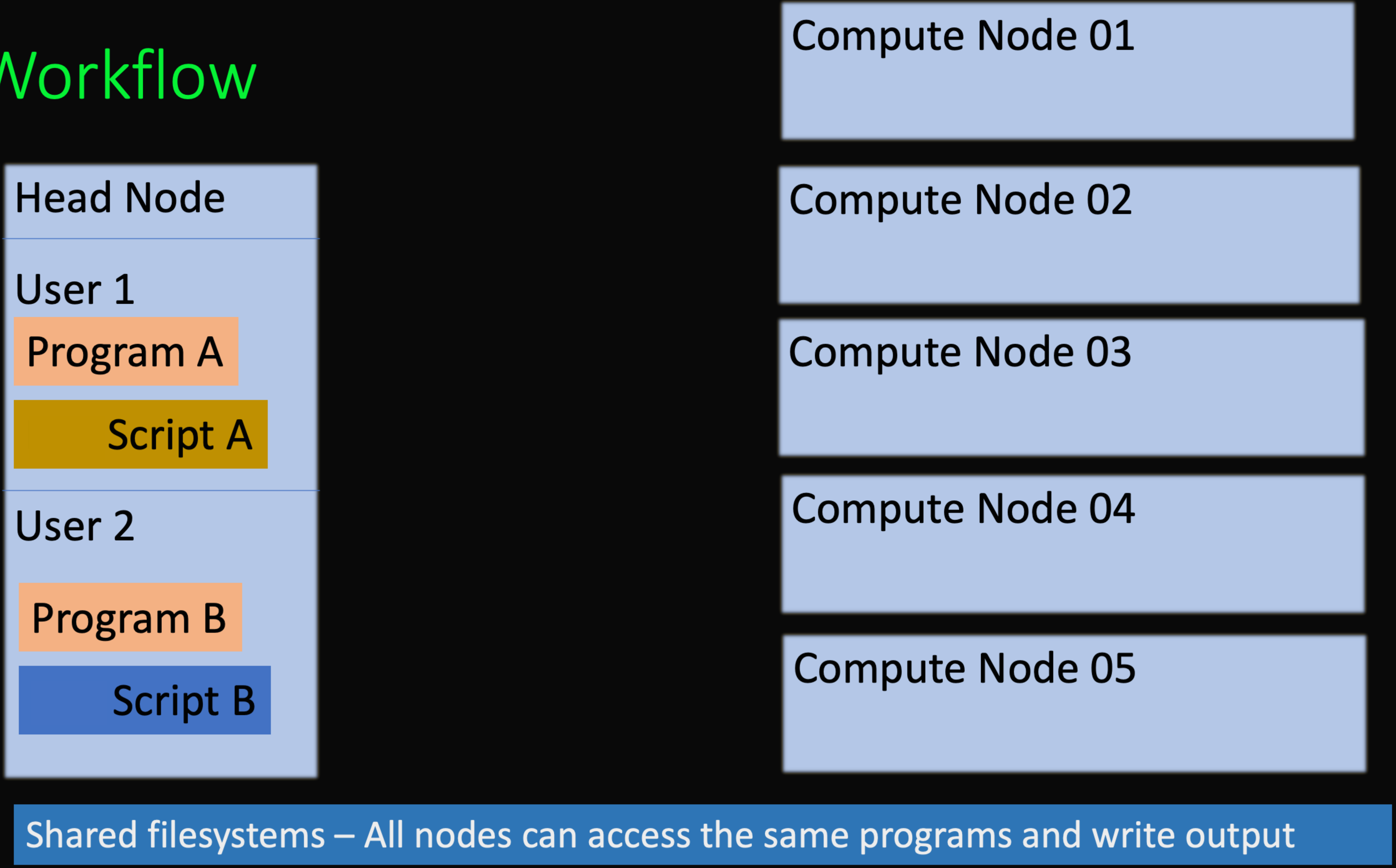
We submit our slurm shell script with the sbatch command.

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

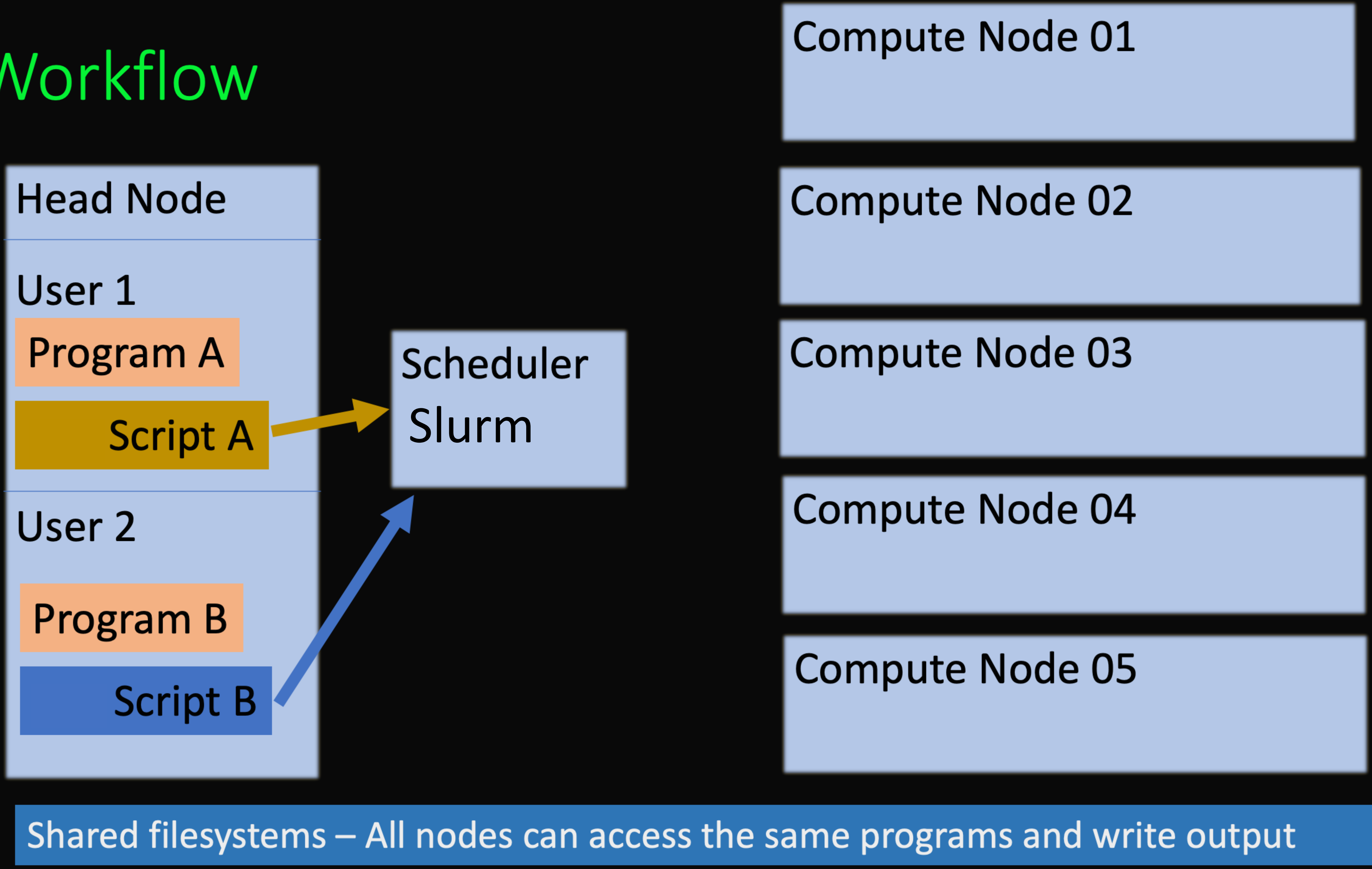
This indicates that the script was successfully sent to the scheduler.

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

Workflow

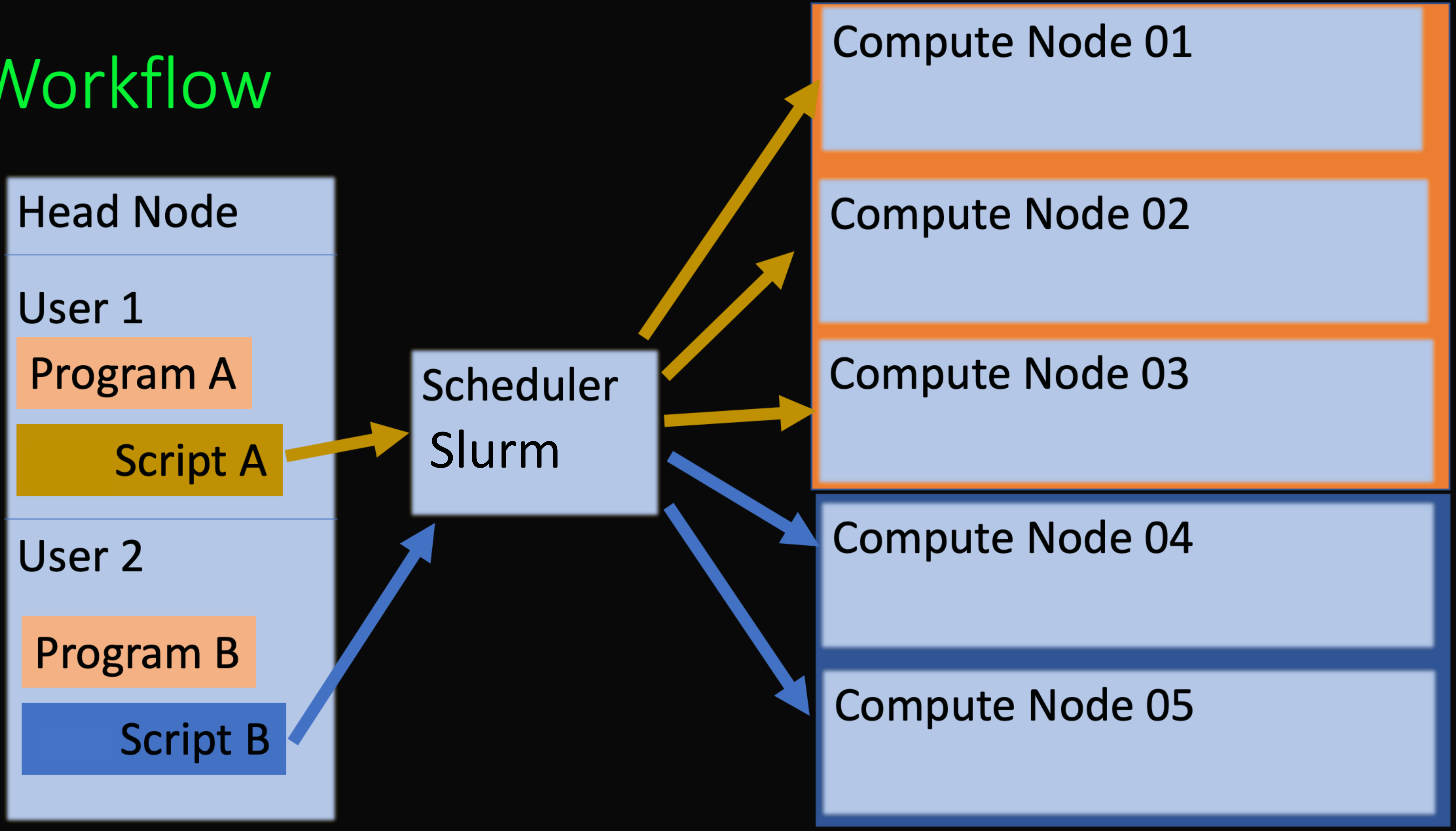

Head Node
User 1
Program A
Script A
User 2
Program B
Script B
Compute Node 01
Compute Node 02
Compute Node 03
Compute Node 04
Compute Node 05
Shared filesystems – All nodes can access the same programs and write output

Workflow



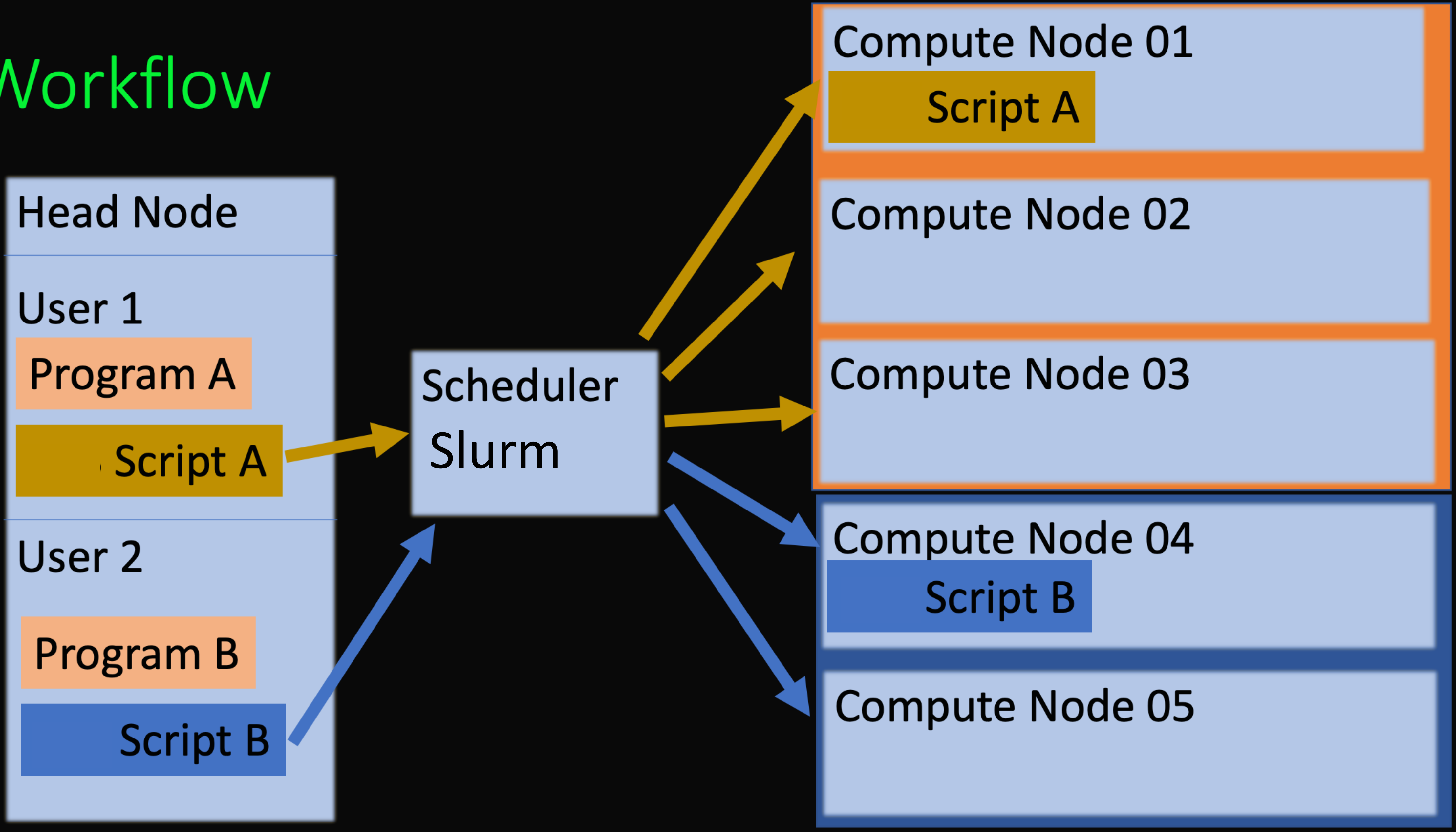
Head Node
User 1
Program A
Script A
User 2
Program B
Script B
Scheduler
Slurm
Compute Node 01
Compute Node 02
Compute Node 03
Compute Node 04
Compute Node 05
Shared filesystems – All nodes can access the same programs and write output

Workflow



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

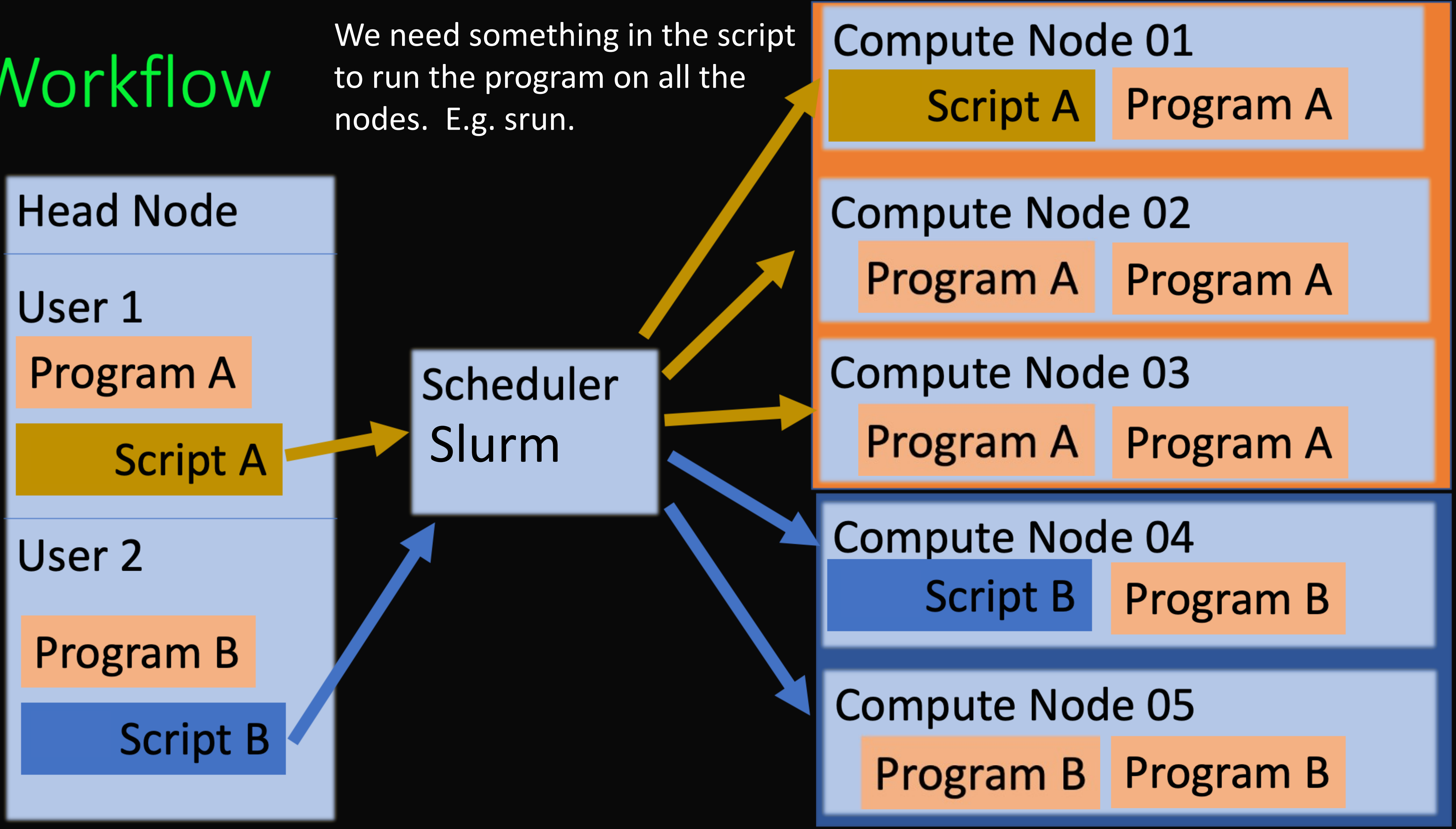
Workflow



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@hopper 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@hopper intro_workshop]$ ls
code  data  pbs  slurm  slurm-5252.out
```

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

```
[vanilla@hopper intro_workshop]$ cat slurm-5252.out
hopper011
```

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

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

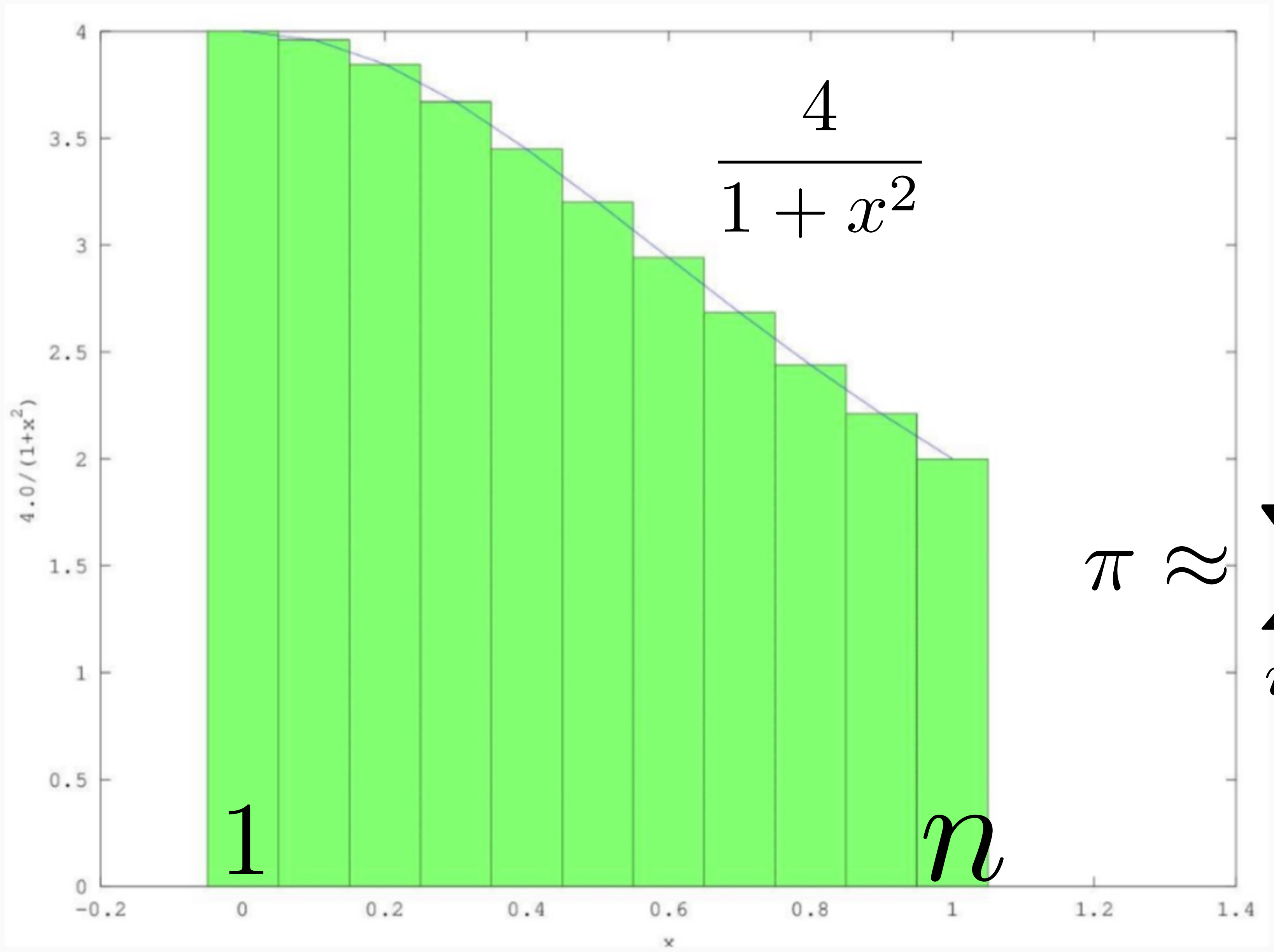
```
[vanilla@hopper intro_workshop]$ cat slurm-5252.out
hopper011
```

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

```
[vanilla@hopper 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@hopper intro_workshop]$
```

Take a look at the output file.

Serial Program to Calculate π



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

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

```python
# 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)
```

```
[vanilla@hopper intro_workshop]$ module load miniconda3
[vanilla@hopper 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

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

```
[vanilla@hopper intro_workshop]$source activate numpy
[vanilla@hopper intro_workshop]$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 = 3.14242598500109870940, (Diff=0.00083333141130559341)
(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@hopper intro_workshop]$ sbatch slurm/calc_pi_serial.sh
 sbatch: Using account 2016199 from ~/.default_slurm_account
Submitted batch job 5263
vanilla@hopper:~/workshops/intro_workshop$ squeue --me
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
5263      debug calc_pi_  vanilla  R       0:44      1 hopper011
```

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.

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@hopper 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 numpy

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@hopper 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 numpy

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@hopper intro_workshop]$ sbatch slurm/calc_pi_array.sh
 sbatch: Using account 2016199 from ~/.default_slurm_account
Submitted batch job 5263
vanilla@hopper:~/workshops/intro_workshop$ watch squeue --me
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
5263      debug calc_pi_  vanilla  R       0:44      1 hopper011
```

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?)

JOB arrays are OK or very simple inputs like programs that take a single file as input. But even passing in a value takes some annoying variable manipulation.

GNU Parallel is much more sophisticated it can take inputs in all sorts of ways. We will look at just 3 ways.

To access GNU parallel enter **module load parallel**

Let’s experiment with parallel interactively…

***NOTE: we don’t use srun to run parallel.**

GNU Parallel

Has been around for a very long time and has lots and lots of great features.

But basically it creates a job for every input it receives. The inputs can be specified in the command, read from a file, or be the output of another program.

It also remembers which jobs have finished and which still need to be run. So when you run out of time and resubmit it will automatically pick up where it left off.

```
[vanilla@hopper intro_workshop]$ salloc --partition debug --ntasks 2
salloc: Using account 2016199 from ~/.default_slurm_account
salloc: Granted job allocation 5275
salloc: Waiting for resource configuration
salloc: Nodes hopper011 are ready for job

$ module load parallel
$ parallel python code/calcPiSerial.py ::: 10 20 30 40


Pi = 3.14180098689309428295, (Diff=0.00020833330330116695)
(calculated in 0.000006 secs with 20 steps)
Pi = 3.14164473692265744376, (Diff=0.00005208333286432776)
(calculated in 0.000008 secs with 40 steps)
Pi = 3.14242598500109870940, (Diff=0.00083333141130559341)
(calculated in 0.000006 secs with 10 steps)
Pi = 3.14168524617974842528, (Diff=0.00009259258995530928)
(calculated in 0.000012 secs with 30 steps)
```

```
[vanilla@hopper intro_workshop]$ seq 10 10 100
10
20
30
40
50
60
70
80
90
100
```

GNU Parallel can read the output of other programs and use them as inputs to your program.
Here a copy of calc pi is run for each row in the output of seq

```
[vanilla@hopper intro_workshop]$ seq 10 10 100 | parallel python code/calcPiSerial.py
Pi = 3.14180098689309428295, (Diff=0.00020833330330116695) (calculated in 0.000007
secs with 20 steps)
Pi = 3.14242598500109870940, (Diff=0.00083333141130559341) (calculated in 0.000006
secs with 10 steps)
Pi = 3.14168524617974842528, (Diff=0.00009259258995530928) (calculated in 0.000007
secs with 30 steps)
etc
```

```
[vanilla@hopper intro_workshop]$ find -name *.sh
./slurm/calc_pi_array.sh
./slurm/calc_pi_mpi.sh
./slurm/calc_pi_parallel.sh
./slurm/calc_pi_serial.sh
./slurm/gaussian.sh
./slurm/hostname_mpi.sh
etc

$ find -name *.sh | parallel wc -l
7 ./code/vecadd/vecaddmpi_cpu.sh
19 ./slurm/calc_pi_array.sh
15 ./slurm/calc_pi_mpi.sh
20 ./slurm/calc_pi_parallel.sh
14 ./slurm/calc_pi_serial.sh
16 ./slurm/gaussian.sh
15 ./slurm/hostname_mpi.sh
etc
```

A common application is to use find to produce a list of paths with some extension.

Then parallel runs some program on each path.

In this case wc -l counts the lines in a file. In some real CARC examples the input files are phylogenetic trees, graphs, neuroimages, or CT scans.

```
(numpy)$ seq 10 10 100 | parallel srun python code/calcPiSerial.py
```

The parallel commands we executed so far ran on the head node (don’t worry it wasn’t much computation).

To get parallel to use slurm and run on a compute node we have to add srun.

```
(numpy)$ seq 10 10 100 | parallel srun python code/calcPiSerial.py

You have not been allocated GPUs. To request GPUs, use the -G option in your submission
script.
Pi = 3.14180098689309428295, (Diff=0.00020833330330116695) (calculated in 0.000015 secs
with 20 steps)
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 = 3.14160966039249744952, (Diff=0.00001700680270433352) (calculated in 0.000029 secs
with 70 steps)
…

Pi = 3.14160098692312539370, (Diff=0.00000833333333227770) (calculated in 0.000036 secs
with 100 steps)
```

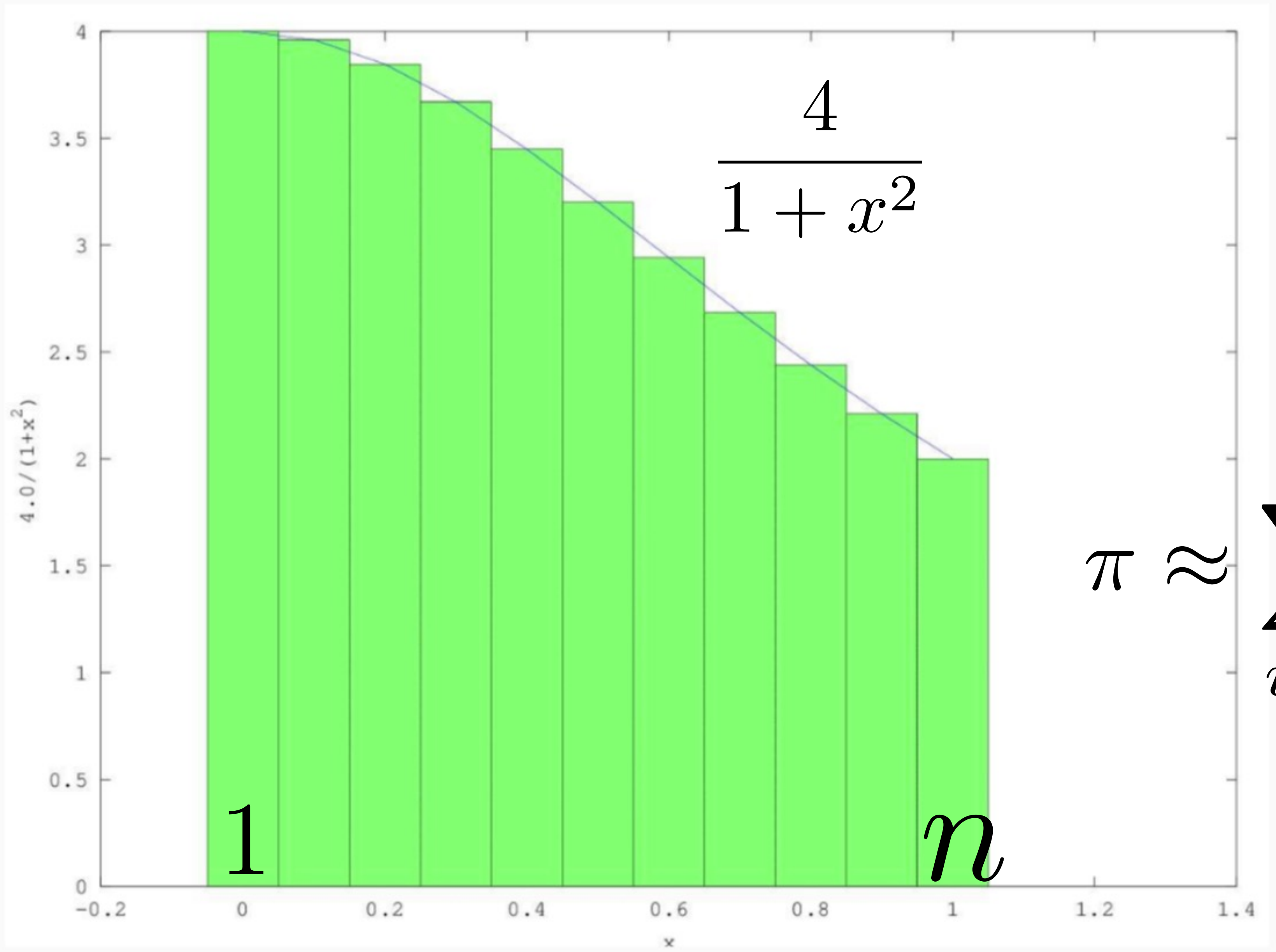
```
[vanilla@hopper intro_workshop]$ exit
exit
salloc: Relinquishing job allocation 5275
```

Don’t forget to exit your salloc allocation.

Parallelism – Coupled Parallelism

- 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.

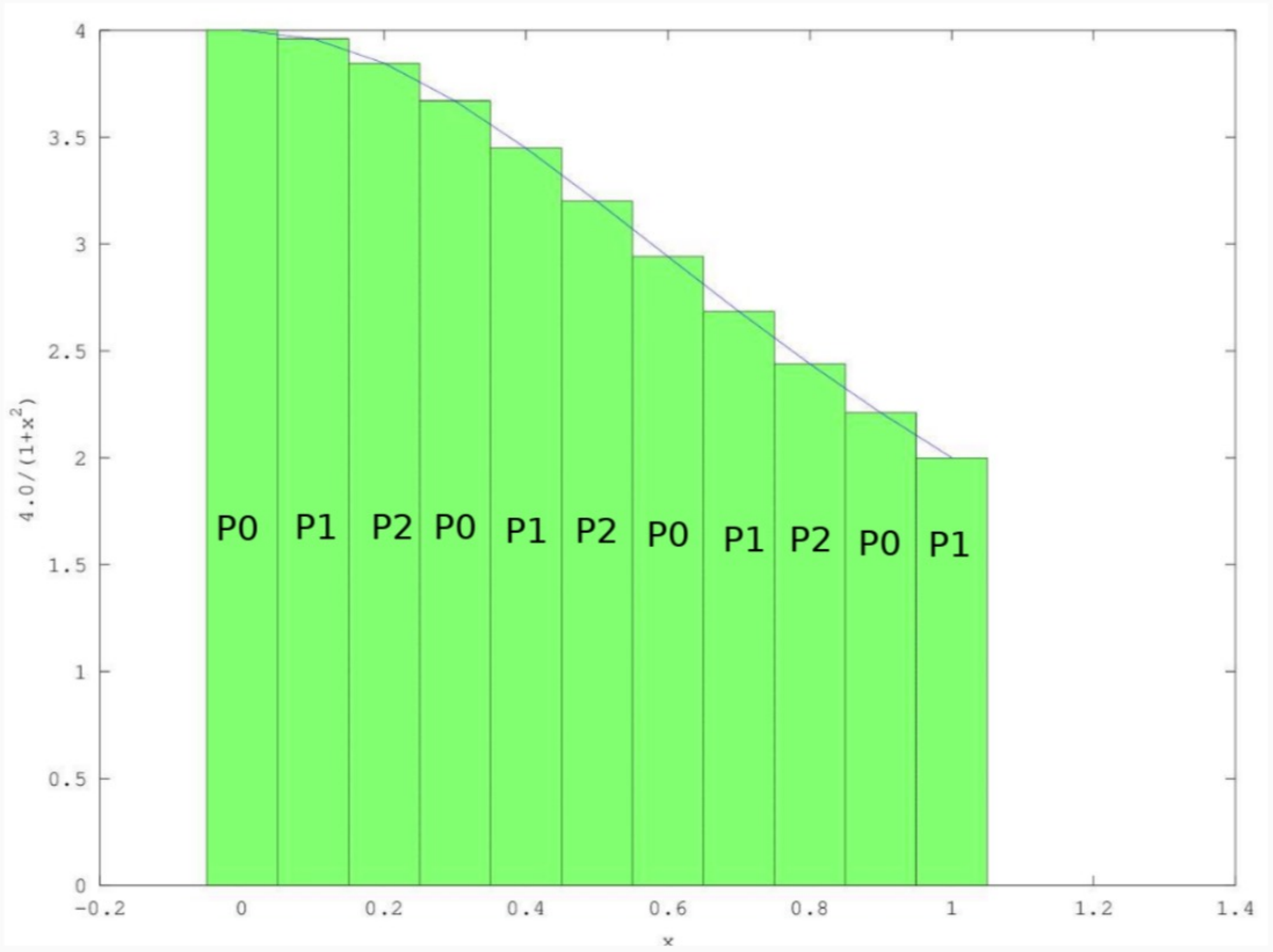
Serial Program to Calculate π



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

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

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:

```
module load minconda3
conda create –n mpi_numpy mpi mpi4py numpy
```

```
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))
```

```
###################### 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
<snip>
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))
```

```
#!/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 mpi_numpy

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

```
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”.

In Class Experiment

Run calc_pi_mpi.sh.

Vary the number of tasks it uses.

Use squeue to monitor the state of your job,

Look at your output files.

What is the relationship between the number of tasks and how fast it calculates pi?

Example Code with MPI and CUDA

- Let’s take a look at ~/workshops/intro_workshop/code/vecadd
- The Makefile allows you to compile C code for CPU using MPI or a MPI and CUDA code.
- First though we have to provide a MPI library

```
module load openmpi
```

Example Code with MPI and CUDA

- This code just takes two vectors and corresponding elements together to create a new array.
- This code is NOT written to be efficient – I tried to write it to make the relationship between MPI processes clear.

```
$ cd ~/workshops/intro_workshop/code/vecadd
$ module load openmpi
$ cd ~/workshops/intro_workshop/code/vecadd
$ module load openmpi
$ cat Makefile

gpu:
nvcc -arch sm_35 -c vecadd_gpu.cu -o vecadd_gpu.o
mpic++ -c vecadd_mpi_gpu.c -o vecadd_mpi_gpu.o
mpic++ vecadd_mpi_gpu.o vecadd_gpu.o -lcudart -o vecadd_mpi_gpu

cpu:
mpic++ vecadd_mpi_cpu.c -o vecadd_mpi_cpu

clean:
rm vecadd_mpi_gpu vecadd_mpi_cpu vecadd_mpi_gpu.o
vecadd_mpi_cpu.o vecadd_gpu.o
```

```
$ cd ~/workshops/intro_workshop/code/vecadd
$ module load openmpi
$ cat Makefile
```

Requires the Xena GPU cluster

```
gpu:
nvcc -arch sm_35 -c vecadd_gpu.cu -o vecadd_gpu.o
mpic++ -c vecadd_mpi_gpu.c -o vecadd_mpi_gpu.o
mpic++ vecadd_mpi_gpu.o vecadd_gpu.o -lcudart -o vecadd_mpi_gpu

cpu:
mpic++ vecadd_mpi_cpu.c -o vecadd_mpi_cpu

clean:
rm vecadd_mpi_gpu vecadd_mpi_cpu vecadd_mpi_gpu.o
vecadd_mpi_cpu.o vecadd_gpu.o
```

```
$ cd ~/workshops/intro_workshop/code/vecadd
$ module load openmpi
$ make cpu
mpic++ vecadd_mpi_cpu.c -o vecadd_mpi_cpu
```

```
$ cd ~/workshops/intro_workshop/code/vecadd
$ module load openmpi
$ make cpu
mpic++ vecadd_mpi_cpu.c -o vecadd_mpi_cpu
$ ls
Makefile  vecadd_gpu.cu  vecadd_mpi_cpu  vecadd
_mpi_cpu.c  vecaddmpi_cpu.sh  vecadd_mpi_gpu.c
```

Useful Slurm Commands

squeue --me --long	shows information about jobs you submitted
squeue --me --start	shows when slurm expects your job to start
scancel jobid	cancels a job
scancel --u $USER	cancels all your jobs
sacct	shows your job history
seff jobid	shows how efficiently the hardware was used