Distributed Computing

(Parameter Sweeps, MPI, and CUDA)

Matthew Fricke

Version 0.2

```
[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 = 3.142425985001098709 40, (Diff=0.00083333141130559341) (calculated in 0.000005 secs
with 10 steps)

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

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.

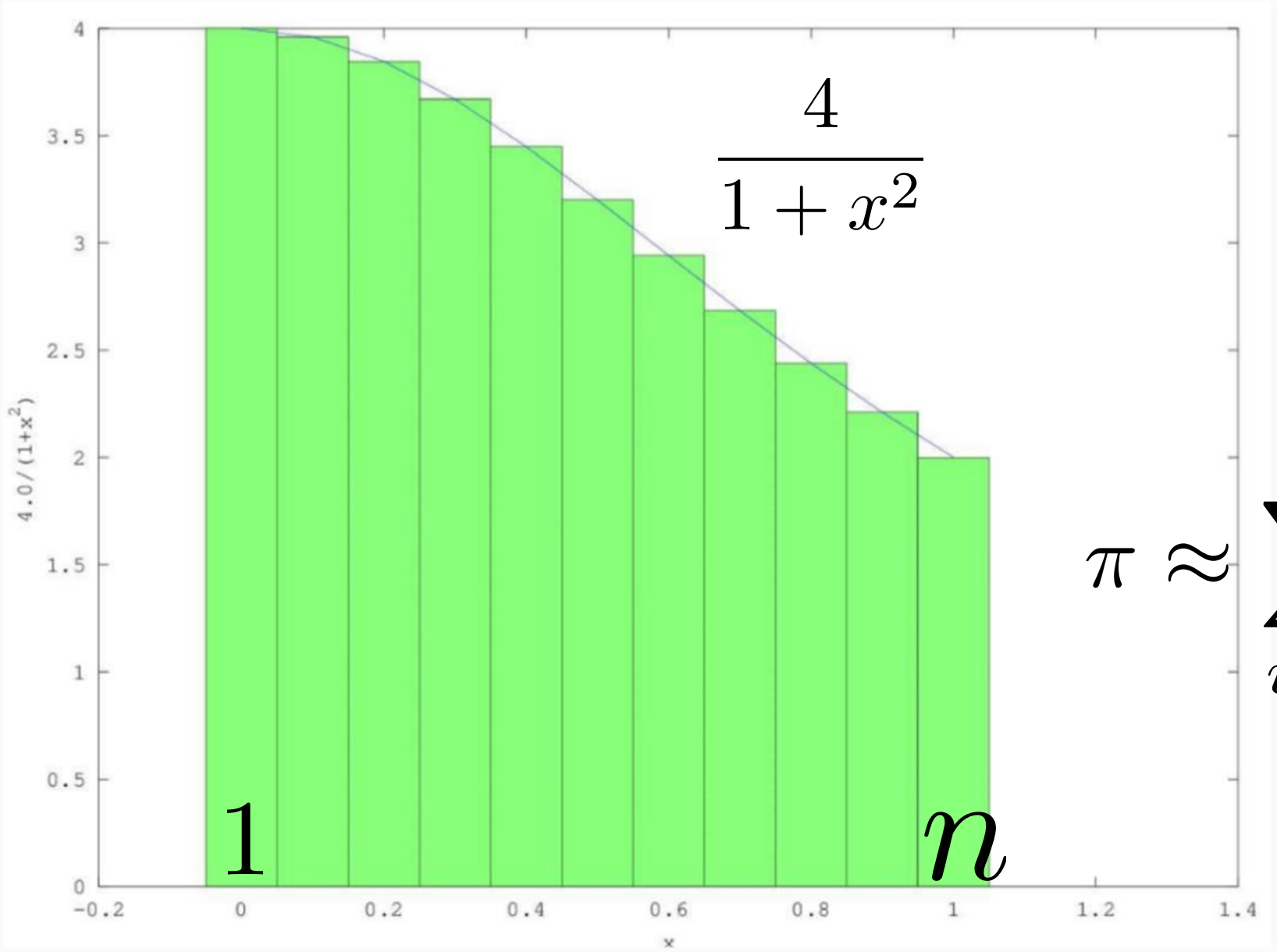
```
[vanilla@hopper intro_workshop]$ sbatch slurm/calc_pi_serial.sh
 sbatch: Using account 2016199 from ~/.default_slurm_account
Submitted batch job 5263
vanilla@hopper:~/workshops/intro_workshop$ squeue -–me
JOBID PARTITION     NAME     USER ST      TIME  NODES NODELIST(REASON)
5263     debug calc_pi_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

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)
```

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?

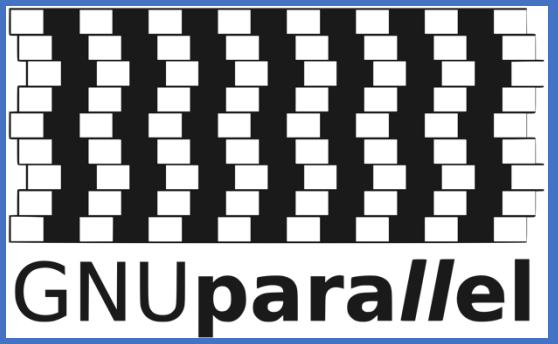
```
[vanilla@hopper intro_workshop]$ sbatch slurm/calc_pi_array.sh
 sbatch: Using account 2016199 from ~/.default_slurm_account
Submitted batch job 5263
vanilla@hopper:~/workshops/intro_workshop$ watch squeue --me
JOBID PARTITION     NAME     USER ST      TIME  NODES NODELIST(REASON)
5263     debug calc_pi_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

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 = 3.141800986893094282 95, (Diff=0.000208333330330116695) (calculated in 0.000006 secs
with 20 steps)
Pi = 3.141644736922657443 76, (Diff=0.000052083333286432776) (calculated in 0.000008 secs
with 40 steps)
Pi = 3.142425985001098709 40, (Diff=0.000833331411305593 41) (calculated in 0.000006 secs
with 10 steps)
Pi = 3.141685246179748425 28, (Diff=0.000092592589955309 28) (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]$ module load parallel
[vanilla@hopper intro_workshop]$ source activate numpy
[vanilla@hopper intro_workshop]$ seq 10 10 100 | parallel python code/calcPiSerial.py
Pi = 3.141800986893094295, (Diff=0.00020833330330116695) (calculated in 0.000007 secs with 20 steps)
Pi = 3.142425985001098709 40, (Diff=0.00083333141130559341) (calculated in 0.000006 secs with 10 steps)
Pi = 3.141685246179748425 28, (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 = 3.141800986893094282 95, (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.

```
(numpy)[vanilla@hopper intro_workshop]$ cat slurm/calc_pi_parallel.sh
#!/bin/bash
#SBATCH --partition debug
#SBATCH --nodes 2
#SBATCH --ntasks-per-node 4
#SBATCH --time 00:05:00
#SBATCH --job-name calc_pi_parallel
#SBATCH --mail-user your_username@unm.edu
#SBATCH --mail-type ALL

module load parallel
module load miniconda3
source activate numpy

parallel --sshloginfile $CARC_NODEFILE \
--env PATH \
--workdir $SLURM_SUBMIT_DIR \
"hostname; python code/calcPiSerial.py {}" :::: data/step_sizes.txt
```

Take a look at slurm/calc_pi_parallel.sh.

In this example parallel reads the parameters from a file and runs a copy of calcPiSerial.py on each row.

```
(numpy)$sbatch slurm/calc_pi_parallel.sh
sbatch: Using account 2016199 from ~/.default_slurm_account
Submitted batch job 5276
```

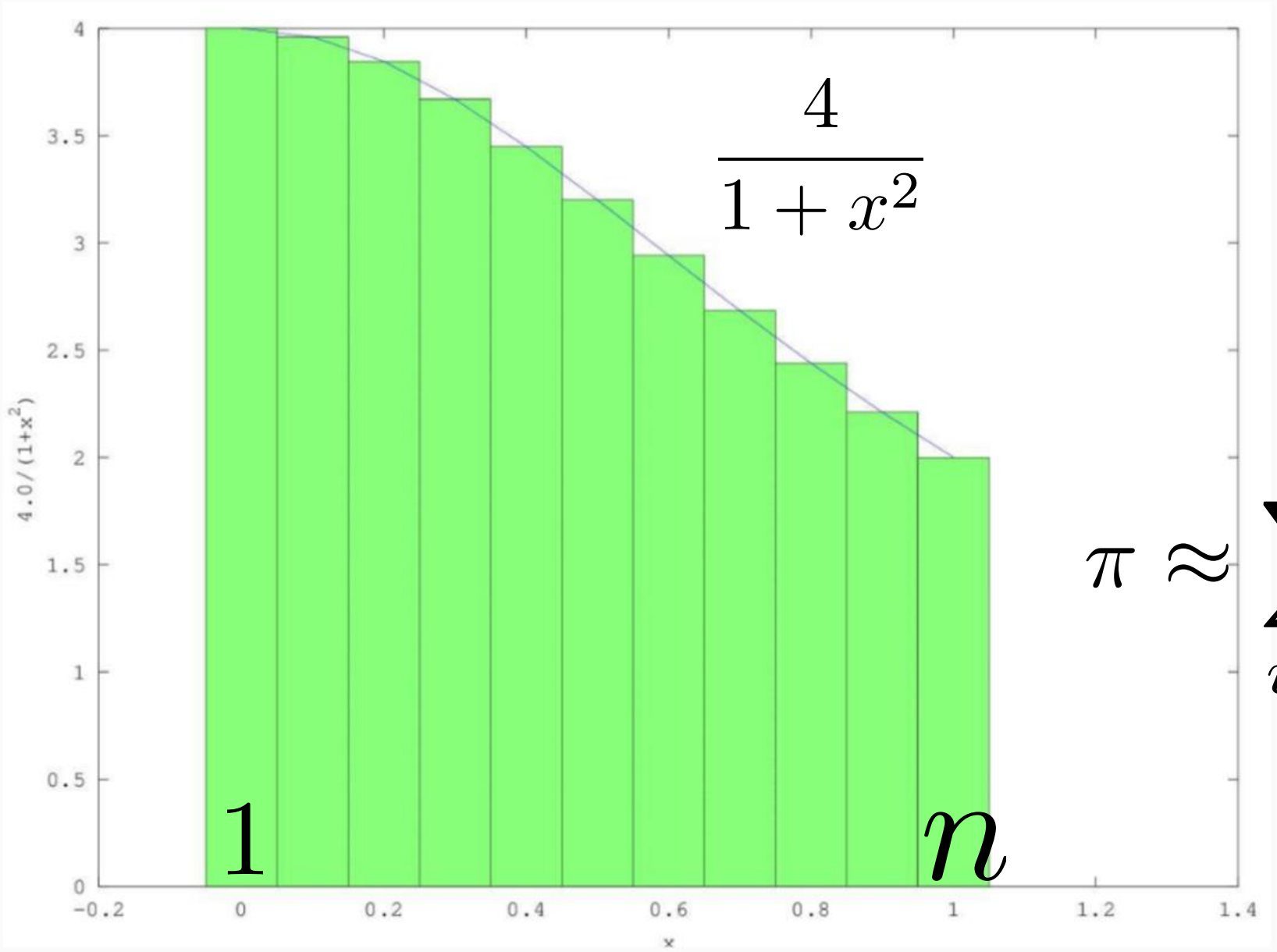
Submit the job, check it's progress with squeue --me, and take a look at the output.

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



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

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

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

Program to estimate π $-$ $Main\ program$

```
[vanilla@hopper code]$ cat calc_pi_serial.f90
program cal_pi_serial
  implicit none
  integer steps
  character(len=100) :: steps_arg
  real :: start, finish
  double precision :: p,pi_ref = 3.14159265358979323846264338

  call getarg(1, steps_arg) ! Read argument
  read(steps_arg,*) steps ! Convert string to integer

  call cpu_time(start) ! Time how long it takes to est pi
  p = pi(steps)
  call cpu_time(finish)

  print '("Pi=", g0, " (Diff=", g0,")")', p, abs(p-pi_ref)
  print '("(calculated in ", g0,"s with ", g0, " steps)")',        finish-start, steps

  contains
```

Program to estimate π $-$ $Function\ pi()$

```
! The area under the curve 4/(1+x^2) is pi.
! Estimate with the Riemann sum.

 function pi(num_steps) result(area)
 double precision x, area, step, sum
 integer num_steps
 sum = 0
 step_size = 1.0/num_steps ! Determine the rectangle size

! Loop over the rectangles under the curve
 do i = 0, num_steps
   x = (i + 0.5)*step_size ! Calc the ith rectangle area
   sum = sum + 4.0/(1.0+x*x) ! Sum them up
 end do

 area = step_size*sum ! Normalise to be between 0 and 1

end function


end program
```

Program to estimate π $-$ $Function\ pi()$

```
! The area under the curve 4/(1+x^2) is pi.
! Estimate with the Riemann sum.

 function pi(num_steps) result(area)
 double precision x, area, step, sum
 integer num_steps
 sum = 0
 step_size = 1.0/num_steps ! Determine the rectangle size

! Loop over the rectangles under the curve
 do i = 0, num_steps
   x = (i + 0.5)*step_size ! Calc the ith rectangle area
   sum = sum + 4.0/(1.0+x*x) ! Sum them up
 end do

 area = step_size*sum ! Normalise to be between 0 and 1

end function


end program
```

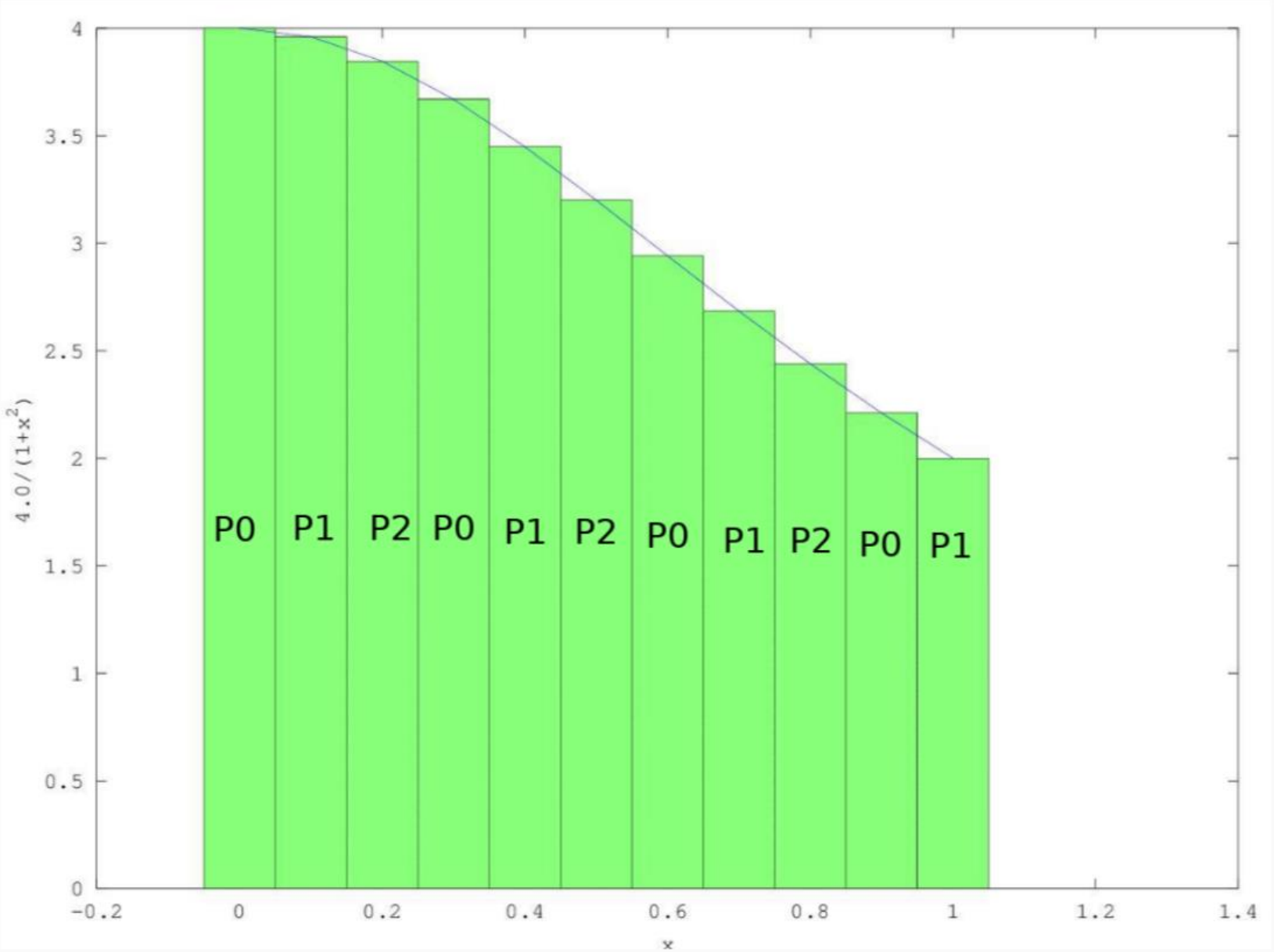
$$\sum_{i=1}^{N} \frac{4}{1+\left(\frac{i+\frac{1}{2}}{N}\right)^2} \approx \pi$$

```
[vanilla@hopper code]$ module load intel/20.0.4
[vanilla@hopper code]$ ifort calc_pi_serial.f90 -o calc_pi_serial
[vanilla@hopper code]$ srun --partition debug ./calc_pi_serial 1000
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=3.143591832167984 (Diff=.1999091155410415E-02)
(calculated in .2999790E-05s with 1000 steps)
```

Load intel compilers then compile our FORTRAN program.

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.

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 miniconda3
conda create –n mpi_numpy mpi mpi4py numpy
```

MPI Program to estimate π

```
program cal_pi_mpi
 use mpi ! Import Message Passing Interface
 implicit none ! Don't allow FORTRAN to infer types
 integer steps, ierr, num_procs, root, rank
 character(len=100) steps_arg
 real :: start, finish
 double precision :: p,pi_ref = 3.14159265358979323846264338327950288419 7


 ! Setup Message Passing Interface
 call MPI_Init( ierr )
 call MPI_Comm_rank( MPI_COMM_WORLD, rank, ierr )
 call MPI_Comm_size( MPI_COMM_WORLD, num_procs, ierr )
 root = 0 ! Call the 0th process "root"

 call getarg(1, steps_arg) ! Read argument
 read(steps_arg,*) steps ! Convert string to integer

 call cpu_time(start) ! Time how long it takes to est pi
 ! Call the pi function and print the result
 p = pi(steps, num_procs, rank)
 call cpu_time(finish)

 if (rank==root) then
   print '("Pi=", g0, " (Diff=", g0,")")', p, abs(p-pi_ref)
   print '("(calculated in ", g0,"s with ", g0, " steps and ", g0, " processes)")', finish-start, steps, num_procs
 endif
```

```
contains
 ! The area under the curve 4/(1+x^2) is pi
 ! Estimate with the Riemann sum
 function pi(num_steps, num_procs, rank) result(area)
  double precision x, area, my_area, step, sum, step_size
  integer num_steps, num_procs, rank, i
 sum = 0
 step_size = 1.0/num_steps ! Normalise the area between 0 and 1
 do i = rank, num_steps, num_procs ! Only loop over our part of the sum
   x = (i + 0.5)*step_size
   sum = sum + 4.0/(1.0+x*x)
 end do
 my_area = step_size*sum


! Get that partial sums from all the processes, add them up, and give to the root  syntax: input variable, output
variable, data size, MPI Type, Operation, output rank, MPI World, error int
 call mpi_reduce(my_area,area,1,MPI_DOUBLE,MPI_SUM,root,MPI_COMM_WORLD,ierr)

end function
end program
```

```
[vanilla@hopper code]$ module load intel-mpi/2019.10.317-ruxn
[vanilla@hopper code]$ mpiifort calc_pi_mpi.f90 -o calc_pi_mpi
```

Compile the code with Intel MPI FORTRAN compiler

```
#!/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”.

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.

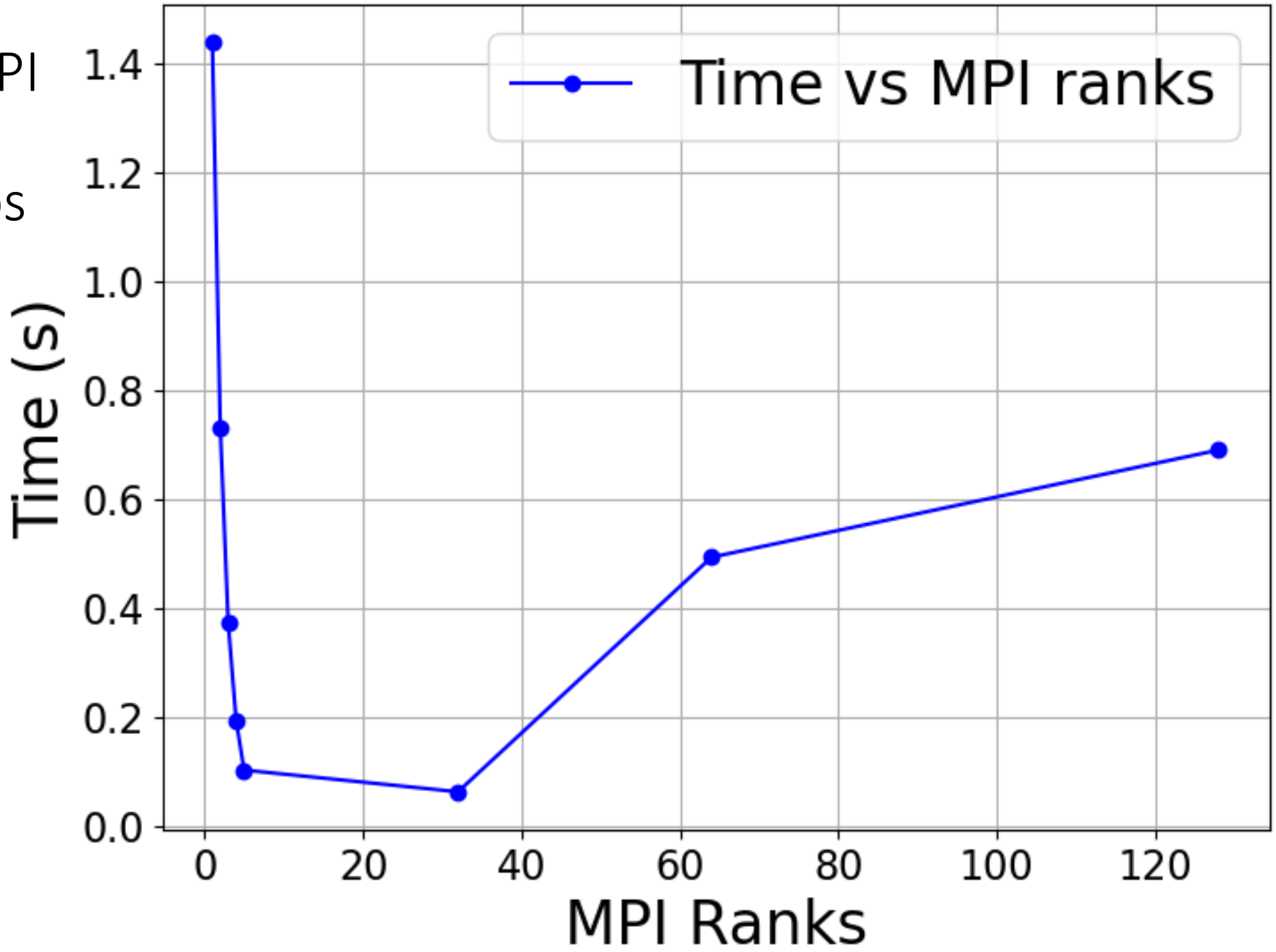
What is the relationship between the number of tasks and how cast it calculates pi?

```
#!/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 intel/20.0.4 intel-mpi/2019.10.317-ruxn

srun --mpi=pmi2  code/calc_pi_mpi 1000000000
```

sbatch slurm/calc_pi_mpi_f90.sh

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

[vanilla@hopper code]$ seq 1 1 6 | parallel srun --partition debug --ntasks {} calc_pi_mpi 100000
Job 3056979 running on hopper011
Pi=3.141612602973879 (Diff=.1986196130587814E-04)
(calculated in .2641998E-02s with 100000 steps and 2 processes)
Job 3056977 running on hopper011
Pi=3.141612602973873 (Diff=.1986196130010498E-04)
(calculated in .2910048E-03s with 100000 steps and 1 processes)
Job 3056976 running on hopper011
Pi=3.141612602973876 (Diff=.1986196130232543E-04)
(calculated in .4189983E-03s with 100000 steps and 3 processes)
Job 3056978 running on hopper011
Pi=3.141612602973876 (Diff=.1986196130321360E-04)
(calculated in .2626002E-02s with 100000 steps and 6 processes)
Job 3056980 running on hopper011
Pi=3.141612602973874 (Diff=.1986196130054907E-04)
(calculated in .3805004E-02s with 100000 steps and 4 processes)
Job 3056981 running on hopper011
Pi=3.141612602973876 (Diff=.1986196130232543E-04)
(calculated in .2314001E-02s with 100000 steps and 5 processes)
```

Speedup with MPI Rank
for 2×10^{10} steps



Vector addition with MPI and CUDA

- In this example we accelerate our vector addition by using GPUs.
- The code is organized so that it is as easy to understand as possible *not* to be ass efficient as possible.

```
$ cd ~/workshops/intro_workshop/code/vecadd
```

```
$ 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
```

Compile the code using mpi C compiler and Nvidia CUDA Compiler

Use the Xena cluster which has K40 GPUs with Compute Architecture 35

```
mfricke@xena:~/workshops/intro_workshop/code/vecadd $ make gpu
nvcc -arch sm_35 -c vecadd_gpu.cu -o vecadd_gpu.o
nvcc warning : The 'compute_35', 'compute_37', 'sm_35', and 'sm_37' architectures are deprecated, and may be
removed in a future release (Use -Wno-deprecated-gpu-targets to suppress warning).
mpic++ -c vecadd_mpi_gpu.c -o vecadd_mpi_gpu.o
mpic++ vecadd_mpi_gpu.o vecadd_gpu.o -lcudart -o vecadd_mpi_gpu
mfricke@xena:~/workshops/intro_workshop/code/vecadd $ make cpu
mpic++ vecadd_mpi_cpu.c -o vecadd_mpi_cpu
```

```
mfricke@xena:~/workshops/intro_workshop/code/vecadd $ ls
Makefile  vecadd_gpu.cu  vecadd_mpi_cpu  vecadd_mpi_cpu.c  vecaddmpi_cpu.sh  vecadd_mpi_gpu.c
```

```
#!/bin/bash

#SBATCH --partition debug
#SBATCH --ntasks=4

module load openmpi

srun ./vecadd_mpi_cpu $SLURM_NTASKS
```

Vecadd Creates MPI Processes to solve the addition together

- Data Node (node here does not refer to a computer as before but to a process)
  - The Data Node manages the compute nodes. It send them data to process and receives and coordinates the results.

The Compute Node process can be either:

- Compute Node (CPU)
  - Performs the vector addition using CPU
- Compute Node (GPU)
  - Performs the vector addition using GPU