Parallel Python Using MPI4Py

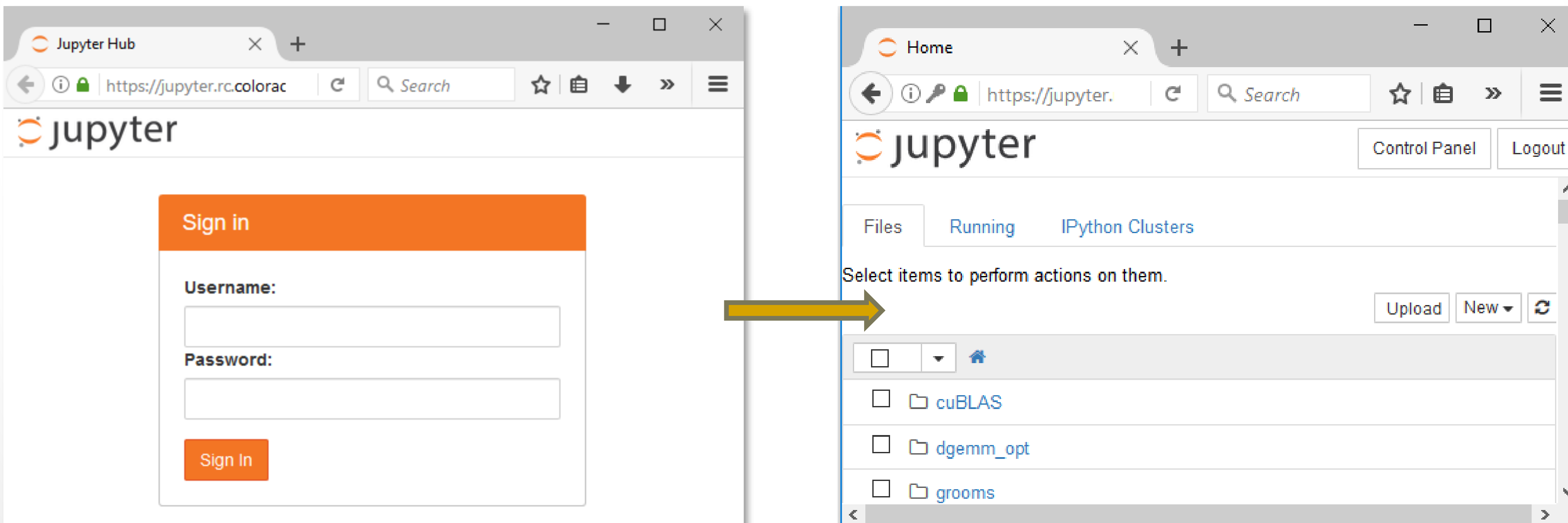
Nick Featherstone
CU Research Computing

Web Link to These Slides

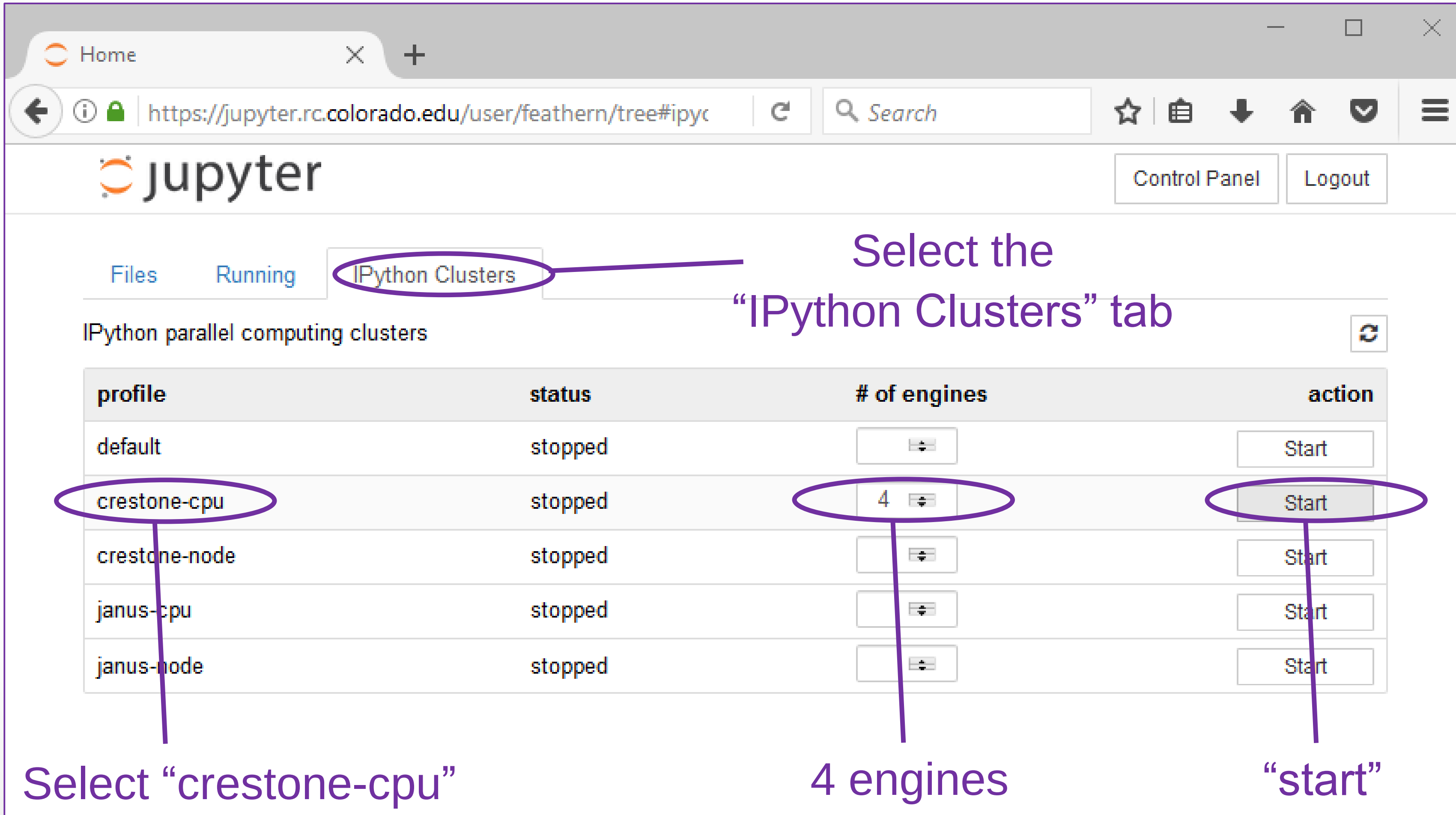
Getting started…

Login to the RC Jupyter Hub:

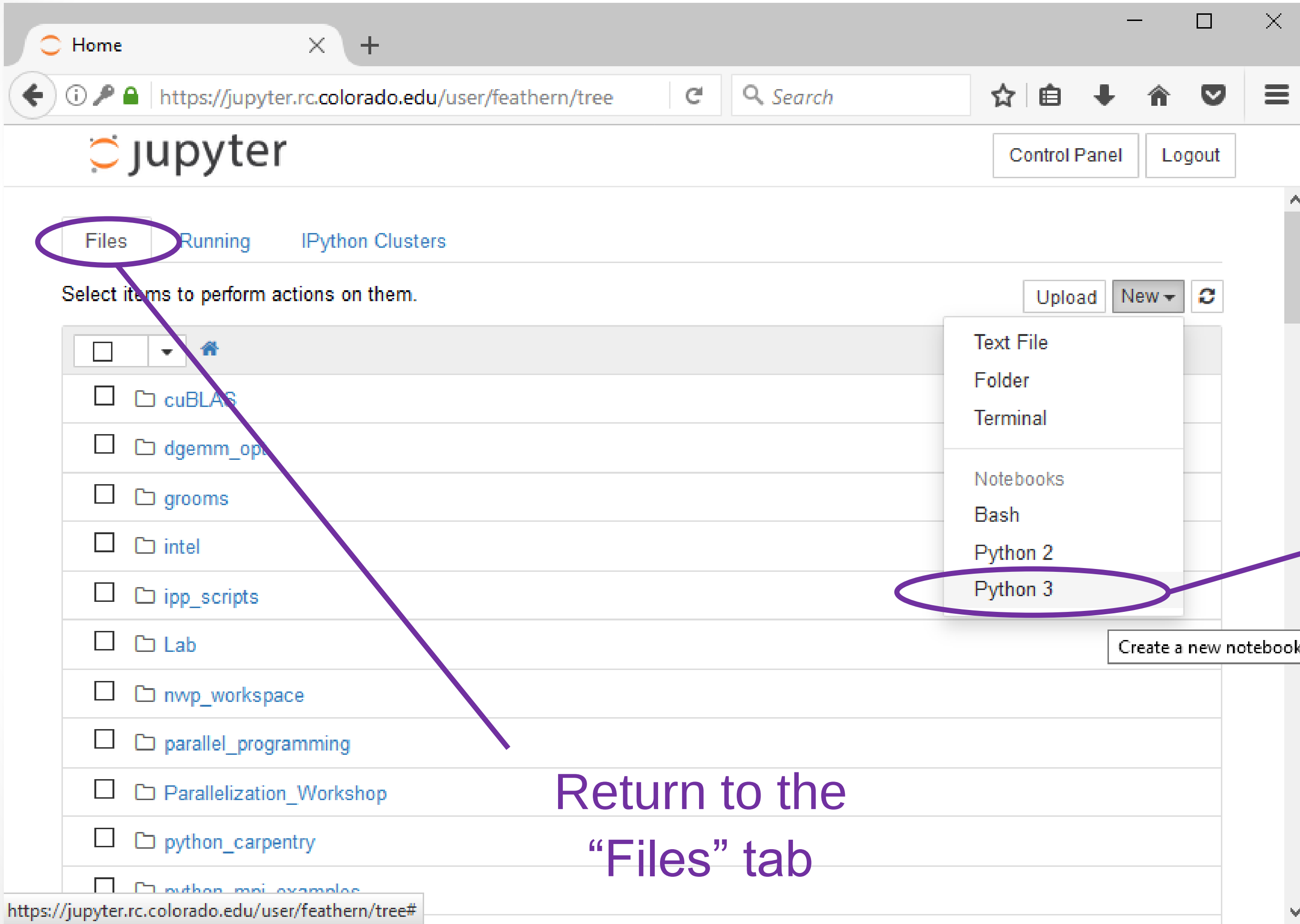
https :// jupyter . rc . colorado . edu



Getting started…



Getting started…

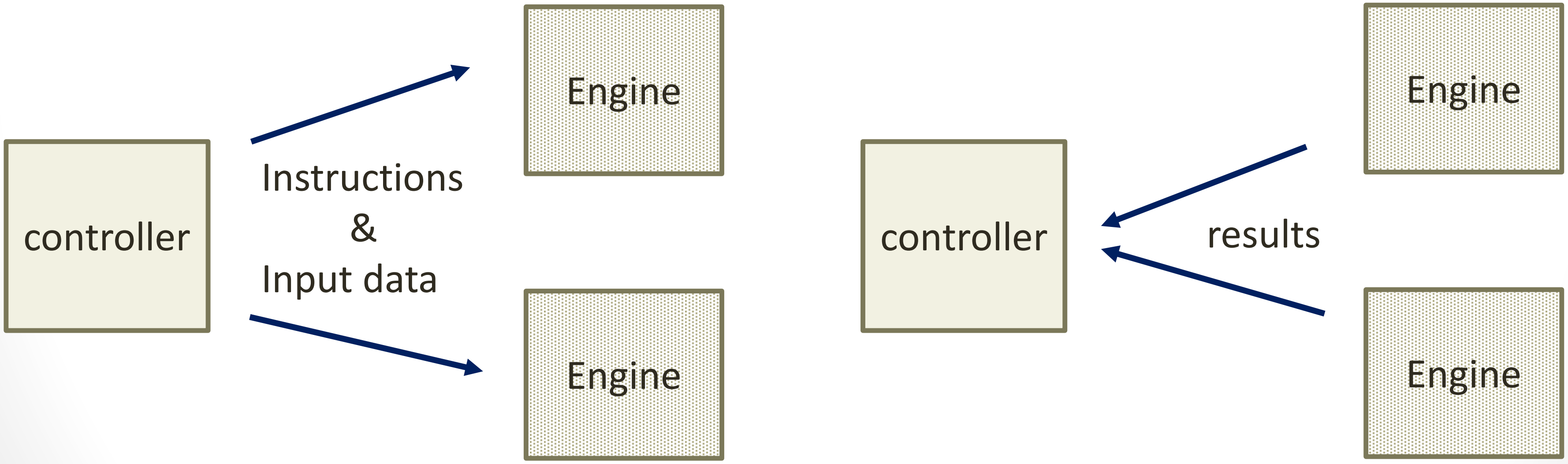


Start a Python 3 Notebook

Return to the "Files" tab

Engine/Controller Paradigm

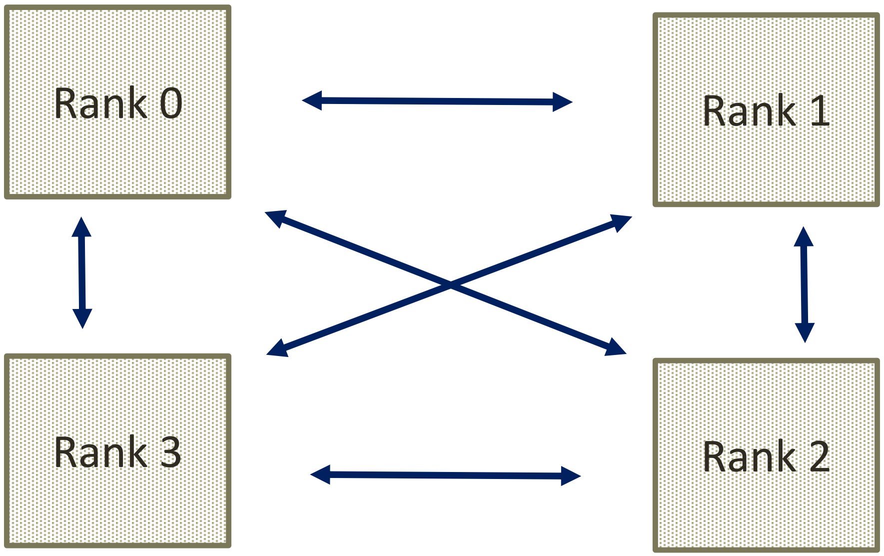
- What did we just do?
  initiated a controller python session and 4 python engines
- **Most** of your code runs on the **controller (Jupyter notebook)**
- **Some** of your code runs on the **engines (the crestone cpu's)**



Documentation: http://ipyparallel.readthedocs.io/en/latest/multiengine.html

Message-Passing Paradigm

- No one is really in control.
- Everyone can communicate with each other
- **Entire code** runs on each process
- Processes referred to as MPI ranks



MPI4PY Documentation: https://pythonhosted.org/mpi4py/usrman/

Message passing

- Most natural and efficient paradigm for distributed-memory systems
- Two-sided, send and receive communication between processes
- Efficiently portable to shared-memory or almost any other parallel architecture:
  - “assembly language of parallel computing” due to universality and detailed, low-level control of parallelism

More on message passing

- Provides natural synchronization among processes (through blocking receives, for example), so explicit synchronization of memory access is unnecessary
- Sometimes deemed tedious and low-level, but thinking about locality promotes
  - good performance,
  - scalability,
  - Portability
- Dominant paradigm for developing portable and scalable applications for massively parallel systems

Hello World

Open this file: ***(DO NOT COPY/RUN THE PROGRAM YET)***

Parallelization Workshop / Day3-Parallel_Python /

session3_mpi4py / examples /

hello1.py

Every MPI program has two very important pieces

- Initialize communication

```
from mpi4py import MPI
```

- Finalize communication

```
MPI.Finalize()
```

- The last very important part is commented out…

Logistics

In this workshop, we will be running our MPI code using the Jupyter notebook

- This is non-standard, but useful for an instructional setting
- We have to 'hack' things a bit to make ipyparallel and MPI work together
- Modified workflow:

Cell 1

```
import ipyparallel
rc=ipyparallel.Client(profile='crestone-cpu')
shift+enter
```

Do this once

Cell 2

```
%%px
Cut+paste code
shift+enter
```

Do this for each new cell

- Normal codes invoking MPI Finalize are found in the *standard* directory
- DO NOT use those programs in today's session
- A sample batch script has been placed in the *standard* directory

Hello World

Most MPI programs have some other common components

- Define a communicator used to communicate within the process pool

```
comm = MPI.COMM_WORLD
```

- Identify number of processes

```
num_proc = MPI.COMM_WORLD.Get_size()
```

- Identify this process rank/ID

```
my_rank = MPI.COMM_WORLD.Get_rank()
```

- Recall that the entire code is running on each MPI rank
- num_proc will be the same on all ranks
- my_rank will receive a unique value for all ranks

Open/Run this file: Parallelization Workshop / Day3-Parallel_Python /

session3_mpi4py / examples /

hello1.py

Hello World (2)

- Code is running on every process
- We don't always want every process performing I/O
- Common practice: use rank 0 in these situations
- Quick exercise:
  - Modify this program so that only even-numbered ranks print 'Hello from node…' message

```
if (my_rank == 0):
    write stuff
```

Open/Run this file: Parallelization Workshop / Day3-Parallel_Python /

session3_mpi4py / examples /

hello2.py

Flow Control using Barriers

- Different ranks may execute code as slightly different speeds

- This can lead to:
  - Jumbled output (everyone prints at once)
  - Race conditions (some ranks begin a task too soon)

- MPI's Barrier provides a method of synchronization

```
comm.Barrier()
```

- When a communicator's barrier method is called, all ranks in that communicator pause until everyone reaches that point in the code.

Flow Control using Barriers

Open/Run this file: Parallelization Workshop / Day3-Parallel_Python /

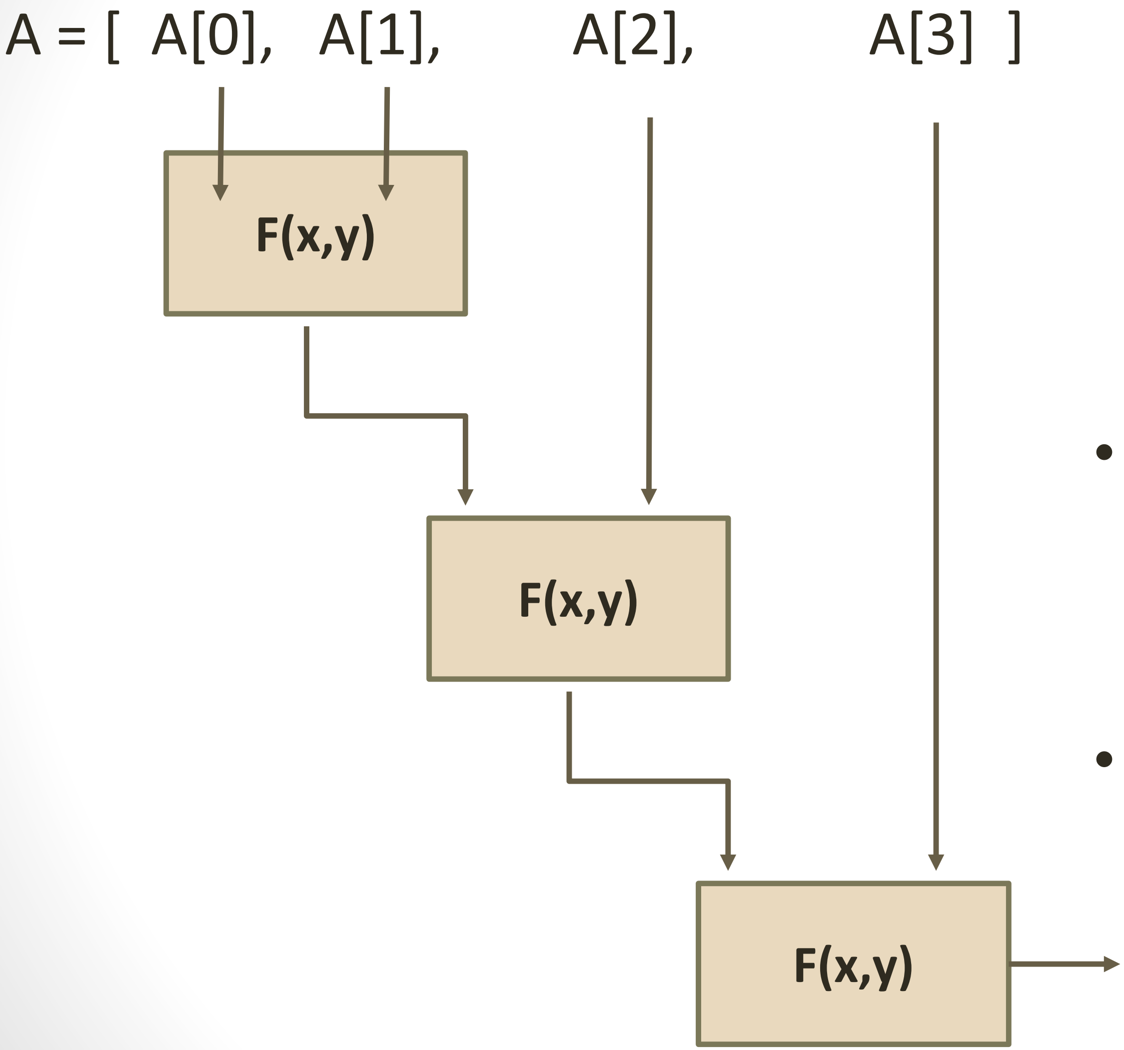
session3_mpi4py / examples /

barrier.py

- Note the use of sys.stdout.flush().
- If we don’t flush the output buffer, the barrier will appear to be ineffective (try taking the flush out).
- EXERCISE:

  Modify exercises/barrier_ex.py so that the different MPI ranks print ‘hello from node…’ in ascending order by rank.

Recall: Reduction



- Sequential function evaluation using results from previous evaluations in tandem with new inputs
- Think:
  - Adding numbers
  - Computing factorials
- Common in parallel applications

Scalar Result

Reduction in MPI

- Without the a dedicated controller, reduction requires some organized communication

- MPI’s AllReduce functionality handles this ‘under the hood’

Open/Run this file: Parallelization Workshop / Day3-Parallel_Python /

session3_mpi4py / examples /

reduction.py

Reduction Syntax in MPI4PY

```
comm.Allreduce([local_sum, MPI.DOUBLE], [global_sum,MPI.DOUBLE], op=MPI.SUM)
```

- When performing an allreduce, we provide a few pieces of information
  - The communicator (implicitly stated via comm.Allreduce)
  - A list containing
    - The variable to be reduced across all processes
    - The TYPE of the reduction variable
  - A list containing
    - The variable in which to store the result
    - The TYPE of the result variable
  - The type of reduction we wish to perform
- For this workshop, we use two variable types:
  - MPI.DOUBLE : equivalent to 'float64' in numpy
  - MPI.INTEGER : equivalent to 'int32' in numpy
- Common reduction operations
  - MPI.MAX : computes the max
  - MPI.MIN : computes the min
  - MPI.SUM : computes the total

Exercise: Reduction

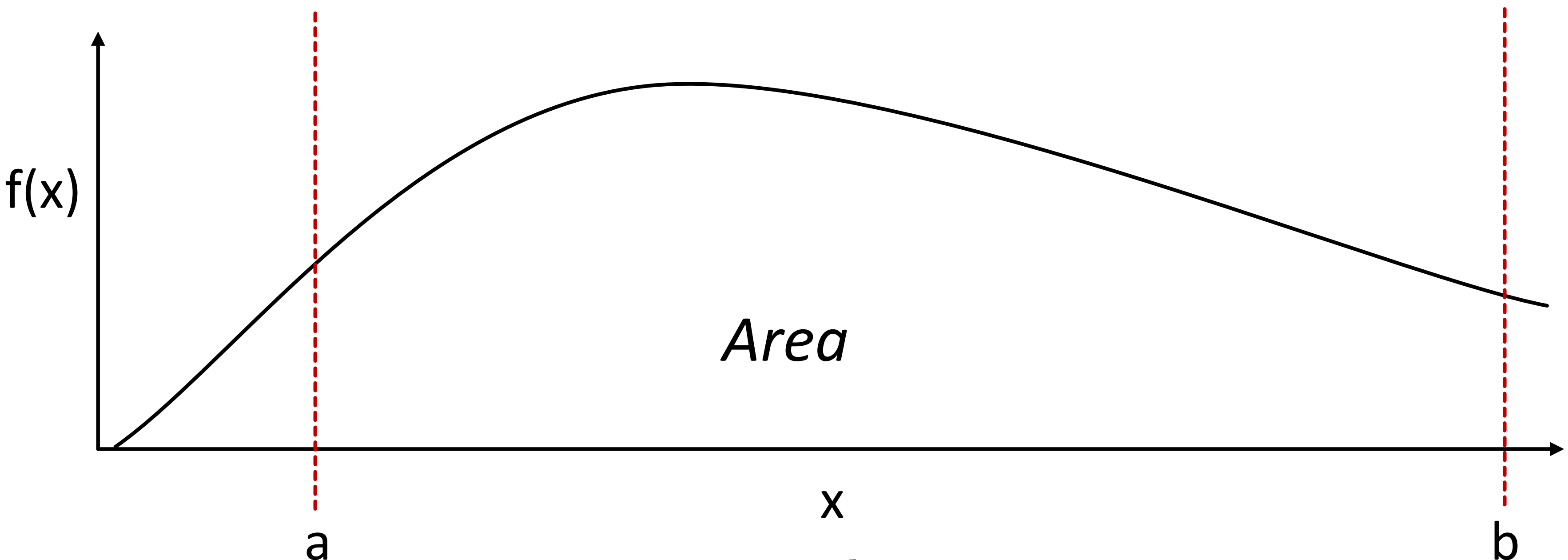
Open/Run this file: Parallelization Workshop / Day3-Parallel_Python /

session3_mpi4py / exercises /

collatz.py

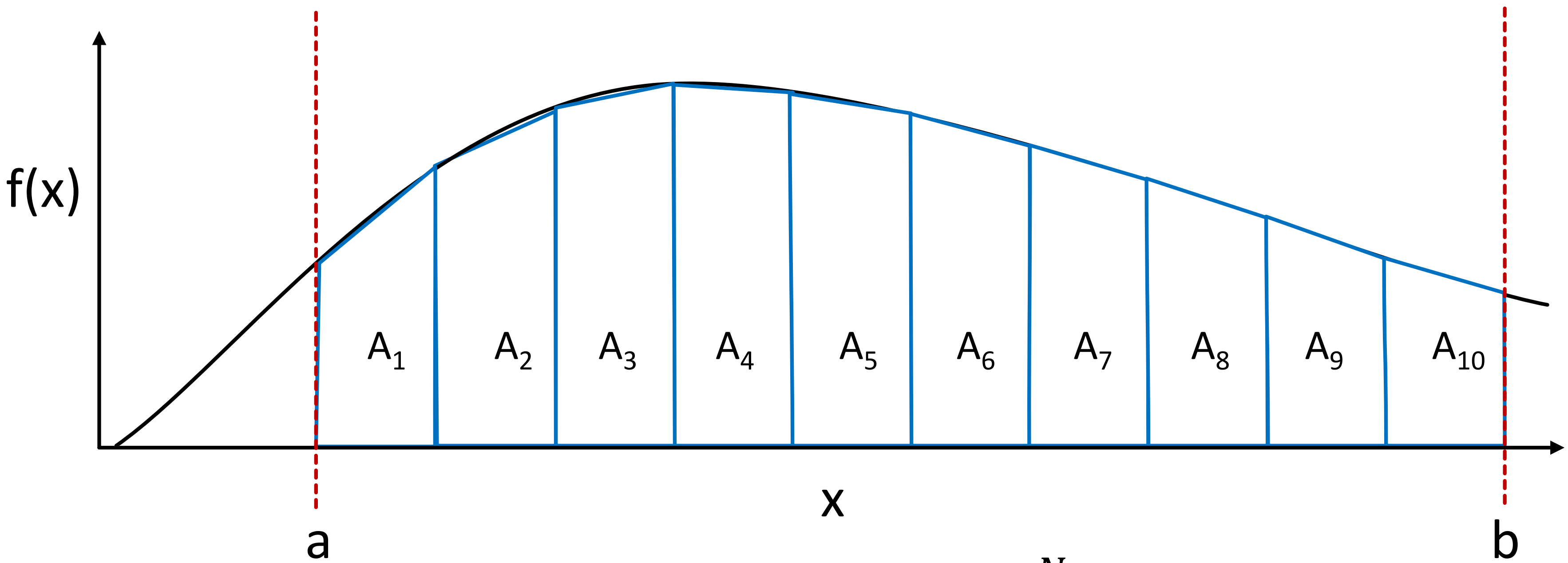
- Modify this code so that each rank uses Allreduce to compute the longest Collatz sequence occurring for numbers from 1 to 4000.

Reduction Application: Integrals



$$Area = \int_a^b f(x)dx$$

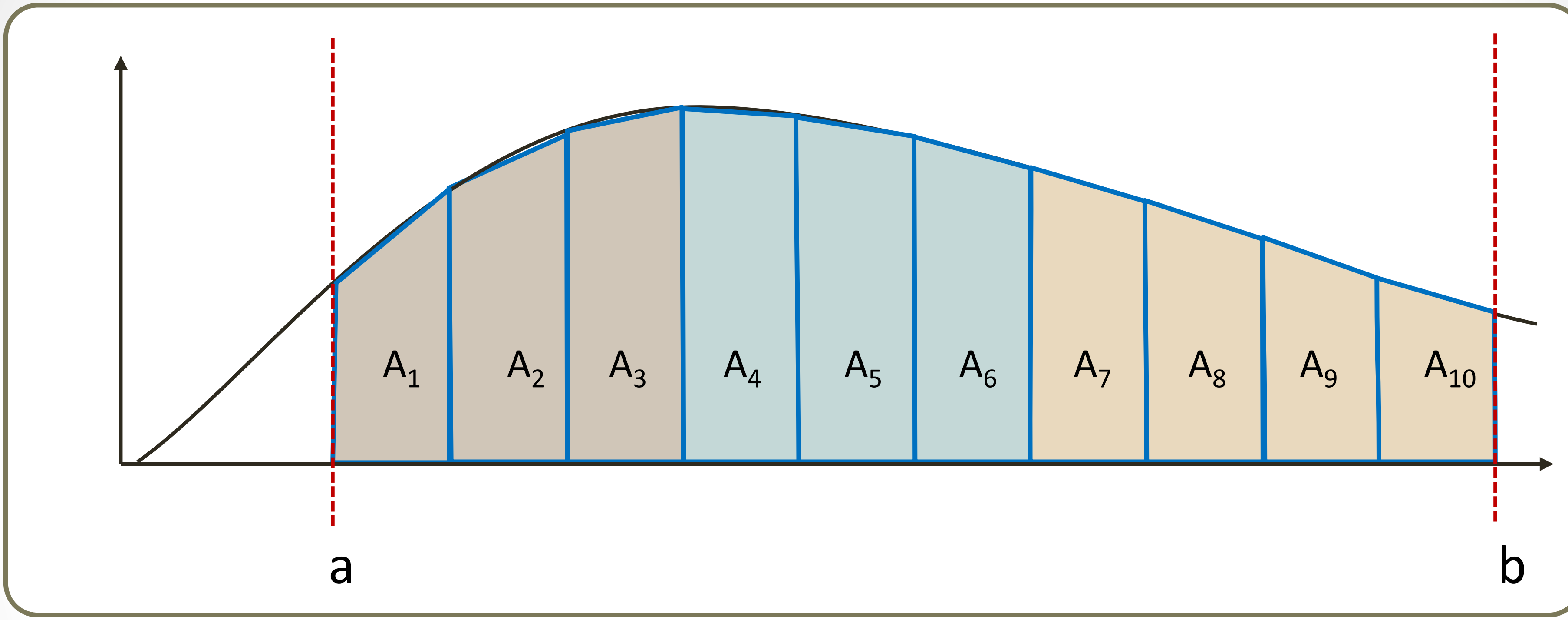
Reduction Application: Integrals



$$Area = \int_a^b f(x)dx \approx \sum_{i=1}^{N} A_i$$

- Trapezoidal Rule

Load Balancing



- Idea: Assign each MPI rank a different range in x
- Sum over areas at the end via Allreduce

Exercise: Integration

Open/Run this file: Parallelization Workshop / Day3-Parallel_Python /

session3_mpi4py / exercises /

trapezoid.py

Let have a look together…

Exercise: Integration

- We've started the problem setup already:

```
ntrap = 1000000/num_proc
```

- New local limits of integration : myxone, myxtwo
- Use my_rank and num_proc to modify these limits appropriately…

```
xone = 1.0
xtwo = 2.0

deltax = (xtwo-xone)
myxone = xone
myxtwo = myxone+deltax
```

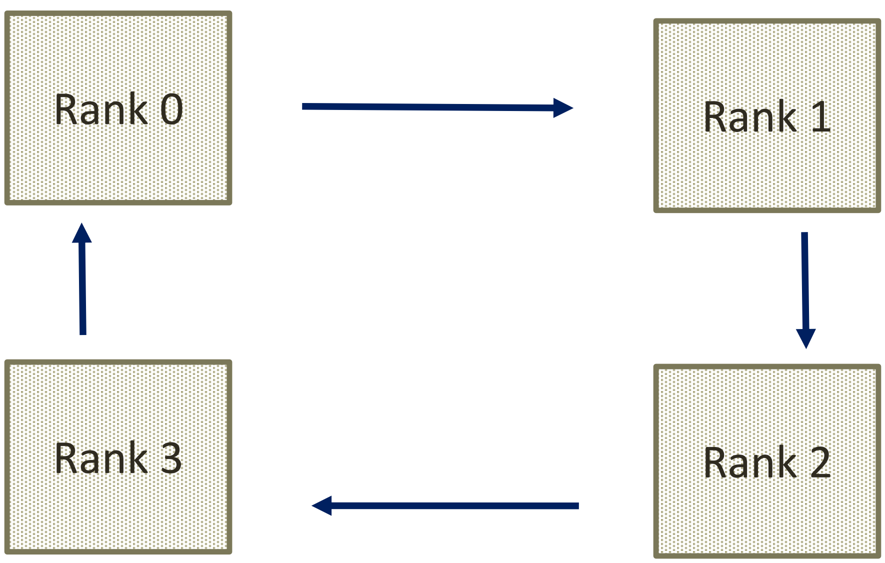
… then run the code!

Message-Passing Example

Open/Run this file: Parallelization Workshop / Day3-Parallel_Python /

session3_mpi4py / examples /

token_pass.py



Message Passing Questions

Which process is sending the message?

Where is the data on the sending process?

What kind of data is being sent?

How much data is there?

Which process is going to receive the message?

Where should the data be stored on the receiving process?

What amount of data is the receiving process prepared to accept?

In C/Fortran, you specify ALL of these. Not in Python! ☺

Message Passing Syntax

- To receive, you must specify
  - Receive buffer (msg)
  - MPI communicator (comm)
  - The MPI rank of the sender (source)
  - An identifying integer tag (tag; must match send tag)

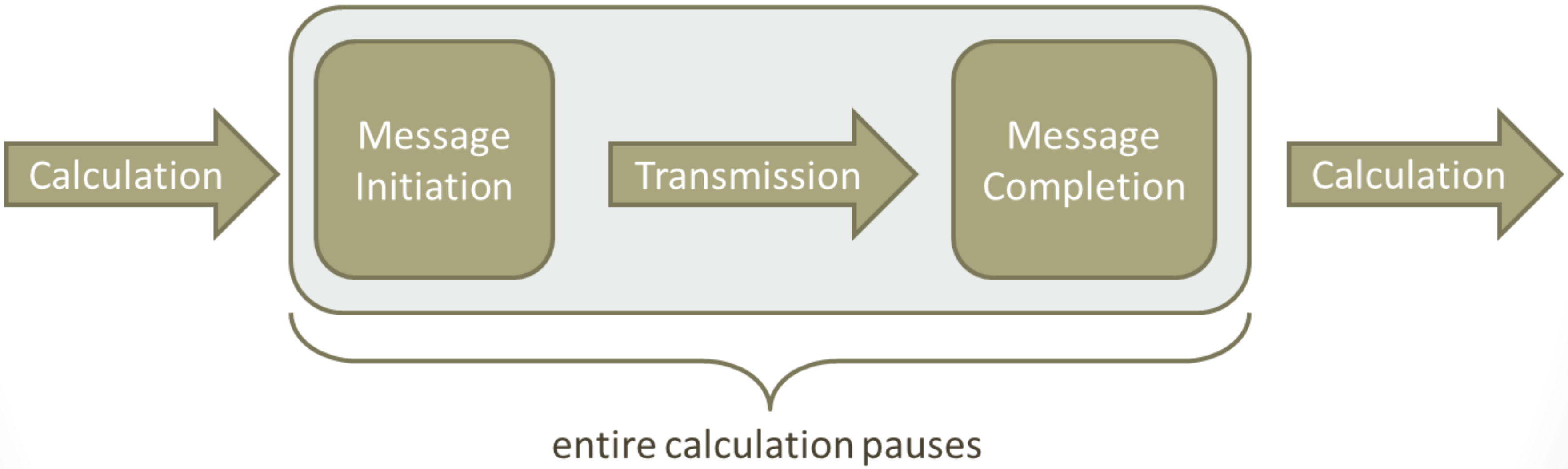
```
msg = comm.recv(source=source_ID, tag=msg_tag)
```

- To send, you must specify
  - The send buffer (msg)
  - MPI communicator (comm)
  - The MPI rank of the receiver (dest)
  - An identifying integer tag (tag; must match receive tag)

```
comm.send(msg,dest=dest_ID, tag=msg_tag)
```

Blocking Communication

- Sends and Receives are blocking
- Non-blocking counterparts do exist (Isends/Irecvs)



- Blocking communication can have unintended consequences…

Deadlock

Good Code

```
If (my_rank == 0):
        send( to rank N)
        receive( from rank N)
If (my_rank == N):
        receive( from rank 0)
        send( to rank 0)
```

Bad Code

```
If (my_rank == 0):
        send( to rank N)
        receive( from rank N-1)
If (my_rank == N):
        send( to rank 0)
        receive( from rank 0)
```

Neither send ever completes

Exercise: Token Passing

Open/Run this file: Parallelization Workshop / Day3-Parallel_Python /

session3_mpi4py / exercises /

token_pass_ex.py

Reverse the order of the message passing:

- Send to rank below
- Receive from rank above