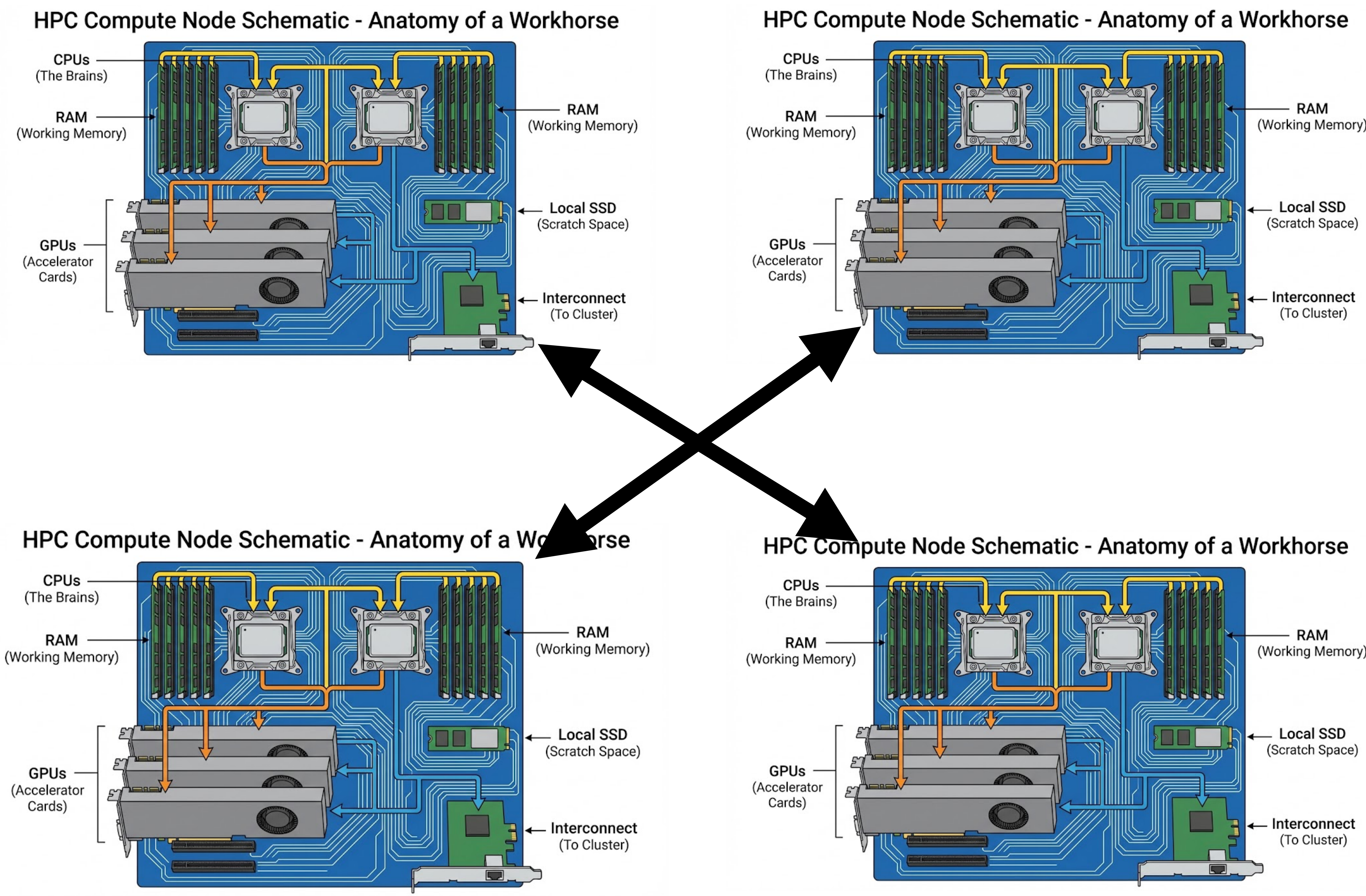
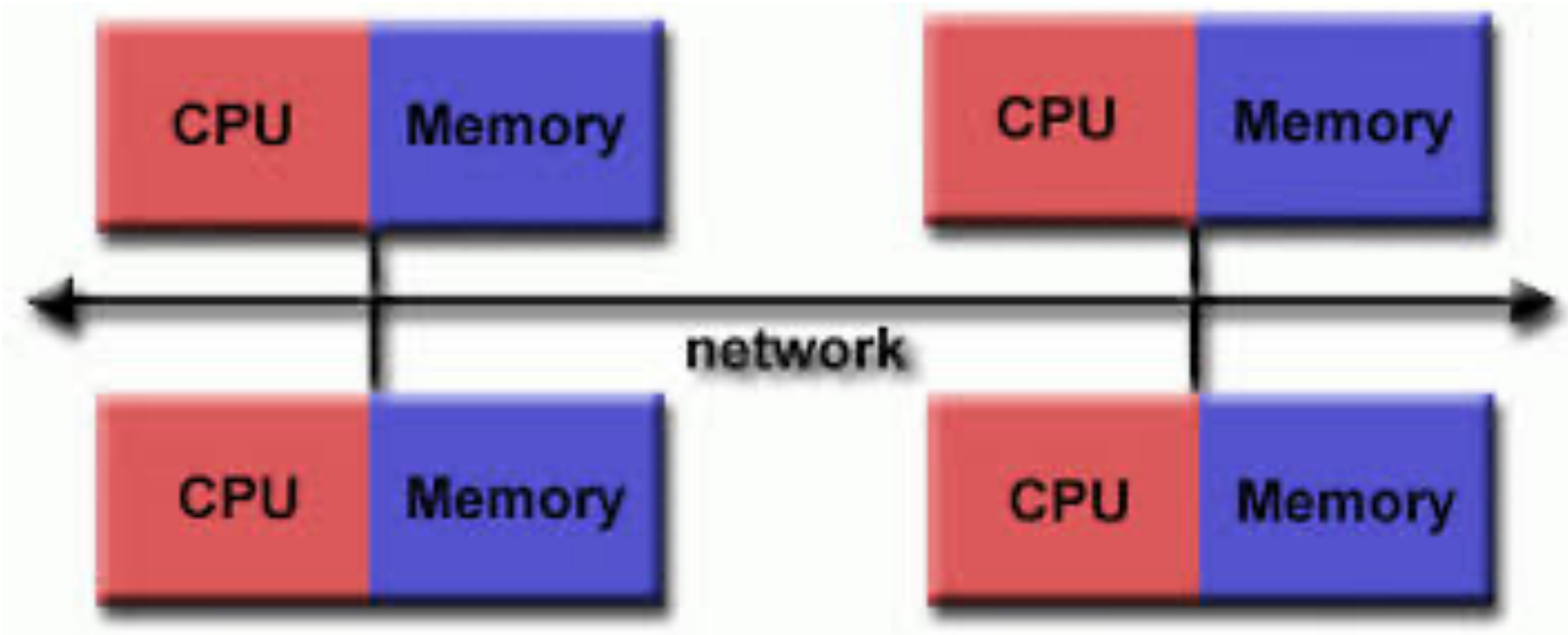
Lecture 11: Distributed memory parallelism I

CEE 690

Connecting compute nodes

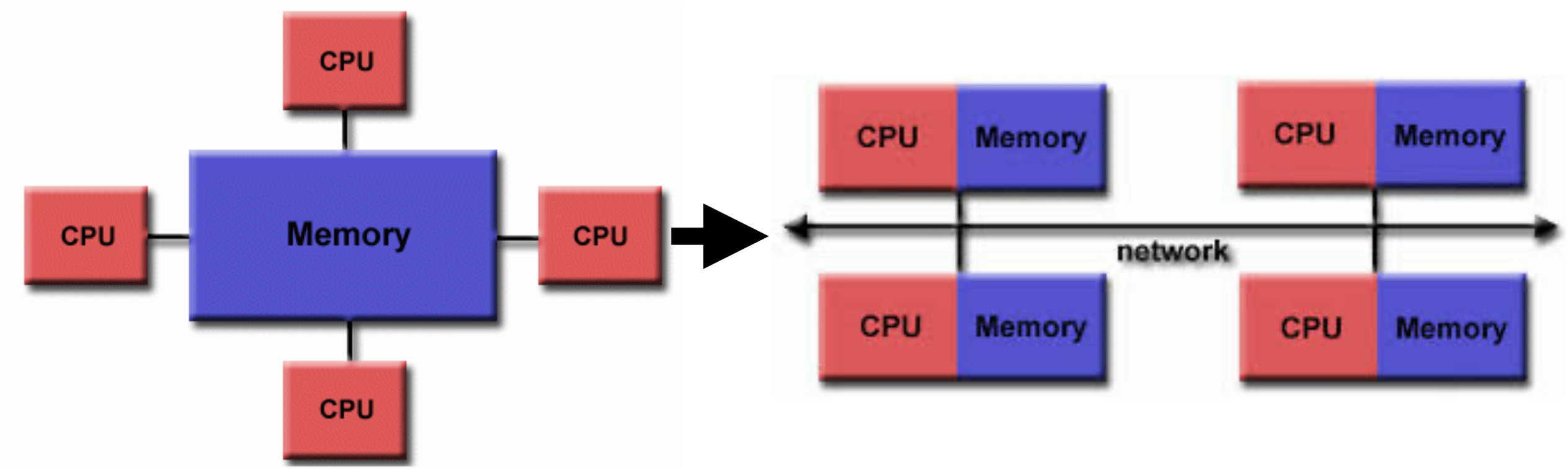


Distributed memory: All CPUs have different buckets



How would you leverage this architecture when developing software?

You can mimic distributed memory on single compute nodes



Personal note: I actually almost always do this and rarely use shared parallelism

Lots of parallel processes

```
[nc153 $ srun -n 16 echo "Hello world!"
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

We need more than just printing statements. We need process ids and we need to be able to potentially “talk” to each other.

Message Passing Interface

What is MPI?

- **Isolated memory** - Every MPI process has its own private RAM. By default, processes are invisible to each other.
- **Same script/different data—** The same script is launched on each process. Process/rank ID is used to divide the work.
- **Explicit communication —** The programmer must manually orchestrate “messages” between the processes.
- **Network-level scaling —** The message passage paradigm allows us to scale this to many, many compute nodes.
- **Collective operations —** There are approaches to have all the processes interact/talk to each other at once.

MPI4PY

MPI for Python

Author: Lisandro Dalcin
Contact: dalcinl@gmail.com
Date: Oct 10, 2025

Abstract

MPI for Python provides Python bindings for the *Message Passing Interface* (MPI) standard, allowing Python applications to exploit multiple processors on workstations, clusters and supercomputers.

This package builds on the MPI specification and provides an object oriented interface resembling the MPI-2 C++ bindings. It supports point-to-point (sends, receives) and collective (broadcasts, scatters, gathers) communication of any *picklable* Python object, as well as efficient communication of Python objects exposing the Python buffer interface (e.g. NumPy arrays and builtin bytes/array/memoryview objects).

Contents

MPI4PY

MPI for Python / MPI for Python View page source

MPI for Python

Distributed parallelization is not hard once you know what you are doing. But initially it can be quite painful.

CONTENTS

This package builds on the MPI specification and provides an object oriented interface resembling the MPI-2 C++ bindings. It supports point-to-point (sends, receives) and collective (broadcasts, scatters, gathers) communication of any *picklable* Python object, as well as efficient communication of Python objects exposing the Python buffer interface (e.g. NumPy arrays and builtin bytes/array/memoryview objects).

Contents

MPI4PY “hello world”

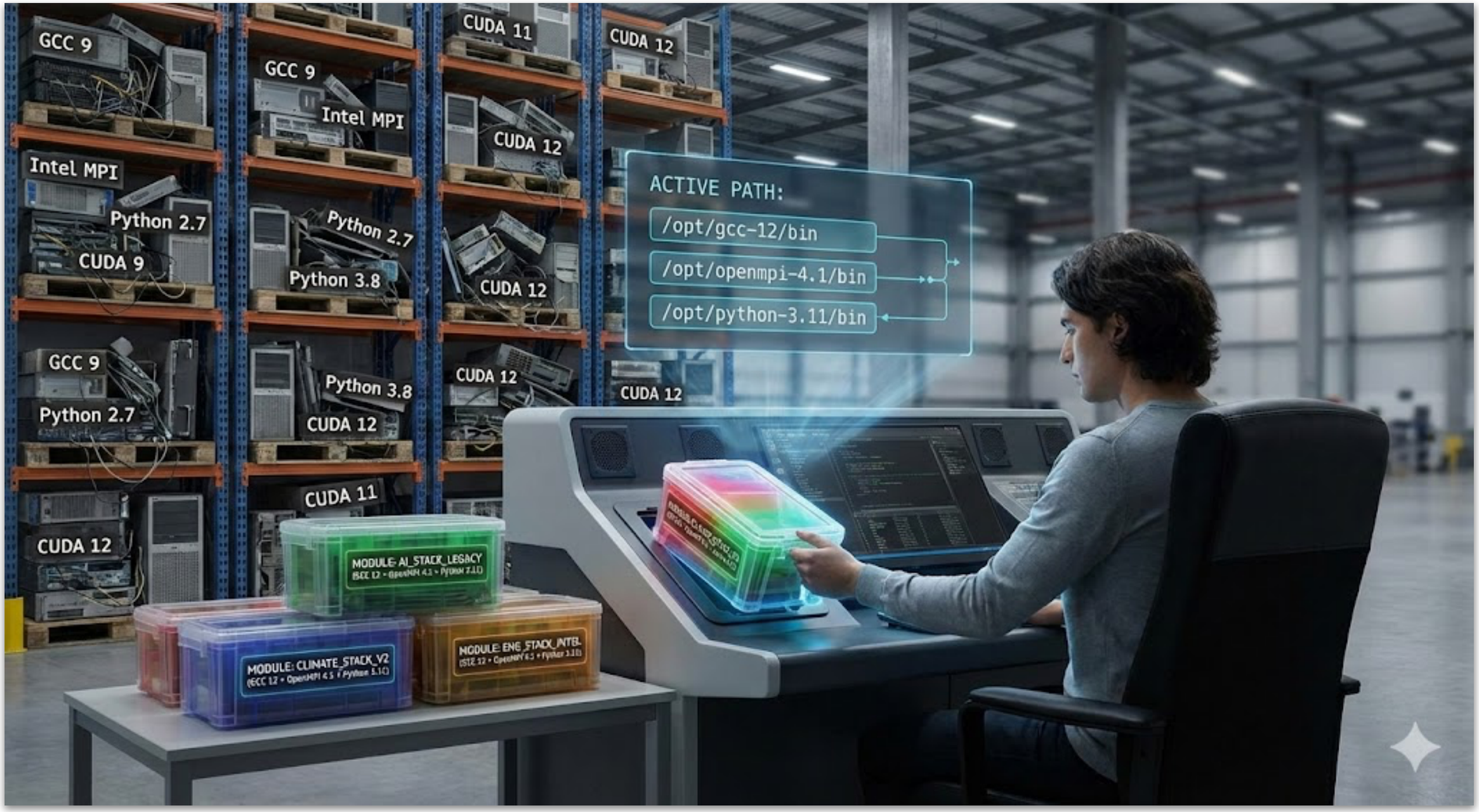
```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

print(f"Hello world from rank {rank} out of {size} total processes.")
```

```
(cee690) nc153 $ mpirun -n 16 python mpi4py_hello_world.py
Hello world from rank 15 out of 16 total processes.
Hello world from rank 12 out of 16 total processes.
Hello world from rank 13 out of 16 total processes.
Hello world from rank 14 out of 16 total processes.
Hello world from rank 11 out of 16 total processes.
Hello world from rank 1 out of 16 total processes.
Hello world from rank 0 out of 16 total processes.
Hello world from rank 2 out of 16 total processes.
Hello world from rank 3 out of 16 total processes.
Hello world from rank 5 out of 16 total processes.
Hello world from rank 9 out of 16 total processes.
Hello world from rank 4 out of 16 total processes.
Hello world from rank 7 out of 16 total processes.
Hello world from rank 6 out of 16 total processes.
Hello world from rank 8 out of 16 total processes.
Hello world from rank 10 out of 16 total processes.
```

Detour: We need to talk about modules



What is a module?

A “module” is a tool that allows us to modify our user environment at the terminal to access specific software versions, libraries, and compilers

Wait, what?

```
nc153 $ module avail
```

```
------------------------------------------ /opt/apps/modulefiles ------------------------------------------
7-Zip/22.01                 GNUPLOT/5.2.6                 null                                 Stacks/2.0b
almaBTE/v1.3.2              Go/1.18.4                     NVidia/nvhpc-byo-compiler/24.5       Stacks/2.1
AMBER/18-GPU                GoogleCloudCLI/5.8            NVidia/nvhpc-hpcx-cuda12/24.5        Stacks/2.55
AMBER/18-GPU-update         GRASS/7.6.0                   NVidia/nvhpc-hpcx/24.5               STAR/2.5.3a
AMBER/18-MPI                groff/1.22.3                  NVidia/nvhpc-nompi/24.5              STAR/2.7.5a
AMBER/20-GPU                GROMACS/2018                  NVidia/nvhpc-openmpi3/24.5           STAR/2.7.5c
AMBER/20-MPI-GPU            GROMACS/2018-2-GPU            NVidia/nvhpc/24.5                    STAR/2.7.10a
AMBER/22-MPI_GPU            GROMACS/2018-4-GPU            nvtop/3.1.0                          STAR/2.7.11a
AMBER/24-MPI_GPU            GROMACS/2020-4                ODBC/2.3.12                          StringTie/2.0
AMBER/24-MPI_GPU-plumed     GROMACS/2020-4-GPU            OpenBLAS/3.23                        Subread/1.6.3
Anaconda2/2.7.13            GROMACS/2020-5-GPU-rhel8      OpenCV/4.0.1                         Subread/2.0.3
Anaconda3/3-2019            GROMACS/2022-5                OpenCV/4.5.1-rhel8                   Subversion/1.14.3
Anaconda3/3.5.2             GROMACS/2024-GPU              OpenMPI/2.0.3                        Tassel/5
Anaconda3/5.1.0             gsl/2.2.1                     OpenMPI/2.1.0                        Tassel/5.0
Anaconda3/2021.05           gsl/2.4                       OpenMPI/4.0.1                        TensorRT/6.0.1.5
Anaconda3/2024.02           gsl/2.6-rhel8                 OpenMPI/4.0.5-rhel8                  TensorRT/7.0.0.11
ANGSD/0.934                 Guppy/6.5.7                   OpenMPI/4.1.1-rhel8                  TK/8.6.11
AnnotSV/3.4.4               Gurobi/8.01                   OpenMPI/4.1.5rc2                     TOPAS/3.9
ANTs/2.4.2                  Gurobi/8.11                   OpenMPI/4.1.6                        TopHat/2.1.1
ARBD/mar21                  harp/140925                   openvscode-server/1.63.2             tree/1.0
```

There are “libraries” that are already installed on the HPC system. Modules allow us to access those “global” installations.

I don’t need conda/pip anymore?

```
Anaconda2/2.7.13
Anaconda3/3-2019
Anaconda3/3.5.2
Anaconda3/5.1.0
Anaconda3/2021.05
Anaconda3/2024.02
```

“Kind of”… It is true that you can load the “stack” that was installed by the administrator. However, you lose control (and my opinion, too much).

Personal opinion: When possible, I always just use your own local install of python->conda/pip

So why are talking about modules then?

- There are certain system-wide libraries that are installed by the administrator to the quirks/ specifications of the HPC system.

- When you need to install/compile a piece of software on HPC that requires the MPI library you WANT to use the system wide installs.

- And modules help you access just that!

Different flavors of MPI

- **Open MPI** - (Arguably) This is the most widely used implementation across academia and research
- **MPICH —** One of the oldest and most influential implementations. It is a reference implementation for many others.
- **Intel MPI —** Commercial implementation based on the MPICH source code but heavily optimized for Intel processors.
- **MVAPICH2 —** Version of MPICH designed specifically for Infiniband interconnect.
- **Vendor specific —** Cray MPT, IBM Spectrum MPI, and Microsoft MPI

MPI modules on the DCC

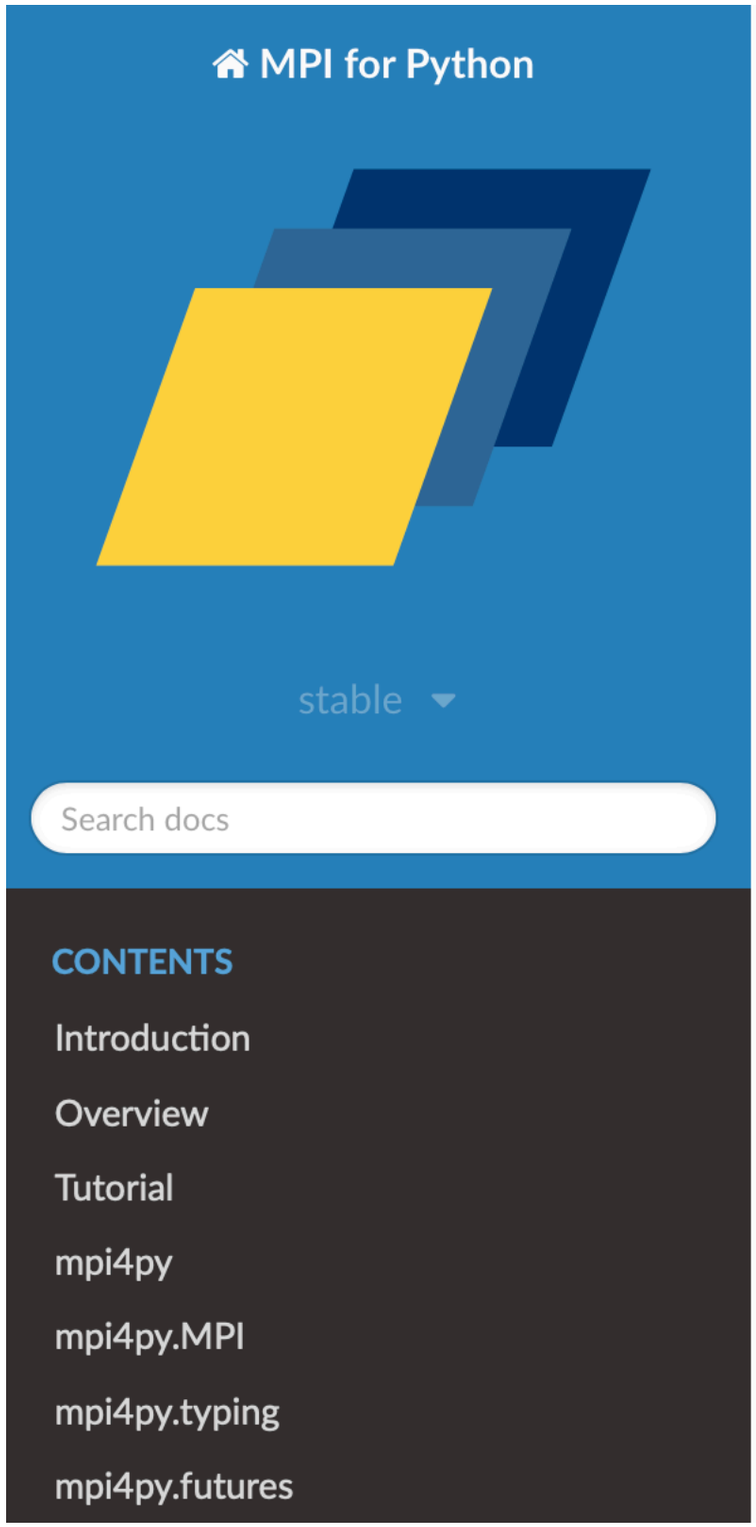
```
OpenCV/4.5.1-rhel8
OpenMPI/2.0.3
OpenMPI/2.1.0
OpenMPI/4.0.1
OpenMPI/4.0.5-rhel8
OpenMPI/4.1.1-rhel8
OpenMPI/4.1.5rc2
OpenMPI/4.1.6
openvscode-server/1.63.2
```

```
Infiniband/MPICH/4.2.0
Infiniband/MPICH/4.2.0-CUDA
Infiniband/MVAPICH/3.0
Infiniband/MVAPICH/3.0-CUDA
Infiniband/OpenMPI/5.0.2
Infiniband/OpenMPI/5.0.2-CUDA
```

```
MPICH/3.2.1
MPICH/3.2.1-intel
```

```
nc153 $ module load OpenMPI/4.1.6
OpenMPI/4.1.6
```

Back to mpi4py



MPI for Python / MPI for Python View page source

MPI for Python

Author: Lisandro Dalcin
Contact: dalcinl@gmail.com
Date: Oct 10, 2025

Abstract

MPI for Python provides Python bindings for the *Message Passing Interface* (MPI) standard, allowing Python applications to exploit multiple processors on workstations, clusters and supercomputers.

This package builds on the MPI specification and provides an object oriented interface resembling the MPI-2 C++ bindings. It supports point-to-point (sends, receives) and collective (broadcasts, scatters, gathers) communication of any *picklable* Python object, as well as efficient communication of Python objects exposing the Python buffer interface (e.g. NumPy arrays and builtin bytes/array/memoryview objects).

Contents

Installing mpi4py

1. Load your preferred MPI implementation

```
(cee690) nc153 $ module load Infiniband/OpenMPI/5.0.2
Infiniband/OpenMPI/5.0.2
```

2. Pip install mpi4py

```
nc153 $ pip install --no-cache-dir --force-reinstall --no-binary=mpi4py mpi4py
Collecting mpi4py
  Downloading mpi4py-4.1.1.tar.gz (500 kB)
     |████████████████████████████████| 500 kB 3.5 MB/s
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Installing backend dependencies ... done
    Preparing wheel metadata ... done
Building wheels for collected packages: mpi4py
  Building wheel for mpi4py (PEP 517) ... done
  Created wheel for mpi4py: filename=mpi4py-4.1.1-cp38-cp38-linux_x86_64.whl siz
e983e954c738e1a1ca5b17fffe5b17c7b53485abad8506ef8
  Stored in directory: /tmp/pip-ephem-wheel-cache-3mqsgh99/wheels/32/ec/98/6a3fb
e6ae65d99d549
Successfully built mpi4py
Installing collected packages: mpi4py
Successfully installed mpi4py-4.1.1
```

Recommendation… Add the modules that you need to your .bashrc

```
# 3. The magic function that captures metadata
log_command() {
    # Only log if the command isn't empty
    local last_cmd=$(history 1 | sed 's/^[ ]*[0-9]*[ ]*//')
    if [ -n "$last_cmd" ]; then
        echo "[$(date '+%Y-%m-%d %H:%M:%S')] [PWD: $PWD] $last_cmd"
    fi
}

# 4. Execute the function every time a command finishes
export PROMPT_COMMAND="log_command; $PROMPT_COMMAND"

# Load modules
module load OpenMPI/4.1.6
```

Recommendation… Add the modules that you need to your .bashrc

```
# 3. The magic function that captures metadata
log_command() {
    # Only log if the command isn't empty
    local last_cmd=$(history 1 | sed 's/^[ ]*[0-9]*[ ]*//')
    if [ -n "$last_cmd" ]; then
```

We can go back to mpi4py now

```
# 4. Execute the function every time a command finishes
export PROMPT_COMMAND="log_command; $PROMPT_COMMAND"

# Load modules
module load OpenMPI/4.1.6
```

MPI4PY “hello world”

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

print(f"Hello world from rank {rank} out of {size} total processes.")
```

```
(cee690) nc153 $ mpirun -n 16 python mpi4py_hello_world.py
Hello world from rank 15 out of 16 total processes.
Hello world from rank 12 out of 16 total processes.
Hello world from rank 13 out of 16 total processes.
Hello world from rank 14 out of 16 total processes.
Hello world from rank 11 out of 16 total processes.
Hello world from rank 1 out of 16 total processes.
Hello world from rank 0 out of 16 total processes.
Hello world from rank 2 out of 16 total processes.
Hello world from rank 3 out of 16 total processes.
Hello world from rank 5 out of 16 total processes.
Hello world from rank 9 out of 16 total processes.
Hello world from rank 4 out of 16 total processes.
Hello world from rank 7 out of 16 total processes.
Hello world from rank 6 out of 16 total processes.
Hello world from rank 8 out of 16 total processes.
Hello world from rank 10 out of 16 total processes.
```

Point to point (P2P) communication: Dictionaries

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
    print(rank,data)
```

```
(cee690) nc153 $ mpirun -n 2 python mpi4py_p2p.py
1 {'a': 7, 'b': 3.14}
```

P2P: Numpy arrays (fastest)

```python
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# passing MPI datatypes explicitly
if rank == 0:
    data = numpy.arange(10, dtype='i')
    comm.Send([data, MPI.INT], dest=1, tag=77)
elif rank == 1:
    data = numpy.empty(10, dtype='i')
    comm.Recv([data, MPI.INT], source=0, tag=77)
    print('Explicit',rank,data)

# automatic MPI datatype discovery
if rank == 0:
    data = numpy.arange(10, dtype=numpy.float64)
    comm.Send(data, dest=1, tag=13)
elif rank == 1:
    data = numpy.empty(10, dtype=numpy.float64)
    comm.Recv(data, source=0, tag=13)
    print('Automatic',rank,data)
```

```
(cee690) nc153 $ mpirun -n 2 python mpi4py_p2p_numpy.py
Explicit 1 [0 1 2 3 4 5 6 7 8 9]
Automatic 1 [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
```

Collective communication: Broadcasting

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
    data = np.arange(size*5, dtype='i')
else:
    data = np.empty(size*5, dtype='i')
comm.Bcast(data, root=0)
print(f"Rank {rank}",data)
```

Every rank receives the exact same data

```
(cee690) nc153 $ mpirun -n 4 python mpi4py_broadcast.py
Rank 1 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
Rank 3 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
Rank 2 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
Rank 0 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
```

Collective: Scattering

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
send_data = None
recv_data = None

if rank == 0:
    send_data = np.arange(size*5, dtype='i')
else:
    recv_data = np.empty(5, dtype='i')
comm.Scatter(send_data,recv_data, root=0)
if rank == 0:
    print(f"Rank {rank}",send_data)
else:
    print(f"Rank {rank}",recv_data)
```

The data is “chunked” as it is sent to the ranks (no overlaps)

```
(cee690) nc153 $ mpirun -n 4 python mpi4py_scatter.py
Rank 2 [10 11 12 13 14]
Rank 0 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
Rank 1 [5 6 7 8 9]
Rank 3 [15 16 17 18 19]
```

Collective: Gathering

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

sendbuf = np.zeros(5, dtype='i') + rank
recvbuf = None
if rank == 0:
    recvbuf = np.empty([size, 5], dtype='i')
comm.Gather(sendbuf, recvbuf, root=0)
if rank == 0:
    print(f"Rank {rank}",recvbuf)
```

The chunks are sent to the main process which aggregates them into a single array (opposite of scatter)

```
(cee690) nc153 $ mpirun -n 4 python mpi4py_gather.py
Rank 0 [[0 0 0 0 0]
 [1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]]
```

Collective communication: allreduce

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Define the data to be reduced (a generic Python integer)
send_data = rank

# Perform the all-reduce operation
result = comm.allreduce(send_data, op=MPI.SUM)

# The result is now available in 'result' on all processes
print(f"Rank {rank}: Result is {result}")
```

```
(cee690) nc153 $ mpirun -n 4 python mpi4py_allreduce.py
Rank 0: Result is 6
Rank 1: Result is 6
Rank 2: Result is 6
Rank 3: Result is 6
```

Collective communication: allreduce

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
```

And a lot more…

```python
result = comm.allreduce(send_data, op=MPI.SUM)

# The result is now available in 'result' on all processes
print(f"Rank {rank}: Result is {result}")
```

```
(cee690) nc153 $ mpirun -n 4 python mpi4py_allreduce.py
Rank 0: Result is 6
Rank 1: Result is 6
Rank 2: Result is 6
Rank 3: Result is 6
```

mpi4py summary I

- **Get_rank() —** It returns the integer ID of the current process.
- **Get_size() —** It returns the number of processes in the communicator.
- **Send(data, dest, tag) —** Pushes the data from rank i to the destination (dest) rank j.
- **Recv(data, source, tag) —** Rank j receives the data from the source rank i. This is a blocking call; it will just sit until it receives the data.
- **Barrier() —** A synchronization point. No process can pass this barrier until every process has reached it.

mpi4py summary II

- **Bcast(data, root=0) —** The main process (root) sends the same data to all processes.
- **Scatter(sendbuf, recvbuf, root=0) —** Chunks the array on the main (root) process and sends it to the processes.
- **Gather(sendbuf, recvbuf, root=0) —** Each worker sends their chunk to the main that then stitches them together.
- **Reduce(sendbuf, recvbuf, op=MPI.SUM, root=0) —** Combines data from all processes using a mathematical operation and delivers the result only to the main process.
- **Allreduce(sendbuf, recvbuf, op=MPI.SUM) —** Same as Reduce except all the processes receive the same result.

Lower-case vs Upper-case directives… It matters

- **Lower-case (send, recv, bcast)** - Uses Python “pickle” to serialize objects. It makes it possible to send lists, dictionaries, etc… but it is slow.
- **Upper-case (Send, Recv, Bcast)** - Uses buffer-optimized communication. It is much faster but requires the data to be a contiguous memory block (e.g., Numpy array)

Calculate mean using mpi4py

```python
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

def calculate_mean_collective():
    rows, cols = 5000, 5000
    data = None

    if rank == 0:
        # MAIN PROCESS: Only rank 0 creates the full matrix
        print(f"Main: Starting collective calculation with {size} processes...")
        data = np.random.randn(rows, cols)
        start_time = time.time()
    else:
        # WORKERS: Start with nothing
        data = None

    # 1. SCATTER: Automatically slice the data and distribute it.
    # We send rows // size to each process.
    rows_per_rank = rows // size
    local_data = np.empty((rows_per_rank, cols), dtype=np.float64)

    # Note the Upper Case 'S': This is the fast, buffer-optimized version.
    comm.Scatter(data, local_data, root=0)
```

■ ■ ■

```
(cee690) nc153 $ mpirun -n 4 python mpi4py_calculate_mean.py
Main: Starting collective calculation with 4 processes...
Main: Final Mean = 0.000318
Main: Total Time = 0.2020s
```

Does it speed up anything?

```
(cee690) nc153 $ mpirun -n 1 python mpi4py_calculate_mean.py
Main: Starting collective calculation with 1 processes...
Main: Final Mean = 0.000127
Main: Total Time = 1.8278s
```

```
(cee690) nc153 $ mpirun -n 2 python mpi4py_calculate_mean.py
Main: Starting collective calculation with 2 processes...
Main: Final Mean = 0.000127
Main: Total Time = 0.1308s
```

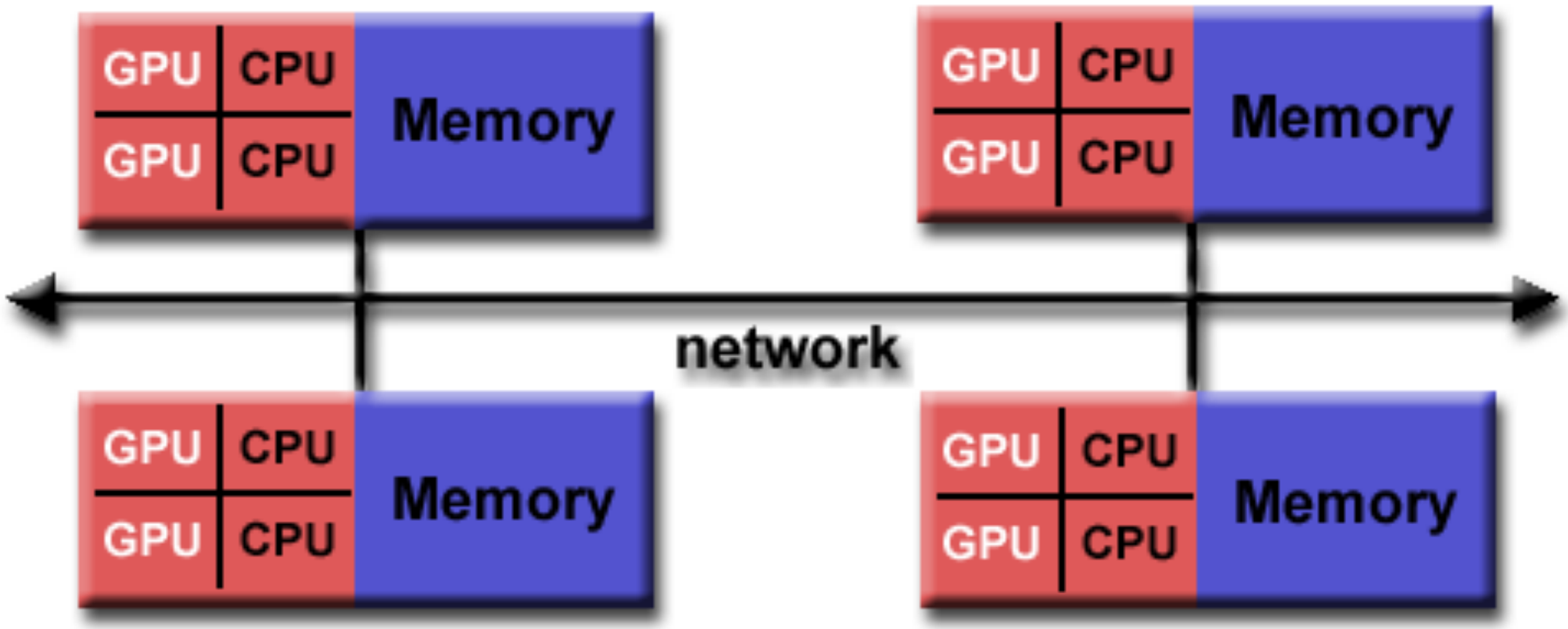
```
(cee690) nc153 $ mpirun -n 4 python mpi4py_calculate_mean.py
Main: Starting collective calculation with 4 processes...
Main: Final Mean = 0.000127
Main: Total Time = 0.2911s
```

```
(cee690) nc153 $ mpirun -n 8 python mpi4py_calculate_mean.py
Main: Starting collective calculation with 8 processes...
Main: Final Mean = 0.000127
Main: Total Time = 4.2435s
```

At these scales… not really

It takes a while to really understand MPI but if you are planning to do studies at high spatial/temporal resolutions over very large spatial/temporal extents you will eventually have no other choice.

Hybrid Parallelism



Calculate mean revisited

⋮

```python
@njit
def compute_local_chunk(chunk):
    rows, cols = chunk.shape
    local_val = 0.0
    local_count = 0

    # OPENMP: Parallelism WITHIN the node
    with openmp("parallel for reduction(+:local_val, local_count) private(j)"):
        for i in range(rows):
            for j in range(cols):
                local_val += chunk[i, j]
                local_count += 1
    return local_val, local_count

def run_hybrid_mean():
    rows, cols = 10000, 10000
    data = None

    if rank == 0:
        np.random.seed(1)
        data = np.random.randn(rows, cols)

    # 1. MPI SCATTER: Distribute large chunks to each node
    rows_per_rank = rows // size
    local_chunk = np.empty((rows_per_rank, cols), dtype=np.float64)
    comm.Scatter(data, local_chunk, root=0)

    # 2. LOCAL CALCULATION: Use OpenMP threads on the local chunk
    v, c = compute_local_chunk(local_chunk)
```

⋮