INTRODUCTION TO MPI

Dany Vohl
Astronomy Data and Computing Services, Swinburne

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

Based on contents developed by Amr H. Hassan

TABLE OF CONTENT

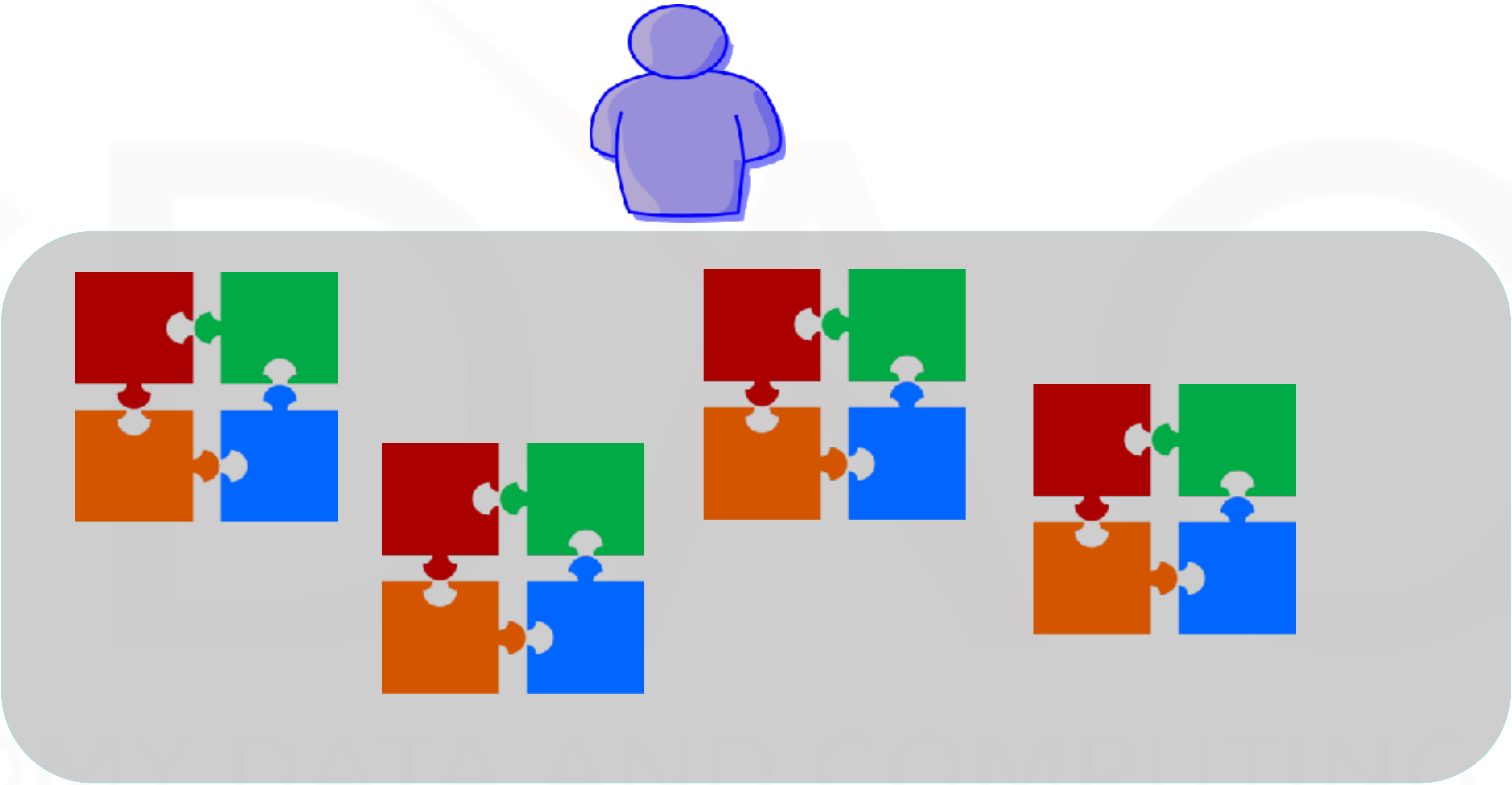
DISCLAIMER

- We chose Python over C/C++ for simplicity and to avoid the hassle of compilations and Linking.
- The code in some cases is not optimized or follow best practices.
  - *The important point is to illustrate the concept.*
- Most of these tools are actually designed and developed for C/C++.
  - *Their python version might not be complete.*

INTRODUCTION

Shared Memory VS Distributed Memory

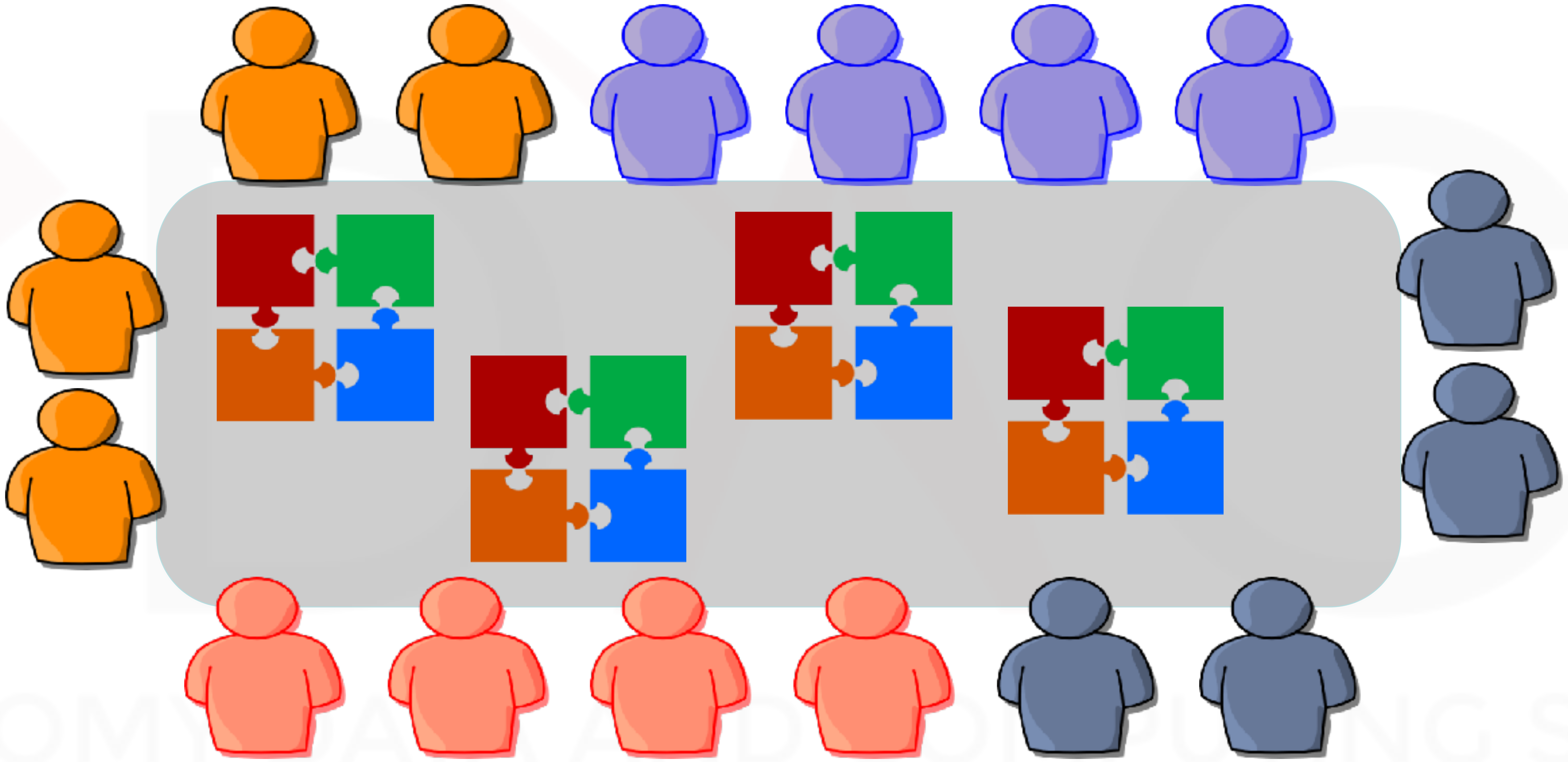
SHARED MEMORY



Modified From: Hanjun Kim, Princeton University

SHARED MEMORY VERSUS DISTRIBUTED MEMORY

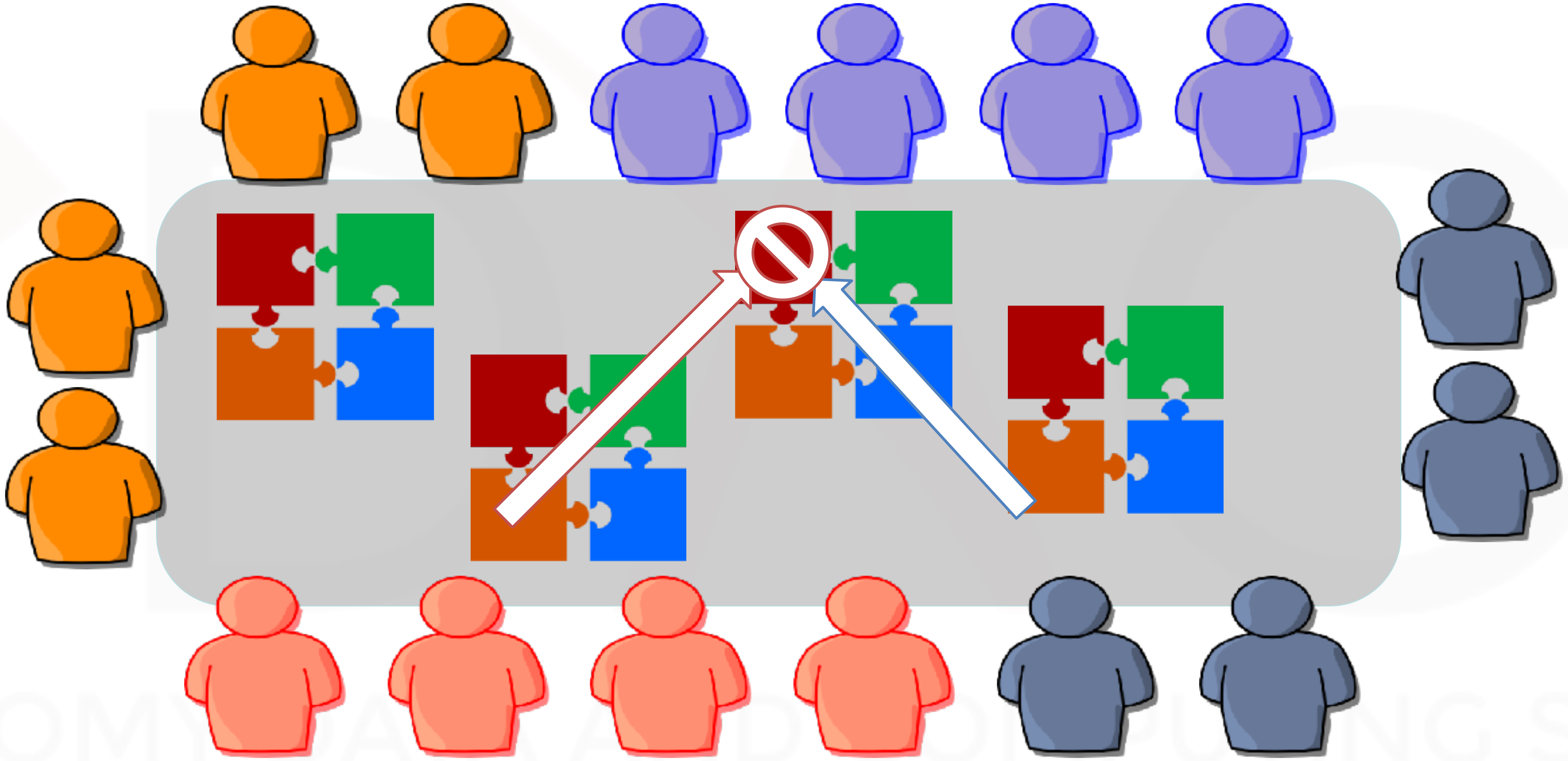
SHARED MEMORY



Modified From: Hanjun Kim, Princeton University

SHARED MEMORY VERSUS DISTRIBUTED MEMORY

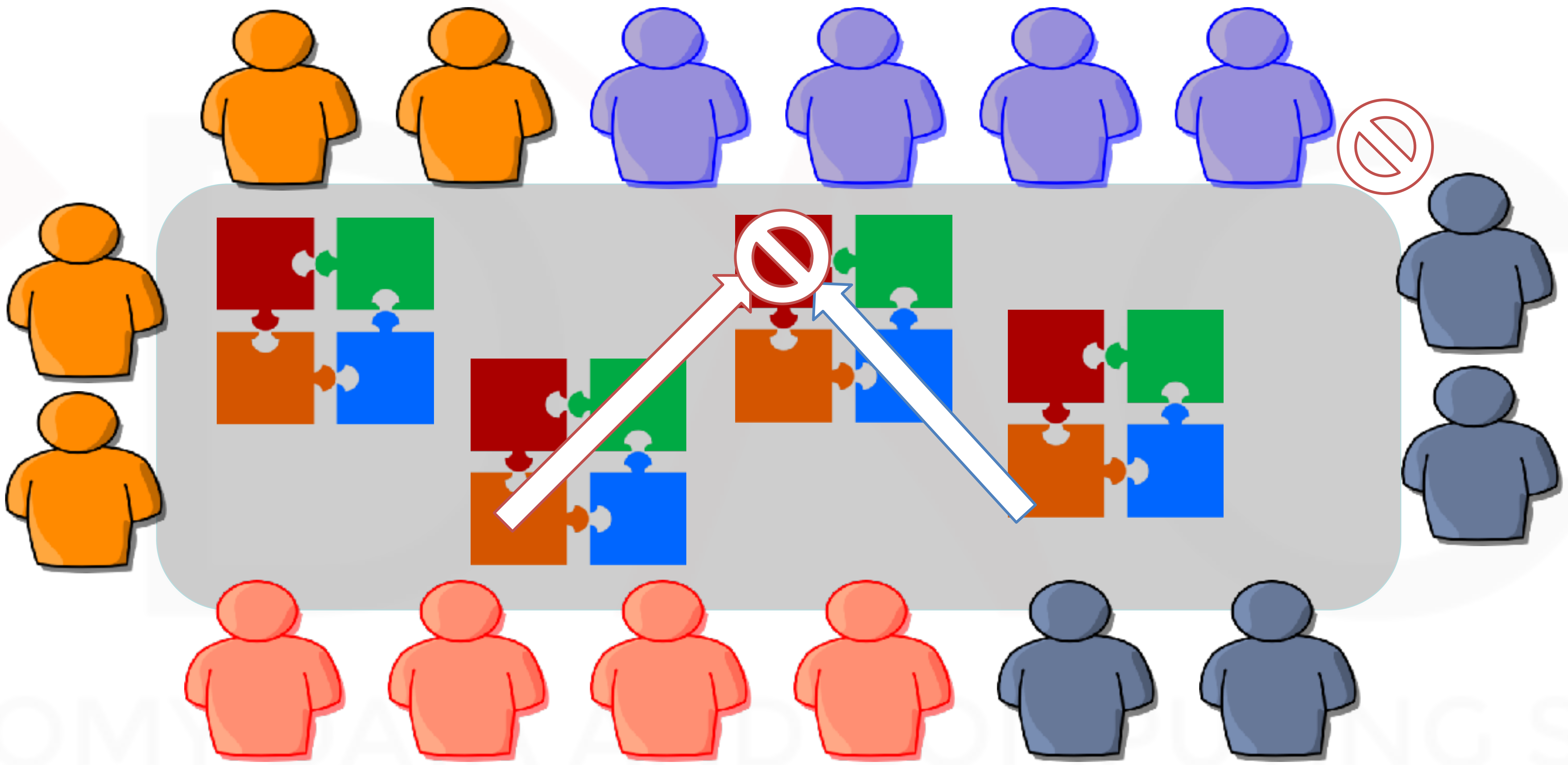
SHARED MEMORY



Modified From: Hanjun Kim, Princeton University

SHARED MEMORY VERSUS DISTRIBUTED MEMORY

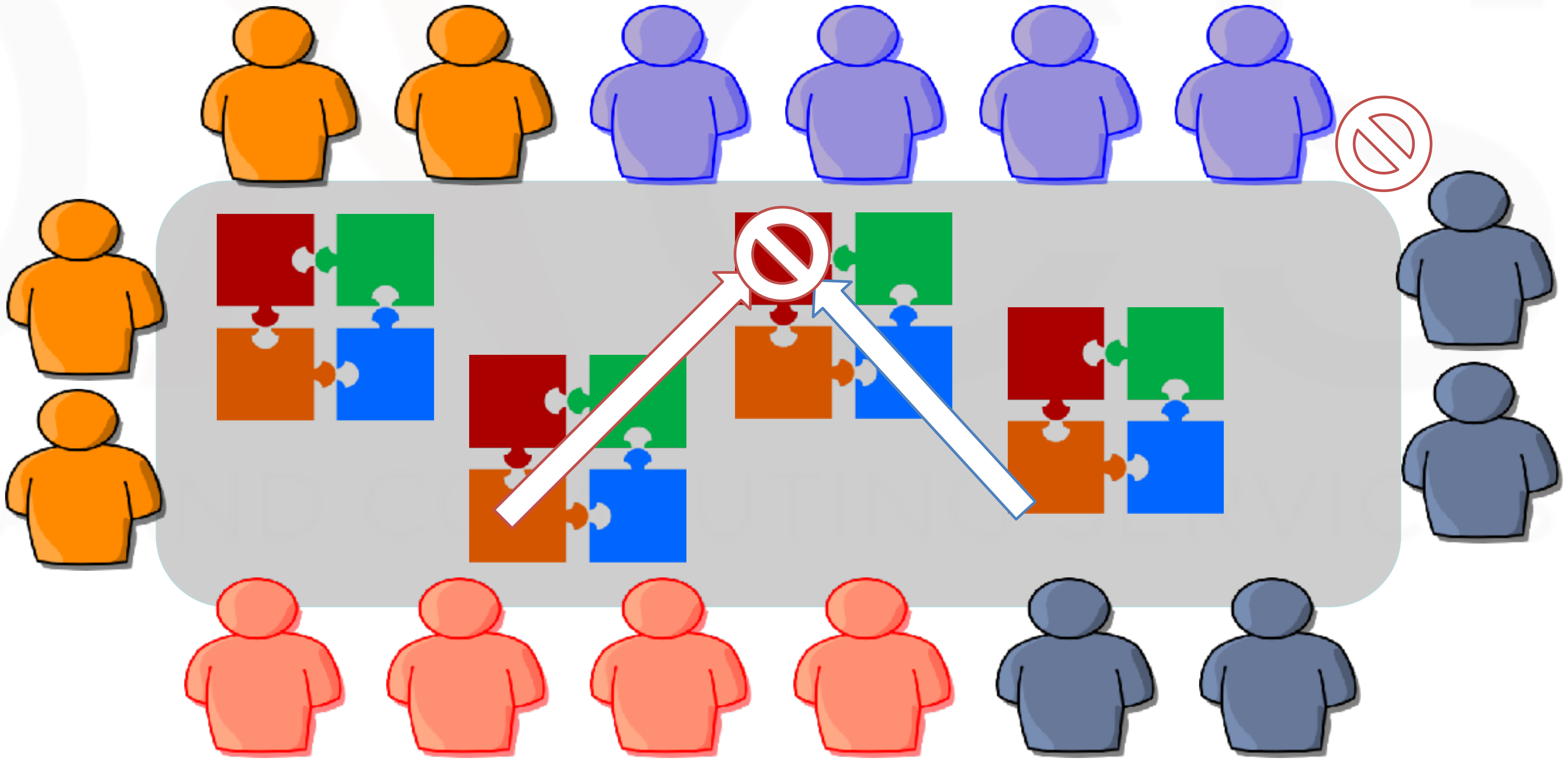
SHARED MEMORY



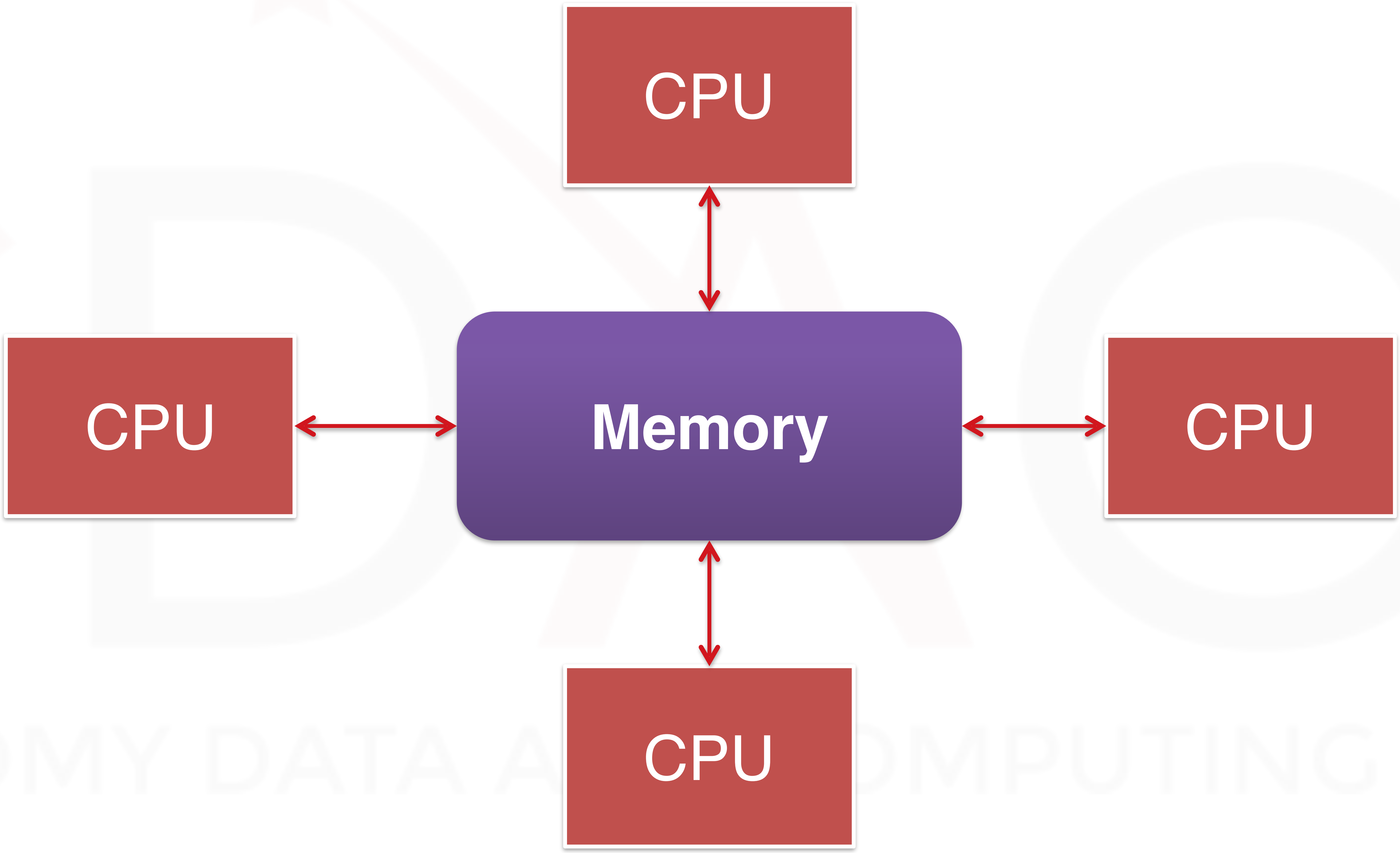
Modified From: Hanjun Kim, Princeton University

SHARED MEMORY

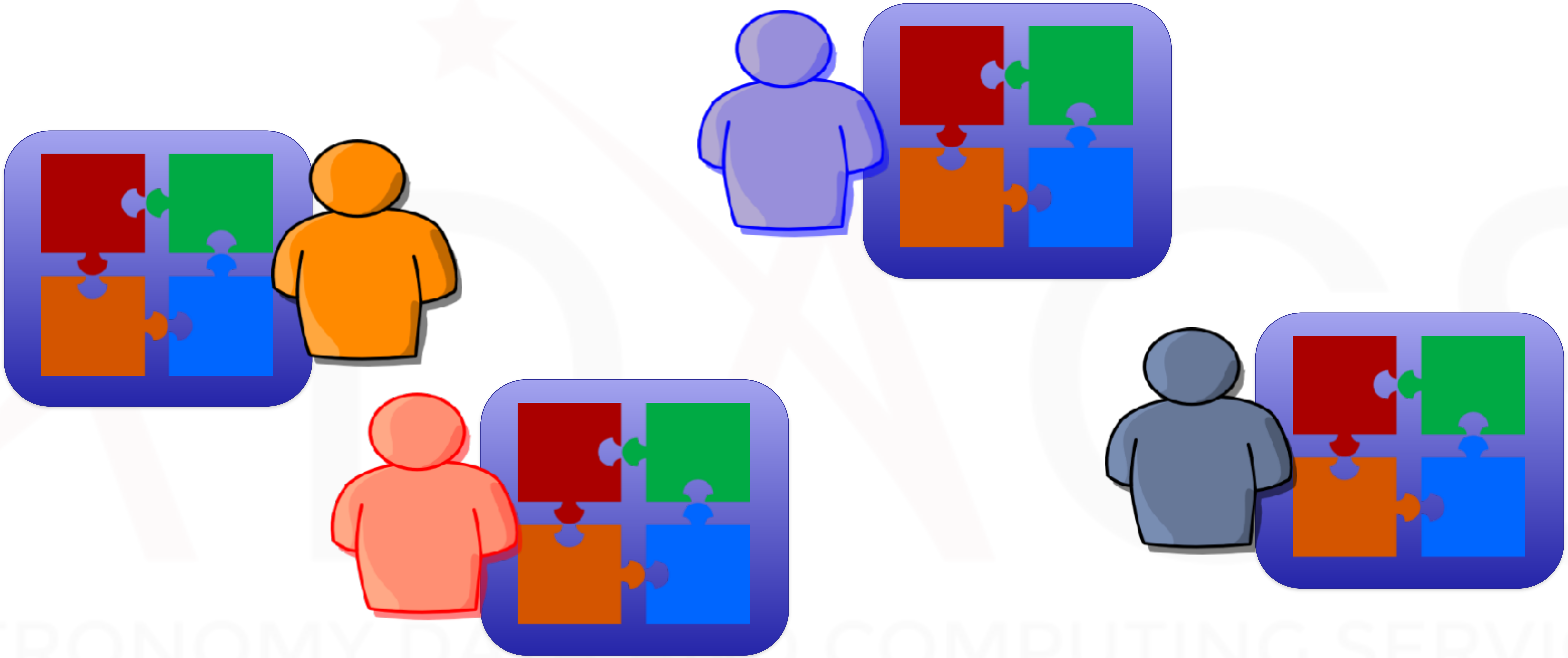
- Splitting the problem among many people in shared memory
  - *Pros - Easy to use*
  - *Cons - Limited Scalability, High coherence overhead*



SHARED MEMORY



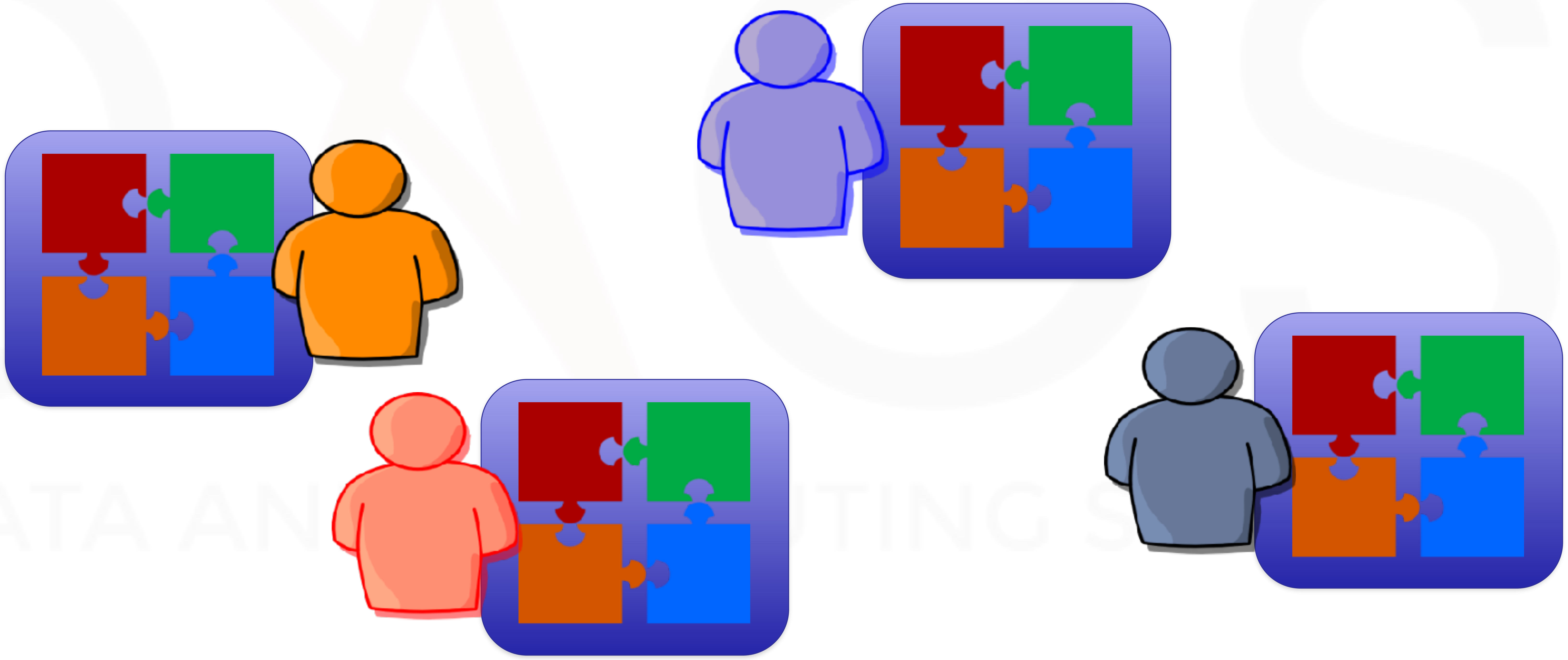
DISTRIBUTED MEMORY



Modified From: Hanjun Kim, Princeton University

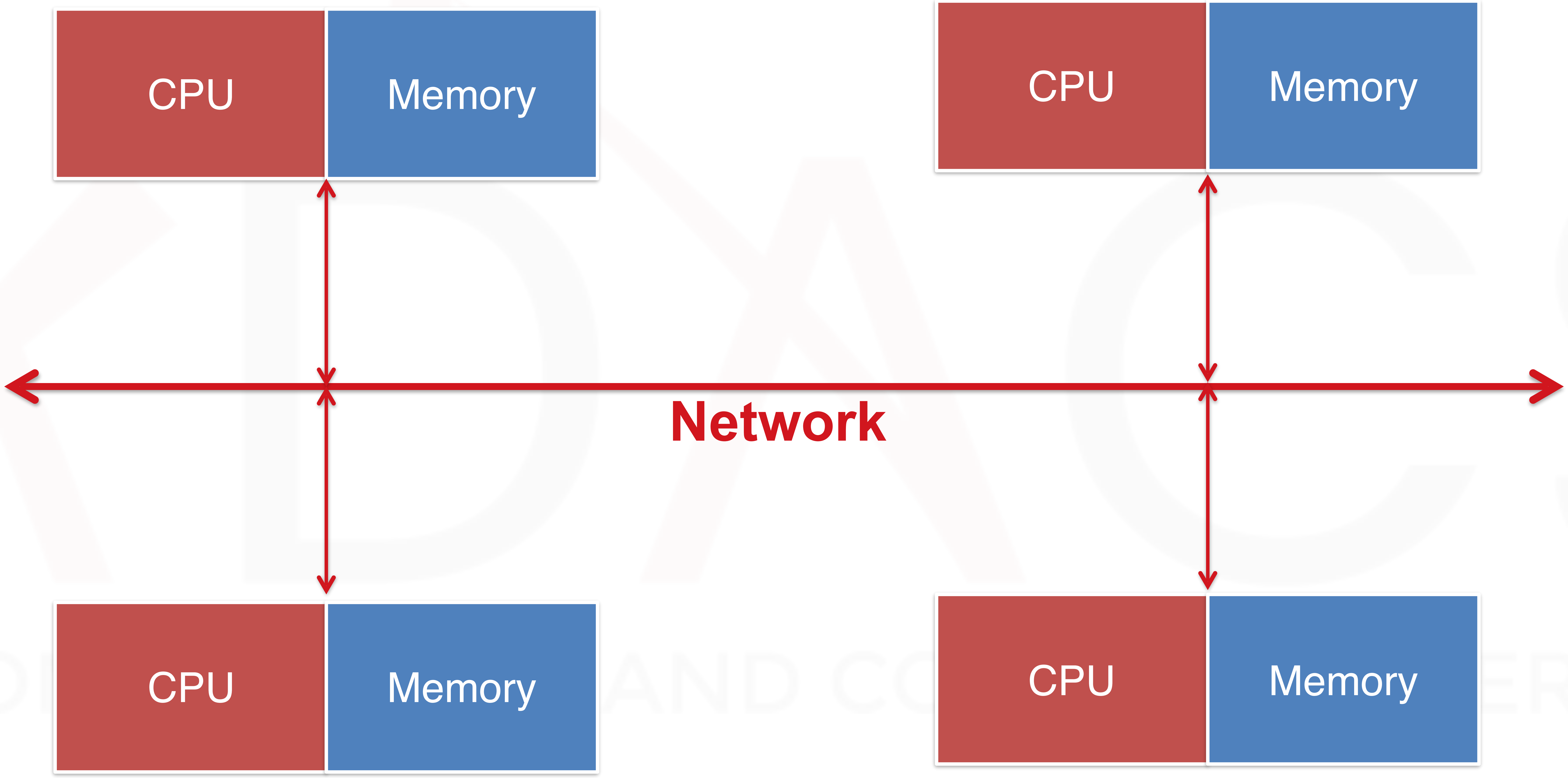
DISTRIBUTED MEMORY

- Splitting the problem among many people distributed memory
  - *Scalable seats (Scalable resources)*
  - *Less contention from private memory spaces*



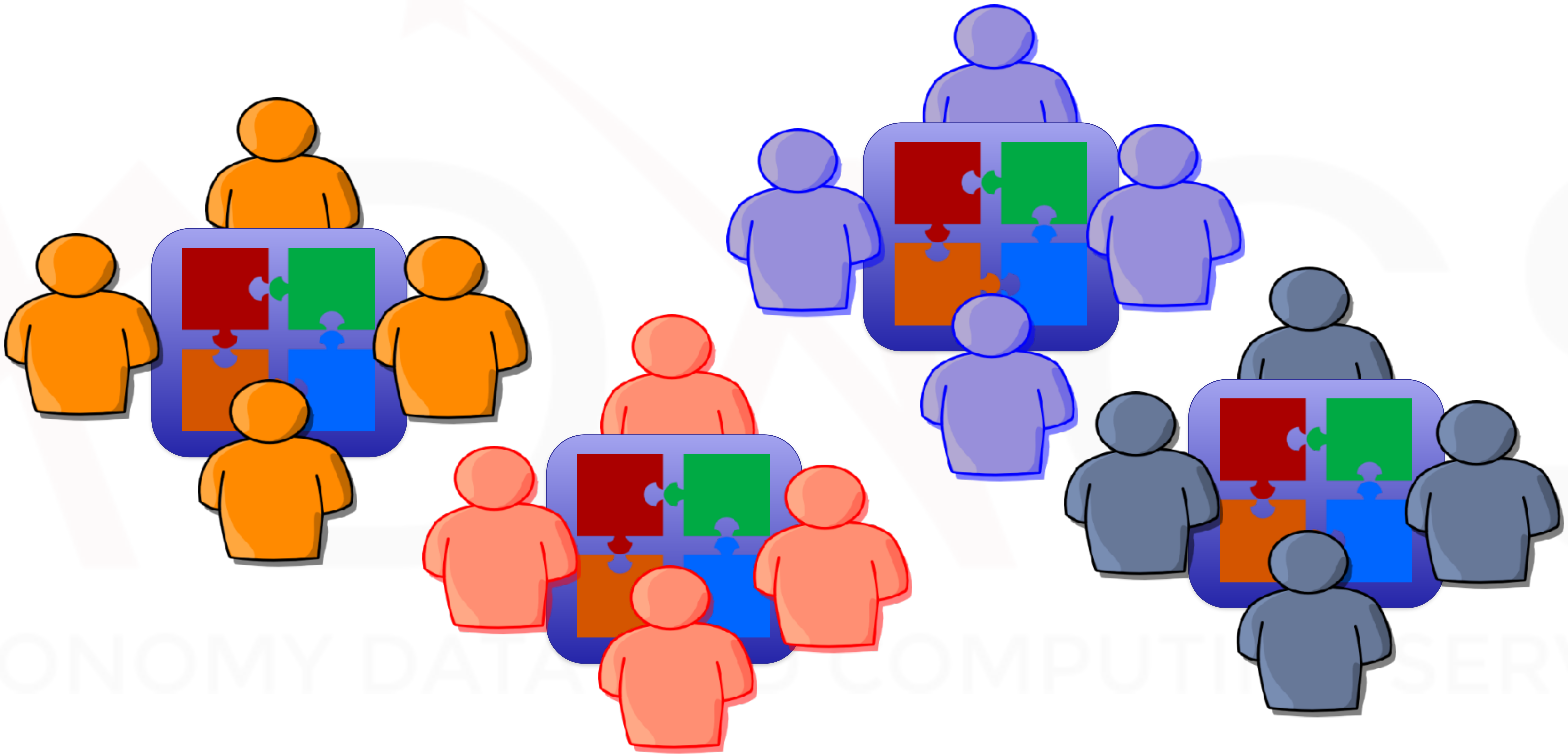
Modified From: Hanjun Kim, Princeton University

DISTRIBUTED MEMORY



SHARED MEMORY VERSUS DISTRIBUTED MEMORY

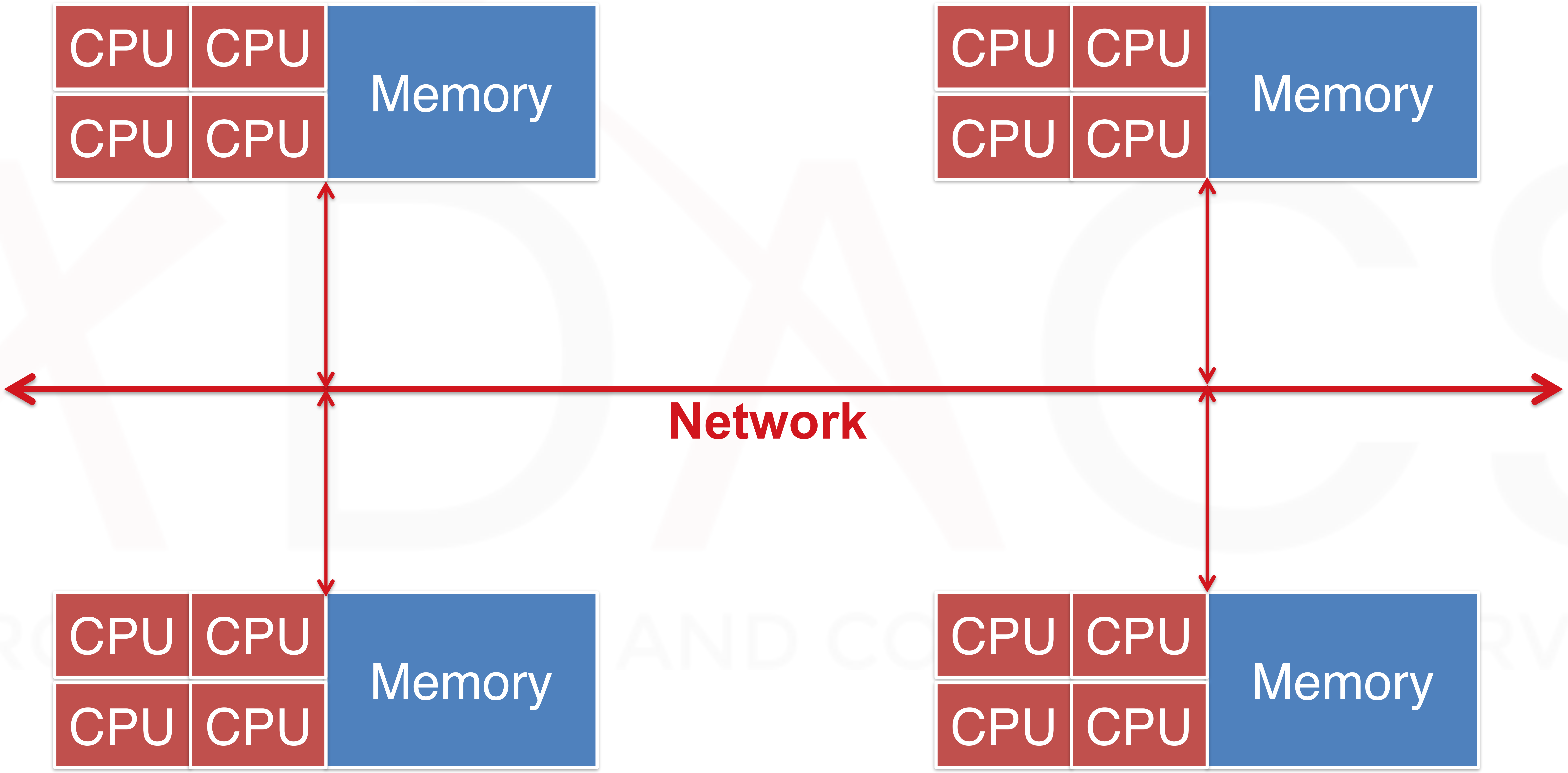
MIXED VERSION



Modified From: Hanjun Kim, Princeton University

SHARED MEMORY VERSUS DISTRIBUTED MEMORY

MIXED VERSION



INTRODUCTION

Message Passing Interface

WHAT IS MPI, AND WHY USE IT?

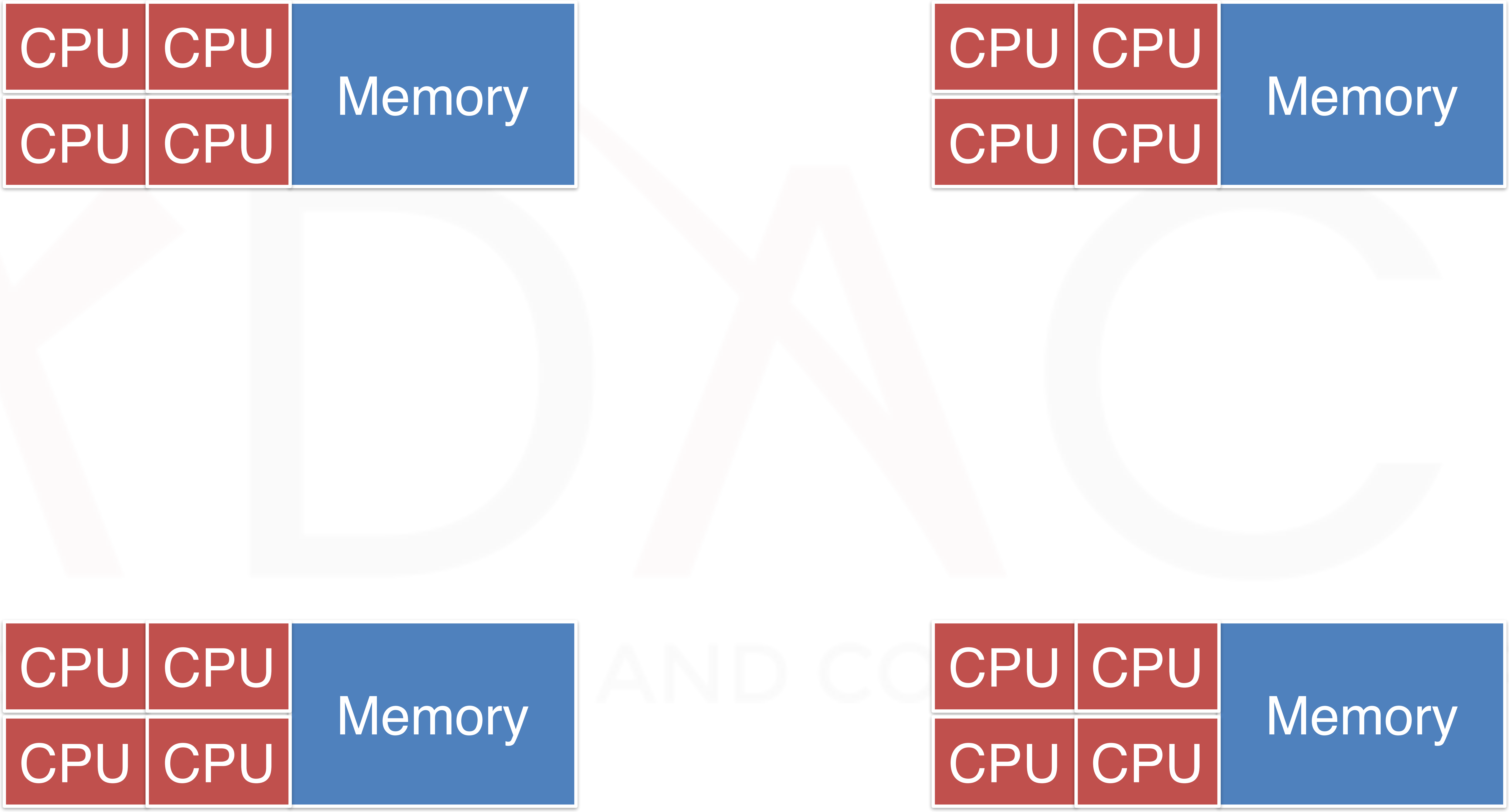
"The Message Passing Interface Standard (MPI) is a message passing library standard based on the consensus of the MPI Forum, which has over 40 participating organizations, including vendors, researchers, software library developers, and users. ***The goal of [MPI] is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs****. (…) The advantages of developing message passing software using MPI closely match the design goals of portability, efficiency, and flexibility.* ***MPI is not an IEEE or ISO standard, but has in fact, become the "industry standard" for writing message passing programs on HPC platforms.****"*

– https://computing.llnl.gov/tutorials/mpi/

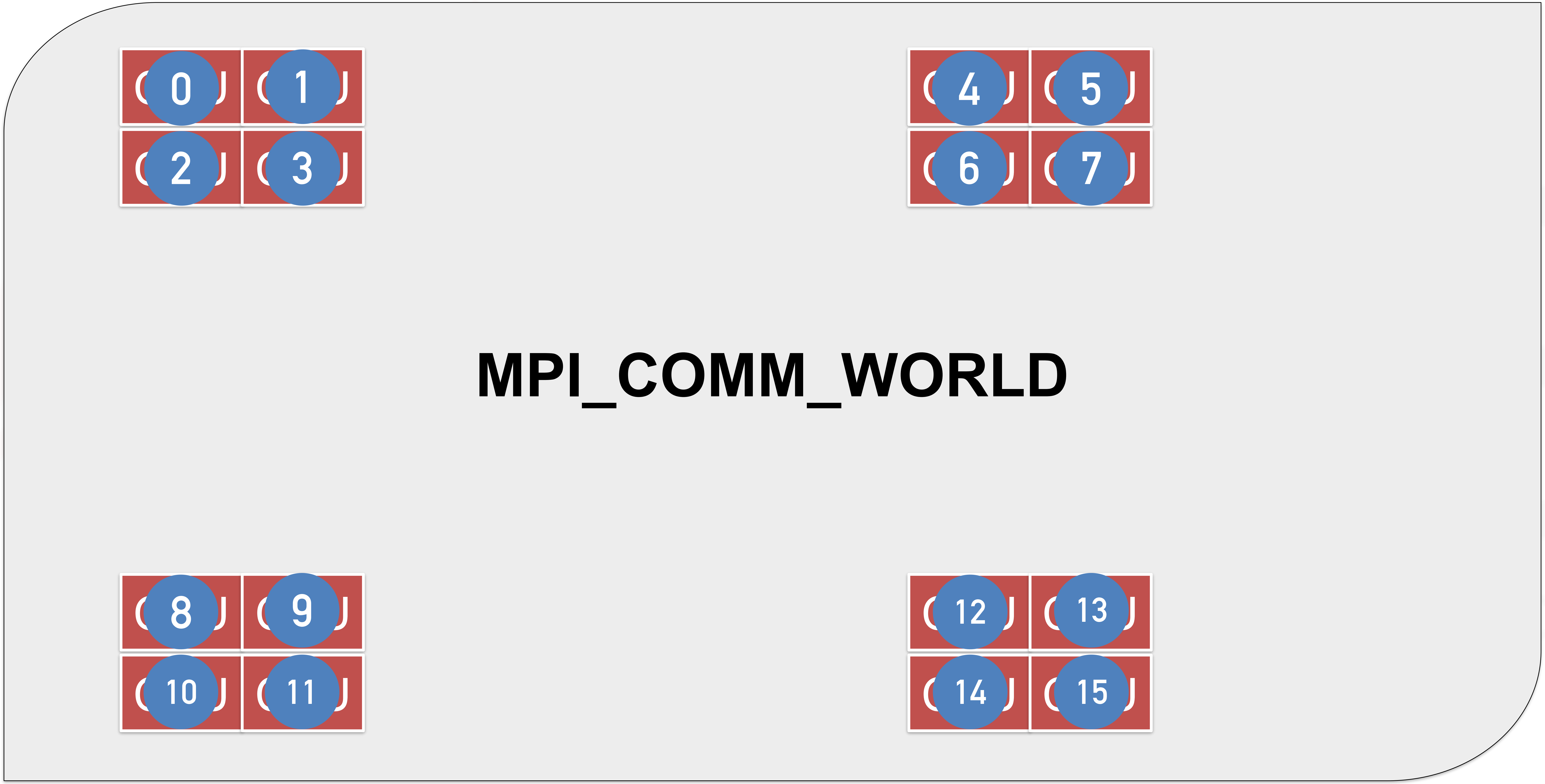
WHAT IS MPI, AND WHY USE IT?

- De facto Standard Framework for Distributed computing.
- Simple communication model between processes in a program.
- Multiple implementations; highly efficient for different platforms.
- Well established community (since 1994).

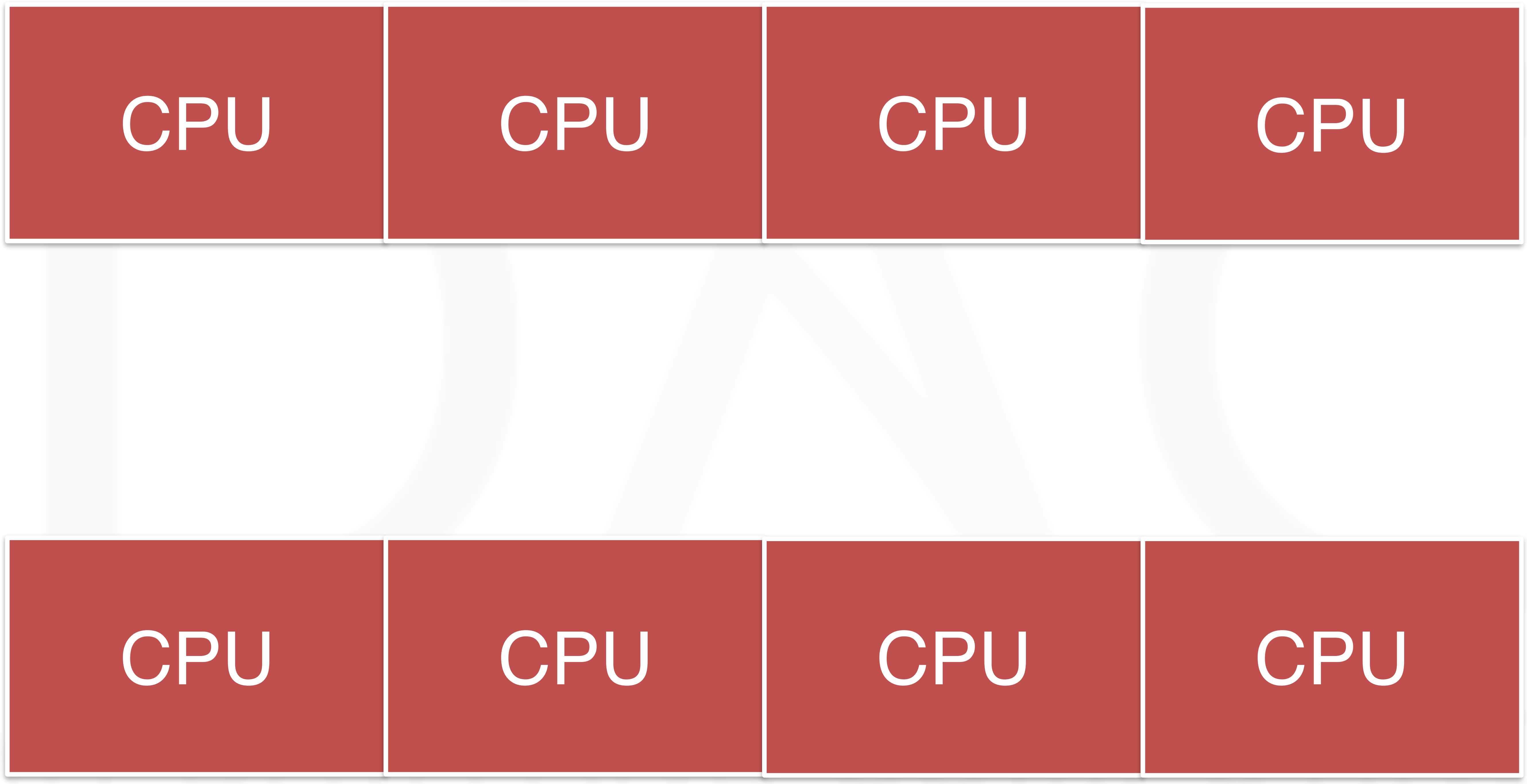
MESSAGE PASSING INTERFACE



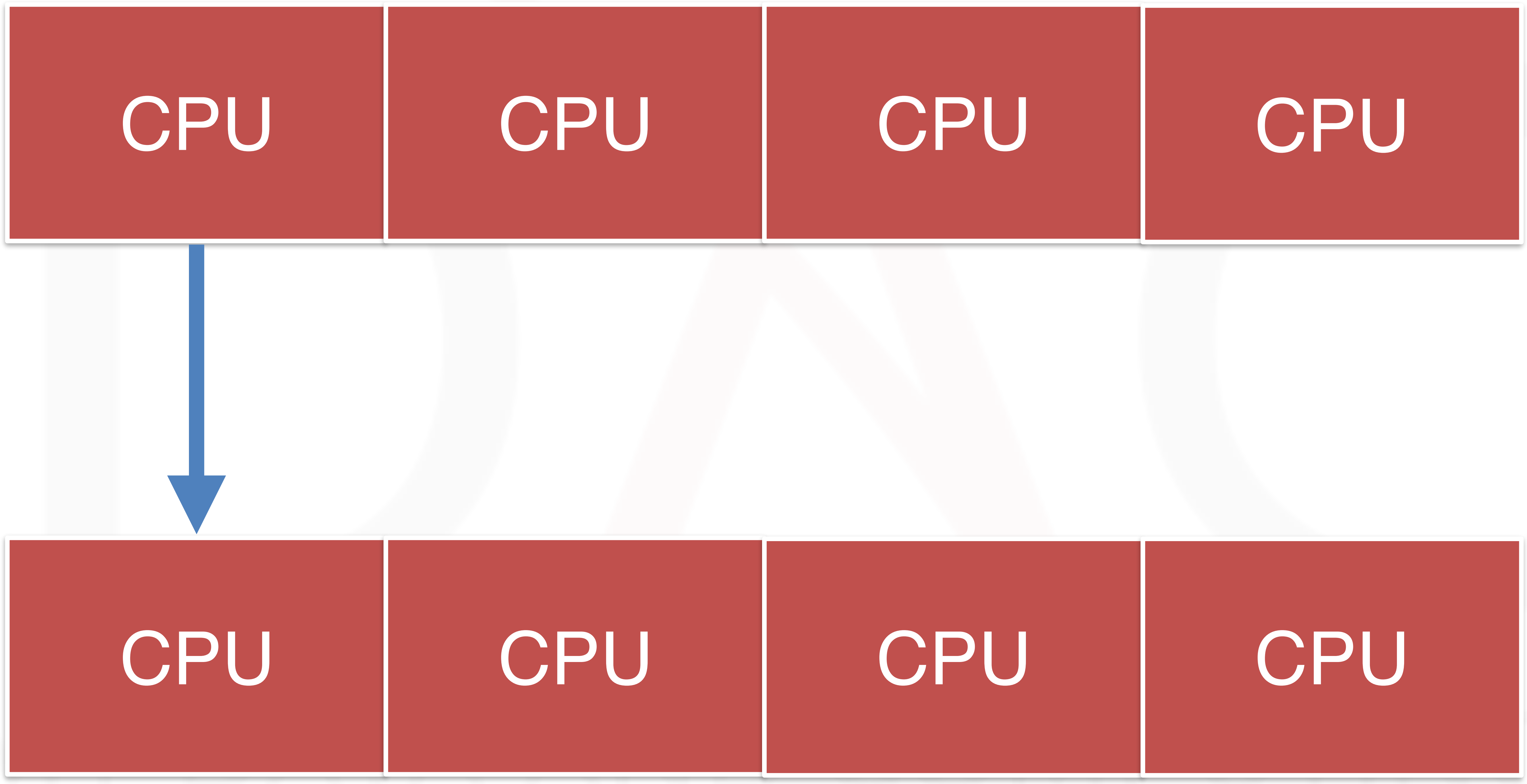
MESSAGE PASSING INTERFACE



MESSAGE PASSING INTERFACE

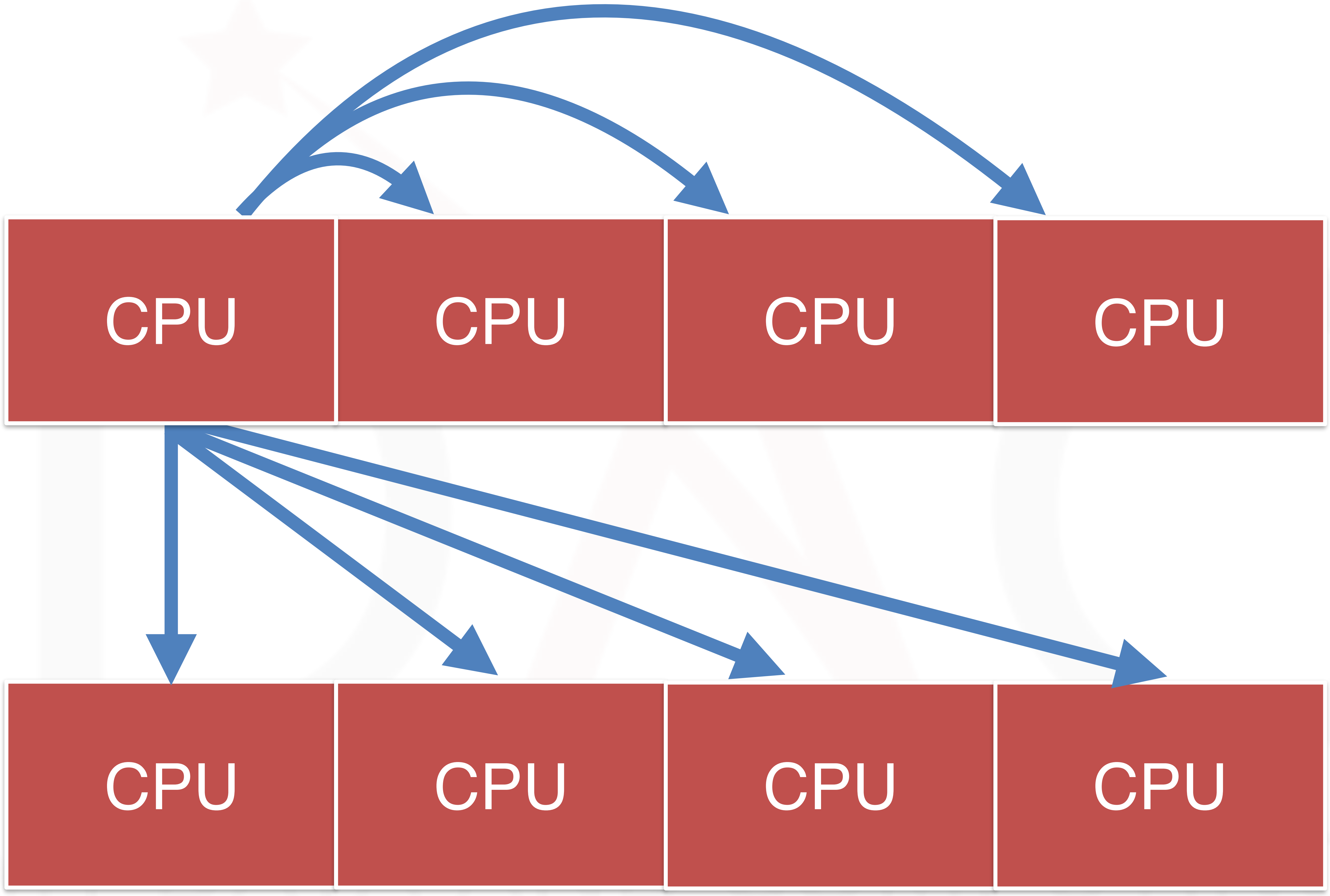


Point-to-Point Communication



MESSAGE PASSING INTERFACE

Collective Communication



WORKING EXAMPLES WITH MPI4PY

Message Passing Interface

MPI FOR PYTHON

- Python package : mpi4py
- mpi4py supports:
  - (1) Pickle-based communication of generic Python object
    - *For convenience*
  - (2) Direct array data communication of buffer-provider objects
    - *Faster option (near C-speed)*
      - (e.g., NumPy arrays).

MPI FOR PYTHON

- Important Class "**Comm**"

1. Communication for Generic python objects
   - *Use "lower case" methods: **send()**, **receive()***
2. Communication for buffer-provider objects (e.g numpy arrays)
   - *Use "upper case" methods: **Send()**, **Receive()***

MPI FOR PYTHON

- Importing library:

```
from mpi4py import MPI
```

- Getting important information:

```
comm = MPI.COMM_WORLD

rank = MPI.COMM_WORLD.Get_rank()

size = MPI.COMM_WORLD.Get_size()

name = MPI.Get_processor_name()
```

HELLO, WORLD

```
from mpi4py import MPI
comm=MPI.COMM_WORLD
print ("Hello! I'm rank %d out of %d processes running in total ..."%(comm.rank,comm.size))
comm.Barrier()
```

```
>> mpirun -np 4 python hello-world.py
```

HELLO, WORLD

```
from mpi4py import MPI
comm=MPI.COMM_WORLD
print ("Hello! I'm rank %d out of %d processes running in
total … running on %s”%(comm.rank, comm.size,
MPI.Get_processor_name()))
comm.Barrier()
```

```
>> qsub -q sstar -I -l walltime=0:5:0,nodes=2,ppn=14,
   mem=200MB

>> mpirun -np 28 python hello-world-2.py
```

WORKING EXAMPLES WITH MPI4PY

Message Passing Interface (Point-to-Point Communication)

SENDING AND RECEIVING DATA (POINT-TO-POINT COMM.)

- Python objects (pickle under the hood)
  - *An object to be sent is passed as a parameter to the communication call, and the received object is simply the return value.*

SENDING AND RECEIVING DATA (POINT-TO-POINT COMM.)

- Python objects (pickle under the hood)

```
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)
```

SENDING AND RECEIVING DATA (POINT-TO-POINT COMM.)

- Python objects with non-blocking communication
  - *The **isend()** and **irecv()** methods return Request instances*
  - *Completion of these methods is managed via the **test()** and **wait()** methods of the Request class.*

SENDING AND RECEIVING DATA (POINT-TO-POINT COMM.)

- Python objects with non-blocking communication

```
from mpi4py import MPI

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

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    req = comm.isend(data, dest=1, tag=11)
    req.wait()
elif rank == 1:
    req = comm.irecv(source=0, tag=11)
    data = req.wait()
```

MPI FOR PYTHON SENDING AND RECEIVING DATA (POINT-TO-POINT COMM.)

- Buffer-provider objects (NumPy arrays; the fast way)
  - *Buffer arguments to these calls must be explicitly specified*
  - *Use a 2/3-list/tuple*
    - e.g. [data, MPI.DOUBLE]
      - uses the byte-size of data and the extent of the MPI datatype to define the count
    - or [data, count, MPI.DOUBLE]

SENDING AND RECEIVING DATA (POINT-TO-POINT COMM.)

- Buffer-provider objects (NumPy arrays; the fast way)

```
from mpi4py import MPI
import numpy

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

# passing MPI datatypes explicitly
if rank == 0:
    data = numpy.arange(1000, dtype='i')
    comm.Send([data, MPI.INT], dest=1, tag=77)
elif rank == 1:
    data = numpy.empty(1000, dtype='i')
    comm.Recv([data, MPI.INT], source=0, tag=77)

(…)
```

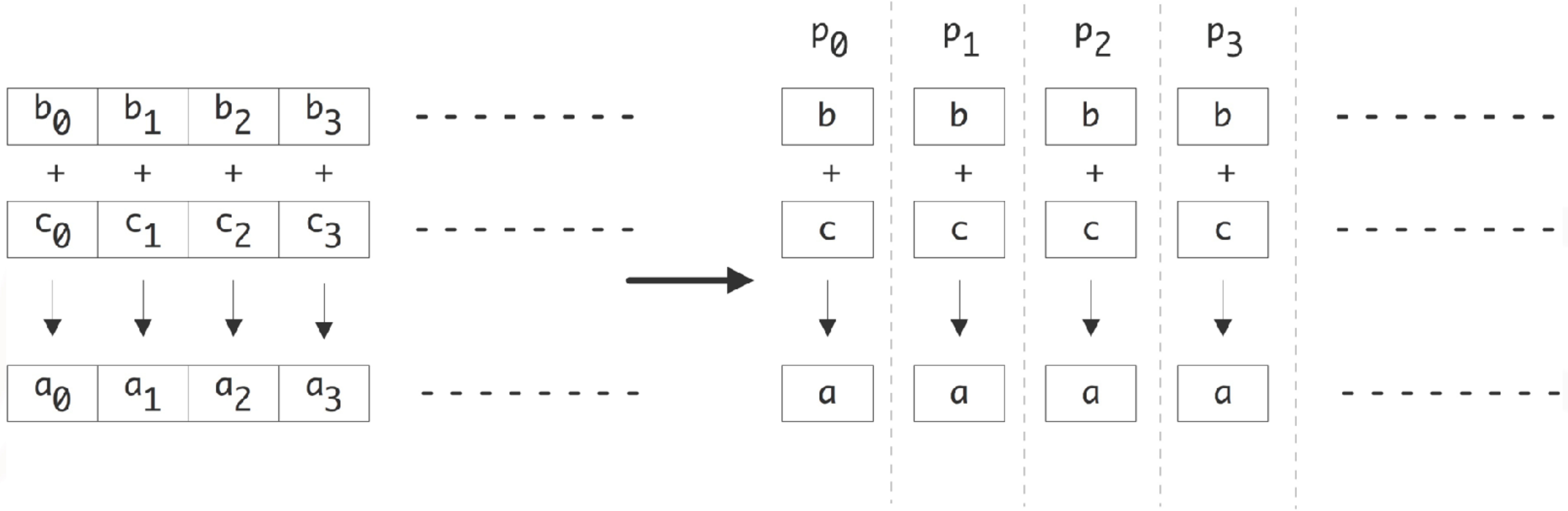
SENDING AND RECEIVING DATA (POINT-TO-POINT COMM.)

- Buffer-provider objects (NumPy arrays; the fast way)

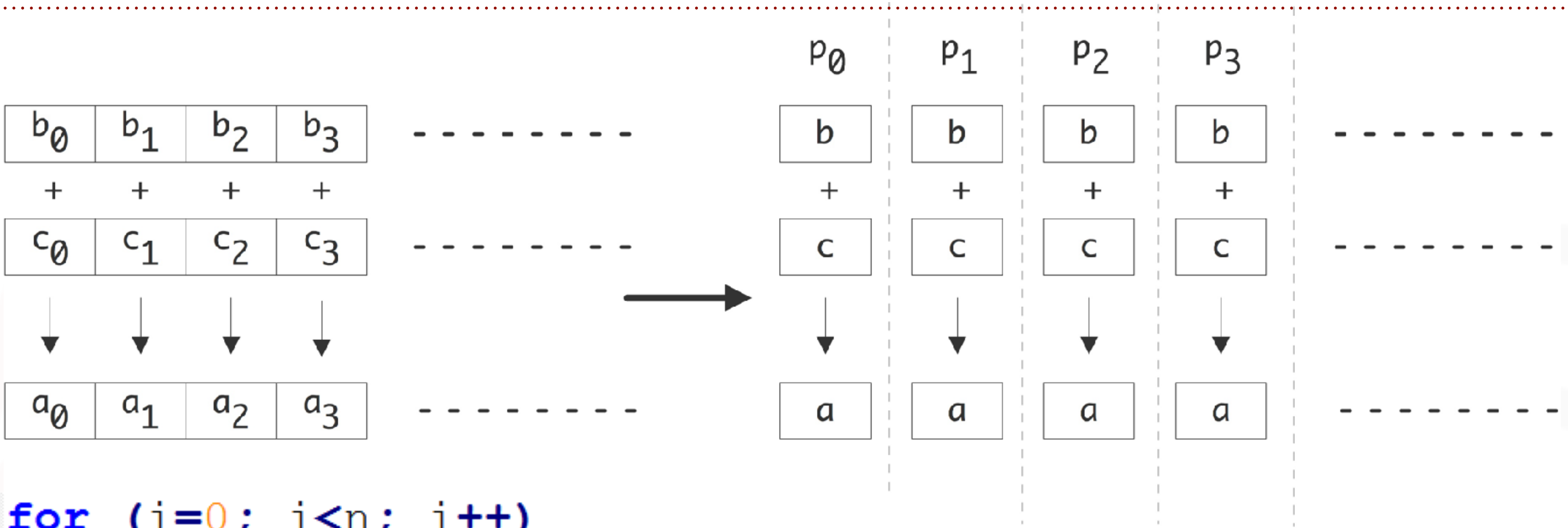
```
(…)

# automatic MPI datatype discovery
if rank == 0:
    data = numpy.arange(100, dtype=numpy.float64)
    comm.Send(data, dest=1, tag=13)
elif rank == 1:
    data = numpy.empty(100, dtype=numpy.float64)
    comm.Recv(data, source=0, tag=13)
```

SUM TWO VECTORS



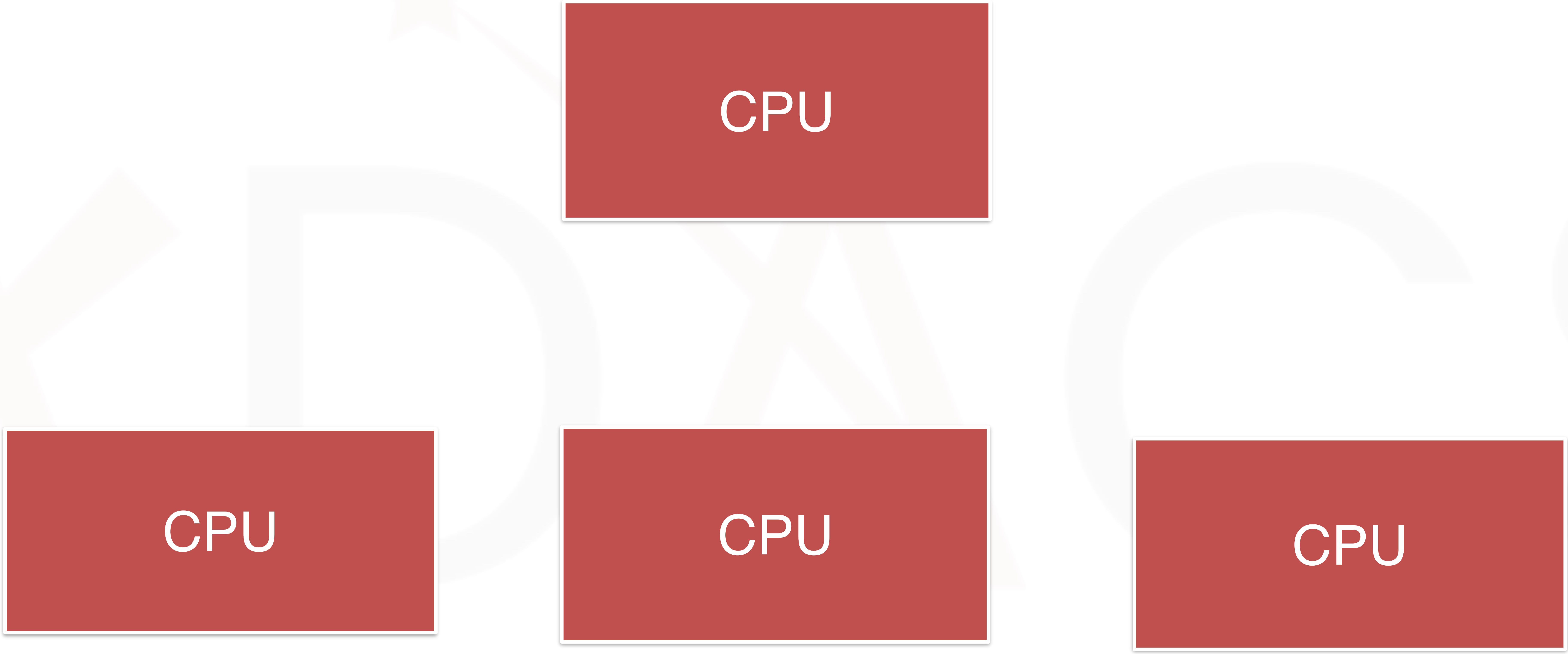
SUM TWO VECTORS




```
for (i=0; i<n; i++)
    a[i] = b[i] + c[i];
```

```
for (i=my_low; i<my_high; i++)
    a[i] = b[i] + c[i];
```

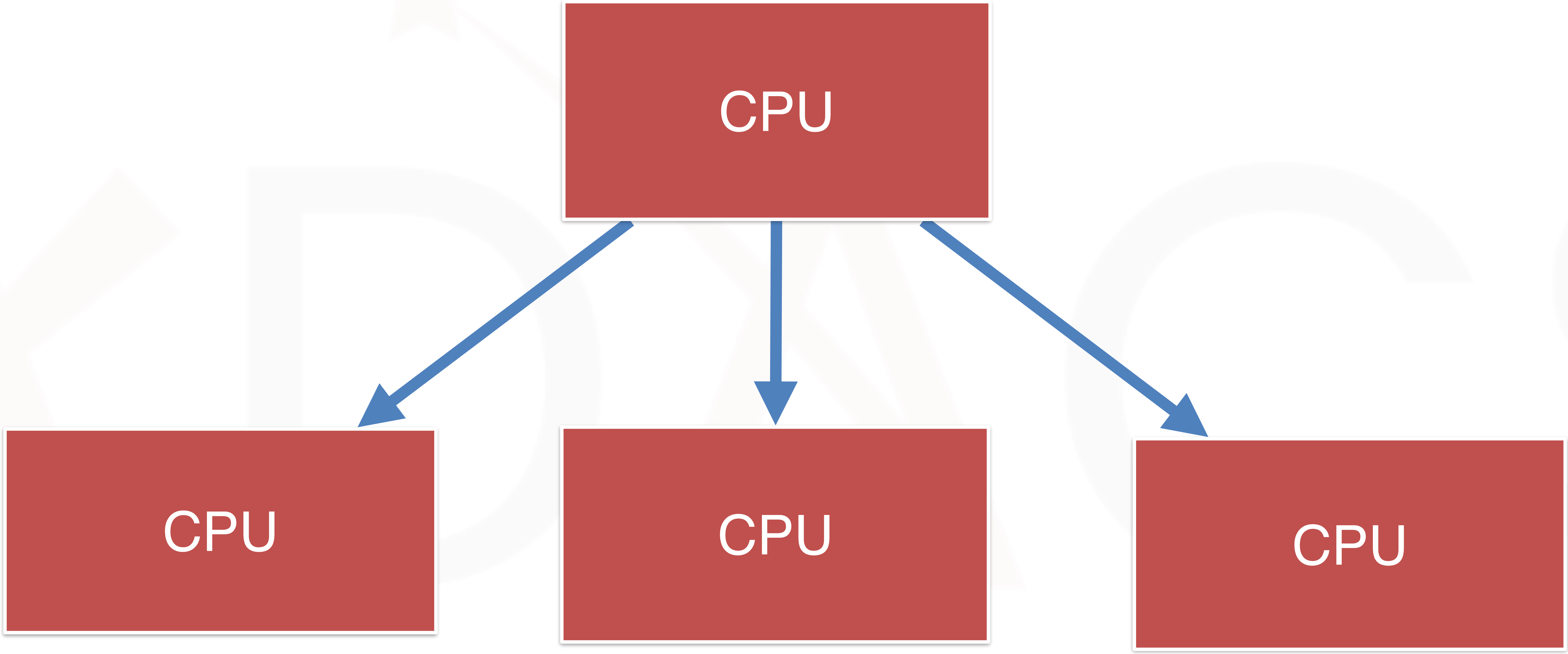
SUM TWO VECTORS



Client-Server Communication

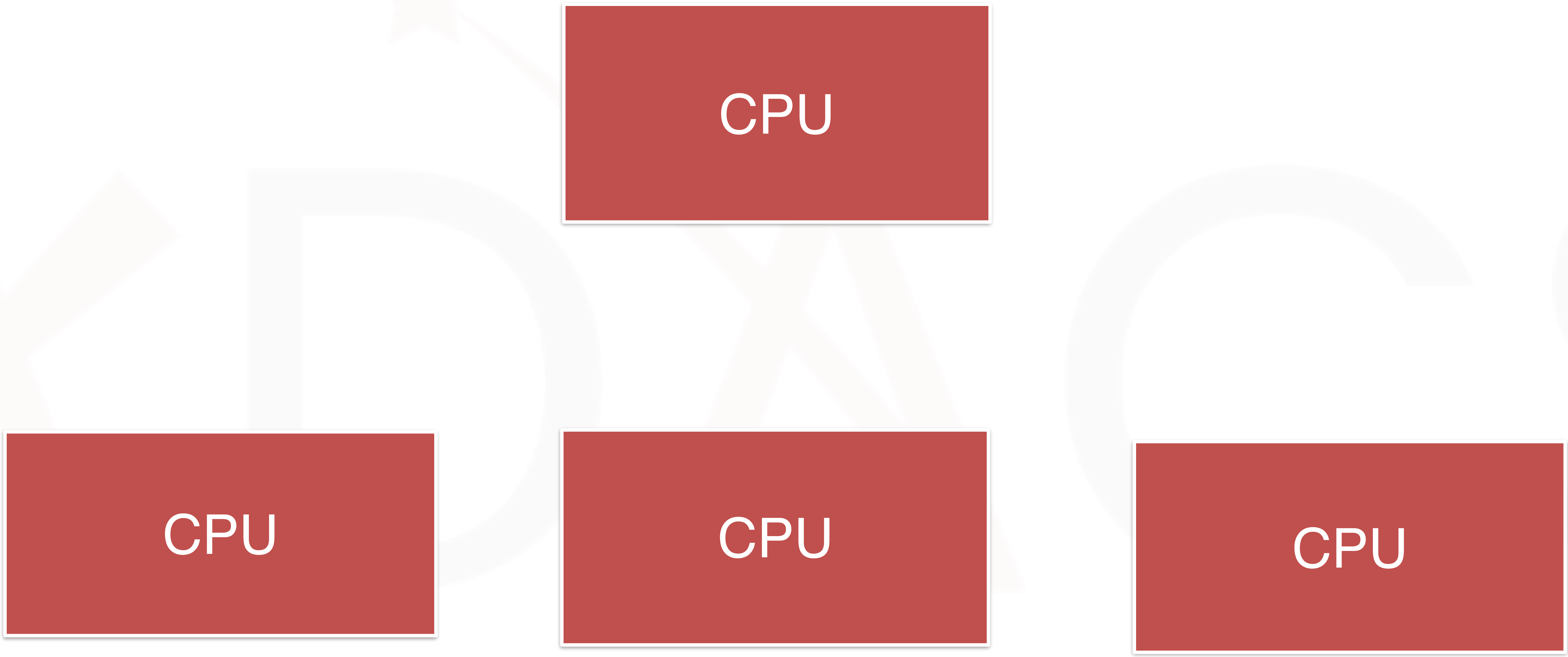
MPI FOR PYTHON

SUM TWO VECTORS



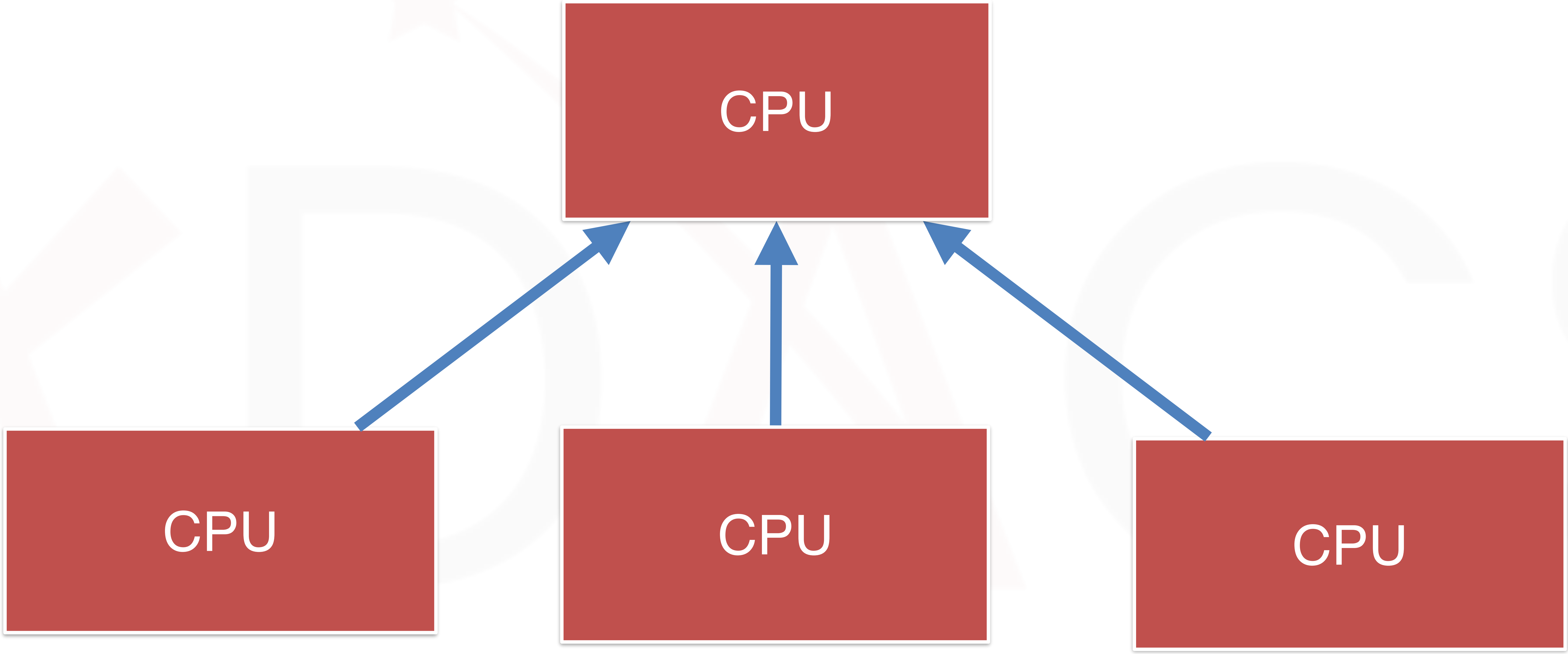
Client-Server Communication

SUM TWO VECTORS



Client-Server Communication

SUM TWO VECTORS



Client-Server Communication

SUM TWO VECTORS

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
Count = 999
PCount = Count / (size - 1)

(…)
```

SUM TWO VECTORS

```
(…)
# pass explicit MPI datatypes
if rank == 0:
    Adata = numpy.arange(0, Count, 1, dtype='i')
    Bdata = numpy.arange(Count, 0, -1, dtype='i')
    Cdata = numpy.empty(Count, dtype='i')
    for p in range(1, size):
        Start = PCount * (p - 1)
        End = PCount * p
        print ("%d:[%d to %d]" % (p, Start, End))

        Dim = [Start, End]
        comm.send(Dim, dest=p, tag=1)
        comm.Send(Adata[Start:End], dest=p, tag=2)
        comm.Send(Bdata[Start:End], dest=p, tag=2)

    for p in range(1, size):
        Start = PCount * (p - 1)
        End = PCount * p
        comm.Recv(Cdata[Start:End], source=p, tag=3)
    print (Cdata)
(…)
```

SUM TWO VECTORS

```
(…)
else:

    Dim = comm.recv(source=0, tag=1)
    print ("Rank %d: Recived[%d to %d]" % (rank, Dim[0], Dim[1]))

    Adata = numpy.empty(Dim[1] - Dim[0], dtype='i')
    Bdata = numpy.empty(Dim[1] - Dim[0], dtype='i')

    comm.Recv(Adata, source=0, tag=2)
    comm.Recv(Bdata, source=0, tag=2)

    Cdata = numpy.add(Adata, Bdata)

    comm.Send(Cdata, dest=0, tag=3)
```

ACCEPTING ANY MESSAGE FROM ANY SOURCE

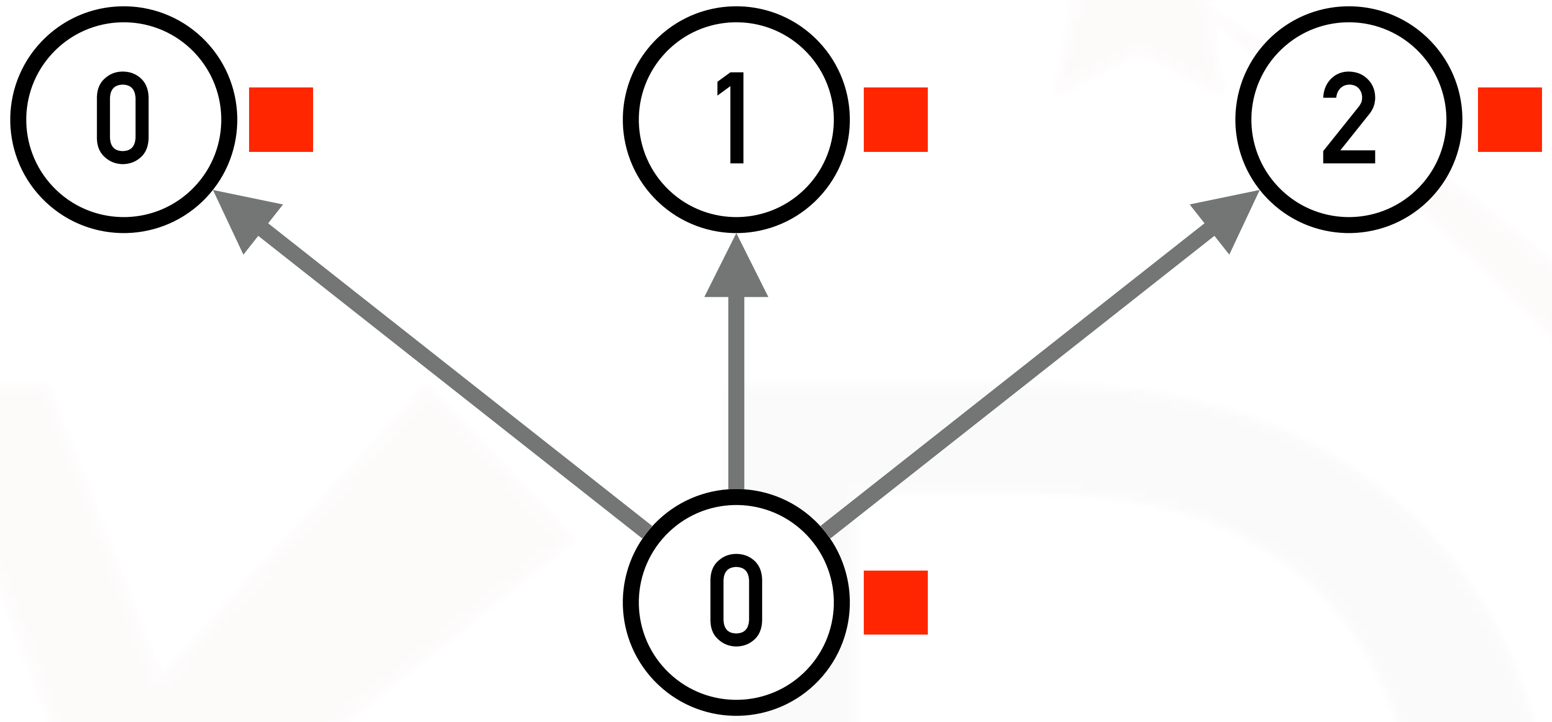
```
status=MPI.status()
data=comm.recv(source=MPI.ANY_SOURCE,
               tag=MPI.ANY_TAG,
               status=status)
source=status.Get_source()
tag=status.Get_tag()
```

WORKING EXAMPLES WITH MPI4PY

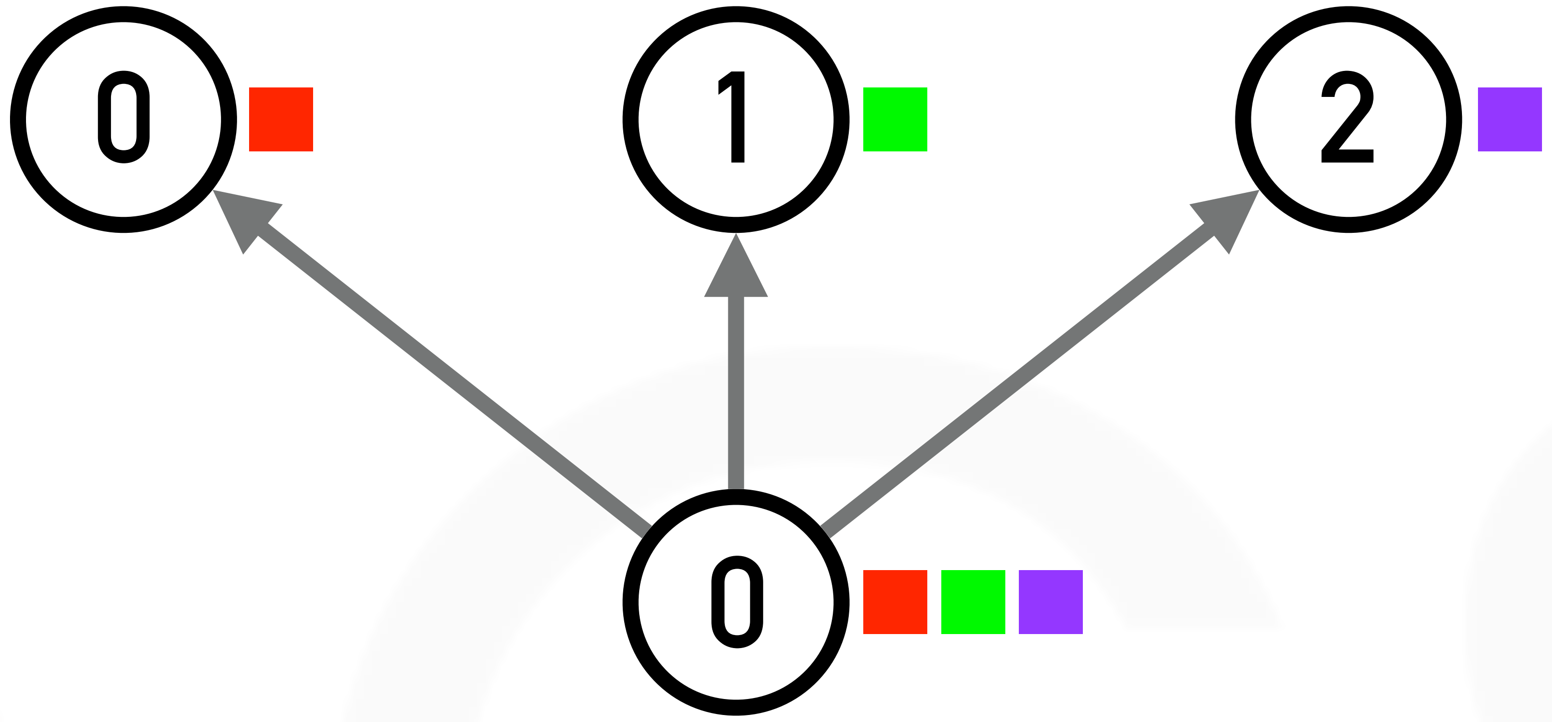
Message Passing Interface (Collective Communication)

SENDING AND RECEIVING DATA (COLLECTIVE COMM.)

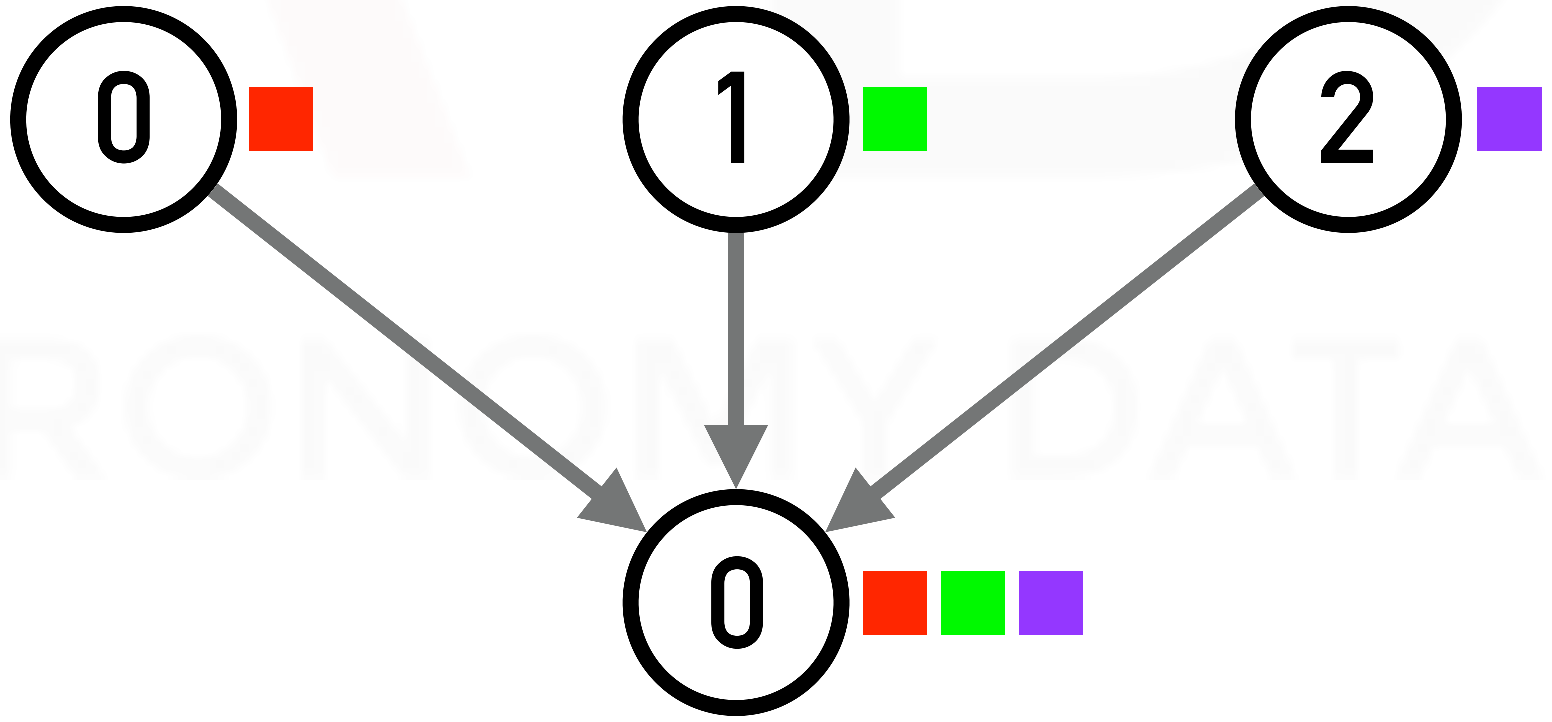
Broadcast



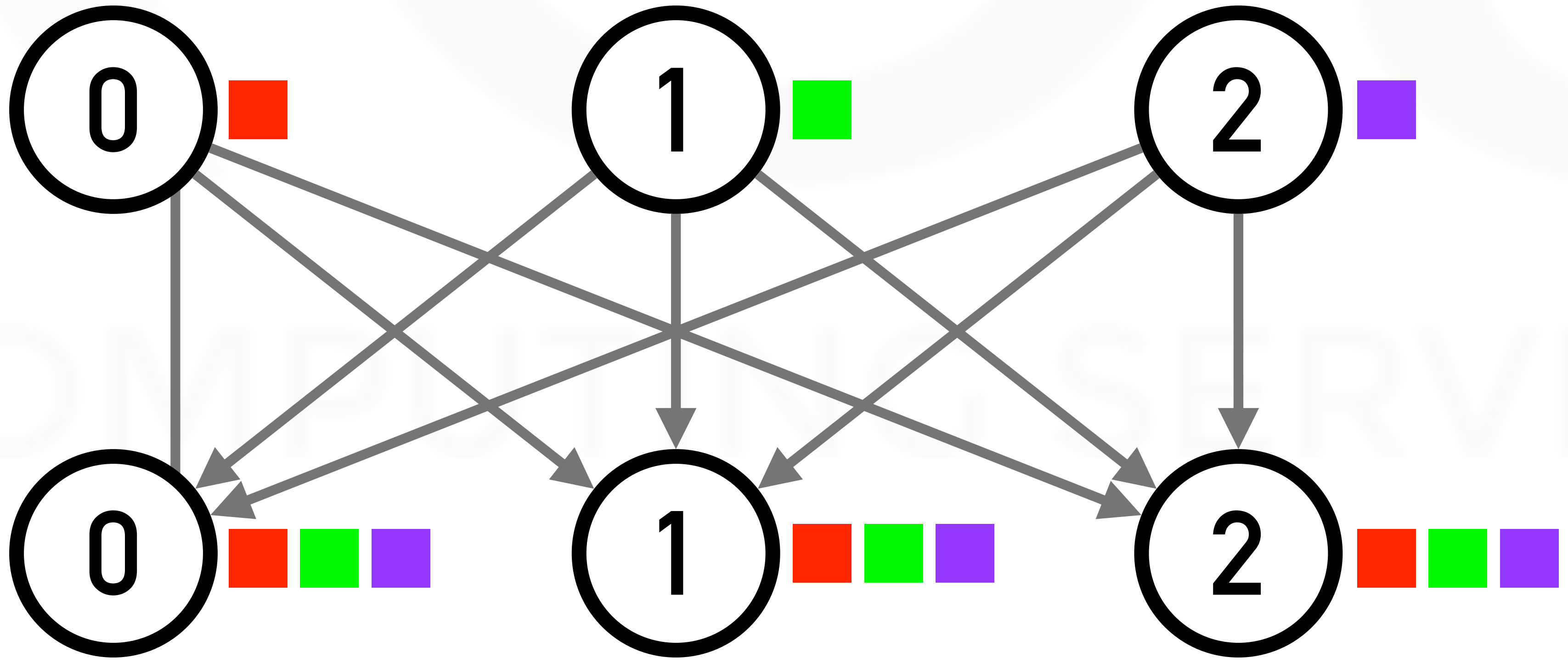
Scatter



Gather

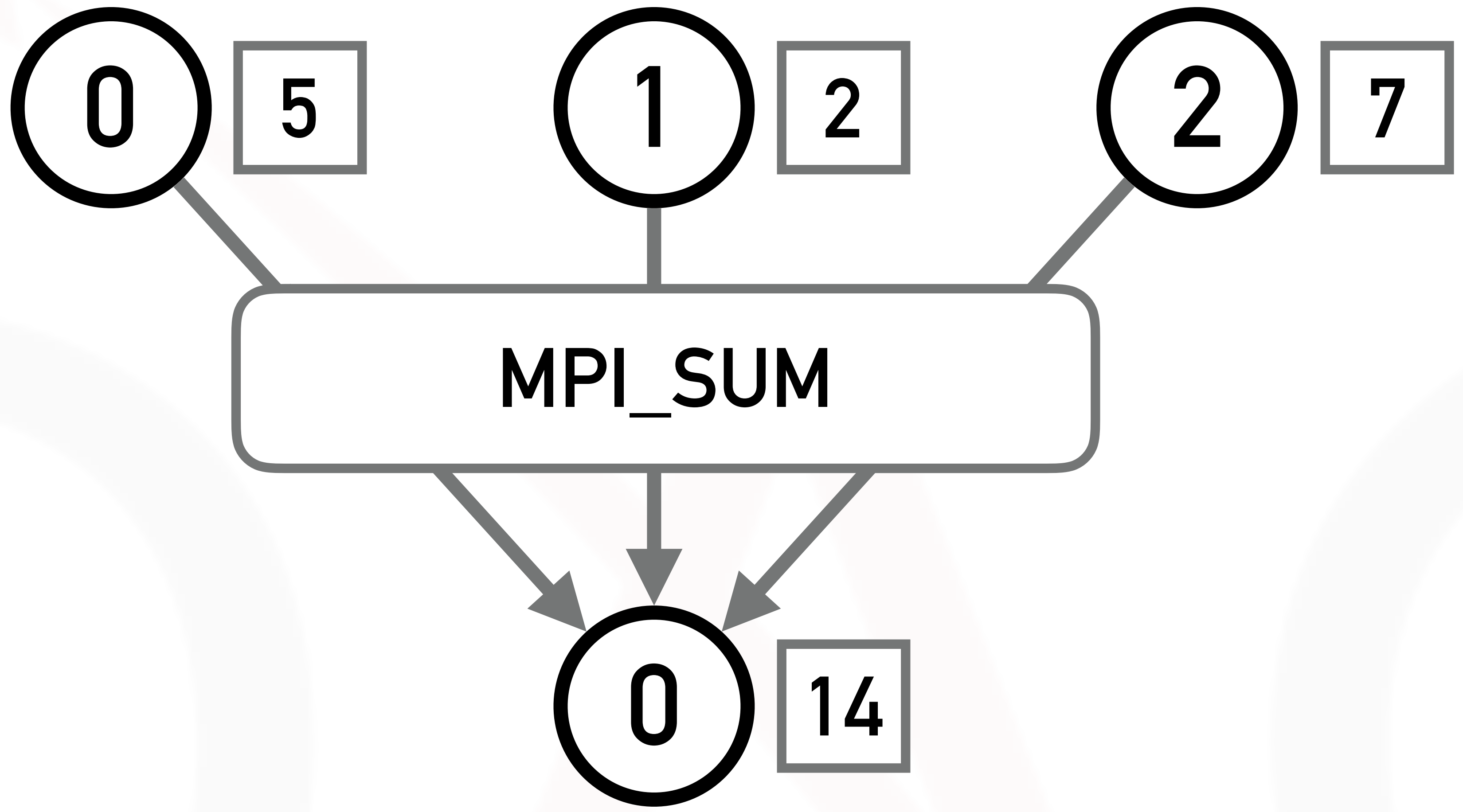


All Gather



SENDING AND RECEIVING DATA (COLLECTIVE COMM.)

Reduce



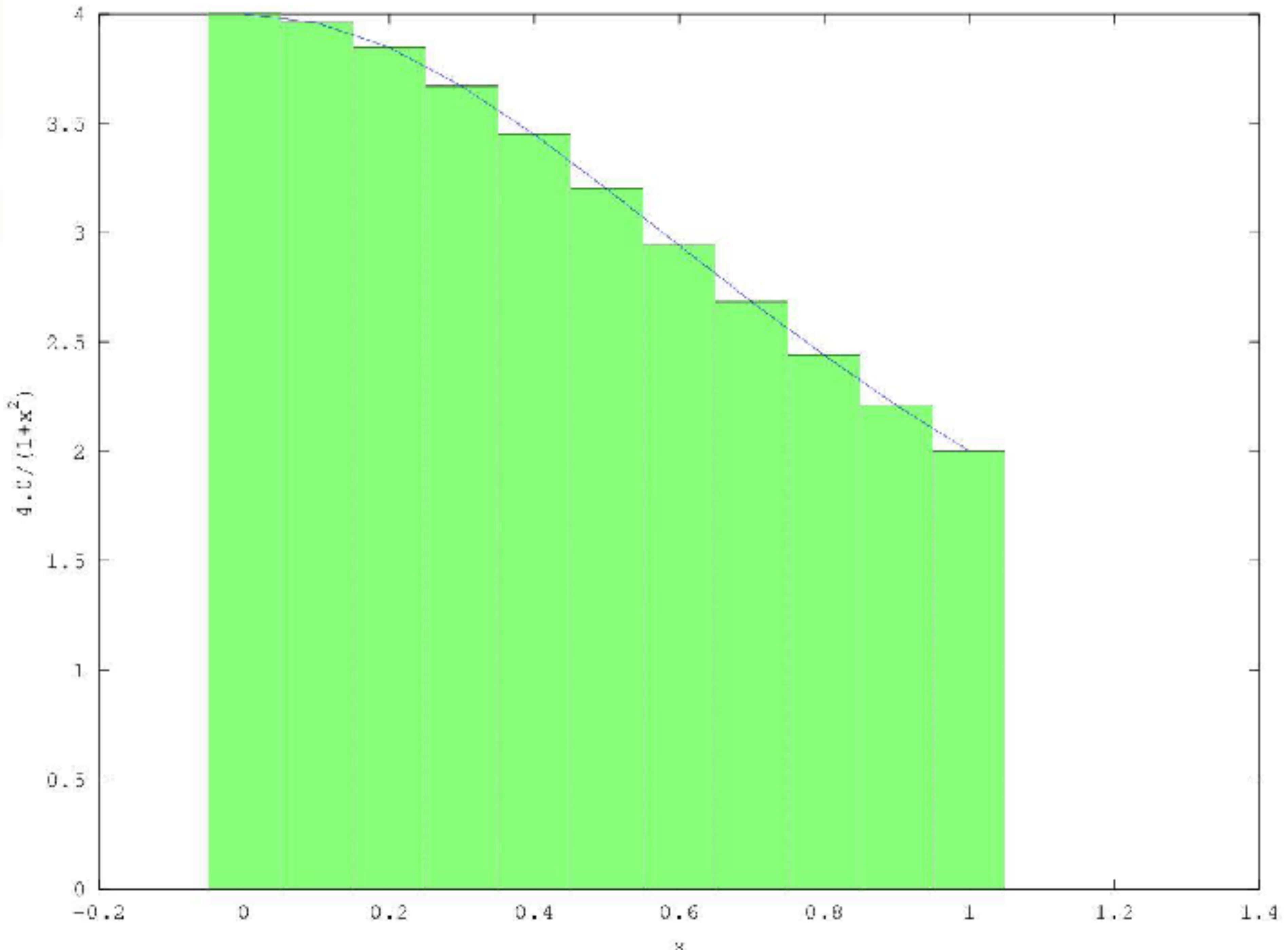
MPI_MAX – Returns the maximum element.
MPI_MIN – Returns the minimum element.
MPI_SUM – Sums the elements.
MPI_PROD – Multiplies all elements.
MPI_LAND – Performs a logical "and" across the elements.
MPI_LOR – Performs a logical "or" across the elements.
MPI_MAXLOC –the maximum value and the rank of the process that owns it.
MPI_MINLOC –the minimum value and the rank of the process that owns it.

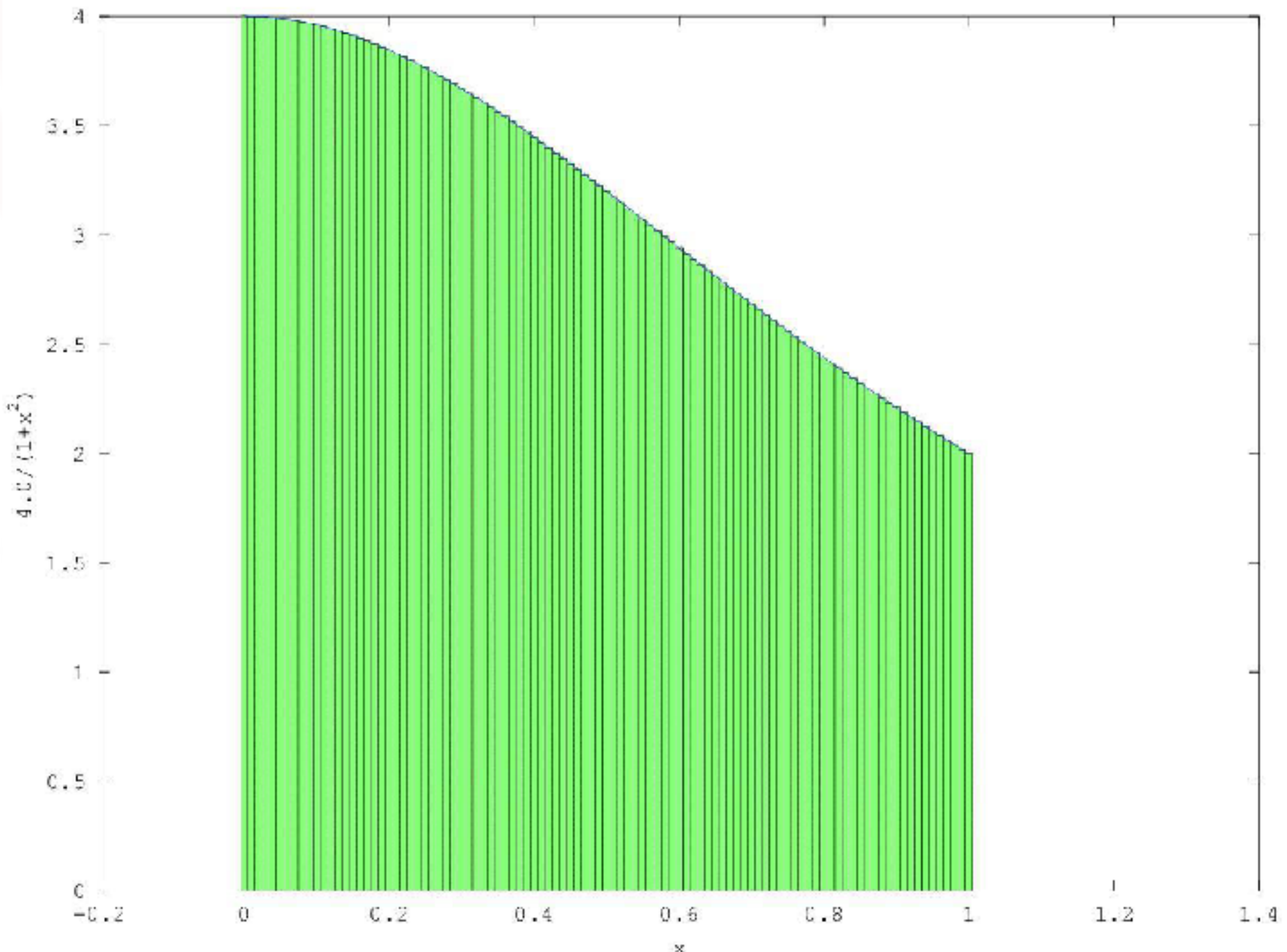
COMPUTING π

$$\pi = \int_0^1 \frac{4.0}{(1 + x^2)} dx = \sum_{i=0}^{N} \frac{4.0}{(1 + x^2)} \Delta x$$

COMPUTING π

$$\pi = \int_0^1 \frac{4.0}{(1+x^2)} dx = \sum_{i=0}^{N} \frac{4.0}{(1+x^2)} \Delta x$$





COMPUTING π (SERIAL VERSION)

```
import time
def Pi(num_steps):
    start = time.time()
    step = 1.0/num_steps
    sum = 0
    for i in xrange(num_steps):
        x= (i+0.5)*step
        sum = sum + 4.0/(1.0+x*x)
    pi = step * sum
    end =time.time()
    print ("Pi with %d steps is %f in %f secs" %(num_steps, pi, end-start))

if __name__ == '__main__':
    Pi(100000000)
```

COMPUTING π (MPI VERSION)

```
def Pi(num_steps):
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    Proc_num_steps = num_steps / size
    Start = rank * Proc_num_steps
    End = (rank + 1) * Proc_num_steps
    print ("%d: [%d,%d]" % (rank, Start, End))

    step = 1.0 / num_steps
    sum = 0
    for i in xrange(Start, End):
        x = (i + 0.5) * step
        sum = sum + 4.0 / (1.0 + x * x)

    return sum

(…)
```

COMPUTING π (MPI VERSION)

```
(…)

if __name__ == '__main__':
    num_steps = 100000000
    start = time.time()
    localsum = Pi(num_steps)

    localpi = localsum / num_steps
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    pi = comm.reduce(localpi, op=MPI.SUM, root=0)
    end = time.time()
    if rank == 0:
        print ("Pi with %d steps is %f in %f secs" % (num_steps, pi, end - start))
```

REFERENCES

REFERENCES

- https://computing.llnl.gov/tutorials/mpi/
- https://mpi4py.readthedocs.io/en/stable/
- https://cas-eresearch.github.io/astroinformatics2017/day3.html