MPI Basics with Python

Iman Rahbari, Computation Scientist

Cesar Sul, Research Computing Facilitator

Outline

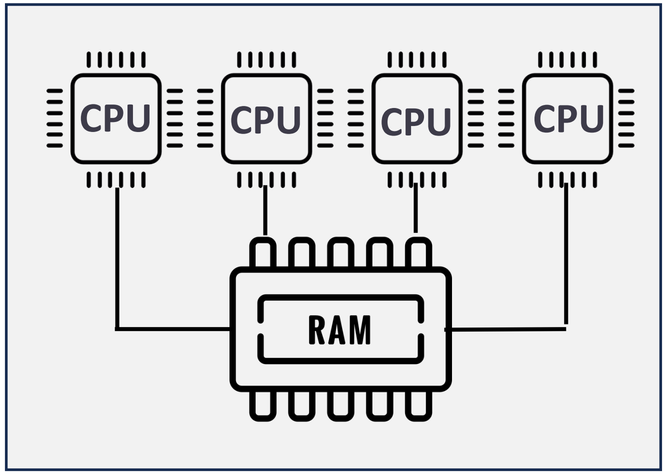
- Parallel Computing
- What is MPI
- Running programs in parallel
- Dividing work among workers
- Point to point communication
- Collective communication

Parallel Computing:

Parallel computing is when *multiple* *processors* or *computers* work together to solve a problem at the same time.

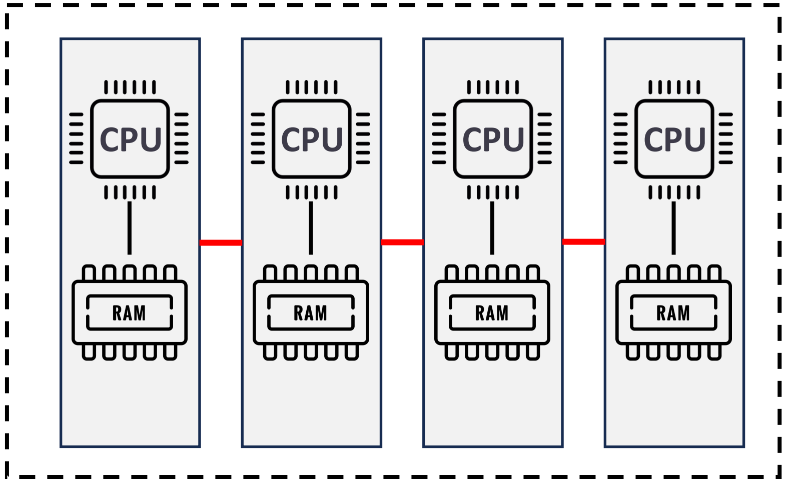
Main difference between these two cases is the **underlying memory**

Multiple processors:
Shared Memory



All access the same memory
Multi-Processing
OpenMP programming

Multiple computers:
Distributed Memory



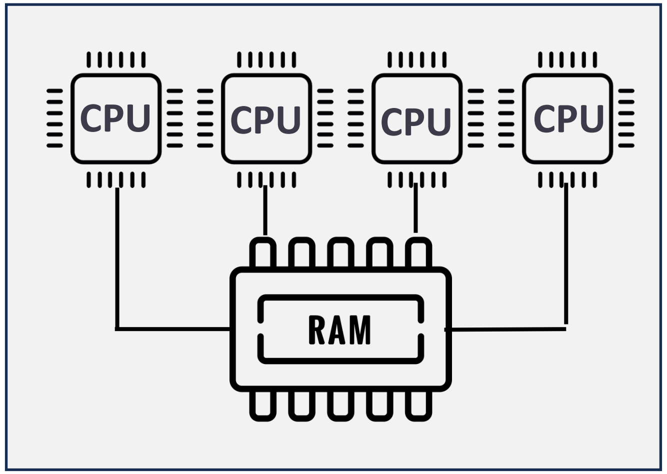
Each CPU has a separate memory
All CPUs communicate via network
MPI programming

Parallel Computing:

Parallel computing is when *multiple processors* or *computers* work together to solve a problem at the same time.

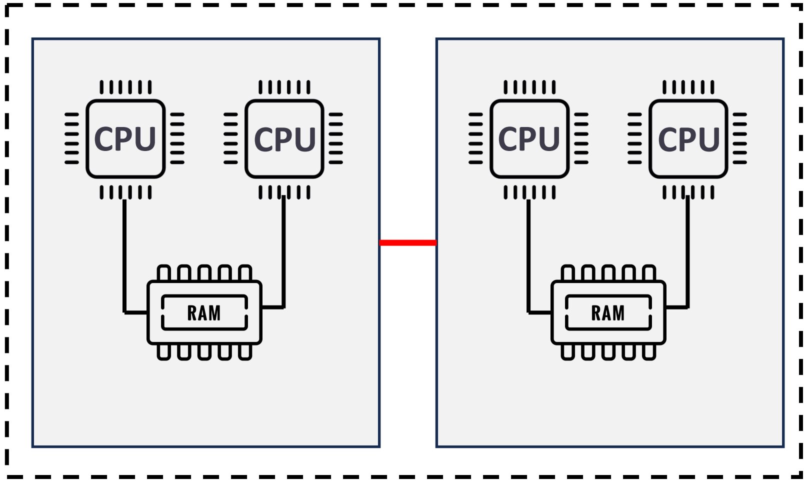
Main difference between these two cases is the **underlying memory**

Multiple processors:
Shared Memory



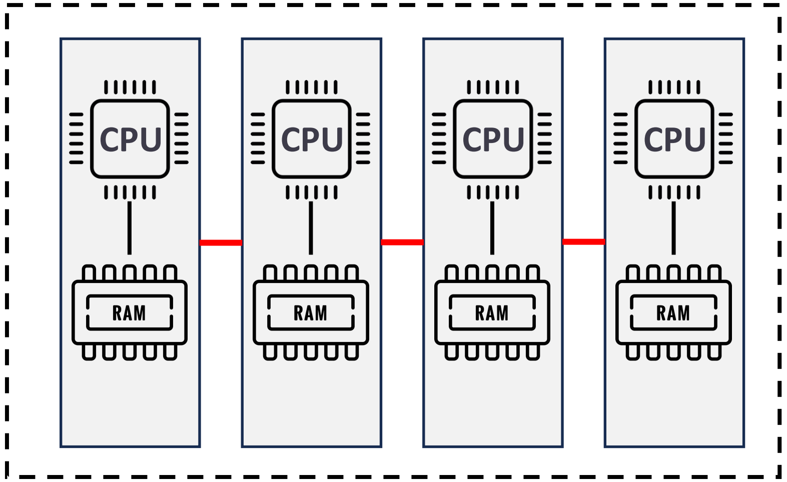
All access the same memory
Multi-Processing
OpenMP programming

Hybrid



A combination of both
MPI+OpenMP hybrid programming

Multiple computers:
Distributed Memory



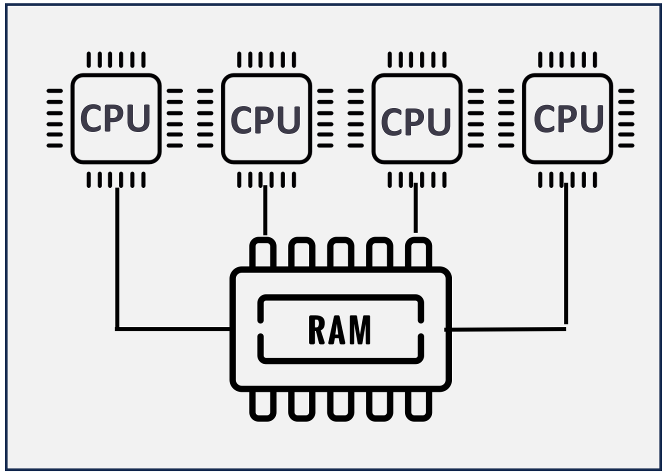
Each CPU has a separate memory
All CPUs communicate via network
MPI programming

Parallel Computing:

Parallel computing is when *multiple* *processors* or *computers* work together to solve a problem at the same time.

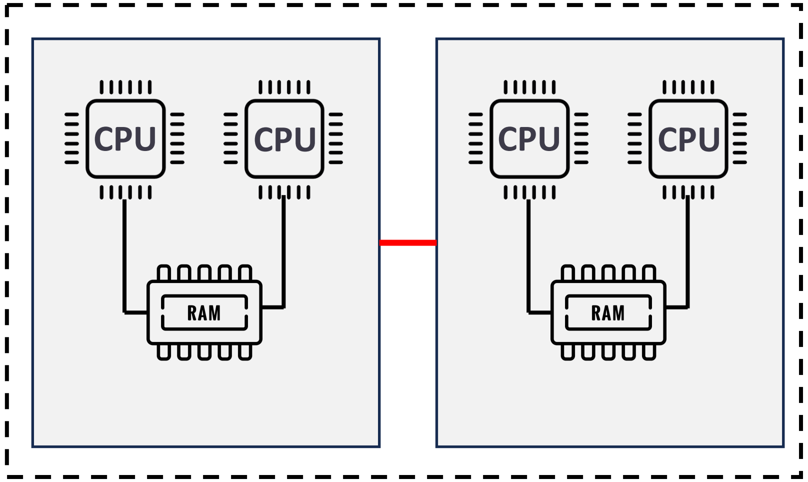
Main difference between these two cases is the **underlying memory**

Multiple processors:
Shared Memory



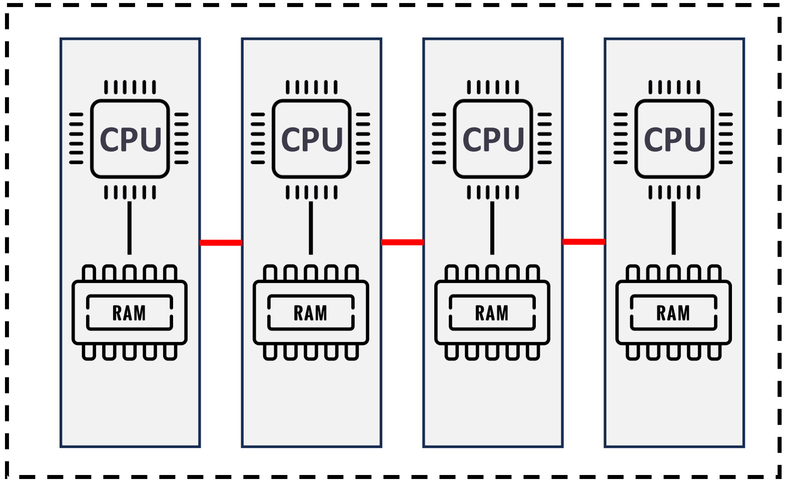
All access the same memory
Multi-Processing
OpenMP programming

Hybrid



A combination of both
MPI+OpenMP hybrid programming

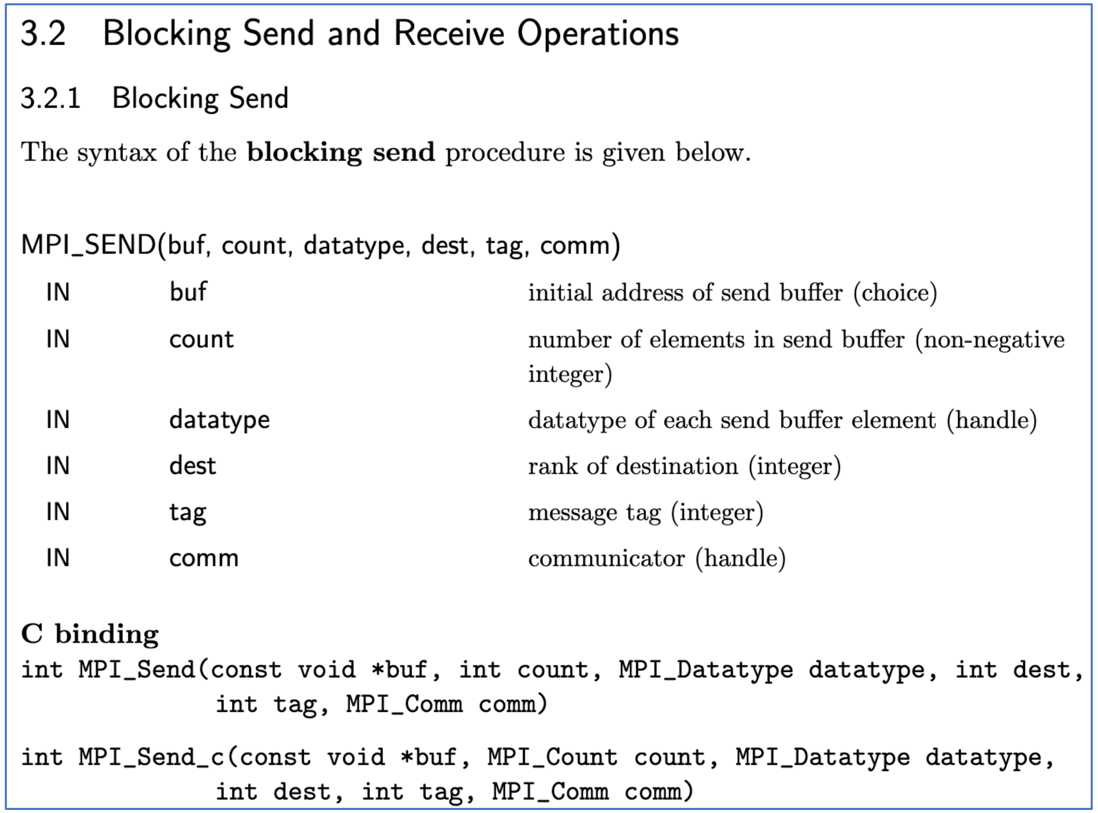
Multiple computers:
Distributed Memory



Each CPU has a separate memory
All CPUs communicate via network
MPI programming
This is our focus here!

What is MPI?

- Message Passing Interface
- Defines a standard for programs to communicate with each other
- Describes a set of functions and their expected behavior
- MPI_Send() is described but not *implemented*
- Implementation is left to others



3.2 Blocking Send and Receive Operations

3.2.1 Blocking Send

The syntax of the **blocking send** procedure is given below.

MPI_SEND(buf, count, datatype, dest, tag, comm)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

C binding

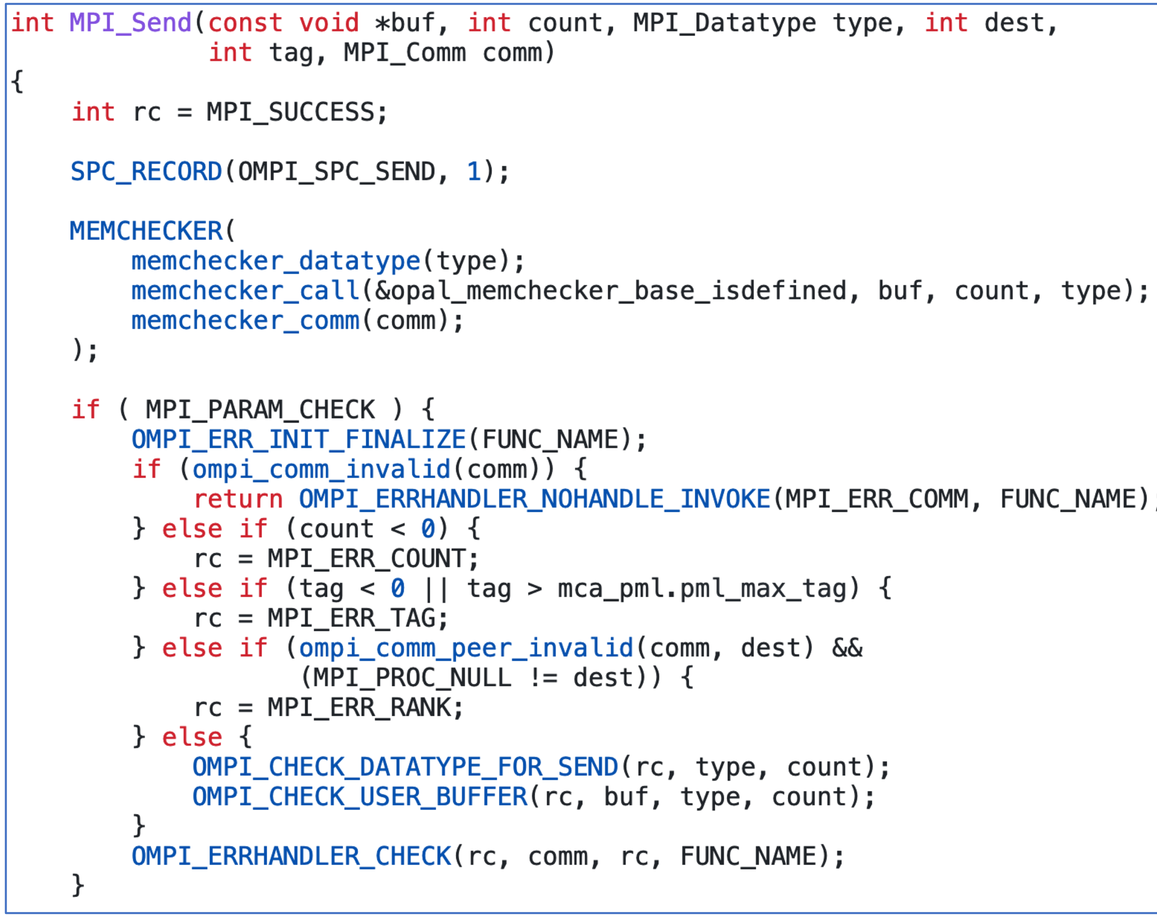
```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

```
int MPI_Send_c(const void *buf, MPI_Count count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

Official description for MPI_Send

What is MPI?

- There are many MPI implementations
- Each implementation must be compliant with MPI description
- On Discovery/Endeavour you will find
    - openmpi
    - mpich
    - intel-mpi
    - mvapich
- Each solves the same problems but in different ways



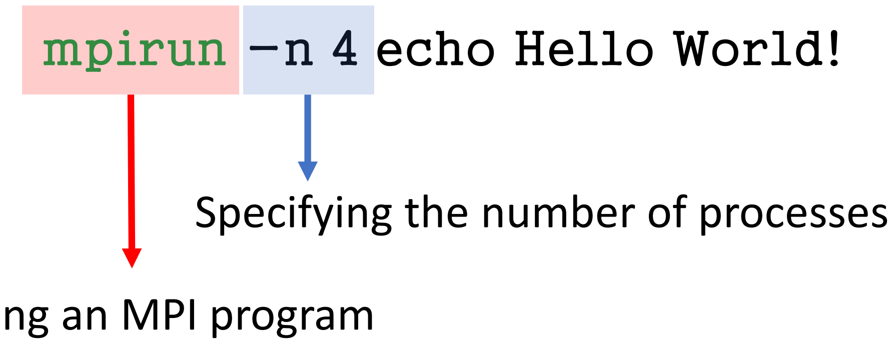
```
int MPI_Send(const void *buf, int count, MPI_Datatype type, int dest,
             int tag, MPI_Comm comm)
{
    int rc = MPI_SUCCESS;

    SPC_RECORD(OMPI_SPC_SEND, 1);

    MEMCHECKER(
        memchecker_datatype(type);
        memchecker_call(&opal_memchecker_base_isdefined, buf, count, type);
        memchecker_comm(comm);
    );

    if ( MPI_PARAM_CHECK ) {
        OMPI_ERR_INIT_FINALIZE(FUNC_NAME);
        if (ompi_comm_invalid(comm)) {
            return OMPI_ERRHANDLER_NOHANDLE_INVOKE(MPI_ERR_COMM, FUNC_NAME)
        } else if (count < 0) {
            rc = MPI_ERR_COUNT;
        } else if (tag < 0 || tag > mca_pml.pml_max_tag) {
            rc = MPI_ERR_TAG;
        } else if (ompi_comm_peer_invalid(comm, dest) &&
                   (MPI_PROC_NULL != dest)) {
            rc = MPI_ERR_RANK;
        } else {
            OMPI_CHECK_DATATYPE_FOR_SEND(rc, type, count);
            OMPI_CHECK_USER_BUFFER(rc, buf, type, count);
        }
        OMPI_ERRHANDLER_CHECK(rc, comm, rc, FUNC_NAME);
    }
```

OpenMPI version of MPI_Send

Serial

```
echo Hello World!
```

Parallel



```
mpirun -n 4 echo Hello World!
```

Specifying the number of processes

Running an MPI program

The name “mpirun” is not part of the standard, other names include:

- SLURM: `srun`
- Mpich2: `mpiexec`
- IBM SP: `poe`
- Stampede2: `ibrun`

If you don’t have enough CPUs to run this command on your machine, you may get:

```
There are not enough slots available in the system
```

Solution: `mpirun --oversubscribe -n 4 echo Hello World!`

Serial `echo Hello World!`

Parallel `mpirun −n 4 echo Hello World!`

Specifying the number of processes

Running an MPI program

- MPI copies a program several times and run them individually

- How to tell a **program**, it is being executed by an MPI command and all pieces should work together?
  - **In C and Fortran:**

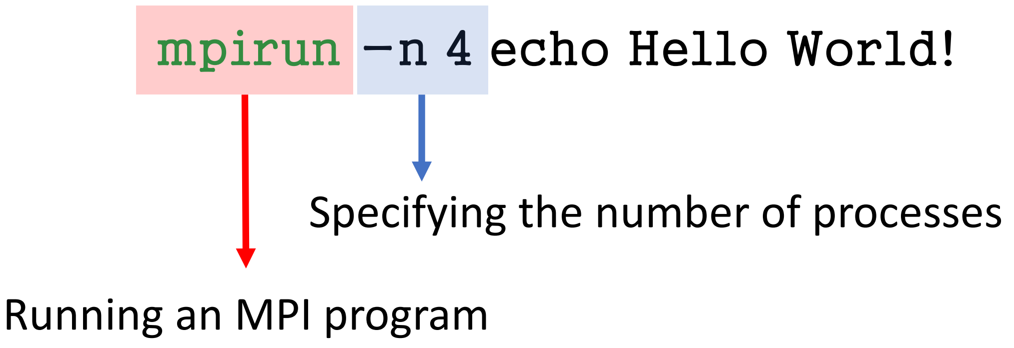
    Start the program with function: `MPI_INIT` and clean up with: `MPI_FINALIZE`
  - **In Python:**

    Handled by the library: `from mpi4py import MPI`

Serial

```
echo Hello World!
```

Parallel



```
mpirun -n 4 echo Hello World!
```

Specifying the number of processes

Running an MPI program

- MPI copies a program several times and run them individually
- How to tell a **program**, it is being executed by an MPI command and all pieces should work together?
  - **In C and Fortran:**

    Start the program with function: `MPI_INIT` and clean up with: `MPI_FINALIZE`
  - **In Python:**

    Handled by the library: `from mpi4py import MPI`

- `np = MPI.COMM_WORLD.Get_size()` Total number of processes
- `myrank = MPI.COMM_WORLD.Get_rank()` Rank of each process (between 0,np-1)

Hello World MPI

```
from mpi4py import MPI

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

print(f"Hello from rank {rank}")
```

- Each process will:
  - Start up
  - Find out their "rank" number
  - Print rank to screen
- See examples/hello_world for code

Hello World MPI

```
from mpi4py import MPI

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

print(f"Hello from rank {rank}")
```

Communicator - Group of processes

ID number within the communicator

- Each process will:
  - Start up
  - Find out their "rank" number
  - Print rank to screen
- See examples/hello_world for code

Getting Compute resources

- MPI programs must be run in Slurm jobs
- Each "rank" must be given a Slurm "task"
- Use reservation for this workshop:
- `#SBATCH --reservation=bootcamp`
- `#SBATCH --account=hpcsuppt_613`
- `#SBATCH --partition=gpu`
- We will be using mpi4py and numpy
- If not installed, use `pip install mpi4py numpy`

Run from interactive job

```
$ salloc --ntasks=4 --reservation=bootcamp
--account=hpcsuppt_613 --time=1:00:00 --partition=gpu




$ module load usc
$ module load python

$ srun python3 hello_world_mpi.py
Hello from rank 0!
Hello from rank 2!
Hello from rank 3!
Hello from rank 1!
```

Run from batch job

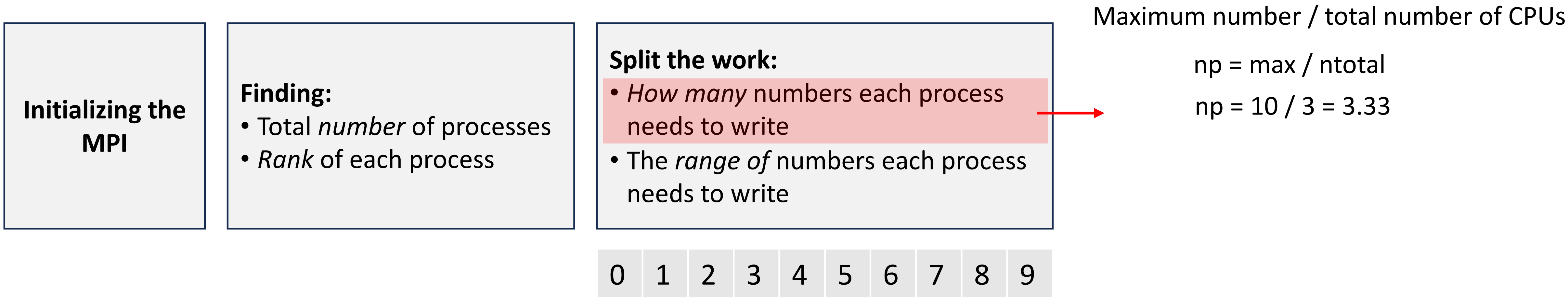
```
#!/bin/bash
#SBATCH --ntasks=4
#SBATCH --partiton=debug
#SBATCH --mem-per-cpu=2GB

module load usc
module load python

srun python3 hello_world_mpi.py
```

- Use interactive job to help build job script
- Everything we did in interactive job
- Output saved to slurm-xxxxx.out file

Doing Calculations in Parallel

Serial Print from 0 to 9

Parallel Print from 0 to 9 with 3 processes, each process prints “its rank”, and “its portion of the numbers”
Last process carries whatever is left



Initializing the MPI

Finding:

- Total *number* of processes
- *Rank* of each process

Split the work:

- *How many* numbers each process needs to write
- The *range of* numbers each process needs to write

Maximum number / total number of CPUs

np = max / ntotal

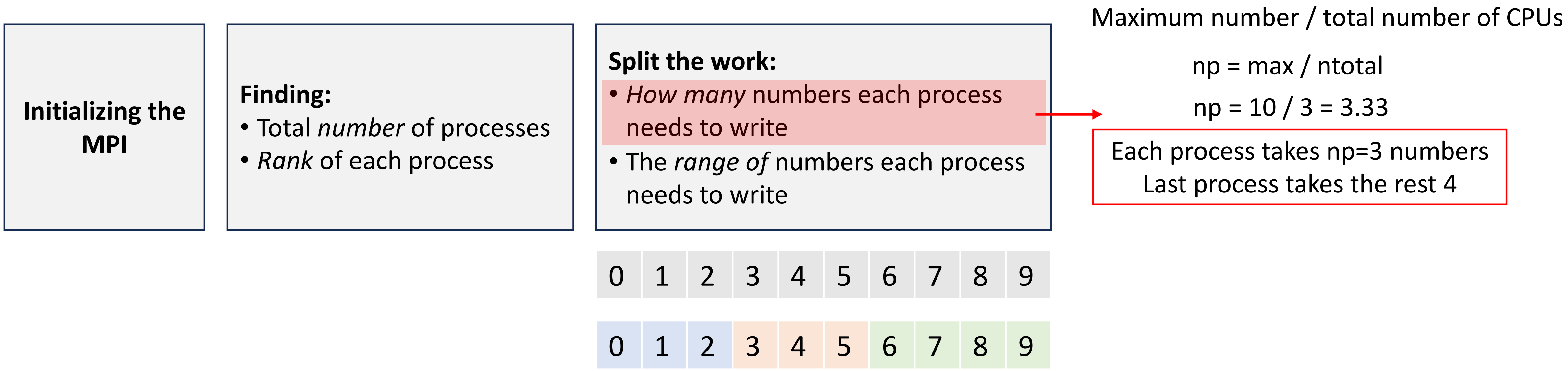
np = 10 / 3 = 3.33

0	1	2	3	4	5	6	7	8	9

Doing Calculations in Parallel

Serial Print from 0 to 9

Parallel Print from 0 to 9 with 3 processes, each process prints “its rank”, and “its portion of the numbers”
Last process carries whatever is left



Initializing the MPI

Finding:
- Total *number* of processes
- *Rank* of each process

Split the work:
- *How many* numbers each process needs to write
- The *range of* numbers each process needs to write

Maximum number / total number of CPUs

np = max / ntotal

np = 10 / 3 = 3.33

Each process takes np=3 numbers
Last process takes the rest 4

0 1 2 3 4 5 6 7 8 9

0 1 2 | 3 4 5 | 6 7 8 9

Doing Calculations in Parallel

Serial Print from 0 to 9

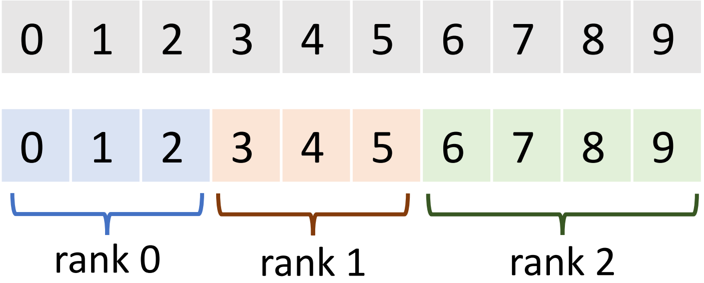
Parallel Print from 0 to 9 with 3 processes, each process prints “its rank”, and “its portion of the numbers”
Last process carries whatever is left

Initializing the MPI

Finding:
- Total *number* of processes
- *Rank* of each process

Split the work:
- *How many* numbers each process needs to write
- The *range of* numbers each process needs to write

Each process takes **np=3** numbers
Last process takes the rest 4



0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

rank 0 rank 1 rank 2

Doing Calculations in Parallel

Serial Print from 0 to 9

Parallel Print from 0 to 9 with 3 processes, each process prints “its rank”, and “its portion of the numbers”
Last process carries whatever is left

Initializing the MPI

Finding:
- Total *number* of processes
- *Rank* of each process

Split the work:
- *How many* numbers each process needs to write
- The *range of* numbers each process needs to write

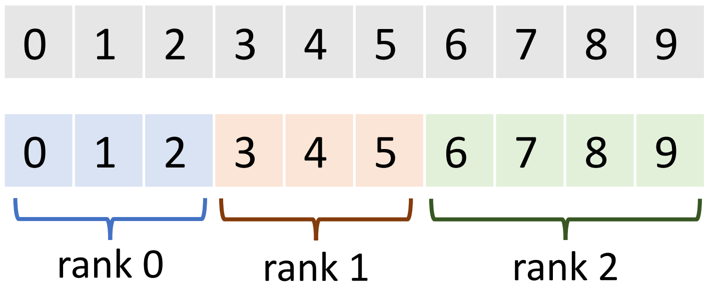
each process

last process

Each process takes **np=3** numbers
Last process takes the rest 4

$rank * np \leq range < r(ank+1) * np$

$rank * np \leq range < max$

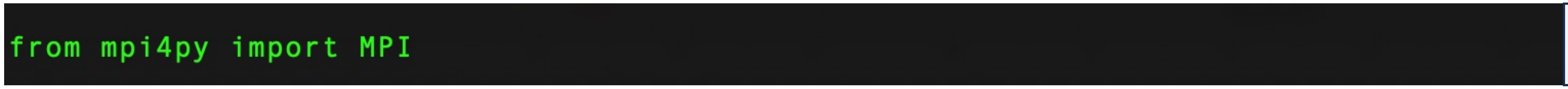


0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

Doing Calculations in Parallel

Serial Print from 0 to 9

Parallel Print from 0 to 9 with 3 processes, each process prints “its rank”, and “its portion of the numbers”
Last process carries whatever is left



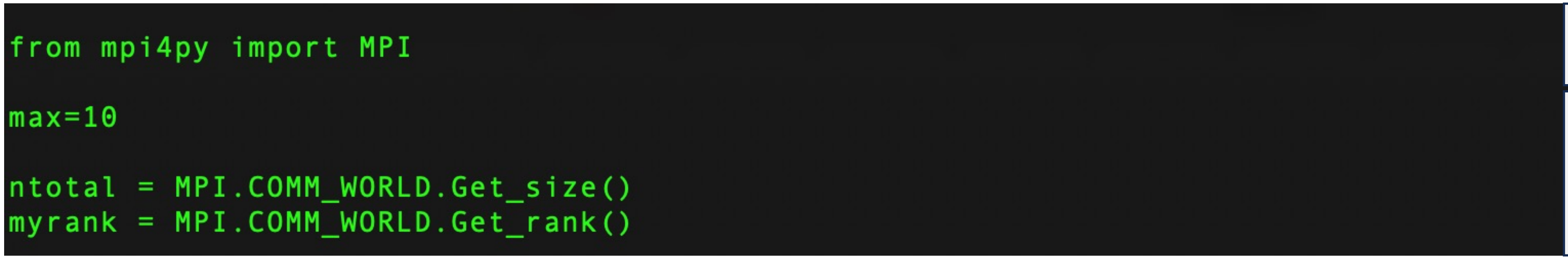
```
from mpi4py import MPI
```

Initializing the MPI

Doing Calculations in Parallel

Serial Print from 0 to 9

Parallel Print from 0 to 9 with 3 processes, each process prints “its rank”, and “its portion of the numbers”
Last process carries whatever is left



```
from mpi4py import MPI

max=10

ntotal = MPI.COMM_WORLD.Get_size()
myrank = MPI.COMM_WORLD.Get_rank()
```

Initializing the MPI

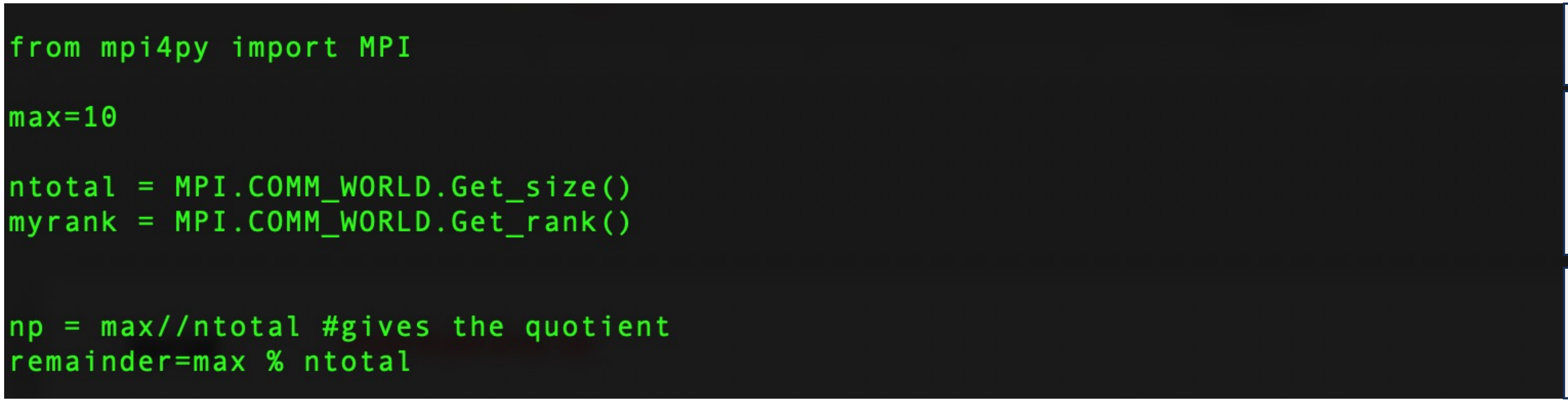
Finding:

- Total *number* of processes
- *Rank* of each process

Doing Calculations in Parallel

Serial Print from 0 to 9

Parallel Print from 0 to 9 with 3 processes, each process prints “its rank”, and “its portion of the numbers”
Last process carries whatever is left



```
from mpi4py import MPI

max=10

ntotal = MPI.COMM_WORLD.Get_size()
myrank = MPI.COMM_WORLD.Get_rank()


np = max//ntotal #gives the quotient
remainder=max % ntotal
```

Initializing the MPI

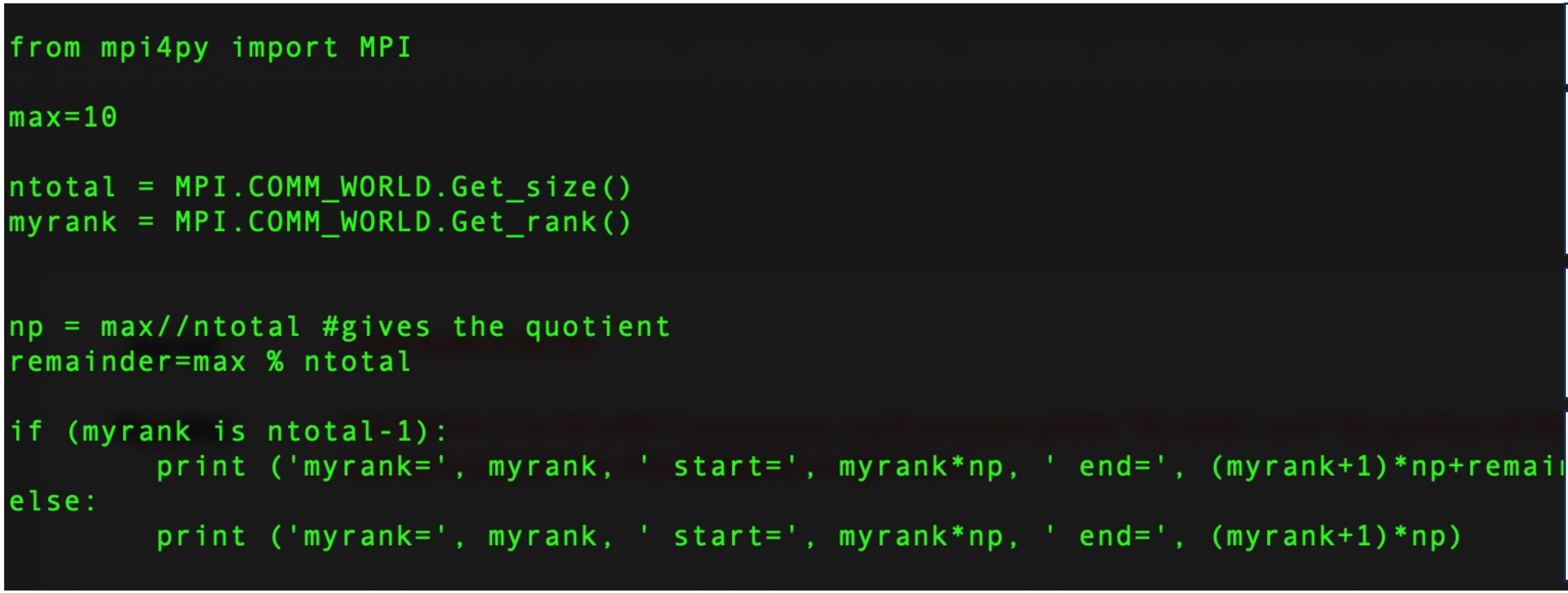
Finding:
- Total *number* of processes
- *Rank* of each process

Split the work:
- *How many* numbers each process needs to write

Doing Calculations in Parallel

Serial Print from 0 to 9

Parallel Print from 0 to 9 with 3 processes, each process prints “its rank”, and “its portion of the numbers”
Last process carries whatever is left



```
from mpi4py import MPI

max=10

ntotal = MPI.COMM_WORLD.Get_size()
myrank = MPI.COMM_WORLD.Get_rank()


np = max//ntotal #gives the quotient
remainder=max % ntotal

if (myrank is ntotal-1):
        print ('myrank=', myrank, ' start=', myrank*np, ' end=', (myrank+1)*np+remai
else:
        print ('myrank=', myrank, ' start=', myrank*np, ' end=', (myrank+1)*np)
```

Initializing the MPI

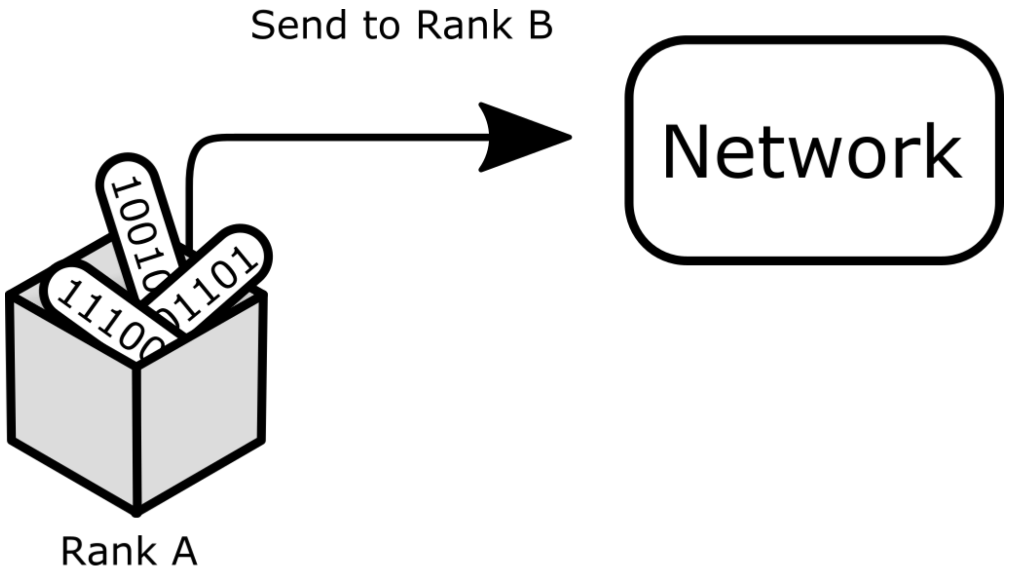
Finding:
- Total *number* of processes
- *Rank* of each process

Split the work:
- *How many* numbers each process needs to write

- The *range of* numbers each process needs to write

Point to Point Communication - Send

- Simplest way to send data
- `comm.send(buf,dest,tag)`
- Different parts
  - buf – the thing you want to send
  - dest – rank to send to
  - tag – label for data
- Send will wait for successful receive before continuing

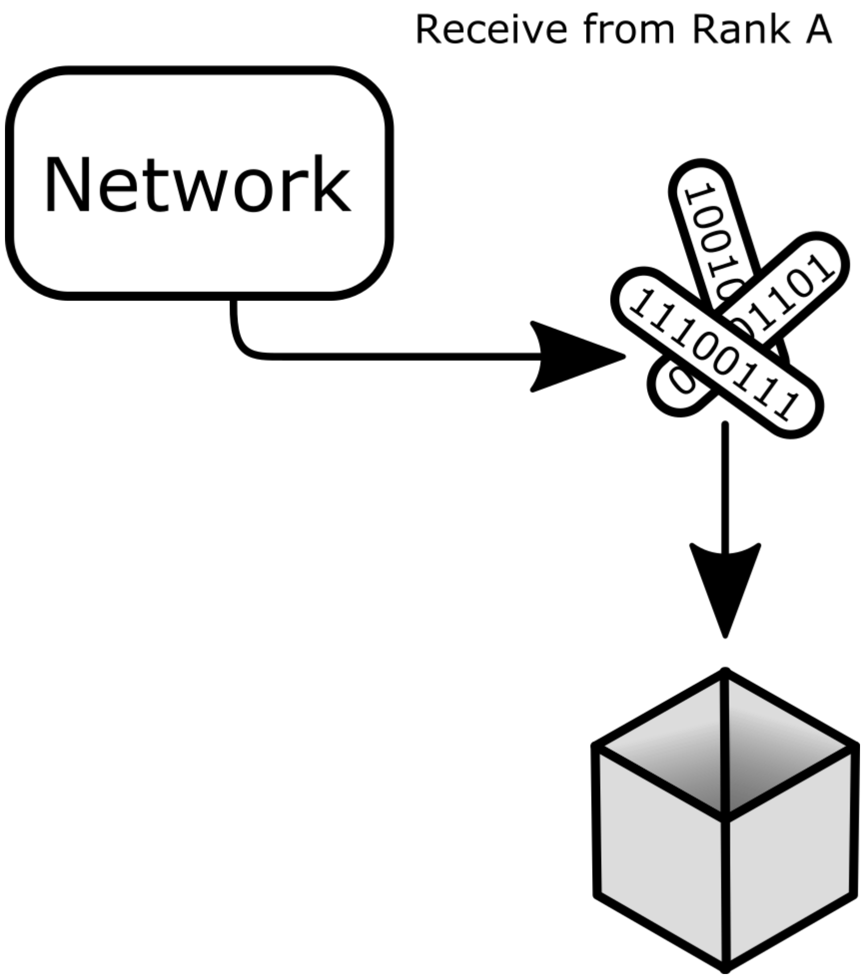


Point to Point Communication - Recv

- Simplest way to receive data

- `comm.recv(buf, source=ANY_SOURCE, tag=ANY_TAG)`

- Different parts
  - buf – container for data
  - source – rank data is expected from
  - tag – label for data
- Recv will wait for data before continuing
- Recv will only accept data from `dest` with matching `tag`



Send/Recv example

```
from mpi4py import MPI

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

print(f'Rank {rank} starting up ...')

good_tag=7
data = None

if rank == 0:
    print(f'Rank {rank}: Sending data to rank 1 with tag {good_tag}')
    comm.send(data,dest=1,tag=good_tag)

if rank == 1:
    #data=None
    print(f'Rank {rank}: Waiting for data from rank 1 with tag {good_tag}')
    data=comm.recv(source=0, tag=good_tag)
    print(f'Rank {rank}: Got data: {data} from rank 0 with tag {good_tag}')

print(f'Rank {rank} shutting down with data={data}.')
```

Send/Recv example - Initialize

```python
from mpi4py import MPI

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

print(f'Rank {rank} starting up ...')

good_tag=7
data = None

if rank == 0:
    data=412
    print(f'Rank {rank}: Sending data to rank 1 with tag {good_tag}')
    comm.send(data,dest=1,tag=good_tag)

if rank == 1:
    #data=None
    print(f'Rank {rank}: Waiting for data from rank 1 with tag {good_tag}')
    data=comm.recv(source=0, tag=good_tag)
    print(f'Rank {rank}: Got data: {data} from rank 0 with tag {good_tag}')

print(f'Rank {rank} shutting down with data={data}.')
```

Send/Recv example - Rank 0

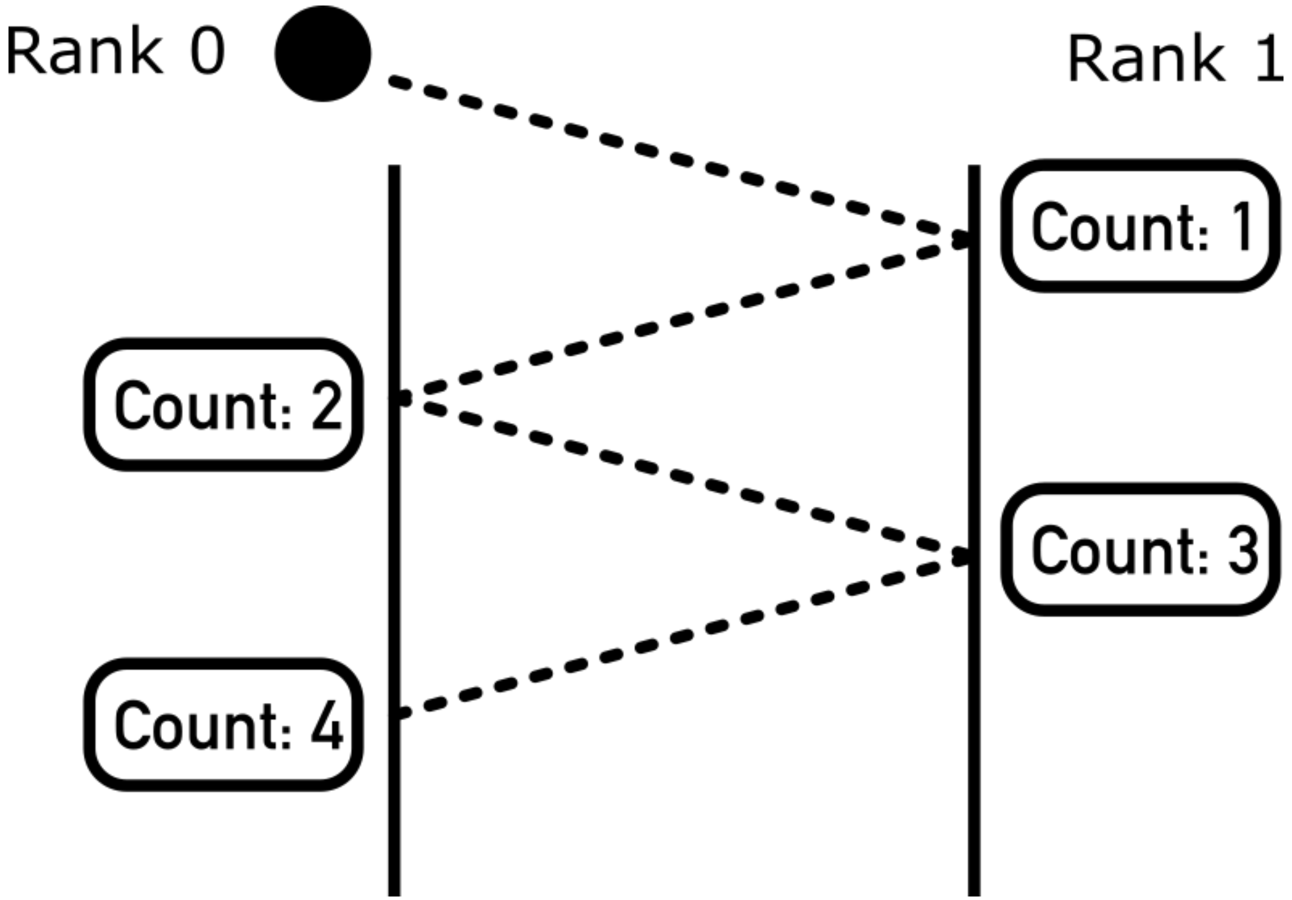
```
from mpi4py import MPI

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

print(f'Rank {rank} starting up ...')

good_tag=7
data = None

if rank == 0:
    data=412
    print(f'Rank {rank}: Sending data to rank 1 with tag {good_tag}')
    comm.send(data,dest=1,tag=good_tag)

if rank == 1:
    #data=None
    print(f'Rank {rank}: Waiting for data from rank 1 with tag {good_tag}')
    data=comm.recv(source=0, tag=good_tag)
    print(f'Rank {rank}: Got data: {data} from rank 0 with tag {good_tag}')

print(f'Rank {rank} shutting down with data={data}.')
```

Send/Recv example - Rank 1

```
from mpi4py import MPI

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

print(f'Rank {rank} starting up ...')

good_tag=7
data = None

if rank == 0:
    data=412
    print(f'Rank {rank}: Sending data to rank 1 with tag {good_tag}')
    comm.send(data,dest=1,tag=good_tag)

if rank == 1:
    #data=None
    print(f'Rank {rank}: Waiting for data from rank 1 with tag {good_tag}')
    data=comm.recv(source=0, tag=good_tag)
    print(f'Rank {rank}: Got data: {data} from rank 0 with tag {good_tag}')

print(f'Rank {rank} shutting down with data={data}.')
```

MPI Ping Pong

- Classic MPI example
- Two ranks alternate sending a message
- Counter is incremented on receipt
- Stop after a N rounds



MPI Ping Pong - Code

```
from mpi4py import MPI
import numpy as np


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


if world_size !=2:
    print("Only two can play")
    sys.exit(1)


print(f'Rank {rank} starting up...')


counter = 0
max_counter = 10


if rank == 0:
    partner = 1
if rank == 1:
    partner = 0


while counter < max_counter:
    counter=comm.recv(source=partner)
    print(f'Rank {rank}: Got message {counter} from rank {partner}')
    counter = counter + 1



    print(f'Rank {rank}: sending message {counter} to rank {partner}')
    comm.send(counter,dest=partner)
```

MPI Ping Pong - Initialize

```
from mpi4py import MPI
import numpy as np


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


if world_size !=2:
    print("Only two can play")
    sys.exit(1)


print(f'Rank {rank} starting up...')



if rank == 0:
    partner = 1
if rank == 1:
    partner = 0
```

MPI Ping Pong - Send/Receive loop

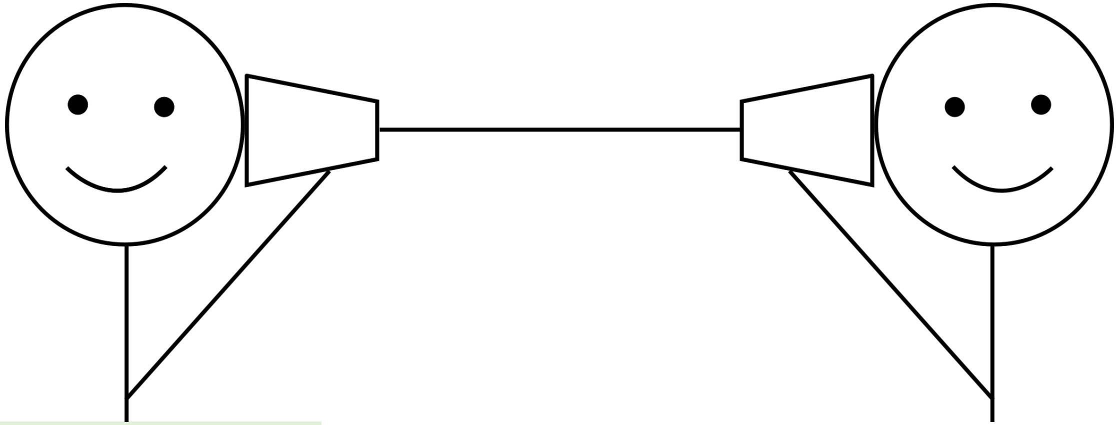
```
counter = 0
max_counter = 10

while counter < max_counter:
    counter=comm.recv(source=partner)
    print(f'Rank {rank}: Got message {counter} from rank {partner}')
    counter = counter + 1



    print(f'Rank {rank}: sending message {counter} to rank {partner}')
    comm.send(counter,dest=partner)
```

MPI Ping Pong

- Something is wrong with this example!
- Run the program on your own
- What could you change to make it work?

Deadlock

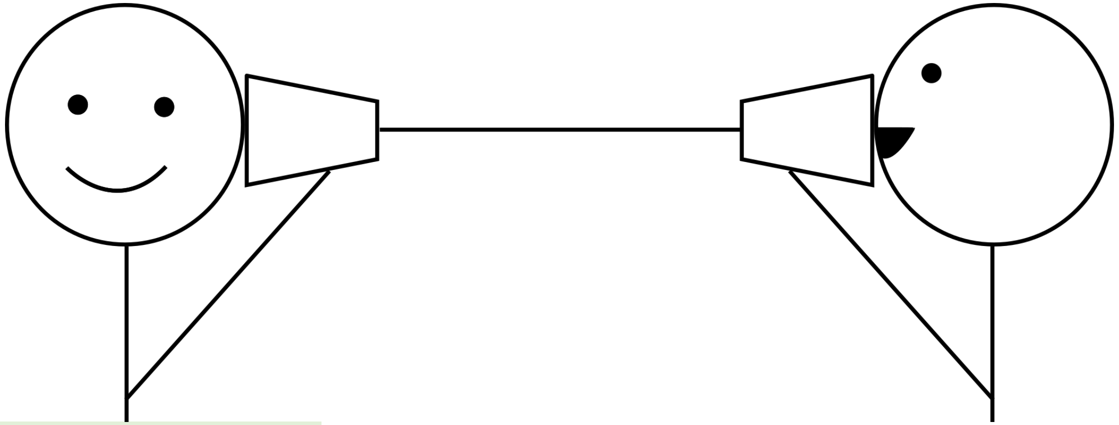
- Both ranks are "receiving"
- Can't proceed until done



```
counter = 0
max_counter = 10




while counter < max_counter:
    counter=comm.recv(source=partner)
    print(f'Rank {rank}: Got message {counter} from rank {partner}')
    counter = counter + 1

    print(f'Rank {rank}: sending message {counter} to rank {partner}')
    comm.send(counter,dest=partner)
```

Deadlock

- Be careful with message coordination



```
counter = 0
max_counter = 10

if rank == 0:
    comm.send(counter,dest=partner)

while counter < max_counter:
    counter=comm.recv(source=partner)
    print(f'Rank {rank}: Got message {counter} from rank {partner}')


    print(f'Rank {rank}: sending message {counter} to rank {partner}')
    comm.send(counter,dest=partner)
    counter = counter + 1
```

Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

0			0		0		
						0	0
	0						
	0						
		0				0	
			0				

Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

Initializing the MPI

Finding:

- Total *number* of processes
- *Rank* of each process

0			0		0		
						0	0
	0						
	0						
		0				0	
			0				

Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

Initializing the MPI

Finding:
- Total *number* of processes
- *Rank* of each process

Creating the matrix

Split the work:
- The *range of* elements each process needs to read
- Calculate

0			0		0		
						0	0
	0						
	0						
		0				0	
			0				

Example

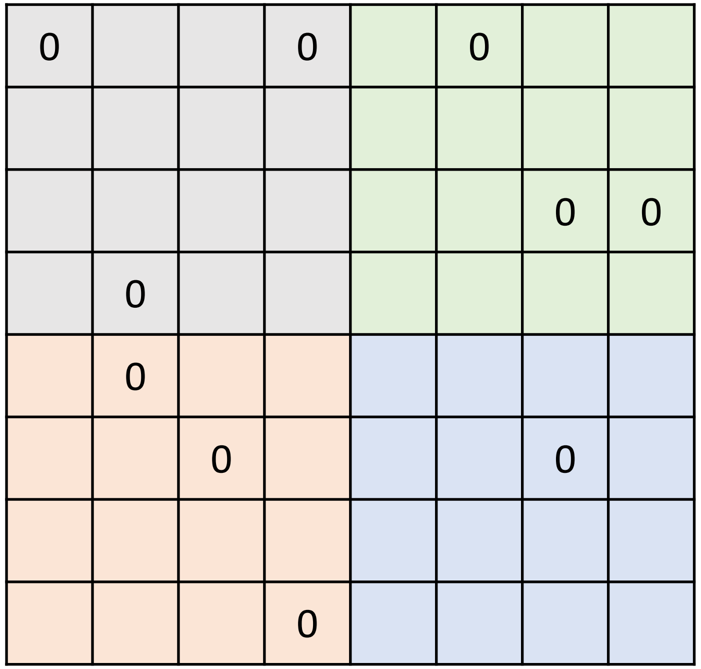
Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

Initializing the MPI

Finding:
- Total *number* of processes
- *Rank* of each process

Creating the matrix

Split the work:
- The *range of* elements each process needs to read
- Calculate



0			0		0		
						0	0
	0						
	0						
		0				0	
			0				

Example

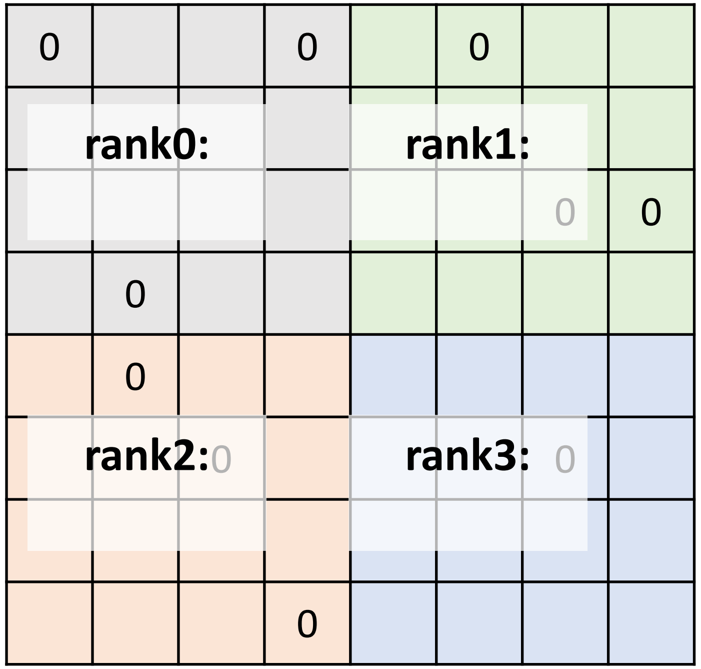
Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

Initializing the MPI

Finding:
- Total *number* of processes
- *Rank* of each process

Creating the matrix

Split the work:
- The *range of* elements each process needs to read
- Calculate



0			0		0		
						0	0
	0						
	0						
		0				0	
			0				

Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

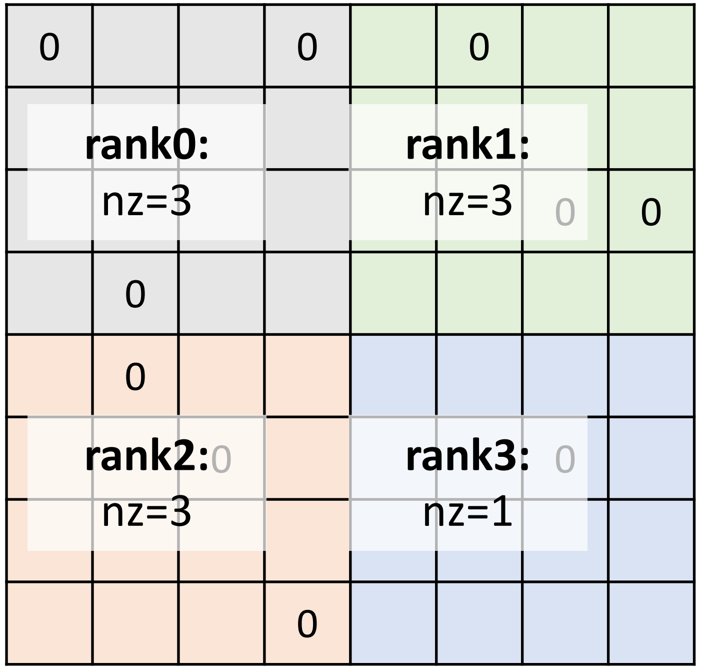
Initializing the MPI

Finding:
- Total *number* of processes
- *Rank* of each process

Creating the matrix

Split the work:
- The *range of* elements each process needs to read
- Calculate

Calculation



Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

Initializing the MPI

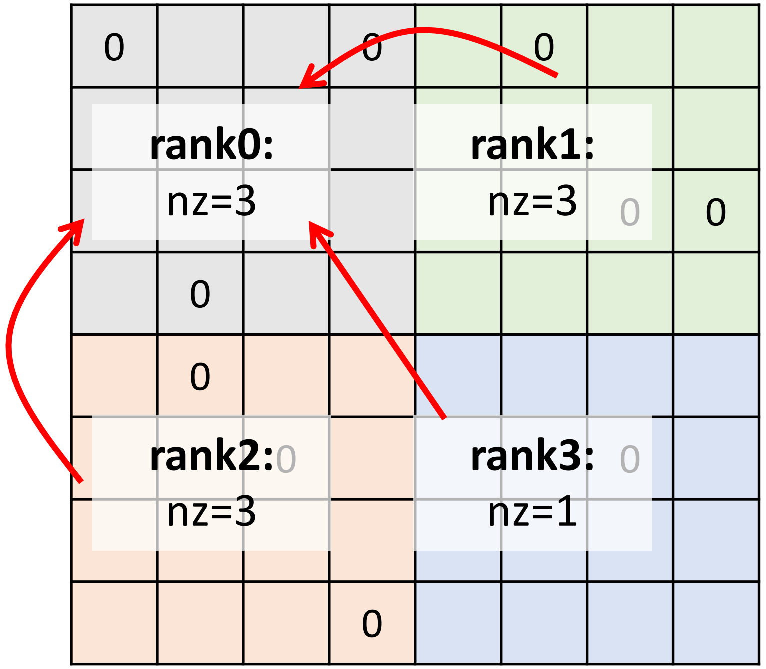
Finding:
- Total *number* of processes
- *Rank* of each process

Creating the matrix

Split the work:
- The *range of* elements each process needs to read
- Calculate

Calculation

Sending the results to rank0



Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

Initializing the MPI

Finding:

- Total *number* of processes
- *Rank* of each process

Creating the matrix

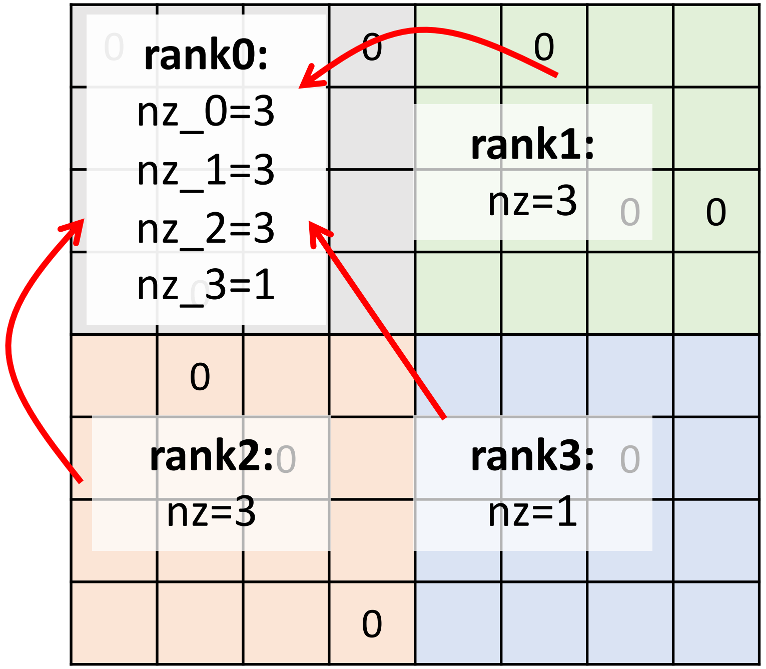
Split the work:

- The *range of* elements each process needs to read
- Calculate

Calculation

Sending the results to rank0

Receiving the results by rank0



rank0:
nz_0=3
nz_1=3
nz_2=3
nz_3=1

rank1:
nz=3

rank2:
nz=3

rank3:
nz=1

Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

Initializing the MPI

Finding:
- Total *number* of processes
- *Rank* of each process

Creating the matrix

Split the work:
- The *range of* elements each process needs to read
- Calculate

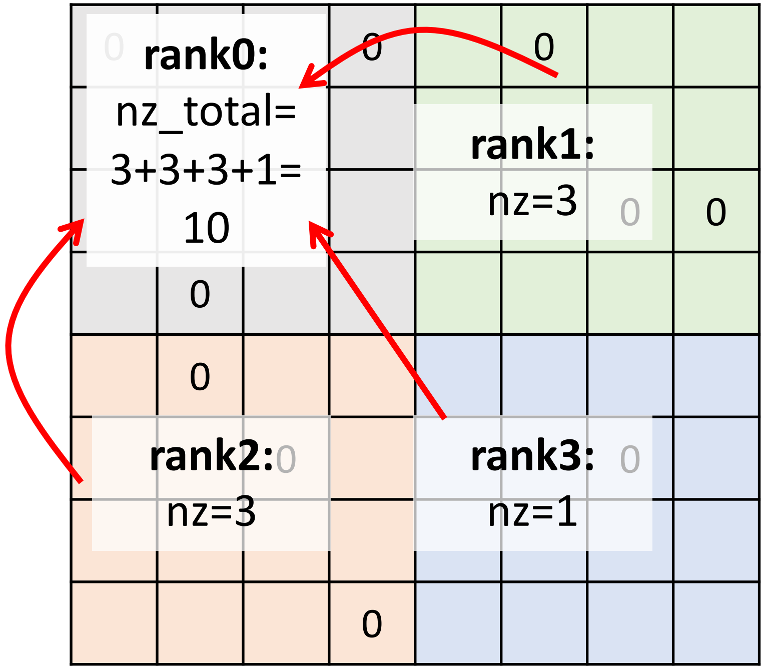
Calculation

Sending the results to rank0

Receiving the results by rank0

Calculation in rank0

Printing the results by rank0



Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

```
from mpi4py import MPI
```

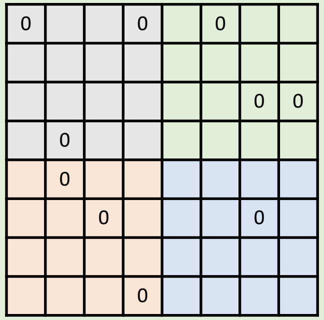
Initializing the MPI

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

Finding:

- Total *number* of processes
- *Rank* of each process

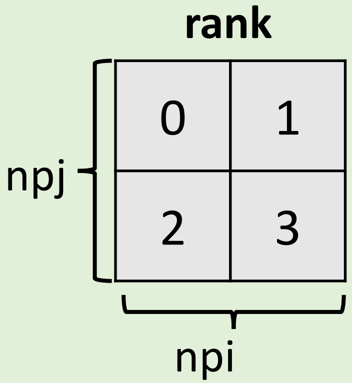
```
import numpy as np
np.random.seed(10)
A=np.random.rand(8,8)
ilist=np.random.randint(0,7,10)
jlist=np.random.randint(0,7,10)
A[ilist,jlist]=0
```



0			0		0		
						0	0
	0						
	0						
		0				0	
			0				

Creating the matrix

```
npi=int(2)
npj=int(ntotal/npi)
```



Split the work:

- *How many* elements each process needs to write

Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

```
from mpi4py import MPI
```

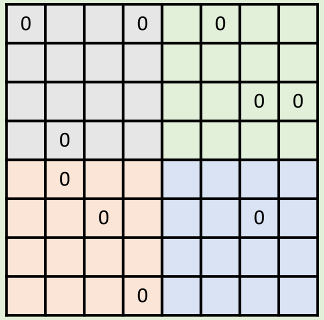
Initializing the MPI

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

Finding:

- Total *number* of processes
- *Rank* of each process

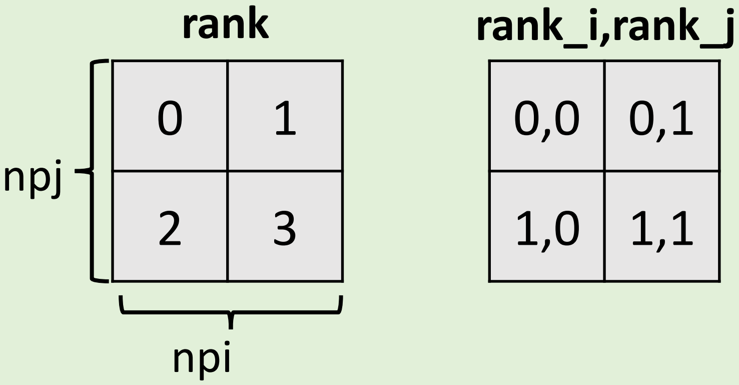
```
import numpy as np
np.random.seed(10)
A=np.random.rand(8,8)
ilist=np.random.randint(0,7,10)
jlist=np.random.randint(0,7,10)
A[ilist,jlist]=0
```



0			0		0		
						0	0
	0						
	0						
		0				0	
			0				

Creating the matrix

```
npi=int(2)
npj=int(ntotal/npi)

rank_i=rank//npi
rank_j=rank%npi
```



rank

0	1
2	3

rank_i,rank_j

0,0	0,1
1,0	1,1

Split the work:

- *How many* elements each process needs to write

Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

```
from mpi4py import MPI
```

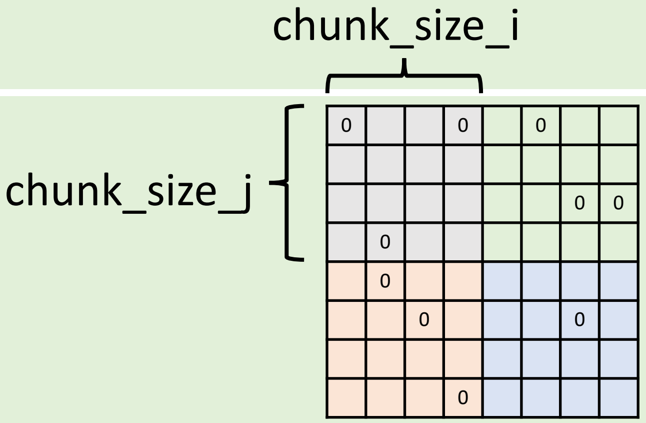
Initializing the MPI

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

Finding:

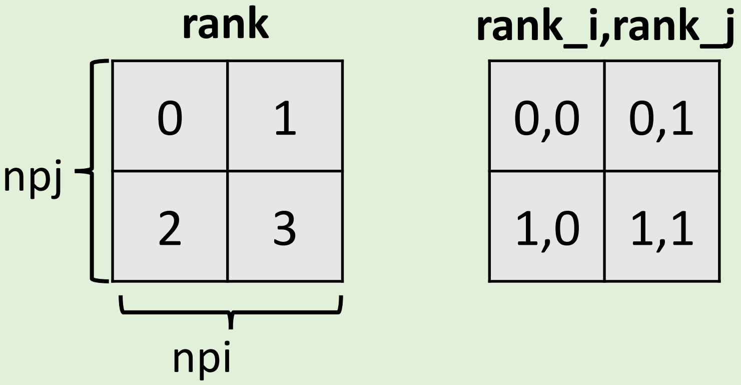
- Total *number* of processes
- *Rank* of each process

```
import numpy as np
np.random.seed(10)
A=np.random.rand(8,8)
ilist=np.random.randint(0,7,10)
jlist=np.random.randint(0,7,10)
A[ilist,jlist]=0
```



Creating the matrix

```
npi=int(2)
npj=int(ntotal/npi)

rank_i=rank//npi
rank_j=rank%npi

ni,nj=A.shape
chunk_size_i=int(ni/npi)
chunk_size_j=int(nj/npj)
```



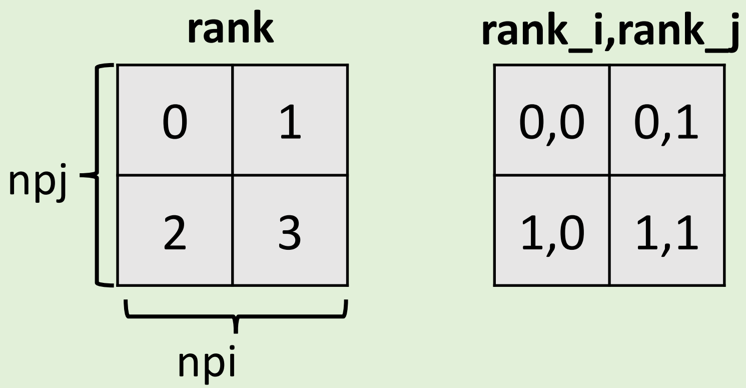
Split the work:

- *How many* elements each process needs to write

Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

```
rank_i=rank//npi
rank_j=rank%npi

ni,nj=A.shape
chunk_size_i=int(ni/npi)
chunk_size_j=int(nj/npj)

A_tmp=A[rank_i*chunk_size_i:(rank_i+1)*chunk_size_i,
      rank_j*chunk_size_j:(rank_j+1)*chunk_size_j]
```



rank	
0	1
2	3

rank_i,rank_j	
0,0	0,1
1,0	1,1

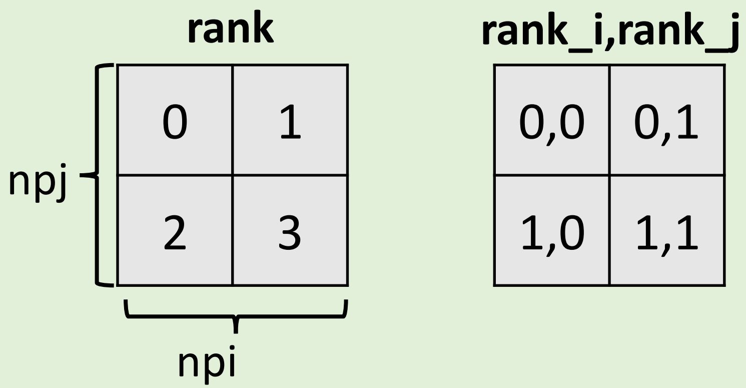
Split the work:

- *How many* elements each process needs to write

Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

```
rank_i=rank//npi
rank_j=rank%npi

ni,nj=A.shape
chunk_size_i=int(ni/npi)
chunk_size_j=int(nj/npj)

A_tmp=A[rank_i*chunk_size_i:(rank_i+1)*chunk_size_i,
       rank_j*chunk_size_j:(rank_j+1)*chunk_size_j]
```



Split the work:
- *How many* elements each process needs to write

```
nz_local =len(np.where(A_tmp==0)[0])
```

Calculation

Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

```
nz_local =len(np.where(A_tmp==0)[0])
```

Calculation

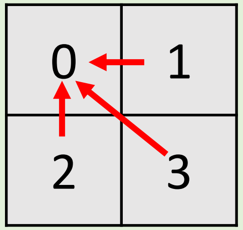
Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

```
nz_local =len(np.where(A_tmp==0)[0])
```

Calculation

```
if rank !=0:
  comm.send(nz_local,dest=0,tag=7)
```



Sending the results to rank0

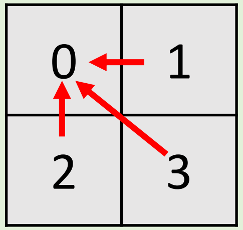
Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

```
nz_local =len(np.where(A_tmp==0)[0])
```

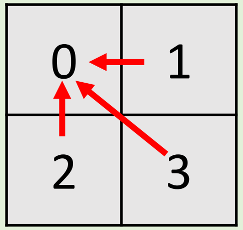
Calculation

```
if rank !=0:
  comm.send(nz_local,dest=0,tag=7)
```



Sending the results to rank0

```
if rank==0:
  nz_list=np.zeros((ntotal,),int)

  nz_list[0]=nz_local

  for i in range(1,ntotal):
   nz_list[i]=comm.recv(source=i,tag=7)
```



Receiving the results by rank0

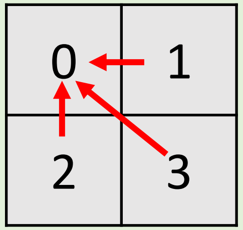
Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

```
nz_local =len(np.where(A_tmp==0)[0])
```

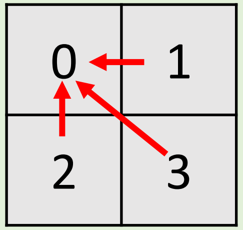
Calculation

```
if rank !=0:
  comm.send(nz_local,dest=0,tag=7)
```



Sending the results to rank0

```
if rank==0:
  nz_list=np.zeros((ntotal,),int)

  nz_list[0]=nz_local

  for i in range(1,ntotal):
    nz_list[i]=comm.recv(source=i,tag=7)
```



Receiving the results by rank0

```
if rank==0:
  nz_global=sum(nz_list)
```

Calculation

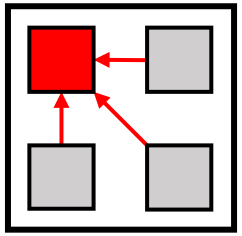
```
if rank==0:
  print(nz_global)
```

Printing the results

Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results
What if we had more processes?

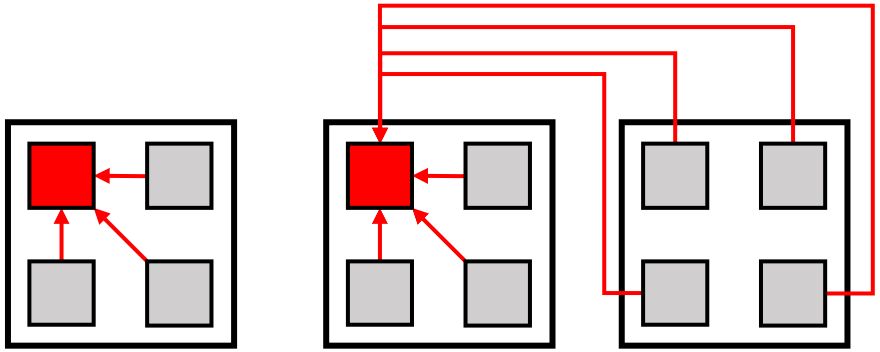
4 Processes



Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results
What if we had more processes?

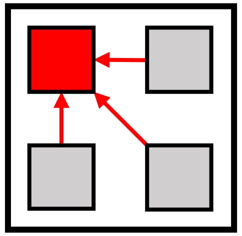
4 Processes **8 Processes**



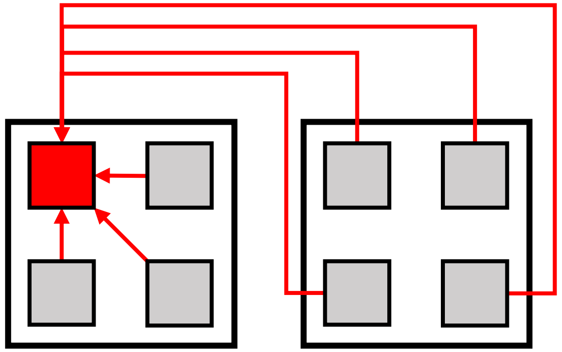
Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results
What if we had more processes?

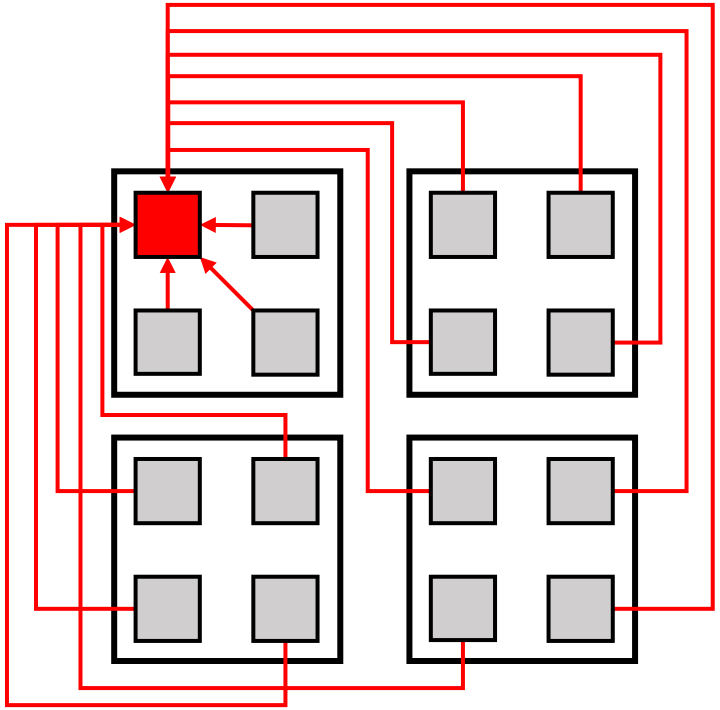
4 Processes



8 Processes



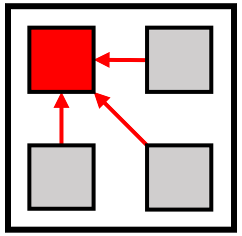
16 Processes



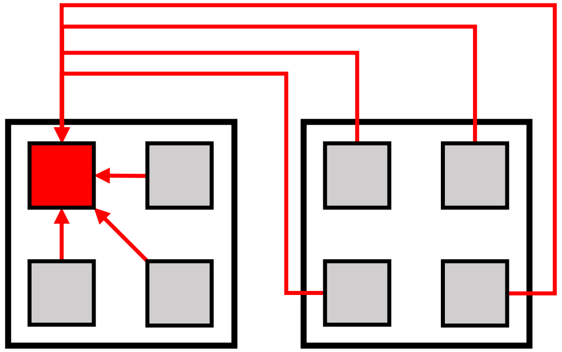
Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results
What if we had more processes?

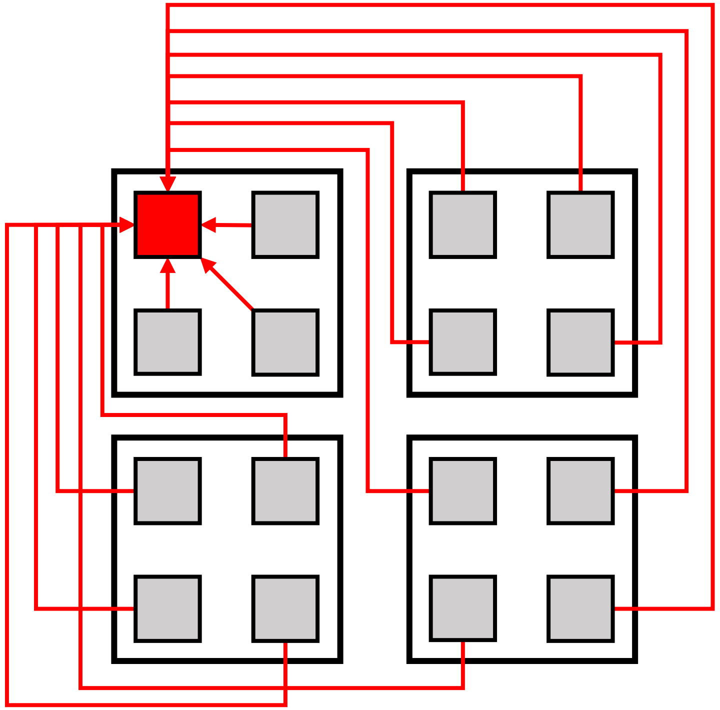
4 Processes



8 Processes

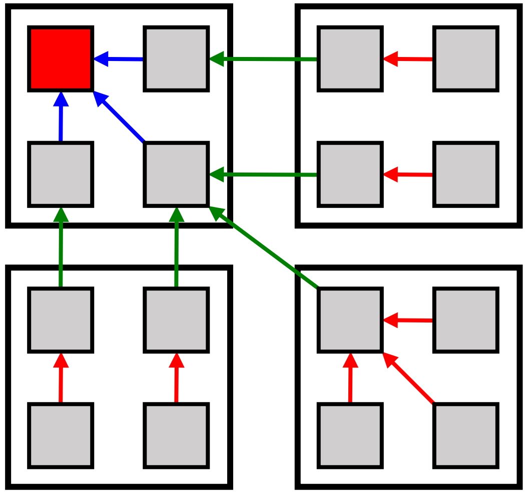


16 Processes



16 Processes:
Smarter implementation

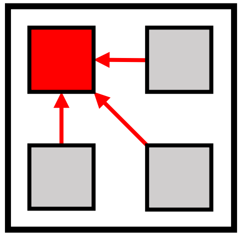
Collective Communication



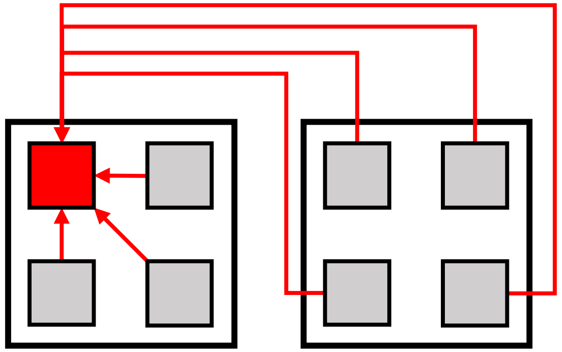
Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results
What if we had more processes?

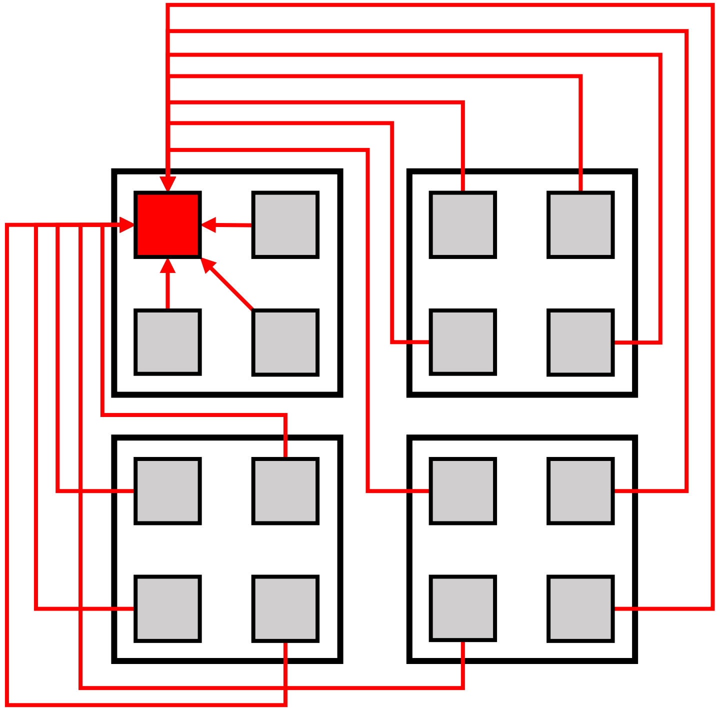
4 Processes



8 Processes

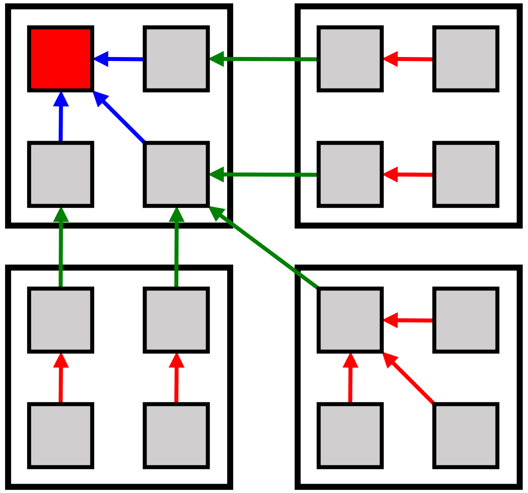


16 Processes



16 Processes:
Smarter implementation

Collective Communication



This specific case is called:
MPI_Gather

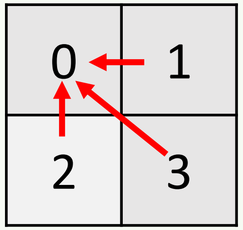
Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

```
nz_local =len(np.where(A_tmp==0)[0])
```

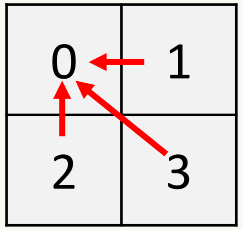
Calculation

```
if rank !=0:
  comm.send(nz_local,dest=0,tag=7)
```



Sending the results to rank0

```
if rank==0:
  nz_list=np.zeros((ntotal,),int)

  nz_list[0]=nz_ local

  for i in range(1,ntotal):
    nz_list[i]=comm.recv(source=i,tag=7)
```



Receiving the results by rank0

```
if rank==0:
  nz_global=sum(nz_list)
```

Calculation

```
if rank==0:
  print(nz_global)
```

Printing the results

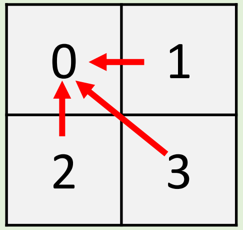
Example

Count the number of zero elements of a matrix in parallel (with 4 processes) and print the results

```
nz_local =len(np.where(A_tmp==0)[0])
```

Calculation

```
nz_list = comm.gather(nz_local, root=0)
```



Gather nz_local

```
if rank==0:
  nz_global=sum(nz_list)
```

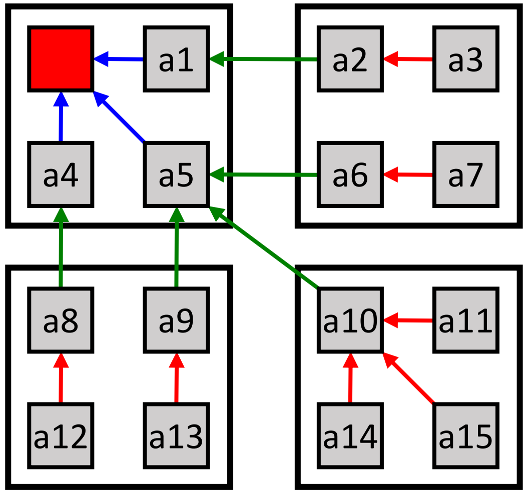
Calculation

```
if rank==0:
  print(nz_global)
```

Printing the results

Collective Communication

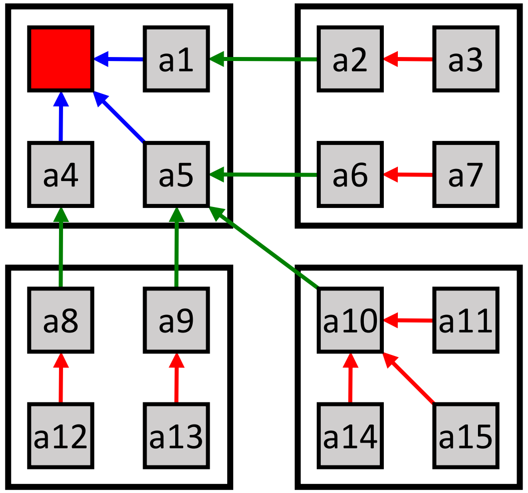
Gather



At rank_i:	value=a_i
vector = comm.gather(**value**, root=0)	
At rank_0:	vector = [a0, a1, a2, …, a15]

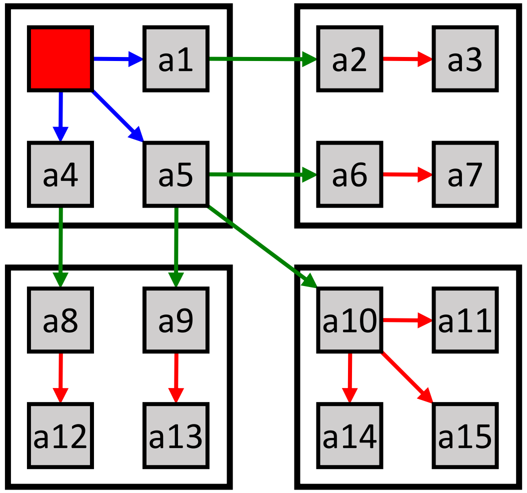
Collective Communication

Gather



At rank_i:	value=a_i
vector = comm.gather(**value**, root=0)	
At rank_0:	vector = [a0, a1, a2, …, a15]

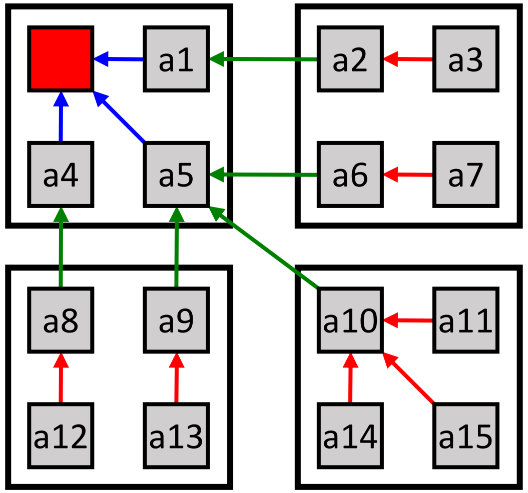
Scatter



At rank_0:	vector = [a0, a1, a2, …, a15]
value = comm.scatter(**vector**, root=0)	
At rank_0:	value=a_i

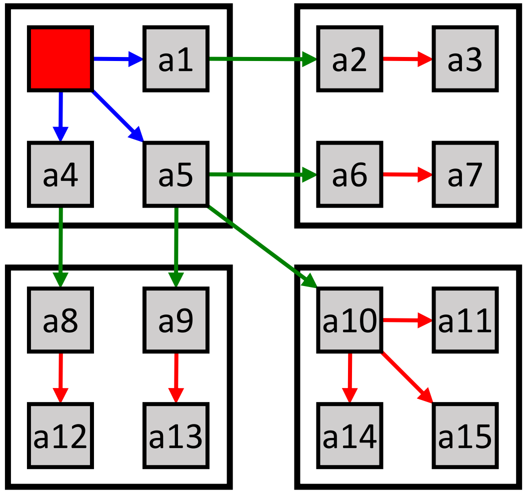
Collective Communication

Gather



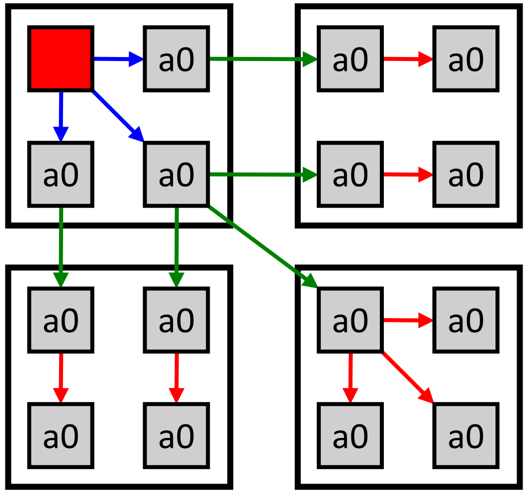
At rank_i:	value=a_i
vector = comm.gather(**value**, root=0)	
At rank_0:	vector = [a0, a1, a2, …, a15]

Scatter



At rank_0:	vector = [a0, a1, a2, …, a15]
value = comm.scatter(**vector**, root=0)	
At rank_0:	value=a_i

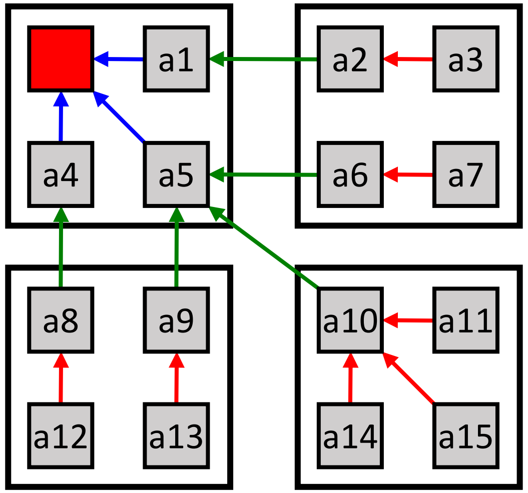
Broadcast



At rank_0:	value=a_0
value = comm.broadcast(**value**, root=0)	
At rank_i:	value=a_0

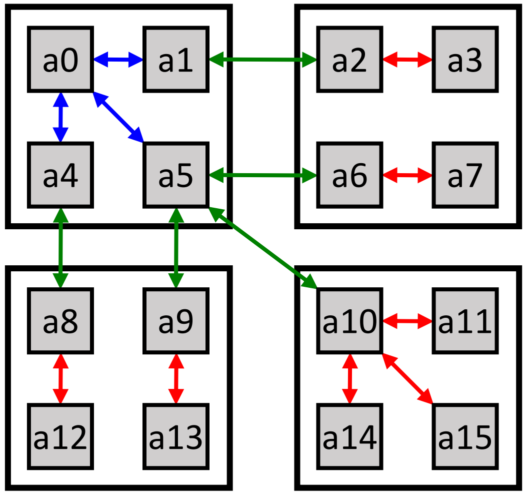
Collective Communication

Gather



At rank_i:	value=a_i
vector = comm.gather(**value**, root=0)	
At rank_0:	vector = [a0, a1, a2, …, a15]

Allgather



At rank_i:	value=a_i
vector = comm.allgather(**value**)	
At rank_i:	vector=[a0, a1, a2, …, a15]

Type of collective operations

Data Movement

- Broadcast
- Scatter/Gather
- Allgather/AlltoAll

Computation

- Reduce
- Allreduce
  - SUM
  - MIN/MAX
  - …
- Scan

Synchronization

- Barrier

Thank you!

- Questions?
- carc-support@usc.edu
- Office hours: LVL 3L and Zoom 2:30-5pm Every Tuesday
- Thanks to Marco for help on these slides!